A Study of Data Partitioning on OpenCL-based FPGAs

Zeke Wang Bingsheng He
Nanyang Technological University, Singapore

Wei Zhang
Hong Kong University of Science and Technology

***Abstract*—A lot of research efforts have been devoted to accelerating relational database applications on FPGAs, due to their high energy efficiency and high throughput. Most of the existing studies are based on hardware description languages (HDLs). Recently, FPGA vendors have started to develop OpenCL SDKs for much better programmability. In this paper, we investigate the performance of relational database applications on OpenCL-based FPGAs. As a start, we study the performance of data partitioning, a core operation widely used in relational databases. Due to random memory accesses, data partitioning is time-consuming and can become a major bottleneck for database operators such as hash join. We start with the state-of-the-art OpenCL implementation which was originally designed for CPUs/GPUs, and find that it suffers from lock overheads and memory bandwidth overheads. To reduce lock overheads, we develop a simple yet efficient multi-kernel approach to leverage two emerging features of Altera OpenCL SDK, namely *task kernel* and *channel*. Moreover, on-chip buckets are employed to reduce the number of memory transactions. We further develop a cost model to guide the parameter configuration. We evaluate the proposed design on a recent Altera Stratix V FPGA. Our results demonstrate 1) our cost model can accurately predict the performance of data partitioning under different parameter settings; 2) our proposed multi-kernel approach can achieve 10.7X speedup over the existing OpenCL implementation. Also, the experiments with three case studies show that the optimized implementations can achieve 4-12X performance improvement over the original implementations.**

I. INTRODUCTION

FPGAs have become an attractive and effective means of accelerating relational database applications, due to their high energy efficiency and high throughput. A lot of fruitful research efforts have been devoted to this direction (e.g., [2], [3], [5], [6], [7], [9], [12], [16], [21]). However, most of those previous studies are programmed with low-level hardware description languages (HDLs) like Verilog and VHDL. The programmability issues of HDLs raise serious concerns on code development and maintenance. High level synthesis (HLS) is to address the programmability issues. For example, FPGA vendors such as Altera [7], [8] and Xilinx [14] have started to develop OpenCL SDKs for much better programmability. On the other hand, OpenCL-based design and implementation for relational databases [10], [11] have been emerging on CPUs/GPUs. A natural question is how those OpenCL implementations perform on such OpenCL-based FPGAs.

The research of relational databases on OpenCL-based FPGAs is still a largely open and challenging problem. As a start, we study the performance of data partitioning with the OpenCL features supported by Altera OpenCL SDK. Data partitioning is a core operation widely used in relational databases and other data processing tasks. Given an input table, the data partitioning operation is used to divide the input into a number of partitions according to some partitioning criteria (for example, a hash function). Due to random memory accesses, data partitioning is time-consuming and can become a major bottleneck for database operators, such as hash joins [15], [18] on CPUs/GPUs. Thus, with the consideration of various design features of OpenCL SDK, we explore the design and implementation space of data partitioning and investigate whether and how we can improve the performance on FPGAs.

We start with the state-of-the-art OpenCL implementation for data partitioning [10], [11] on FPGAs. We find that, the performance is far from ideal, because of the severe lock overheads and memory bandwidth overheads. To reduce lock overheads, we develop a simple yet efficient multi-kernel approach to leverage two emerging features in Altera OpenCL SDK, namely *task kernel* and *channel* (FIFO buffer). With the channel, data partitioning is designed with a producer-consumer paradigm. Since the throughput of producer kernel is much higher than that of the consumer kernel, the scheme with one producer kernel and multiple consumer kernels is proposed, where each consumer kernel is responsible for some particular partitions. Besides, another consumer kernel is added for efficient skew handling. To reduce memory bandwidth overheads, on-chip buckets are employed to combine multiple memory transactions into a single transaction. Since each of the above optimization methods (for example, the number of consumer kernels) requires the FPGA resource to implement, a cost model is developed to guide the effective parameter configuration to maximize the data partitioning performance on FPGAs, given the FPGA resource constraint.

We evaluate the proposed design on an Altera Stratix V GX FPGA. Our results demonstrate that 1) our cost model can accurately predict the performance of data partitioning under different parameter settings; 2) our proposed approach can achieve 10.7X speedup over the existing OpenCL implementation. We study three cases for data partitioning, including hash join, histogram and hash search, and demonstrate the efficiency of our optimized data partitioning scheme.

