Melia: A MapReduce Framework on OpenCL-Based FPGAs

Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang

Abstract—MapReduce, originally developed by Google for search applications, has recently become a popular programming framework for parallel and distributed environments. This paper presents an energy-efficient architecture design for MapReduce on Field Programmable Gate Arrays (FPGAs). The major goal is to enable users to program FPGAs with simple MapReduce interfaces, and meanwhile to embrace automatic performance optimizations within the MapReduce framework. Compared to other processors like CPUs and GPUs, FPGAs are (re-)programmable hardware and have very low energy consumption. However, the design and implementation of MapReduce on FPGAs can be challenging: firstly, FPGAs are usually programmed with hardware description languages, which hurts the programmability of the MapReduce design to its users; secondly, since MapReduce has irregular access patterns (especially in the reduce phase) and needs to support user-defined functions, careful designs and optimizations are required for efficiency. In this paper, we design, implement and evaluate *Melia*, a MapReduce framework on FPGAs. Melia takes advantage of the recent OpenCL programming framework developed for Altera FPGAs, and abstracts FPGAs behind the simple and familiar MapReduce interfaces in C. We further develop a series of FPGA-centric optimization techniques to improve the efficiency of Melia, and a cost- and resource-based approach to automate the parameter settings for those optimizations. We evaluate Melia on a recent Altera Stratix V GX FPGA with a number of commonly used MapReduce benchmarks. Our results demonstrate that 1) the efficiency and effectiveness of our optimizations and automated parameter setting approach, 2) Melia can achieve promising energy efficiency in comparison with its counterparts on CPUs/GPUs on both single-FPGA and cluster settings.

Index Terms—FPGA, MapReduce, programming frameworks, cost model, OpenCL

1 Introduction

MapReduce, originally developed by Google for search applications, has become a popular programming framework in data centers with thousands of machines [15] or parallel architectures such as a machine with multi-core CPUs [34], Xeon Phi [29] or GPUs [18], [19], [23]. Many applications such as machine learning and data mining algorithms can be easily implemented with MapReduce, with a small set of simple and sequential APIs. MapReduce has abstracted the complexity of underlying hardware and systems from users. For example, Mars [18] allows users to adopt MapReduce interfaces to program GPUs, without worrying about the underlying details on GPU architectures. There are MapReduce design and implementation on other parallel architectures including multi-core CPUs [34] and CPU-GPU architectures [19]. In those studies, MapReduce is designed as a software library to improve the programmability of parallel architectures. Advanced features such as fault tolerance are usually neglected, which allows the design and implementation of MapReduce concentrating on individual parallel architectures.

On the other hand, Field Programmable Gate Arrays (FPGAs) have been an effective means of accelerating and optimizing many data processing applications such as relational databases [9], [32], [46], data mining [40], image processing [30] and streaming databases [41]. Quite different from CPUs and GPUs, FPGAs are (re-)programmable hardware and have very low energy consumption. Moreover, FPGA vendors such as Xilinx and Altera and have recently released OpenCL SDKs as a new generation of high-level synthesis (HLS) tools to users. Under the OpenCL abstraction, FPGAs can be viewed as massively parallel architectures. Encouraged by the success and wide adoptions of MapReduce, a MapReduce framework on FPGAs is able to enable users to program FPGAs with simple and familiar interfaces. The key problem is how to enable automatic performance optimizations for a MapReduce framework on FPGAs.

Despite the recent success of FPGAs in data processing applications, we have identified the following two key obstacles in the design and implementation of MapReduce on FPGAs. First, FPGAs are usually programmed with low-level hardware description languages (HDL) like Verilog and VHDL (e.g., [9], [32], [39], [46]). Although there has been a MapReduce implementation on FPGAs [37], the users are still required to implement map/reduce functions through VHDL/Verilog, which hurts the programmability and requires a long learning curve on both programming and performance optimizations. It is desirable that users can implement their custom data processing tasks with a high-level language. Second, since MapReduce has irregular access patterns (especially in the reduce phase) and needs to support user-defined functions, careful designs and optimizations are required for efficiency. Compared with CPUs/GPUs, FPGAs have lower clock frequency. Memory

- *Z. Wang, S. Zhang, and B. He are with Nanyang Technological University, Singapore.*
E-mail: {wangzeke638, tonyzhang19900609}@gmail.com, bshe@ntu.edu.sg.
- *W. Zhang is with the Hong Kong University of Science and Technology, Hong Kong. E-mail: wei.zhang@ust.hk.*

Manuscript received 31 Mar. 2015; revised 15 Dec. 2015; accepted 10 Feb. 2016. Date of publication 3 Mar. 2016; date of current version 16 Nov. 2016.
Recommended for acceptance by Y. Solihin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2016.2537805

stalls can be even more significant on FPGAs, especially for the irregular accesses from MapReduce.

To address those two obstacles, we implement and evaluate *Melia*, an OpenCL-based MapReduce framework on FPGAs. Melia takes advantage of the recent HLS tools developed by Altera, which provides an OpenCL SDK [10], [14], [38], to allow users to write OpenCL programs for FPGAs. In particular, the Altera's OpenCL SDK provides the pipeline parallelism technology to simultaneously process data in inherently multithreaded fashion. With the OpenCL abstraction, the FPGA can be modeled as a parallel device consisting of multiple pipelining execution units.[1] Based on OpenCL, Melia enables users to write simple and familiar MapReduce interfaces in C. To improve the efficiency of Melia on FPGAs, we evaluate a series of FPGA-centric optimizations such as *memory coalescing* and *private memory optimizations* for memory efficiency, and *loop unrolling* and *pipeline replications* for pipeline efficiency. Those optimizations introduce a series of tuning parameters which significantly affect the performance and resource utilization of Melia on FPGA. We develop a simple yet effective cost- and resource-based approach to determine suitable settings of those parameters.

Our experiments are conducted in two parts: real experiments on a single FPGA, and back-of-envelop performance/energy consumption analysis on multiple FPGAs in a cluster setting. We first evaluate Melia on the Terasic's DE5-Net board with an Altera Stratix V GX FPGA. We choose five commonly used MapReduce benchmarks. Our experiments demonstrate that 1) our parameter setting approach can predict the suitable parameter settings that have the same or comparable performance to the best setting, 2) our FPGA-centric optimizations significantly improve the performance of Melia on FPGA with an overall improvement of 1.4-43.6 times over the baseline (without optimizations) on FPGA; 3) As a sanity check, Melia achieves averagely 3.9 times higher energy efficiency (performance per watt) than the CPU- and the GPU-based counterparts. We further extend Melia to multiple FPGAs in a distributed setting, and evaluate the energy efficiency of Melia with performance/energy consumption analysis.

In summary, this paper makes the following three contributions. First, we propose the first OpenCL-based MapReduce framework for FPGAs to address the programmability problem of FPGAs. Compared with commercial tools such as Altera OpenCL SDK, this study offers a higher-level programming framework with MapReduce, which further abstracts the hardware details of FPGA, and resolves the programming complexity of FPGAs. Second, we implement our proposed system on the latest Altera FPGA, and empirically demonstrated the efficiency and effectiveness of FPGA-centric optimizations and our automated parameter tuning approach. Third, we discuss the lessons we have learned from experiences and provide insights and suggestions on programming FPGA.

The rest of the paper is organized as follows. We briefly introduce the background in Section 2. Section 3 describes the detailed design and implementations of Melia, followed by the experimental results on a single FPGA in Section 4. We extend the framework to FPGA cluster design in Section 5. We discuss our experiences from this study and point out a number of open problems in Section 6 and conclude this paper in Section 7.

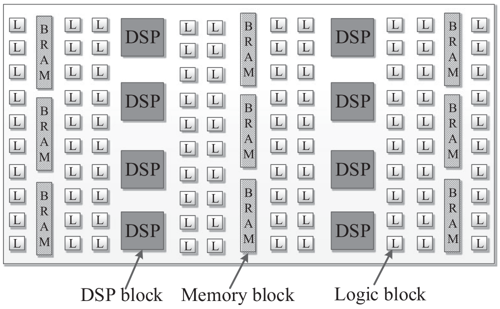


Fig. 1. Resource features on FPGA.

2 Background and Related Work

2.1 FPGAs

Generally, FPGA technology is low-power and offers a reconfigurable hardware solution for many applications. The FPGA implementation generally needs the input design specified at Register-transfer-level (RTL) or gate level using a HDL, such as Verilog and VHDL. Since HDL is cycle-sensitive and error-prone, generally good knowledge of hardware design detail and hand-on experiences are required to guarantee a successful design or implementation.

The most common part in the FPGA architecture [25] is logic blocks (called Configurable Logic Block, CLB (Xilinx), or Logic Array Block, LAB (Altera)), as shown in Fig. 1. They are fine-grained logic and capable to implement bit-level computation. Modern FPGA families expand to include coarse-grain function blocks into the silicon, such as DSP blocks and Memory blocks. Having these dedicated hardware-based macros embedded into FPGA helps implementation of computational intensive applications with less area and higher throughput.

There have been many studies on leveraging FPGAs in data processing applications (e.g., [17], [22], [44], [47]). We refer readers to a tutorial [31] for more details on FPGA-based data processing. Roughly, we can classify them into two major categories: integrating FPGA into the data path (e.g., [17]) and viewing FPGA as a co-processor/accelerator (e.g., [9], [30]). Using FPGAs in the data path, Netezza [17] employs FPGAs to filter and transform tuples from the disks prior to processing. Also, as an I/O engine, the FPGA-based circuits are developed for various data streaming operators, such as projection, selection and windowed aggregation [32], [33], [46]. Designed as an accelerator, FPGAs have been used to accelerate various database operations or applications such as join [9], [44] and frequent pattern mining [40]. Most previous studies implement specific applications with HDL. In contrast, this paper focuses on the implementation with high level synthesis.

