Noname manuscript No.
(will be inserted by the editor)

An FPGA-based Integrated MapReduce Accelerator platform

Christoforos Kachris · Dionysios Diamantopoulos · Georgios Sirakoulis · Dimitrios Soudris

Received: date / Accepted: date

Abstract MapReduce is a programming framework for distributed systems that is used to automatically parallelize and schedule the tasks to distributed resources. MapReduce is widely used in data centers to process enterprise databases and Big Data. This paper presents a novel MapReduce accelerator platform based on FPGAs that can be used to speedup the processing of the MapReduce data. The proposed platform consists of specialized hardware accelerators for the *Map* tasks and a shared configurable accelerator for the *Reduce* tasks. The hardware accelerators for the Map tasks are developed using a modified source-to-source High-level Synthesis (HLS) tool while the Reduce accelerator is based on a novel hashing scheme. The proposed scheme is implemented, mapped and evaluated to a Virtex 7 FGPA. The performance evaluation is based on a benchmark suite that represent typical MapReduce applications and it shows that the proposed scheme can achieve up to 2 or-

This paper was partially supported by Horizon 2020 EU-funded project #644906, AEGLE, http://www.aegle-uhealth.eu/

Christoforos Kachris
Athens Information Technology
Athens, Greece Tel.: +30-210-6682792
E-mail: kachris@ait.edu.gr

Dionysios Diamantopoulos
School of Electrical and Computer Engineering
National Technical University of Athens, Athens, Greece
E-mail: diamantd@microlab.ntua.gr

Georgios Sirakoulis
Department of Electrical and Computer Engineering
Democritus University of Thrace, Xanthi, Greece
E-mail: gsirak@ee.duth.gr

Dimitrios Soudris
School of Electrical and Computer Engineering
National Technical University of Athens, Athens, Greece
E-mail: dsoudris@microlab.ntua.gr

ders of magnitude energy reduction compared to General Purpose Processors (GPPs).

Keywords MapReduce · accelerator · data center · FPGAs · reconfigurable computing

1 Introduction

Emerging web applications like streaming video, social networks, IoT, Bib Data and cloud computing has created the need for warehouse scale data centers hosting thousands of servers [5]. One of the main programming frameworks for processing large data sets in the data centers and other clusters of computers is the MapReduce framework [4]. MapReduce is a programming model for processing large data sets using high number of nodes. The user specifies the *Map* and the *Reduce* functions and the MapReduce scheduler performs the distribution of the tasks to the processors. One of the main advantages of the MapReduce framework is that it can be hosted in heterogeneous clusters consisting of different types of processors. The majority of the data centers are based on high performance General Purpose Processors (GPPs) such as Intel Xeon, AMD Opteron and/or IBM Power processors. However, the main drawback of these processors is that they consume high amount of power.

The power consumption in the data centers is one of the most challenging constraint that need to be addressed in order data center operators to be able to sustain the increasing network traffic. According to Greenpeace's Make IT Green report [3], it is estimated that the global demand for electricity from data centers was around 330bn kWh in 2007 (almost the same amount of electricity consumed by UK). This demand in power consumption is projected to more than triple by 2020 (more than 1000bn kWh). A high portion of this power is consumed by the servers (servers consume around 40% of the total IT power [2]).

The power consumption of the data centers has also a major impact on the environment. In 2007, data centers accounted for 14% of the total ICT greenhouse gases (GHG) emissions (or 2% of the global GHG), and it is expected to grow up to 18% by 2020 [1]. The global data center footprint in greenhouse gases emissions was 116 Metric Tonne Carbon Dioxide ($MtCO_2e$) in 2007 and this is expected to more than double by 2020 to 257 $MtCO_2e$, making it the fastest-growing contributor to the ICT sector carbon footprint.

However, when the MapReduce framework is mapped to typical high performance processors or low-power embedded processors, many resources are consumed for the mapping of the tasks to the cores and the reduction functions of the MapReduce framework.

In this paper an integrated platform for the efficient acceleration of applications based on MapReduce is performed. The proposed platform allows the efficient mapping of MapReduce applications in FPGAs that allows the speedup of the applications and the significant reduction of the power consumption compared with typical server processors. The proposed platform is

used to accelerate both the *Map* and the *Reduce* tasks. In the latter case, a configurable *Reduce* co-processor is developed that can be used to offload the processor from the Reduce tasks. The co-processor that has been developed can be configured to meet the different processing requirements depending on the application requirements. In the case of the *Map* tasks, an integrated framework is proposed that allows the efficient development of specialized *Map* accelerators using an extended High Level Synthesis toolflow. Therefore, these accelerators can be easily customized based on the applications requirements.

To evaluate the proposed architecture, the hardware accelerator has been implemented in a Virtex 7 FPGA. The proposed MapReduce accelerator can be hosted in an FPGA located inside a typical rack in data centers. The performance evaluation using a benchmark of typical MapReduce applications shows that the proposed platform can achieve up to 2 orders of magnitude lower energy consumption compared to typical processors.

Overall the main contributions of the paper are the followings:

- An integrated platform based on FPGAs for the efficient acceleration of MapReduce applications in data centers
- A configurable Reduce co-processor that can be used to speedup the processing of the Reduce tasks
- An integrated toolflow that allows the automatic development of specialized Map accelerators by extending with several directives a commercial HLS tool
- Efficient mapping and implementation of the proposed architecture to a Virtex 7 FPGA board
- Performance evaluation using the Phoenix MapReduce framework showing up to 2 orders of magnitude lower energy consumption.