The remainder of the paper is organized as follows. In Section II, we introduce the background of OpenCL-based

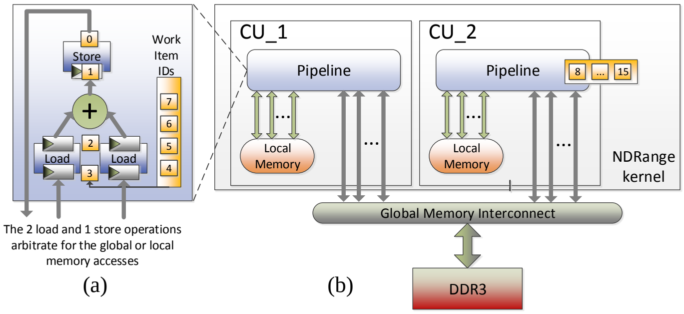


Fig. 1. Architecture of Altera NDRange kernel (adopted from [19])

FPGA and data partitioning. In Section III, we present the motivation of this study, followed by the proposed design and the cost model in Section IV. We present the experiment results in Section V and conclude in Section VI.

II. Background

A. Altera's OpenCL Architecture

OpenCL [13] has been developed for heterogeneous computing environments, e.g. CPU+GPU. It targets at a host-accelerator model of program execution, where a host processor (e.g. CPU) runs control-intensive task and offloads computation-intensive code (i.e., *kernel*) onto an external accelerator (e.g. GPU).

Recently, Altera provides the OpenCL SDK [19] to abstract the hardware complexities from the FPGA implementation. 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 achieve the whole data path.

From the perspective of OpenCL, the memory component of OpenCL-based FPGA 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 test bed, 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 sufficient number of registers, which should be employed to store intermediate results 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, and each work item executes an instance of the kernel. Figure 1a shows the pipelined parallelism with the example of a simplified vector addition example [19], where each work item executes one addition operation 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, as shown in Figure 1b. Then the CUs can execute in parallel, in terms of different work items. That is, work items are assigned to CUs for executions in parallel. Each CU has its own local memory interconnect, while all the CUs share the global memory interconnect. Compared with global memory, the on-chip local memory is low-latency and high-throughput. Thus, the local and private memory should be employed whenever possible to reduce global memory accesses. One main disadvantage is that the atomic operations are required when multiple work items attempt to update shared data structures.

2) Task kernel: The task kernel 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 parallelism at the compilation time [1] based on the dependency. The task kernel is preferred in the case that the fine-grained data are shared among many work items, since expensive atomic operations for the NDRange kernel are requried 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.

Another significant feature provided by the Altera OpenCL SDK is the channel [1], which can be used to efficiently pass data (at the *private memory* level) between two OpenCL kernels (either NDRange or task). In the conventional OpenCL implementation, the communication between two kernels are executed via the global memory. In contrast, the channel, wih a channel ID and buffer depth (e.g. *CD*), is implemented with on-chip FIFO buffer. The channel has two types: blocking channel and unblocking 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.

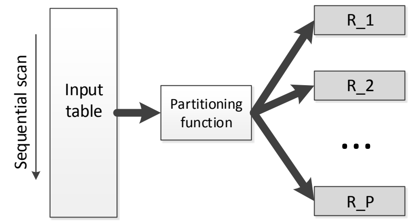


Fig. 2. Data partitioning

B. Data Partitioning

The data partitioning operation divides the input table into a number (P) of disjoint partitions according to the *partitioning function*, as shown in Figure 2. Then, each tuple, one row of input table, will be stored into the corresponding output partition i (R_i), where i ranges from 1 to P.

Data partitioning is widely used as a building block in relational database applications. For example, partitioned hash join is one of the most efficient hash join algorithms [10]. In the partitioned hash join algorithm, both tables are partitioned into the same number of partitions with the same partitioning functions. The data partitioning operation is used in this step. Next, for the corresponding partition pair, it uses simple hash join algorithm to perform the join on the partitions in that pair.

III. Motivation

In this section, we present the observations from a NDRange-kernel-based implementation of the data partitioning on OpenCL-based FPGAs to motivate our multi-kernel design. We focus on two key aspects, including lock overhead and memory performance. The detailed experimental setup can be found in Subsection V-A.

A. Lock Overhead

The conventional implementation of data partitioning using NDRange kernel achieves the pipelined parallelism in terms of multiple work items. Since work items could be in contention for the same partition, the lock mechanisms (implemented with atomic operations) are used. Therefore, the lock overhead can be an important factor for the total execution time. The corresponding implementation is shown in Algorithm 1. The lock mechanism can be implemented in global memory or in local memory with the trade-off as discussed below. The locks implemented in global memory and local memory are referred as *global locks* and *local locks*, respectively. We maintain an array of locks, and perform acquire/release operations using the index of the lock in the array.

