Accelerating Database Query Processing on OpenCL-based FPGAs

Zeke Wang Johns Paul Huiyan Cheah
Nanyang Technological University, Singapore

Bingsheng He
NUS

Wei Zhang
HKUST

***Abstract*—The release of OpenCL support for FPGAs represents a significant improvement in extending database applications to the reconfigurable domain. Taking advantage of the programmability offered by the OpenCL HLS tool, an OpenCL database can be easily ported and re-designed for FPGAs. A single SQL query in these database systems usually consists of multiple operators, and each one of these operators in turn consists of multiple OpenCL kernels. Due to the specific properties of FPGAs, each OpenCL kernel can have different FPGA-specific optimization combinations (in terms of CU (Compute Units) and SIMD (Kernel vectorization)) which are critical to the overall performance of query processing. Due to the resource limitation of an FPGA image, our query plan also considers the possibility of using multiple FPGA images. In this paper, we propose an FPGA-specific cost model to efficiently determine the optimal query plan in less than one minute. Our cost model can significantly reduce the FPGA synthesis time by avoiding the need to evaluate all the feasible query plans on real FPGAs. Our FPGA-specific cost model has two components: *unit cost* and *optimal query plan generation*. The first component generates multiple (unit cost, resource utilization) pairs for each kernel. The second component employs a dynamic programming approach to generate the optimal query plan which considers the possibility of using multiple FPGA images. The experiments show that 1) our cost model can accurately predict the performance of each feasible query plan for the input query, and is able to guide the generation of the optimal query plan, 2) our optimized query plan achieves a performance speedup 1.5×–4× over the state-of-the-art query processing on OpenCL-based FPGAs.**

I. Introduction

FPGAs have become an attractive and effective hardware accelerator for many relational database applications. Many of the previous studies, e.g., [7, 18, 19, 21, 25], have demonstrated significant performance improvement and superb energy efficiency. However, those systems are mostly implemented in low-level hardware description languages (HDLs) like Verilog and VHDL. The programmability issues of HDLs call for advanced high level synthesis (HLS). Recently, FPGA vendors such as Altera [8, 9] and Xilinx [22] have started to develop OpenCL SDKs. Since OpenCL explicitly exposes the data-level parallelism in the kernel programming, it is very suitable for developing database applications that inherently have extensive data parallelism. As a fact, a number of database systems have been re-designed for CPU/GPU in OpenCL (e.g., [15, 16, 37]). In this study, we investigate whether and how we can improve the query processing performance on OpenCL-based FPGAs.

A SQL query in OpenCL-based database systems usually consists of multiple operators, each of which consists of multiple OpenCL kernels. Due to the specific properties of FPGA, each OpenCL kernel has different optimization combinations, in terms of **CU** (Compute Units) and **SIMD** (Single Instruction Multiple Data). Each optimization combination then requires different amount of FPGA resources in LUTs, REGs, RAMs, and DSPs. Furthermore, if we enable a more aggressive optimization, which requires more FPGA resources, for one kernel, other kernels in the same query may not be able to implement desired optimizations due to the resource constraints or we may have to generate another FPGA image to hold these kernels. In the latter case, the FPGA switches from one image to another during query processing, which requires FPGA reconfiguration and buffer transfer via PCI-e. Therefore, finding an optimal query plan (optimization combination for each kernel) for the input query is essential to achieve good performance on OpenCL-based FPGAs.

In order to efficiently generate the optimal query plan for the input query, we present an FPGA-specific cost model, so that we do not need to iteratively evaluate all feasible query plans on FPGAs to determine the optimal query plan. Since there are many optimization combinations for each kernel, the number of feasible query plans is large. What makes search space even larger is that we also consider another dimension of multiple FPGA images to implement the input query. Since each FPGA image takes hours to synthesize, it is very time-consuming to evaluate all feasible query plans directly on FPGAs. Fortunately, our FPGA-specific cost model can generate the optimal query plan by estimating the execution time of feasible query plans and then choosing the query plan with minimum estimation time.

Our cost model has two components: *unit cost* and *optimal query plan generation*. First, we implement each operator kernel with different optimization combinations, each of which has multiple (unit cost, resource utilization) pairs. Second, based on (unit cost, resource utilization) pairs for each kernel, we present the dynamic programming based algorithm to determine the optimal query plan, which consists of the proper optimization combination for each kernel and the number of FPGA images.

One advantage of this layered approach is that we only need to re-run the dynamic programming based algorithm to determine the new query plan for the input query when one operator kernel is further optimized to have one better implementation. In particular, the implementation has smaller unit cost and relatively low resource utilization.

We integrate our the FPGA-specific cost model into OmniDB [15, 16, 37], and evaluate the proposed design on an Altera Stratix V GX FPGA. The experiments show that 1) our cost model is able to accurately predict the performance of each feasible query plan for the input query, and to guide the generation of optimal query plan, 2) the proposed cost model

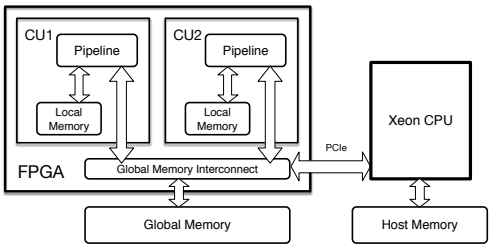


Fig. 1: Architecture Overview of Altera OpenCL SDK.

achieves a 1.5×–4× performance speedup in the database query processing over running OmniDB on FPGA.

The main contributions of this work are summarized as follows:

- We explore the implementations of each operator kernel using different FPGA-specific optimization combinations (such as CU and SIMD).
- We present a dynamic programming-based approach to generate the optimal execution plan for the input query, so that the optimal query plan (the proper optimization combination for each kernel and the number of FPGA images) can be determined in less than a minute.
- We evaluate the feasible execution plans with multiple FPGA images on OpenCL-based FPGAs.

The remainder of the paper is organized as follows. In Section II, we introduce the background of OpenCL-based FPGA and database query processing. In Section III, we present two observations of this study. We present the system overview in Section IV, and the details on the query plan generation in Section V. We present the experiment results in Section VI. We review the related work in Section VII and conclude and present our future work in Section VIII.

II. Background

A. Altera OpenCL SDK

Altera OpenCL SDK [1] abstracts away the complexities involved in programming FPGAs with HDL, and the FPGA bitstream file is directly created by compiling the input OpenCL kernel file. It takes hours to successfully generate a single bitstream file. With Altera OpenCL SDK, the FPGA is viewed as a massively parallel architecture and a single OpenCL kernel can have one or more kernel pipelines (i.e., compute units), which increases the parallelism of the kernel. An example kernel with 2 kernel pipelines is shown in Figure 1.

The FPGA memory hierarchy has three layers. a) Global memory with high latency and low bandwidth which resides in the DDRs of the FPGA board, b) Local memory with low latency and high bandwidth, and c) Private memory that stores data associated with each *work item*, existing in *Pipeline*, as shown in Figure 1. An OpenCL kernel consists of multiple work groups, and each work group consists of multiple work items. Both the local and private memories are located within the FPGA. Local memory has four banks and acts as scratch pad for one compute unit.

Recent OpenCL frameworks on FPGAs have already explored the three optimization methods that are originally proposed for GPUs: thread parallelism (**TP**), local memory (**SM**) and memory coalescing (**MC**). TP allow users to employ multiple work items to achieve the thread parallelism. SM employs local memory to buffer the intermediate data and then the number of global memory accesses is reduced. MC combines multiple global memory transactions with small data size into one coalesced global memory transaction so that the number of global memory accesses is reduced.

