A Study of Pipeline Query Executions on OpenCL-based FPGAs

Johns Paul [1], Bingsheng He [2], Chiew Tong Lau [1]

[1] *Nanyang Technological University, Singapore*
[2] *National University of Singapore*

***Abstract*—Traditionally, FPGAs were usually programmed using low-level Hardware Description Languages (HDLs) like Verilog or VHDL, which made it extremely difficult to design, debug and build systems for FPGAs. However, the recent release of OpenCL SDKs by FPGA vendors like Xilinx and Altera have significantly improved the programmability of FPGAs and have brought new research opportunities for query processing systems on FPGAs. Although OpenCL makes FPGA programming easier, there remain two major questions which have been left unanswered. First, whether and how we can optimize OpenCL based database engines for FPGAs. There is a gap on optimizations and tuning between OpenCL and FPGA, since OpenCL is mainly designed for parallel multi-/many-core architectures. Second, the performance and energy consumption comparison to other accelerators like GPUs. In this paper, we attempt to answer these two questions under the context of pipelined query execution. We perform a detailed study of FPGA based database engines and compare their energy consumption and performance to other GPU based systems. Specifically, we design an FPGA based shared pipeline query execution system (FADE) which achieves up to 10x performance speedup over an existing pipelined single query executions on FPGA and 2x lower energy consumption than GPU based shared execution systems. Still, FADE is 2x slower than its GPU-based counterparts. Therefore, the comparison between FPGA and GPU is inconclusive for OpenCL-based database pipeline execution.**

I. Introduction

Over the last few decades, there have been significant efforts to accelerate query processing systems [10], [61], [31], [30], [57], [25], [53], [21]. One major field of interest has been the use of accelerators like GPUs and FPGAs to speedup query processing [57], [42], [**?**], [25], [55]. GPUs are more widely used [57], [42], [25]. One of the major reasons was due to the ease of programming the GPU (using languages like CUDA and OpenCL) when compared to FPGAs which are notorious for the difficulty involved in programming, due to the use of low-level HDLs like Verilog and VHDL. However, we have witnessed a renewed interest in FPGAs due to the emergence of OpenCL SDKs released by FPGA vendors [4], [1], [2] as a high-level synthesis (HLS) framework which has made it possible for developers to expose the data parallelism of the FPGAs using OpenCL. Still, the following questions regarding OpenCL-based query processing systems on on FPGAs remain unanswered.

First, whether and how we can optimize OpenCL based database engines for FPGAs. There is a significant gap on optimizations and tuning between OpenCL and FPGA because OpenCL is mainly designed for parallel multi-/many-core architectures and the architecture design of FPGAs is significantly different to that of CPU/GPU. This makes it extremely difficult to efficiently port existing query processing systems (designed for CPU/GPU) to FPGAs. For example, due to hardware differences like the limited availability of FPGA resources, it is not possible to accommodate a wide variety of pipelines in a single FPGA image. This coupled with the rigid nature of a FPGA based systems results in high reconfiguration overhead in systems that needs to handle a wide variety of queries. Further, existing OpenCL-based systems are often optimized to take advantage of the cache hierarchy of CPUs/GPUs, making them inefficient on FPGAs which often lack a cache. Second, the performance and energy consumption comparison to other accelerators like GPUs. Since both FPGA and GPU supports OpenCL, a natural question is which accelerator is faster or more energy-efficient.

In this paper, we attempt to answer these two questions under the context of pipelined query execution, which has been a classic query processing paradigm for data warehouses. In addition to executing a single query at a time, modern data warehouses provide significant opportunities for sharing data and computation. Hence, these data warehouses need shared execution systems that are capable of taking advantage of these sharing opportunities. Due to this very reason, shared systems like SharedDB [21], CJoin [13] MQJoin [35] have gained significant attention in the recent years.

Hence, we develop FADE, the first shared execution engine for FPGAs which solves the problems of high reconfiguration and communication overhead of existing database engines in the following ways. First, we design efficient shared pipelines that allows multiple queries to share the same pipeline simultaneously; thus reducing the impact of limited FPGA resources. Second, through clever hardware design we make the hardware pipelines in FADE capable of executing a wide variety of queries thus reducing the reconfiguration overhead ecnountered by modern query processing systems on FPGAs. Third, we adopt a fine grained pipeline approach that helps FADE reduce the number of costly global memory transactions through efficient use of local memory. We then use FADE to conduct an in-depth study of the performance and energy consumption of FPGA based shared execution systems in comparison with GPU based systems.

