Heterogeneous Architectures for Big Data Batch Processing in MapReduce Paradigm

M. Goudarzi, *Senior Member, IEEE*
Computer Engineering Department, Sharif University of Technology, Tehran, I.R.Iran

Abstract— The amount of digital data produced worldwide is exponentially growing. While the source of this data, collectively known as Big Data, varies from among mobile services to cyber physical systems and beyond, the invariant is their increasingly rapid growth for the foreseeable future. Immense incentives exist, from marketing campaigns to forensics and to research in social sciences, that motivate processing increasingly bigger data so as to extract information and knowledge so as to improve processes and benefits. Consequently, the need for more efficient computing systems tailored to such big data applications is increasingly intensified. Such custom architectures would expectedly embrace heterogeneity to better match each phase of the computation. In this paper we review state of the art as well as envisioned future large-scale computing architectures customized for batch processing of big data applications in the MapReduce paradigm. We also provide our view of current important trends relevant to such systems, and their impacts on future architectures and architectural features expected to address the needs of tomorrow big data processing in this paradigm.

***Index Terms*—Big data, hardware accelerator, FPGA, MapReduce, Hadoop, data center, efficiency.**

I. INTRODUCTION

Past few decades have constantly witnessed an increasingly deeper penetration of computers and various sensors into virtually all aspect of our daily lives. We collectively produce large amounts of data such that it is forecast [1] that the volume of digital data in 2020 will be 300 times that of 2005. Mining this data helps various entities to better accomplish their missions. This spans the growing interest in big data by companies for customer satisfaction to increase the profits, by health sector for better medicine and various healthcare services, by social scientists and politicians for predicting societal needs and trends, and even by police and security agencies for forensics. Processing this huge amounts of data is a daunting task that has motivated introduction of new programming paradigms, such as MapReduce [2], as well as computing infrastructures, such as Warehouse Scale Computers (WSC) [3].

Paradigms for big data analysis can be broadly classified into batch processing and stream processing. As the names imply, the former is used when the data is already collected, such as the case of index generation for internet-wide search by Google, whereas the latter is typically used when the data is produced online and is meant to be processed on the fly, such as the case of analyzing the twits posted on Twitter. Here we focus on the former class and the architectures to improve its performance. MapReduce [2] introduced by Google is among the most widely used programming paradigms in this class, and Hadoop [4] is its open-source implementation that made it available to many other users outside that company. A number of other surveys exist on MapReduce [5-7], but their goal is mainly providing a deeper understanding of the MapReduce paradigm and its software implementations or a specific use of it [6]; to the best of our knowledge this is the first survey focusing on various architectures, spanning GPGPU, hardware-software, and hardware-only ones, proposed to improve performance and efficiency of MapReduce computation.

MapReduce programming paradigm is based on concepts from functional programming and is designed to relieve the programmer from intricate details of data distribution, processing, and failure handling on the cluster; this is a major advantage compared to prior widespread parallel programming paradigms such as MPI [8]. MapReduce consists of four main phases (and two optional ones—see Section II.B) in general: distributing the data among the computing nodes in the cluster, running the *Map function* on each node to produce (key, value) pairs in parallel, shuffling these pairs to gather all pairs with the same key on a single machine, and finally running the *Reduce function* on each machine to combine values of same-key pairs into a single value. For cost efficiency, MapReduce was originally developed to run on clusters of commodity computers in WSCs, but today and future efficiency requirements necessitated by rapid increase of volumes and number of big data jobs, require more advanced features and innovation in the hardware architectures, as well as hardware-aware software improvements, for efficient processing of big data.

It is noteworthy that a number of extensions to the basic MapReduce paradigm or general big data batch processing also exist that are gaining popularity; this includes Spark [9] and also Pregel [10]. Spark is famous for its in-memory computing capability as well as other features such as support for iterative computations, execution of a flow graph of operations, and even streaming features. Pregel is specifically designed to parallelize graph processing algorithms intended to be applied on big graphs. Spark introduces Resilient Distributed Datasets (RDD) as its fundamental data structure, and Pregel defines Supersteps as its unit of computation progress where operations are performed in and on vertices of the big graph distributed on worker nodes. Consequently, the execution model, including computation as well as communication models and patterns, in such paradigms is different from the original MapReduce, and hence, they need their own specific architectural analysis and improvements. In this paper, we focus on the original MapReduce paradigm, its execution model, and proposed custom architectures for it.

The factors involved in efficient execution of MapReduce big data jobs can be broken down into the following elements: initial data distribution and management over several computers, network communication required during the shuffle phase, computation on the processing elements for Map and Reduce functions, memory footprint and memory access pattern of Map and Reduce functions on the processing nodes, and usage as well as access pattern to local storage on each node. In each, and all, of these factors improvements can be envisioned, especially when noting that special-purpose hardware can be designed and used here, even though it incurs higher costs than commodity hardware; the financial benefit as well as the demand scale is so high that justifies the higher non-recurring engineering (NRE) costs and amortizes it over multiple jobs or runs.

There are many fold other motivations for the use of hardware accelerators in today and future data centers as well. The end of Dennard scaling and the dawn of dark silicon era are serious concerns such that [11] predicts in 8nm processes more than 50% of chip area will remain unpowered, and hence, more energy-efficient approaches such as custom-made hardware accelerators are a must. Moreover, many data centers run a few dedicated workloads in large scales which consequently well lend themselves to more efficient (performance- as well as energy-wise) implementations by adding custom hardware.

We briefly review the basics of MapReduce processing paradigm in Section II.A and classify the phases of its operation in Section II.B, of which we also take advantage to identify open challenges and opportunities available for improvement.

We review architectures and proposed techniques for more efficient MapReduce computing in Section III for four major architectural choices in widespread use today: clusters of computers, multicore and many-core architectures, GPGPU-accelerated architectures, and FPGA/hardware-accelerated ones. A few of the proposed techniques are specifically designed for certain use cases and application domains. These approaches are briefly reviewed in Section IV.

MapReduce was initially introduced for use on clusters of commodity computers, but as mentioned above, dark silicon era and the ever increasing need for higher efficiency has also necessitated the move toward hardware-software implementations. As we review in this paper, most proposed techniques still need manual and labor-intensive coding and tuning for a hardware-software co-designed MapReduce processing. This imposes additional burdens to the user/programmer and distracts her from focusing on her main job: i.e., developing algorithms for processing the big data. This clarifies the need for programming frameworks and compilation/synthesis tools to seamlessly integrate hardware and software components for MapReduce processing. Currently proposed such tools and frameworks are discussed in Section V.

The surveyed custom architectures are classified into four categories in Section VI based on their scope and major optimization approaches. This helps to better understand the state of the art in this area, and more importantly, to identify gaps for further research and development.

At the end of each section, we briefly mention our view of open challenges and avenues for further research in that area. In addition, a number of emerging technologies and industrial actions are observed on the horizon that can influence the future of this track of computing paradigm. In Section VI, we provide our view of these items, along with their effects and the new opportunities as well as challenges introduced by them concerning the research community. This section also explains our view and position in research toward more efficient custom computing systems for future big data processing in the MapReduce paradigm.

Finally the paper is summarized and concluded in Section VIII.

II. Big Data Batch Processing by MapReduce

Gartner [12] defines: “Big data is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation.” In a similar definition [13], “IBM data scientists break big data into four dimensions: volume, variety, velocity and veracity”, where *volume* states that the scale of data is big, *variety* reflects that data comes in different forms, *velocity* corresponds to streaming data which is steadily produced at high speeds, and *veracity* represents the uncertainty of data. In some cases, such as index generation from documents for internet-scale search by Google or email search by Yahoo, the big data job involves batch processing of large volumes of data already collected or stored. In 2004, researchers at Google introduced MapReduce [2] to address issues of programming at datacenter-scale and Google proved it successful in practice by implementing it at scale in its huge data centers. In this section MapReduce model of computation is briefly reviewed.

A. Introduction to MapReduce

MapReduce borrows ideas from functional programming by introducing two functions: map and reduce. The map function is to produce (key, value) pairs from input data, and the reduce function is to reduce all same-key pairs into a single pair with that key and a value computed from all values of the pairs. Big data jobs in principle provide large data-level parallelism since they typically repeat a given processing on every data item input to them. Consequently, the map as well as reduce functions should in principle provide high levels of parallelism that the MapReduce implementation frameworks, such as Hadoop [4], take care of distributing it over available processing nodes, and also handle all failures automatically. The programmer is only responsible to provide the map and reduce functions; the rest is done by the MapReduce runtime.

