Multikernel Data Partitioning With Channel on OpenCL-Based FPGAs

Zeke Wang, Johns Paul, Bingsheng He, and Wei Zhang

***Abstract*— Recently, field-programmable gate array (FPGA) vendors (such as Altera) have started to address the programmability issues of FPGAs via OpenCL SDKs. In this paper, we analyze the performance of relational database applications on FPGAs using OpenCL. In particular, we study how to improve the performance of data partitioning, which is a very important building block in relational database. Since the data partitioning causes random memory accesses, it is time-consuming, and then, it has been the major bottleneck for database operators, such as partitioned hash join. In particular, we import the state-of-the-art OpenCL implementation of data partitioning from OmniDB, which was originally designed and optimized for CPUs/GPUs, and we find that this implementation suffers from both lock overhead and memory bandwidth overhead. Accordingly, we present a multikernel approach to address the lock overhead by leveraging two emerging features (*task kernel* and *channel*) from Altera OpenCL software development kit. In order to reduce the memory bandwidth overhead, on-chip buckets are used to reduce the number of random global memory transactions. We further develop an FPGA-specific cost model to guide the parameter configuration. We evaluate the proposed design on a recent OpenCL-based FPGA. We have applied our optimized partitioning method to a number of data processing tasks, including hash join, histogram, and hash search. Our experimental results demonstrate that our cost model can accurately guide the user to determine the optimal parameter combination for data partitioning and the optimal parameter combination can achieve 16.6× speedup over the default multithreaded implementation.**

***Index Terms*— Channel, data partitioning, database, field-programmable gate array (FPGA), high-level synthesis, OpenCL.**

I. Introduction

RECENTLY, FPGAs have become an effective means of accelerating relational database applications due to their high-throughput and low-power consumption. Meanwhile, It has been a fruitful research direction to leverage FPGAs to improve the performance of database applications [3], [4], [6], [11], [12], [14]. Furthermore, more and more efforts have been devoted to this direction. However, those previous studies suffer from the programmability issues, which raise serious concerns on the long learning curve and code maintenance. Fortunately, high-level synthesis (HLS) has been used to address the programmability issues. For example, field-programmable gate array (FPGA) vendors, such as Altera [7], have provided OpenCL SDKs for much better programmability of FPGAs. There have been some initial studies on the performance of OpenCL programs on such FPGAs [16]. At the same time, OpenCL-based relational databases [9], [10], [19] have already designed and implemented on CPUs/GPUs. Therefore, a natural question to be asked is how those OpenCL-based databases perform on such FPGAs, whose architecture is significantly different from that of CPUs/GPUs.

The research of relational databases on OpenCL-based FPGAs is still a largely open and challenging problem. As a start, on OpenCL-based FPGAs, we study the performance of data partitioning, which is an important building block of relational databases and other data processing tasks. Given an input table, the data partitioning operation is used to divide the input table into a number of partitions according to the input partitioning criteria (e.g., hash function).

We import the state-of-the-art OpenCL implementation of data partitioning from OmniDB [9], [19], which utilizes a lock-based implementation; more details regarding this implementation can be found in Section III. Though it can achieve good performance on CPUs/GPUs, we find that its performance on OpenCL-based FPGAs is far from ideal due to the severe lock overhead and memory bandwidth overhead.

To reduce lock overhead, we leverage two emerging features (namely *task kernel* and *channel*) to develop an efficient multikernel approach, which follows a producer–consumer paradigm, where producer stage and consumer stage are connected via channel. In particular, the consumer stage contains multiple processing units to do the partitioning concurrently, and each processing unit, which is responsible for a portion of partitions requires multiple cycles to handle one tuple. Furthermore, the producer stage can dispatch one tuple per cycle to one of processing units, as shown in our previous work [15]. Therefore, it can achieve much better performance than the original multithreaded implementation.

On the tested FPGA board in our experiment, the practical memory bandwidth of random memory accesses is linear to the data access unit size, as shown in Section III-B, since there is no cache hierarchy on FPGAs. Due to the random memory accesses, data partitioning is very time-consuming on OpenCL-based FPGAs. To reduce memory bandwidth overhead, on-chip buckets are employed to combine multiple single-tuple memory transactions into one multituples transaction, thus significantly improving the memory performance.

Manuscript received August 12, 2016; revised November 17, 2016; accepted December 30, 2016. Date of publication February 15, 2017; date of current version May 22, 2017. This work was supported in part by the NUS startup grant and in part by MoE AcRF Tier 1 grant in Singapore under Grant T1 251RES1610. The work of W. Zhang was supported by HKUST startup grant in Hong Kong under R9336.

Z. Wang and B. He are with the National University of Singapore, Singapore 119077. (e-mail: wangzeke638@gmail.com).

J. Paul is with Nanyang Technological University, Singapore 639798.

W. Zhang is with The Hong Kong University of Science and Technology, Hong Kong.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2653818

After employing the above-mentioned two optimizations, there is still space for the further performance improvement of the above producer–consumer design. For example, when the number of processing units (e.g., 16) is large enough at the consumer stage, the issue rate of the producer stage of our previous work [15] can be the new performance bottleneck. Therefore, we present a converging technology (i.e., *Converge* kernel) to collect tuples simultaneously from multiple channels and then send one tuple per cycle to the output channel.

Since each of the above-mentioned optimizations (for example, multiple processing units at consumer stage) requires the FPGA resources to implement, an FPGA-specific cost model is required to determine the optimization combination to maximize the data partitioning performance on OpenCL-based FPGAs, given the budget of FPGA resources.

We evaluate the proposed multikernel design on an Altera Stratix V GX FPGA. Our experimental results demonstrate that: 1) our cost model can roughly predict the performance of data partitioning with different parameter combinations and 2) our optimal parameter combination can achieve 16.6× speedup over the existing lock-based implementation.

The remainder of this paper is organized as follows. In Section II, we introduce the backgrounds about OpenCL-based FPGA and data partitioning. In Section III, we present the motivations of this paper, followed by the design methodology in Section IV, the proposed design in Section V, and the cost model in Section VI. We present the experiment results in Section VII and the conclusion in Section VIII.

II. Background and Related Work

A. Altera's OpenCL Architecture

Recently, Altera provides the OpenCL software development kit (SDK) [7] to abstract the hardware complexities from the FPGA programmer. In particular, the Altera's SDK can translate the OpenCL kernel to low-level hardware implementation by creating the dedicated circuits for each instruction of the kernel and pipeline them together to achieve the complete data path.

From the perspective of OpenCL, the memory component of OpenCL-based FPGAs contains three layers. First, the *global memory* resides in DDRs with long-latency global memory access. Second, the *local memory* is low latency and high bandwidth. On our tested FPGA board, it is implemented by on-chip memory with four read/write ports. Third, the *private memory*, storing the variables or small arrays, is implemented using completely parallel registers. Compared with CPU/GPU, FPGA has a sufficient number of registers, which should be employed to store intermediate results for each work item (executing an instance of the kernel) for efficiency.

The OpenCL kernel [1] has two types: NDRange kernel and task kernel.

1) NDRange Kernel: NDRange kernel is the default OpenCL kernel model, which achieves the pipelined parallelism by executing the kernel in terms of multiple work items. Fig. 1(a) shows the pipelined parallelism with the example of a simplified vector addition example [7], where each work item executes one addition operation out of the total eight addition operations, with the throughput of one work item finished per cycle. We can configure multiple compute units (CUs) for the NDRange kernel to increase overall kernel throughput, as shown in Fig. 1(b). In particular, all the CUs can execute in parallel, and each CU is dispatched by the hardware scheduler with different work groups, where work items can share the local memory and make progress in the presence of barriers. Each CU has its own local memory, while all the CUs share the global memory. The on-chip local memory is low latency and high throughput. Thus, the local memory should be employed whenever possible to reduce global memory accesses. One main disadvantage of NDRange kernel is that the atomic operations are required when multiple work items attempt to update shared data structures.

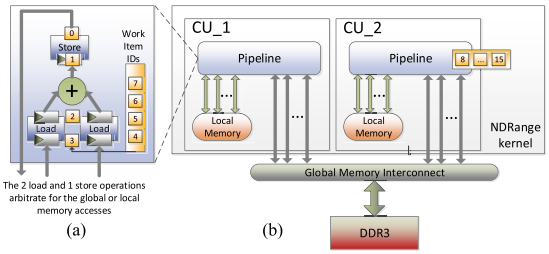


Fig. 1. Architecture of altera NDRange kernel. (a) One CU with eight work items. (b) One kernel with two CUs.

