High-level Synthesizable Dataflow MapReduce Accelerator for FPGA-coupled Data Centers

Dionysios Diamantopoulos[1] and Christoforos Kachris[2]

[1]School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2]Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece
Email: diamantd@microlab.ntua.gr, ckachris@ee.duth.gr

***Abstract*—Manipulating big-data entries of emerging server workloads requires a design paradigm shift towards more aggressive system-level architecture solutions. From software perspective, the MapReduce framework is a prominent parallel data processing tool as the volume of data to analyze grows rapidly. FPGAs can be used to accelerate the processing of data and reduce significantly the power consumption. However, FPGAs have not been deployed in data centers due to the high programming complexity of hardware. In this paper we present HLSMapReduceFlow, i.e. a novel reconfigurable MapReduce accelerator that can be scaled-up to data centers and it can speedup the processing of Map computation kernels, while promising minimum energy footprint and high programming efficiency due to the use of HLS. We propose the complete decoupling of MapReduce's tasks data-paths to distinct buses, accessed from individual processing engines. Such a dataflow approach implies a holistic C/C++ to RTL domain-level MapReduce transition. In this work, we further extent HLS tools, with systematic source-to-source code annotation of HLS optimization directives, by adding as a state-of-art system-level implementation toolflow. The proposed architecture is implemented, mapped and evaluated to a Virtex-7 FPGA and shows that the proposed scheme can achieve up to 4.3× overall throughput improvement in MapReduce applications, while offering two orders of magnitude power/energy improvements compared to a high-end multi-core processor.**

***Keywords* - MapReduce, Hardware accelerator, High-Level Synthesis, Reconfigurable computing, Dataflow computing**

I. Introduction

Breaking the exascale barrier [1] has been recently identified as the next big challenge in computing systems. Manipulating huge data entries requires a design paradigm shift towards more aggressive system-level architecture solutions. One of the main promising programming frameworks for processing large data sets in the data centers and other clusters of computers is the MapReduce framework [2]. MapReduce was firstly inspired by Google for application development on data-centers with thousands of servers. It allows programmers to write functional-style code that is automatically parallelized and scheduled in a distributed system.

MapReduce can be used to easily utilize the resources of large distributed systems for processing large data sets. However, such large data sets, that the data centers have to process under constrained time and energy budgets, have increased significantly, due to emerging applications like big data and cloud computing. The increase of the traffic in the data centers has also resulted to higher power consumption. The server processors need to provide higher throughput without consuming excessive power. Currently, one of the main challenges in the data center operators is the power consumption of the servers that account for over 45% of the overall power consumption in the data centers.

Therefore, novel architectures are required that can increase the performance of the data centers and also be more energy efficient. FPGAs can be utilized to increase the performance of the systems and also can be used to reduce the total power consumption due to the specialized accelerators for specific tasks. However, the main drawback of utilizing FPGAs in the data centers is the high programming complexity. In this paper, we present a novel framework that allow the seamless development of FPGA-based hardware accelerator for data centers by extending the current HLS toolflow to include the MapReduce framework. We develop several hardware accelerators for typical MapReduce application using the proposed framework and we compare the performance and the energy efficiency with high-end multi-core processors. The main contributions on this paper are the followings:

- a novel HLS-based MapReduce dataflow architecture,
- development of several hardware accelerators for typical MapReduce application based on the HLS-enabled MapReduce dataflow architecture
- performance evaluation on typical MapReduce applications (wordcount, histogram, etc.) on a Virtex7 FPGA that shows up to 4.3x throughput gains and
- up to two orders of magnitude energy consumption savings

The main advantage of the proposed scheme is that we utilize the available resources of the FPGA to achieve higher parallelism. In the terms of raw performance the proposed scheme is comparable to the General Purpose Processor (GPP), but by exploiting the parallelism and much lower clock frequency compared to GPP, we can achieve much lower power consumption. The rest of the paper is organized as follows. Section 2 reviews prior art regarding MapReduce implementations, while Section 3 presents our FPGA-based

The research project is partially implemented within the framework of the Action "Supporting Postdoctoral Researchers" of the Operational Program "Education and Lifelong Learning" (Action's Beneficiary: General Secretariat for Research and Technology), and is co-financed by the European Social Fund (ESF) and the Greek State.
Christoforos Kachris is now with Athens Information Technology (AIT).

implementation. Section 4 presents the evaluation results and Section 5 concludes the paper.

