Seamless FPGA deployment over Spark in cloud computing: A use case on machine learning hardware acceleration.

Christoforos Kachris, Ioannis Stamelos, Elias Koromilas, and Dimitrios Soudris

Institute of Computer and Communications Systems (ICCS), GR

Abstract. Emerging cloud applications like machine learning and data analytics need to process huge amount of data. Typical processor architecture cannot achieve efficient processing of the vast amount of data without consuming excessive amount of energy. Therefore, novel architectures have to be adopted in the future data centers in order to face the increased amount of data that needs to be processed. In this paper, we present a novel scheme for the seamless deployment of FPGAs in the data centers under the Spark framework. The proposed scheme, developed in the VINEYARD project, allows the efficient utilization of FPGAs without the need to change the applications. The performance evaluation is based on the KMeans ML algorithm that is widely used in clustering applications. The proposed scheme has been evaluated in a cluster of heterogeneous MPSoCs. The performance evaluation shows that the utilization of FPGAs can be used to speedup the machine learning applications and reduce significantly the energy consumption.

Keywords: hardware accelerators, data centre, heterogeneous, big data

1 Introduction

Machine learning, data analytics and Big Data are some of the emerging cloud applications responsible for the significant increases in data-center workloads during the last years. In 2015, the total network traffic of the data centres was around 4.7 Exabytes and it is estimated that by the end of 2018 it will cross the 8.5-Exabyte mark, following a cumulative annual-growth rate (CAGR) of 23% [1]. In response to this scaling in network traffic, data-centre operators have resorted to utilizing more powerful servers. Relying on Moore's law for the extra edge, CPU technologies have scaled in recent years through packing an increasing number of transistors on chip, leading to higher-performance ratings. 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 one could increase server performance without increasing their clock frequency. Unfortunately, this solution was soon found to scale poorly in the longer term, as well. 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 [2].

The failure of Dennard scaling, to which the shift to multicore chips is partially a response, has limited multicore scaling just as single-core scaling has been curtailed. This issue has been identified in the literature as the dark-silicon era in which some of the areas in a chip are kept powered down in order to comply with thermal constraints [3].

A solution that can be used to overcome this problem is the use of application-specific accelerators. Specialized multicore processors with application-specific acceleration modules can leverage the underutilized die area to overcome the initial power barrier, delivering significantly higher performance for the same power envelope [4]. The main idea is to use the abundant die area by implementing application-specific accelerators and dynamically powering up only those accelerators suitable for a given workload. This approach can be applied either at fine-grain level (using accelerators inside the chip) or at coarse-grain level (using rack-based accelerators). In the latter case, the accelerators can either be located on the same board with the server processor or in a different blade/rack. The use of highly specialized units designed for specific workloads can greatly enhance server processors and can also increase significantly the performance of data centres subject to a maximum power budget.

The VINEYARD project aims towards the development of an integrated platform for the efficient utilization of hardware accelerators in the data centers. VINEYARD aims to develop an integrated platform for energy-efficient data centers based on programmable hardware accelerators. It also developed a high-level framework for allowing end-users to seamlessly utilize these accelerators in heterogeneous computing systems by using typical data-center programming frameworks (e.g. Spark, etc.).

2 VINEYARD project

VINEYARDs goal is to develop the technology and the ecosystem that will enable the efficient integration of the hardware acceleration in the data centre applications, seamlessly. The deployment of energy-efficient hardware accelerators will be used to improve significantly the performance of cloud computing applications and reduce the energy consumption in data centres.