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 firstly acquire the global lock according to the hash value (Line 6), secondly write to the corresponding partition (Lines 7-8), and thirdly release the global lock (Line 9). One corresponding optimization method is to use 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 have to be located in the global memory and a global lock shared by all the work items is required, incurring long access latency.

Algorithm 1: Lock-based single-kernel Data Partitioning

Input : $data\_in$(the input table in global memory), $counters$ (the counters in the local memory for each partition), N(number of input tuples)

Output : $data\_out$(tuple 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 having 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);
```

2) Single-kernel partitioning with local lock: Since the atomic operation on global memory may be the bottleneck for the partitioning implementation, 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, containing all the work items, can be launched to execute the partitioning algorithm, with the lock in local memory. In particular, each tuple acquires the local lock

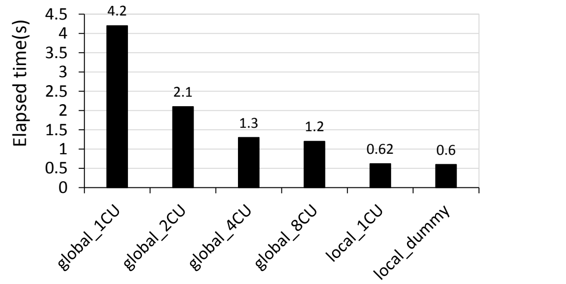


Fig. 3. Performance of single-kernel partitioning with global/local locks

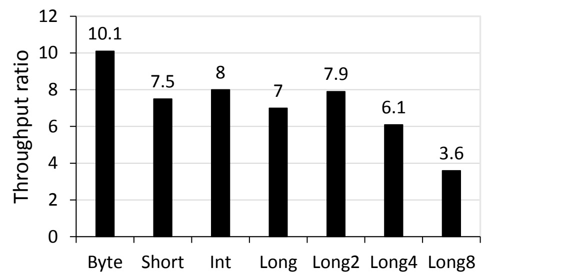


Fig. 4. Throughput ratio of sequential v.s. random memory accesses

according to the hash value, writes out to the corresponding partition, and then releases the corresponding local lock.

We compare the performance of data partitioning with global locks and local locks. Figure 3 demonstrates the performance of single-kernel partitioning implementations with global and local locks, where *global_xCU* means the partitioning implementation (with x CUs) using global lock and $x = \{1, 2, 4, 8\}$. *local_1CU* means the partitioning implementation with local lock, and *local_dummy* means the microbenmark which only acquires and releases local locks without executing the tuple-related instructions (Lines 4 and 7-8). Two observations can be obtained from Figure 3.

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

Observation 2: performance of local_dummy *is roughly the same as that of* local_1CU. Lock overhead can be the performance bottleneck for data partitioning. Therefore, lock overhead should be significantly improved to accelerate the performance of partitioning.

B. Data Access Unit Size

Another factor impacting the data partitioning performance is the global memory bandwidth utilization, since the partitioning is a memory-intensive operation. Therefore, we need to qualitatively analyze the throughput characteristics of global memory accesses on FPGAs. On the other hand, we need to accurately develop the cost model which is used to guide our design.

Observation 3: The throughput of sequential memory access is much higher than that of random memory access. Figure 4 shows the throughput ratio of sequential memory access to

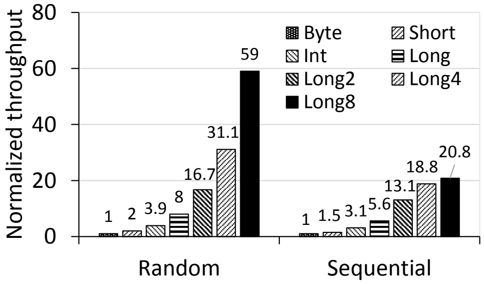


Fig. 5. Normalized throughput trend of different data access unit sizes

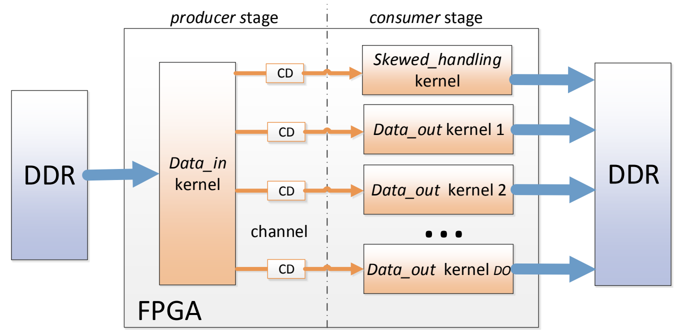


Fig. 6. The architecture of multi-kernel partitioning with channel

random memory access. The experiment measures the elapsed time of doing the sequential and random scans on the memory buffer. And each scan, with different data types (e.g. short and long4), has four independent read instructions to saturate the memory subsystem. Almost each random memory access can cause a row-buffer miss [17], which severely degrades the overall memory performance.

Observation 4: The random memory access throughput is more sensitive to the data access unit size than the sequential memory access. Figure 5 shows the normalized throughput of different data types over the throughput with data access unit size of byte for both sequential access and random access. One finding is that the global memory bandwidth of random memory access is roughly proportional to the data access unit size. That is because each random memory access generates one real transaction to memory subsystem and causes one row-buffer miss. The number of memory transactions directly affects the memory performance, and the bandwidth of random accesses is almost proportional to the data access unit size. The trend can apply to the random memory transaction whose access unit size is smaller than the page size of DDR. In contrast, the sequential 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 the output pattern of data partitioning is random access, large data size should be used to more efficiently utilize the global memory bandwidth.