We have conducted experiments using the SSB [5] and TPC-W [6] benchmark on an Intel/Altera Stratix V FPGA, in comparison with query processing on an AMD R9 Fury

X GPU. Our pipelined implementation that supports shared query execution makes efficient use of FPGA resources and shows up to 10x performance improvement over the existing pipeline approach [42] of single query evaluations. The results also show that our FPGA based system can achieve up to 2x lower energy consumption than GPU based shared execution systems. Still, FADE is 2x slower than its GPU-based counterparts. Therefore, the comparison between FPGA and GPU is inconclusive for OpenCL-based database query pipeline execution, and database systems may choose accelerators according to the design requirements on performance and energy consumption.

To summarize, the contributions of this paper are as follows. First, we propose techniques like efficient shared pipeline design and hardware interconnections to address the problems of high reconfiguration and communication overhead in FPGAs. Second, we develop FADE, an FPGA based shared execution system that takes maximum advantage of the opportunities for sharing data and computation among different queries. Finally, using FADE, we perform detailed study of the performance benefits and energy consumption of pipelined as well as shared query execution systems on FPGAs, in comparison with GPUs.

The rest of the this paper is organized as follows. In Section II, we introduce the background on FPGA SDK for OpenCL and shared query execution. We illustrate our motivations for proposing a pipelined shared query execution engine in Section III. We elaborate the design and implementation details of FADE in Section IV. In Section V we present our experimental results and we review the related work in Section VI. Finally, we discuss the lessons learnt in Section VII and conclude in Section VIII.

II. Background

A. FPGA Hardware & OpenCL SDK

Modern FPGA hardware is manufactured by two major vendors: Intel/Altera and Xilinx. The FPGAs from both these vendors usually consists of four different kinds of hardware units: logic units (or LUTs), memory blocks, registers and DSPs. In addition to the on-chip block RAM, FPGAs usually contain off-chip RAM (HBM or DDR) which usually has higher capacity, higher latency and lower throughput than the on-chip memory. Unlike GPUs, most FPGAs lack an on-chip cache which automatically stores frequently used data items. Traditionally, programmers used HDLs to specify a low level design of the system in which different hardware units are interconnected to realize the system functionality. The OpenCL SDKs on the other hand abstracts away such complexities and allows users to compile OpenCL code into a hardware design (bitstream). Both Intel/Altera and Xilinx offer OpenCL SDKs, referred to as Intel/Altera FPGA SDK for OpenCL [1] and SDAccel [4] respectively. Throughout the rest of this paper, we refer to both of them collectively as simply *OpenCL SDKs*.

An OpenCL-based system is designed as a collection of kernels which are mapped to what is referred to as a compute units (CU) in the actual FPGA hardware. The OpenCL memory hierarchy consists of a local memory and a global memory, which are mapped to the on-chip block RAM and the off-chip DDR/HBM RAM respectively. Due to its high latency and low bandwidth, global memory accesses can be a major bottleneck for system performance when designing OpenCL-based query processing engines for FPGAs. To minimize the use of global memory and associated overhead, the OpenCL SDKs support the use of OpenCL pipes (hereafter referred to as simply *pipes*), which are mapped to hardware FIFOs on FPGAs. The pipes allow operators to communicate with each other using FPGA on-chip memory, thus minimizing the number of global memory transactions. However, due to their hardware design these pipes are fixed between a pair of opertors and cannot be used for dynamically routing the data between different operators.

B. Pipelined Query Execution

Traditional database engines usually adopt one pipeline for one query [33], [17]. This single pipeline execution has recently been implemented on the GPU. Johns et. al. [42] proposed a pipeline execution approach named GPL, where the data is passed between kernels in small tiles using software buffers. This helps improve the overall system performance due to increased GPU cache utilization as well due to the improved resource utilization achieved by supporting concurrent kernel executions in GPL.