2) Task Kernel: It can execute the kernel on only one CU that contains only one work item. It follows a sequential model like C programming, and the OpenCL SDK determines the degree of parallelism at the compile time [1] based on the inner dependence. The task kernel is preferred in cases where the fine-grained data are shared among many work items, since expensive atomic operations from the NDRange kernel are required to keep the correctness of fine-grained data. Thus, it is the developer's responsibility to extract the parallelism from task kernel, while the parallelism of NDRange kernel is explicitly achieved via multiple work items.

a) Channel: Provided by the Altera OpenCL SDK [1], the important feature (channel) can be used to efficiently pass data (at the *private memory* level) between two OpenCL kernels (either NDRange or task), while in traditional OpencL-based systems, the communication between two kernels are executed via the global memory in the conventional OpenCL implementation. The channel, with a channel ID and buffer depth [e.g., channel depth (*CD*)], is implemented with on-chip first-in-first-out (FIFO) buffer. The channel has two types: blocking channel and nonblocking channel. The write/read operation to blocking channel (using the API: write_channel/read_channel) will not return if the operation does not successfully commit, while the write/read operation to nonblocking channel (using the API: write_channel_nb/read_channel_nb) will return even when the operation does not successfully commit.

With the HLS on FPGAs, the recent studies [5], [16]–[18] have gained a lot of attention in accelerating different kinds of applications.

B. Data Partitioning

Its functionality is to divide the input table into a number of disjoint partitions according to the input *partitioning function*.

Algorithm 1 Lock-Based Data Partitioning

```
   Input  : data_in(Input table in global memory),
            counters (Counters in the local memory for each partition),
            N(Number of input tuples)
   Output : data_out(Partition output address in global memory)
1  gid = get_global_id(0);
2  gsize = get_global_size(0);
3  for (i = gid; i < N; i += gsize ) do
4      tuple = data_in[i];
5      index = hash(tuple.key);
       /* Wait until acquiring global/local lock[index]   */
6      get_lock(index);
7      counter_index = counters[index]++;
8      data_out[counter_index] = tuple;
       /* Release the global/local lock[index]            */
9      release_lock(index);
10 end
```

Then, each tuple (one row of the input table) will be stored in the corresponding output partition.

Data partitioning is widely used as a building block in relational database applications. For example, the partitioned hash join is one of the most efficient hash join algorithms [9]. There have been optimizations for data partitioning on the CPU/GPU. He *et al.* [8] proposed optimized scatter and gather operations to improve the locality of data partitioning on the GPU. This paper goes beyond our previous work in three major ways. First, we further proposed a new multikernel approach to adapt the throughput among producer and consumer kernels. Second, we have studied the data partitioning in three case studies, including hash join, histogram, and hash search. Third, we have conducted more robust experiments including cost model evaluations.

III. Motivation

In this section, we begin with an NDRange-kernel-based implementation of the data partitioning on OpenCL-based FPGAs. We focus on analyzing two key overheads, including lock overhead and memory bandwidth overhead, with the detailed experimental setup in Section VII-A.

A. Lock Overhead

The conventional single-kernel implementation of data partitioning achieves the pipelined parallelism via multiple work items. Since work items could be in contention for the same partition at the same cycle, the lock mechanisms (implemented with atomic operations) are used to guarantee the consistency among work items. In particular, we maintain an array of locks, and perform acquire/release lock operations on the index of the lock in the array. The lock array can be implemented in global memory or in local memory with the tradeoff as discussed in the following. The locks implemented in global memory (or local memory) are referred as *global locks* (or *local locks*). The detailed implementation is shown in Algorithm 1.

1) Single-Kernel Partitioning With Global Lock: Algorithm 1 shows the single-kernel implementation of partitioning, where the lock means global lock. Each tuple should first acquire the global lock according to the hash value (Line 6), second, write to the corresponding partition (Lines 7 and 8), and, third, release the global lock (Line 9). One potential optimization technology is to employ multiple CUs. In particular, dividing the work items into multiple CUs will enable parallel execution and allow more concurrent accesses to the shared partitions. However, the shared partitions are located in the global memory and global locks are required by all the work items to guarantee the consistency, incurring long access latency. This implementation (with minor modification to avoid deadlock due to lockstep execution) can achieve good performance on GPU, since GPU contains powerful memory hierarchy to support multiple processing units. However, the memory hierarchy of our FPGA board is not as powerful as that of GPU.

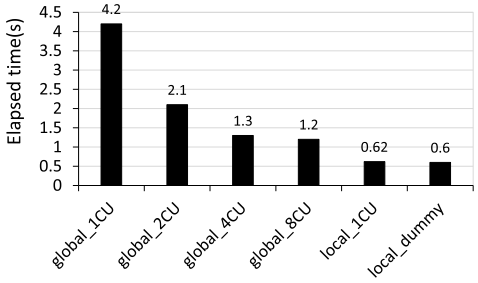


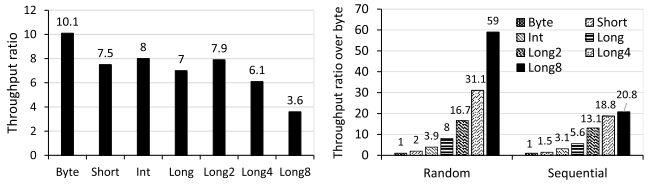
Fig. 2. Performance of single-kernel partitioning with global/local locks.

2) Single-Kernel Partitioning With Local Lock: Since the atomic operation on long-latency global memory may be the main performance bottleneck for single-kernel partitioning on OpenCL-based FPGAs, we try to relocate the atomic operation from global memory to local memory. Since the local memory is private to each work group, only one work group can be launched to execute the partitioning on FPGAs. In particular, each tuple acquires the local lock according to the hash value, writes out to the corresponding partition, and then releases the corresponding local lock.

We compare the performance of single-kernel data partitioning with global and local locks, as shown in Fig. 2, where *global_xCU* indicates the partitioning implementation (with x CUs) using global locks and $x = \{1, 2, 4, 8\}$. *local_1CU* indicates the partitioning implementation with local lock, and *local_dummy* means the microbenmark, which only acquires and releases local locks (Lines 6 and 9) without executing the tuple-related instructions. Note, when we choose the local lock, only one CU and one work group can be used. Otherwise, the consistency is not guaranteed. From Fig. 2, we obtain two observations as follows.

Observation 1: Performance of partitioning with global lock is worse than that with local lock. In particular, the performance of *global_8CU* (the best case with global lock) is slower than *local_1CU*, since the performance of local lock is much better than that of global lock and the overhead of global lock cannot be compensated by using more CUs.

Observation 2: Performance of local_dummy is roughly the same as that of local_1CU. It means that the lock processing ability dominates the performance of data partitioning, and the lock processing ability in the original single-kernel implementation should be significantly improved to accelerate the performance of partitioning. Therefore, we resort to the proposed multikernel approach, which follows a producer–consumer paradigm.



(a) (b)

Fig. 3. Memory bandwidth characteristics. (a) Sequential over random. (b) Trend with different granularities.

B. Memory Bandwidth Overhead

Since the partitioning is a memory-intensive operation containing plenty of random memory accesses, we need qualitatively to analyze the performance characteristics of sequential and random global memory accesses on FPGAs. Furthermore, we need to develop an FPGA-specific cost model to accurately predict the performance of memory subsystem. Each scan with different data types (e.g., short and long4) has four independent read instructions to saturate the memory subsystem. Two observations about the memory characteristics are shown as follows.

Observation 3: The practical memory bandwidth of sequential memory access is much higher than that of random access. Fig. 3(a) shows the throughput ratio of sequential memory access to random memory access. We can see that the sequential memory access has the significant throughput advantage over the random memory access, since almost each random memory transaction can cause a row-buffer miss [13], which severely degrades the performance of overall memory subsystem.