IV. Design and Implementation

Motivated by the observations, we have developed a multi-kernel approach by leveraging task kernel and channel. This section describes the details on the design and implementation. We present the overall design methodology, followed by the details on the data partitioning implementation and the cost model.

A. Overall Design Methodology

In order to address the first challenge of low lock overhead due to atomic operations in NDRange kernel, task kernel is considered to eliminate the atomic operation. However, one producer kernel of the producer stage is much faster than one consumer kernel of the consumer stage. In our test bed, the producer kernel achieves one cycle per tuple, while the consumer kernel can only deliver seven cycles per tuple. To resolve this performance mismatch, the consumer stage is implemented by multiple consumer kernels, each of which uses one dedicated channel to receive the tuples from the producer kernel of the producer stage, based on the result of the partitioning function on each tuple. Also, one consumer kernel dedicated for handling skewed data is required. Using this design, all the kernels of the consumer stage execute together to reduce the lock overhead of the consumer stage. In particular, each kernel handles only a subset of overall disjoint partitions.

In order to address the second challenge of low global memory bandwidth utilization, on-chip buffers are employed in all the kernels of the consumer stage to reduce the number of global memory transactions. Another potential benefit of the multi-kernel design is that the required amount of local memory can be distributed into each consumer kernel. It tends to achieve a higher frequency than that of one large on-chip buffer.

Both optimizations require certain amounts of FPGA resources (e.g. on-chip RAMs), and especially the design with on-chip buffers requires a large number of RAMs. Hence, how to efficiently utilize the limited FPGA resource is critical. We present a cost model to guide our design of multi-kernel partitioning on FPGAs.

B. Implementation of Multi-kernel Partitioning via Channel

As shown in Figure 6, the proposed architecture of multi-kernel partitioning comprises one *Data_in* kernel of the producer stage, and *DO Data_out* and one *Skewed_handling* kernels of the consumer stage to address the throughput imbalance between producer stage and consumer stage. The *Data_in* kernel (producer) reads the input tuples from global memory and then determines which *Data_out* or *Skewed_handling* kernel to deliver each input tuple based on the partitioning function, while the *Data_out* and *Skewed_handling* kernels (consumer) read from their corresponding channels and then store the received tuples to the corresponding partitions.

In our multi-kernel partitioning architecture, the buffered channels (*C_SKEW* and *C_PARTITION*), with the buffer depth (*CD*), are used to reduce the load imbalance between different kernels of the consumer stage.

The implementation details of *Data_in*, *Data_out* and *Skewed_handling* kernels are given below.

TABLE I
SUMMARY OF THE PARAMETERS USED IN COST MODEL

Model Parameter	Definition	Range on our test bed
N	Number of tuples for data partitioning	Input
P	Number of partitions of data partitioning	Input
W	Number of tuples for each memory read	64/tuple size
I	Issue rate of the *Data_in* kernel of the consumer stage	1
CD	Depth of buffered channel	0, 2, 4, 8, 16, 32
DO	Number of *Data_out* kernels of the consumer stage	1, 2, 4, 8,16
L	Number of cycles consumed by one tuple in one *Data_out* kernel	7
S_H	Number of cycles consumed by one tuple in one *Skewed_handling* kernel	1
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 memory transactions served per cycle	$\leq \#MemoryBanks$

1) Data_in Kernel (producer): The *Data_in* kernel executes the function as shown in Algorithm 2. Generally, its memory read order is sequential, and it is relatively easy to efficiently utilize the global memory bandwidth. In particular, it loads the W tuples (Line 2) each time. For each tuple, the key value (key) and the index ($partition\_index$) are computed (Line 4 and Line 5), then the tuple is transferred to the *Skewed_handling* kernel by using the API (*write_channel*) to write the tuple to the specific blocking channel *C_SKEW* (Line 8), when $partition\_index$ is equal to the index ($skewed\_index$) of the skewed partition. Otherwise, the tuple is transferred to the key-th *Data_out* kernel via the corresponding blocking channel *C_PARTITION[key]* (Line 13).

Algorithm 2: DATA_IN KERNEL