Modern data warehouses often need to support a large number of concurrent queries that show significant overlap of input data and operators. In such a scenario, a shared execution model avoids repeated invocation of operators and unnecessary data reads and can achieve much higher throughput. Previous studies on shared execution [35], [13], [21], [47], [22], [8], [19] use either *simultaneous pipelining* (SP) or *global query plan* (GQP) to implement shared query execution. In SP, sharing is achieved by dynamically routing the intermediate data between the operators. The GQP based shared execution system merges the individual query plans into a single GQP after identifying opportunities for shared execution. SP is not suitable when using hardware components like pipes to ensure low overhead communication between operators. A more suitable approach for FPGAs is GQP. However, due to the rigid nature of hardware components like the pipes, we need to carefully revisit GQP to maximize the sharing of data and hardware resources among queries.

III. Motivations

In this section, we first analyze the performance pitfalls of existing pipeline execution approaches on CPUs/GPUs when ported to run on FPGAs. Next, we elaborate the potential advantages of supporting shared executions on the FPGA, rather than one query at a time.

A. Inefficiency of Existing Pipeline Executions

Pipelined query processing systems were proposed to minimize the high communication overhead encountered by traditional operator at a time implementations. Paul et. al. [42]

proposed the current stat-of-the-art pipelined OpenCL based pipelined query processing engine for GPUs. However, existing pipeline execution strategies (including GPL) are inefficient on FPGAs, due to the following reasons.

First, GPL still needs to store each block of intermediate data in the global memory. The limited level of concurrency available in the FPGA (due to resource limitations) and the overhead associated with repeated kernel invocations means that the use of very small tile size is not feasible on FPGAs, leading to high global memory usage.

Second, GPL encounters a large number of global memory transactions on FPGAs. GPL reduces the overhead associated with memory transactions by taking advantage of small tile sizes which could fit within the GPU cache. However, FPGAs do not have an on-chip cache and the data in the block RAM cannot be shared across kernels without using primitives OpenCL pipes. Hence, reading and writing of each data tile needs to go through the global memory resulting in high communication overhead.

B. Benefits of Shared Execution

Previous studies [21], [13], [35] have already demonstrated the huge opportunities for data and operator sharing in modern data warehouses. In addition to this, shared execution has the following potential advantages when implemented on accelerators, especially for FPGAs.

The first is the reduced PCIe overhead when implementing shared execution systems on accelerators. Our experimental results in Section V-B show that accelerator based query processing engines spend significant amount of time transferring data over the PCIe bus. In fact, the PCIe overhead is even more than the query execution time in most cases and hence it is even impossible to hide the PCIe overhead by overlapping computation with data transfer. To reduce the impact of the PCIe overhead on the overall performance, shared pipeline execution is an effective approach to achieving maximum reuse of data and operators.

The second is on resolving resource under-utilization in single query systems. A single pipeline may not fully utilize the hardware resources of an FPGA. Our experimental results in Section V-B show that a single query is not capable of making efficient use of FPGA resources and will result in severe resource under-utilization in FPGAs. This is because, the operations required by a single query is limited and is hence incapable of taking advantage of all the hardware resources within the system. This problem can be addressed by making sure that multiple queries are active within the same FPGA context simultaneously. However, a naive approach of running a separate pipeline for each query will fail to take advantage of the opportunities for sharing data and computation. Allowing multiple queries to share the same pipeline on the other hand addresses these limitations and makes it possible to take advantage of data sharing opportunities as well as improve the resource utilization on FPGAs.

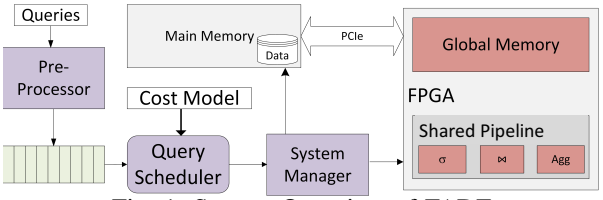

Fig. 1: System Overview of FADE.