Observation 4: The random memory access throughput is more sensitive to the data access unit size than the sequential memory access. Fig. 3(b) shows the throughput ratio of the sequential (or random) memory access with different data access unit sizes (e.g., Int and Long8) over that of the sequential (or random) memory access with byte. One interesting finding here is that the memory bandwidth of random memory access is roughly proportional to the data access unit size. The reason is that each random memory access can generate one real memory transaction and cause one row-buffer miss to memory subsystem. Hence, the number of memory transactions directly determines the global memory performance. In contrast, the sequential memory access will generate much fewer row-buffer misses. The speedup trend of sequential access is much more flat when the data access unit size increases, and finally approaches the bandwidth limitation of the memory subsystem. Hence, since data partitioning has a random memory access pattern, large data unit size should be used to efficiently utilize the global memory bandwidth on OpenCL-based FPGAs.

IV. Design Methodology

In Sections IV–VI, we will present the overall design methodology for the data partitioning, followed by the implementation details of the multikernel design for data partitioning and the corresponding cost model.

A. Addressing the Challenge From Lock Overhead

In order to address the issue about severe lock overhead due to atomic operations in the NDRange kernel, the multikernel approach (with producer–consumer model) is employed to improve the locking processing performance. However, the producer stage can deliver multiple tuples to the consumer stage per cycle, while each consumer kernel requires seven cycles to absorb one tuple generated at the producer stage. To resolve this throughput mismatch between producer stage and consumer stage, we allocate multiple consumer kernels for the consumer stage. Then, each of consumer kernels handles the one part of total partitions based on the partitioning function. Using this design, all the consumer kernels of the consumer stage execute concurrently to reduce the lock overhead.

When there are a sufficient number of consumer kernels at the consumer stage, the issue rate of the producer stage in our previous work [15] can be the new performance bottleneck, since it can only deliver one tuple per cycle. In this paper, the proposed producer stage, integrated with the *Converge* kernel, can issue multiple tuples per cycle. Therefore, the performance of data partitioning on OpenCL-based FPGAs is significantly improved.

B. Addressing the Challenge From Memory Overhead

In order to address the second challenge of severe global memory bandwidth overhead, we employ on-chip buffers to reduce the number of global memory transactions in all the consumer kernels of the consumer stage. Another potential benefit of the multikernel design is that the required amount of local memory can be distributed among the consumer kernels. This tends to achieve a higher frequency than that of one large on-chip buffer.

Both optimizations require certain amounts of FPGA resources (e.g., multiple consumer kernels). Especially, the design with on-chip buffers requires a large number of block RAMs. Hence, how to efficiently utilize the limited FPGA resources is critical. Therefore, we present a cost model to guide our design of multikernel partitioning on FPGAs, as shown in Section VI.

V. Implementation Details of Partitioning

Motivated by the above-mentioned design methodology, we present a multikernel design of data partitioning by leveraging two emerging features (*task kernel* and *channel*).

A. Overall Architecture

The proposed architecture of multikernel partitioning comprises two stages: producer stage and consumer stage, as shown in Fig. 4. The producer stage reads each input tuple from global memory and then delivers it (via channel) to the corresponding intermediate buffer, which connects to one kernel in the consumer stage. The kernel in the consumer stage reads the tuple from the intermediate buffer (via channel) and then do the partitioning task.

The producer stage, containing one *Data_in* and $DO + 1$ *Converge* kernels, can issue IR tuples per cycle. In particular,

TABLE I
SUMMARY OF THE PARAMETERS

Model Parameter	Definition	Range on our tested FPGA board
N	Number of tuples of data partitioning	Input
P	Number of partitions for data partitioning	Input
IR	Issue rate of the *Data_in* kernel	1, 2, 4
W	Number of tuples read from one each memory transaction in the *Data_in* kernel	64/tuple size
CD	Depth of buffered channel	0, 2, 4, 8, 16, 32
DO	Number of *Data_out* kernels in the consumer stage	1, 2, 4, 8, 16
S_H	Number of cycles consumed by one tuple in one *Skewed_handling* kernel	1
L	Number of cycles consumed by one tuple in one *Data_out* kernel	7
S	Slot size of tuples for each on-chip bucket	1, 2, 4, 8, 16, 32
B	Number of buckets in the *Data_out* kernel	$\frac{P}{DO}$
TPC	Number of global memory transactions served per cycle by memory sub-system	$\leq \#MemoryBanks$

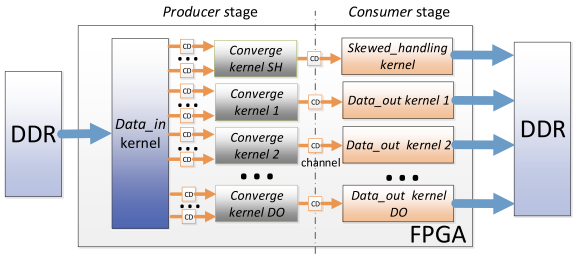


Fig. 4. Architecture of multikernel partitioning with channel.

the *Data_in* kernel (producer) requires $DO + 1$ channels to dispatch one tuple to the consumer stage per cycle, and each channel is dedicated to the *Skewed_handling* kernel or one of the *Data_out* kernels (consumer). It means that only one channel can get the issued tuple and the other channels are idle, as described in our previous work [15]. For example, when IR is 1 and DO is 8, there are nine (8 + 1) channels written by the *Data_in* kernel at the producer stage and only one of the nine consumer kernels has the chance to receive the issued tuple from the producer stage via its dedicated channel within each cycle.

In order to issue IR tuples per cycle for the *Data_in* kernel, $IR \times (DO + 1)$ channels are required by the *Data_in* kernel. The reason is that it is possible for all the IR tuples to be sent to the same consumer kernel within one cycle and then each consumer kernel requires IR channels to receive the potential IR tuples to avoid the potential conflict. Since all the $DO+1$ consumer kernels share the IR tuples, each consumer kernel averagely can receive $IR/(DO + 1)$ tuples within each cycle. For example, when IR is 2 and DO is 8, there are $2 \times (8 + 1)$ channels written by the *Data_in* kernel in the producer stage.

We need to reduce the design complexities of consumer kernel, so that the consumer kernel can read the tuple from one input channel, not the default IR channels. Therefore, the *Converge* kernel is imported to combine the tuples from IR input channels (from the *Data_in* kernel) into the output channel (to the consumer kernel), as shown in Fig. 4.

The consumer stage contains one *Skewed_handling* and DO *Data_out* kernels. The consumer kernel reads from its input channel connected the corresponding *Converge* kernel at the producer stage, and then stores the received tuples in on-chip buffers, so that multiple tuples can be stored in global memory using a large memory transaction.

In our multikernel partitioning, the buffered channels with the buffer depth (CD) are used to decouple the execution between producer and consumer kernels. The implementation details of *Data_in*, *Converge*, *Data_out*, and *Skewed_handling* kernels are given in the following.

Algorithm 2 Data_in Kernel

```
   Input  : data_in (input table in global memory),
            N (number of input tuples),
            skewed_index (index of partition with the skewed data),
            DO (number of Data_out kernels)
1  for (k ← 0 to N/W ) do
       /* Load W tuples for efficient DDR utilization   */
2      tuples[W] = data_in[k × W];
3
4      #pragma unroll IR
5      for (i ← 0 to (W − 1) ) do
6          key = channel_hash(tuples[i]);
7          partition_index = partition_hash(tuples[i]);
8
9          if (partition_index == skewed_index) then
10             write_channel(C_IN_SKEW[i%IR], tuples[i]));
11         end
12         else
13             #pragma unroll
14             for (j ← 0 to (DO − 1) ) do
15                 if (key == j) then
16                     write_channel(C_IN_PAR[key][i%IR], tuples[i]);
17                 end
18             end
19         end
20     end
21 end
```

B. Design Details of Data_in Kernel (Producer)

One design goal of the *Data_in* kernel is to dispatch IR input tuples to $DO+1$ consumer kernels per cycle. Therefore, this kernel contains $(DO+1)\times IR$ channels, with IR channels dedicated to each consumer kernel. The other design goal is to fully utilize the global memory bandwidth. The *Data_in* kernel sequentially loads the tuples from global memory and then issues each tuple for the further processing, as shown in Algorithm 2. The *Data_in* kernel has sequential global memory access pattern, and hence, it is suitable to efficiently utilize the global memory bandwidth. Next, we demonstrate the exact work process of this kernel.