WordCount is a famous example to show MapReduce model of computation. The map and reduce functions are [2]:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
     EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
     result += ParseInt(v);
  Emit(AsString(result));
```

The map function generates a (w, “1”) pair for each word in the given document(s), and the reduce function sums up the values for all pairs with the same word w as key. The MapReduce runtime distributes data among available compute nodes already setup in configuration files to participate in the processing, sends the map and reduce functions to appropriate nodes (in other words, the *code* is sent to where the *data* resides, instead of the conventional vice versa), transfers (key, value) intermediate data among the nodes where needed, and collects the final, possibly merged, output from the compute nodes. Next subsection describes each phase of a MapReduce computation.

B. Classification of MapReduce Operation Phases

A MapReduce job typically involves 6 phases, with two of them being optional:

1. *Initial data distribution*: In this phase, the data to be processed is distributed among processing nodes so as to benefit from data-level parallelism inherently available in the MapReduce processing paradigm.
2. *Map function*: The Map function is executed on each data block stored in each processing node. This is one of the two major data-parallel tasks involved in MapReduce paradigm.
3. *Data combine*: In this optional phase, (key, value) pairs produced on each processing node during previous phase are combined (using the same Reduce function as below) on the same machine so that each node has only one pair per key.
4. *Data shuffle*: the (key, value) pairs with the same key, but residing on different processing nodes, need to be moved to a single node so as to apply the Reduce function on them. This is done in the data shuffle phase.
5. *Reduce function*: all pairs with the same key are now on a single processing node. The Reduce function is now applied to each set so as to obtain the final set of (key, value) pairs with unique keys. Depending on the big data job in hand, the reduce phase may not be needed and can be left empty.
6. *Merge*: This final optional phase sorts the pairs based on the key, and may apply a Merge function to produce an output file representing the outcome of the big data processing without necessarily reflecting the (key, value) pairs to the user.

III. Architectures for MapReduce Processing

MapReduce was originally developed for clusters of commodity computers Google employed in their data centers, but it has since gained widespread use for other big data applications and has been ported and customized to other computing platforms. We categorize and briefly review MapReduce implementations proposed for these computing platforms in this section, especially putting more emphasis on the hardware-software platforms.

A. A Classification of Studied Architectures

We divide the architectures for MapReduce implementation into four categories as shown in Fig. 1. The conventional case is the heterogeneous clusters often found in data centers and computing farms. The second class we study comprises many-core and multicore processors now common in most desktop and laptop computers. Availability of physically shared caches and memories as well as closely coupled cores in such architectures provides opportunities for innovations that warrant a separate section to study them. The third class covers GPGPU that is obviously an appealing choice for MapReduce implementation due to the vast parallelism visible in this model of computation. Finally, taking advantage of hardware accelerators opens up a large class of other possibilities that we survey in the last class. We break this class further down by separating the architectures that accelerate solely the map function, only the reduce function, the operations commonly used in either functions, and finally full-hardware or hardware-software implementations of the entire model of computation. Below subsections provide further details per class.

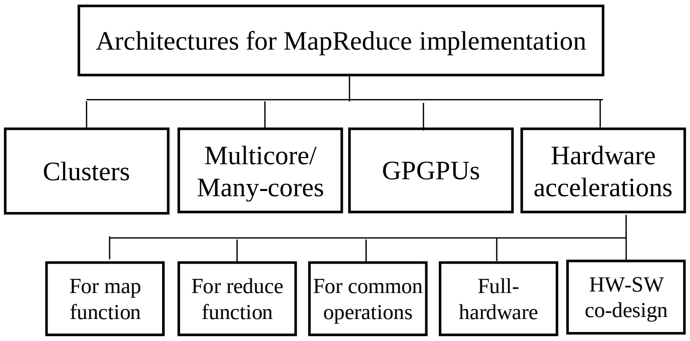


Fig. 1 A classification of architectures used to implement MapReduce.

B. Processing on Clusters

MapReduce was originally developed for clusters of computers, and hence, no surprise that its dominant use remains there. The open source Apache Hadoop [4] implementation of MapReduce played a significant role in its widespread adoption by academia and industry. Tuning various parameters influential to Hadoop performance, including hardware as well as software tuning techniques spanning BIOS, OS, JVM, and Hadoop configuration parameters [14], is a common way of improving performance on clusters. Hadoop has a large set of configuration parameters which are typically set manually, resulting in suboptimal performance. Auto-tuning these parameters [15] has an important influence on total system performance. Hadoop file system (HDFS), the distributed file system used in Hadoop implementation, is implemented in Java for portability and can be improved by trading off portability and performance [16]. Other system-level optimizations, spanning map- and reduce-task binding and scheduling such as [17], communication-aware load balancing and scheduling of map-tasks as well as predictive load-balancing of reduce-tasks on heterogeneous clusters [18], performance estimation enhancements such as [19], memory optimizations such as [20], and multi-job scheduling such as [21] are also important improvement opportunities, but remain out of scope of this paper.

C. *Processing on Multicore and Many-core processors*

There are a number of software API and runtime systems that implement MapReduce for systems other than clusters. Although these are mainly software API and runtime systems, we include them here since some of them employ architecture-specific optimizations as well as some functional characteristics of MapReduce applications that can be employed in other implementations as well.

Phoenix [22, 23] implements a runtime system and user API in C/C++ to allow the user to develop and run MapReduce computations on symmetric multiprocessors (SMP) as well as many-core or CMP architectures. The developer provides the Map and Reduce functions, and in addition, can optionally provide a *splitter* and a *partitioner* function; these functions respectively split the input data among Map tasks, and partition the intermediate data among Reduce tasks to potentially take advantage of the developer's application-specific knowledge of the distribution of values. The default splitter function of Phoenix sets the size of each data block (the unit to be processed by each Map task) such that it fits in the L1 cache. A number of further improvements are also proposed in [23] but not evaluated. This includes a user-specific prefetch engine to bring data to L2 cache in parallel with processing current ones, hints on cache replacements, and hardware compression-decompression of intermediate data. Fig. 2 visualizes the operations in Phoenix: the key data structure is the *Intermediate Buffer* in the middle of the figure where each Map task stores its produced values in its dedicated row in the column designated by a hash function applied to the corresponding key; Reduce tasks then aggregate the values column-wise and write the outcome to the *Final Buffer* data structure. This organization is designed to remove memory address conflicts among concurrent workers (i.e., cores).

Compared to sequential code, Phoenix demonstrates almost linear speedup in most cases, and even superlinear speedups in cases such as MatrixMultiply due to caching effects. The obtained speedup is comparable to that achievable by manually-tuned non-MapReduce PThread implementations [23]. Specifically, for the applications that well fit MapReduce model, e.g. word-count, MatrixMultiply and ReverseIndex, Phoenix outperforms PThread implementation.

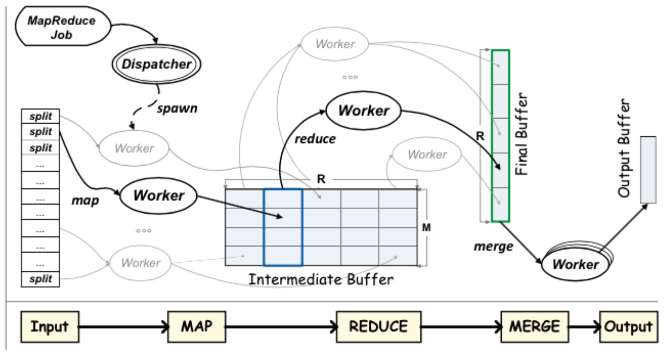


Fig. 2. Conceptual view of Phoenix implementation of MapReduce. Map tasks save their outputs to rows of Intermediate Buffer, where Reduce tasks aggregate values stored in columns. [24]

In cluster-based MapReduce implementations, first-order performance bottlenecks are typically network and storage traffic. In shared memory multi- and many-core systems, where none of these conventional bottlenecks exist, application-specific details such as key-value data storage, memory conflict, and framework overheads become primary performance limiters. To eliminate some of these inefficiencies, Tiled-MapReduce [24] executes a group of Map and Combine tasks together for lower resource usage and higher locality. MATE [25] takes a further step by proposing a modification of the MapReduce paradigm and requiring the user to provide combined Map+Reduce functions. These modifications result in up to 3.5x speedup compared to Phoenix.

To eliminate same above inefficiencies, Phoenix++ in [22] rewrites Phoenix with below major improvements which in total brings 4.7x performance boost: (i) three classes of user-selectable data structures are provided to store key-value pairs—see below, (ii) combiner is run after each Map task, (iii) final sort is optional, and (iv) calls to the Map and Combine functions are inlined by C++ tweaks for higher efficiency.

Regarding efficient data structures to store key-value pairs, an analysis of workloads is presented in [22] that can help other MapReduce implementations as well: MapReduce workloads are more deeply analyzed in [22] and are classified based on 3 characteristics as illustrated in Fig. 3:

(i) Map task to intermediate key distribution:

: case: each map task may produce any key where the number of keys is not known before execution. Word-count is a good example of such case.

*:k case: each map task may produce a key from only among k fixed known values. Histogram application represents one such case.

1:1 case: each map task produces only one unique key. Matrix multiply, where each map task produces one element of the product matrix, falls into this category.

(ii) number of values per key.

(iii) computation load per task.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2017.2736557, IEEE Transactions on Big Data

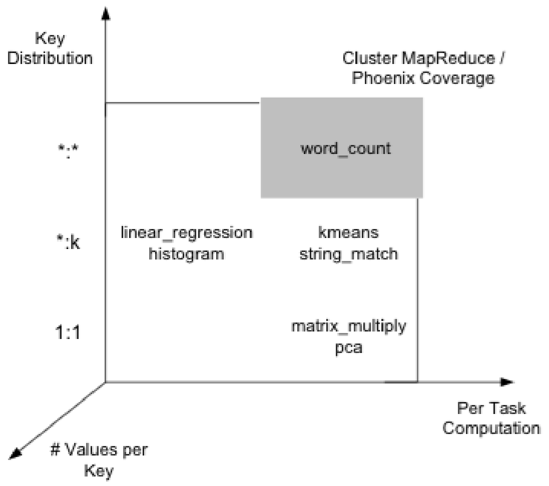


Fig. 3. Workload characteristics in terms of key-value pairs and per task computation load [22].