The paper is organized in the following way: Section II presents the related work on the mapping of cloud applications in FPGAs. Section III presents the MapReduce framework and the tasks that are off-loaded into the hardware acceleration unit. Section IV presents the HLS MapReduce flow for the creation of the Map accelerators for specific Map tasks. Section V presents the configurable Reduce accelerator that can be tuned based on the application requirements. Section VI presents the performance evaluation in terms of execution time, area and power consumption. Finally the conclusions of this work are drawn in Section VII.

2 Related work

In [10], a reconfigurable MapReduce framework is presented but the proposed scheme is implemented as a custom design that is used to implement only the RankBoost application entirely on an FPGA. Both of the *Map* and *Reduce* tasks for the specific application have been mapped to configurable logic and thus for any new application a new design has to be implemented.

In [12] a MapReduce Framework on FPGA accelerated commodity hardware is presented where a cluster of worker nodes is designed for the MapReduce framework, and each worker node consists of commodity hardware and special hardware. However, in this case specialized accelerators have been developed for specific MapReduce applications that does not allow easy modifications of the applications.

Microsoft has also recently presented a reconfigurable fabric for accelerating large-scale data center services in [7]. The FPGA ared placed into each server accessible through PCIe, and wired directly to other FPGAs with pairs of 10 Gb SAS cables. However, the specific architecture is only used to accelerate the web search engine applications and cannot be easily extended to other applications.

The main advantage of our scheme is that we have developed the required toolflow based on an extension of the HLS tool to support the MapReduce structures in order to facilitate the efficient development of MapReduce applications directly from the source files. Therefore, using the proposed scheme we can achieve both the high performance of the hardware and the flexibility of the typical software development in data centers.

3 The Phoenix MapReduce framework

One of the most widely used frameworks that are hosted in the data centers is the MapReduce framework. MapReduce is a programming framework for processing and generating large data sets [4]. Users specify a **Map** function that processes a *key/value* pair to generate a set of intermediate *key/value* pairs, and a **Reduce** function that merges all intermediate values associated with the same intermediate key. Finally, the last stage merge together all the *key/value* pairs (Figure 1).

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system. In fact, many of the cloud computing applications are based on the MapReduce framework [4].

The MapReduce framework has been also implemented as a programming framework for multi-core architectures by Stanford University (called Phoenix MapReduce) [8]. Phoenix MapReduce framework uses threads to spawn parallel *Map* or *Reduce* tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules tasks dynamically across the available processors in order to achieve load balancing and maximizing task throughput. Locality is managed by adjusting the granularity and assignment of parallel tasks. Google's MapReduce implementation facilitates processing of Terabytes on clusters with thousands of nodes.

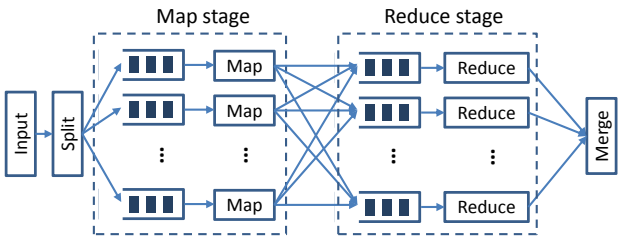

Fig. 1 The MapReduce programming framework

The Phoenix MapReduce implementation is based on the same principles but targets shared-memory systems such as multi-core chips and symmetric multiprocessors. The main advantage of the Phoenix MapReduce framework is that it can provide a simple, functional expression of the algorithm and leaving parallelization and scheduling to the run-time system and thus making parallel programming much easier.

4 HLS MapReduce tooflow architecture

In this paper we propose an integrated platform for the efficient acceleration of the MapReduce application in data centers by utilizing FPGAs in order to reduce the power consumption and increase the performance of the applications. In most of the MapReduce applications the *Map* functions are different for each application while the *Reduce* functions in most of the cases are common (i.e. merging of key/value pairs).

The proposed scheme addresses both of these tasks. A novel toolflow has been developed that is used to generate accelerators for the Map tasks by utilizing High Level Synthesis tools (HLS). In that case, a commercial HLS tool has been extended to support data type structures that are used in the Map tasks. Using the HLS toolflow, the proposed platform provide both flexibility (i.e. the accelerators can be easily modified to support other functions) and high performance. In the case of the Reduce tasks, a special Reduce coprocessor has been developed that can be configured to merge the key/value pairs based on the application requirements.

For the Map tasks, we propose the complete decoupling of MapReduce's tasks data-paths to distinct buses, accessed from individual processing engines, eliminating the necessity of the supervisor on-board CPU, i.e. the processor-centric SoC. Such an approach implies a holistic C/C++ to RTL domain-level MapReduce transition. In this work, we employ HLS tools as a state-of-art system-level implementation toolflow, in order to examine the performance exploitation options, yet constrained by the HLS limitations of such a complex framework.