Besides, we also employ two FPGA-specific optimizations on our OpenCL kernels: compute unit (CU) and kernel vectorization (SIMD).

CU: If there is sufficient hardware resources available within the FPGA, then the kernel pipeline can be replicated to generate multiple compute units to achieve higher throughput. The inner hardware scheduler automatically dispatches the work groups among compute units.

SIMD: It can be applied to translate multiple scalar arithmetic operations to a single vector arithmetic operation. With SIMD, the number of total work items can be reduced, while each work item has the same amount of workload.

B. Query Processing

In relational databases, an SQL query is executed based on a query plan generated by the database system. A query plan is defined as an ordered set of steps (i.e., operators) that is used to retrieve and process data from a database. It is represented by a tree structure. For a database system implemented in OpenCL (OmniDB in this study), each operator is implemented as one or more OpenCL kernels.

When an SQL query is submitted to a database system, the system performs a number of steps for query processing, including parsing the input query to generate a query plan, optimizing the query plan and then evaluating the optimized query plan. In OmniDB, a query is evaluated by executing the OpenCL kernels for each of the operator in the query plan.

Figure 2 shows the query plan for the following query Q4, where 'key' (or 'payload') is an attribute of both the relations R and S, and 'Lo' ('Hi') is the lower (upper) bound for the selection.

```
Q4:  SELECT R.key, R.payload, S.payload  FROM R, S
     WHERE Lo <= S.key <= Hi AND R.key = S.key
     ORDER BY R.key
```

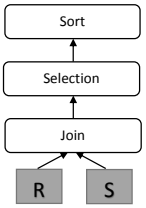


Fig. 2: A example query (Q4) and the corresponding query tree.

III. Observations

Our FPGA-specific cost model is motivated by the following two observations. First, for the same operator/query processing, different optimization combinations can result in significantly different resource consumptions and different execution times. Second, FPGA reconfiguration overhead is an important performance factor for generating the optimal query plan when multiple FPGA images are considered. The detailed experimental setup can be found in Subsection VI-A.

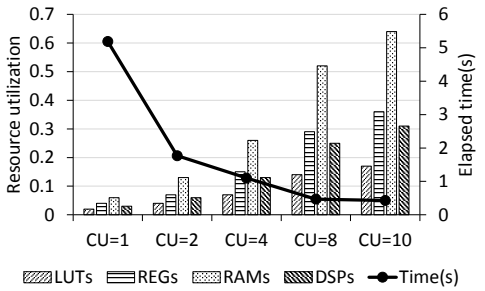


Fig. 3: The *scanLargeArrays* kernel with different optimization combinations has different performances and resource utilizations.

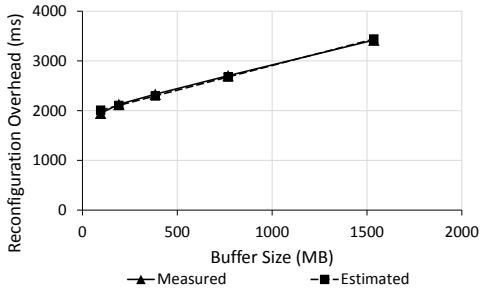


Fig. 4: Measured and estimated values of reconfiguration overhead

A. Impact of Optimization Combination

The *scanLargeArrays* kernel is one of three kernels belonging to the primitive prefix scan (i.e., prefix sum) [37]. Figure 3 shows the impact of different optimization combinations on the execution time and resource utilization of the *scanLargeArrays* kernel when the input table has 128M tuples. We give the utilization of LUTs, REGs, RAMs and DSPs, and also present the execution time, where $CU = x$ stands for x number of CUs for the kernel *scanLargeArrays*. The exact meanings of the *scanLargeArrays* kernel are not important here, and we present more details in Section IV-A1. The results clearly show that different optimization combinations result in very different execution times as well as the resource usages for the *scanLargeArrays* kernel. A considerable decrease in total execution time can be achieved by applying more aggregative optimization (which results in an increase in resource utilization).

B. FPGA Reconfiguration Overhead

According to Altera, the FPGA reconfiguration overhead includes the following two components for current OpenCL-based FPGA boards.

The first component of the reconfiguration overhead is the time taken to fully reconfigure the FPGA, denoted by *F_O*. In particular, the new bitstream is transferred to FPGA context. Then, the PCIe bus and DDR controller are re-initialized. Since this bitstream loading delay is generally stable, we model it as a constant.

The second component includes the time to transfer the active contents (memory footprint) of the FPGA memory to host memory via PCIe before the full reconfiguration and the time to transfer the active contents from host memory to FPGA memory after the full reconfiguration. This transfer of active contents is needed since FPGA memory contents can be corrupted during the re-initialization of DDR controller. Also, the Altera FPGA under study does not support runtime partial reconfiguration when using OpenCL. The time for this component is linear to the memory footprint, with respect to the bandwidth of PCI-e bus. The corresponding ratio of the time to the memory footprint is denoted by *U_T*.

Therefore, the relationship between the FPGA reconfiguration overhead (denoted by *reconf_overhead*) and the FPGA memory footprint (denoted by *buffer_size*) can be linear, as shown in Equation 1.

$$reconf\_overhead = U\_T * buffer\_size + F\_O \quad (1)$$

In order to identify *U_T* and *F_O*, we collect five pairs of training data sets, each of which is of the form (*buffer_size*, *reconf_overhead*), as shown in Figure 4.

We use a linear regression model with the least square fitting to predict the reconfiguration overhead. Figure 4 shows the estimated and the measured reconfiguration overhead of the tested FPGA. The linear regression model achieves a reasonably high accuracy in predicting the reconfiguration overhead. And we observe that *U_T* is 0.993 ms/MB and *F_O* is 1914.6 ms.

IV. System Overview

Two observations in Section III, motivating our study of query processing on OpenCL-based FPGAs, are 1) more aggressive optimizations which require more FPGA resources can have better performance for each kernel which means multiple FPGA images can reduce the execution time of the input query, 2) the latency for FPGA reconfiguration is a non-negligible overhead which can impact performance of query processing on OpenCL-based FPGAs. Therefore, there is an FPGA-specific performance tradeoff. Hence, how to efficiently utilize FPGA resources, by using appropriate optimization combination for each kernel and multiple FPGA images, is critical to achieve the good performance on OpenCL-based FPGAs. Our FPGA-specific cost model can generate the optimal query plan by estimating the execution times of all the feasible query plans for the input query plan and then choosing the query plan with minimum estimation time.

In the following, we present the implementation details of query processor as well as the FPGA-specific cost model.

A. Implementation of Query Processor

As a start, we study OmniDB, state-of-the-art databases designed and implemented in OpenCL [37]. As OmniDB is originally designed for CPU/GPU, we need to revisit its design and implementation on the abstraction of OpenCL SDK on FPGAs, as shown in Figure 5.

The query processor of OmniDB is implemented using a layered design: *storage*, *data-parallel primitives*, *access methods*, and *operators*. The primitives are data-parallel operations which are used as basic blocks for implementing operators. We adopt the implementation of primitives and operators from OmniDB [37], and focus on the impact of FPGA-centric optimization techniques to the existing implementations. More details of the design and implementation of individual kernels can be found in [37]. One advantage of the primitive-based approach is that all the corresponding operators can have the speedup when the specific primitive is accelerated. We briefly describe the primitives and operators on FPGAs.