VINEYARD is developing an integrated framework for energy-efficient data centres based on programmable hardware accelerators. It is working towards a high-level programming framework that allows end-users to seamlessly utilize these accelerators in heterogeneous computing systems by using typical data-centre cluster frameworks (i.e. Spark). The VINEYARD framework and the required system software hides the programming complexity of the heterogeneous computing system based on hardware accelerators. This programming framework also allows, the hardware accelerators to be swapped in and out of the heterogeneous infrastructure so as to offer both efficient energy use and flexibil-

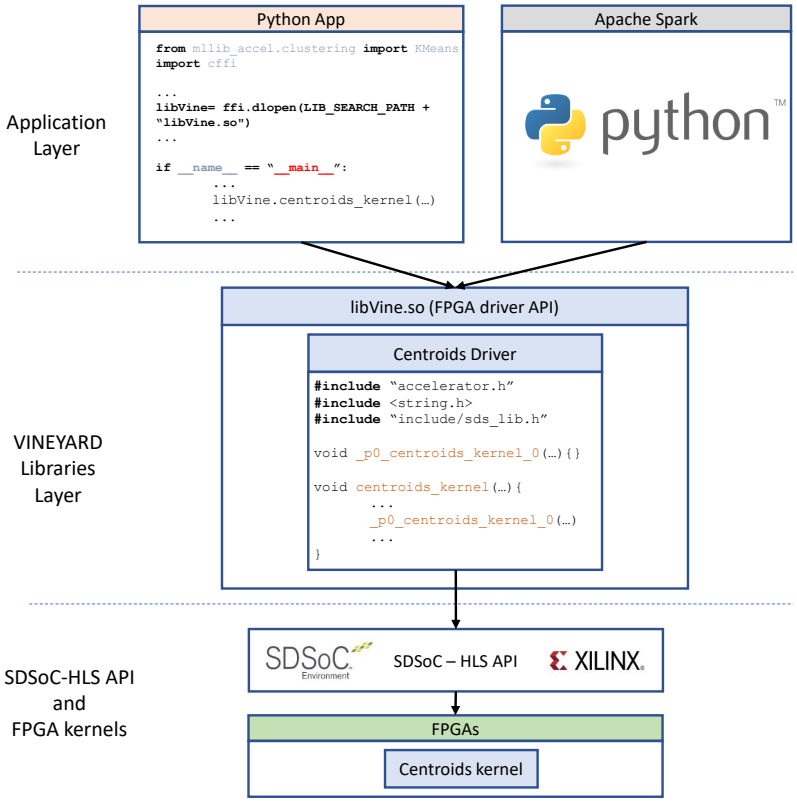

Fig. 1. High-level block diagram of the VINEYARD library for seamless integration with programming frameworks

ity. To allow the efficient utilization of the accelerators from several applications, a novel VM appliance model for provisioning of data to shared accelerators has been developed. The enhanced VINEYARD middleware augments the functionality of the resource manager, by enabling more informed allocation of tasks to accelerators.

Figure 1 depicts the high-level overview of the VINEYARD library. Applications that are targeting heterogeneous data centers using traditional servers or micro-servers are programmed using traditional data center frameworks, such as Spark, or more application specific frameworks such as the PyNN framework that is used for neural networks. In these applications, VINEYARD provides the required APIs that enable the utilization of the heterogeneous infrastructures without any other modifications in the source code.

The figure shows all the layers of the controllers developed in VINEYARD for the efficient communication of the FPGAs with the programming frameworks. The controllers that we developed support the Xilinx Zynq platforms. We created a unified software stack, tailored to our new needs, that would be able to support

expansions for supporting more platforms or new accelerators. The FPGA driver API is packed in a shared object library and can be used in a transparent way hiding all the low level details. What is more, we implemented top level APIs in Python for standalone and Apache Spark integrated use, that are easy to be used and are also easily maintained since the middle layer, our shared library remains the same for all of the above. In other words, we implemented a 3-tier style software stack. The top level hosts the users' applications, the middle layer hosts our libraries and the lower layer hosts the SDSoC-HLS API which is used to actually invoke the accelerator. This 3-tier scheme has a lot of advantages which we will go through in more detail in the next paragraphs.