Using the extended HLS toolflow, we create customized Map accelerators that exploit high data locality and thus eliminate the need of large shared-memory architectures or distributed systems. Instead of brute-force arbitrary splitting the input data to multiple subsets for further scheduling to CPUs,

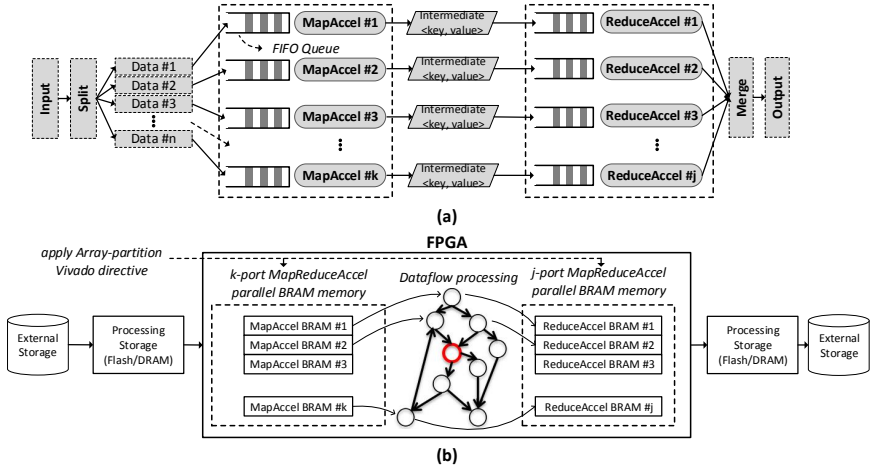

Fig. 2 HLSMapReduceFlow dataflow architecture: Every dataflow computation node is working in its unique memory. The system memory is partitioned to *k-port* and *j-port* banks for the k-map and j-reduce tasks respectively.

we select to split the input according to the application's data processing flow, in a way that optimized chunks of data are processed independently by distinct accelerators. Using this approach we manage to increase the system's throughput by a) increasing data locality, b) decreasing inter-connection latency among computation nodes and c) increasing computation parallelism by exploiting dataflow processing.

Specifically, we investigate the optimal point of dataflow processing for every application, i.e. splitting and scheduling is based on control-flow-graph (CFG), data-flow-graph (DFG) and variable liveness analysis (LA). With these information we built the corresponding optimal Map accelerator engines. Figure 2(a) shows the basic HLSMapReduceFlow architecture. While, this looks similar to the original Phoenix architecture, we highlight in Figure 2(b) the novel architecture modifications of our approach. Firstly, the on-board available block RAM (BRAM) of the FPGA is organized in distinct memory banks. Every bank has its own unique address and data bus, while it is accessed by only one computation node. This scheme allows for full parallel simultaneous operation of the computation nodes in FPGA.

The critical step of this procedure relies on the efficient mapping of application's computation paths which have the potential of parallelism. For this step we employ the Vivado HLS tool. Apart from typical high level synthesis steps, i.e. resource binding, scheduling etc. Vivado HLS provides a high number of architecture exploration options through the source code annotation with special pre-processor directives. In this work we force the exploration with the *DATAFLOW, INLINE* and *ARRAY PARTITION* directives.

Firstly we employ the partition, map and reshape directives in order to re-configure arrays on the interface they are accessed. Arrays are partitioned

into multiple smaller arrays, each implemented with its own interface. This includes the ability to partition the array into fine grain elements. On the function interface, this results in a unique port for every element in the array. This provides maximum parallel access, but creates many more ports and may introduce routing issues in the hierarchy above. By partitioning the arrays, on which input data of every map task are stored, we reduce the possibility of simultaneous access of the same data, given the inherent locality of the application, which may exploit parallelism. Locality is managed by adjusting the granularity and assignment of parallel tasks.

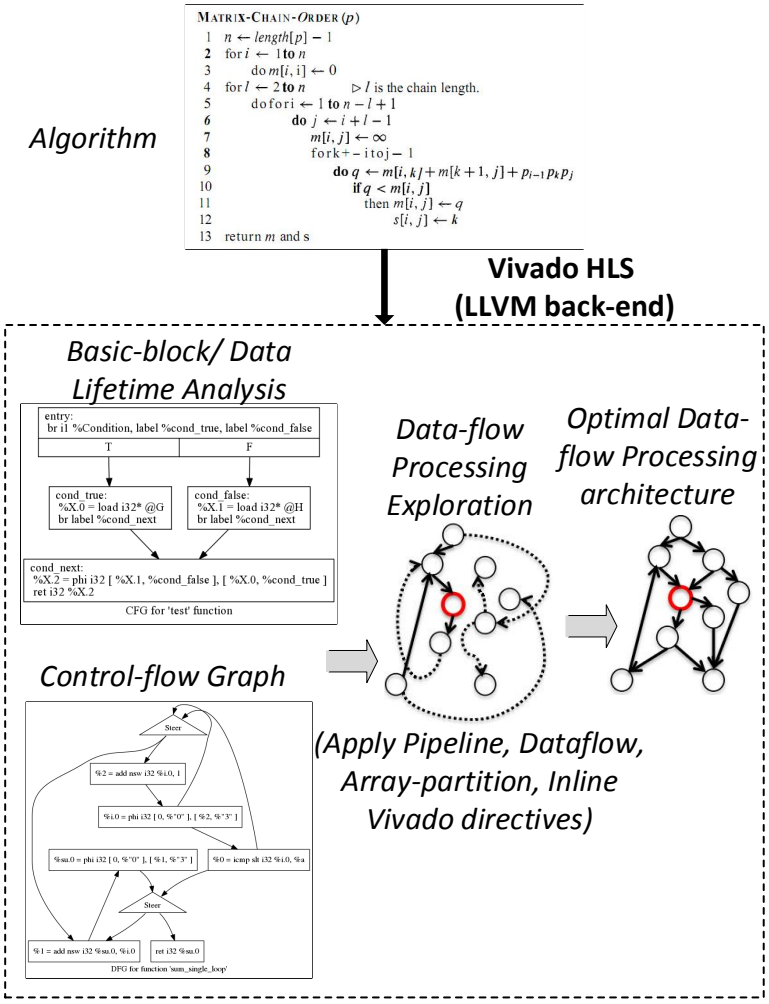

Fig. 3 Forcing dataflow exploration from control-flow algorithm description with Vivado HLS