2.2 Altera's OpenCL Architecture

OpenCL [24] has been developed for heterogeneous computing environments. OpenCL is a platform-independent standard where the data parallelism is explicitly specified in the code. This programming model targets at a host-accelerator

[1] This paper focuses on Altera FPGAs. Other vendors like Xilinx also have similar plans to support OpenCL.

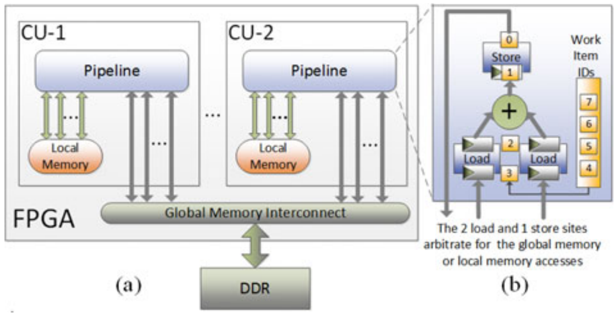

CU-1
CU-2
Pipeline
Pipeline
Local Memory
Local Memory
...
FPGA
Global Memory Interconnect
DDR
(a)
0
Store
1
Work Item IDs
7
6
5
4
+
2
3
Load
Load
The 2 load and 1 store sites arbitrate for the global memory or local memory accesses
(b)

Fig. 2. Altera OpenCL system implementation.

model of program execution, where a host processor runs control-intensive task and offloads computationally intensive portions of code (i.e., kernel) onto an external accelerator.

Recently, Altera provides the OpenCL SDK [38], [45] to abstract away the hardware complexities from the FPGA implementation. Fig. 2a illustrates the Altera architecture for OpenCL system implementation. An OpenCL kernel execution contains multiple kernel pipelines and their interconnects with global memory and local memory. The Altera's SDK can translate the OpenCL kernel to low-level hardware implementation by creating the circuits for each operation of the kernel and interconnect them together to realize the data path of the kernel. Fig. 2b shows the pipelined parallelism in the case of a simplified vector addition example [38], which can achieve the throughput of one work item finished per cycle. The frequency of FPGA kernel can vary with the OpenCL kernel. It mainly depends on the FPGA resource utilization by the OpenCL kernel. Ideally, the more resource that the kernel consumes, the lower frequency that the FPGA execution has.

From the perspective of OpenCL, the memory component of FPGA computing system contains three layers. First, the *global memory* resides in DDRs on the FPGA board. The accesses to global memory has long latency. Second, the *local memory* is a low-latency and high-bandwidth scratchpad memory. In our tested FPGA, the local memory has 4 banks in general. The *private memory*, storing the variables or small arrays for each *work item* (i.e., the basic execution unit in OpenCL), are implemented using completely-parallel registers which are plentiful resources in FPGA. Compared with CPU/GPU, FPGA has relatively sufficient number of registers, which should be employed efficiently to store intermediate results for each individual work item.

As in Fig. 2, we can configure multiple kernel pipelines, i.e., Compute Unit (CU), if resource allows. Different CUs are executed in parallel. Each CU implements a massive pipelined execution for the OpenCL program, and has its own local memory interconnect while all the pipelines share the global memory interconnect. In particular, the load/store operations to local memory access in one CU can combine together to arbitrate for the local memory. However, the load/store operations to global memory access will compete for the on-board global memory bandwidth [38]. Compared with global memory, the on-chip local memory is low-latency and high-throughput. Moreover, the global memory system is lack of dedicated cache hierarchy which causes the global memory transactions of FPGA are less efficient than that of GPU/CPU. Thus, the local and private memory should be employed whenever possible to reduce global memory accesses.

2.3 MapReduce

MapReduce is a programming framework, originally developed by Google and mainly used for parallel and distributed data processing. In the big-data era, MapReduce has gained a significant amount of interests from both industry and academia. The basic idea of MapReduce is to offer simplified data processing and to hide the details of parallel and distributed executions from users. Formally, MapReduce consists of two user-defined functions: *Map* and *Reduce*. The Map function takes as input a key-value pair (key1, value1) and generate intermediate key-value pairs in the form of (key2, value2). Next, the system automatically groups the intermediate key-value pairs on the key, and forms the pairs of a key and the values of the same key (key2, list(value2)). For each key2, the Reduce function processes its corresponding value list. Many previous studies (e.g., [15], [16], [18], [23], [34]) have demonstrated that MapReduce offers simplified yet reasonably efficient parallel and distributed data processing. More details about MapReduce and its usage in parallel data processing can be founded from recent surveys [27], [28].

Closely related to this study, FPMR [37] attempted to implement MapReduce on FPGA. However, those studies are limited in two aspects. First, the developers [37] are still required to implement map/reduce functions through VHDL/Verilog. Second, FPMR is rigid in some specific application (without flexible data shuffling). Instead, this paper has the full OpenCL-based MapReduce framework on FPGAs, and the MapReduce can also support flexible data shuffling. In [43], [48], FPGAs (together with GPUs) are adopted to implement the MapReduce framework, where the host CPU implements the scheduling task and the FPGAs (together with GPU) are considered as co-processors. There have also been two studies [13], [36] on offering the capability of executing MapReduce functions in OpenCL. Still, they are very preliminary in the sense that they only implement very basic form of MapReduce. The major contributions of our paper include 1) offering a more FPGA friendly MapReduce framework, and 2) the optimizations are guided by the cost model.

On parallel architectures, there have been OpenCL-based MapReduce implementations [11], [12], which target at the multi-core CPU or the GPU in a single host. The state-of-the-art OpenCL implementation of MapReduce on CPUs/GPUs [12] is imported to FPGAs, denoted as *baseline*. We have observed that the baseline implementation, which does not include optimizations (e.g., loop unrolling), suffers from severe memory stalls and pipeline inefficiency (as we will see in the experiments).

3 Design and Implementation of Melia

This section presents design and implementation of Melia on a single FPGA. Based on the single-FPGA implementation, we extend our design to multiple FPGAs in Section 5.

3.1 Melia Overview

We have identified the following two key challenges for an efficient MapReduce on FPGAs. The first problem is on high-latency global memory transactions. Unlike the CPU/GPU, the FPGA does not have dedicated cache hierarchy. Then, the global memory access transactions generated on the FPGA directly interface with the memory controller of the external memory. Second, writing the OpenCL program should consider the efficiency of pipeline executions on FPGAs.

With the abstraction of Altera OpenCL SDK, the FPGA can be modelled as a massively parallel architecture with a multi-level memory hierarchy. Many design and implementation optimizations that have been developed for the CPU and the GPU can be applicable to the OpenCL program, and their impact should be revisited under the new FPGA abstraction. Example optimizations include memory coalescing and local memory optimizations to resolve the memory stalls. On the other hand, there are some new optimization strategies that are particularly attractive on the new FPGA abstraction. Examples include pipeline replications and loop unrolling. Altera OpenCL SDK explicitly supports loop unrolling to take advantage of the flexible hardware resource allocations on FPGAs. Pipeline replications enable multiple replicated pipelines to execute in parallel to fully take advantage of hardware resources on FPGAs. Those optimizations are correlating factors in performance tuning for the OpenCL-based MapReduce on FPGAs, including hardware frequency and resource utilization. Due to the architectural difference between FPGAs and CPUs/GPUs, many tuning knobs [11], [12] from CPUs/GPUs are no longer applicable to FPGAs.

Taking those issues into account, our design of Melia addresses the aforementioned challenges. Our optimizations improve the memory efficiency and pipeline efficiency. To ease the complexity in performance tuning, we develop a simple yet effective cost- and resource-based approach to automatically determine suitable settings of those parameters. The approach takes into consideration the cycles of the pipeline, the frequency and resource limitation of FPGA, and recommends the best parameter configuration. We first present the overall workflow of our implementation, and details of our optimizations and automated parameter settings in the later two sections.

Melia is currently designed and implemented as a software library. Users are able to use Melia, almost in the same way as other MapReduce frameworks [18], [19], [23]. Specifically, users need to first implement a *map()* and a *reduce()* function in C. For the *reduce* function, users can annotate whether it is an associative and communicative function. If so, Melia can enable *early reduction* optimization. Given the two user-defined functions, Melia first determines the suitable execution parameters (Section 3.3). Next, the user compiles and executes the program on the FPGA. During the execution, Melia executes the two user-defined functions according to the overall workflow in Algorithm 1.

The overall execution of Melia is designed as two stages: map and reduce. The map function takes one input unit and then generates one key-value pair. Whenever an intermediate key/value is emitted, the *insert()* is invoked (in Algorithm 2). The system maintains a bucket based hash table. The bucket stores the key-value pairs or *reduction object* [11], [12] for each key. The usage of reduction object is to represent the partial reduction result. If the reduce function is associative and communicative, the key-value pair is inserted to a reduction object. Otherwise, it is directly appended to hash table. Multiple OpenCL work items access the shared hash table. Locks are used for synchronization among work items. In the reduce phase, each work item is responsible for one bucket of the hash table. If reduction objects are used, no explicit reduction phase is conducted.