TABLE I: Summary of parameters

Name	Definition
N	Number of kernels at the operator kernel array
K_i	Kernel i at the kernel array, $1 \leq i \leq N$
S_i	Kernel array with kernels (K_1, ..., K_i), $1 \leq i \leq N$
$T^{one}_{j...i}$	Minimum execution time for kernels K_j, ..., K_i in one image
$C^{one}_{j...i}$	The optimization combination for kernels K_j, ..., K_i in one image with minimum execution time
R_i	Reconfiguration overhead when kernel K_i has one new image
T^{dp}_i	Minimum execution time to compute the kernel array S_i (considering multiple FPGA images), T^{dp}_0 = 0
Z_j	Number of candidate implementations with different optimization combinations for kernel K_j
P_j	Index of candidate implementation for kernel K_j,$P_j < Z_j$
$\hat{P}_j$	Index of candidate implementation for kernel K_j in the FPGA image which has the minimum execution time
$E_{P_j...P_i}$	Execution time for kernels with the optimization combinations ($P_j, \ldots, P_i$) in one image
$F_{P_j...P_i}$	Image Frequency for kernels with the optimization combinations ($P_j, \ldots, P_i$) in one image
$Q_{P_j...P_i}$	Cycles for kernels with the optimization combinations ($P_j, \ldots, P_i$) in one image
C_i	Information (optimization combination for each kernel and number of FPGA images) of kernel array S_i
U_{P_j}	Unit cost for the candicate implementation with optimization combination P_j of kernel K_j
W_j	Number of input tuples for the kernel K_j
V^{inc}_j (V^{dec}_j)	Unit ratio of increasing (decreasing) memory amount for the kernel K_j
G_i	Memory footprint for the kernel K_i of the kernel array

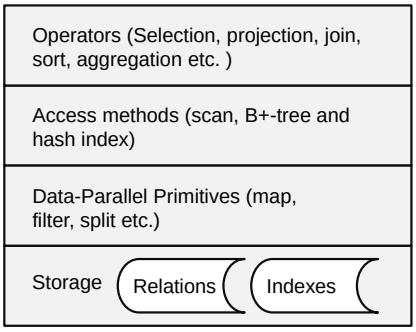


Fig. 5: The layered design of query processor, adopted from *OmniDB* [37]

1) Primitives: The primitives, which form the relational operators, are implemented using OpenCL kernels with different optimization methods for query processing.

Map: The *map* primitive applies the input map function to every tuple in the input relation. The implementation, with only one OpenCL kernel (*map*), has already explored the two optimization methods: TP and MC. Both can lead to good performance on GPUs and FPGAs. Besides, we also explore one FPGA-specific optimization method (CU) to accelerate the kernel performance of *map* primitive. However, SIMD cannot work since the memory output address is random.

Scatter and Gather: The *scatter* primitive performs sequential reads from the input relation and indexed writes to the output relation with input location array, while the *gather* primitive performs indexed reads from the input relation with input location array and performs sequential writes to the output relation. When the input locations are random, *scatter* and *gather* can both have the property of random memory access. Therefore, the existing optimal implementations for GPUs employ multi-pass optimization scheme [14] to efficiently utilize the GPU cache and then improve the temporal locality of indexed memory accesses. However, as FPGAs do not possess the cache hierarchy of CPUs/GPUs, we use the one-pass implementation. The implementation of *scatter* (or *gather*) primitive, with only one OpenCL kernel *scatter* (or *gather*), has already explored the two optimization methods: TP and MC. Besides, we also explore one FPGA-specific optimization method (CU) to accelerate their performances. However, SIMD cannot work since the memory input (or output) address of *gather* (or *scatter*) primitive is random.

Prefix scan: It is an important building block for many parallel database applications [13], such as filter and aggregation. Its parallel implementation [37] contains three OpenCL kernels (*scanLargeArrays*, *prefixSum* and *blockAddition*), and they are executed sequentially. The implementations of *scanLargeArrays* and *prefixSum* kernels have already explored the three optimization methods: TP, SM and MC. Besides, we also explore one FPGA-specific optimization method (CU) to accelerate the two kernels. However, SIMD cannot work since consecutive work items suffer from path divergences. The implementations of *blockAddition* kernel has already explored the two optimization methods: TP and MC. Besides, we also explore two FPGA-specific optimization methods (CU and SIMD) to accelerate the kernel. Because the execution time of kernel *prefixSum* is very small compared with other kernels in this primitive, it has only one optimization combination.

Filter: The *filter* primitive produces a subset of tuples from a input relation with the input filter condition. Its implementation contains three primitives (*map*, *prefix scan* and *scatter*), which are executed sequentially. The detailed implementation for each primitive has been described above.

Reduce: The *reduce* primitive computes a value (using the specific arithmetic function) from input relation, based on the specific key. Its implementation [37] utilizes the optimization method SM to reduce the number of global memory accesses. Besides, for the implementation on OpenCL-based FPGAs, we also explore the FPGA-specific optimization method (CU) to accelerate performance. However, SIMD cannot work since consecutive work items suffer from path divergences.

Sort: The *sort* primitive transforms the input relation of unordered tuples into the output relation of tuples ordered based on the specific input key. Based on the implementation from OmniDB, we further apply the optimization method SM to improve the temporal locality and then improve the sorting performance. We also employ optimization method (kernel fusion) to merge the similar kernels and then only two OpenCL kernels (*bitonicSortShared* and *bitonicMergeGlobal*) are required in our implementation. They are executed repeatedly. The kernel (*bitonicSortShared*) employs the optimization method SM. Besides, both kernels can explore the optimization

method (CU), where *L_CU* stands for the number of CUs for kernel (*bitonicSortShared*) and *G_CU* stands for the number of CUs for kernel (*bitonicMergeGlobal*). However, SIMD cannot work for two kernels since consecutive work items suffer from path divergences.

2) Relational Operator: Each relational operator is comprised of one or more primitives. We briefly present the implementation for completeness.

Selection: The relational operator *selection* is implemented by the primitive *filter*, and the predicate evaluation of *selection* is corresponding to the filter function of primitive *filter*.

Order-by: The relational operator *Order-by* is implemented by the primitive *sorting*.

Grouping and Aggregation: The *grouping* is implemented by the two primitives *sort*, *prefix scan* and the kernels *scanGroupLabel*, *groupByImpl_write*. We explore the FPGA-specific optimization method (CU) to accelerate both kernels (*scanGroupLabel* and *groupByImpl_write*). The *aggregation* is implemented by the primitive *reduce*.

Joins: The relational operator *hash join* takes two relations as input, and finds the matching tuple pairs from the two relations according to the join predicate. In the paper, we mainly focus on the simple hash join [37]. It has two kernels, *buildTable* and *probeTable*. We explore the FPGA-specific optimization method (CU) to accelerate both kernels. Besides, we also explore the FPGA-specific optimization method (**Local**) to use local memory based lock instead of the default global memory based lock.

B. FPGA-specific Cost Model

In order to efficiently generate the optimal query plan, our FPGA-specific cost model needs to estimate the execution times of feasible query plans for the input query and then chooses the query plan with minimum execution time.

The FPGA-specific cost model, following layered design, has two components: *unit cost* and *optimal query plan generation*. The description of each component is shown in the following.