After having partitioned the input memory, we force the micro-architecture exploration within Vivado HLS, following a dataflow computation model. From the definition back in 80's [11], we consider dataflow machines to be all programmable computers of which the hardware is optimized for fine-grain data-driven. Fine grain means that the processes that run in parallel are approximately of the size of a conventional machine code instruction. We deploy a fully spatial architecture for every map task by applying recursive inline option of Vivado HLS, i.e. *#pragma AP inline recursive*. Although this approach leads to increased resources utilization, it allows for parallel instances of shared sub-functions and removed hierarchy of sub-functions, which leads to logic optimization across function boundaries and improved latency/interval by the reduction of function call overhead.

After the above optimizations, we have already forced the creation of fine-grain fully-parallel map tasks which does not share neither data nor computation elements among them. The last optimization of the proposed scheme deals with the controlling of the the way the input data are fed to these tasks. We force a dataflow approach. Figure 3 shows the basic idea behind this approach. The input code is decomposed by Vivado's back-end LLVM compiler to basic blocks, i.e. single-entry single-exit section of code, connected through a control-flow network, i.e. control-flow-graph (CFG). Having already applied above optimizations, we further force the dataflow optimization, i.e. *#pragma AP dataflow* which takes a series of sequential tasks (functions and or loops) and creates a parallel process architecture from it. Dataflow optimization in Vivado HLS is a very powerful method for improving design throughput.

4.1 HLS MapReduce Flow Methodology for Vivado-HLS

Figure 4 shows an overview of the proposed HLSMapReduceFlow design and verification flow. The flow is based on Xilinx Vivado-HLS, a state-of-art and industrial strength HLS tool. The HLSMapReduceFlow extension is applied explicitly to the high-level source code of the application, thus it keeps minimum implementation overhead to the designers. A source-to-source code modification stage is the step where the original code is transformed to synthesizable one. These transformations cover limitations regarding the lack of dynamic memory management support, pointer arithmetic, complete ANCI C functions etc, in Vivado HLS. Moreover this step includes the process of architecture optimization directives insertion. Currently, this step is performed manually. An automated flow is considered a highly useful utility for wide and transparent adoption in data centers deployment. The transformed code is augmented by the HLSMapReduceFlow function calls, i.e. *Emit_Intermediate_accelerator(key,value)* and it is synthesized into RTL implementation through the back-end of Vivado HLS tool.

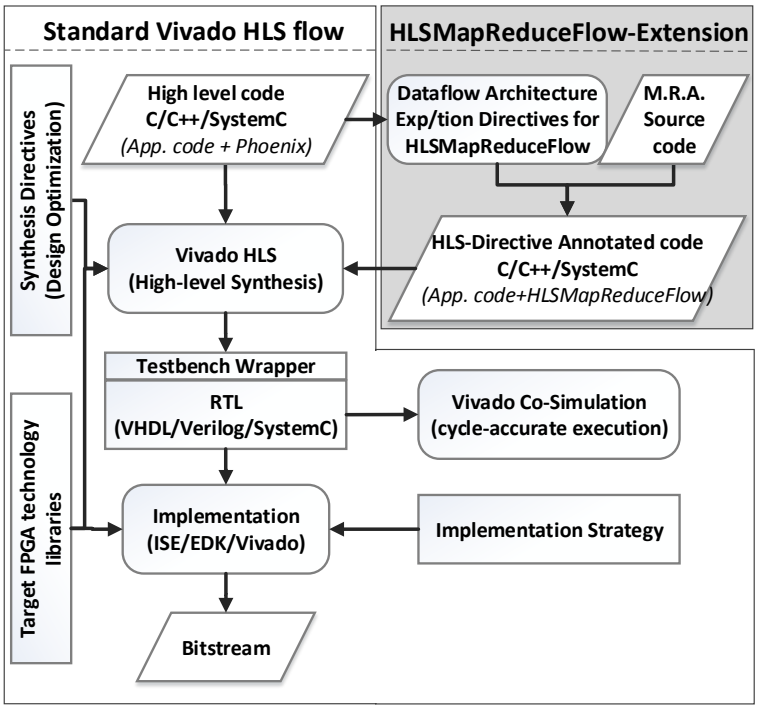

Fig. 4 Proposed extension on Vivado HLS flow to support MapReduce framework for FPGA-based systems.

Table 1 Applications Characterization

Domain	Kernel	Description	Parameters	Bytes/Iter.
Image Processing	Histogram	Determine frequency of image RGB channels.	M_{size} = 640×480	307,200
Scientific Computing	Matrix Mul.	Dense integer matrix multiplication.	M_{size} = 100×100	40,000
Enterprise Computing	String Match	Search file with keys for 4 encrypted words.	N_{keys} = 307,200	307,200
Enterprise Computing	Word Count	Counts occurrence frequency of words in file.	N_{words} = 50,000	90,094
Artificial Intelligence	Linear Regr.	Compute the best fit line for a set of points.	N_{points} = 100,000	400,000
Artificial Intelligence	PCA	Principal components analysis on a matrix.	M_{size} = 250×250	250,000
Artificial Intelligence	K_{means}	Clustering 3-D data points into 10 groups	N_{points} = 20,000	240,000

4.2 Vivado-HLS Extensions for MapRecude

During the development of HLSMapReduceFlow we faced several limitations regarding the implementation of the complex Phoenix's API in Vivado HLS. To overcome these limitations we have extended the VivadoHLs with the following features:

– **Dynamic Memory Management**: The Phoenix framework highly uses malloc/free calls for effective memory operations and reduced run-time footprint during map and reduce tasks. All DMM functions are not sup-

ported by Vivado HLS tools. We replace DMM calls with static code allocation on the heap of every application's code segment. This replacement affects the BRAM resource utilization, while it also forces the predefined at compile time variable definition. When the application uses dynamic size for specific variables, e.g. the length of word that is searched in Word Count application, then the designer has to set a static maximum variable's size, thus decreasing runtime flexibility.