II. Related Work

We survey related work on accelerator-based implementations of MapReduce framework. In [3], a reconfigurable MapReduce framework is presented but the proposed scheme is implemented as a custom RTL-design that is used to implement only the RankBoost application entirely on an FPGA. Although the basic architecture of this work is very close to the one presented in this paper, we would like to note that the proposed approach is implemented in C/C++ level and it is seamlessly synthesized to RTL level using HLS tools, thus its employment is highly transparent to new applications exploiting MapReduce framework. On the contrary, in [3] both of the Map and Reduce tasks for a specific application are mapped to custom RTL logic and thus for any new application a new design has to be implemented. In [4] 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. Although this approach offers high flexibility and run-time optimization of the framework, it still increases the programming difficulty of both custom-RTL and CPU for every node, while new applications have to be custom tailored to such a diverse hybrid-node implementation layer. Regarding general purpose GPU platforms, MapReduce framework was also explored [5]. “However, GPU prefers coalesced memory access pattern, which makes it fumble while dealing with complex data structure and the SIMT architecture restricts its computation performance to handle irregular applications” [3]. Authors in [6] adopted a hybrid architecture approach combining both GPU and FPGA to implement a MapReduce framework, which leaves scheduling work to the host CPU and employs GPU and FPGA for co-processing, while in [7] a MapReduce framework is implemeted targeting to an embedded many-core Network-on-Chip platform. However, regardless the implementation medium, a recent MapReduce survey verifies that MapReduce technique shall complement database management system with scalable and flexible parallel processing for various data analysis such as scientific data processing [8]. In [9] Microsoft has used FPGAs to increase the performance of the page ranking applications. More specifically a medium-scale deployment on a bed of 1,632 servers, measuring its efficacy in accelerating the Bing web search engine, reported improvements on the ranking throughput of each server by a factor of 95%. However, until now the utilization of FPGAs in data centres is limited mainly due to the high programming complexity of FPGAs. However this work is a standalone research and currently there is no wide adoption of FPGAs in the data centers.

In [10], we have previously developed a hardware acceleration unit for the MapReduce framework that can be combined efficiently with ARM cores in fully programmable platforms . To develop and evaluate the proposed scheme, authors selected the Xilinx Zynq-7000 All Programmable SoC, which comes hardwired with a dual-core Cortex-A9 processor on-board.

In essence, this accelerator was used to alleviate the processors from executing the Reduce tasks, and thus executing only the Map tasks and emitting the intermediate key/value pairs to the hardware acceleration unit that performs the Reduce operation. The performance evaluation shows that the proposed accelerator can achieve up to 1.8x system speedup of the MapReduce applications. Motivated by these results, we identify the following performance-boost limitations in this work:

- Low parallelism exploitation: The partitioner, mapper and scheduler are controlled by the same on-board CPU, i.e. ARM Cortex-A9 on Zynq, which may limit the inherent parallelization opportunity of these tasks.
- High memory conflicts: Although the *reduce* tasks may exploit DMA engines to work directly to memory, still the main framework, including both the MapReduce tasks and the working kernels, have their data-path on the same AXI bus, thus the memory is over-populated by read/write (R/W) calls from the multiple processing threads, which eventually are serialized.
- Low acceleration opportunity regarding overall system execution time breakdown, due to the speed-up of only Reduce step. According to [11], the Reduce step accounts for less than 5% of the total execution time in the original implementation of a MapRecude framework, i.e. Phoenix, using a chip multi-processor (CMP) with 2, 4 and 8 cores[1]. The Figure 1 presents the execution time breakdown between Map, Reduce, and Merge tasks for the CMP system. It is evident that the execution time of Map task, where the main algorithm’s processing occurs, is significantly reduced by the Phoenix framework, as long as more cores are employed, which motivate us to investigate acceleration scenarios for map tasks on the coarse-grain parallelization potential of FPGA devices.

Identifying these issues, 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-level 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.