The authors in [26] state that the data structure used in Phoenix, a fix-sized hash table, covered the general case shown in gray in Fig. 3, but more efficient data structures can reduce hashing, resizing, and synchronization overheads. They introduce two workload-tailorable data structures and the user of the framework can choose and adapt from among these choices based on her knowledge and insight on the application: the previously fix-sized hash-table can now be resized per case, a k element vector is introduce for *:k cases, and a common array needing no synchronization among reduce tasks is introduced for 1:1 case of intermediate key distribution. Where appropriate, the user can also introduce her own storage class in C++, the language Phoenix++ is developed in. Phoenix++ and its prior versions are available online as open source software at [27].

D. Acceleration by GPGPU

Although GPGPU can also be viewed as a many-core architecture, its specific properties in terms of architecture as well as programming warrants a separate subsection to review as another platform for MapReduce processing.

MapReduce inherently provides lots of data-parallel tasks, and hence at the first glance, it seems to well fit the massive number of cores available in GPUs. The major barrier to benefit from this parallelism is the synchronization required among Reduce tasks as well as the data movements required between CPU and GPU memories.

Mars [28] flow of operations to implement MapReduce on GPU is shown in Fig. 4 where upper parts are run on CPU and lower parts on the GPU. In the *input data preparation* step, the CPU first reads the input data—since GPU has no access to disk files—and splits it into equally sized chunks, and copies them to GPU memory to be processed in parallel by GPU threads implementing Map tasks. The intermediate (key, value) pairs are optionally sorted on GPU and then splitting decisions are made on CPU—see below—for Reduce tasks. Reduce and Merge phases are then run on GPU cores, and finally output results are copied back to CPU memory. Since GPU memory allows only static allocation and non-atomic writes, the Map function is implemented in two stages to avoid access conflicts among Map tasks: in the first stage, only sizes of outputs are counted by a prefix-sum operation (*Map count* box in Fig. 4) but actual outputs are only produced in the second stage after Mars scheduler uses information of the first stage to allocate memory for intermediate data and to assign non-overlapping memories to Map tasks to be run as GPU threads. Reduce tasks are similarly run in two stages for the same reason. Although in the worst case, computations are doubled by this scheme, the overhead is actually application-dependent; e.g. in the Matrix-Multiply case, the counting stage simply returns the size needing no computation [29].

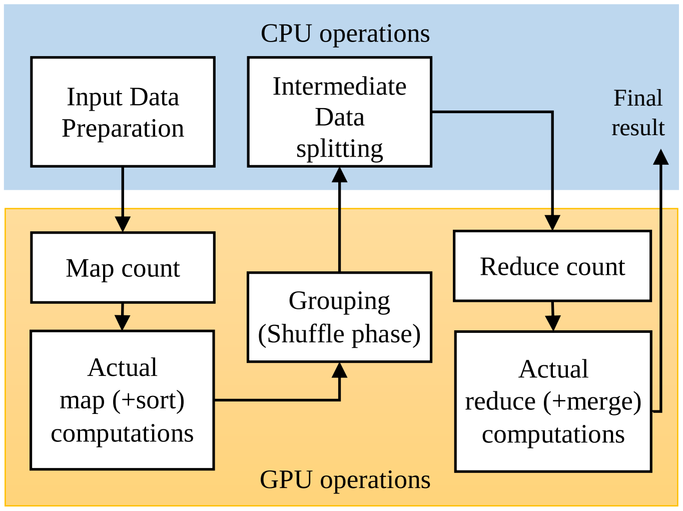


Fig. 4. Workflow of MapReduce processing in Mars [29] project.

A number of other optimizations are also applied in Mars: Different threads in a group process consecutive items of the input data at each step so that memory accesses are coalesced, and hence, memory bandwidth is better utilized. Built-in vector data types are also used instead of singular data types when accessing device memory for the same above goal. The number of threads per group is decided by iteratively using the CUDA offline calculator [30] for various numbers of threads, and is set such that GPU occupancy and registers utilization is sufficiently high. Index directories are also used so as to avoid moving (key, value) pairs around when sorting or performing other relevant operations. Specially designed string manipulation library for GPU, and hashing the keys while using a two-stage comparison technique to reduce the comparison overhead are among other optimizations reported in Mars [28]. Experimental evaluation shows up to 16x speedup for G80 GPU with 16 multiprocessors compared to Phoenix running on Intel Core2 Quad processor.

There are a number of other ports or APIs to incorporate GPUs in MapReduce [4, 31, 32]; some of them restrict the general MapReduce framework to special cases for more efficient implementation [32], or deal mainly with multi-GPU/CPU implementations rather than GPU-specific architectures which is of interest in this paper. The proposals dealing with programming frameworks for heterogeneous systems spanning CPUs, GPUs, and Hardware are discussed in Section V.

Challenges and opportunities: The two-stage tactic employed by Mars, for memory allocation and computation, is too general; ideas similar to the application classes as in Phoenix++ can be employed for more efficient data structures and allocation. Other ideas, such as combined Map+Reduce function, described in multicore/many-core MapReduce section above, can also be customized to the GPU case. Storage and network, which have traditionally been the bottlenecks of MapReduce on clusters, are not involved in MapReduce on GPU. Major bottlenecks here are memory structure and allocation, and conflicts on accessing memory by parallel tasks especially when the model of computation necessitates it, such as in the reduce and merge tasks. Efficient mechanisms to tackle above bottlenecks, in software as well as in hardware, are needed to take the best advantage from GPU massively parallel cores for MapReduce computation.

E. Acceleration by Custom Hardware

A survey of hardware acceleration methods for data centers is provided in [33]. The survey spans instruction-set extensions such as SSE and AVX, to GPUs, FPGAs, and custom-accelerators for various applications in data centers including database engines, MapReduce processing, and graph applications. More specifically, [33] classifies current major open problems for integration of accelerators to existing systems as below: (i) host-accelerator and accelerator-accelerator interconnection, (ii) memory hierarchy for the accelerator, and (iii) programming models and management of the accelerator. Automatic accelerator design from higher level languages (i.e., HLS: High Level Synthesis) is also considered important in [33] since RTL (Register Transfer Level) design is time-consuming and hard, especially for domain experts. Thus, HLS is becoming important again when many custom accelerators must be designed for a reconfigurable fabric to be used at large scale in data center. The above factors are equally important for MapReduce hardware-software implementations as well, but to further explore current practice and shed light on gaps for co-design of MapReduce model of computation, in this section we focus on the internal architectures designed for such MapReduce implementations.

Accelerating the Map function: A study on the performance bottlenecks of Hadoop implementation of MapReduce is performed in [34] on TeraSort and Grep benchmarks. The authors state that an implicit assumption in original MapReduce was the use of commodity hardware (such as hard disk and routers), and hence, their performance bottleneck was overcome by adding commodity servers in higher volumes. They continue by showing that current high speed solid-state disks and network devices are not fully utilized by today Hadoop benchmarks, and conclude that CPU has once again become the performance bottleneck and the need for accelerators is evident again. They identify and categorize the following three bottlenecks to accelerate in hardware: (i) key-value pair generation and sort by key in the Map tasks, (ii) merging sorted files in Map and Reduce tasks, and (iii) numerical calculations in Map and Reduce tasks. Their solution for the first category above, as shown in Fig. 5, is the use of a multi-core board connected to the Xeon-based host over PCIe bus; the board on the lower part of the figure receives data from the host computer by DMA over PCIe bus, dispatches them among the cores, and returns the result back to the host where final combine is done. The 36-core board achieves much better performance than the standalone host which according to the authors, its lower performance is mainly due to Java overhead since the multicore board and the host have almost the same SPECint benchmark score. In terms of energy efficiency, however, the multicore board is much better: it consumes only 35W compared to 130W by the host.

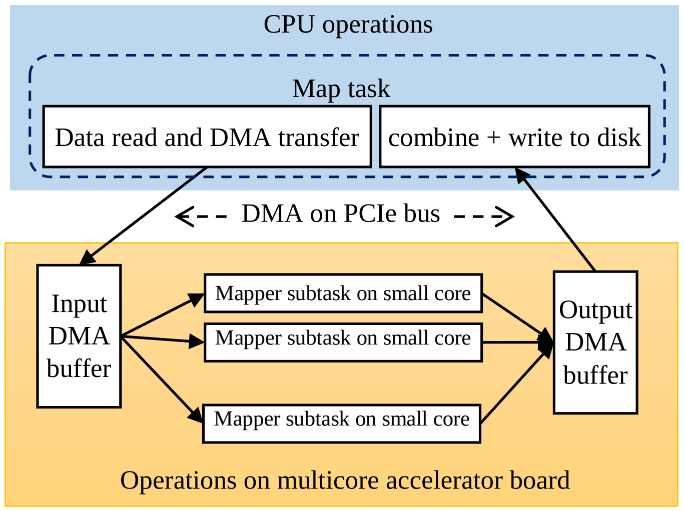

CPU operations
Map task
Data read and DMA transfer
combine + write to disk
DMA on PCIe bus
Input DMA buffer
Mapper subtask on small core
Mapper subtask on small core
Mapper subtask on small core
Output DMA buffer
Operations on multicore accelerator board

Fig. 5. Overview of the multicore-hardware accelerated system in [34].

Accelerating commonly used computations: An interesting point from [34] is that building accelerators for common application-independent operations involved in Hadoop, such as sort and merge operations, is an effective way that benefits all MapReduce jobs. Similarly, [35] implements simple Reduce functions—merely accumulation or averaging—in hardware to contribute to all jobs that use such Reduce functions.

Intel QuickAssist technology can also be used [36] to accelerate compression and decompression parts which are optional parts in Hadoop computation mainly used to reduce network and data storage overheads. QuickAssist software API redirects the compress/decompress calls to the hardware-accelerated units embedded in recent Intel processors. FPGA-based compression accelerators can also be connected over PCIe slot to Hadoop nodes for the same purpose above: to compress data before network transmissions so as to reduce network traffic, and hence, enhance performance [37]. It is noteworthy that general compression/decompression is a common solution applicable to most network-based paradigms, and not MapReduce-specific.

Accelerating the Reduce function: An accelerator for Reducers is proposed in [38] and implemented in logic blocks of Xilinx Zynq FPSoC to relieve the main processors (i.e., the ARM processors embedded in the FPGA chip) from Reduce tasks. A simplified view of the proposed structure is shown in Fig. 6; the upper part of the figure shows processor cores that perform the map functions and send the (key, value) pairs to the lower part where the reduce function is implemented in hardware on the FPGA logic blocks. A memory structure for efficient key-value storage and key-search is proposed [35] and simple Reduce functions (i.e. accumulation and averaging) are implemented in hardware—see the small gray boxes in the figure. As the figure shows, current value for each processed key is stored in the internal data structure. Two hardware-implemented hash chains are used to find the corresponding row for each key. Then, the stored key is compared to the new key to see if there is a hit; if neither of the hash chains results in a hit, the new key is stored in a queue (not shown in the figure). In case of a hit, the newly received value is combined with the previous one by the gray adder boxes (i.e. the hardware-implemented reduce function) and are stored back in the data structure. The number and the size of keys storage can be configured based on the application needs. Speedups up to 2.3x are reported [38]. In another work [39] on the same thread, hardware implementation of map tasks is also generated using HLS [40] and thus the whole system is implemented in hardware, reporting 2 orders of magnitude energy reduction compared to CPU-only implementation. Note that [39] belongs to next category below (all-hardware) and is mentioned here to illustrate the evolution of the authors' works.