1) Unit Cost: Since the specific implementations of OpenCL kernels are not made available to users, it is very difficult to accurately develop an analytical model for each database operator kernel. Therefore, we treat the OpenCL-based FPGAs as a black box and measure the unit cost of each operator kernel with different optimization combinations, each of which requires different amount of FPGA resources. The unit cost of each kernel is supposed to be the number of total clock cycles (not the elapsed time) divided by the number of tuples in the input relation, since the evaluated kernel can have different frequency when it stays at different FPGA images each of which may have various other OpenCL kernels in the practical implementation. We experimentally measured the unit costs of each kernel with different optimization combinations. We calculate the total cost of a kernel as the unit cost multiplied by the number of input tuples for the kernel. Due to the layered design of OmniDB, we can estimate the cost of each primitive/operator similarly.

For each primitive, we study the unit cost of applying different optimization techniques to its implementation. In particular, we log down each implementation with the format: (CU, SIMD, LEs, REGs, BRAMs, DSPs, Unit cost). Clearly, we observe that different optimization techniques result in very different unit costs, which should be factored into the query plan generation.

2) Optimum Query Plan Generation: The input query consists of multiple operators, each of which consists of multiple OpenCL kernels. Since each kernel has multiple implementations (unit cost, resource utilization), we present a dynamic programming based approach to determine the optimal query plan, which consists of the proper optimization combination for each kernel and the number of FPGA images. Therefore, the input query can achieve the best performance on OpenCL-based FPGAs. The implementation details are described in Section V.

3) Advantage of Layered Design: The layered design of our cost model has the advantage of effectively handling various optimizations within operators. When an operator is further optimized with any of the FPGA-centric optimizations (discussed in Subsection IV-A), the cost model only needs to profile new (unit cost, resource utilization) pairs and then rerun the query plan generation. People can still keep exploring other FPGA-specific optimizations, e.g., kernel fusion and loop unrolling, to further accelerate primitives on FPGAs, and then produce more efficient (unit cost, resource utilization) pairs.

V. Generation of Optimum Query Plan

In this section, we firstly present the problem formulation, secondly the implementation of our dynamic programming based approach to generate the optimal query plan, thirdly performance evaluation on a single FPGA image containing multiple kernels, and finally reconfiguration overhead when FPGA reconfiguration happens. Table I summarizes the key parameters in the section.

A. Problem Formulation

Given the input query, our goal is to achieve the optimal query plan which requires the minimum execution time on OpenCL-based FPGAs. Since the input query, in terms of operators, is represented using an operator tree, we employ the topological sorting [20] to generate all the feasible solutions (operator arrays). We evaluate all the feasible operator arrays and then select the optimal operator array which requires the minimum execution time.

The operator array consists of M relational operators (O_1, O_2, ..., O_M), which are executed one by one on OpenCL-based FPGAs. Since each operator O_i consists of n_i OpenCL kernels the operator array is converted to a kernel array with N OpenCL kernels (K_1, K_2, ..., K_N). In this paper, we do not consider the case where several operator kernels execute concurrently, concurrent execution of several kernels cannot always achieve better performance due to the memory interference, given the very limited memory bandwidth on FPGAs.

Therefore, the original problem of generating the optimal query plan is converted to the problem of minimizing the execution time (denoted as T_N^{dp}) of the kernel array.

B. Dynamic Programming Approach

Dynamic programming is an effective algorithmic design approach for complex problems [4]. In the dynamic programming approach, a complex problem (finding the optimal query plan among all the feasible query plans) is solved by breaking it down into a series of simpler subproblems (minimizing the execution time of kernel sub arrays), solving

Algorithm 1: DYNAMIC PROGRAMMING ALGORITHM

```
   Input  : N, T^one_{j...i}, C^one_{j...i} and R_i as defined in Table I.
   Output : C_i and T^dp_i as defined in Table I.
1  T^dp_0 = 0;
   /* Compute optimal substructure T^dp_i at the loop i     */
2  for (i ← 1 to N) do
       /* T_min is initialized to be ∞.                     */
3      T_min ← ∞;
4      for (j ← 0 to i − 1 ) do
5          T_temp = T^dp_j + T^one_{j+1...i} + R_{j+1};
6          if (T_temp < T_min) then
7              T_min = T_temp;
8              config_tmp = {C_j, j, R_{j+1}, C^one_{(j+1)...i}};
9          end
10     end
11     T^dp_i = T_min;
12     C_i = config_tmp;
13 end
```

these subproblems and then using the solution of these sub-problems to evaluate the result of the complex problem. The overlap of these sub-problems is used to reduce the computation time of our program. Now we define T_i^{dp} to be the minimum execution time for the kernel array with i kernels (1,..., i), where i ranges from 1 to N. The corresponding implementation is shown in Algorithm 1.

Since the kernel array S_0 is empty, T_0^{dp} is set to 0 (Line 1). T_i^{dp} is computed during the i^{th} iteration of loop i (Lines 2-13). Once T_N^{dp} is computed, the loop terminates and the optimal query plan is generated. Also initially T_min is set to infinity (Line 3) to make sure that all feasible solutions are considered.

Each iteration of the loop j (Lines 4-10) evaluates one feasible solution for the sub-array (K_1, K_2, ..., K_i). j^{th} iteration of the loop j evaluates the execution time of the kernel sub array S_i as the sum of three components (Line 5). The first component is the minimum execution time of the kernel sub-array S_j (T_j^{dp}), the second component is the execution time of the kernel array (K_{j+1}, K_{j+2}, ..., K_i) using a single FPGA image ($T_{j+1...i}^{one}$), and the third component is the reconfiguration overhead (R_{j+1}). This sum is then stored in T_temp. To evaluate T_temp in each iteration of the loop j we employ the concept of dynamic programming by reusing the value of (T_j^{dp}) computed in the previous loop i.

When T_temp is smaller than T_min (Line 6), T_min is set to be T_temp (Line 7) and the corresponding configuration information, including C_j, j, R_{j+1}, and $C_{(j+1)...i}^{one}$ are stored in $config\_tmp$ (Line 8). Since loop j iterates through every possible implementation for the sub-array of i kernels T_min is guaranteed to be the minimum possible execution time for these kernels. Finally, T_i^{dp} is set to be T_min (Line 11), and C_i to $config\_tmp$ (Line 12).

C. Evaluating $T_{j...i}^{one}$ for One FPGA Image

In this subsection, we evaluate $T_{j...i}^{one}$ and the optimal configuration $C_{j...i}^{one}$ by executing all possible implementations of kernels (K_j, K_{j+1}, ..., K_i) as a single FPGA image.

Since the FPGA resources are limited the number of candidate implementations with different optimization combinations for each kernel is reasonably small. Therefore, we estimate each possible optimization combination $(P_j, P_{j+1}, \ldots, P_i)$, where $0 < P_j < Z_j$ and $0 < P_i < Z_i$. Then, we choose the combination $(\hat{P_j}, P_{j+1}, \ldots, \hat{P_i})$ with the minimum execution time ($T_{j...i}^{one}$), as shown in Equation 2, where Z_q is the number of implementations with different optimization combinations for the kernel K_q (q from j to i).

$$T_{j...i}^{one} = \min_{\forall P_j \le Z_j, \ldots, \forall P_i \le Z_i} E_{P_j \ldots P_i} \tag{2}$$

Since the specific implementation of one kernel can interfere with the other implementations of the other kernels, each combination can achieve different frequency. Therefore, the corresponding execution time is equal to the estimated number of cycles $Q_{P_j \ldots P_i}$ divided by the estimated frequency $F_{P_j \ldots P_i}$, as shown in Equation 3, where $Q_{P_j \ldots P_i}$ and $F_{P_j \ldots P_i}$ are estimated in the following descriptions.

$$E_{P_j \ldots P_i} = \frac{Q_{P_j \ldots P_i}}{F_{P_j \ldots P_i}} \tag{3}$$

1) Estimation of Frequency: It is hard to present an accurate analytical model to estimate the hardware frequency for each optimization combination $(P_j \ldots P_i)$ for kernels $(K_j \ldots K_i)$. It is used to roughly predict the performance of the feasible query plan (in terms of FPGA image), which has already demonstrated the accuracy in determining the optimal query plan.