TABLE I: Notations used in this paper

Notation	Definitions
DB_{ij}	bit string associated with tuple j of table i
$DB_{ij}[k]$	the k^{th} bit of DB_{ij}
WO	weighted sum of the overlap of each operator
$Ovlp[l]$	Overlap achieved by the l^{th} operator/kernel
C_l	Relative weight of the l^{th} operator/kernel

IV. System Design

In this section we present the design of FADE, the shared pipeline query execution system designed for OpenCL-based FPGAs. Figure 1 shows the high level view of the main components of FADE that supports shared execution. Our system consists of four major components: pre-processor, query scheduler, system manager and the shared pipeline which executes the query. In the remainder of this section we explore the detailed design of each component. A summary of the notations used in this section is given in Table I.

A. Pre-Processor

The queries submitted to FADE first go through a pre-processor.It generates the query plan and estimates the resource requirements and the selectivity of each operator. The query plan generation follows the pipeline generation method of the previous study (SharedDB) [21]. The system uses a frequency based histogram to estimate the selectivity of each query [26], [46], [45]. After the pre-processing, the queries are added to the *query pool*, waiting for execution.

B. Shared Pipeline Design

In order to support shared pipeline execution on FPGA, we redesign the query operator for distinguishing individual queries in a lightweight manner. Further, we develop FPGA-specific optimizations, one to minimize the communication overhead between operators and the other to minimize the reconfiguration overhead.

1) Operator Design: The design of our operators are actually rooted at many of the previous pipelined execution engines [42], [17], [33]. For each operator, we revisit the existing designs and modify these implementations to support shared execution on FPGAs. To support shared execution, FADE maintains an additional attribute (*bit string*) for each table in the database. This bit string allows FADE to distinguish individual queries in a lightweight manner. This design of bit string is adopted from the previous study [13]. Each bit of the bit string allows FADE to keep track of whether a specific query is interested in the tuple. Particularly, a bit in this bit string represents the involvement of a single query in the shared pipeline. It is set to '1' if the tuple will be processed by the corresponding query, '0' otherwise. Next, we present the

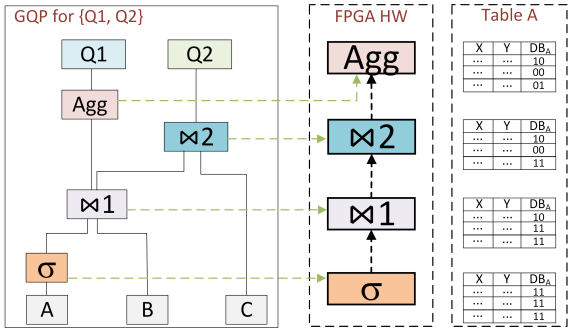

Fig. 2: A valid mapping between GQP and hardware pipeline

design and implementation of major operators, with a highlight on the modification made from pipeline implementation in order to support shared execution.

Selection. In a shared environment, the selection on tuple j of table i sets the k^{th} bit of the associated bit string, $DB_{ij}[k]$, to '1' if the tuple is selected by query k and '0' otherwise. The next kernel in the pipeline simply ignores all the tuples with '0' values. To allow shared execution, FADE tries to merge together the selection predicates of individual queries into a single predicate. However, if this cannot be done for the given set of queries, then FADE performs the selection operation in multiple steps.

Group By and Aggregation. Both group-by and aggregate operations are performed by a single kernel in our implementation. The kernel reads each tuple and its associated bit string and then determines the group id for the tuple using the group-by attributes. Finally, the aggregation operation is performed for each query that is interested in the tuple. To improve performance, the aggregation is done using local memory and the final results are written into the global memory.

Hash Join. FADE adopts the same hash join implementation used in the previous study [42], where the operation is completed in two stages: build and probe. In FADE, the bit string associated with each tuple is stored in the hash table along with the data. The probe operation reads a tuple and its associated bit string from the memory and then probes the hash table. On finding a match, the probe operation performs a *bitwise AND* operation of the bit string in the hash table and DB_{ij}. The result of this operation is then passed to the next operator.

Sort. FADE makes use of a hash based sort implementation. The implementation is the same as the build stage of the hash join.

Figure 2 demonstrates how a GQP is executed by the shared pipeline and how the bit strings associated with a table are modified by operators as the data advances through the shared pipeline. Assuming there are two queries for shared execution, the bit string has two bits.