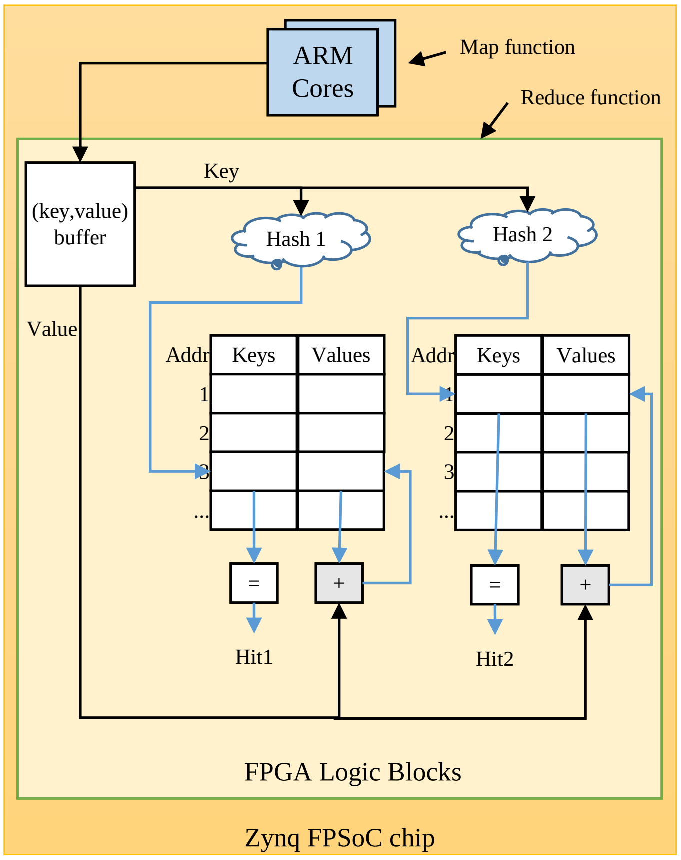


Fig. 6. A simplified view of the accelerator block diagram for simple Reduce functions proposed in [38].

All-hardware MapReduce: FPMR [41] provides a general architecture on FPGA that realizes an almost complete MapReduce system in hardware: the architecture contains mapper and reducer modules as well as on-chip scheduler to assign tasks to the mappers and reducers, and also a data controller module to manage host-FPGA data transmission as well as passing initial data to the mappers, and storing final data from reducers (or possibly the merger whenever required). FPGA board is connected over a PCIe bus to the host that implements less time consuming tasks. Their implementation of RankBoost data mining algorithm achieves above 31x speedup against a software implementation. A major goal of the design is generality and reusability for other algorithms, and this is reflected in the application-independent design of FPMR architecture shown in Fig. 7. Replicated hardware modules are added for mapper and reducer tasks, shown in the middle lower part of the FPGA box in the figure. The *Data Controller* and *Processor Scheduler* boxes manage proper data dispatching and module activation of mapper and reducer modules. Local memory on the FPGA is used for storing intermediate key-value pairs. The user must design the mapper and reducer hardware internals, and may need to improve memory structure per application needs (in case of RankBoost implementation in [41], this includes: dual global memory banks, double buffering, and also a mechanism to reduce bandwidth pressure by avoiding redundant data transmission). Parallel mappers and reducers contribute to the decrease in execution time, but pipelining is not employed.

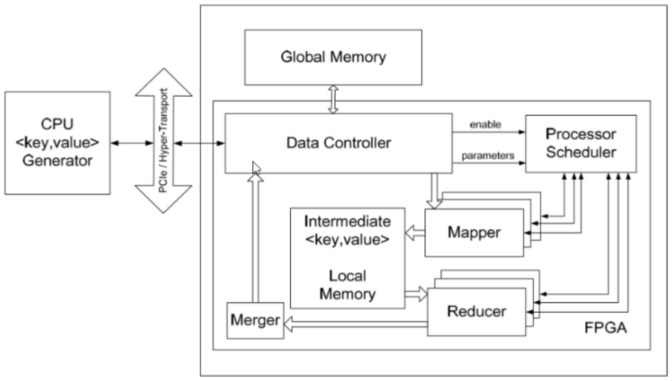


Fig. 7. FPMR architecture to implement on FPGA [41]

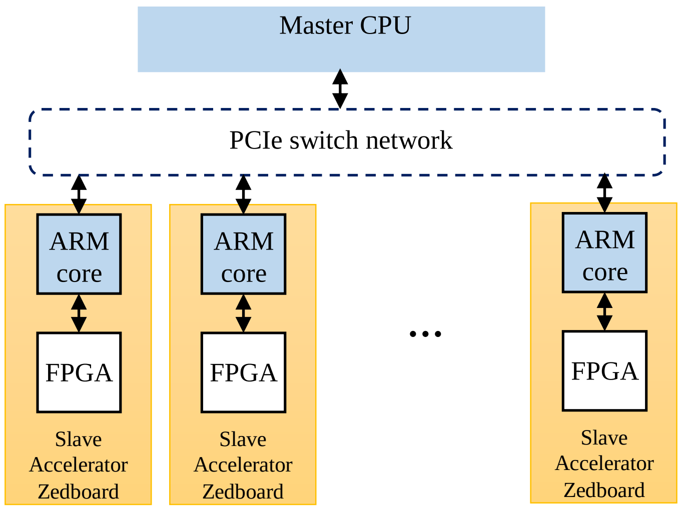
The work in [39] is another work in this category that we discussed in previous section along with other works of the authors—see Fig. 6 and its corresponding text.

A more recent all-hardware MapReduce implementation is Melia [42] where OpenCL is used as a vehicle to implement user-provided map and reduce functions directly on FPGA. This relieves the end-user from intricate details of FPGA hardware design and hardware description languages, and consequently, increases her productivity by allowing her to focus on describing the target computation in the familiar MapReduce paradigm. Furthermore, [42] demonstrates nearly 4 times higher energy efficiency (performance per Watt) for FPGA implementation of MapReduce over CPU and GPU implementations.

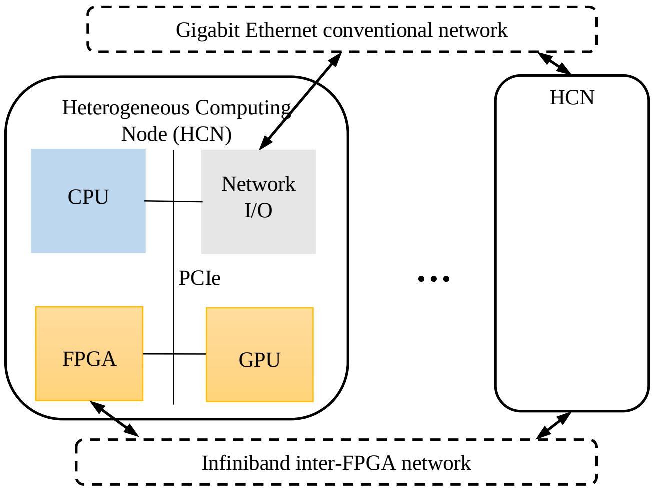
Melia is provided as a software library that the designer uses, along with her map and reduce functions in C, to describe her desired functionality. Then, implementation parameters such as local (on-FPGA) memory usage, loop unrolling, and map/reduce pipeline replications are determined by a number of guidelines and calculations. Finally, the design is compiled and executed on the FPGA using OpenCL toolkit. The map and reduce phases are non-overlapping, which suggests either partial reconfiguration to occupy less FPGA space, or multi-job MapReduce computation to increase the utilization of FPGA resources, although none of them are implemented in [42].

Raising the abstraction level of system model usually results in lower performance of the system implementation. To mitigate this, [42] applies a number of FPGA-specific optimizations: Memory optimizations include memory coalescing and use of private memories, while computation optimizations include loop unrolling and pipeline replication. As described above, Melia framework helps the designer to tune above parameters, and [42] reports up to nearly 44x performance boost over the non-optimized baseline FPGA design obtained by these tunings and optimizations.

Hardware-software co-designed MapReduce: A co-design approach is proposed in [43] for acceleration as well as for energy efficiency of MapReduce jobs by implementing only the time-consuming parts of the application in hardware. As Fig. 8 shows, a number of accelerators (Xilinx Zynq field-programmable system-on-chip, or FPSoC, boards) are connected as slaves over a PCIe bus to an x86 processor (Xeon and Atom processors) as the master node. Four data mining benchmarks, namely K-means, KNN, SVM, and Naïve Bayes, are profiled to identify time-consuming functions which are then synthesized into hardware by Xilinx Vivado HLS tool (we are confused whether the profiled applications were originally merely core functions or fully implemented MapReduce applications in C, since the paper mentions C-based source codes [43]). The system is assumed to send the big data from the master node to the slave boards over PCIe and collect the results back; slave nodes are to process time-consuming functions of the algorithm in hardware on FPGA blocks. Analytical models are then developed based on Amdahl's law as well as the overheads corresponding to all the communication links in Fig. 8 so as to estimate potential speedups if the co-designed system were implemented in practice. Estimated end-to-end speedups up to 2.7x, total power ratios of above 2x, and EDP ratios of above 15x, are reported [43]. Changing the master node from Xeon to Atom processor obviously affects power and EDP ratios but not the achievable speedup. Although the case studies are all data mining benchmarks, but the framework is general enough to be fairly easily adapted to other applications as well. The HLS tool provides some pipelining as well but its effect, and possible improvements of it, are not analyzed in depth and are potential paths for enhancement. The speedup is mainly obtained by hardware implementation of software functions, and replicating it as multiple workers.


Master CPU
PCIe switch network
ARM core
FPGA
Slave Accelerator Zedboard
ARM core
FPGA
Slave Accelerator Zedboard
...
ARM core
FPGA
Slave Accelerator Zedboard
Fig. 8. HW-SW co-design architecture in [43] based on FPSoC boards.

Axel [44] is another heterogeneous platform encompassing CPU, GPU, and FPGA. Authors in [44] demonstrate the use of this platform to accelerate MapReduce processing. The Axel cluster is a Non-uniform-Node Uniform-System (NNUS) heterogeneous system as shown in Fig. 9; each node consists of a processor, a GPU and an FPGA along with their corresponding memories (not shown in the figure) and a shared PCIe bus and a network I/O system for conventional inter-node communication over Gigabit Ethernet. Intra-node communication is provided by the PCIe bus, whereas two inter-node communication media are provided: general Gigabit Ethernet for conventional networking, as well as Infiniband high-speed links through FPGAs for direct higher speed FPGA-to-FPGA communication.


Gigabit Ethernet conventional network
Heterogeneous Computing Node (HCN)
CPU
Network I/O
PCIe
FPGA
GPU
...
HCN
Infiniband inter-FPGA network
Fig. 9. NNUS architecture of Axel [44] heterogeneous cluster.

For MapReduce processing, two implementation paradigms are envisioned as shown in Fig. 10 and Fig. 11. The former uses FPGAs for the Reduce functions where high-speed low-latency direct link among FPGAs are used, whereas the latter employs lower-speed CPUs and the Ethernet channels among them. The framework provides a general flow for employing the heterogeneous cluster, but details of GPU and FPGA implementations are left to the user to provide. The authors use the latter above approach for an N-body simulation and report 22.7x down to 4.4x speedups respectively for 1-node and 16-node clusters; they put the non-linear speedup and its degradation to the all-to-all communication needed due to the nature of the N-body simulation application.

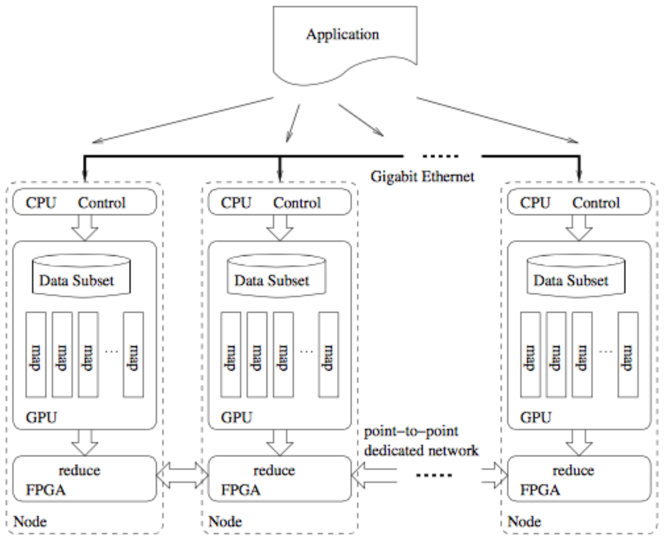


Fig. 10. Reduce-by-FPGA paradigm in Axel [44].

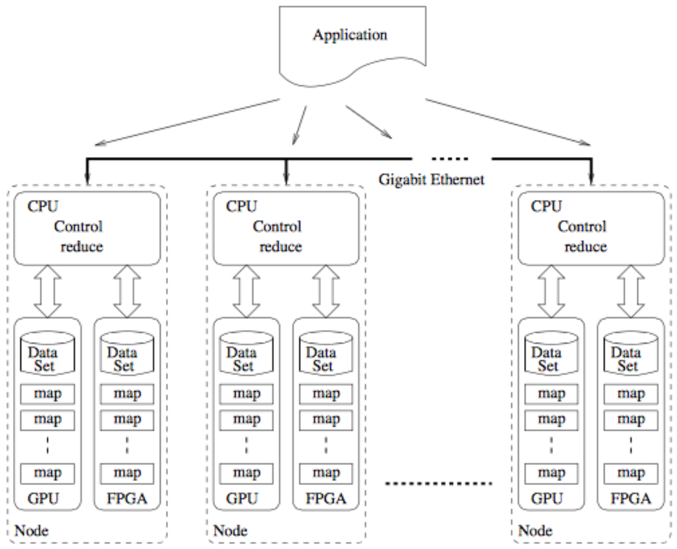


Fig. 11. Reduce-by-CPU paradigm in Axel [44].

Authors in [45] provide MapReduce API in C and use the architecture in Fig. 12 to distribute data among worker nodes implemented in FPGA, GPU, or even conventional CPU. This architecture is viewed as a custom computing machine for the MapReduce paradigm. The *scheduler* distributes data among worker nodes which can be FPGA, GPU, or even multicore boards. For each worker node, e.g. for the FPGA one, a *data distributor* distributes its assigned data among instances of the Map function on the FPGA, and returns the Reduce function outputs back to the system, all in a pipelined manner. For the FPGA part of this hardware-software MapReduce system, the authors manually (claimed to be easily automatable) convert the code to pipelined Handle-C code which is then synthesized into FPGA bitstream. Speedups up to 30x are reported for simple benchmarks and 16 pipeline instances implemented in the FPGA.

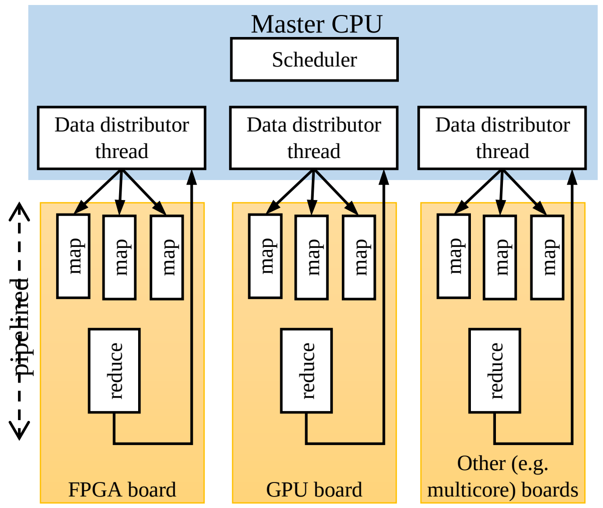


Fig. 12. Architecture of the co-designed MapReduce custom computing machine in [45].

A number of other proposals also exist for co-designed MapReduce with goals other than sole performance and power. Authors in [46] integrate FPGA with commodity hardware (processor) and reconfigure the FPGA upon failures for fault-tolerance while the data is redirected to commodity hardware during FPGA unavailability. Advantages of using OpenCL framework is demonstrated in design showcases in [47] where OpenCL is used to describe a variety of computation structures, including static as well as streaming pipeline of MapReduce computation, which is then implemented on FPGA boards connected to a host micro-server with Atom processor. Large power savings, in addition to performance gains, are key achievements demonstrated in practice in this work [47].