III. HLSMapReduceFlow Architecture

A. Phoenix MapRecude Framework

We adopt the open-source Phoenix MapRecude framework [11] as the initial code base of our work. In this framework, 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. Programs written in this functional style are automatically parallelized and executed on a large cluster of computation nodes. The run-time system takes care of the details of partitioning the input data, scheduling

[1]The referred CMP system is based on the UltraSparc T1 multi-core chip with 8 multithreaded cores sharing the L2 cache.

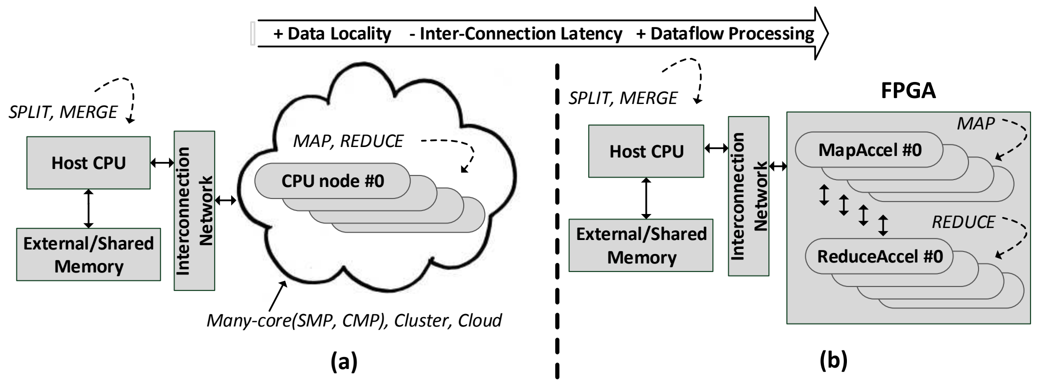


Fig. 2. Architecture topology of a) original MapRecude framework and b) proposed HLSMapReduceFlow.

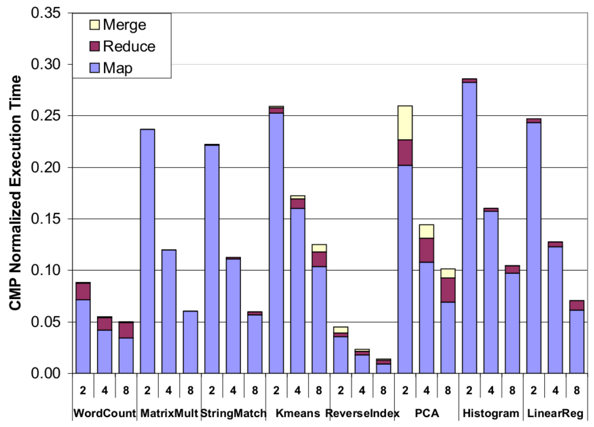


Fig. 1. Execution time breakdown for a CMP system on Phoenix MapRecude framework [11].

the program's execution across a set of computation nodes, providing fault-tolerance, and managing the required inter-machine communication. This allows programmers without any experience of parallel and distributed systems, to easily utilize the resources of a large distributed system.

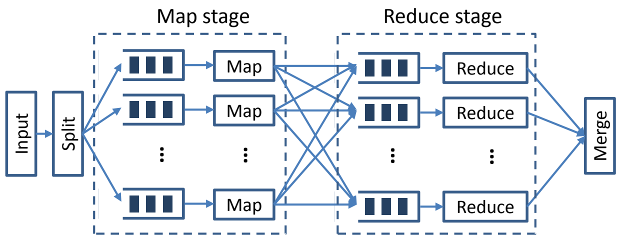


Fig. 3. The MapReduce programming framework.

The Figure 3 shows the basic data flow for the runtime system. The runtime is controlled by the scheduler, which is initiated by user code. The scheduler creates and manages the threads that run all Map and Reduce tasks. It also manages the buffers used for task communication. After initialization, the scheduler determines the number of cores to use for this computation. For each core, it spawns a worker thread that is dynamically assigned some number of Map and Reduce tasks.