In the beginning, it loads the W tuples for each global memory transaction (Line 2). In the next W/IR cycles, IR tuples are issued to the consumer stage within each cycle (Lines 4–20), with the help of the directive *#pragma unroll IR*. Each of IR tuples contains $DO+1$ dedicated channels to issue and only one channel can receive the tuple.

For the ith tuple, the index (key) of *Data_out* kernel and the index of the partition ($partition\_index$) in any *Data_out* kernel are computed (Lines 6 and 7), and then, the tuple is issued to the corresponding *Converge* kernel by using the

Algorithm 3 Converge Kernel

```
   Input  : N_x (number of tuples sent from the Converge kernel x),
            BUFF_DEPTH (Number of available on-chip tuples),
1  TUPLE buffer[BUFF_DEPTH];
2  int tuple_num = 0;
3  int wr_id = 0, rd_id = 0, i = 0;
4  int tuples_sent = 0;
   /* Loop continues when the number of tuples sent to
      consumer kernel is less than size.                    */
5  while (i < N_x) do
       /* Continue only if on-chip buffers have enough
          space for all IR channels.                        */
6      if (tuple_num <= (BUFF_DEPTH - IR)) then
7          #pragma unroll
8          for (j = 0; j < IR; j++) do
9              bool valid = false;
10             TUPLE tuple_in = read_channel_nb(C_IN_PAR[x][j], &valid);
11             if (valid) then
12                 buffer[wr_id] = tuple_in;
13                 wr_id = (wr_id+1)&(BUFF_DEPTH-1);
14                 tuple_num = tuple_num + 1;
15             end
16         end
17     end
       /* Write one tuple to consumer kernel when active
          tuples are available on this kernel.              */
18     if (tuple_num > 0) then
19         tuple_num = tuple_num - 1;
20         TUPLE tuple_out = buffer[rd_id];
21         rd_id = (rd_id+1)&(BUFF_DEPTH-1);
22         write_channel(C_OUT_PAR[x](or C_OUT_SKEW), tuple_out);
23         tuples_sent++;
24     end
25 end
```

dedicated OpenCL channel (API: *write_channel*) and only one of $DO+1$ consumer kernels at the consumer stage can receive the ith tuple. In particular, when $partition\_index$ is equal to the index of the input skewed partition ($skewed\_index$), the tuple is written to the ($i\%IR$)th channel *C_IN_SKEW[i%IR]* dedicated for the skewed partition (Line 10). Otherwise, the tuple is issued to the keyth *Converge* kernel via the ($i\%IR$)th channel *C_IN_PAR[key][i%IR]* dedicated for the keyth *Data_out* kernel (Line 15).

C. Design Details of Converge Kernel (Producer)

Without *Converge* kernel, the previous work [15] can only issue one tuple to the consumer stage per cycle. Hence, it can be the performance bottleneck for many optimization combinations according to our cost model in Section VI.

Accordingly, in order to issue IR tuples per cycle to the consumer stage, the Converge kernel is used to read the tuples from its IR input channels (connected to *Data_in* kernel) in a nonblocking fashion, based on the *simplifying loop-carried dependence* from Altera [1]. It then writes one tuple per cycle to the corresponding consumer kernel via the output channel. Therefore, the consumer kernel only requires to read data from one input channel, not directly from IR channels connected to the *Data_in* kernel.

The working process of the *Converge* kernel, as shown in Algorithm 3, can be summarized as follows. This kernel uses a variable *buffer* to store *BUFF_DEPTH* tuples in private memory (Line 1) and uses the variable $tuple\_num$ to keep track of the number of active tuples stored in *buffer* (Line 2). wr_id and rd_id denote the write and read indices of *buffer* (Line 3). The kernel keeps working when the number of tuples (*tuples_sent*) sent to corresponding consumer kernel is less than N_x (Line 5). The *Converge* kernel contains two parts: reading part and writing part.

Algorithm 4 Data_out Kernel

```
   Input  : N_x (number of tuples handled by the x-th Data_out kernel),
            B (number of partitions),
            buckets (on-chip buckets for B partitions),
            counters (counter for each partition in local memory).
   Output : data_out (output address of tuples in global memory)
1  for (i ← 0 to N_x ) do
       /* read one tuple from Data_in kernel.              */
2      tuple = read_channel(C_OUT_PAR[x]);
3      index = dest_hash(tuple.key)%B;
4      counter_index = counters[index]++;
5      index_slot = counter_index & (S-1);
6      buckets[index * S + index_slot] = tuple;
7      if (index_slot == (S-1)) then
           /* Store the whole bucket to data_out.          */
8          data_out[counter_index − index_slot] =
           buckets[index * S];
9      end
10 end
```

The reading part is active only when the on-chip *buffer* has enough space for receiving IR tuples from all input channels at the current cycle (Line 6). Otherwise, it will stop receiving the tuples from input channels. We specify the "#pragma unroll" directive (Line 7) to fully unroll the loop (Line 8), and then each iteration generates the custom hardware for one input channel. In particular, when one tuple arrives at one input channel with *valid == true* (Lines 10 and 11), the received tuple, *tuple_in*, is stored into the on-chip *buffer* (Line 12). During the same cycle, wr_id and $tuple\_num$ are updated (Lines 13 and 14).

The writing part is active when the number of active tuples is larger than 0 (Line 15). In particular, one tuple, *tuple_out*, is written to the corresponding consumer kernel via the output channel *C_OUT_PAR* (Line 19). During the same cycle, $tuple\_num$ and *rd_id* are updated (Lines 16 and 18).

D. Design Details of Data_out Kernel (Consumer)

The *Data_out* kernel reads from its input channel connected to the corresponding *Converge* kernel of the producer stage, and then combines multiple one-tuple memory transactions into a single many-tuples transaction to the global memory.

In general, we employ the OpenCL task kernel to implement *Data_out* kernel, where the degree of parallelism is determined at the compile time and only one work item is active. In particular, its lock processing ability is L cycles per tuple (e.g., $L = 7$), since the current tuple and the next tuple might belong to the same partition and then the critical path exists on the read/write updating of $counters$ in local memory (Line 4). Therefore, the *Data_out* kernel would read one tuple from its blocking channel every L cycles for the data partitioning. The lock processing ability can be significantly improved by running multiple *Data_out* kernels concurrently.

In this kernel, we resolve the overhead due to random memory accesses by combining multiple original one-tuple memory transactions into a single many-tuples transaction to the global memory. In particular, on-chip buckets ($buckets$), allocated in local memory, can accommodate $B * S$ tuples, where B is the number of on-chip buckets and the S is the slot number of tuples for each bucket. For each bucket, S tuples can be temporarily buffered on FPGA before they are stored back to the global memory in one memory transaction. That is, the number of random transactions is reduced by S times, thus significantly reducing the memory bandwidth overhead.

Algorithm 5 Skewed_Handling Kernel

```
   Input  : N_skew (number of skewed tuples),
            bucket_skew (on-chip bucket for the skewed partition),
            counter_skew (counter (private memory) for skewed partition).
   Output : data_out (tuple output address in global memory)
1  for (i ← 0 to N_skew ) do
       /* read one tuple from Data_in kernel.                    */
2      tuple = read_channel(C_OUT_SKEW);
3      counter_index = counter_skew++;
4      index_slot = counter_index & (S-1);
5      bucket_skew[index_slot] = tuple;
6      if (index_slot == (S-1)) then
           /* Store the whole bucket to data_out.                */
7          data_out[counter_index − index_slot] = bucket_skew[0];
8      end
9  end
```

In particular, the working process of *Data_out* kernel can be summarized, as shown in Algorithm 4.

First, the kernel reads one tuple ($tuple$) from the blocking channel (*C_OUT_PAR*) connected to the *Converge* kernel (Line 2) using the API (*read_channel*). Second, the index ($index$) of the partition is calculated (Line 3) and the corresponding counter ($counters[index]$) is updated (Line 4). According to the above-mentioned analysis, the critical path lies on this local memory based updating. Third, $tuple$ is stored to the corresponding on-chip bucket (Lines 5 and 6). Fourth, if the on-chip bucket for the corresponding partition (with $index\_slot$) is full (equal to $S-1$), then the bucket is entirely stored in the global memory via a single memory transaction (Line 8).

E. Design Details of Skewed_Handling Kernel (Consumer)

The design goal of the Skewed_handling Kernel is to handle the tuples of the skewed partition, as shown in Algorithm 5.