Challenges and opportunities: Although pipelining has been used in a few of researches we reviewed above, there is still room to better utilize its advantages especially since the massive identical computations in the map and reduce functions well lend themselves to pipelined execution and the high throughput it can provide. Data distribution and intermediate-data shuffle phases are the other parts that need better hardware support and application-specific architectures. Reconfiguration capabilities of FPGAs is also not well taken advantage of up to now. For example, imagine an architecture with multiple pipelines for map and reduce functions where the input data and the intermediate ones are respectively streamed to each map and reduce pipeline; the number of map and reduce pipelines can be dynamically adjusted, by FPGA (partial) reconfiguration, upon production of intermediate data where the number of pairs per key is partially known. Recent move toward closely coupled FPGA-CPU chips—see Section VII.A—can make room for novel co-designed proposals with higher communication efficiency and easier computation migration, both online as well as offline, between the CPU and the FPGA.

F. *Summary of Custom Architectures*

TABLE 1 summarizes major features of the custom architectures surveyed above. The stages of MapReduce that are the main focus of each work for acceleration, are given in the second column of the table. Third column gives the target platform of the work, and the fourth column gives more details of it in terms of number and type of the considered nodes. The interface employed between the accelerator and the CPU is given in the next column, and finally the last column describes the language(s) used to model the function implemented in the accelerator. Where important, the high level language (e.g. C) used by the programmer, as well as the hardware description language (e.g. Handle-C) used for FPGA design are both mentioned for emphasis.

Lack of a commonly approved benchmark suit and comparison environment is a major obstacle to quantitatively compare the approaches and their outcomes, and hence, one has to suffice to subjective descriptions and comparisons. Major conclusions from the above survey and the summary provided in below table are discussed in *Challenges and Opportunities* subsections above. A further classification of our envisioned approaches are given in Section VI; it provides a big picture of current state of the art, and helps identify missing parts for further work.

TABLE 1- A Summarized Comparison of Major Hardware-Acceleration Techniques for MapReduce.

Proposal	Phases accelerated	Platform	Single/ multi-node	CPU-accelerator interface	Programm-ing model
[34]	map	server + multicore board	Single node	PCIe	Java
[35]	common functions for reduce	FPSoC	Single FPSoC	On-chip ARM AXI bus	HDL
[36]	compress-decomp.	intel new processors	Multi-node	internal to core	Java
[37]	compress-decomp.	server + FPGA	Single node	PCIe	HDL
[38]	reduce	FPSoC	Single FPSoC	On-chip ARM AXI bus	HDL
[39]	all	FPSoC	Single FPSoC	On-chip ARM AXI bus	HLS
[41]	(virtually) all	server + FPGA	Single node	PCIe	HDL
[42]	all	FPGA	Single FPGA	Custom on-chip	C, OpenCL
[43]	benchmark-dependent	server + FPSoC	Multi-FPSoC	PCIe	HLS
[44]	map or reduce	server + GPU+FPGA	Multi-node	Ethernet + Infiniband	HDL
[45]	(virtually) all	server + GPU+FPGA	Multi-node	Custom	C, HDL (Handle-C)
[47]	all	FPGA	Single node	Custom	OpenCL

IV. Application-Specific Architectures for MapReduce Big Data Processing

A number of other architectures focus on a specific application or application domain and propose acceleration techniques based on characteristics of those applications. Reference [33] briefly reviews a number of such proposals for general data center workloads. We review application-specific accelerators for MapReduce workloads in this section.

A. *Simple/Core Functions*

An FIR filter in MapReduce is implemented in [48] where Xilinx Zynq-based ZedBoards are used as slave nodes connected by an Ethernet switch to a Core-i5 master node. The Zynq FPGA on ZedBoard integrates dual ARM Cortex-A9 processors clocked at 667MHz with Artix-7 FPGA blocks. The ARM processors run Xillinux OS on which Hadoop framework is run, and the FPGA blocks realize hardware implementation of the FIR filter in a pipelined design running at 100MHz. They report speedup of over 3.3x compared to a conventional (non-MapReduce) optimized all-software FIR implementation on a single ARM processor, and 20% higher speed compared to the same ARM-based Hadoop cluster without the hardware accelerators.

B. *Graph Applications*

While there are proposals for designing hardware templates for general graph applications [33], as well as MapReduce implementation of graph processing algorithms such as [49], we are unaware of hardware implementation of MapReduce applications of graph processing algorithms. Given wide usage of graph algorithms in largescale big data processing, e.g. in social network analyses, acceleration by hardware for such applications, as well as related models of computation such as Pregel [10], seems a promising path to follow for further research.

C. *Data Mining Applications*

Authors in [50] connect multiple computers each equipped with an FPGA board for faster computation of k-means algorithm. Each iteration of k-means is implemented as a MapReduce job where distance calculation of data points to current centroids are done by the Map tasks, and calculation of new centroids is left to the Reduce tasks. Each FPGA board is connected to its computer via PCIe slot, and all boards are also directly connected by Ethernet switches as in Fig. 13. A key point here is the use of *inter-FPGA network,* as shown in the figure, for faster direct communication among FPGAs to eliminate the overhead of communication through CPUs and general purpose local network. The host CPU in each compute node merely configures the FPGA and marshals the data to/from it. DMA is used to transfer data to/from each FPGA where either the map or the reduce function is implemented; the (key, value) pairs are passed from mapper FPGAs to reducer FPGA by the Ethernet switch. They report to obtain speedups up to over 16x compared to optimized Mahout software-only implementation. The bottleneck was host-to-FPGA communication channel as per [50].

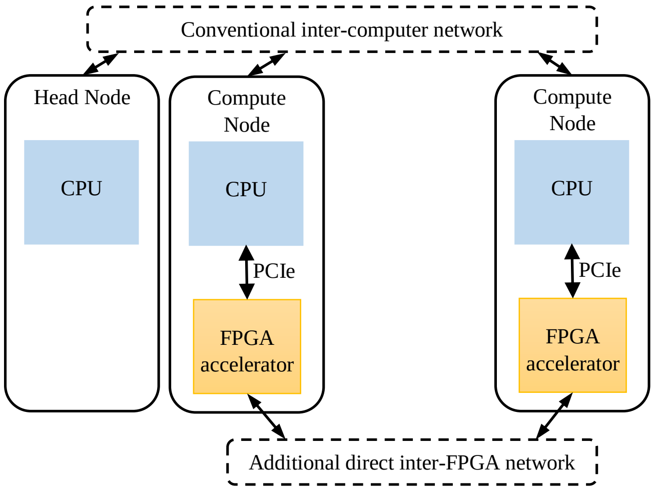


Fig. 13. Overview of the accelerator-powered cluster in [50].

FPMR [41] is the other work that uses FPGA to improve or analyze possible speedup of data mining MapReduce applications (RankBoost, SVM, and PageRank)—see Section III.E for further details. It is noteworthy that although all the considered benchmarks are data mining applications, the proposed structure is general and nothing is application-specific.

Co-design architecture of [43] is also analyzed for a number of data mining benchmarks, but again no specific property of such applications is taken advantage of, and the structure is applicable to other general big data applications as well—see Section III.E and Fig. 8.

D. Genomics

Many big data applications exist in the field of genomics which could potentially benefit from MapReduce and accelerator implementation. Next generation genome sequencing with short or long reads is among such well known applications. Authors in [51] distribute the reads among computing nodes each responsible to identify potential places in the reference genome to align the read. As the Map function, the computing nodes use FPGA accelerators for the string matching problem, and produce (key, value) pairs each of which represents the index of one occurrence of the read in the reference genome. The pairs are shuffled as usual and the Reduce tasks combine them to write all occurrences of the read in an HBase table which is then processed to produce the final sequenced genome. Their experimental results report 2.73x speedup for each hardware accelerator occupying less than 1% of a Virtex-5 LX110T FPGA.

Challenges and opportunities: Ever increasing significance of application domains such as Genomics, makes increasingly more demand for higher efficiency at lower cost and power. Widening use of MapReduce model of computation in such application domains shall make more room for innovation and specialization toward above goals. There is a large design space to explore here for application-specific improvements spanning memory, processing, and communication architectures especially on modern CPU-FPGA architecture recently on rise.

V. Frameworks for Programming at Scale

Data centers when viewed as a WSC [3] are large computing systems with enormous computing capacity, but taking best advantage of this capacity, i.e. programming them, is a daunting task. MapReduce is one of the paradigms that helps here, but when hardware accelerators are added to the WSC for higher efficiency, new programming tools and frameworks are needed to complement current MapReduce frameworks such as Hadoop. This subsection reviews some of currently proposed or envisioned frameworks incorporating hardware accelerators.

Blaze [52] provides FPGA-as-a-Service (FaaS) concept comprising a programming interface and a runtime system to allow datacenter-scale deployment and use of FPGA accelerators with minimal programming effort by big data programmers. FaaS programming interface essentially hides the FPGA accelerators, and their intricate details of programming and usage, behind a set of well-defined programming API that enables easy use of accelerators by big data analytics developers. The Blaze runtime system then takes care of (i) sharing the accelerators among multiple jobs and applications, (ii) covering multiple accelerators installed on multiple nodes under the same FaaS umbrella, (iii) scheduling accelerator functions on available FPGAs along with other related details such as (re-)programming the device and marshaling input and output data, and (iv) other advantages such as providing fault tolerance and hiding the latencies by pipelining and caching. The hardware design for the FPGA, however, still needs to be provided by an expert HDL designer which takes several weeks in their experiments [52]; automating this task is left to other and future work [52].

OpenACC (for Open Accelerators) [53] is a programming standard for parallel processing on CPU/GPU systems. The standard provides directives that the programmer can embed in her code to help the compiler derive efficient parallel code to run on GPU and/or multi-core CPU. It basically provides a mechanism for the programmer to transfer her knowledge of the field and application to the compiler. Efficient use of this information is up to the compiler and libraries usually developed by device/CPU vendors such as Nvidia. OpenACC framework, however, cannot run parallel code such as MapReduce-style code simultaneously on CPU and GPU [54]. Frameworks such as Panda [54] provide the capability to simultaneously distribute Map as well as Reduce tasks among CPUs and GPUs. Fig. 14 shows the flow of operations in Padna with its two-stage scheduling: scheduling tasks among compute nodes comprising CPUs and GPUs, and then within each CPU or GPU. The blue boxes in the figure are run on the CPU and the orange ones on the GPU. Data partitioning and task scheduling is done statically on the CPU, then the tasks are run in parallel on CPU and GPU. Intermediate data are also copied back from GPU memory to CPU memory for the shuffle stage. The Map and Reduce functions can be different for CPU and GPU, and must be provided by the programmer. Panda framework is available as an open source project at [31].