2) Minimizing Communication Overhead: A shared pipeline essentially consists of a sequence of OpenCL kernels. The tuples are passed through one kernel to another during the execution. To minimize the communication overhead between kernels, FADE moves the data between kernels using pipes wherever possible, thus reducing the communication overhead and the immediate result materialization to the global memory.

The OpenCL SDKs from both Altera and Xilinx allow system designers to tune bot the width and the depth of these hardware pipes. The width is set to be the bit string size, which enables fine-grained pipeline execution; while the depth needs some careful tuning. A larger depth allows the producer (consumer) kernel to queue (consume) more entries before it gets blocked. In our testing, we found that most pairs of kernels connected by these hardware pipes show very low variation in data processing rate. Further, increasing the depth results in severe increase in FPGA resource utilization, and thus reducing the resources available for performing computations. In our experiments, we find that, it achieves a good performance when we keep the depth to a minimum (less than 512 bytes).

3) Minimize Reconfiguration Overhead: Because the pipes are implemented with one read and one write port, it is impossible to dynamically route the data between arbitrary kernels once the bitstream is generated. Further, any change in the connected kernels require a full reconfiguration of the FPGA (current OpenCL SDKs do not support partial reconfiguration), adding significant overhead to query processing.

To avoid unnecessary reconfigurations and to make the system design flexible, FADE supports two operating modes for each kernel: *pass-through* mode and *processing* mode. The processing mode is the default operating mode of a kernel/-operator where the kernel fulfills its designed functionality. A kernel in the pass-through mode just reads data from the previous kernel and passes it to the next kernel using the local memory. That essentially allows passing the tuples from one kernel to another in a lightweight manner, although they are not directly connected by pipes. Such an approach allows FADE to efficiently handle a wide variety of queries without requiring an FPGA reconfiguration.

To show the benefit of pass through kernels we use the following as a motivating example. Figure 3 shows an example mapping of a query plan to the operators in the hardware pipeline. In this example, the query only needs to perform one filter, one join and one aggregate operation. However, the pipeline present in the FPGA has one filter operator, two join operators and one aggregate operator connected using pipes. There are two nave ways to execute the query on the FPGA: 1) by reconfiguring the FPGA with a new bitstream that can execute the given GQP. This approach results in a costly FPGA reconfiguration 2) by passing the data from the first join operator to the aggregate operator using the global memory. This results in an increase in the number of global memory transactions. With two-mode support in a kernel, FADE invoke the second join operator in the pipeline in the pass-through mode as shown in Figure 3. This avoids FPGA reconfiguration and costly global memory transactions, compared with the two nave approaches.

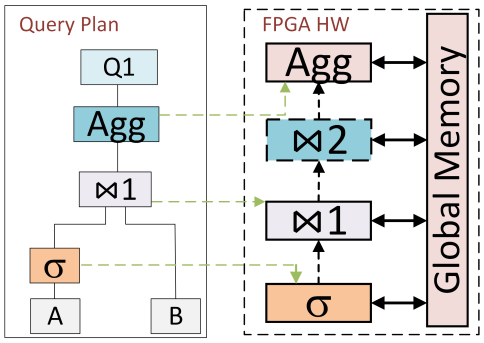

Fig. 3: Use of pass-through mode for operators

C. Query Scheduler

Before we detail the working of the query scheduler, we define the following key terms, which are essential to understand the query scheduler design.

Candidate group: Any subset of the queries in the query pool is defined to be a candidate group for scheduling consideration.

Valid group: Any candidate group for which it is possible to map every operator in the GQP of the group to an operator in the shared hardware pipeline, while keeping the data flow dependency. An example of such mapping is shown in in Figure 2. Due to the support of pass-through mode in the kernel, the mapping from GQP to the shared hardware pipeline is in fact more flexible in supporting queries with only some of the operators in the pipeline.

Overlap: The overlap is defined as the percentage reduction in the number of tuples accessed by the GQP with respect to the sum of the number of tuples accessed by individual queries that constitute the GQP. We can also compute the overlap at the operator level by considering only the number of tuples accessed by that particular operator.