One observation is that there is a strong correlation between the resource utilizations of four FPGA features and the actual hardware frequency. Each feature (s) belongs to the feature set F = (LE, REG, RAM, DSP) [1]. We propose a resource oriented approach to estimate the frequency. In particular, we employ the largest resource utilization ($R_{P_j \ldots P_i}$) of any feature in the feature set to estimate the frequency, as shown in Equation 4, where $L_s^{P_k}$ is the number of resources of the feature s for the kernel k with the implementation P_k (its value is collected during the kernel profiling at Subsection IV-B1). Next, $(L_s^{min} + \sum_{k=j}^{i} L_s^{P_k})$ is the estimated number of resources of the feature s, required by the FPGA image which has the kernels $(K_j \ldots K_i)$ with the optimization combination $(P_j \ldots P_i)$.

$$R_{P_j \ldots P_i} = \max_{\forall s \in F} \frac{(L_s^{min} + \sum_{k=j}^{i} L_s^{P_k})}{L_s^{max}} \tag{4}$$

L_s^{min} is the number of resources for the feature s when the FPGA image has no OpenCL kernel in order to have the minimum resource usage. They are initially used by DDR and PCI-e controller. In our test FPGA, we obtain L_{LE}^{min}, L_{REG}^{min},L_{RAM}^{min}, and L_{DSP}^{min} to be 35437, 54855, 283, and 0, respectively. L_s^{max} is the number of available resources for the feature s of our test FPGA. L_{LE}^{max}, L_{REG}^{max},L_{RAM}^{max}, and L_{DSP}^{max} [2] are 636928, 944128, 2560, and 256, respectively.

The training data sets are collected by the 39 actual FPGA images from our experiments. For each FPGA image, we obtain the maximum resource utilization and the actual frequency. Thus, we develop a quadratic regression model to predict the hardware frequency ($F_{P_j \ldots P_i}$) based on the largest resource utilization ($R_{P_j \ldots P_i}$), as shown in Equation 5. This regression model works well in query processing and can be factored into the optimal query plan generation.

$$F_{P_j \ldots P_i} = -86 \times (R_{P_j \ldots P_i})^2 - 22 \times R_{P_j \ldots P_i} + 279(MHz) \tag{5}$$

TABLE II: Unit ratio of memory footprint for each kernel

Kernel	V^{inc}	V^{dec}
sort	0	0
buildTable	2	1
probeTable	1	1
groupBy_write	SEL_RATE	2
GroupLabel	1	0
ScanLargeArrays	1.01	0
map	1	0
reduce	SEL_RATE	1+SEL_RATE
prefixSum	0.01	0
blockAddition	0.01	0
gather	0	1.02

2) Estimation of Clock Cycles: Based on the property of unit cost in Subsection IV-B, the estimated number of cycles for the kernel K_k is equal to $U_{P_k} \times W_k$, where P_k is the index of candidate optimization combination for the kernel K_k, and W_k is the number of input tuples for the kernel K_k.

Since the kernels (K_j, ..., K_i) execute sequentially, the required number of cycles for all the kernels with the optimization combination ($P_j \dots P_i$) is estimated as shown in Equation 6.

$$Q_{P_j \dots P_i} = \sum_{k=j}^{i} (U_{P_k} \times W_k) \tag{6}$$

D. Evaluating Reconfiguration Overhead

R_i is the FPGA reconfiguration overhead when the kernel K_i belongs to the new FPGA image. Based on the second observation, R_i is evaluated as shown in Equation 7, where $tuple\_size$ stands for the size of each tuple and G_i is the corresponding device memory footprint of the i-th kernel K_i of the input kernel array. The detailed implementation of G_i is descripted in the following paragraph. When i is equal to 1, R_1 is 0, since no FPGA image reconfiguration happens.

$$R_i = \begin{cases} 0.993 \times tuple\_size \times G_i + 1914.6, & i > 1 \\ 0, & i = 1 \end{cases} \tag{7}$$

When executing the i-th kernel K_i of the input kernel array, the corresponding device memory footprint (G_i) is evaluated as shown in Equation 8, where W_1 means the input data size and W_j is the number of input tuples for the kernel K_j. For the kernel K_j, V_j^{inc} (or V_j^{dec}) is the unit ratio of increasing (decreasing) memory amount to the number of input tuples of the kernel K_j. Therefore, V_j^{inc} (or V_j^{dec}) is fixed and the increasing (decreasing) memory amount for the kernel K_j is equal to $V_j^{inc} \times W_j$ ($V_j^{dec} \times W_j$). We collect the unit ratio of each operator kernel during the benchmarking, as shown in Table II, where *SEL_RATE* means the selectivity.

$$G_i = W_1 + \sum_{j=1}^{i-1} ((V_j^{inc} - V_j^{dec}) * W_j) + V_i^{inc} * W_i \tag{8}$$

VI. Experimental Evaluation

In this Section, we firstly present the experimental setup, secondly we evaluate our cost model which generates the optimal query plan, and thirdly evaluate the overall performance speedup of our proposed design, in comparison with the original OmniDB [37].

A. Experimental Setup

Hardware configuration. We conduct our experiments on a Terasic DE5-Net board with an Altera Stratix V FPGA and 4GB 2-bank DDR3 device memory. We design our kernels using Altera OpenCL SDK version 14.0. The FPGA board is connected to the host via a x8 PCI-e 2.0 interface.

Workloads. Four queries (Q1, Q2, Q3 and Q4) are evaluated, as shown in Table III. Q1 is a simple query with only the *selection* operator, Q2 uses the *grouping and aggregation* operator, Q3 is a complicated query with two operators, *selection* and *grouping and aggregation*, and Q4 also has 3 operators, *join*, *filter* and *order-by*.

We use column stores for query processing on OpenCL-based FPGAs. Each tuple of input relations `R` and `S` has the format <key, payload>. Both keys and payloads are random 4-byte integers and the number of tuples ranges from 1M to 128M so that the input can fit into the FPGA memory. However, the number of input tuples for Q4 ranges from 1M to 64M because input data with 128M tuples cannot fit inside the FPGA because of the larger memory footprint of Q4 as compared to the other three queries.

TABLE III: Evaluated Queries

ID	Queries
Q1	`SELECT * FROM S` `WHERE Lo ≤ S.key ≤ Hi`
Q2	`SELECT S.key, MAX(S.payload) FROM S` `GROUP BY S.key`

Q3	`SELECT S.key, SUM(S.payload) FROM S`
	`WHERE Lo≤S.paylaod ≤ Hi`
	`GROUP BY S.key`

Q4	`SELECT R.payload, S.payload FROM R, S`
	`WHERE Lo≤S.key≤ Hi AND R.key=S.key`
	`ORDER BY R.payload`

B. Cost Model Evaluation

In this subsection, our cost model firstly analyzes the relationship between optimization combination and unit cost for each kernel, secondly generates the proper query plan for each query, thirdly analyzes the break-even points for each query and finally we present the performance breakdowns.

1) Unit Cost: We exhaustively evaluate all the optimization combinations subject to two conditions: (i) more aggressive optimization combinations are considered only if they improve the performance of the kernel when compared to a less aggressive optimization combination and (ii) the resulted hardware must be able to fit into one FPGA. There are instances where a combination produces better performance, but the generated hardware exceeded the resource capacity of our FPGA. In particular, we take the kernel *blockAddition* of the *prefix scan* primitive as an example.