- **Pointer Manipulation**: The Phoenix framework uses direct memory addressing, using pointer-based memory access. Also it uses arithmetic operations, arithmetic re-interpretation, and pointer casting. However, none of these features is available in Vivado HLS. We had to refactor the code by eliminating such coding forms.
- **Data structures**: The Phoenix framework uses a lot of complex data structures, i.e. structs with array and pointer elements. While scalar pointers that point to statically reserved data are normally deployed in Vivado HLS, the same does not happen with double and beyond pointers, i.e. pointer-to-pointer. We had to refactor such complex data types to simple scalar or simple pointer based structures.
- **ANCI C synthesizable subset**: The Phoenix framework uses many functions of ANCI C that are not synthesizable by Vivado HLS., e.g. limitation of memory copy operations such as memmove, memcpy, etc., string functions, e.g. strcmp, strlen, strcpy, toupper, etc. and math functions, e.g. rand, sort, etc.. For all of these functions we developed synthesizable versions, working on byte/cycle rate. Depending on application characteristics, we customized these functions to be more efficient using pipelining and loop unrolling techniques.

5 Configurable Reduce accelerator

In the original implementation of the MapReduce framework, every core processes a specific portion of the input data and whenever it encounter the predefined keys, it emits the key and the value to the *Reduce* tasks. The *Reduce* tasks merge all intermediate values associated with the same intermediate key. Every time that a *key/value* pair has to be updated with the new value, the processor has to load the key and the value from the memory, to process (e.g. accumulate) the old and the new value and then to store back the key and the value. Even if the *key/value* pair is in the cache, this operation takes many CPU clock cycles. Furthermore, several Reduce threads have to be deployed across the distributed resources increasing the execution time due to the communication overhead. The proposed Reduce accelerator is used to replace the Reduce threads (Figure 1) with a single special unit that is used to store and automatically process (e.g. accumulate)the values of the keys in MapReduce application. In essence, the Reduce accelerator is used for the efficient implementation of the *Reduce* tasks that merges all intermediate values associated with the same intermediate key.

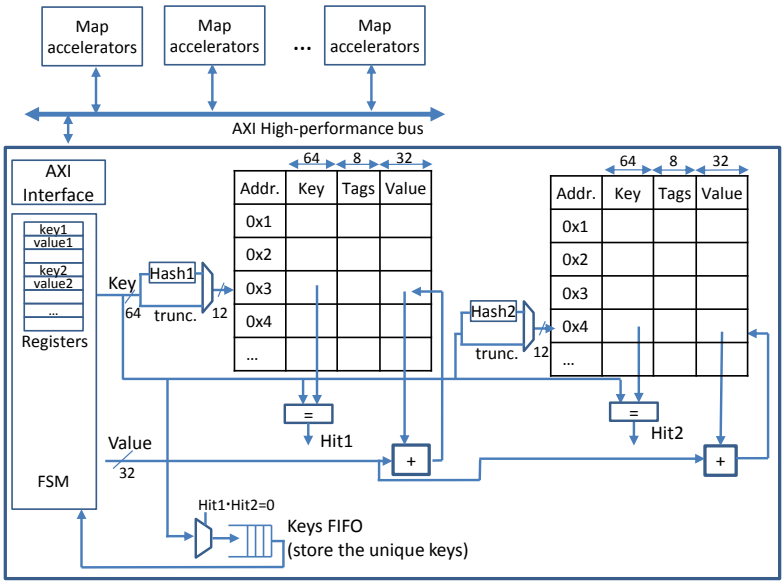

Fig. 5 Block diagram of the MapReduce accelerator

The main advantages of the proposed Reduce accelerator are twofold: Firstly, it is a separate memory structure that is used only for the key/value pairs and thus it decreases the possibility of a cache miss if the *key/value* pairs were stored in the ordinary cache. Secondly, it merges efficiently the storing and the processing of the MapReduce values since there is an accumulator that can accelerate the addition of the current value with the new value every time that a processor emits a new value for a key that already exists. Therefore, it can reduce significantly the number of instructions that are required to accumulate the value of a *key/value* pair.

5.1 Memory Architecture

The architecture of the MapReduce scratchpad memory is depicted in Figure 5. Each row of the scratchpad memory stores the key, the tags and the value for each MapReduce *key/value* pair. Since the key can be several bytes long, a hash unit is used to reduce the number of bytes to the maximum size of the cache.

The hash function can accelerate the indexing of the keys but it may create collision in case those two different keys have the same hash value. To address this problem, cuckoo hashing has been selected for resolving hash collisions. Cuckoo hashing [6] uses two hash functions instead of only one. When a new entry is inserted then it is stored in the location of the first hash key. If the entry is occupied the old entry is moved to its second hash address and the

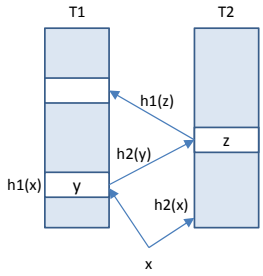

Fig. 6 Cuckoo hashing

procedure is repeated until an empty slot is found. This algorithm provides constant lookup time $O(1)$ (lookup requires just inspection of two locations in the hash table) while the insert time depends on the cache size $O(n)$. In case that the procedure enters an infinite loop the hash table is rebuild.