```
Input  : data_in (input table in global memory),
         N (number of input tuples),
         skewed_index (index of partition containing 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      for (i ← 0 to W ) do
4          key = channel_hash(tuples[i]);
5          partition_index = partition_hash(tuples[i]);
6
7          if (partition_index == skewed_index) then
8              write_channel(C_SKEW, tuples[i]));
9          else
10             #pragma unroll
11             for (j ← 0 to (DO − 1) ) do
12                 if (key == j) then
13                     write_channel(C_PARTITION[j], tuples[i]));
```

Algorithm 3: DATA_OUT KERNEL

```
Input  : N_x (number of tuples received by the Data_out kernel x),
         B (number of partitions),
         buckets (on-chip buckets for B partitions),
         counters (counter (local memory) for each partition).
Output : data_out (tuple output address in global memory)
1  for (i ← 0 to N_x ) do
       /* read one tuple from Data_in kernel. */
2      tuple = read_channel(C_PARTITION[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];
```

2) Data_out Kernel (consumer): The *Data_out* kernel executes the function as shown in Algorithm 3. In our design, we choose the OpenCL task kernel, where only one work item is active and the parallelism is determined at the compilation time. In particular, the lock overhead of one *Data_out* kernel is L cycles per tuple in our design (e.g. L = 7); that is, the kernel would read from its blocking channel every L cycles for the data partitioning. Since $counters$, which logs the counters of B partitions (e.g. B = 1024), are stored in the local memory, the critical path exists on the read/write updating of $counters$ (Line 4). In particular, the current tuple and the next tuple might belong to the same partition.

We resolve the overhead of random accesses by combining several original one-tuple write transactions into one many-tuples transaction to the global memory. Thus, the total number of memory transactions can be significantly reduced. In the *Data_out* kernel, on-chip buckets ($buckets$) are allocated in the local memory, and it can accommodate $B*S$ tuples, where B is the number of buckets in the *Data_out* kernel and the S is the slot size of tuples for each bucket existed in the on-chip memory. For each bucket, S tuples can be temporarily buffered on FPGA before they are stored back to the global memory in one write transaction. That is, the number of global memory write transactions is reduced by S times.

The working process of *Data_out* kernel can be summarized as follows. First, the kernel reads one tuple ($tuple$) from the blocking channel (*C_PARTITION*) connected to the *Data_in* kernel (Line 2), using the API (*read_channel*). Second, the index of the partition ($index$) is calculated (Line 3) and the corresponding counter is updated (Line 4). Third, $tuple$ is stored to the corresponding on-chip bucket (Lines 5-6). Fourth, if the on-chip bucket for the corresponding partition (with $index\_slot$) is full (equal to $S-1$), then the bucket is totally stored to the global memory (Line 8).

Algorithm 4: 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_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];
```

3) Skewed_handling Kernel (consumer): It handles the tuples, belonging to the skewed partition, as shown in Algorithm 4. The working process of *Data_handling* kernel can be summarized as follows. First, the kernel reads one skewed tuple ($tuple$) from the specific blocking channel (*C_SKEW*) connected to the *Data_in* kernel (Line 2). Second, the dedicated counter ($counter\_skew$) is updated (Line 3). Third, $tuple$ is stored to the dedicated on-chip bucket ($bucket\_skew$) (Lines 4-5). Fourth, if the on-chip bucket for the corresponding partition is full (equal to $S-1$), then the whole bucket is totally written to the global memory (Line 7) in one global memory transaction for the skewed partition.

The main difference between *Skewed_handling* kernel and *Data_out* kernel lies on the lock overhead. It 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.

C. Cost Model

Choosing the optimal configuration for various tuning parameters is an important and challenging task. In this subsection, we develop a cost model to estimate the execution time of multi-kernel partitioning which handles N tuples, with the corresponding parameters shown in Table I.