To start the Map stage, the scheduler uses the Splitter to divide input pairs into equally sized units to be processed by the Map tasks. The Splitter is called once per Map task and returns a pointer to the data the Map task will process. The Map tasks are allocated dynamically to workers and each one emits intermediate $< key, value >$ pairs. The Partition function splits the intermediate pairs into units for the Reduce tasks. The function ensures all values of the same key go to the same unit. Within each buffer, values are ordered by key to assist with the final sorting. At this point, the Map stage is over. The scheduler must wait for all Map tasks to complete before initiating the Reduce stage. Reduce tasks are also assigned to workers dynamically, similar to Map tasks. The one difference is that, while with Map tasks we have complete freedom in distributing pairs across tasks, with Reduce we must process all values for the same key in one task. As the last step, the final output from all tasks is merged into a single buffer, sorted by keys.

B. *Dataflow FPGA-based Acceleration*

The basic proposed architecture scheme is inspired by our group's prior novel implementation, presented in [10]. Authors developed a MapReduce configurable accelerator which is used to alleviate the processors from executing the Reduce tasks, and thus executing only the Map tasks and emitting the intermediate key/value pairs to the hardware acceleration unit that performs the Reduce operation. On top of this architecture we further built the acceleration infrastructure for the Map tasks. Figure 2 shows a high-level differentiator of the proposed architecture, compared to the current state of art.

Originally, the Map and Reduce tasks are running as software threads on the CPU cores of the available MapRecude deploying infrastructure (Figure 2(a)). The success of such an architecture relies on the availability of a large shared-memory that facilitate communication without excessive data copying. Moreover, at runtime, an efficient scheduler is highly required to schedule tasks dynamically across the available processors in order to achieve load balance and maximize task throughput. In the previous approach we highlight two major drawbacks:

- A shared-memory is usually available in monolithic datacenter architectures. Such architectures are composed by single-board/single-die many-core systems (CMP, SMP). However, there is limited shared-memory organization support for clusters and cloud datacenter topologies. Thus in case of inefficient data splitting, the shared data have to travel along different datacenter nodes, reserving resources and spending energy. Such an inefficiency in data splitting procedure may be the case in which the same subset of data is required by two Map steps which have been scheduled in long-distance computation nodes.

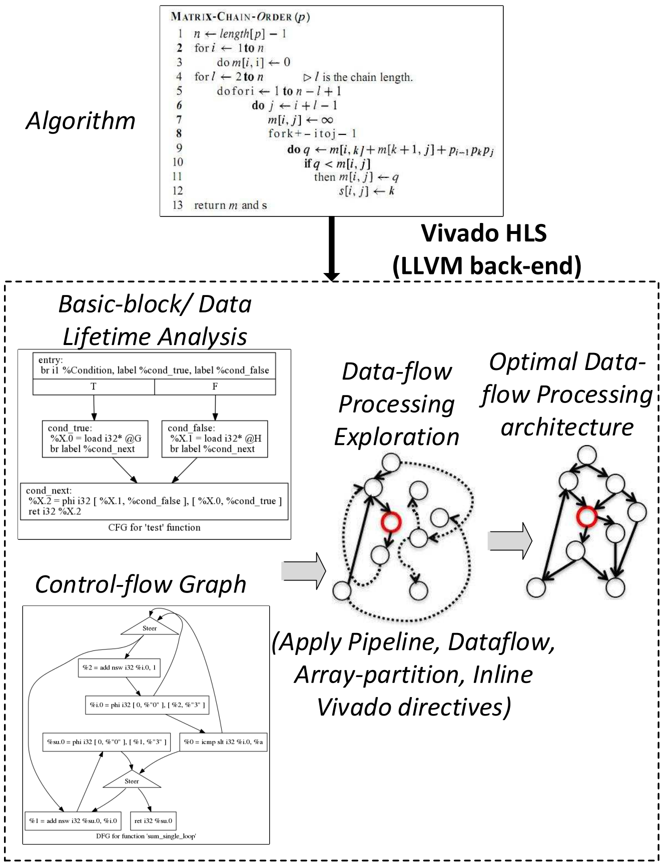
- So far, the state-of-art MapReduce implementations do not provide an efficient scheduler that checks the inter-application control and data flow graphs, prior to scheduling. Thus, it turns out that there is not real application partitioning and scheduling among computation nodes, but rather a quick-and-dirty application's runtime slicing, followed by a first-come-first-serve distribution on computation nodes.