The cuckoo hashing algorithm can be implemented using two tables T1 and T2 for each hash function, each of size r. For each of these elements, a different hash function is used, $h1$ and $h2$ respectively, to create the addresses of T1 and T2 (Figure 6. Every element x is stored either in T1 or in T2 using hash function h1 or h2 respectively (i.e. $T1[h1(x)]$ or $T2[h2(x)]$. Lookups are therefore straightforward. For each of the element x that we need to look we just check the two possible locations in tables $T1$ and $T2$ using the hash functions $h1$ and $h2$, respectively.

To insert an element x, we check if $T1[h1(x)]$ is empty. If it is empty, then we store it in this location. If not, we replace the element y that is already there in $T1[h1(x)]$ with x. We then check if $T2[h2(y)]$ is empty. If it is empty, we store it in this location. If not, we replace the element z in $T2[h2(y)]$ with y. We then try to place z in $T1[h1(z)]$, and so on, until we find an empty location. According to the original cuckoo hashing paper [6], if an empty location is not found within a certain number of tries, the suggested solution is to rehash all of the elements in the table. In the current implementation, whenever the operation enters in such a loop it stops the operation and return zero to the function call. The function call then it may initiate a rehashing or it may select to add the specific key in the software memory structure as in the original code.

Two block RAMs are used to store the entries for the two Tables, T1 and T2 as it is shown in Figure 5. These block RAMs store the key, the value and the tags. In the tag field one bit is used to indicate if a specific row is valid or not. Two hash functions are used based on simple XOR functions that map the key to an address for the block RAMs. Every time that an access is required to the block RAMs, the hash tables are used to create the address and then two comparators are used that indicate if there is a HIT on the block RAMs (i.e. the key is the same as in key in the RAM and the valid bit is 1). The system is controlled by the Control unit (depicted in Figure **??**). The Control Unit of the MapReduce scratchpad memory is implemented as a Finite State Machine (FSM) that executes the cuckoo hashing.

Table 2 Hash and Array mode configuration

Keys	Description	Hash units
:	Map tasks can emit any key	Enabled
*:k	Map tasks can emit to a fixed number of keys	Disabled

5.2 Accessing the keys

In many applications that are based on the MapReduce framework, the number of keys is deterministic and limited to a certain number. For example, in the *Histogram* application, the number of keys is limited and we are only interested about the final value of each key. Therefore, after the processing of the image, the processor can gather all of the *key/value* pairs just by requesting the value of a certain key.

However, in other applications like the *WordCount* application that will be described in the next section, the keys and the number are not known in advance. Therefore, after the end of the processing, the processors cannot request for the values of the keys since the keys are not know in advance. In the software implementation of the MapReduce, this problem is resolved by keeping a linked list that contains all of the keys. In the case of the software-hardware implementation using the proposed hardware accelerator, this issue can be resolved either in the software domain or in the hardware domain. In the latter case, the processors can keep a linked list of all the keys that have been emitted to the hardware accelerator. However, this solution requires a fast indexing of the keys as in the original code which may increase significantly the total execution time.

To solve this problem a hardware equivalent of the linked list has been implemented and augmented to the hardware accelerator. The hardware linked list is actually a FIFO that keeps all of the keys that have been inserted into the block RAMs (as it is depicted in Figure 5). Every time that a new *key/value* pair is added to the block RAMs of the hardware accelerator, the key is added to the FIFO. If the key is already in the block RAMs then the key does not have to be inserted in the FIFO. In this way, at the end of the processing the FIFO hosts all the unique keys stored in the block RAMs.

However, in case that the keys are known in advance, the proposed scheme can be configured to bypass the hashing units. In that case, the control unit receives the keys and the keys are directly used as index in the memory blocks. For example, in the case of the histogram applications, the keys are integer ranging from 0 to 255. Hence, the keys can be directly used as an index to the memory blocks (and a constant offset can be added to the key for different colors; blue; green and red). After the processing of the data, the user can just read the values for each key directly from the memory units, without the need of the hash units.

Table 3 Reduce coprocessor configurations

Function	Hashing	Processing
Version1	YES	Accummulate
Version2	NO	Accummulate
Version3	YES	Calc. Average
Version4	NO	Calc. Average

5.3 Configurable Reduce accelerator

As it was described in the previous section, each application that is based on the MapReduce framework may have different requirements for the Reduce function. In the *Wordcount*, the *Histogram* and the *Linear Regression* applications, the *Reduce* function needs to accumulate the values for each key; In the *KMeans* application, the *Reduce* function calculates the average of the values for each key.

On the other hand, in the *Wordcount* and the *KMeans* applications the keys needs to be stored using a hash function while in the case of the *Histogram* and the *Linear Regression*, the keys are used directly as an index in the memory structures. Therefore, four different versions of the MapReduce accelerator has been developed as it is shown in the Table 3. The first and the third version uses hashes to index the keys, while the second and the forth version uses directly the keys as an index. The first and the second version just accumulate the values of each key while the last versions calculate the average value for each key.