Algorithm 1. Overall workflow of Melia

```
/* Stage 1: the map stage; */
for each key/value pair in the input do
    execute map(); //when an intermediate key/value is
    emitted, the insert() is invoked.
end
/* Stage 2: the reduce stage; */
for each key/value pair in the intermediate output from the map
stage do
    execute reduce();
end
```

Algorithm 2. Insert $(key, key\_size, val, val\_size)$

```
$index$ = hash($key, key\_size$)%NUM_BUCKETS;
$DoWork$ = 1;
while ($DoWork$) do
    /* wait until having lock[$index$]                     */
    $with\_lock = 0$;
    while($with\_lock == 0$) do
        $with\_lock$ = get_lock($index$);
    end
    $index\_base = index$;
    /* (coalescing read from 128-bit memory) :
       valid, key_addr, val_addr,key_val_size */
    $bucket\_unit4 = buckets[index]$;
    /* bucket[index] is empty                                */
    if ($bucket\_unit4.valid == 0$) then
        ($key\_addr, val\_addr$) = atomic_alloc($key\_size, val\_size$);
        /* (coalescing write to 128-bit memory):
           valid, key_addr, val_addr,key_val_size*/
        $bucket\_unit4 = (1, key\_addr, val\_addr, (key\_size, val\_size))$;
        $buckets[index] = bucket\_unit4$;
        /* store key and value data                          */
        copy($key\_addr, key, key\_size$); copy($val\_addr, val, val\_size$);
        $DoWork = 0$;
    end
    /* $key$ is same as bucket[$index$]                        */
    else
        if (equal($bucket\_unit4.key\_addr$, $bucket\_unit4.key\_size$,
        $key$, $key\_size$)) AND reduce is associative and commu-
        nicative then
            /*reduce $val$ to bucket[$index$]                   */
            reduce($bucket\_unit4.val\_addr$, $bucket\_unit4.val\_size$,
            $val$, $val\_size$); $DoWork = 0$;
        end
        /* $key$ is not same as bucket[$index$]                */
        else
            $DoWork = 1$;
            $index$ = update_index($index$);
        end
    end
    /* release the lock[$index\_base$]                         */
    release_lock($index\_base$);
end
```

Our implementation requires quite some design and engineering efforts in optimizing the efficiency of Melia. We take as one example the insertion of a key-value pair into a reduction object in MapReduce, illustrated in Algorithm 2. When a key-value pair is to be inserted into the reduction object, the index is calculated via the hash value of the key. Since there are read/write conflicts to the same bucket, a lock mechanism is employed. The work item polls the corresponding lock of the index until the work item acquires the lock. If the bucket of the index is empty, Melia first creates a new bucket in the hash table. If the key of the bucket is same as the inserted key, Melia atomically reduces the key-value pair to the bucket, using the reduce function provided by the user. If the keys are not the same, the computing work item calculates a new index for the next round.

Melia employs the static memory coalescing, in terms of built-in vector type *unit4*, to combine several small-sized global memory accesses to form the vector load/store accesses (e.g., the register *bucket_unit4*). Therefore, the global memory transactions for the bucket information in Melia are one vector load operation (Line 9) and one vector store operation (Line 12). With the reduced number of load/store operations, the OpenCL kernel can use less hardware resource and then might achieve higher frequency.

3.2 Optimization Techniques

To reduce the number of global memory transactions, Melia employs a series of memory optimizations such as *memory coalescing* and *private memory optimizations* [4]. To improve the pipeline execution efficiency, Melia converts multiple nested loops into a single loop and combines the replicated instructions whenever possible. Then, it is more efficient to map to the FPGA pipeline. Furthermore, we apply *loop unrolling* and *pipeline replications* to better utilize the FPGA resource. Those optimizations are automatically included in our framework implementation. For user-defined functions, only loop unrolling is automatically applied in Melia (by identifying the target loops through source code analysis), and other optimizations are left to users.

Private memory. The private memory on FPGA are implemented using completely-parallel registers (logics), which are plentiful resources in FPGAs. Then, the private memory is useful for storing single variables or small arrays in the OpenCL kernel [4]. The kernel can access private memories completely in parallel, and no arbitration is required for access permission. Therefore, the private memory has significant advantages, in terms of bandwidth and latency, over local memory and global memory. Since the general MapReduce applications require a lot of memory accesses, we should use private memory, instead of local memory and global memory, whenever possible.

Local memory. The local memory on the FPGA is considerably smaller than global memory; however, it has significantly higher throughput and much lower latency. The local memory are implemented in on-chip memory blocks [5] in the FPGA. The on-chip memory blocks have two read and write ports, and have twice the operating frequency as the frequency of the OpenCL kernel pipelines. Thus, the local memory is able to support four simultaneous memory accesses. Therefore, the local memory is good for the intermediate data between the work items in the same work group. In Melia, we maintain reduction objects in the local memory.

Kernel pipeline replication. If the resource is sufficient on the FPGA, the kernel pipeline can be replicated to generate multiple compute units (CUs) to achieve higher throughput. Generally, each CU can execute multiple work-groups simultaneously. The inner hardware scheduler can automatically dispatch the work-groups among CUs. For example, if two CUs are implemented, each CU executes a half of the work-groups.

Since kernel pipeline replication can consume more resource, the frequency tends to be lower than that of one kernel pipeline. That means, two CUs cannot double the performance. Another issue is that the global memory load/store operations from the multiple compute units compete for the global memory accesses. Nevertheless, we find that more compute units can still bring performance gains in most cases. Hence, we simply take the largest number of CUs that can fit into the resource budget of FPGA.

Loop unrolling. If a large number of loop iterations exist in the kernel pipeline, the loop iterations could potentially be the critical path of the kernel pipeline. Then, unrolling the loop by an unroll factor could increase the pipeline throughput by decreasing the number of iterations. However, on FPGA, loop unrolling is achieved at the expense of increased hardware resource consumption. Different from loop unrolling on CPUs/GPUs, the FPGA allocates more hardware resources to the execution of unrolled loops.

Loop unrolling might have another side-product benefit: the load/store operations with simple array indexes, are coalesced so that more valid data can be loaded per memory transaction. This reduces the number of total memory accesses, which further improves the performance.

3.3 Parameter Settings for Melia

The FPGA compilation time is long (hours) and there are several optimization parameters to tune the performance in Melia. The design space of optimizations is large, since there are a number of optimization methods and we need to determine where to apply these optimizations in the OpenCL-based MapReduce applications. It is critical to address the main bottleneck by the proper optimizations. Therefore, it is necessary to have an automated tool which can guide the parameter settings, under the resource constraints in FPGA. Additionally, since different kinds of optimizations consume different amount of hardware resources on FPGAs, this paper presents the FPGA-specific cost model to guide the suitable optimization configuration for MapReduce. Due to the resource constraints of an FPGA, the selection and configuration of individual optimizations significantly affect the application performance, as we demonstrated in Section 4.

The flow contains three stages to determine tuning parameters for local memory, loop unroll and replicated kernel pipelines accordingly.

Stage 1: It is the user to determine whether the local memory is employed, according to the specific MapReduce application. MapReduce applications can be roughly divided into the reduction-intensive and map computation-intensive applications [12]. The former kind has a large number of key-value pairs for each unique key, and then the reduction computation time is significant. The later kind represents the applications that spend most of their time for computation in the map stage. Therefore, the local memory is recommended for the reduction-intensive applications and the size of local memory are determined by the user. However, the local memory is not suitable for the applications of map computation-intensive applications (e.g., no key-value pairs share the same key).

Stage 2: The design flow guides how to determine the unroll factor f in the Map/Reduce function. If no fixed loop iterations exist in the Map/Reduce function, then f is 1 and the design flow directly go to the next stage (CU_num). Otherwise, there are $total\_loop\_num$ iterations in the map/reduce function, and we roughly estimate the unroll factor (f) as follows.

On the current version of Altera OpenCL SDK, it is recommended that f is a divisor of $total\_loop\_num$. The system iterates all possible unroll factor(f), ranging from the smallest divisor (1) to the biggest divisor ($total\_loop\_num$) in the map/reduce function. Next, the OpenCL kernel with the unroll factor(f) is passed to the *Altera resource estimation tool* [4] to estimate the resource utilization of the OpenCL kernel. While the entire compilation process may take hours, the resource estimation can give the statistics on resource usage in seconds or minutes. Then, the *cost model* roughly provide the rough trends of the execution cycles and kernel frequency. We estimate the execution time by multiplying the estimated execution cycles with the estimated frequency. The details on estimating the frequency and clock cycles are described in Sections 3.3.1 and 3.3.2. We accept the unrolling factor only if the kernel can fit into the FPGA.

Stage 3: We determine the CU_num, the maximum number of replicated kernel pipelines under the constraint that the required utilization of each feature (such as logic, memory block and DSP block) is less than a predefined resource usage threshold (95 percent in our study).

In the following, we present the details on our cost models. The proposed cost models are used to guide the developer how to determine the parameter setting for the MapReduce applications, not to accurately predict the frequency and clock cycles. The unique architectural feature of FPGA actually allows us to simplify the cost estimation. In our experiment, we observe that our cost models can roughly predict the suitable parameter configuration, and the simplified cost models are sufficient for the purpose.

3.3.1 Cost Model for Estimating Frequency

It is hard to develop an accurate analytical model to estimate the hardware frequency due to the internal complexity of FPGA. Fortunately, we observe that there is a strong correlation between the resource utilization on FPGA and the hardware frequency. Thus, we develop a simple linear regression model for hardware frequency based on resource utilization, which is generally accurate enough for our experiments.