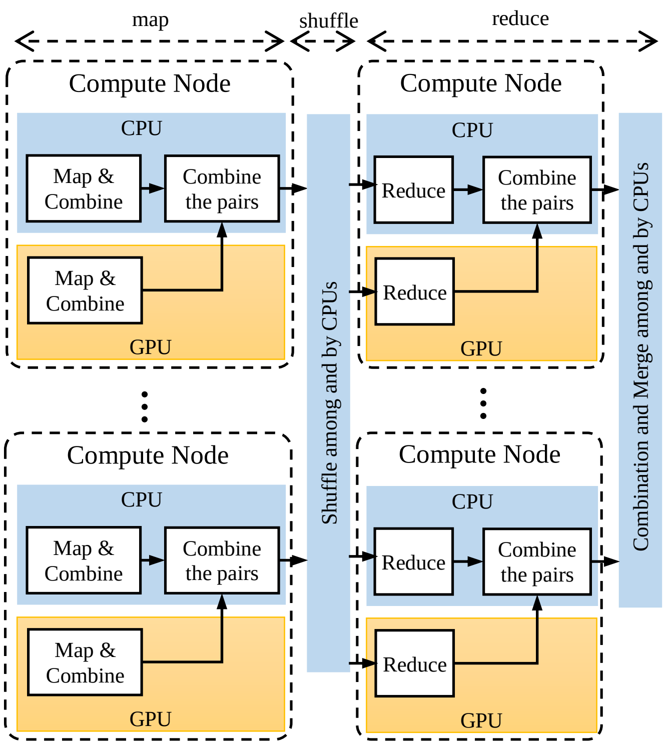


Fig. 14. Panda [54] flow of operations on CPU and GPU.

HLS is used in [40] to produce Mapper as well as Reducer hardware from original C/C++ code of MapReduce applications originally developed to run on multicore frameworks such as Phoenix. Up to 4.3x throughput enhancement as well as two orders of magnitude power and energy improvement are reported compared to multicore implementation on an AMD 8-core processor [40]. The proposed extension (top-right of Fig. 15) to Vivado HLS flow is shown in Fig. 15. The extensions include directives and source code annotations to guide Vivado HLS flow in generating synthesizable HDL from the given MapReduce application. The rest of the flow (top-left and bottom of the figure) is standard Vivado HLS flow providing simulation as well as synthesis and bitstream generation.

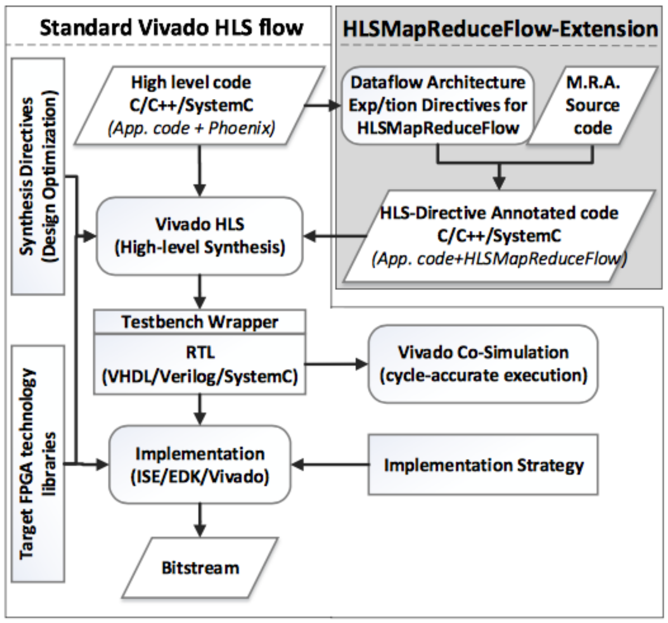

Standard Vivado HLS flow
HLSMapReduceFlow-Extension
Synthesis Directives (Design Optimization)
High level code C/C++/SystemC (App. code + Phoenix)
Dataflow Architecture Exp/tion Directives for HLSMapReduceFlow
M.R.A. Source code
Vivado HLS (High-level Synthesis)
HLS-Directive Annotated code C/C++/SystemC (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. 15. Proposed HLS flow for MapReduce accelerator generation [40]. (In the figure, Exp/tion stands for Exploration, and M.R.A.is an acronym for MapReduce Application)

VINEYARD [55] is a conceptual high-level framework aimed at providing an integrated workflow for hardware-software implementation of big data applications in data centers equipped with heterogeneous computing nodes spanning general purpose processors, FPGAs, embedded processors such as ARM, and full-custom application-specific dataflow engines. As shown in Fig. 16, the programmer still uses familiar programming models for big data processing such as MapReduce, Spark, and Storm, and then the VINEYARD tools and compilation/synthesis framework (middle of the figure) uses library of IP blocks to efficiently implement the user application on the data center resources. Commonly used functions are envisioned to be already synthesized and stored in a *Repository* of IP blocks, shown on the right side of the figure. The hardware implementation (DFE: Dataflow Engine) or hardware-software implementation (ARM processor together with programmable logic) of such IP blocks would have been deployed in the envisioned datacenter and the VINEYARD runtime scheduler would choose from among them.

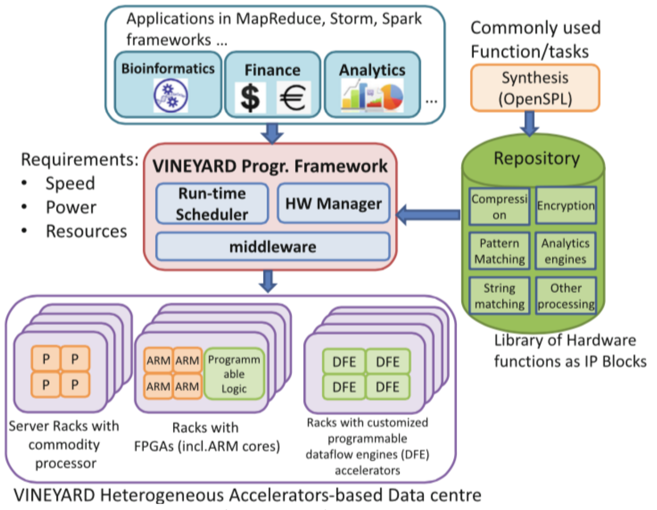

Applications in MapReduce, Storm, Spark frameworks ...
Bioinformatics
Finance
Analytics
Commonly used Function/tasks
Synthesis (OpenSPL)
Requirements:
Speed
Power
Resources
VINEYARD Progr. Framework
Run-time Scheduler
HW Manager
middleware
Repository
Compression
Encryption
Pattern Matching
Analytics engines
String matching
Other processing
Library of Hardware functions as IP Blocks
P P P P
Server Racks with commodity processor
ARM ARM ARM ARM
Programmable Logic
Racks with FPGAs (incl.ARM cores)
DFE DFE DFE DFE
Racks with customized programmable dataflow engines (DFE) accelerators
VINEYARD Heterogeneous Accelerators-based Data centre

Fig. 16. VINEYARD framework for accelerator-rich data centers [55].

Other proposals also exist that use OpenCL for describing pipelines for static as well as streaming processing of data blocks in MapReduce, and use EDA tools to produce FPGA bitstreams from the OpenCL code [47], and Handle-C and associated tools to describe a pipelined C code in MapReduce model and produce pipelined FPGA implementation as in Fig. 12 [45]. Melia [42] also uses OpenCL as an intermediate language to enable end-users describe their applications in MapReduce paradigm, and also write their map and reduce functions in C. Melia also provides guidelines and software libraries so that the application is compiled and directly run on the FPGA so as to achieve easier programming of the FPGA as well as higher energy efficiency.

Challenges and opportunities: While it is widely admitted that exposing underlying hardware details to the programmer can potentially result in more efficient implementations, c.f. assembly language vs. high-level programming, it definitely distracts the programmer from her main job of processing the big data, and is less likely to find widespread use. Future frameworks should seamlessly integrate with big data processing frameworks, such as Hadoop, so that the programmer does not need to get involved with intricate details of using hardware accelerators. Such high-level frameworks capable of hiding underlying details from the programmer, while at the same time producing efficient mapping and implementation on heterogeneous accelerators as well as general-purpose computers, are essentially required for the success of future accelerators at scale. A number of such frameworks have been proposed as described above and the large body of research on hardware-software co-design can be revisited here, but such tools are still in their infancy and need deeper further research and development.

VI. A Classification of Architectural Approaches

In this section we provide our view of possible approaches to MapReduce implementation using custom architectures. This not only classifies currently proposed ones, but also helps to identify gaps and avenues for further research.

A. *Custom architectures for the computation steps*

This is the most general case which covers all architectures presented in Sections III.D and III.E. In such approaches, the computation model (i.e., MapReduce) is analyzed and a custom organization is proposed to connect processing elements spanning CPUs, GPUs, and FPGAs. The proposed custom architecture may provide acceleration for all (e.g. FPMR [41]) or parts (e.g. [38]) of the computation steps. Nevertheless, although all such architectures are custom to the computation model of MapReduce, they are designed to be general and applicable to all MapReduce applications.

Most of the proposed custom MapReduce architectures fall into this category as surveyed in the above sections. Consequently, this category is mature and one needs to dig further deeper to propose new methods.

B. *Architectures customized to application characteristics*

The other class we envision, applies application-specific features to the proposed architecture such that it works more efficiently for that specific class of applications. Phoenix [26] is a good example of this class—see Section III.C and Fig. 3. In such approaches, specific features of the application are taken into consideration, and the architecture is customized such that the best advantage is taken for that certain application or class of applications.

Although Section IV reviewed a number of proposals, most of them lack proper customization to the application needs. Thus, we found only a few truly *application-specific* architectures in the survey we provided above. Thus, we believe more opportunities exist in this class for exploration and novel proposals. Hot applications such as graph processing, machine learning, machine vision, and Genomics are among good candidates to target in this class.

C. *Architectures customized to data characteristics*

As the third class of custom architectures, we propose to customize the architectures to the features of the specific data items being processed. Our experiments [56] show that different sources of data, have different characteristics and moreover, have different influences on the final outcome of the processing. Thus, for instance, one could add more parallelism into the architecture for more influential data blocks, or could use lower power architectures for the others. Similarly, custom precision in calculations could be used accordingly. Such customizations should obviously involve dynamic adaptation of the architecture since the data is only known at runtime, and hence, *re-configurable architectures* are a good fit for such optimizations.

Our survey in this paper found no such custom architecture in the literature, and hence, this area looks unexplored and potentially very productive for further research.

D. *Other criteria for further classification*

Another criterion that can be imagined to further classify each of above classes is static architectures vs. dynamically adaptive ones. The conventional view of hardware architectures is a static organization of modules. However, modern FPGAs provide features such as runtime-reconfiguration as well as partial reconfiguration that are barely explored for custom MapReduce architectures. These features can be applied to any of the above three classes to come up with novel adaptive architectures. For example, in the first class one could imagine FPGA architectures that initially use the entire FPGA for the map function, and then reconfigures the FPGA for the shuffle phase and/or reduce function. Adaptive architectures for the second and third classes above are easier to imagine.

Conventionally, PCIe slots are used to connect FPGA boards to the high performance CPUs of datacenter servers, providing a loosely coupled processor-FPGA system. This makes PCIe a serious performance bottleneck in many cases. Recent industrial advances in more closely integrating these two processing elements—see Section VII.A—can provide a dramatic increase in processor-FPGA communication bandwidth and pave the way to proposing novel architectures in any of the above three classes in this new closely-coupled processor-FPGA platform.

VII. Trends in Large Scale Computing

A number of market and technology movements are observed at the moment that can potentially change the landscape of *computing at scale* in near future. We list a number of them at this subsection and briefly describe what we envision as their effects and outcomes. Each below topic basically provides an opportunity for MapReduce application developers to improve efficiency of their applications by taking advantage of the new technology, but at the same time is a challenge for them since the developer now needs to choose from among the new choices enabled by the technology. This challenge, however, provides another opportunity in the design-automation field for the corresponding researchers to work on optimizations by automatic exploration of the design space enabled by these choices. We provide a brief view of these challenges and opportunities in each section below.