T_e is the estimated execution time of the multi-kernel partitioning implementation, as shown in Equation 1, where C_e is the estimated number of cycles required by the multi-kernel partitioning and $\#Freq$, which is the synthesized frequency of the multi-kernel partitioning, is obtained from the Altera complication report [1].

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

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

Since the lock-processing and global memory access cycles are overlapped, C_e is calculated as the larger value between the lock overhead cycles (C_{comp}) and global memory access cycles (C_{mem}), as shown in Equation 2.

1) Cost model for estimating C_{comp}: Since the kernels of the producer and consumer stages are executed in parallel, the estimation of C_{comp} is given in the Equation 3, where C_{in} is the estimated number of cycles required by the *Data_in* kernel of the producer stage and C_{out} is the estimated number of cycles required by all the kernels of the consumer stage.

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

C_{in} is estimated as shown in Equation 4, where the assumption is that the global memory has the sufficient memory bandwidth and can provide W tuples per cycle for the *Data_in* kernel. I means the issue rate of the *Data_in* kernel (e.g. one tuple per cycle) to the consumer stage, and N means the number of input tuples.

$$C_{in} = \frac{N}{Min(W, I)} \tag{4}$$

Since all the kernels of the consumer stage work concurrently, C_{out} is evaluated as shown in Equation 5, where the assumption is that the global memory has the sufficient memory bandwidth and all the memory transactions from all the kernels of the consumer stage can be immediately written to the global memory. N_i (or N_{skew}) is the number of tuples processed by the i-th *Data_out* kernel (or *Skewed_handling* kernel) and L (or S_H) is the number of cycles consumed by one tuple in one *Data_out* kernel (or *Skewed_handling* kernel). As a constraint, the total tuples processed by the consumer stage is N, as shown in Equation 6.

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

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

2) Cost model for estimating C_{mem}: Based on *observation 4* about the memory subsystems, the number of global memory transactions is a key performance indicator. Therefore, C_{mem} is evaluated in Equation 7, where Mem_trans stands for the total number of global memory read/write transactions, and TPC is the number of global memory transactions served per cycle.

$$C_{mem} = \frac{Mem\_trans}{TPC} \tag{7}$$

Mem_trans has two resources, one from *Data_in* kernel (left), and the other from *Data_out* or *Skewed_handling* kernels (right), as shown in Equation 8. W means that the number of global memory input transactions is reduced by W times and S means that the number of global memory output transactions is reduced by S times.

$$Mem\_trans = \frac{N}{W} + \frac{N}{S} \tag{8}$$

TPC is estimated as the sum of random and sequential memory transactions, as shown in Equation 9. TPC_{seq} and TPC_{rand} are the numbers of sequential and random global memory transactions handled by the memory subsystem per cycle. TPC_{seq} and TPC_{rand} are determined by the calibrations, as shown in Subsection III-B. In our experiments, in order to calibrate TPC_{seq} (or TPC_{rand}), we measure the elapsed time of the sequential (random) scan, using four load operations with *long8*, and calculate the result accordingly.

$$TPC = TPC_{seq} \times Rate_{seq} + TPC_{rand} \times (1 - Rate_{seq}) \tag{9}$$

$Rate_{seq}$ is the ratio of sequential memory transactions among all memory transactions, given in Equation 10.

$$Rate_{seq} = \frac{S}{S + W} \tag{10}$$

Parameter setting. Given the cost model, we can determine the suitable setting for a series of parameters, including S and DO. Since their ranges are reasonably small due to the limitation of FPGA resource, we consider all the possible combinations. For each combination, we calculate the cost model, and choose the setting with the smallest estimation cost. The parameter (e.g. CD) is not included in the cost

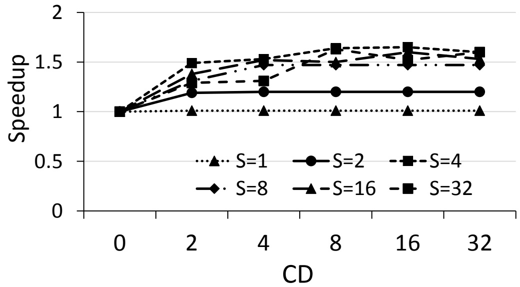


Fig. 7. Speedup of different channel depth (CD) over zero channel depth, with varying S

model, since the performance become stable when its value is sufficiently large (e.g. CD = 8), as shown in our experiments.

V. Experimental Evaluation

The experiments are divided into two groups, one on evaluating the impact of each parameter, and the other on evaluating the performance of our proposed design, as well as case studies.