The main task of the query scheduler is to examine the possible candidate groups and find a valid group that achieves good levels of operator and data sharing. Due to the large solution space of the scheduling problem, we use a heuristic based greedy algorithm. Prior to presenting our heuristic based approach, we present a few experimental observations regarding shared pipeline execution on FPGA, to demonstrate the key factors affecting the performance in shared execution.

In Figure 4, we present the performance of 20 different groups of SSB queries (shown in Table II), along with the number of unique tuples accessed by the GQP of each group. More experimental setup can be found in Section V. The results show that there is a strong correlation between the two factors, since memory accesses are still a major bottleneck for the performance of pipeline execution on FPGA.

Next, we analyze the impact of data sharing on different operators. Figure 5 shows speedup achieved by each individual operator when the overlap is increased from 0% to 50%, for two queries running concurrently. The speedup is measured with respect to execution time when the overlap is 0%. The result shows that different operators achieve different levels of speedup in shared execution.

TABLE II: Query groups for shared query execution

Group ID	SSB Queries
1	2.1 2.2
2	2.2 2.3
3	2.2 3.1
4	3.1 3.2
5	2.1 2.2 2.3
6	2.2 2.3 3.1
7	2.1 3.2 3.4
8	3.1 3.2 4.1
9	2.3 3.4 4.2
10	2.1 2.2 2.3 3.1
11	2.2 2.3 3.1 3.2
12	3.1 3.2 3.3 3.4
13	2.1 3.1 3.4 4.1 4.3
14	2.1 2.2 2.3 3.1 3.2
15	2.1 2.2 3.1 3.3 4.1
16	2.2 2.3 3.2 3.3 3.4 4.2
17	2.1 2.2 2.3 3.1 3.2 3.3 3.4
18	2.1 2.2 2.3 3.1 3.2 3.3 3.4 4.1
19	2.1 2.2 2.3 3.1 3.2 3.3 3.4 4.1 4.2
20	2.1 2.2 2.3 3.1 3.2 3.3 3.4 4.1 4.2 4.3

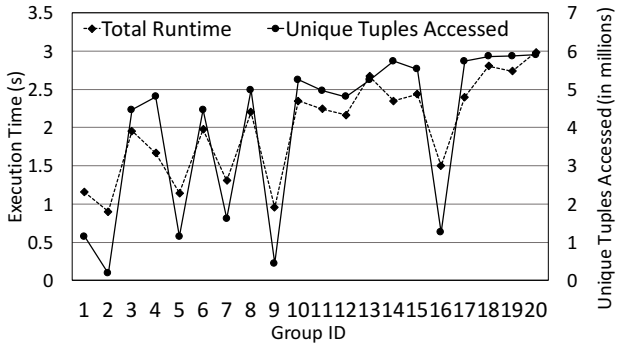

Fig. 4: Tuples accessed vs total runtime for shared query execution.

Considering the above key factors, we need to identify the valid group that achieves the highest levels of data and operator sharing. Due to the large number of valid groups that needs to be considered, it is almost impossible to find the most suitable group by evaluating the overlap of all possible groups. The problem of identifying the best group of queries that achieves maximum sharing of data is equivalent to the minimum weight perfect matching problem in contention aware scheduling [28], [59], [60]. The problem has been proved to be NP-Hard [28], when more than two tasks/queries need to be co-scheduled. Further, existing approximation algorithms (e.g., [28]) adds very high runtime overhead when the number of queries

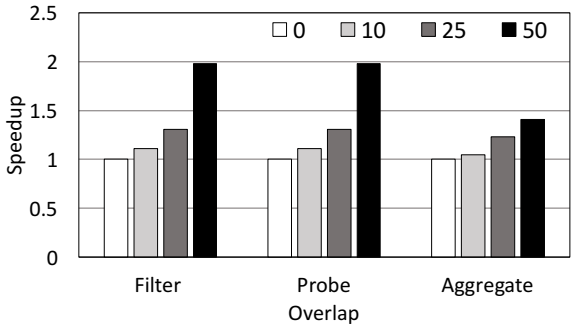

Fig. 5: Speedup achieved by different operators with varying overlap.

involved is quite high. Hence, we propose a simple and effective greedy approach to find a 'good enough' group of queries without incurring significant runtime overhead (less than 15 ms in our study).