Application Layer: This layer hosts users' applications. The applications can run natively using Python. Users are able to perform a plethora of methods (i.e. train(), test(), load() etc.) on their machine learning models. Users already having their machine learning applications running standalone or in an Apache Spark cluster, don't need to change a single line of code except from the imported library. Except from that, changes in the lower layers of our stack won't affect this. This way we are able to make changes, optimize and add stuff or functionality to our libraries and drivers without affecting any top-level applications.

Vineyard Layer: This layer hosts the whole functionality of our framework. The key element of this layer is the implemented shared library (libVine.so). It hosts the FPGA drivers for each application, written in C/C++ and is used to communicate with the SDSoC - HLS API. Each kernel driver (e.g centroids_driver) invokes the corresponding FPGA kernel to perform the requested tasks.

SDSoC-HLS API and FPGA layer: The bottom layer, that serves as the FPGA runtime, is basically consisted of the SDSoC-HLS library that is provided from Xilinx along with the FPGA itself hosting any implemented kernels.

3 Seamless deployment of FPGA under Spark: A use-case on KMeans clustering

In this section we present a use-case for the evaluation of the VINEYARD framework under the Spark framework. Apache Spark [5] 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 [6]. It was developed in

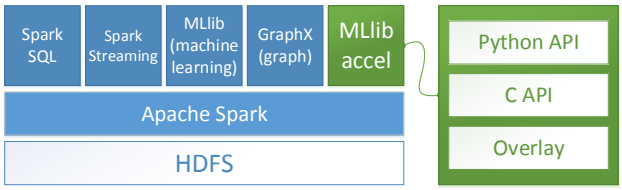

Fig. 2. VINEYARD's library for the deployment of FPGAs in the Spark library

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.

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

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

In this project, we have developed the required APIs for python and C that allows the direct invoking of the hardware accelerators from the python level used in Spark. The 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. Therefore, the only modification that is required is the extension of the Python library with the new function calls for the communication with the hardware accelerator.

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.

3.1 Python API for Spark

Most machine learning techniques have a common characteristic that makes them ideal as means to explore the performance benefits of our heterogeneous cluster. They are iterative algorithms that make multiple passes over the data set, while also allow the computations in each iteration to be performed in parallel on different data chunks.

In KMeans, for example, the computation of the partial sums and counts for each new cluster is performed on the available Workers, and then the Master aggregates the results and calculates the new centroids.

Taking into account and understanding the structure of Sparks MLlib, we developed new libraries for KMeans clustering, that take advantage of the accelerator that is available in the workers. As a result, when a Spark user wants to utilize the hardware accelerator in an existing application, the main change that needs to be made, is the replacement of Sparks mllib library, that is imported, with our mllib_accel one. Therefore, a user can speedup the execution time of a Spark application by simply replacing the library package.

The first approach was to simply replace the mapper functions (*centroids_kernel*) with Python APIs that drive the hardware accelerators. Inside these new functions we used to download the equivalent overlay, create the necessary DMA objects, store the data inside the corresponding buffers, perform the DMA transfers and finally destroy them, free the allocated memory and return the results. After profiling the applications though, we concluded that most of the time (99%) is wasted on writing the train RDD data to the allocated DMAs buffers.

However, since the data remain the same (cached) over the whole execution of the training, we have managed and implemented a novel scheme that allows the persistent storing of the RDD in contiguous memory, avoiding in-memory transfers every time the accelerator is invoked. For this reason, we developed a new mapper function that allocates and fills contiguous memory buffers with the training data, in order to remain there for the rest of the application execution. So when the DMA objects are created in each iteration, there is no need to create new buffers for them and fill them with the corresponding data, they just get assigned the previously created ones. Also, before destructing these DMA objects, their assigned buffers are set to 'None', so that they remain intact and are not freed as it is shown in Figure 3.