The FPGA mainly has three features (logic element, memory block and DSP block), and each feature can have different resource utilizations. For simplicity, we assume that the feature with the largest utilization is chosen to determine the frequency of kernel. Next, we use the applications in the Altera OpenCL SDK as training data sets. For each application, we obtain the maximum resource utilization and the kernel frequency. Finally, we apply least squares method to determine the linear model function that can best fit the training data set, and obtain the estimated frequency $F_{estimated}$. In our experiment, we obtain the linear model in Eq. (1), where the $R_{max\_utilization}$ is the maximum resource utilization of the given OpenCL kernel, reported from Altera resource estimation tool [4]

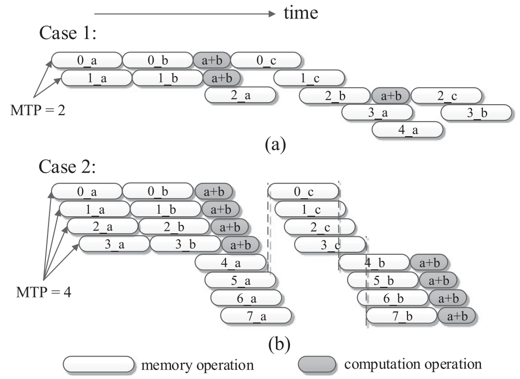


Fig. 3. OpenCL kernel execution flow: (a) MTP = 2, (b) MTP = 4.

$$F_{estimated} = -79 * R_{max\_utilization} + 245\text{MHz}. \tag{1}$$

3.3.2 Cost Model for Estimating Clock Cycles

The Altera's OpenCL Compiler [38] translates the OpenCL kernel to a hardware pipeline, which implements each operation in the OpenCL kernel by the specific circuit. Then, these circuits are wired together to execute the pipeline. Then, the massive parallelism exists in the global memory accesses and arithmetic computations. The total clock cycles for the execution highly depends on the degree of global memory parallelism in the kernel pipeline. We adopt one metric [20], *MTP (Memory Thread Parallelism)*, to represent the maximum number of threads that can access the global memory simultaneously.

To further explain how the multiple threads are executed in the kernel pipeline, we illustrate the pipeline execution of the vector addition, as shown in Fig. 3. For Case 1 in Fig. 3a, the global memory system can service two global memory transactions simultaneously (MTP = 2), and the "m_x" indicates the work item with ID (m) loads from (x = a or b) or stores to (x = c) the global memory. In this case, the computation operations are completely hidden behind the global memory operations, the kernel throughput is bounded by the global memory transactions. For Case 2 in Fig. 3b, it can service four global memory transactions simultaneously (MTP = 4), and then the kernel throughput is greatly improved.

We estimate the total number C_{FPGA} of elapsed clock cycles on the FPGA to be the larger one of the clock cycles for memory accesses and computations (Eq.(2)). C_{mem} and C_{comp} denote the total number of clock cycles in global memory accesses and the total number of clock cycles in computations, respectively. This estimation simplifies the interaction between memory accesses and computation, which assumes a maximum overlapping between C_{mem} and C_{comp}. Due to the massive parallel pipeline on FPGAs, this overlapping is high in practice and the simplified estimation is sufficient

$$C_{FPGA} = Max(C_{mem}, C_{comp}). \tag{2}$$

TABLE 1
Latency (cycles) of Each Kind of Instructions

fp_sqrt	fp_mul	fp_add/sub	fp_div
28	5	7	14
int32_add/sub	int32_mul	int32_div	global memory
1	3	32	35

Estimating C_{comp}. Based on the full pipelined property of the arithmetic operation implemented on FPGA, the arithmetic operation can achieve the throughput with one operation per cycle. Another advantage of arithmetic operation is that each arithmetic operation in the OpenCL is implemented with specific circuit, then no resource competition will occur among arithmetic operations. Therefore, we estimate C_{comp} to be the total number of clock cycles for all instructions in the critical path. We have developed a tool to count the number of instructions in each kind, and multiply the unit cost of each kind of instruction. Table 1 lists a sample of instructions and their unit costs on the FPGA used in our experiments. We obtained the unit costs from profiling the FPGA IP cores of the Altera OpenCL SDK.

Estimating C_{mem}. We consider two major factors: total number of memory accesses and how memory accesses are served in parallel on the FPGA. Eq.(3) gives the estimation on C_{mem}, where L_{mem} and N_{mem} denote the clock sum of the total global memory accesses and the latency of one global memory access and the number of global memory accesses, respectively. Thus, $L_{mem} \times N_{mem}$ denotes the total clock cycles for memory accesses, if memory requests are served one by one. On FPGAs, memory accesses are severed in parallel with a degree of MTP. L_{mem} is obtained from profiling the FPGA, and N_{mem} and MTP can be obtained with the simulation tool [49]. Differently, we consider that the FPGA does not have dedicated cache hierarchy, when counting N_{mem}

$$C_{mem} = \frac{L_{mem} \times N_{mem}}{MTP}. \tag{3}$$

4 Experimental Evaluation

This section presents the experimental studies on a single FPGA. The major goal of the experiments is to evaluate the efficiency and effectiveness of the optimization techniques in Melia over the baseline implementation on FPGA [12].

4.1 Experimental Setup

Our experiments were conducted on a machine with CPU and one FPGA board (Terasic's DE5-Net board) which includes 4 GB DDR3 device memory, and an Altera Stratix V GX FPGA (5SGXEA7N2F45C2). The FPGA [5] includes 622 K logic elements, 2560 M20 K memory blocks (50 Mbit) and 256 DSP blocks. The FPGA board is connected to the host via an X8 PCI-e 2.0 interface.

We compare Melia with the state-of-the-art OpenCL MapReduce [12] on the high-end 2.40 GHz Intel Xeon CPU E5645 (12 cores) and an AMD FirePro V7800 GPU. The peak DRAM bandwidth of the high-end Intel CPU is around 32 GB/sec. The low-end CPU is the Intel Xeon Processor E3-1230 L. The GPU has 18 streaming multiprocessors (SM), and each SM has 128 Radeon cores, with a clock rate of 700 MHz. Thus, there are 1440 Radeon cores on this GPU. Each SM has 32 KB local memory. The device memory is 2 GB DDR5, with 1200 MHz clock frequency and peak bandwidth of 153.6 GB/sec. The GPU is connected to the host via an X16 PCI Express 3.0 interface.

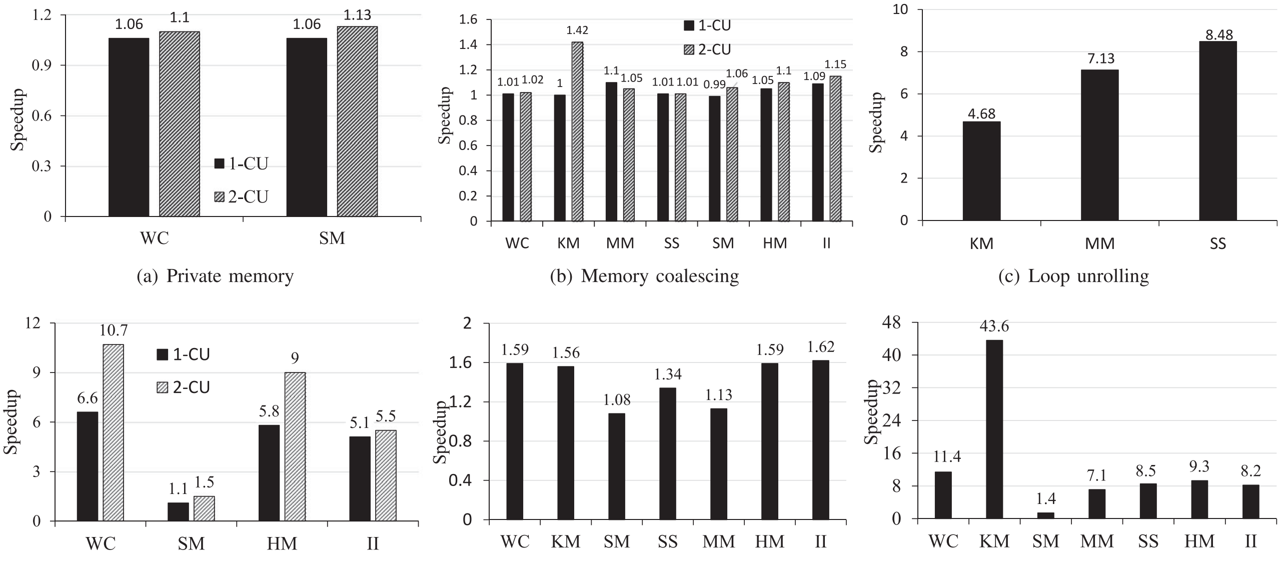
TABLE 2
Application and Datasets Used in our Experiments

Application	Dataset Size
K-means, K = 40 (KM)	200 M points
Word Count (WC)	100 MB text file
String Matching (SM)	100 MB text file
Matrix Multiplication (MM)	2,000*2,000 matrices
Similarity Scope (SS)	2,000 files each with 2,000 features
Histogram movies (HM)	100 M movive rating points
Inverted index (II)	200 M tuples

A fair and accurate comparison on the energy consumption across multiple platforms is a nontrivial task, since these three platforms can have very different hardware and peripheral equipment in practice. Thus, we adopt two methods to compare the energy efficiency among three platforms. The first method is an estimation with multiplying the execution time by the corresponding TDP (Thermal Design Power) of the platform. This methodology is used in the previous studies [7], [10]. In practice, this offers a good estimation on the energy consumption of each platform, since we have various optimizations to maximize the resource utilizations on high-end CPU, GPU and FPGA. The second method is to further add a low-end CPU power consumption for the FPGA/GPU implementation, in addition to the first method. The reason of using a low-end CPU is, since the CPU is roughly idle during OpenCL kernel on FPGA/GPU are running, it is unfair to count the power consumption of full-fledged Intel CPU into the power consumption of FPGA/GPU platform. In this study, we assume the energy consumption of the low-end CPU to be 25W. The TDPs of the high-end CPU, the GPU and the FPGA are 80, 150, and 25 W, respectively.

Applications. We have used seven common MapReduce benchmarks, which have been used in the experiments of previous studies [1], [12], [18], [21].