Nine optimization combinations were explored, and the relationship between resource consumption and unit cost is shown in Table IV. Different optimization combinations may have different unit cost for the same kernel. More SIMDs always yield better performance since the memory transactions are coalesced and then the total number of memory transactions is reduced. One interesting finding here is that more CUs cannot yield better performance since the kernel *blockAddition*

TABLE IV: Unit cost for blockAddition

CU	SIMD	LUTs	REGs	RAMs	DSPs	Unit cost
1	1	7275	10305	80	0	1.05
1	2	7010	10163	80	0	0.53
2	1	13689	19462	160	0	0.66
2	4	7010	10163	80	0	0.53
1	8	7395	10877	80	0	0.14
4	8	26997	40064	320	0	0.12
1	16	8240	12108	80	0	0.11
8	4	50333	74556	640	0	0.11
2	16	15619	23068	160	0	0.19

is memory-intensive and more CUs do not reduce the number of global memory transactions. For example, the unit cost of the optimization combination (*CU*=1, *SIMD*=16) is less than that of combination (*CU*=2, *SIMD*=16). Therefore, the first condition is satisfied and we do not need to try more aggressive optimizations (more CUs) even when the FPGA has enough resource to support them. The optimization combination (*CU*=1, *SIMD*=16) can roughly achieve the best performance with reasonable FPGA resource requirement.

Because of page limitation, we cannot analyze all the eleven kernels used in the four queries. So we only present the number of optimization combinations (*Number* field) and the best optimization combination (with *LUTs*, *REGs*, *RAMs*, *DSPs*, *UC* fields) for each kernel (shown in Table V). The *Number* filed shows the exploration space (optimization combinations) for each kernel.

In the experiments, we observe significant performance differences when varying the parameter values for CU and SIMD. Thus, it is important to have a cost model guided approach to guide the query plan generation.

2) Query Plan Generation: For each query (Q1, Q2, Q3 or Q4), we choose the kernel array which has the minimum execution time, as shown in Figure 6. The detailed implementation of each kernel can be found on OmniDB [37].

Each kernel might have different optimization combinations (in terms of CU and SIMD) and then have different unit costs. In the following, we briefly present the details on how to determine the optimization combination for each kernel and number of FPGA images, so that the execution time for the input query is minimized. Overall, our cost model can effectively guide the optimal query plan generation for all the tested queries upon different configurations.

Q1. Our cost model recommends that one FPGA image is sufficient to implement Q1, and the corresponding optimizations for five kernels are 2 CUs for *map*, 8 CUs for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition* and 2 CUs for *gather*.

Q2. When the number of input tuples is less than 16M, the cost model recommends the *Execution Plan 1* which contains only one FPGA image, and the corresponding optimizations for the seven kernels are 4 S_CUs and 2 G_CUs for *sort*, 1 CU for *scanGroupLabel*, 1 CU for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition*, 1 CU for *gather* and 1 CU for *reduce*.

When the number of input tuples is more than 16M, the cost model recommends the *Execution Plan 2* which contains two FPGA images. The first image contains only the *sort* kernel, whose optimizations are 10 S_CUs and 2 G_CUs. The second image contains six kernels and the corresponding optimizations are 8 CUs for *scanGroupLabel*, 2 CUs for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition*, 8 CUs for *gather* and 1 CU for *reduce*.

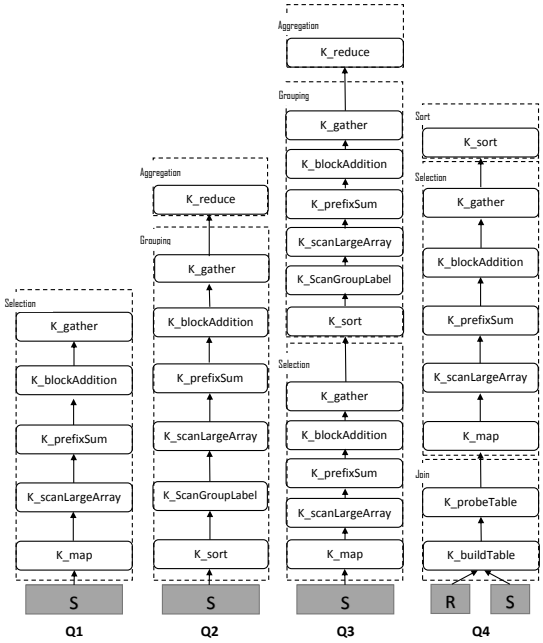


Fig. 6: Queries implemented with kernels

Q3. When the number of input tuples is less than 16M, the cost model recommends the *Execution Plan 1* which contains only one FPGA image, and the corresponding optimizations are 1 CU for *map*, 1 CU for *gather*, 2 S_CUs and 1 G_CU for *sort*, 1 CU for *scanGroupLabel*, 2 CUs for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition*, 1 CU for *gather*, and 1 CU for *reduce*. The estimated and measured resource consumptions and frequency of the FPGA image (*Execution Plan 1*) are shown in Table VI.

When the number of input tuples is more than 16M, the cost model recommends the *Execution Plan 2* which contains three FPGA images. In particular, the first image has five kernels and the corresponding optimizations are 2 CUs for *map*, 8 CUs for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition*, and 4 CUs for *gather*; the second image contains only one kernel *sort*, whose optimizations are 10 S_CUs and 2 G_CUs. The third image contains six kernels and the corresponding optimizations are 8 CUs for *scanGroupLabel*, 2 CUs for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition*, 8 CUs for *gather* and 1 CU for *reduce*. The estimated and measured resource consumptions and frequencies of two FPGA images (*Execution Plan 2*) are shown in Table VI.

Q4. When the number of input tuples is less than 16M, the cost model recommends the *Execution Plan 1* which contains only one FPGA image, and the corresponding optimizations are 1 CU and "Local" for *buildTable*, 4 CUs for *probeTable*, 1 CU for *map*, 1 CU for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for *blockAddition*, 1 CU for *gather*, and 4 S_CUs and 2 G_CUs for *sort*.

When the number of input tuples is more than 16M, the cost model recommends the *Execution Plan 2* which contains two FPGA images. The first image contains seven kernels and the corresponding optimizations are 1 CU and "Local" for *buildTable*, 1 CU for *probeTable*, 2 CUs for *map*, 2 CUs for *scanLargeArrays*, 1 CU for *prefixSum*, 1 CU and 16 SIMDs for

TABLE V: Summary of unit costs (UCs) for 11 kernels

Kernel	Number	Best Combination	LUTs	REGs	RAMs	DSPs	UC
sort	6	L_CU = 10 , G_CU = 2	99192	291232	2138	0	26.63
buildTable	5	CU = 1 , Local = 1	9032	21341	248	2	9.22
probeTable	4	CU = 8 , SIMD = 1	137973	224924	1728	72	7.74
groupBy_write	4	CU = 8 , SIMD = 1	27357	87332	472	32	5.04
GroupLabel	5	CU = 16 , SIMD = 1	96877	187964	1056	64	1.46
ScanLargeArrays	5	CU = 10 , SIMD = 1	111131	223228	1650	80	0.7
gather	4	CU = 8 , SIMD = 1	70757	134268	1040	0	0.55
map	4	CU = 8 , SIMD = 1	31725	55924	304	0	0.4
reduce	4	CU = 8 , SIMD = 1	103803	195320	1572	60	0.2
blockAddition	9	CU = 1 , SIMD = 16	8240	12108	80	0	0.11
prefixSum	1	CU = 1 , SIMD = 1	10598	20726	137	8	0.001