6 Performance evaluation

This section describes the experimental setup we used to evaluate the proposed integrated platfrom for MapReduce acceleration using FPGAs as well as the respective obtained results. We evaluated the efficiency of the proposed HLS MapReduceFlow framework considering a MapReduce accelerator for a FPGA-based architecture, targeting to emerging application domains, e.g. artificial intelligence, scientific computing, enterprise computing etc. In this paper, we considered six applications evaluation test-bed of Phoenix MapReduce framework for shared-memory systems [9]. The performance evaluation covers a representative set of application that typically use the MapReduce framework. The characterization setup of the employed applications is summarized in Table 1.

To evaluate our framework in performance and scalability, we build up a testbed for the HLSMapReduceFlow. Since the main scope of this work is the acceleration of Map tasks (95% of total execution time in Phoenix), we explore different architecture exploiting Map accelerators in the FPGA, while for the following measurements we have used the developed Reduce co-processor. Also we do not measure communication overhead for transferring input data streams to the FPGA. Instead we use the on-board FPGA memory to store

Table 4 Real-word representative comparison between HLSMapReduceFlow-accelerated FPGA and commodity workstation.

Framework	**GNU/Linux 3.18.6 x86-64**			**HLSMapReduceFlow**						
Platform	**AMD 8-core FX-8350 4GHz**			**Virtex7-XC7VX485T 150MHz**				**Ratio**		
Metrics	**Time(ms)**	**Power(W)**	**Energy(J)**	T_p**(ms)**	T_c**(ms)**	**Power(W)**	**Energy(J)**	**T**	**P**	**E**
Histogram	344	41.1	14.1	72.2	4.8	1.84	0.13	0.21	0.04	0.009
Matrix Mul/tion	177	41.3	7.3	208	0.6	1.02	0.21	1.17	0.03	0.029
String Match	206	41.6	8.5	95	4.9	2.33	0.22	0.46	0.06	0.026
Word Count	172	40.8	7.0	84	1.4	1.87	0.16	0.48	0.05	0.023
Linear Reg/sion.	158	41.6	6.6	73	6.4	2.08	0.15	0.46	0.05	0.023
PCA	392	41.9	16.4	964	4.1	1.17	1.13	2.45	0.03	0.070
K_{means}	435	40.3	17.5	503	3.8	1.03	0.52	1.16	0.03	0.029
Average	269	41.2	11	285	3.7	1.62	0.36	**1.06**×	**0.04**×	**0.03**×

input streams. This scheme may not be a complete architecture for datacenters where new requests are coming constantly. However in this work we study the performance micro-architecture exploitation by instantiating dataflow-based Map accelerators in the FPGA, regardless the input source and the way input data are reaching the Map tasks. However, the communication overhead for transferring the data in and out of the FPGA could be hided by pipelining the communication and the computation of the key/ value pairs. In that case, the next frame of data could be stored while the current frame of data is being processed by the MapReduce accelerators.

6.1 Area resources

Figure 7 shows the self overall performance-scalability tradeoff results for every employed application, when we use, or not, the HLSMapReduceFlow framework. Every horizontal axis scales the Map accelerators from single-instance to the maximal number of accelerators. This number is limited by the reserved FPGA resources of applications. As shown by the comparison of the architecture without the MapRecude framework (No-MR), and the 1-Map accelerator instance, the implementation of HLSMapReduceFlow framework introduces a cost in both resources and execution time by a factor of 18% and 38% respectively. However, as long as more Map accelerators are instantiated, the performance in terms of throughput is almost linear boosted. However some applications reach a saturation point where the instantiation of more accelerators does not lead to expected speedup. This is the case for String Match, PCA, Kmeans and Word Count. We found that both the dynamic behavior of these applications and data dependency among calculations prevents Vivado HLS for applying effective dataflow processing optimizations. For instance, the PCA kernel is a streaming application with no dynamism. However its computations have high data dependencies without equivalent data locality. Thus, the fine-grain splitting of input data to data chunks that include elements needed by more than one map accelerators, causes performance drop due to stalled Map processing tasks.

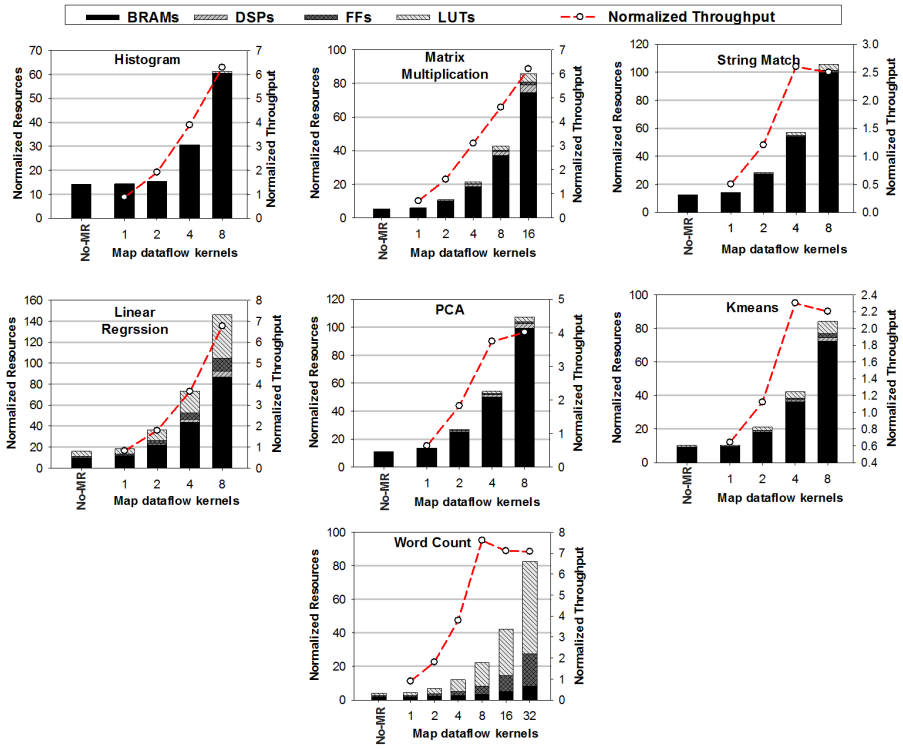

Fig. 7 Self performance-scalability tradeoff of HLSMapReduceFlow framework.