We propose to 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, 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). Based on such information, we built the corresponding optimal Map accelerator engines. Figure 4(a) shows the basic HLSMapReduceFlow architecture. While, this looks similar to the original Phoenix architecture, we highlight in Figure 4(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 parallel-ready computation paths. For this step we employ the Vivado HLS tool. Apart from typical high level synthesis steps, i.e. resource binding, scheduling etc., Vivado HLS also 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.



```
MATRIX-CHAIN-ORDER (p)
n ← length[p] − 1
for i ← 1 to n
    do m[i, i] ← 0
for l ← 2 to n            ▷ l is the chain length.
    do for i ← 1 to n − l + 1
        do j ← i + l − 1
            m[i, j] ← ∞
            for k ← i to j − 1
                do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                            s[i, j] ← k
return m and s
```

```
entry:
 br i1 %Condition, label %cond_true, label %cond_false

cond_true:
 %X.0 = load i32* @G
 br label %cond_next

cond_false:
 %X.1 = load i32* @H
 br label %cond_next

cond_next:
 %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
 ret i32 %X.2
```

Algorithm
Vivado HLS
(LLVM back-end)
Basic-block/ Data Lifetime Analysis
T
F
CFG for 'test' function
Control-flow Graph
Steer
%2 = add nsw i32 %i.0, 1
%i.0 = phi i32 [0, %"0"], [%2, %"3"]
%su.0 = phi i32 [0, %"0"], [%1, %"3"]
%0 = icmp slt i32 %i.0, %a
Steer
%1 = add nsw i32 %su.0, %i.0
ret i32 %su.0
DFG for function 'sum_single_loop'
Data-flow Processing Exploration
Optimal Data-flow Processing architecture
(Apply Pipeline, Dataflow, Array-partition, Inline Vivado directives)

Fig. 5. 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 [12], 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.

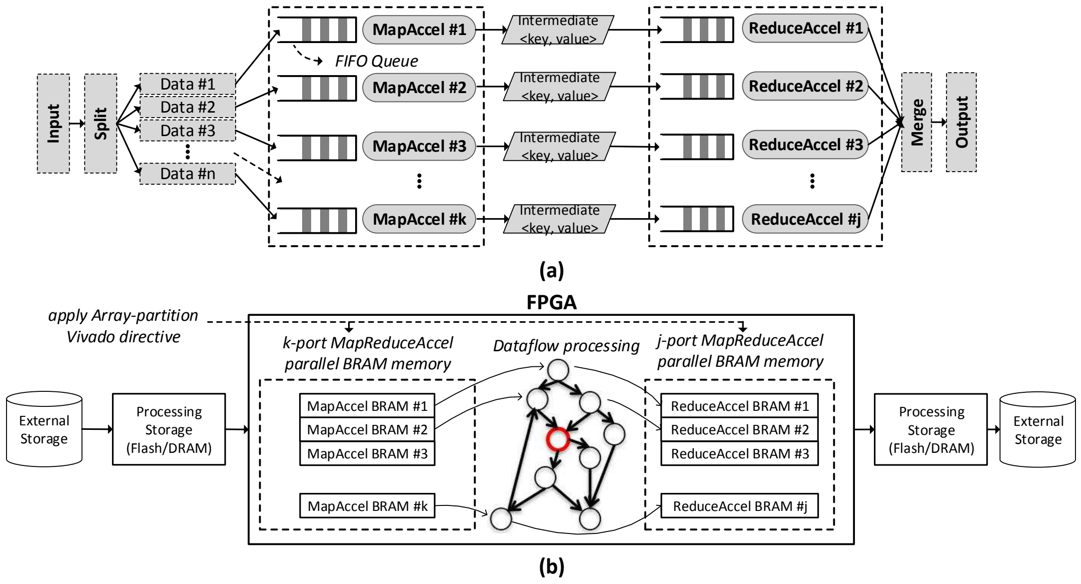
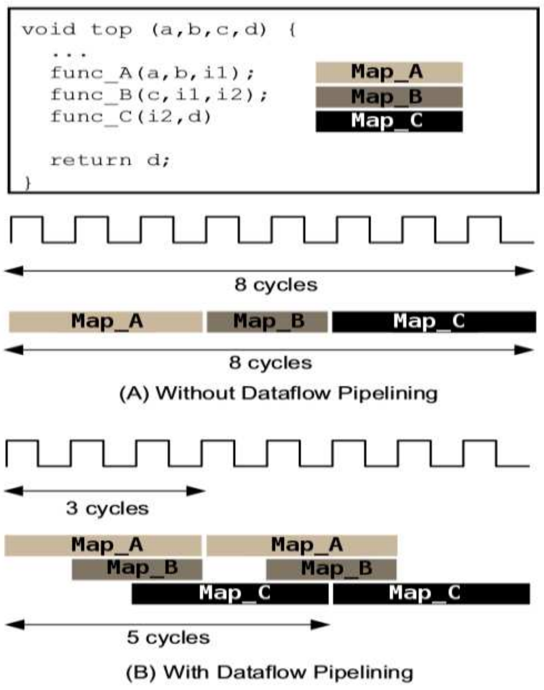