Based on the above, we have created Python APIs which basically consist of three calls:

- **cma** (contiguous memory allocate): This call is used for the creation of the buffers and the further allocation of contiguous memory. Also at this point the overlay is downloaded and the training data is written to the corresponding buffers. Using cma, a new RDD, which contains only information about these buffers (memory addresses, sizes, etc.), is created and persisted.
- **kernel_accel** (centroids): In this call, the DMA objects are created using Xilinxs built-in modules and classes; previously allocated buffers are assigned to DMAs, current weights/centers are written in memory and finally data are transferred to the programmable logic. Counts and sums are computed in return, buffers are dis-assigned from DMAs and the last ones are destructed.
- **cmf** (contiguous memory free): This call is explicitly used to free all previously allocated buffers.

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

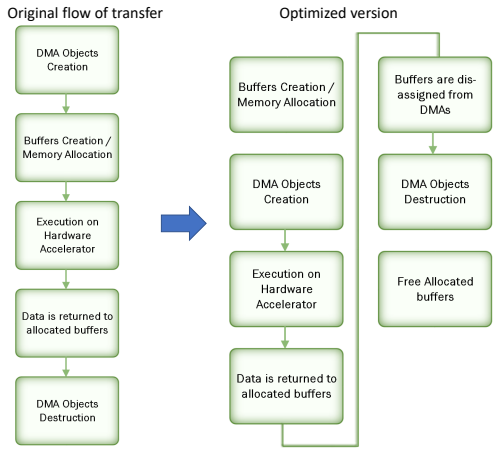

Fig. 3. Flow of the original and optimized method for the DMA transfers to the accelerator

4 Use-case on Machine Learning under Spark

To evaluate the proposed framework, we have developed a hardware accelerator for KMeans clustering and more specifically for the computation of the centroids. The hardware accelerator has been implemented using the Xilinx Vivado High-Level Synthesis (HLS) tool. The algorithm have been written in C and has been annotated with HLS *pragmas* for the efficient mapping in reconfigurable logic.

4.1 Algorithmic approach of KMeans

KMeans is one of the simplest unsupervised learning algorithms that solve the well known clustering problem and is applicable in a variety of disciplines, such as computer vision, biology, and economics. It attempts to group individuals in a population together by similarity, but not driven by a specific purpose.

The procedure follows a simple and easy way to cluster the training data points into a predefined number of clusters (K). The main idea is to define K centroids c, one for each cluster.

Given a set of $numExamples$ (n) observations $\{x^0, x^1, , x^{n-1}\}$, where each observation is an m-dimensional real vector, KMeans clustering aims to partition the n observations into K ($\leq n$) sets $\{s^0, s^1, , s^{K-1}\}$ so as to minimize total intra-cluster variance, or, the squared error function:

$$J = \sum_{k=1}^{K} \sum_{x \in s^k} ||x - c^k||^2$$

The KMeans clustering algorithm is as follows:

1 : **procedure** $train(x)$
2 : initialize c with K random data points
3 : **while** not converged:
4 : $centroids\_kernel(x, c)$
5 : **for** every $k = 0, ..., K-1$:
6 : $c^k = \frac{1}{|s^k|} \sum_{x \in s^k} x$

7 : **procedure** $centroids\_kernel(x, c)$
8 : **for** every $k = 0, ..., K-1$:
9 : $s^k = \left\{ x : \left\| x - c^k \right\|^2 \leq \left\| x - c^{k'} \right\|^2 \ \forall k', 0 \leq k' \leq K-1 \right\}$

The algorithm as described, starts with a random set of K centroids (c). During each update step, all observations x are assigned to their nearest centroid, while afterwards, these center points are repositioned by calculating the mean of the assigned observations to the respective centroids.

5 Performance evaluation

As a case study, we built a KMeans clustering model with 784 features and 14 centers, using 40k available training samples, for a handwritten digits recognition problem. The data are provided by Mixed National Institute of Standards and Technology (MNIST) database [7]. To evaluate the performance of the system and to perform a fair comparison we built a cluster of four nodes based on the Zynq platform and we compared it with four Spark worker nodes using the Intel Xeon cores [8]. Table 1 shows the features of each platform.