These applications cover different performance aspects of MapReduce: *reduction-intensive* and *map computation-intensive* applications. The former kind of applications usually have a large number of key-value pairs for each unique key, whereas the map tasks spend most of the time in the latter kind of applications. The details on the applications and their data sets are summarized in Table 2. The default data have uniformly distributed input keys. K-means clustering (KM) is one of the most popular data mining algorithms. Word Count (WC) can be reduction-intensive if the number of distinct words (DW) is small. We use DW=500 as the default setting. String Matching (SM) is used to check whether the target string is in the file. For simplicity, the first string in the file is set to be the target string to search. Matrix Multiplication (MM) is a map computation-intensive application. Similarity Scope (SS) is used in web document clustering, which computes the pair-wise similarity score for a set of documents. It is also a map computation-intensive application. Histogram movies



(a) Private memory

(b) Memory coalescing

(c) Loop unrolling

(d) Local memory

(e) 2-CU

(f) Melia over baseline on the FPGA

Fig. 4. Performance speedup of individual optimization on the FPGA, where K-means (KM), Word Count (WC), String Matching (SM), Matrix Multiplication (MM), Similarity Scope (SS), Histogram movies (HM) and Inverted index (II).

(HM) generates a histogram of the movie rating data. It is a reduction-intensive application. Inverted index (II) generates word-to-document indexing for a list of documents. It is a reduction-intensive application. Among them, KM and WC are in HiBench [21], while HM and II are in PUMA [1].

In summary, MM and SS are map computation-intensive, and others are reduction-intensive. The input data sets are initially loaded into the device memory, excluding the cost of PCI-e data transfer time.

4.2 Impacts of FPGA-Centric Optimizations

In this section, we study the separate impact of individual FPGA-centric optimizations in Melia, through manually enabling/disabling certain optimizations in Melia. It is important to study the impacts of these optimizations, since the performance can be significantly improved with proper optimizations.

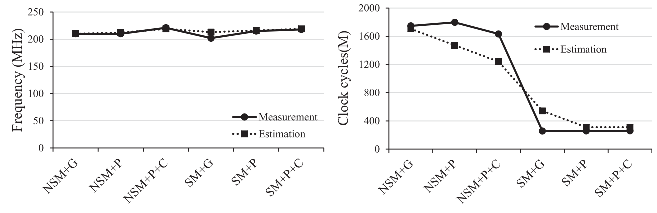
Private memory. We first study the performance impact of the private memory access optimization. Fig. 4a shows the *speedup* of private memory on Melia with one and two CUs (denoted as 1-CU and 2-CU, respectively). We define the performance speedup of an optimization technique to be the ratio of the elapsed time without the optimization technique to that with the optimization technique. We recommend that the private memory should be chosen for storing intermediate data in the Melia framework and user-defined map/reduce functions whenever possible. One reason is that FPGA has a plentiful amount of reconfigurable logics for the private memory. The usage of the private memory reduces the number of long-latency global memory accesses. Since the multiple kernel pipelines are more global memory intensive than that of one kernel pipeline, the 2-CU case can achieve a higher performance speedup than that of 1-CU case. We do not include the results for SS, MM, KM, HM and II, because the private memory optimization is not necessary for those applications.

Memory coalescing. Fig. 4b shows the performance speedup of the static memory coalescing on the seven applications. With memory coalescing, multiple global memory transactions are combined, and the total number of global memory accesses is reduced. Similar to the results on private memory optimizations, the 2-CU case also achieves more performance speedup than that of 1-CU case. Specific to FPGA, this optimization also reduces the hardware required resource consumption. We use KM as an example, and memory coalescing has a significant speedup of 1.42 on KM. The 2-CU KM variants with and without coalescing require 72 and 93 percent of the total FPGA resource, respectively. Even worse, the high resource consumption also leads to a lower frequency. Those two factors contribute to the relatively high overall speedup of memory coalescing on KM.

Loop unrolling. Fig. 4c shows the performance speedup of the loop unrolling on the FPGA. Loop unrolling is not applicable to SM and WC, due to their irregular loops. For the other three applications, loop unrolling achieves very significant performance speedup (up to 8.48). The throughput of the pipeline in the FPGA is always determined by the slowest part of the pipeline. Through loop unrolling, we can allocate more resource to the slowest part of the pipeline, and make the throughput of each part of pipeline more balanced.

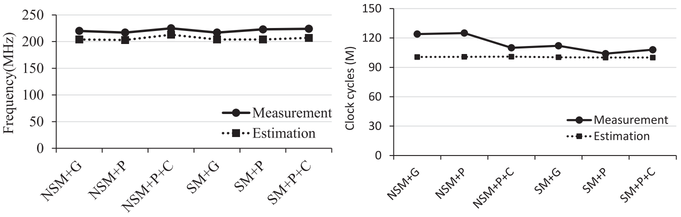
Local memory. Fig. 4d shows the performance speedup of the local memory for WC SM HM and II. The local memory has significant advantages in latency and throughput over global memory. Another advantage is that each kernel pipeline has its own local memory, the pipeline do not need to compete with the other kernel pipelines for local memory accesses, unlike global memory accesses. Since each kernel pipeline has its own local memory, the 2-CU case can achieve more significant performance speedup than 1-CU case.

Pipeline replication. Fig. 4e shows the performance speedup of the multiple kernel pipelines (CU) on the FPGA. Increasing the pipelines from one to two results in the speedup of 1.08-1.59 on the seven applications. That shows the importance of fully utilizing the hardware resource.



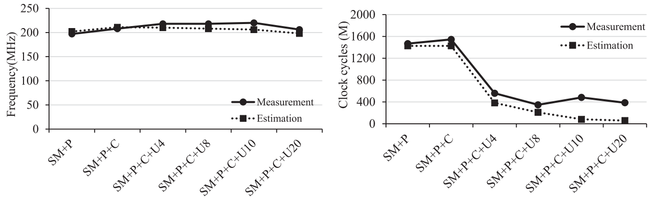
(a) Frequency estimation (b) Clock Cycle estimation

Fig. 5. Frequency and clock cycle estimations of WC on the FPGA.



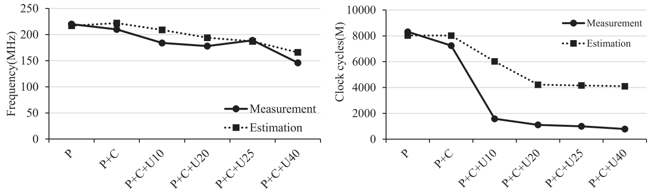
(a) Frequency estimation (b) Clock cycle estimation

Fig. 6. Frequency and clock cycle estimations of SM on the FPGA.



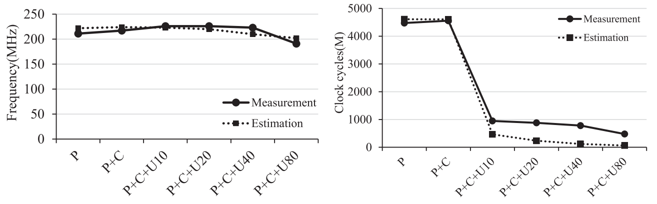
(a) Frequency estimation (b) Clock cycle estimation

Fig. 7. Frequency and clock cycle estimations of KM on the FPGA.



(a) Frequency estimation (b) Clock cycle estimation

Fig. 8. Frequency and clock cycle estimations of MM on the FPGA.



(a) Frequency estimation (b) Clock cycle estimation

Fig. 9. Frequency and clock cycle estimations of SS on the FPGA.

Put them all together. Finally, we compare Melia with the baseline approach (without FPGA-specific optimizations) on FPGA, as shown in Fig. 4f. The speedup of all FPGA- centric optimizations is 1.4-43.6 times over the baseline approach. This validates the importance of FPGA-centric optimizations in writing an efficient OpenCL program for FPGAs.

4.3 Cost Model Evaluations

In this section, we evaluate our cost models from two aspects: cycles and frequency estimations and optimization parameter setting.

Estimations of cycles and frequency. We first study our predictions on the clock cycles and hardware frequency. We have studied three reduction-intensive applications (WC, KM and SM) and two map computation-intensive applications (MM and SS). We observe that our predictions can generally capture the trend of clock cycles and frequency. In the following, we present the detailed results for two representative applications, WC and SS, without and with loop unrolling optimizations, respectively. Additionally, they cover a series of memory optimizations.

For each application, we consider different combinations of FPGA-centric optimizations. Thus, we use the following abbreviations to represent the optimizations and their parameters used in the evaluation: G, P, C, SM, NSM and Uf represent the baseline global memory version, private memory, static memory coalescing, local memory, non local memory, and loop unrolling with unrolling factor f, respectively.

Figs. 5a, 6a, 7a, 8a and 9a show the predictions on hardware frequency of running WC, SM, KM, MM and SS with Melia, respectively, in comparison with the measured frequency after the real FPGA compilation. Our simple approach can roughly predict the hardware frequency of the OpenCL kernel, with the input from the corresponding estimated resource utilization provided by the *Altera resource estimation tool*.

Figs. 5b, 6b, 7b, 8b and 9b show the predictions on the elapsed clock cycles. Generally, our prediction on clock cycles is able to capture the trend of the MapReduce application with different parameter configurations. On WC, our estimation can predict the clock cycle reductions of the memory optimizations (local memory, private memory and static memory coalescing), and the corresponding MTP value used in Fig. 5b is 11.3. For SS, KM and MM, our estimation can also predict the impact of loop unrolling, which significantly reduces the clock cycles by shortening the critical path of the kernel pipeline, and their corresponding MTP values are 30.4, 60 and 70, respectively. For SS, our estimation can predict the clock cycle trend with varying unrolling factor f. For MM and KM, our estimation can not accurately predict the clock cycle trends, but the performance of the estimated parameter configuration can be very close to the optimum performance.