The main difference between *Skewed_handling* kernel and *Data_out* kernel lies on the lock processing ability. The lock processing requires S_H (e.g., 1) cycles for each tuple in the *Skewed_handling* kernel, since the read/write updating of $counter\_index$ stored in the private memory can be finished in one cycle. However, it requires L cycles (e.g., $L = 7$) per tuple in the *Data_out* kernel. The working process of the *Data_handling* kernel can be summarized as follows.

First, the kernel reads one skewed tuple ($tuple$) from its dedicated channel (*C_OUT_SKEW*) connected to the *Converge* kernel (Line 2). Second, the dedicated counter ($counter\_skew$) in private memory is updated within one cycle (Line 3). Third, $tuple$ is stored to the dedicated on-chip bucket ($bucket\_skew$) (Lines 4 and 5). Fourth, if the on-chip bucket for the corresponding skewed partition is full (equal to $S-1$), then the whole on-chip bucket is entirely written to the global memory (Line 7) in one global memory transaction.

VI. FPGA-Specific Cost Model

It is an important and challenging task to choose the optimal configuration for various tuning parameters, such as IR, DO, and S. In this section, we develop an FPGA-specific cost model to estimate the execution time of multikernel data partitioning with different parameter combinations (IR, DO, and S), as shown in Table I.

We observe that the number of clock cycles is independent of the frequency. In particular, one OpenCL kernel can exist at different FPGA images, and each FPGA image may have several other OpenCL kernels and then has different frequency. However, the number of clock cycles required by the kernel is fixed in different FPGA images. For example, if the OpenCL kernel takes 1 s in one FPGA image with a frequency of 200 MHz, the kernel needs 2 s in the other FPGA image with a frequency of 100 MHz. Therefore, it evaluates the execution time T_e of the OpenCL kernel to be the number of clock cycles divided by the frequency, as shown in (1), where C_e is the estimated number of cycles required by the multikernel partitioning, and $\#Freq$ is the frequency of the FPGA image containing the multikernel data partitioning. It is extremely different to develop an analytical model to accurately predict the frequency of FPGA image, which may contain several OpenCL kernels. The impacting factors include the FPGA resource consumption, size of local memory, and so on. Therefore, we use the frequency from the Altera compilation report [1]

$$T_e = \frac{C_e}{\#\text{Freq}}. \tag{1}$$

Since the computation and memory cycles can be overlapped, C_e is calculated to be the larger value between computation cycles (C_{comp}) and global memory access cycles (C_{mem}), as shown in

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

A. Cost Model for Estimating C_{comp}

Since the kernels of the producer and consumer stages have their own custom FPGA resources to implement, they execute completely in parallel. Therefore, C_{comp} is estimated to be the larger value between C_{in} and C_{out} in (3), where C_{in} is the estimated number of cycles required by the producer kernel in the producer stage, C_{out} is the estimated number of cycles required by all the kernels of the consumer stage and N is the number of input tuples

$$C_{\text{comp}} = \text{Max}(C_{\text{in}}, C_{\text{out}}). \tag{3}$$

B. Evaluating C_{in}

It is estimated to be the larger value between the input cycles (N/W) required to load the tuples and the output cycles (N/IR) required to issue tuples to the consumer stage, as shown in (4). Regarding the estimation of input cycles, the assumption is that the global memory has the sufficient memory bandwidth and can provide W tuples per cycle for the *Data_in* kernel. Since we only consider the estimation of computation cycles, the assumption is always satisfied

$$C_{\text{in}} = \text{Max}\left(\frac{N}{W}, \frac{N}{IR}\right) \tag{4}$$

IR means the issue rate (e.g., two tuples per cycle) from the producer stage to the consumer stage. In order to issue IR tuples per cycle, the producer stage uses one *Converge* kernel to combine IR input channels (from *Data_in* kernel)

into one output channel, so that the corresponding consumer kernel can read the tuples from only one channel. With this *Converge* kernel, IR can be greater than 1 and the *Data_out* kernel at the consumer stage can remain the same as that of our previous work [15]. However, our previous work [15] can only support the case IR=1, which can be the performance bottleneck when there are sufficient consumer kernels.

C. Evaluating C_{out}

Since all the kernels of the consumer stage work concurrently, C_{out} is evaluated to be the number of cycles required by the slowest consumer kernel, as shown in (5). N_i (or N_{skew}) is the number of tuples processed by the ith *Data_out* (or *Skewed_handling*) kernel, and L (or S_H) is the number of cycles consumed by one tuple in one *Data_out* (or *Skewed_handling*) kernel. The assumption here is that all the global memory transactions generated from all the kernels of the consumer stage can be immediately written to the global memory. It is always satisfied, since we only consider the estimation of computation cycles

$$C_{\text{out}} = \text{Max}\Big(\max_{1\le i\le DO}(N_i \times L), N_{\text{skew}} \times S_H\Big). \tag{5}$$

As a constraint, the total tuples processed by the consumer stage are N, as shown in (6)

$$N = \sum_{1\le i\le DO} N_i + N_{\text{skew}}. \tag{6}$$

D. Cost Model for Estimating C_{mem}

Based on *observation 4* about the memory subsystem of OpenCL-based FPGAs, there is a significant performance difference between sequential and random memory accesses. However, it is very difficult to develop an analytical model to accurately predict the performance of the memory subsystem. We evaluate C_{mem} to be the sum of the clock cycles required by sequential and random memory accesses, as shown in (7), where $C_{\text{mem}}^{\text{seq}}$ stands for the number of clock cycles for sequential memory references, and $C_{\text{mem}}^{\text{rand}}$ stands for the number of clock cycles for random memory references. Our experimental result shows that our estimation can capture the performance trend with different optimization combinations and can thus guide the user to choose the optimal parameter configuration

$$C_{\text{mem}} = C_{\text{mem}}^{\text{seq}} + C_{\text{mem}}^{\text{rand}}. \tag{7}$$

E. Evaluating C_{mem}^{seq}

It is estimated to be N/W (number of sequential memory transactions) divided by TPC_{seq} (number of sequential global memory transactions handled by the memory subsystem within one cycle), as shown in

$$C_{\text{mem}}^{\text{seq}} = \frac{N/W}{TPC_{\text{seq}}}. \tag{8}$$

The sequential memory transactions come from the *Data_in* kernel, where each memory transaction can read W tuples from global memory, thus reducing the number of input global memory transactions by W times. TPC_{seq} is determined by the calibrations, as shown in Section III-B. In our experiments, in order to calibrate TPC_{seq}, we measure the number of clock cycles of the sequential scan, using four load operations with *long8*, and calculate the result accordingly.

F. Evaluating C_{mem}^{rand}

It is estimated to be N/S (number of random memory transactions) divided by TPC_{rand} (the number of random global memory transactions handled by the memory subsystem within one cycle), as shown in

$$C_{\text{mem}}^{\text{rand}} = \frac{N/S}{TPC_{\text{rand}}}. \tag{9}$$

The random memory transactions come from the consumer kernels, where each memory transaction can write S tuples back to the global memory in one memory transaction, and then the number of generated global memory output transactions is reduced by S times. Like TPC_{seq}, TPC_{rand} is also determined by the calibrations. In particular, we measure the number of clock cycles of the random scan, using four load operations with *long8*, and calculate the result accordingly.

G. Parameter Setting

In order to achieve good performance of data partitioning on OpenCL-based FPGAs, we can leverage the cost model to determine the suitable setting for a series of parameters, including IR, S, and DO. Since their ranges are reasonably small due to the limitation of FPGA resources, we can evaluate all the possible combinations, and then choose the parameter combination with the minimal estimation time.

Table I also shows the possible range of each parameter used in our model. The larger IR can have wider issue rate of the producer stage, and it does not require a lot of FPGA resources. However, the FPGA frequency is decreased significantly when IR increases, since the *Converge* kernel requires to combine the potential tuples from IR input channels in one cycle, and then, the critical path becomes longer when IR increases. So IR belongs to the set $\{1, 2, 4\}$. S (or DO) is set to be less than 32 (or 16) due to the limited FPGA resources. In particular, either more consumer kernels or larger on-chip buffers require FPGA resources to implement. Both conditions are satisfied to achieve good performance on FPGAs. Therefore, it requires one cost model to determine the optimal parameter combination, rather than evaluating all the possible combinations on real FPGAs.