A. Experimental Setup

Hardware configuration. Our experiments were conducted on a Terasic's DE5-Net board. It includes 4GB 2-bank DDR3 device memory, and an Altera Stratix V GX FPGA, with the Altera OpenCL SDK version 14.0. Each DDR3 bank has 64-bit width. The FPGA has 622K logic elements, 2560 M20K memory blocks (50Mbit) and 256 DSP blocks. The FPGA board is connected to the host via an X8 PCI-e 2.0 interface.

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

B. Performance with Different Parameter Combinations

Impact of CD. We first study the performance impact of the channel depth (CD). Figure 7 shows the speedup of data partitioning with varying CD over the case (CD = 0). Since the partitioning with different DO values has roughly the same trend as that of (DO = 8), we fix DO to be 8. The experimental result shows that the implementation with different S reaches its best performance when CD is greater than 4. Therefore, in the following experiments, we set CD to be 8.

Impact of S and DO. 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 the estimated execution time of data partitioning with different combinations of DO and S values, as shown in Figure 8. Our estimation is able to accurately capture the performance trend of different parameter combinations. With the accurate prediction, we are able to find the suitable parameter settings to achieve best data partitioning performance. We give more details about the performance trend of different parameter combinations.

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 case $DO = 4$, 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.

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 8, the *Data_in* kernel in the producer stage dominates the execution time. One interesting finding is about the case $DO = 16$ and $S = 16$. It is slower than the case $DO = 16$ and $S = 8$, since they roughly require the same number of cycles and the achieved frequency (267M) of the case $DO = 16$ and $S = 16$ is lower than that (296M) of the case $DO = 16$ and $S = 8$.

In summary, the performance bottleneck shifts for different settings on DO and S. Our model can capture the trend when the parameter setting changes.

Impact of *Skewed_handling* kernel. We study the impact of *Skewed_handling* kernel with varying the Zipf factor. Figure 9 shows the elapsed time of the data partitioning without the *Skewed_handling* kernel ("*original*") and the data partitioning with the *Skewed_handling* kernel ("*skewed_handling*"). The experimental result shows the effectiveness of skew handling when z is larger than 1, since the number of tuples in the skewed partition is large and the *skewed_handling* kernel handles the skew efficiently.

C. Performance Comparison

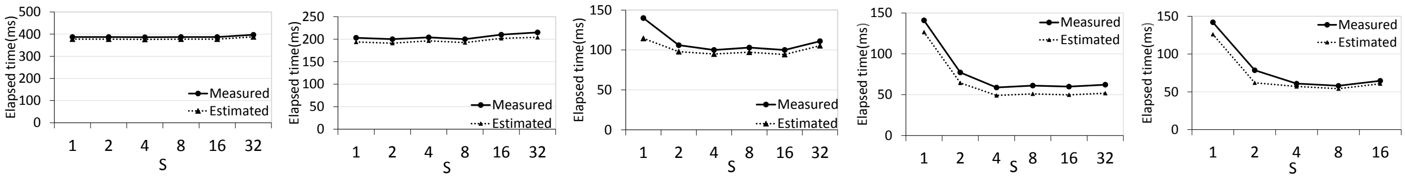
We study the performance of our multi-kernel partitioning, in comparison with the original data partitioning algorithm that has been presented in Section III. The multi-kernel approach is chosen with the parameters ($CD = 8$, $DO = 16$, $S = 8$, $B = 1024$), according to our cost model.

Impact of data size. Figure 10(a) shows the elapsed time of data partitioning with the input sizes (16MB, 32MB, 64MB, 128MB, 192MB). The number of partitions is 8K. The performance scales well for increasing data sizes. Our proposed multi-kernel approach is 10.7X faster than the original *local_1CU* implementation.

Impact of the number of partitions. Figure 10(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 multi-kernel approach is faster and more stable than the *local_1CU* implementation.

D. Case Studies for Data Partitioning

We study three case studies for data partitioning, including hash join, histogram and hash search. Those three operations are common in relational databases, and all of them use



(a) $DO = 1$ (b) $DO = 2$ (c) $DO = 4$ (d) $DO = 8$ (e) $DO = 16$

Fig. 8. Cost model evaluations with different settings for DO and S.