Optimization parameter setting. We now evaluate the effectiveness of our models in predicting the suitable parameter settings in Melia. We study the predicted optimization configuration of parameter settings for the seven applications in comparison with the best configuration in Table 3. We obtain the best/worst/medium configurations by experimentally

TABLE 3
Configuration of Best and Predicted Cases for the Five Applications

Configuration	Best Case	Predict
WC	SM+P+C+2CU	SM+P+C+2CU
KM	SM+P+C+U8+2CU	SM+P+C+U20+1CU
SM	SM+P+C+2CU	SM+P+C+2CU
MM	NSM+P+C+U25	NSM+P+C+U40
SS	NSM+P+C+U80	NSM+P+C+U80
HM	SM+P+C+2CU	SM+P+C+2CU
II	SM+P+C+2CU	SM+P+C+2CU

TABLE 4
The Best, Worst, Medium Execution Time for Different Configurations, and the Execution Time of Our Predicted Configuration

	Worst	Best	Medium	Predicted
WC	1,269 ms	510 ms	810 ms	510 ms
KM	7,450 ms	1,131 ms	3,456 ms	1,872 ms
SM	506 ms	416 ms	470 ms	416 ms
MM	37.8 s	5.3 s	20.6 s	5.4 s
SS	21.2 s	2.5 s	9.6 s	2.5 s
HM	28.9 s	3.12 s	4.96 s	3.12 s
II	53.4 s	6.48 s	10.48 s	6.48 s

measuring the execution time of all possible configurations. Our model is able to match the best cases for the five applications (WC, SM, SS, HM and II). For MM and KM, the performance of the predicted configuration is comparable to or very close to the best case, as shown in the Table 4. More importantly, our prediction can effectively avoid the worst configuration, and significantly outperform the medium case in all applications.

4.4 Comparisons Between FPGA and CPU/GPU

We evaluate the execution time, and energy efficiency (performance per watt) of Melia, in comparison with its state-of-the-art counterparts on CPU/GPU. Note, we directly use the implementation [11], [12] from the author.

Comparisons with GPU. We show the ratios of Melia over the GPU-based counterpart [11], [12] on the execution time and energy efficiencies (with/without low-end CPU), as shown in Figs. 10a, 10b, and 10c. In particular, Melia achieves averagely 3.6 (2.1) times higher energy efficiency (performance per watt) than the GPU-based counterparts without (or with) low-end CPU. Due to the low power feature of the FPGA, Melia has a lower power consumption on all applications.

For the execution time, there is no conclusive comparison between FPGA and the GPU. On KM, Melia significantly outperforms the GPU-based MapReduce on all the two metrics, since the KM implementation utilizes the optimization methods: local memory and loop unrolling. In particular, FPGA is good for computation-intensive MapReduce applications with regular memory access pattern, since FPGA can provide multiple custom pipelines (via loop unrolling) to efficiently improve the computing ability and on-chip buffers to reduce global memory accesses. For example, KM can employ the loop unrolling to improve computation ability and on-chip buffers to reduce the number of memory accesses. Compared with the GPU-based counterpart, Melia achieves slower performance on other applications. Take SS and MM as examples. Melia fully utilizes the loop unrolling optimization. However, still many global memory transactions impede the further performance improvement since no dedicated cache is involved on the FPGA. In contrast, GPU is good for the computation-intensive application with irregular memory access pattern, since GPU has powerful computation ability and high memory bandwidth.

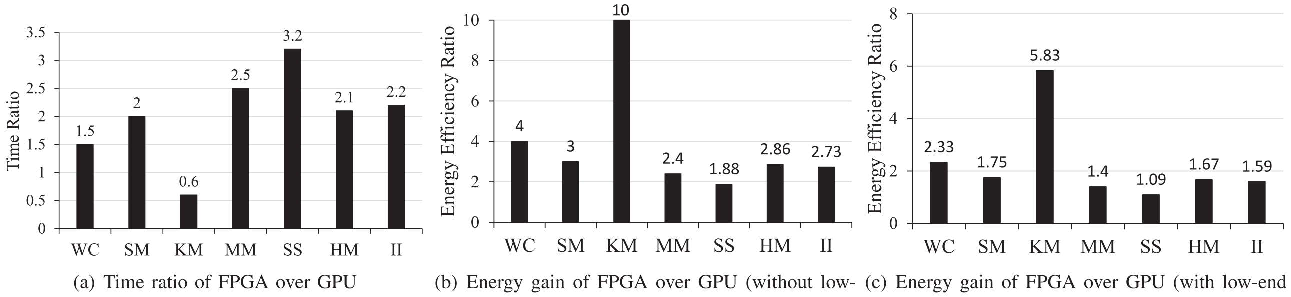
Comparisons with CPU. We present the overall comparison with the CPU-based MapReduce without figures. Previous studies [11], [12] have compared the MapReduce performance on the CPU and the GPU. Our results are consistent with their studies. Eventually, Melia has higher energy efficiency than the CPU-based MapReduce on all the seven applications, with the improvement of up to 16.7 times. In general, CPU is good for the control-intensive application, since CPU has powerful cache hierarchy and superscalar technology to reduce the latency of memory access.

For the seven MapReduce applications presented at our experiment, we summarize our findings as follows. First, FPGA is good for computation-intensive applications with regular memory access pattern, since FPGA can provide multiple custom pipelines to efficiently do the computation and on-chip buffers to efficiently read/write data. For example, KM can employ the loop unrolling to improve computation ability and on-chip buffers to reduce the number of global memory accesses. Second, GPU is good for the computation-intensive applications with irregular memory access patterns, since GPU has powerful computation ability and high memory bandwidth. For example, MM and SS requires the powerful computation ability to efficiently do the computation and requires high memory bandwidth to efficiently deal with a lot of global memory accesses. Third, CPU is good for the control-intensive applications, since CPU has powerful cache hierarchy and superscalar technology to reduce the average latency of memory access. For example, SM, WC, HM and II require powerful cache hierarchy and powerful superscalar technology to deal with plenty of branches.

4.5 Other Studies

In this section, we study the robustness of Melia in the following aspects.

Different data sizes. We also study the different data sets of the application (WC) for the case study. Fig. 11 shows the elapsed times of WC with input sizes (100, 200, . . . , 500,



(a) Time ratio of FPGA over GPU

(b) Energy gain of FPGA over GPU (without low-end CPU)

(c) Energy gain of FPGA over GPU (with low-end CPU)

Fig. 10. Comparison of Melia on FPGA over on GPU.

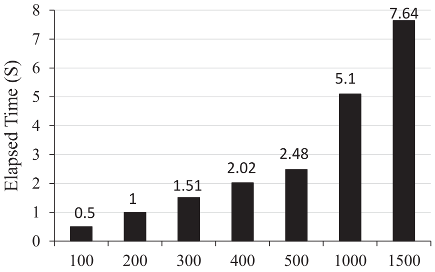


Fig. 11. WC with varying data sizes (MB).

1,000, 1,500 MB). The experimental result shows that the performance scales well for increasing data sizes.

Locking overhead. We also study the locking overheads of five MapReduce applications (WC, KM, SM, MM and SS) on OpenCL-based FPGAs. We estimate the locking overhead as subtracting the MapReduce application without locking operations from the same MapReduce application with locking operations. The time breakdown is shown in Fig. 12. The experimental result shows that the locking overhead is one important component of the total execution time for each MapReduce application, since FPGA cannot efficiently accommodate the standard locking mechanism (e.g., atomic_cmpxchg) from OpenCL specification.

Input data characteristics. We also study the impact of input data characteristics [2], [42] of the MapReduce application (WC) on FPGA/GPU, as shown in Fig. 13. In particular, we adopt the two cases of input data in the previous study: skewed key occurrence (SKO) and uniform key occurrence (UKO). The SKO is the case that the same key occurs consecutively, which implies that work items of MapReduce framework need to compete for the same lock (one distinct key has one corresponding lock). On the other hand, UKO is when keys uniformly appear, which implies that the possibility of lock contention is relatively low. Based on the experimental result, there are two observations. First, the input data with UKO has much better performance than that with SKO, since the lock contention is serious for SKO, which significantly degrades the performance of OpenCL-based Melia. Second, FPGA has significant performance advantage over GPU when the number of input distinct keys is small, since the lock-step execution model of GPU cannot efficiently address the serious lock contention, then work items actually execute sequentially.

When the number of distinct keys is known before MapReduce runtime performs, we can allocate proper FPGA on-chip buffer to store the reduction object and the proper hashing function can be used, so that FPGA on-chip buffer can be fully utilized. Then, the amount of FPGA resource can be reduced and more aggressive optimizations (e.g., more CU) can be applied to MapReduce programs. Take WC as an example, we can allocate three CUs for the implementation when the number (500) of input distinct keys is known before execution, then we get the performance improvement by 1.21X, compared with the default implementation with two CUs.

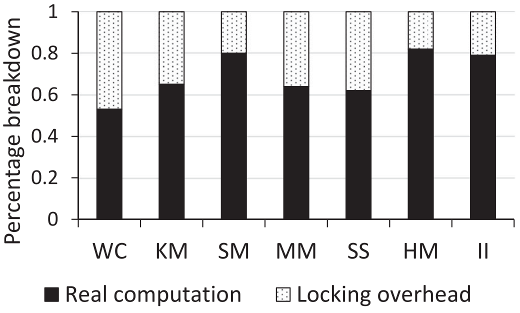


Fig. 12. Lock overheads for seven MapReduce applications.