A. Closely Coupled FPGA-Multicore

Intel recently acquired one of the two biggest FPGA companies, Altera [57]. Although not much is revealed on future plans, but it is easy to imagine that Intel who had abandoned its own field-programmable technology sector over a decade ago, will soon be releasing its next generation multicore processors equipped with closely-coupled FPGA blocks so as to enable programmers to gain higher efficiency by (potentially dynamically) migrating suitable computations to the hardware. Current host-accelerator communication bottlenecks imposed by limitations of interconnection standards such as PCIe and others will be removed, and software API such as QuickAssist will boost programmers' productivity and bring ease of integration of hardware into the development processes. Xilinx is also moving toward the same goal, but from the other end of the spectrum by integrating multicore processors, such as ARM Cortex A9 cores, into its FPGAs such as its Zynq line of products. Xilinx Vivado HLS tool helps to hide hardware-software integration details to some extent, but further productivity boosters are still missing and needed. Other recent CPU-FPGA platforms provide various micro-architectural features that affect achievable performance and energy efficiency in co-designed applications, and this landscape of choices is further expanding by introduction of more platforms from a variety of vendors. A quantitative comparison of choices and features of such modern CPU-FPGA platforms is provided in [58] providing guidelines for platform users as well as designers.

Challenges and opportunities. In terms of new opportunities, such closely-coupled FPGA-multicore platforms shall open up a wider design space for designers to explore and to take advantage of static as well as dynamic computation offloading, and furthermore, design more efficient co-designed architectures for widely, or extensively although not widely, used big data applications.

The challenges can be divided into two categories: hardware-software integrated development environments (IDE), and automatic co-design optimization tools. Providing easy to use software development environments that hide intricate details of underlying heterogeneous hardware from the application developer, and automatically generate hardware and software units and the hardware-software interfaces, is a daunting task and big challenge for success of such closely coupled FPGA-multicore platforms. The model of computation these environments support is an important choice. Furthermore, such environments need to be complemented with design space exploration tools that evaluate various architectural as well as system-level choices. In case of MapReduce model of computation, such choices include, but are not limited to, hardware vs. software implementation of each map/shuffle/reduce/merge task, the number of parallel processing elements for each task, hardware-software interface mechanisms, and dynamic reconfiguration alternatives (both total- as well as partial-reconfiguration).

B. Memory Technologies and Memory-Centric System Design

Various non-volatile memory technologies are under research and development that are expected to gain more popularity and financial viability in near future. On the other hand, memory-access behavior of big data applications and programming models such as MapReduce, meaningfully differ from conventional server and service workloads [59]. Moreover, big data applications, by definition, deal with large amounts of data; this has motivated MapReduce to actually send the *computation* to where the *data* resides, and not the other way around as traditionally done in distributed processing.

Challenges and opportunities. Many new memory technologies, such as PCM, STT, and others, are non-volatile memories (NVRAM) and consume much less power than conventional DRAM and SRAM, but in terms of speed, throughput, and power, they show asymmetric behavior for read vs. write, which although makes them harder to use, but opens up new opportunities for innovation and research. New memory hierarchy designs are needed to efficiently combine SRAM, DRAM, and such NVRAM technologies while architecting the computation nodes around the memory system as motivated above, leading to a memory-centric design. Such approaches are already under development by academic [60] as well as industrial [61] researchers.

As briefed above, new opportunities for various memory optimizations are enabled by these new technologies. For hardware researchers interested in custom architectures for MapReduce implementation, this means design of custom memory-centric system organizations that reduce amount of data movement among memories, and multiplex the data among processing elements each of which implements one (or more) phase of MapReduce computation. The memory hierarchy can also be a hybrid system of SRAM, DRAM, and NVRAM tailored to the memory access pattern of MapReduce applications. Another challenge for software and algorithm researchers will be on design automation to properly map data blocks, variables, and codes of the application to this memory hierarchy such that efficiency metrics such as performance and power are improved. Co-optimization of memory access patterns (e.g. by scheduling of map/reduce tasks or by data placement) and memory organization in a reconfigurable system is another challenging research area in the design automation field.

C. *Software-Defined Architectures*

While software defined control at microarchitecture level, c.f. microprogramming, is debatably inefficient, it has proved successful at large scales. Software Defined Network (SDN) is a good example here, and similar proposals and activities exist for Software-Defined Data Center (SDDC) and Software-Defined Infrastructure (SDI), which basically generalize the same idea to a higher level at data center resources scale, as well as software resources such as software frameworks and licenses. In general, SDX approaches view the available resources—whether hardware ones such as the computing, communication, and storage, or software ones mentioned above—as pools of components that are (virtually) separated, and are employed on demand, under control of a data-center-wide software controller without, or with minimum, human intervention so as to use and share the resources more effectively based on the dynamically changing needs. Intel Rack Scale Architecture [62] is among such proposals in the industry. Other proposals also exist for resource interconnection and pooling such as [63] which connects commodity hardware resources using PCIe switches to provide remote access at affordable cost and higher-than-Ethernet speeds. Also, Catapult [64] provides a reconfigurable fabric to interconnect multiple FPGA accelerator boards each deployed on PCIe slot of a server; the FPGA boards are directly connected by pairs of SAS cables so that whole system provides high performance computation and communication for Microsoft Bing search engine among other possible uses. Venice [65] is another data center server architecture focusing on providing a strong communication substrate among server chips so that non-local resources can be efficiently used remotely by all chips.

Challenges and opportunities. As an opportunity, such Software-Defined Architectures further widen the design space available to users to more efficiently implement MapReduce, and more importantly, to adaptively decide how much of the provisioned resources are assigned to the MapReduce jobs at each time slot. Recently, even FPGA-as-a-Service (i.e., FPGA boards attached to VM instances) are offered by cloud providers such as Amazon, expectedly at a higher price. Thus the challenge of choosing the right amount of resources, and furthermore, adapting to the dynamics of the datacenter as well as the application, is deepening. Automating this process of resource allocation and also providing runtime systems for application-specific resource management and adaptation is an opportunity for researchers in the software and design-automation fields to provide automatic hardware-software implementation frameworks that efficiently map and elastically adapt MapReduce models to available processors and accelerators.

VIII. Summary and Conclusion

We reviewed the architectures, encompassing software as well as hardware, proposed for efficient implementation of batch big data processing in MapReduce paradigm as one of the most widespread programming models for large scale processing in today warehouse-scale computers. Several current open challenges and avenues for further enhancements were discussed. Furthermore, a number of trends envisioned in the industry, that are deemed to further deepen the need and motive to work further in this area, were described. Given the rapid growth of big data applications and their financial, social, and health benefits, the demand for higher throughput, computing capacity, and performance per Watt will only increase in foreseeable future. This increasingly intensifies the need and interest for further research in this area to fill in the gaps partly identified in this survey. Recent deployment of FPGAs in production datacenters [66] is a clear sign that the trend toward heterogeneous architectures for large-scale custom computing systems has only begun.

Acknowledgments

This work is supported by grant number G930826 from Sharif University of Technology. We are grateful for their support.

References

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," *IDC iView: IDC Analyze the future,* vol. 2007, pp. 1-16, 2012.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Int'l Symp. on Operating System Design and Implementation (OSDI),* 2004.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2nd Edition)*: Morgan & Claypool Publishers, 2013.

[4] (2016, Oct. 2016). *Apache Hadoop Project*. Available: http://hadoop.apache.org/

[5] R. Li, H. Hu, H. Li, Y. Wu, and J. Yang, "Mapreduce parallel programming model: A state-of-the-art survey," *International Journal of Parallel Programming*, vol. 44, pp. 832-866, 2016.

[6] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in MapReduce," *The VLDB Journal*, vol. 23, pp. 355-380, 2014.

[7] V. Vijayalakshmi, A. Akila, and S. Nagadivya, "The survey on MapReduce," *Int J Eng Sci Technol*, vol. 4, 2012.