TABLE VI: Resource consumptions of FPGA images for Q3

Execution Plan 1					
FPGA image	LUTs	REGs	RAMs	DSPs	Freq.
E	151460	339134	2175	42	198
M	154509	283131	1973	34	233
Execution Plan 2					
FPGA image 1	LUTs	REGs	RAMs	DSPs	Freq.
E_1	187738	349051	2416	72	182
M_1	184082	334093	2342	72	192.5
FPGA image 2	LUTs	REGs	RAMs	DSPs	Freq.
E_2	179428	331045	2538	0	163
M_2	134629	346087	2421	0	182
FPGA image 3	LUTs	REGs	RAMs	DSPs	Freq.
E_3	155187	294559	1950	90	223
M_3	171434	348651	2112	90	203

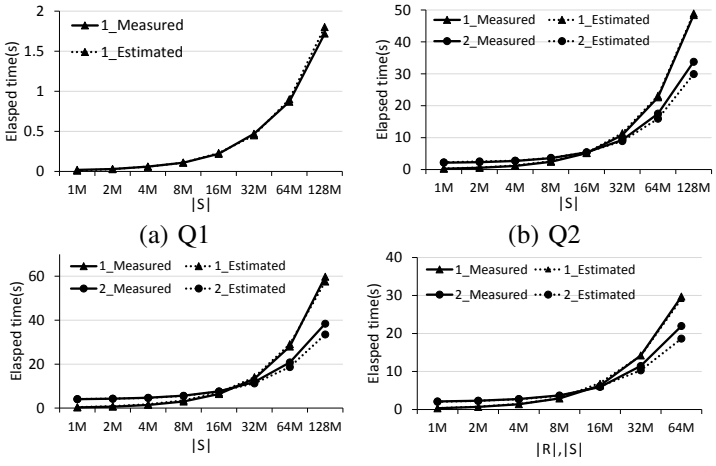


Fig. 7: Cost model evaluation for four queries.

blockAddition, 2 CUs for *gather*. The second image contains only one kernel *sort*, whose optimizations are 10 S_CUs and 2 G_CUs.

Summary: The optimization combination with minimum unit cost for each kernel is not always chosen for the query plan even when multiple FPGA images are considered. The reason is that optimization combination which has minimum unit cost always requires plenty of FPGA resource to implement, and then the corresponding FPGA image may not have high operating frequency.

3) Break-even Points for Four Queries: Figure 7 shows the measured and estimated elapsed times of the input queries (Q1, Q2, Q3 and Q4), with different numbers of input tuples (1M, 2M, ..., 128M), where *x_Measured* is the real execution time for the *Execution Plan x*, *x_Estimated* is the estimated time for the *Execution Plan x*, and *x* is equal to 1 or 2. However, 128M tuples would not fit inside the global memory for Q4, and thus Figure 7d does not include the results for 128M.

The experimental result shows that our cost model can roughly predict the performance for each query with different number of input tuples under different execution plans.

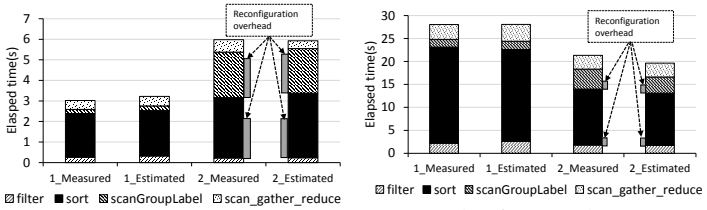
Our cost model can determine the break-even points between two execution plans for queries (Q2, Q3 and Q4). When the number of input tuples is less than 16M, *Execution Plan 1* with only one FPGA image is faster than *Execution Plan 2* with two (three) FPGA images, since the corresponding FPGA reconfiguration overhead dominates the whole execution time. That is, it does not have any advantage when multiple FPGA images are used. When the number of input tuples is larger than 16M, *Execution Plan 1* is slower than *Execution Plan 2*, since the kernel execution time dominates the whole execution time.

In summary, our cost model can accurately recommend the optimal execution plan for queries (Q2, Q3 and Q4) with different number of input tuples.

4) Performance Breakdowns: Because of page limitation, we only present the time breakdowns for the query Q3 with 8M and 64M input tuples, as shown in Figures 8a and 8b, respectively. *x_Measured* means the execution time for *Execution Plan x* and *x_Estimated* means the estimated time for *Execution Plan x* provided by our cost model, where x is equal to 1 or 2. *filter* stands for five OpenCL kernels (*map*, *scanLargeArrays*, *prefixSum*, *blockAddition* and *gather*), and *scan_gather_reduce* stands for the five kernels (*scanLargeArrays*, *prefixSum*, *blockAddition*, *gather* and *reduce*).

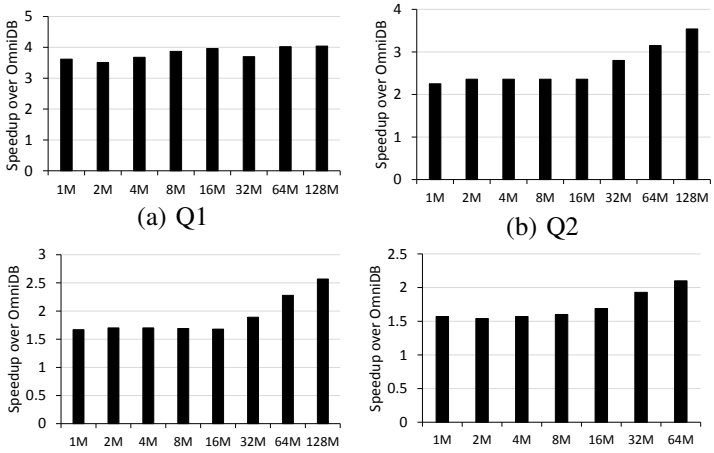
We show the time breakdowns for the query Q3 with 8M input tuples. *Execution Plan 1* has the better performance than that of *Execution Plan 2* since the FPGA reconfiguration overhead from *Execution Plan 2* is larger than the benefit from the reduced execution time when using multiple FPGA images. Since there are three FPGA images, two FPGA reconfigurations are required and hence, the corresponding reconfiguration overhead takes the majority of total execution time for *Execution Plan 2*. In particular, the time for first FPGA reconfiguration exists at the kernel *sort*, since Altera OpenCL SDK [1] counts the reconfiguration time into the execution time of the first kernel *sort* in the second FPGA image. Similarly, the time for the second FPGA reconfiguration is shown along with the execution time of the kernel *scanGroupLable* in the third FPGA image.

Our cost model can roughly predict the performance for all the kernels with FPGA reconfiguration and then choose the right execution plan (*Execution Plan 1*) for the query Q3 with 8M input tuples.



(a) 8M tuples (b) 64M tuples

Fig. 8: Time breakdown for Q3.



(a) Q1 (b) Q2

(c) Q3 (d) Q4

Fig. 9: Performance speedup over OmniDB (on FPGA) for four queries.

We show the time breakdowns for the Q3 with 64M input tuples. *Execution Plan 2* has the better performance than that of *Execution Plan 1* since the FPGA reconfiguration overhead (*Execution Plan 2*) is less than the benefit from the reduced execution time when using multiple FPGA images. With multiple FPGA images, each operator kernel of the query plan can have more aggregative optimizations. Our cost model can choose the right execution plan (with multiple FPGA images) for this case as well.