In order to provide a more real-word representative comparison, we evaluated HLSMapReduceFlow against a high-end workstation. The workstation is powered by the 8-core AMD FX-8350 processor clocked at 4GHz. This processor has a TDP value of 125 Watts. We compiled the employed applications using GCC compiler (v4.9.2) and run the applications with glibc runtime linking, in a GNU/Linux (Kernel 3.18.6) 64-bit OS, enabling many compiler omptimizations (-O2), including vector processing ones (SSE, AVX etc.). The derived measurements for execution time, power and energy are shown in the first three columns of Table 4. The next three columns show the respective metrics for a system composed of a Virtex-7 FPGA (XC7VX485T) clocked at 150MHz utilizing the HLSMapReduceFlow framework. The PC-FPGA communication is established with a PCI Express 3.0 link, offering maximum bandwidth of 8Gbps. The overall measured time for the FPGA deployment is represented by the time for processing on the FPGA, T_p and the time for PC-FPGA communication, T_c (downloading data from PC to FPGA and uploading results from FPGA to PC). In the current design, data are stored in the block RAMs (BRAMs) embedded in the FPGAs. In order to hide the communication overhead we could pipeline the I/O transfer with the computation tasks. For example we could use additional BRAMs to store the

next stream of data that is going to be processed by the FPGA while computing the current stream of data. The power values have been derived through the usage of PowerTop[1] utility for the CPU and Xilinx Xpower[2] utility for the FPGA. As shown, the proposed framework delivers extremely performance-per-watt efficient solutions, reporting two orders of magnitude less energy for the same execution timing window. Consequently we show that the proposed scheme is an elegant candidate implementation infrastructure for data centers that promises high energy efficiency for specific types of applications.

In the current configuration, the memory structure of the Reduce accelerator has been configured to host 2K *key/value* pairs. Each key can be 64-bits long and the value can be 32-bits long. The total size of the memory structure is $2K \times 104$ bits. The first 64 bits are used to store the key in order to compare if we have a hit or a miss using the hash function. The next 8 bits are used for tags and the next 32 bits are used to store the value. In the current configuration the maximum value of key is 64 bits and a hash function is used to map the *key* (64 bits) into the address (12 bits).

The main advantage of the implementation in an FPGA is that this scratch-pad is configurable and it can be tuned based on the application requirements. The maximum size of each key is 8 bytes in the current configuration. In applications that the key is larger than 8 bytes for a specific application then the user can either increase the size of the words or follow a hybrid approach. In the hybrid approach, keys that are smaller than 9 bytes are stored in the hardware accelerator while keys that are 9 bytes or longer (rare case) are stored in a software structure. In total, the Reduce co-processor occupies 4184 Look-Up Tables (LUts) (8% of the overall area) and 29 BRAMs.

7 Conclusions

The performance evaluation of the integrated platform shows that FPGA-based acceleration of MapReduce applications can be used to significantly reduce the energy consumption in the data centers. The FPGA-based acceleration can provide up to 2 orders of magnitude lower energy consumption compared to the typical General Purpose Processors (GPPs). The proposed platform that extends the HLS flow to support the required programming structures for the Map functions allows the fast and efficient implementation of the required Map accelerators. The Reduce accelerator can also be configured to meet the application's requirements and speedup the processing of the key/value pairs. Overall, the proposed integrated platform shows how FPGA-based acceleration could utilized in data centers to reduce the energy consumption and face the demanding increase of the network traffic.

[1] https://01.org/powertop

[2] http://www.xilinx.com/products/design_tools/logic_design/verification/xpower.htm

References

1. *SMART 2020: Enabling the low carbon economy in the information age*. A report by The Climate Group on behalf of the Global eSustainability Initiative (GeSI), 2008.
2. *Where does power go?* GreenDataProject, available online at: http://www.greendataproject.org, 2008.
3. *Make IT Green, Cloud Computing and its Contribution to Climate Change*. Greenpeace, March, 2010.
4. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commununications of the ACM*, 51(1):107–113, Jan. 2008.
5. U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.
6. R. Pagh and F. F. Rodler. Cuckoo Hashing. *Proceedings of ESA 2001, Lecture Notes in Computer Science*, 2161, 2001.
7. A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.
8. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, 2007.
9. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24, Feb 2007.
10. Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. FPMR: MapReduce Framework on FPGA: A Case Study of RankBoost Acceleration. In *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 93–102, 2010.
11. A. H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, Dec. 1986.
12. D. Yin, G. Li, and K.-d. Huang. Scalable mapreduce framework on fpga accelerated commodity hardware. In S. Andreev, S. Balandin, and Y. Koucheryavy, editors, *Internet of Things, Smart Spaces, and Next Generation Networking*, volume 7469 of *Lecture Notes in Computer Science*, pages 280–294. Springer Berlin Heidelberg, 2012.