VII. Experimental Evaluation

A. Experimental Setup

We conduct our experiments on one Terasic's DE5-Net board and one full-fledged CPU. The configurations of FPGA and CPU are summarized in Table II.

B. FPGA Board

It contains 4-GB two-bank DDR3 device memory (with theoretical bandwidth 25.6 GB/s) and an Altera Stratix V GX FPGA, with the Altera OpenCL SDK version 14.0.

TABLE II
COMPARISON OF EXPERIMENT PLATFORMS

	Intel Xeon E5-2620	Altera Stratix V GX A7
Frequency	2G	200-300M
Memory	4-bank 64-bit DDR3	2-bank 64-bit DDR3
Memory Bandwidth	42.6 GB/s	25.6GB/s
TDP	95W	25W
On-chip memory	15MB L3 cache	6.25MB Memory blocks
Computing ability	6 cores, 12 threads	622K LEs, 256 DSPs

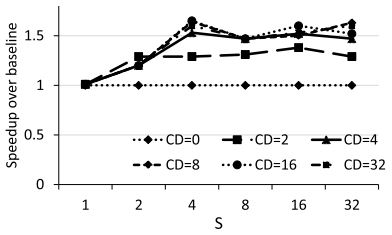


Fig. 5. Speedup of different CDs over zero CD, with varying S.

The FPGA has 622-K logic elements, 256 DSP blocks, and 2560 M20-K memory blocks (50 Mb). The host employs one X8 PCI-e 2.0 to communicate with the FPGA board.

C. CPU

We use the Intel Xeon E5-2620, which contains six cores and 15-MB L3 cache. Its frequency is 2 GHz.

The input data are a relation (i.e., table) with the tuple format of <key, payload>. Both keys and payloads are 4-B integers. The probability of referencing individual keys follows a Zipf distribution and the Zipf factor varies between 0 and 1.75 [2]. The default Zipf factor is 0. We vary the relation size and the default size is 128 MB (i.e., 16 million tuples). The partitioning function is radix function (least-significant bits). This paper mainly focuses on the performance on the FPGA itself, so the input data sets are initially loaded into the device memory to exclude the time of data transfer via PCI-e.

D. Performance Impact of Each Parameter

1) Impact of CD: We now study the performance impact of the CD. Fig. 5 shows the speedup of data partitioning with varying CD over the case (CD=0). Since the partitioning with different IR and DO values have roughly the same trend as that of ($IR = 1$ and $DO = 8$), IR is set to be 1 and DO is set to be 8. The experimental result shows that the implementation with different CD reaches its best performance when CD is greater than 4. Therefore, in the following experiments, we set CD to be 8.

2) Impact of S and DO: Since IR is not the bottleneck in the majority of parameter combinations of IR, S, and DO, IR is set to be 1 to analyze the impact of S and DO. Besides, we will present the impact of IR in Section VII-D3. Since the input relation in the experiment has the Zipf factor (0), the impact of *Skewed_handling* kernel is insignificant. Therefore, we focus on the *Data_out* kernels.

We study the measured and estimated execution times of data partitioning with different combinations of DO and S values, as shown in Fig. 6. Our cost model is able to roughly capture the performance trend of different parameter combinations. Hence, we are able to find the suitable parameter settings to achieve the best performance of data partitioning. We give more details about the performance trend of different parameter combinations.

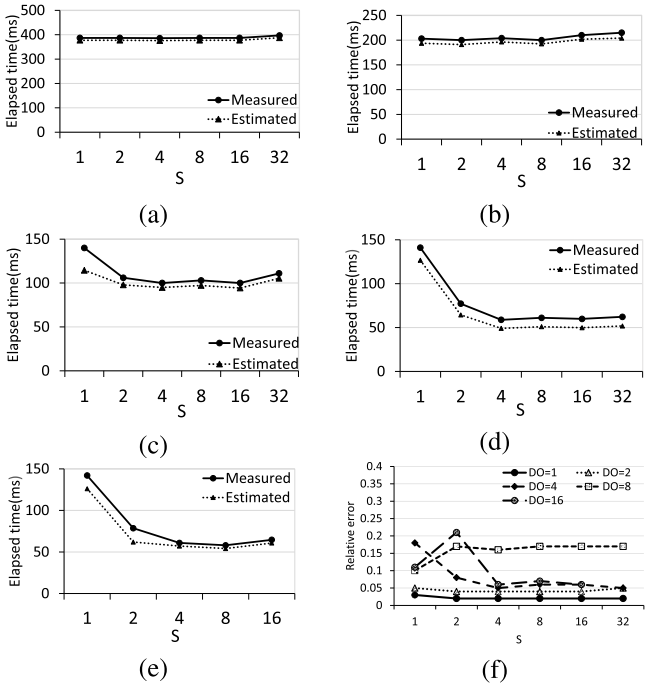


Fig. 6. Cost model evaluations with different settings for DO and S. (a) $DO = 1$. (b) $DO = 2$. (c) $DO = 4$. (d) $DO = 8$. (e) $DO = 16$. (f) Relative error.

For the cases ($DO = 1$) and ($DO = 2$), the main bottleneck is the lock overhead of the consumer stage, due to the lack of $Data\_out$ kernels.

For the cases ($DO = 4$), when S is equal to 1, the global memory performance (C_{mem}) dominates the overall elapsed time, since there are too many single-tuple random memory write operations. When S is greater than 1, the number of memory write operations is reduced by S times, and then, the lock overhead dominates due to the lack of $Data\_out$ kernels.

For the cases ($DO = 8$) and ($DO = 16$), C_{mem} dominates the total execution time when S is less than 8. When S is larger than or equal to 8, the *Data_in* kernel in the producer stage dominates the execution time. It means that IR is the bottleneck and we will talk about these cases in Section VII-D3. One interesting finding is that the case ($S = 16$ and $DO = 16$) is slower than the case ($S = 8$ and $DO = 16$), since they roughly require the same number of cycles and the frequency (267 MHz) of the case ($S = 16$ and $DO = 16$) is less than that (296 MHz) of the case ($S = 8$ and $DO = 16$).

In order to validate the effectiveness of our cost model, the relative error is defined in (10), where T_m is the measured execution time for the data partitioning and T_e is the estimated execution time. The experimental result in Fig. 6(f) shows that our cost model can roughly predict the performance (in terms of relative error) of each parameter combination, and the cost

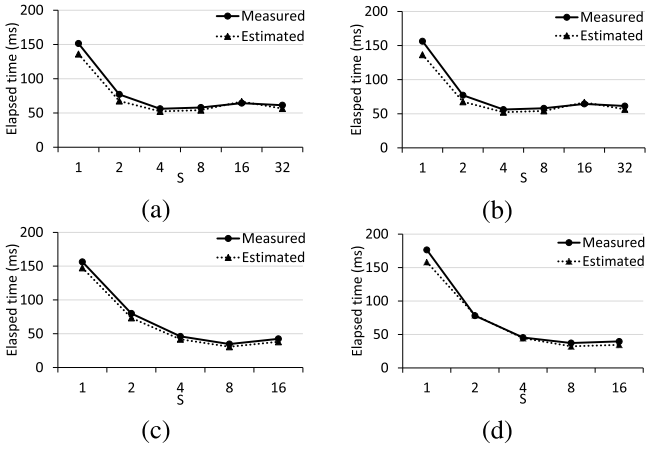


Fig. 7. Cost model evaluations with different IR. (a) $IR = 2$ and $DO = 8$. (b) $IR = 4$ and $DO = 8$. (c) $IR = 2$ and $DO = 16$. (d) $IR = 4$ and $DO = 16$.

model has the good enough accurarcy to determine the optimal parameter combination of data partitioning

$$relative\_error = \frac{|T_m - T_e|}{T_m}. \tag{10}$$

In summary, the performance bottleneck shifts for different settings of S and DO, with IR equal to 1. Our cost model can capture the performance trend when the parameter combination changes.

3) Impact of IR: Based on the discussion on S and DO (with $IR = 1$) in Section VII-D2, IR can be the performance bottleneck of the data partitioning when DO is equal to 8 or 16. Therefore, we study the measured and the estimated execution time of data partitioning with different combinations of IR, S, and DO values, where DO is 8 (or 16) and IR is 2 (or 4), as shown in Fig. 7.