[8] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, pp. 657-673, 1994.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, p. 95, 2010.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, *et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135-146.

[11] E. B. H. Esmaeilzadeh, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," presented at the International Symposium on Computer Architecture (ISCA), New York, 2011.

[12] (2016, Sep. 2016). *What is Big Data? by Gartner Research*. Available: http://www.gartner.com/it-glossary/big-data/

[13] (2016, Sep. 2016). *The Four V's of Big Data, by IBM*. Available: http://www.ibmbigdatahub.com/infographic/four-vs-big-data

[14] S. B. Joshi, "Apache hadoop performance-tuning methodologies and best practices," presented at the International Conference on Performance Engineering (ICPE), 2012.

[15] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema, "Towards machine learning-based auto-tuning of mapreduce," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*, 2013, pp. 11-20.

[16] J. Shafer, S. Rixner, and A. L. Cox, "The Hadoop distributed filesystem: Balancing portability and performance," presented at the International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010.

[17] C.-H. Chen, J.-W. Lin, and S.-Y. Kuo, "MapReduce Scheduling for Deadline-Constrained Jobs in Heterogeneous Cloud Computing Systems," *IEEE Transactions on Cloud Computing*, 2015.

[18] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing MapReduce On Heterogeneous Clusters," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, 2012, pp. 61-74.

[19] S. M. NabaviNejad and M. Goudarzi, "Energy Efficiency in Cloud-Based MapReduce Applications through Better Performance Estimation," presented at the International Conference on Design, Automation and Test in Europe (DATE), Dresden, Germany, 2016.

[20] S. M. NabaviNejad, M. Goudarzi, and S. Mozaffari, "The Memory Challenge in Reduce Phase of MapReduce Applications," *IEEE Transactions on Big Data*, 2016.

[21] W. Hu, C. Tian, and X. Liu, "Multiple-Job Optimization in MapReduce for Heterogeneous Workloads " presented at the International Conference on Semantics Knowledge and Grid (SKG), 2010.

[22] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for Shared-Memory Systems," presented at the MapReduce, San Jose, California, 2011.

[23] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," presented at the International Symposium on High Performance Computer Architecture (HPCA), 2007.

[24] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling," presented at the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2010.

[25] W. Jiang, V. T. Ravi, and G. Agrawal, "A Map-Reduce System with an Alternate API for Multi-Core Environments," presented at the International Conference on Cluster, Cloud and Grid Computing (CCGrid), 2010.

[26] R. M. Yoo, Anthony Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," presented at the International Symposium on Workload Characterization (IISWC) 2009.

[27] C. Kozyrakis. (2016, Oct. 2016). *Phoenix: An API and runtime environment for data processing with MapReduce for shared-memory multi-core & multiprocessor systems.* Available: https://github.com/kozyraki/phoenix

[28] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with Graphics Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 608-620, 2011.

[29] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," presented at the International Conference on Parallel architectures and compilation techniques 2008.

[30] (2016, Sep. 2016). *CUDA Occupancy Calculator by nVidia Corp.* Available: http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

[31] (2016, June 2016). *Panda Framework*. Available: https://github.com/futuregrid/panda

[32] Bryan Catanzaro, N. Sundaram, and K. Keutzer, "A Map Reduce Framework for Programming Graphics Processors," presented at the Workshop on Software Tools for MultiCore Systems (STMCS), Boston, 2008.

[33] S. Yesil, M. M. Ozdal, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, "Hardware Accelerator Design for Data Centers," presented at the Int'l Conference on Computer Aided Design (ICCAD), 20.15

[34] T. Honjo and K. Oikawa, "Hardware Acceleration of Hadoop MapReduce," presented at the Int'l Conf. on Big Data, 2013.

[35] C. Kachris, G. C. Sirakoulis, and D. Soudris, "A MapReduce scratchpad memory for multi-core cloud computing applications," *Microprocessors and Microsystems*, vol. 39, pp. 599-608, 2015.

[36] (2013, Apr. 2016). Accelerating Hadoop* Applications Using Intel® QuickAssist Technology. Available: http://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/accelerating-hadoop-applications-brief.pdf

[37] S. D. Kim, S. M. Lee, S. M. Lee, J. H. Jang, J.-G. Son, Y. H. Kim, *et al.*, "Compression accelerator for hadoop appliance," in *International Conference on Internet of Vehicles*, 2014, pp. 416-423.

[38] C. Kachris, G. C. Sirakoulis, and D. Soudris, "A Reconfigurable MapReduce Accelerator for multi-core all-programmable SoCs," presented at the Int'l Symp. on System-on-Chip, 2014.

[39] C. Kachris, D. Diamantopoulos, G. C. Sirakoulis, and D. Soudris, "An FPGA-based integrated mapreduce accelerator platform," *Journal of Signal Processing Systems*, pp. 1-13, 2016.

[40] D. Diamantopoulos and C. Kachris, "High-level Synthesizable Dataflow MapReduce Accelerator for FPGA-coupled Data Centers," presented at the SAMOS Conf., 2015.

[41] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce Framework on FPGA, A Case Study of RankBoost Acceleration," presented at the Symp. on Field Programmable Gate Arrays (FPGA), 2010.

[42] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A MapReduce Framework on OpenCL-Based FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 3547-3560, 2016.

[43] K. Neshatpour, M. Malik, M. A. Ghodrat, A. Sasan, and H. Homayoun, "Energy-Efficient Acceleration of Big Data Analytics Applications Using FPGAs," presented at the Int'l Conf. on Big Data, 2015.

[44] K. H. Tsoi and W. Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," presented at the Symp. on Field Programmable Gate Array (FPGA), Monterery, California, 2010.

[45] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan*, et al.*, "Map-reduce as a Programming Model for Custom Computing Machines," presented at the International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008.

[46] D. Yin, G. Li, and K.-d. Huang, "Scalable MapReduce Framework on FPGA Accelerated Commodity Hardware," presented at the Internet of Things, Smart Spaces, and Next Generation Networking, 2012.

[47] J. Costabile. (2015). *Hardware Acceleration for Map/Reduce Analysis of Streaming Data Using OpenCL*. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/technology/system-design/solutions/design-solution-syncopated-engineering.pdf

[48] Z. Lin and P. Chow, "ZCluster: A Zynq-based Hadoop Cluster," presented at the Int'l Symp. on Field Programmable Technology (FPT), 2013.

[49] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii, "Filtering: A Method for Solving Graph Problems in MapReduce," presented at the Symposium on Parallelism in Algorithms and Architectures 2011.

[50] Y.-M. Choi and H. K.-H. So, "Map-reduce processing of k-means algorithm with FPGA-accelerated computer cluster," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, 2014, pp. 9-16.

[51] C. Wang, X. Li, P. Chen, A. Wang, X. Zhou, and H. Yu, "Heterogeneous Cloud Framework for Big Data Genome Sequencing," *IEEE/ACM Transaction on Computational Biology and Bioinformatics*, vol. 12, pp. 166-178, 20.15

[52] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, *et al.*, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, 2016, pp. 456-469.

[53] ,2016)Sep. 2016). *OpenACC: Directives for Accelerators*. Available: http://www.openacc.org/

[54] H. Li, G. Fox, G. Laszewski, Z. Guo, and J. Qiu, "Co-processing SPMD Computation on GPUs and CPUs on Shared Memory System," presented at the International Conference on Cluster Computing (CLUSTER), 2013.

[55] C. Kachris, D. Soudris, G. Gaydadjiev, H.-N. Nguyen, D. S. Nikolopoulos, A. Bilas, *et al.*, "The VINEYARD Approach: Versatile, Integrated, Accelerator-Based, Heterogeneous Data Centres," presented at the Applied Reconfigurable Computing, 2016.

[56] H. Ahmadvand and M. Goudarzi, "Using Data Variety for Efficient Progressive Big Data Processing in Warehouse-Scale Computers," *IEEE Computer Architecture Letters*, 2016.

[57] (Dec. 2015, Sep. 2016). *Intel Completes Acquisition of Altera*. Available: https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[58] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, 2016, pp. 1-6.

[59] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing Data Analysis Workloads in Data Centers," presented at the International Symposium on Workload Characterization (ISWC), 2013.

[60] K. Asanović, "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers " presented at the Usenix Conference on File and Storage Technologies (FAST), 2014.

[61] (2016, Sep. 2016). *The Machine: The future of technology, by Hewlett Packard Labs*. Available: http://www.labs.hpe.com/research/themachine/

[62] (2016, Sep. 2016). *Intel® Rack Scale Design*. Available: www.intel.com/content/www/us/en/architecture/rack-scale-design-overview.html

[63] R. Hou, T. Jiang, L. Zhang, P. Qi, J. Dong, H. Wang*, et al.*, "Cost effective data center servers," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, 2013, pp. 179-187.

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

[65] J. Dong, R. Hou, M. Huang, T. Jiang, B. Zhao ,S. A. McKee, *et al.*, "Venice: Exploring server architectures for effective resource sharing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, 2016, pp. 507-518.

[66] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016, pp. 1-13.

Maziar Goudarzi (S’00, M’11, SM'16) is an Associate Professor at the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. He received the B.Sc., M.Sc., and Ph.D. degrees in Computer Engineering from Sharif University of Technology in 1996, 1998, and 2005, respectively. Before joining Sharif University of Technology as a faculty member in September 2009, he was a Research Associate Professor at Kyushu University, Japan from 2006 to 2008, and then a member of research staff at University College Cork, Ireland in 2009. His current research interests include architectures for large-scale computing systems, green computing, hardware-software co-design, and reconfigurable computing. Dr. Goudarzi has won two best paper awards, published several papers in reputable conferences and journals, and served as member of technical program committees of a number of IEEE, ACM, and IFIP conferences including ICCD, ASP-DAC, ISQED, ASQED, EUC, and IEDEC among others.