It is important to note that a single Spark executor JVM process requires most of the available 512 MB RAM on PYNQ-Z1s, placing a restriction on the Spark application, which requires main memory to cache and repeatedly access the working dataset from FPGAs off-chip RAM once read from HDFS. This results in delays during the execution as inevitably are performed transfers between the memory and the swap file, which is stored inside the SD card. This memory restriction is also the reason why we limit the number of Spark executors to 1 ARM core per node, thus preventing both cores from performing Spark tasks simultaneously.

On the other hand, the Xeon system consists of 12 cores with 2 threads each core. The Spark cluster started on this platform allocates 4 out of 24 threads, as worker instances, in order to compare it with the 4 nodes of the Pynq cluster. We also compared the accelerated platform with the software only scenario in which the algorithm is executed only on the ARM cores. Such comparison is valuable as there are applications where only embedded processors can be used and big-core systems like Xeon cannot be supported due to power constraints.

5.1 Latency and Execution time

Figure 4 depicts the execution time of the KMeans clustering application running on a high-performance x86_64 Intel processor (Xeon E5 2658) clocked at 2.2 GHz

Table 1. Main features of the evaluated processors.

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

Fig. 4. KMeans speedup versus the number of the iterations (Intel XEON vs Pynq).

and a Pynq cluster which makes use of the Programmable Logic, for an input dataset of 40000 lines splitted in chunks of 5000 lines, for various numbers of iterations. In the PYNQ-Z1 boards the data extraction part, for the KMeans clustering, takes about 80 sec to complete, while every iteration of the algorithm is completed in 0.54 sec, since the train input data is already cached into the previously allocated buffers. On the other hand, Xeon CPU reads, transforms and caches the data in only 7.5 sec, but every iteration takes approximately 3.3 sec. This is the reason why the speedup actually depends on the number of iterations that are performed. For this specific example the LR model converges, and achieves up to 91.5% accuracy, after 100 iterations of the algorithm, in which up to 2x system speedup is achieved compared to the Xeon processor. However, there are cases in which much higher number of iterations is required, until the convergence criteria is met, and thus much higher speedup can be observed.

X

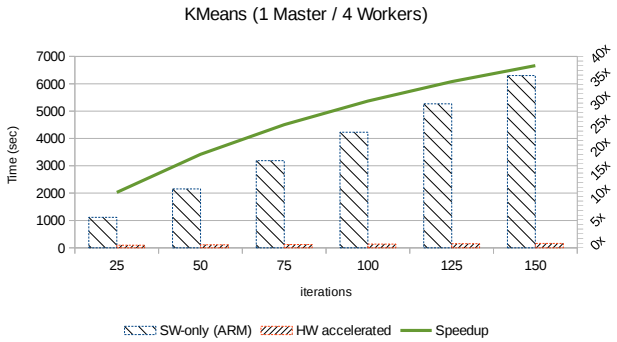

Fig. 5. KMeans speedup versus the number of the iterations (ARM vs Pynq).

Table 2 shows the execution time of the main mapper functions which are executed on the worker nodes. In the Xeon platform and the ARM-only case, both the data extraction and the algorithm computations are performed on the CPUs, while in the Pynq workers the data extraction is executed on the ARM core while the algorithmic part is offloaded to the programmable logic. Figure 5 show the speedup of the accelerated execution compared to the software only solution running on the same cluster but using only the ARM processors. In this case, we can achieve up to 31x for KMeans, compared to the software only case, which shows that it is definitely crucial to provide accelerator support for future embedded datacenters.

Worker Type	**Data Extraction**	**KMeans Algorithm Computations (per iteration)**
Intel XEON	7.5	3.3
ARM	80	41.5
Pynq	80 (ARM)	0.54 (FPGA)

Table 2. Execution time (sec) of the worker (mapper) functions.