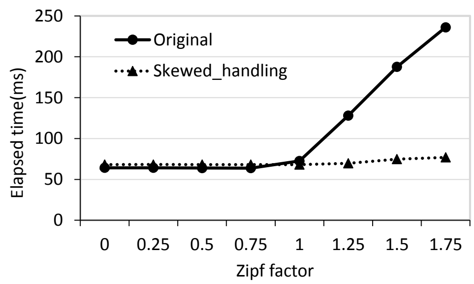
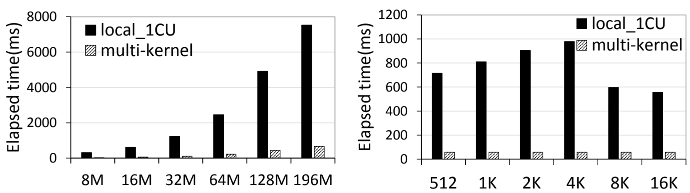


Fig. 9. Partitioning performance with varying Zipf factors



(a) Performance comparison with the varying number of tuples (b) Performance comparison with the varying number of partitions

Fig. 10. Performance comparisons between our proposal and the original implementation

data partitioning as a building block. In most cases, data partitioning is one of the major performance factors for those operations. We compare the performance between the one with the proposed data partitioning and one without the proposed data partitioning. The experiments on three case studies show that the optimized implementations can achieve 4-12X improvement over the original implementation. Due to the space limitation, we present the detailed results in our technical report [20].

VI. Conclusions

The OpenCL SDKs from FPGA vendors have become a significant leap on high level synthesis of FPGAs, due to the portability of OpenCL across heterogeneous platforms. We argue that existing OpenCL implementations that are specifically designed and optimized for CPUs/GPUs need to be carefully revisited on FPGAs. As a start, this paper focuses on data partitioning, one of the key and basic operations in relational databases. Our study reveals the significant overheads on locks and memory accesses of data partitioning on FPGAs. We develop a new multi-kernel partitioning approach together with on-chip buckets to address those overheads. Moreover, we develop a cost model to guide the parameter settings. Our results demonstrate 1) our cost model can accurately predict the performance of data partitioning under different parameter settings; 2) our proposed approach can achieve 10.7X speedup over the existing OpenCL implementation. The experiments on three case studies show that the optimized implementations can achieve 4-12X performance improvement over the original implementations.

VII. Acknowledgement

We thank Amelie Chi Zhou, Xuntao Cheng and Hui Yan Cheah for their comments on the paper. This work is in part supported by a MoE AcRF Tier 2 grant (MOE2012-T2-1-126) in Singapore.

References

[1] Altera. Altera SDK for OpenCL Optimization Guide. 2014.
[2] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A Secure Coprocessor for Database Applications. In *FPL*, 2013.
[3] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan. An Empirical Evaluation of High-Level Synthesis Languages and Tools for Database Acceleration. In *FPL*, 2014.
[4] C. Balkesen, J. Teubner, G. Alonso, and M. Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *ICDE*, 2013.
[5] A. Becher, F. Bauer, D. Ziener, and J. Teich. Energy-aware SQL Query Acceleration through FPGA-based Dynamic Partial Reconfiguration. In *FPL*, 2014.
[6] J. Casper and K. Olukotun. Hardware Acceleration of Database Operations. In *FPGA*, 2014.
[7] D. Chen and D. Singh. Invited Paper: Using OpenCL to Evaluate the Efficiency of CPUs, GPUs and FPGAs for Information Filtering. In *FPL*, 2012.
[8] 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 *FPL*, 2012.
[9] R. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating Join Operation for Relational Databases with FPGAs. In *FCCM*, 2013.
[10] J. He, M. Lu, and B. He. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. In *VLDB*, 2013.
[11] J. He, S. Zhang, and B. He. In-Cache Query Co-Processing on Coupled CPU-GPU Architectures. In *VLDB*, 2015.
[12] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. A Flexible Hash Table Design for 10Gbps Key-value Stores on FPGAs. In *FPL*, 2013.
[13] Khronos OpenCL Working Group. The OpenCL Specification. 2009.
[14] Loring Wirbel. Xilinx SDAccel A Unified Development Environment for Tomorrows Data Center. 2014.
[15] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main Memory Join on Modern Hardware. In *TKDE*, 2002.
[16] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. In *VLDB*, 2009.
[17] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.
[18] A. Shatdal, C. Kant, and J. F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*, 1994.
[19] D. Singh. Field-programmable Gate Array. *Altera Whitepaper*, 2011.
[20] Z. Wang, B. He, and W. Zhang. A Study of Data Partitioning on OpenCL-based FPGAs. Technical report, Nanyang Technological University, Singapore, http://pdcc.ntu.edu.sg/xtra/tr/2015-TR-Partitioning.pdf, 2015.
[21] L. Woods, Z. Istvn, and G. Alonso. Ibex-An Intelligent Storage Engine with Support for Advanced SQL Off-loading. In *VLDB*, 2014.