C. Performance Comparison

Figure 9 shows the performance speedup of each input query over the original *OmniDB* implementation (1 CU and 1 SIMD for each kernel), which is also recommended by Altera Opencl optimization tool [1]. When the number of input tuples is less than 16M, our cost model recommends the *Execution Plan 1* for each query (Q1, Q2, Q3 and Q4). The performance speedup ($1.5\times$–$4\times$) is roughly stable for each query over the original *OmniDB* implementation. Our cost model recommends *Execution Plan 1* for query Q1 even when there are more than 16M input tuples.

More interestingly, when the number of input tuples is larger than 16M, our cost model recommends *Execution Plan 2* which has multiple FPGA images, for queries Q2, Q3 and Q4. The speed up achieved by our implementation over the original omniDB implementation increases as the number of input tuples increases.

It is critical to achieve the good performance for query processing on OpenCL-based FPGAs and our cost model is able to recommend the optimal execution plan (with correct optimization combinations) for different number of input tuples for the four queries.

VII. Related Work

Accelerating database applications using FPGAs has received widespread attention due to their performance and power benefits in addressing the complexity of database processing operations. Although multicore CPUs and GPUs have been the major research platform for database query processing systems [6, 13], FPGAs are gaining popularity due to technology advances in programmable devices [24, 29].

The related studies of accelerating databases on FPGAs can be roughly divided into two categories: individual operators and queries.

Accelerations of individual database operators: There have been a number of studies on accelerating individual database operations with FPGAs, such as selection, projection, aggregation, sorting and hash table [5, 7, 11, 12, 18, 21, 25, 33], where FPGA is used as an accelerator. Besides, FPGA can also act as an additional hardware component in a database system to support its processing operations [19]. In particular, histograms are computed as a side effect when data are transferred from storage to the CPU. Most of those studies are based on low-level hardware description lanugages. Instead, this paper focuses on how the HLS framework (OpenCL in particular) and its features can be leveraged to improve the performance. Similar to our study, Arcas-Abella et al. [3] explores four HLS tools, e.g., Altera OpenCL SDK, to accelerate individual commonly-used database operators and analyzes the different trade-offs when using a representative set of HLS tools in the context of database acceleration. It focuses on optimizing the individual database operators while our approach focuses on accelerating the input query, which consists of multiple database operators, on one OpenCL-based FPGA.

Accelerating queries with FPGAs: Glacier [26] is a query-to-hardware compiler with library which can only support streaming operators, not all the database operators. Partial dynamic reconfiguration [10] is employed to support fast switch from one query to the next query using RTL. However, our approach can only support full reconfiguration since our approach is based on OpenCL which cannot support partial reconfiguration now. [10] can only support projections and restrictions while our approach can support all database operators. FQP [27] is presented to support selection, project and join over event streams at line rate. However, its join cannot produce all the satisfied join results since it is not for relational database. A hardware/software approach [30] was proposed to accelerate database queries with FPGA which acts as the accelerator, while our approach implements the entire query on FPGA. *Ibex* [36] acts as an intelligent storage engine to offload partial queries to FPGAs. In contrast, our work is based on the HLS framework, and focuses on the resource-constrained query optimizations.

With the OpenCL programming support on FPGAs, plenty of recent works [17, 23, 28, 31, 32, 34, 35] have gained great success in accelerating different kind of applications. In contrast, this study focuses on how to optimize existing OpenCL-based databases on FPGAs.

VIII. Conclusion

The recent OpenCL SDKs released by FPGA vendors have enabled programmers to design and implement database systems on FPGAs in OpenCL. In this abstraction, database query processing can be viewed as the execution of a series

of OpenCL kernels. Each kernel can have very different optimization combinations, which results in very different resource usages and performances. In this paper, we propose the FPGA-specific cost model to determine the optimal query plan for the input query. The experiments show that 1) our cost model can accurately predict the performance of each feasible query plan for the input query, and is able to guide the generation of the optimal query plan, 2) our optimized query plan achieves a performance speedup 1.5×–4× over the state-of-the-art query processing on OpenCL-based FPGAs.

IX. Acknowledgement

We thank Altera University Program which has denoted Terasic's DE5-Net FPGA board for our research. This work is supported by a MoE AcRF Tier 1 grant (MOE 2014-T1-001-145), an NUS startup grant R9336, and a Startup grant in HKUST.

References

[1] Altera. Altera SDK for opencl optimization guide. 2013.

[2] Altera. Stratix V device overview. 2014.

[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 *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[4] L. Art and H. Mauch. *Dynamic Programming: A Computational Tool*. Springer.

[5] A. Becher, D. Ziener, K. Meyer-Wegener, and J. Teich. A co-design approach for accelerated sql query processing via fpga-based data filtering. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 192–195, 2015.

[6] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR 2005*.

[7] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[8] D. Chen and D. Singh. Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 5–12, Aug 2012.

[9] 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 *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug 2012.

[10] C. Dennl, D. Ziener, and J. Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 45–52, April 2012.

[11] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. Fpga-based multithreading for in-memory hash joins. In *Conference on Innovative Data Systems Research*, 2015.

[12] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 17–20, April 2013.

[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[15] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[16] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4):329–340, Dec. 2014.

[17] M. Hosseinabady and J. L. Nunez-Yanez. Optimised opencl workgroup synthesis for hybrid arm-fpga devices. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–6, Sept 2015.

[18] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. A flexible hash table design for 10gbps key-value stores on fpgas. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[19] Z. Istvan, L. Woods, and G. Alonso. Histograms as a side effect of data movement for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1567–1578, New York, NY, USA, 2014. ACM.

[20] D. E. Knuth and J. L. Szwarcfiter. A structured program to generate all topological sorting arrangements. In *Articles of IPL 1974*.

[21] D. Koch and J. Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[22] Loring Wirbel. Xilinx SDAccel A Unified Development Environment for Tomorrow's Data Center. 2014.

[23] V. Mirian and P. Chow. Exploring pipe implementations using an opencl framework for fpgas. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 112–119, Dec 2015.

[24] R. Mueller and J. Teubner. Fpga: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM.

[25] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[26] R. Mueller, J. Teubner, and G. Alonso. Glacier: A query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1159–1162, New York, NY, USA, 2010. ACM.

[27] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on fpgas. *Proc. VLDB Endow.*, 6(12):1310–1313, Aug. 2013.

[28] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides. A case for work-stealing on fpgas with opencl atomics. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 48–53, New York, NY, USA, 2016. ACM.

[29] K. Shagrithaya, K. Kepa, and P. Athanas. Enabling development of opencl applications on fpga platforms. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 26–30, June 2013.

[30] B. Sukhwani, M. Thoennes, H. Min, P. Dube, B. Brezzo, S. Asaad, and D. Dillenberger. A hardware/software approach for database query acceleration with fpgas. *International Journal of Parallel Programming*, 43(6):1129–1159, 2015.

[31] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow. Opencl library of stream memory components targeting fpgas. In *Field Programmable Technology (FPT), 2015 International Conference on*, pages 104–111, 2015.

[32] Z. Wang, B. He, and W. Zhang. Improving data partitioning performance on opencl-based fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 34–34, May 2015.

[33] Z. Wang, B. He, and W. Zhang. A study of data partitioning on opencl-based fpgas. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8, Sept 2015.

[34] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing opencl applications on fpgas. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 114–125, March 2016.

[35] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2016.

[36] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[37] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proc. VLDB Endow.*, 6(12):1374–1377, Aug. 2013.