5.2 Power and energy consumption

To evaluate the energy savings we measured the average power running the algorithm both in the SW-only, and the HW accelerated cases. In order to measure the power consumption of the Xeon server, we used Intels Processor Counter Monitor (PCM) API, which, among others, enables capturing the power consumed by the CPU and DRAM memory for executing an application. We also measured the power consumption in the accelerated case using the ZC702 Evaluation board, which hosts the same Zynq device as the PYNQ-Z1 board, taking advantage of the on-board power controllers.

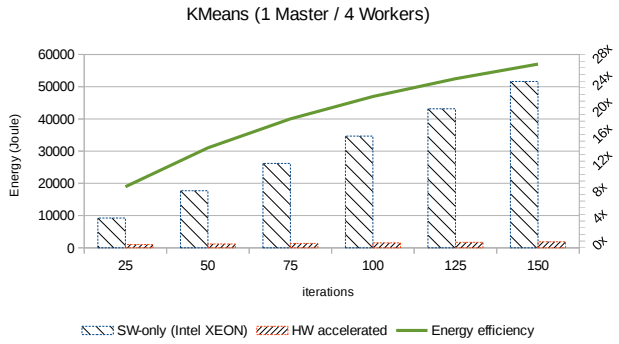

Fig. 6. KMeans energy consumption based on the number of iterations (Intel XEON vs Pynq).

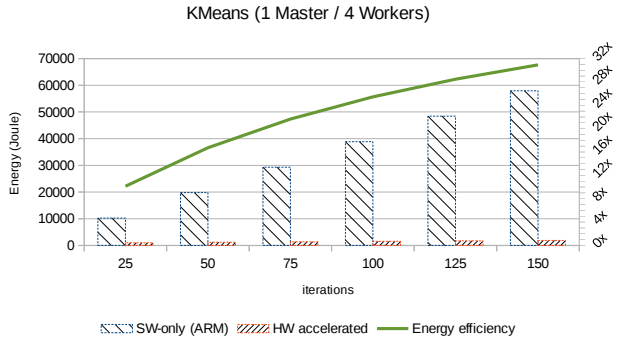

Fig. 7. KMeans energy consumption based on the number of iterations (ARM vs Pynq).

Figure 6 show the energy consumption of the Xeon processor compared to the Pynq cluster. The average power consumption of the Xeon processor and the DRAMs is 100 Watt, while a single Pynq node (both the AP SoC and the DRAM) consumes about 2.6 Watt during the data extraction and 3.2 Watt during the hardware computations. In that case, we can achieve up to 23x better energy efficiency due to the lower power consumption and the lower execution time.

In Figure 7 is depicted the energy consumption comparison between the SW-only (ARM) and HW-accelerated execution of the application on the Pynq cluster. It is clear that the average power consumption of the accelerated case is slightly higher than the power consumption of the ARM-only one, because of the need to power supply also the programmable logic. However, due to the significant much higher execution time of the ARM-only solution, eventually, up to 29x lower energy consumption is achieved.

6 Conclusions

The main goal of the VINEYARD project is to develop a new framework for the efficient integration of accelerators into commercial data centres. The VINEYARD project will not only develop novel accelerator-based servers but will also develop all the required systems (hypervisor, middleware, APIs and libraries) that will allow the users to seamlessly utilize the accelerators as an additional cloud resource. The efficient utilization of accelerators in data centres will significantly improve the overall performance of cloud-based applications and will also reduce the energy consumption in the data centres. Finally, VINEYARD aspires to foster the innovation of soft-IP accelerators in the domain of cloud computing by the promotion of a central repository for the hosting of the relevant accelerators.

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. In *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update 20142019 White Paper*.
2. Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, and Uri C Weiser. Many-core vs. many-thread machines: Stay away from the valley. *IEEE Computer Architecture Letters*, 8(1):25–28, 2009.
3. Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.
4. Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, July 2011.
5. Apache, spark, http://spark.apache.org/.
6. 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.
7. Mazdak Fatahi. Mnist handwritten digits, 2014.
8. C. Kachris, E. Koromilas, I. Stamelos, and D. Soudris. Fpga acceleration of spark applications in a pynq cluster. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–1, Sept 2017.