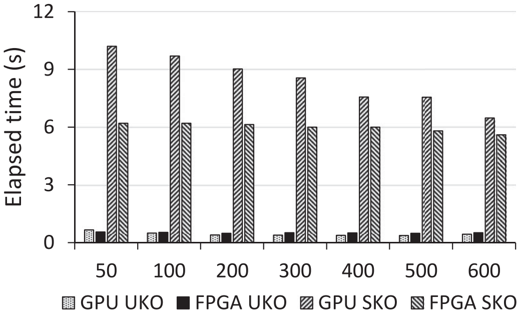


Fig. 13. Execution time for various number of distinct keys on FPGA and GPU.

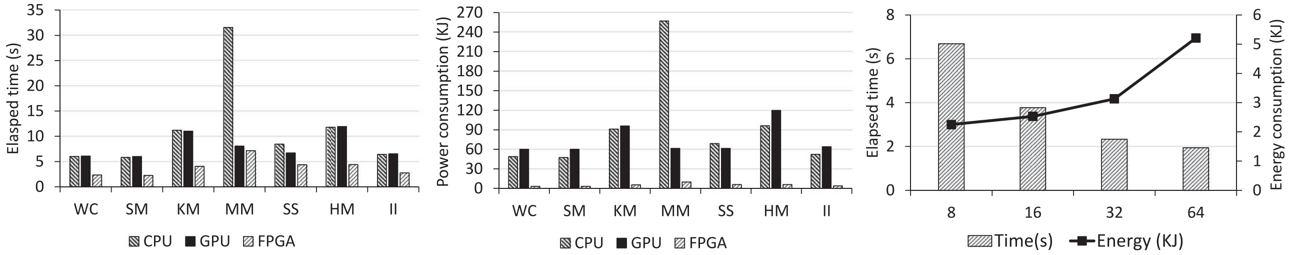
Comparison with direct HLS acceleration. We have compared the HLS enabled MapReduce runtime Melia with direct HLS acceleration. The implementation based on Melia requires more FPGA resources than the direct HLS acceleration. On the other hand, Melia improves the programmability so that the user only needs to implement two primitives (map and reduce), and MapReduce is able to exploit the parallelism in the underlying computing resources. Take MM with full optimizations for example. With Melia, the HLS enabled MapReduce roughly requires 10 percent more resources than the direct HLS acceleration, as shown in Table 5. The execution time of Melia (5.41 s) is much larger than that of HLS implementation (3.45 s) since the locking overhead of Melia is significant.

4.6 Finding Summary

Overall, FPGA demonstrates the significant energy efficiency, in comparison with its CPU- and GPU-based counterparts. The performance and energy consumption comparisons of FPGA-based MapReduce over the CPU/GPU-based MapReduce are resulted from the differences in the architectures as well as the algorithm design. First, the FPGA usually has much lower hardware frequency than CPU/GPU, respectively. In our experiments, the FPGA has the frequency of hundreds of MHz, while GHz for the CPU/GPU, respectively. Moreover, compared with CPU/GPU, FPGA does not have coherent cache hierarchy, e.g., L1/L2 caches. For some applications, Melia can still be faster than the MapReduce implementations on CPU/GPU, thanks to the FPGA-centric optimizations. Second, FPGA by design has much lower power consumption than CPU/GPU. This is a direct factor contributing to the superb energy efficiency of FPGA over CPU/GPU.

TABLE 5
Comparison with Direct HLS Acceleration (MM)

	LUTs	REGs	RAMs	DSPs	time
With Melia	179,630	273,103	1,886	32	5.41s
Direct HLS	160,480	244,187	1,657	32	3.45s



(a) Time consumption of CPU/GPU/FPGA

(b) Energy consumption of CPU/GPU/FPGA

(c) WC with varying FPGA nodes (8, 16, 32 and 64)

Fig. 14. Comparisons (time and power consumption) of Melia on CPU/GPU/FPGA clusters.

5 Extensions to Multiple FPGAs

Our extension (simulation) follows the common MapReduce design [15]. Many good mechanisms of MapReduce are inherited, including task scheduling and fault tolerance. Thus, we focus on how FPGAs are interconnected to make a large-scale system. While FPGAs can be integrated as a co-processor, we adopt a radical approach by viewing FPGAs as individual nodes. The Melia implementation on a single FPGA is used to process the map and the reduce tasks on a chunk of input data and a chunk of intermediate key-value list generated from the map task, respectively.

We design a FPGA-based computing cluster with master/slave architecture. The master node runs on a standard server, which is responsible for task scheduling and other management in MapReduce. Each slave node is a standalone FPGA board, which is plugged into one slot of a custom direct point-to-point backboard [35]. The backboard employs the high-speed Transceivers (MGTs) on the FPGA, named RocketIOs [5], to provide a custom high-speed data network. In particular, since MGT is full duplex and no software overhead is required, the data transfer bandwidth between any pair of two FPGA nodes at either direction can achieve 800 MB/s via 14.1 Gb/s transceiver. This is significant data transfer bandwidth advantage of FPGA over CPU/GPU. Dozens of FPGAs (16 in our performance/energy consumption analysis) forms a *pod*. All the FPGAs within a pod are fully connected via the backboard. To support a larger number of FPGAs, we leverage existing cluster network topologies [3], which connect pods with Ethernet switches in a tree-like network topology. Our cluster design is a hybrid one with both the features of FPGA backboard and Ethernet switches. For CPU/GPU-based cluster, we consider a common setting: a 10 Gb/s Ethernet switch within the pod of 16 machines each, and pods are connected with 10 Gb/s switch. The FPGA cluster uses the same cross-pod design. We use the power consumption model [6] for Ethernet switches. For example, an 10 Gb/s 32-port switch roughly consumes 786 Watts.

There are two issues that are worth discussion. The first one is on cost efficiency. The FPGA board used in the experiment costs 8,000 USD each, and the workstation costs 2,000 USD each. The FPGA board is more expensive than the server. In the real production environment, only the FPGA itself is required, rather than the entire FPGA board. Thus, the price per FPGA should be much lower than the FPGA board. Second, we adopt the fair scheduling policy in Apache Hadoop 2.5.1 – YARN, to handle job/task scheduling and fault tolerance. Both CPU/GPU- and FPGA-based clusters use the same policy in the simulation. Thus, we omit the experimental studies on those issues.

Simulation setup. We conduct the simulation about performance and energy consumption analysis according to the approach introduced by Lang et al. [26]. The basic idea is that, in the map phase, we consider the computation time of the map tasks; in the reduce phase, we estimate the time of network transfers required by the data shuffling and the computation time of the reduce tasks. For more details, we refer readers to the original paper [26].

We scale the data size by a factor ($\times f$, meaning that we scale the input data size or dimensions in Table 2 by f); that is, each node roughly has the same amount of data as shown in Table 2. We use the machine and FPGA setup in Section as the input hardware profile in the performance/energy consumption analysis.

Performance/energy analysis. Figs. 14a and 14b show the performance/energy consumption analysis results of Melia on CPU/GPU/FPGA clusters. The results are shown with 32 slave nodes (either FPGAs or servers with CPUs/GPUs) and the input data scale of (×32). Overall, in the cluster setting, seven MapReduce applications of Melia even more significantly outperforms its CPU/GPU counterparts in terms of performance and energy efficiency, in comparison with the results in Section 4. In particular, the performance of Melia is better than CPU/GPU cluster as show in Fig. 14a, since the RocketIO network in FPGA cluster can provide much more data transfer bandwidth than Ethernet of CPU/GPU cluster. Therefore, the time required for data shuffling in FPGA cluster is significantly less than that in CPU/GPU cluster. Furthermore, our FPGA cluster design has taken the backboard support of FPGAs, which eliminates the standard server components, which are required by the CPU/GPU cluster. Therefore, the energy consumption advantage of FPGA cluster over CPU/GPU cluster is much more significant than the performance advantage, as shown in Fig. 14b.

Scalability. We also study the impact of different FPGA nodes for the MapReduce application WC as the case study, as shown in Fig. 14c. The results are shown with varying slave nodes (8, 16, 32 and 64) and the input data scale of (×32). The experimental result shows that more FPGA nodes can have better performance, since the data set for each FPGA node is accordingly reduced. However, the cluster with more FPGA nodes may have more power consumption, consumed by more data shuffling between FPGA nodes.

6 Experiences and Open Problems

Our initial studies show a few opportunities for further improving the performance and energy efficiency of MapReduce on FPGAs.

First, with OpenCL abstractions, FPGAs can be viewed as a highly parallel architecture with strong and efficient support on hardware pipeline executions. This fits extremely well with massively parallel processing like MapReduce. The fast inter-"thread" communication within the same hardware pipeline can significantly accelerate the performance and ease the programming.

Second, the FPGA programmability for more complex applications has been improving greatly. Besides Altera OpenCL SDK, Xilinx C/C++ HLS tools significantly reduce the programming complexity on FPGAs.

Third, as energy efficiency has a more significant role in system designs, FPGAs are more likely to become an important citizen in MapReduce, and other data processing systems. Through proper optimizations, we demonstrate that FPGAs achieve significantly higher energy efficiency than CPUs/GPUs, with slight performance degradations or even better performance on FPGAs.

We have also identified a few open problems:

First, MapReduce in specific and data processing in general are complex in its runtime logic. Even though FPGAs have low power, we still require a significant amount of design and implementation effort to further improve the performance and energy efficiency of Melia.

Second, even with OpenCL abstraction, reconfigurable computing still has other challenges. More advanced system features such as the partial reconfiguration capability is still preliminary [8]. Also, as our experiments show, memory stall optimizations and pipeline execution efficiency are two most important performance factors. For example, the hardware reconfigurable capability also requires careful algorithmic designs, since even the unexecuted code in runtime has to consume resources on FPGA. FPGAs now do not offer coherent cache memory hierarchy. The locality and coherency are left to programmers.

Third, similarly to GPU, FPGA is relatively weak on synchronization handling and memory subsystems (no cache coherence). For example, we found that the atomic-lock seriously affect performance. It is desirable to develop software or hardware techniques to improve those issues on FPGAs.