For the cases ($IR = 2$ and $DO = 8$) and ($IR = 4$ and $DO = 8$), the global memory performance (C_{mem}) dominates the overall elapsed time when S is less than 8, since there are too many small-granularity random memory write operations. When S is greater than or equal to 8, the number of memory write operations is reduced by S times, and then, the lock overhead dominates due to the lack of $Data\_out$ kernels.

For the cases ($IR = 2$ and $DO = 16$) and ($IR = 4$ and $DO = 16$), the global memory performance (C_{mem}) dominates the overall elapsed time of partitioning with different S values. However, the elapsed time (34.8 ms) of the case ($IR = 2$, $S = 16$, and $DO = 16$) is larger than that (42.5 ms) of the case ($IR = 2$, $S = 8$, and $DO = 16$), since the frequency (215 MHz) of the case ($IR = 2$, $S = 16$, and $DO = 16$) is significantly less than that (267 MHz) of the case ($IR = 2$, $S = 8$, and $DO = 16$).

In summary, we compare the performance of three cases ($IR = 1$ and $DO = 16$), ($IR = 2$ and $DO = 16$), and ($IR = 4$ and $DO = 16$) with varying S to demonstrate the impact of *IR*, as shown in Fig. 8(a). From the comparison, we can see that ($IR = 1$ and $DO = 16$) has the best performance when S is less than or equal to 2, since the performance bottleneck lies on the memory bandwidth overhead and its frequency is higher than that of ($IR = 2$ and $DO = 16$) and ($IR = 4$ and $DO = 16$). When S is larger than 2, their bottleneck shifts from the memory bandwidth overhead to the issue rate of producer stage. Therefore, ($IR = 2$ and $DO = 16$) and ($IR = 4$ and $DO = 16$) can achieve better performance than that of ($IR = 1$ and $DO = 16$).

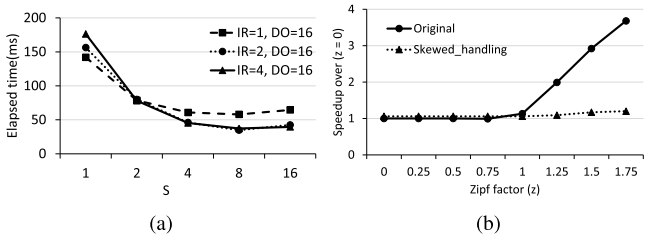


Fig. 8. Performance comparison. (a) Varying IR. (b) Varying Zipf factors.

TABLE III
FPGA RESOURCE UTILIZATIONS

	LUTs	REGs	MEMs	DSPs	Frequency
Original	7.4%	8.3%	20.1%	1%	208 MHz
IR=1	16%	25.4%	46.9%	12.5%	296 MHz
IR=2	22%	34.7%	68.5%	12.5%	267 MHz

4) Impact of Skewed_Handling *Kernel:* We study the impact of *Skewed_handling* kernel with the varying Zipf factor. The data partitioning without the *Skewed_handling* kernel is denoted by "*original*," and the data partitioning with the *Skewed_handling* kernel is denoted by "*skewed_handling*." Fig. 8(b) shows the performance speedups of "*original*" and "*skewed_handling*" over the baseline, where the baseline is "*original*" with z equal to 0. The experimental result shows the effectiveness of the *Skewed_handling* kernel when z is larger than 1, since this kernel has higher throughput.

E. Performance Comparison

In this section, we study the performance of our multikernel partitioning, in comparison with the original algorithm (denoted by *Local_1CU*) in Section III, and in comparison with the state-of-the-art implementation on CPU.

1) Comparison With Local_1CU: We compare the performance of our multikernel design and the original lock-based design. In order to demonstrate the impact of IR, we choose the optimal case for different IR values according to our cost model: ($IR = 1$, $S = 8$, $DO = 16$, $CD = 8$, and $B = 1024$) and ($IR = 2$, $S = 8$, $DO = 16$, $CD = 8$, and $B = 1024$). They are denoted by $IR = 1$ and $IR = 2$, respectively. The resource consumption and achieved frequency for each implementation are shown in Table III. From Table III, we can see that *Local_1CU* requires the minimum FPGA resources and lowest frequency, since it contains the atomic primitives, which impede the achieved frequency of the entire FPGA image [1]. Another thing to be mentioned is that we cannot improve the performance of *Local_1CU* via reducing the number of memory transactions (which requires more FPGA resource), since its bottleneck lies on the lock overhead.

F. Impact of Data Size

We compare the three data partitioning implementations with the varying input sizes (16, 32, 64, 128, and 192 MB), as

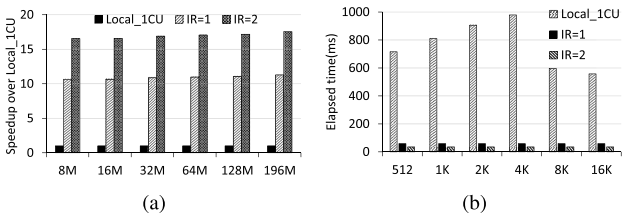


Fig. 9. Performance comparisons between the original and proposed implementations. (a) Varying tuples. (b) Varying partitions.

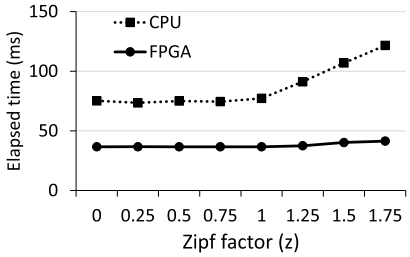


Fig. 10. Comparison with CPU.

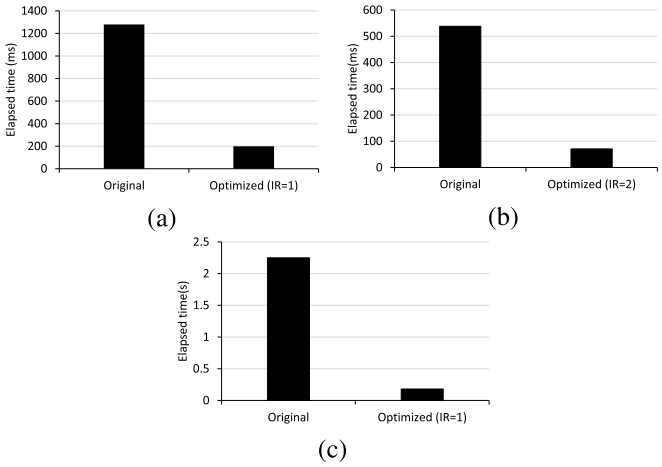


Fig. 11. Performance comparisons for three applications. (a) Hash join. (b) Histogram. (c) Hash search.

shown in Fig. 9(a), where the number of partitions is 8 K. We can see that all the three cases scale well when increasing the data size. And our proposed multikernel approach ($IR = 2$) is 16.6× faster than the original *Local_1CU* implementation.

G. Impact of the Number of Partitions

Fig. 9(b) shows the elapsed time of data partitioning with different numbers of partitions (from 512 to 16384). With varying number of partitions, the performance of multikernel approach ($IR = 1$ and $IR = 2$) is much higher and more stable than the *Local_1CU* implementation.

1) Comparison With CPU: Fig. 10 shows the performance comparison between the state-of-the-art partitioning [2] on CPU and our multikernel implementation on FPGA with different Zipf factors. The CPU implementation, written in C++, has already explored the optimization methods, such as 128-b single-instruction-multiple-data (SIMD), software managed buffer, and multipass partitioning. Besides, all the hardware threads (12) are leveraged to accelerate the data partitioning. The implementation on CPU is denoted by "CPU" and the implementation on FPGA, denoted by "FPGA," uses the case ($IR = 2$, $S = 8$, $DO = 16$, $CD = 8$, and $B = 1024$). The experimental result shows that our multikernel partitioning has better performance than the CPU implementation, since the CPU version suffers from cache misses and TLB misses due to random memory accesses. Another interesting observation is that the performance speedup of our implementation over the CPU implementation becomes larger when Zipf factor is larger than 1, since skewed data cause the load unbalances among CPU cores in the CPU implementation.