Fig. 4. HLSMapReduceFlow dataflow architecture: Every dataflow computation node is working in its unique memory. The system memory is partitioned to *k-port* and *j-port* banks for the *k*-map and *j*-reduce tasks respectively.

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 5 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) (Figure 6(a)) and creates a parallel process architecture from it (Figure 6(b)). Dataflow optimization in Vivado HLS is a very powerful method for improving design throughput. The channels shown in Figure 6(a) ensure a task is not required to wait until the previous task has completed all operations before it can begin. Figure 6(b) shows how DATAFLOW optimization allows the execution of tasks to overlap, increasing the overall throughput of the design and reducing latency.

C. HLSMapReduceFlow Methodology for Vivado-HLS

Figure 7 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



```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

Fig. 6. (a) Sequential Functional Description (b) Parallel Process Architecture

into RTL implementation through the back-end of Vivado HLS tool.

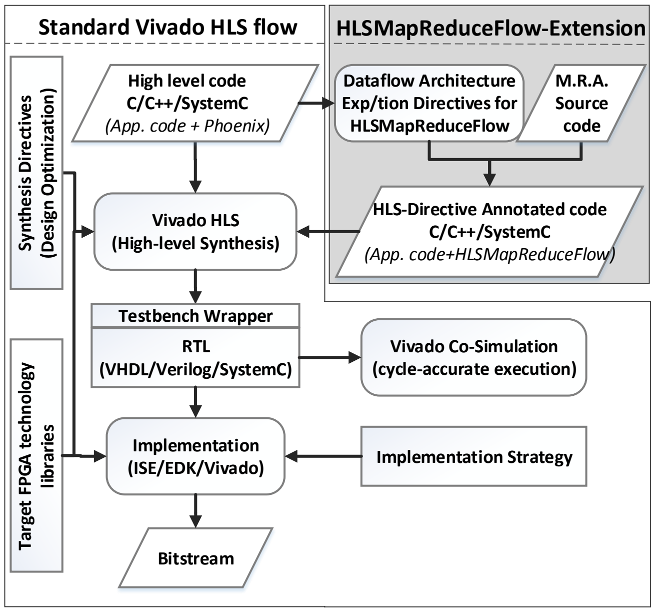
D. Vivado-HLS Limitations for MapRecude

During the development of HLSMapReduceFlow we faced several limitations regarding the implementation of the complex Phoenix's API in Vivado HLS. These limitations are reported as follows:

- **Dynamic Memory Management**: The Phoenix

TABLE I. APPLICATIONS CHARACTERIZATION

Domain	Kernel	Description	Parameters	Bytes/Iteration
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



Standard Vivado HLS flow
HLSMapReduceFlow-Extension
Synthesis Directives (Design Optimization)
High level code C/C++/SystemC (App. code + Phoenix)
Dataflow Architecture Exp/tion Directives for HLSMapReduceFlow
M.R.A. Source code
Vivado HLS (High-level Synthesis)
HLS-Directive Annotated code C/C++/SystemC (App. code+HLSMapReduceFlow)
Testbench Wrapper
RTL (VHDL/Verilog/SystemC)
Vivado Co-Simulation (cycle-accurate execution)
Target FPGA technology libraries
Implementation (ISE/EDK/Vivado)
Implementation Strategy
Bitstream

Fig. 7. Proposed extension on Vivado HLS flow to support MapReduce framework for FPGA-based systems.

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 supported 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.

IV. EXPERIMENTAL RESULTS