7 Conclusion

MapReduce has become a popular programming framework in parallel architectures. In this paper, we implement and evaluate an OpenCL-based MapReduce framework (*Melia*) with a series of optimizations for FPGAs, based on the recently released Altera OpenCL SDK. We evaluate Melia on a recent Altera FPGA. Our evaluations show that memory stalls and pipeline execution efficiency have significant impact on the overall performance and energy efficiency of FPGAs. Our results demonstrate that 1) our parameter setting approach can predict the suitable parameter settings that have the same or comparable performance to the best setting, 2) our FPGA-centric optimizations significantly improve the performance of Melia on FPGA with an overall improvement of 1.4-43.6 times over the baseline on FPGA. Both real experiments on a single FPGA and performance/energy consumption analysis on a cluster setting demonstrate the significant performance and energy efficiency improvement of Melia over its CPU/GPU-based counterparts.

One interesting future direction is to schedule the execution among heterogeneous environments (including FPGAs, GPUs and CPUs), and to extend the methodology to general OpenCL programs. We have made Melia open-sourced in http://www.ntu.edu.sg/home/bshe/Melia.html.

Acknowledgments

We thank Altera University Program for their kind support in our research. This work is in part supported by MoE AcRF Tier 2 grants (MOE2012-T2-1-126 and MOE2012-T2-2-067) in Singapore.

References

[1] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar. (2012). Puma: Purdue MapReduce benchmarks suite. Technical report, Purdue University, IN, USA. [Online]. Available: http://core.ac.uk/download/pdf/10238137.pdf

[2] S. Ahmed and D. Loguinov, "On the performance of MapReduce: A stochastic approach," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2014, pp. 49–54.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun.*, 2008, pp. 63–74.

[4] (2013). Altera, "Altera SDK for OpenCL optimization guide," [Online]. Available: www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[5] (2014). Altera, " Stratix v device overview," [Online]. Available: www.altera.com/literature/hb/stratixv/stx5_51001.pdf

[6] G. Ananthanarayanan and R. H. Katz, "Greening the switch," in *Proc. Conf. Power Aware Comput. Syst.*, 2008, p. 8.

[7] O. Arnold, S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner, "An application-specific instruction set for accelerating set-oriented database primitives," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 767–778.

[8] C. Beckhoff, D. Koch, and J. Torresen, "Migrating static systems to partially reconfigurable systems on spartan-6 FPGAs," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2011, pp. 212–219.

[9] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2014, pp. 151–160.

[10] D. Chen and D. Singh, "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms," in *Proc. 18th Asia South Pacific Des. Autom. Conf.*, 2013, pp. 297–304.

[11] L. Chen and G. Agrawal, "Optimizing MapReduce for GPUs with effective shared memory usage," in *Proc. 21st Int. Symp. High Perform. Distrib. Comput.*, 2012, pp. 199–210.

[12] L. Chen, X. Huo, and G. Agrawal, "Accelerating MapReduce on a coupled CPU-GPU architecture," in *Proc. Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, p. 25.

[13] J. Costabile, "Hardware acceleration for MapReduce analysis of streaming data using OpenCL," Syncopated Engineering Inc., Tech. Rep. DS-1001, Altera, 2015.

[14] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, "From OpenCL to high-performance hardware on FPGAs," in *Proc. 22nd Int. Conf. Field Programmable Logic Appl.*, 2012, pp. 531–534.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in presented at the 6th Symp. Operating System Design and Implementation, San Francisco, CA, USA, 2004.

[16] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with graphics processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 608–620, Apr. 2011.

[17] (2015). IBM Software, "IBM Netezza analytics the advanced analytics platform inside every IBM Netezza appliance," IMD14365-USEN-03, http://zementis.com/wp-content/uploads/2015/06/IMD14365USEN.pdf

[18] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.

[19] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: writing parallel program portable between CPU and GPU," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 217–226.

[20] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 152–163.

[21] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int.Conf. Data Eng. Workshops*, 2010, pp. 41–51.

[22] Z. Istvan, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10GBPS key-value stores on FPGAs," in *Proc. 23rd Int. Conf. Field Programmable Logic Appl.*, 2013, pp. 1–8.

[23] W. Jiang and G. Agrawal, "MATE-CG: A MapReduce-like framework for accelerating data-intensive computations on heterogeneous clusters," in *Proc. IEEE 26th Int. Symp. Parallel Distrib. Process.*, 2012, pp. 644–655.

[24] (2009). Khronos OpenCL Working Group, "The OpenCL specification, v1.1.48," [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[25] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Found. Trends Electron. Des. Autom.*, vol. 2, no. 2, 2008.

[26] W. Lang, S. Harizopoulos, J. M. Patel, M. A. Shah, and D. Tsirogiannis, "Towards energy-efficient database cluster design," in *Proc. Very Large Databases Endowment*, 2012, pp. 1684–1695.

[27] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with MapReduce: A survey," *ACM SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, 2012.

[28] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed data management using MapReduce," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 31–42, Jan. 2014.

[29] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. Goh, "MrPhi: An optimized MapReduce framework on intel xeon phi coprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 11, pp. 3066–3078, Nov. 2015.

[30] S. McBader and P. Lee, "An FPGA implementation of a flexible, parallel image processing architecture suitable for embedded vision systems," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2003, pp. 22–26.

[31] R. Mueller and J. Teubner, FPGA: What's in it for a database? in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 999–1004.

[32] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAs," in *Proc. Int. Conf. Very Large Databases*, 2009, pp. 910–921.

[33] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: A query compiler for FPGAs," in *Proc. Very Large Databases Endow.*, 2009, pp. 229–240.

[34] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. 13th Int. Symp. High-Perform. Comput. Archit.*, 2007, pp. 13–24.

[35] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, and P. Beeraka, "Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing," in *Proc. IEEE 15th Annu. Symp. Field-Programmable Custom Comput. Mach.*, 2007, pp. 127–140.

[36] O. Segal, M. Margala, S. R. Chalamalasetti, and M. Wright, "High level programming for heterogeneous architectures," in *Proc. FPGAs Softw. Program.*, 2014, pp. 41–51.

[37] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, "FPMR: MapReduce framework on FPGA," in *Proc 18th Annu. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2010, pp. 93–102.

[38] Altera, "Implementing FPGA Design with the OpenCL Standard, Whitepaper 01173," [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf

[39] S. Kestur, J. D. Davis, and O. Williams, "Blas comparison on FPGA, CPU and GPU," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2010, pp. 288–293.

[40] J. Teubner, R. Mller, and G. Alonso, "FPGA acceleration for the frequent item problem," in *Proc. IEEE 26th Int. Conf. Data Eng.*, 2010, pp. 669–680.

[41] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 625–636 .

[42] D. Tiwari and D. Solihin, "Modeling and analyzing key performance factors of shared memory MapReduce," in *Proc. IEEE 26th Int. Symp. Parallel Distrib. Process.*, 2012, pp. 1306–1317.

[43] K. H. Tsoi and W. Luk, "Axel: A heterogeneous cluster with FPGAs and GPUs," in *Proc. 18th Annu. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2010, pp. 115–124.

[44] Z. Wang, B. He, and W. Zhang, "A study of data partitioning on OpenCL-based FPGAs," in *Proc. 25th Int. Conf. Field Programmable Logic Appl.*, 2015, pp. 1–8.

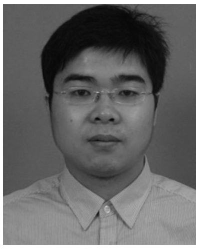
[45] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *Proc. Int. Symp. High-Performance Comput. Archit.*, 2016, pp. 114–125.

[46] L. Woods, Z. István, and G. Alonso, "Ibex:An intelligent storage engine with support for advanced SQL off-loading," in *Proc. Very Large Databases Endowment*, 2014, pp. 963–974.

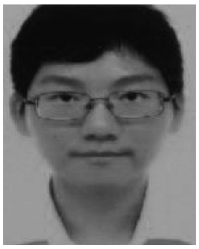
[47] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with FPGAs," in *Proc. Very Large Databases Endowment*, 2010, pp. 660–669.

[48] J. Yeung, C. Tsang, K. Tsoi, B. Kwan, C. Cheung, A. Chan, and P. Leong, "MapReduce as a programming model for custom computing machines," in *Proc. 16th Int. Symp. Field-Programmable Custom Comput. Mach.*, 2008, pp. 149–159.

[49] Y. Zhang and J. D.Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, 2011, pp. 382–393.



Zeke Wang received the BSc degree from Harbin University of Science and Technology, China, in 2006 and the PhD degree from Zhejiang University, China, in 2011. He is a research fellow at Parallel Distributed Computing Center, School of Computer Engineering, Nanyang Technological University. His current research interests include heterogeneous computing (with a focus on FPGA) and database systems.



Shuhao Zhang received the bachelor's degree from Nanyang Technological University in 2014. He is currently working towards the PhD degree in the Department of Computer Science and Engineering, Nanyang Technological University. His major research interests include high performance computing, stream processing, parallel and distributed systems.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University, in 2003, and the PhD degree in computer science in Hong Kong University of Science and Technology, in 2008. He is an associate professor in School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Wei Zhang received the PhD degree from Princeton University, Princeton, NJ, in 2009. She joined Hong Kong University of Science and Technology in 2013 as an assistant professor and established Reconfigurable System Lab. She was an assistant professor in School of Computer Engineering at Nanyang Technological University, Singapore from 2010 to 2013. She has authored and co-authored more than 60 book chapters and papers in peer-reviewed journals and international conferences. Her current research interests include reconfigurable system, FPGA-based design, low-power high-performance multicore system, embedded system security and emerging technologies.