H. Case Studies for Data Partitioning

We perform three case studies in relational databases to show the importance of data partitioning, including hash join, histogram, and hash search. All of them employ the data partitioning as a building block. In most cases, data partitioning is one of the major performance factors. We compare the performance of the proposed data partitioning method (denoted as *optimized*) against the one without the multikernel data partitioning (denoted as *original*). Overall, for all the three case studies, the optimized version is 6×–12× faster than the original version, thanks to the proposed multikernel data partitioning.

1) Hash Join: The partitioned hash join [2] is an efficient implementation for hash joins, and is widely used in relational databases. The hash join takes two relations as input, and finds the matching tuple pairs from the two relations according to the join predicates. The algorithm has two major phases: partitioning and build_probe. In the partitioning phase, each relation is divided into a predefined number of partitions. The number of partitions is determined, so that each partition can fit into the on-chip memory. Then, the build_probe phase is to perform a simple hash join algorithm on each partition pair with the same partition ID. For the simple hash join algorithm, it first scans one partition to build one hash table ("build"). Next, for each tuple of the partition, a search is performed ("probe") on the hash table to find the matching tuples. The *Local_1CU* implementation (*original*) is used as the baseline. Fig. 11(a) shows the elapsed time of partitioned hash joins. Each of the input relations has 16 million tuples (with a 4-B key and a 4-B value). The *optimized* implementation ($IR = 1$) is roughly 6.4 times faster than the *original* one. The reason why ($IR = 1$) is chosen for the *optimized* implementation is that the build_probe phase dominates the total execution time and ($IR = 2$) with the same optimization level for build_probe cannot fit into one FPGA image.

2) Histogram: The histogram is a crucial part of query planning and it is memory-intensive operation. The histogram divides the input tuples into a number of bins. Then, each tuple of the input relation will be collected into the bin i, where i ranges from 1 to B_N and B_N is the total number of bins. One important issue lies on the write conflict to the bins, where more than one work items try to add their values into the same location of the same bin concurrently.

Then, the basic implementation with NDRange model will employ the atomic operation to protect each bin. If any work item wants to update any bin, the corresponding bin should be updated atomically. The bins are implemented using either global memory or local memory. The bin using global memory is slow but visible to all work groups, while the bin using local memory (*original*) is fast but only visible to one work group.

With the multikernel data partitioning scheme, several task kernels (*optimized*) work concurrently to compute the histogram, and then, each kernel computes on a fraction of bins. The input relation has 32 million tuples and B_N is 4 K. The *optimized* implementation ($IR = 2$) is roughly 7.6 times faster than the *original* implementation, as shown in Fig. 11(b). Since there are still sufficient FPGA resources available, ($IR = 2$) can be chosen to further improve the performance of histogram, compared with ($IR = 1$).

3) Hash Search: The hash table consists of multiple hash headers, each of which contains a pointer to the corresponding bucket. Each bucket stores the records that have the same hash value. Since the hash table is usually so large that the entire hash table and buckets cannot reside on the local memory of FPGAs, the hash table and buckets reside on the global memory. The hash search takes as input a number of search keys, performs probes on the hash table with these keys, and outputs the matching records. In the *original* implementation, each search key directly searches the global memory for matching, resulting in bad global memory performance. The conventional optimization method is to use multiple CUs. However, the performance improvement is limited, since the global memory bandwidth is the main bottleneck. In our *optimized* implementation, the search keys are divided into a predefined number of partitions, so that the corresponding hash headers and buckets can fit into the local memory of FPGAs for each partition. Then, for each partition of search keys, the corresponding hash headers and buckets are first loaded into local memory, and the corresponding hash search is second executed. Fig. 11(c) shows the elapsed time of hash search. The number of search keys is 4 million and the number of keys in the hash table is 16 M and the average number of keys for each bucket is 32. The *optimized* implementation ($IR = 1$) is roughly 12.5 times faster than the *original* implementation. The reason why ($IR = 1$) is chosen for the *optimized* implementation is that the data partitioning with ($IR = 2$) cannot fit into FPGA when the same optimization effort as the local memory based hash search is applied.

VIII. Conclusion

Motivated by recent popularity of OpenCL-based FPGAs on data-intensive applications, we develop a new multikernel partitioning approach for databases. Moreover, we develop an FPGA-specific cost model to guide the parameter settings. Our experimental results demonstrate that: 1) our cost model can accurately predict the performance of data partitioning with different parameter combinations and 2) the proposed multikernel approach can achieve 16.6× speedup over the original implementation.

References

[1] *Altera SDK for OpenCL Optimization Guide*, Altera, San Jose, CA, USA, 2014.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 362–373.

[3] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2014, pp. 151–160.

[4] D. Chen and D. Singh, "Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2012, pp. 5–12.

[5] Y. Chen *et al.*, "FCUDA-NoC: A scalable and efficient network-on-chip implementation for the CUDA-to-FPGA flow," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 6, pp. 2220–2233, Jun. 2016.

[6] Y. Chen, B. Schmidt, and D. L. Maskell, "Reconfigurable accelerator for the word-matching stage of BLASTN," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 4, pp. 659–669, Apr. 2013.

[7] T. S. Czajkowski *et al.*, "From OpenCL to high-performance hardware on FPGAS," in *Proc. 22nd Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2012, pp. 531–534.

[8] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proc. ACM/IEEE Conf. Supercomput.*, New York, NY, USA, Nov. 2007, pp. 46:1–46:12.

[9] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled CPU-GPU architecture," *Proc. VLDB Endowment*, vol. 6, no. 10, pp. 889–900, Aug. 2013.

[10] J. He, S. Zhang, and B. He, "In-cache query co-processing on coupled CPU-GPU architectures," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 329–340, Dec. 2014.

[11] 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 Program. Logic Appl. (FPL)*, Sep. 2013, pp. 1–8.

[12] S. Paul, A. Krishna, W. Qian, R. Karam, and S. Bhunia, "MAHA: An energy-efficient malleable hardware accelerator for data-intensive applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 6, pp. 1005–1016, Jun. 2015.

[13] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, 2000, pp. 128–138.

[14] P. Wang and J. McAllister, "Streaming elements for FPGA signal and image processing accelerators," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 6, pp. 2262–2274, Jun. 2016.

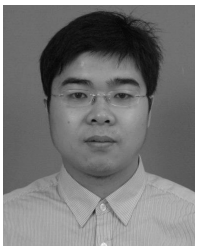
[15] Z. Wang, B. He, and W. Zhang, "A study of data partitioning on OpenCL-based FPGAs," in *Proc. 25th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2015, pp. 1–8.

[16] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 114–125.

[17] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang, "Relational query processing on OpenCL-based FPGAs," in *Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2016, pp. 1–10.

[18] Z. Wang, S. Zhang, B. He, and W. Zhang, "Melia: A MapReduce framework on OpenCL-based FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 12, pp. 3547–3560, Dec. 2016.

[19] S. Zhang, J. He, B. He, and M. Lu, "OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures," *Proc. VLDB Endowment*, vol. 6, no. 12, pp. 1374–1377, Aug. 2013.



Zeke Wang received the B.Sc. degree from the Harbin University of Science and Technology, Harbin, China, in 2006, and the Ph.D. degree from Zhejiang University, Hangzhou, China, in 2011.

He is currently a Research Fellow with the School of Computing, National University of Singapore, Singapore. His current research interests include heterogeneous computing with a focus on field-programmable gate array and database systems.

Johns Paul received the bachelor's degree from the National Institute of Technology, Calicut, India. He is currently pursuing the Ph.D. degree with the School of Computer Science and Engineering, Nanyang Technological University, Singapore.

His current research interests include heterogeneous computing (field-programmable gate array & GPU) and database systems.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 2003, and the Ph.D. degree in computer science from The Hong Kong University of Science and Technology, Hong Kong, in 2008.

He is currently an Associate Professor with the School of Computing, National University of Singapore, Singapore. His current research interests include high performance computing, distributed and parallel systems, and database systems.

Wei Zhang received the Ph.D. degree from Princeton University, Princeton, NJ, USA, in 2009.

She was an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore, from 2010 to 2013. In 2013, she joined The Hong Kong University of Science and Technology, Hong Kong, as an Assistant Professor and established the Reconfigurable System Laboratory Department of Electronic & Computer Engineering, HKUST. Her current research interests include reconfigurable system, field-programmable gate array-based design, low-power high-performance multicore system, embedded system security, and emerging technologies.