This section describes the experimental setup we used to evaluate HLSMapReduceFlow, as well as the respective obtained results. We evaluated the efficiency of the proposed HLSMapReduceFlow framework considering a MapRecude 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 [11]. 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 I.

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 only one Reduce task. 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 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.

Figure 8 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

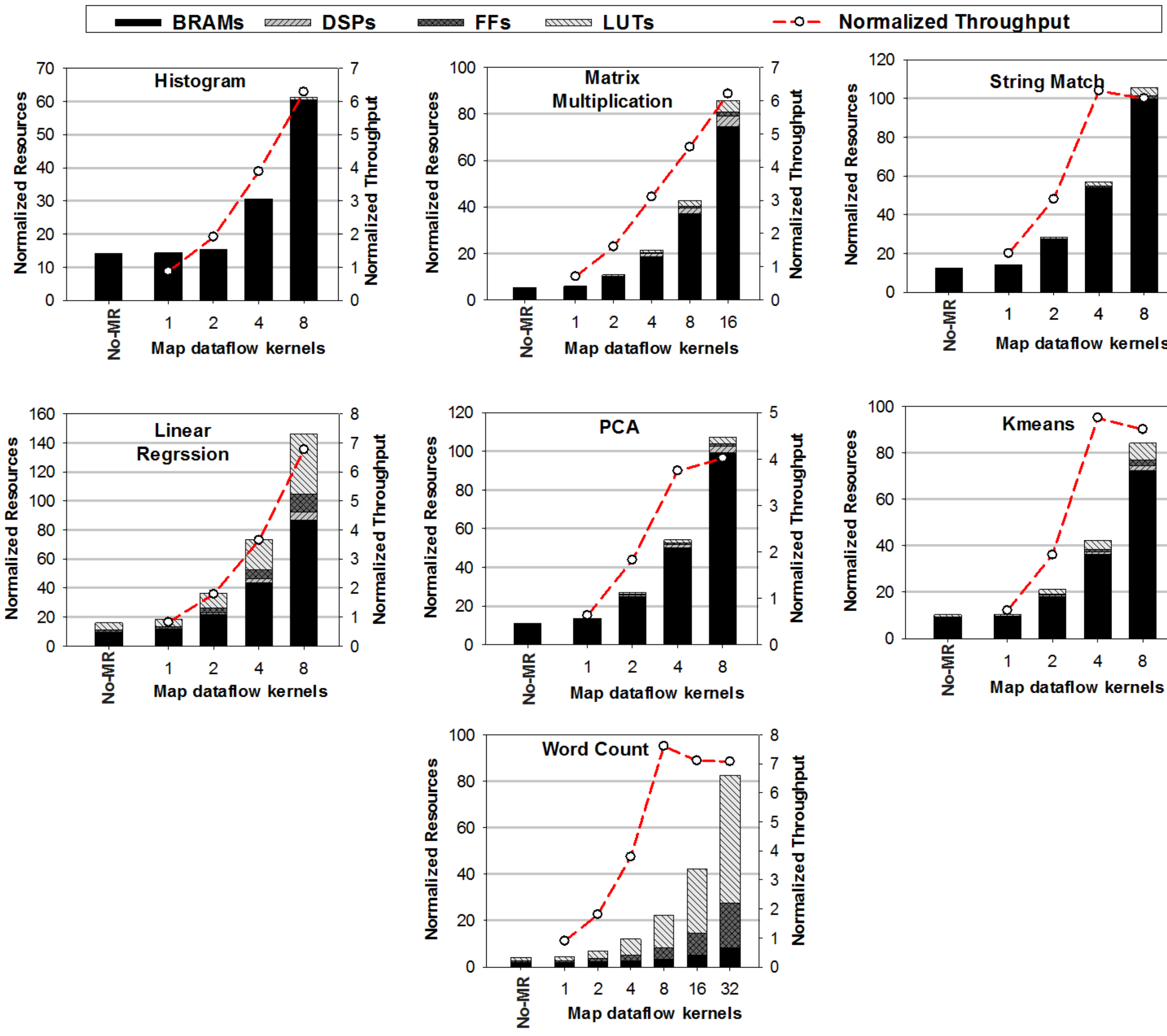


Fig. 8. Self performance-scalability tradeoff of HLSMapReduceFlow framework.

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.

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 II. The next four 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

TABLE II. Real-word representative comparison between HLSMapReduceFlow-accelerated FPGA and commodity workstation.

Framework	**GNU/Linux 3.18.6 x86-64 / GCC-glibc**			**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×**

the time for processing on the FPGA, T_p and the time for PC-FPGA communication, T_c (downloading input 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[2] utility for the CPU and Xilinx Xpower[3] 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.

V. Conclusions

This paper introduces HLSMapReduceFlow, a MapReduce framework on FPGA devices, which provides implementation abstraction, hardware architecture and an exemplary system-level framework for system designers. The HLSMapReduce-Flow framework can be widely used as a programming framework both for FPGA-coupled data centers and for cloud-computing applications. The proposed hardware accelerator can be used to reduce the total execution time for a workstation coupled with an FPGA device such as the Xilinx Virtex-7 and the cloud-computing applications based on the MapReduce framework by accelerating the Map/Reduce tasks of these applications. Due to micro-architecture exploitation offered by the employed state-of-art Vivado HLS tool, the framework succeed in high-parallelism by applying fundamental concepts from spatial and dataflow computing. We evaluated HLSMapReduceFlow with well-established server workloads against a high-end workstation and we obtained throughput gains of 4.3× while the FPGA device saved up to two orders of magnitude energy consumption. As a future step we plan to integrate to HLSMapReduceFlow our novel synthesizable dynamic memory management allocator [13], exploiting hardware memory paging capabilities and thus offering support for complex dynamic applications.

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

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

References

[1] Michael J. Flynn, Oskar Mencer, Veljko Milutinovic, Goran Rakocevic, Per Stenstrom, Roman Trobec, and Mateo Valero. Moving from petaflops to petadata. *Commun. ACM*, 56(5):39–42, May 2013.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), January 2008.

[3] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, New York, NY, USA, 2010. ACM.

[4] Dong Yin, Ge Li, and Ke-di Huang. Scalable mapreduce framework on fpga accelerated commodity hardware. In Sergey Andreev, Sergey Balandin, and Yevgeni Koucheryavy, editors, *Internet of Things, Smart Spaces, and Next Generation Networking*, volume 7469 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[5] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[6] Jackson H. C. Yeung, C. C. Tsang, K. H. Tsoi, Bill S. H. Kwan, Chris C. C. Cheung, Anthony P. C. Chan, and Philip H. W. Leong. Map-reduce as a programming model for custom computing machines. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 149–159, Washington, DC, USA, 2008. IEEE Computer Society.

[7] Konstantinos Gyftakis, Iraklis Anagnostopoulos, Dimitrios Soudris, and Dionysios Reisis. A mapreduce framework implementation for network-on-chip platforms. In *Electronics, Circuits and Systems (ICECS), 2014 21st IEEE International Conference on*, pages 120–123, Dec 2014.

[8] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4):11–20, January 2012.

[9] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014.

[10] C. Kachris, G.C. Sirakoulis, and D. Soudris. A reconfigurable mapreduce accelerator for multi-core all-programmable socs. In *System-on-Chip (SoC), 2014 International Symposium on*, pages 1–6, Oct 2014.

[11] 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.

[12] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.

[13] Dionysios Diamantopoulos, Sotirios Xydis, Kostas Siozios, and Dimitrios Soudris. Dynamic memory management in vivado-hls for scalable many-accelerator architectures. In *Proceedings of the 20015 11th International Symposium on Applied Reconfigurable Computing*, ARC 2015, 2015 - accepted for presentation in April, 2015.