To account for the above factors, our query scheduler tries to maximize the weighted sum of the overlap of each operator in the GQP (WO) instead of simply maximizing the overlap of GQP. We use Equation 1 to compute the WO value for each GQP. Here, $Ovlp[l]$ is the overlap of the l^{th} operator in the pipeline, C_l is a constant determined experimentally through profiling and m is the total number of operators in the pipeline. We profile the FPGA bitstream by executing a set of test queries with varying memory access pattern.

$$WO = \sum_{l=1}^{m} C_l * Ovlp[l] \quad (1)$$

When the FPGA is ready for execution, we use a greedy algorithm to add queries from the query pool to the *Execution List* (EL) one at a time. The selection is in the sorted order of the query selectivity to maximize the probability of achieving good overlap. If the addition of a query, Q, to the EL does not result in a valid group, then the system removes some queries from the EL, such that 1) the new set of queries form a valid group and 2) the decrease in WO due to the removal is minimal. Next, we compare the WO value of the group of queries in the EL before Q was added and the WO value of the group of queries currently in the EL. The group with the highest WO value is then chosen as the EL for the next iteration. Once we check all the queries in the query pool, the queries currently in EL form a valid group, and will be executed on the FPGA.

To avoid starvation [49], we associate a *starvation counter* (initialized to 0) with each query submitted to the system. Every time a group is chosen for execution, the starvation counter of all the remaining queries in the query pool is incremented by one. Later, queries are chosen from the query pool in the decreasing order of their starvation counter value. Also, we do not allow the removal of a query with higher starvation counter value over a query with lower starvation counter value.

In our heuristic approach, the order of considering queries has a significant impact on achieving better overlap. This is a classic problem associated with a greedy approach. Hence, to improve the accuracy of our heuristic, whenever a new query is added, we do not discard the old EL if its WO value is above a certain threshold, and view them as *active* for further consideration. Then, for each iteration we try to add the query to all the active ELs. In our implementation, we limit the number of active EL to 8 due to increase in scheduling overhead with increase in the number of active ELs. We experimentally evaluate the impact of this tuning parameter in Section V.

D. System Manager

The system manager prepares the system for execution in two ways. First, it transfers the necessary input data from the CPU main memory to the FPGA global memory. The system manager adopts an existing memory management approach [12] to keep track of the data that is present in the FPGA. When a set of queries are scheduled for execution, only the necessary data needs to be transferred over the PCIe bus. The system manager also has the additional responsibility of removing cold data when the FPGA runs out of memory. Second, it initializes all the necessary bit strings for selected queries to support shared execution.

Reconfiguration of the FPGA is also handled by the system manager. In this study we assume that all the necessary bitstreams are pre-compiled and made available to the system manager when it begins execution. This is because due to the limitations of the current OpenCL SDKs, the generation of even simple bitstreams take hours and hence its impossible to perform this stage at runtime. Each of these bitstreams will be capable of executing a different set of queries. FPGA reconfiguration is required whenever the current bitstream is incapable of executing a given set of queries. The reconfiguration time is around 2 seconds on average. This is a limitation of current FPGAs to support more dynamic workloads. The partial reconfiguration capability of FPGA can potentially reduce this overhead. However, current OpenCL SDK does not support this capability.

V. Experimental Evaluation

A. Experimental Setup

Hardware. We study our proposed design using a Terasic DE5Net board with an Intel/Altera Stratix V FPGA. The board contains 4GB DDR3 global memory. The bitstreams were generated using Intel/Altera FPGA SDK for OpenCL version 16.0. The FPGA is connected to the CPU via an x8 PCIe 2.0 interface, with peak bandwidth of 4GB/sec (bi-directional). For comparison against CPU, we use an Intel Xeon E5-1650 CPU running at 3.2GHz with 12 logical cores and 64GB of main memory. For comparison against GPU, we use AMD R9 Fury X with 4 GB global memory. We use AMD CodeXL and Quartus to measure the power consumption of the GPU and FPGA, respectively.

Workload. We evaluate FADE using the following public benchmarks: SSB [5] and TPC-W [6]. The SSB data set used for experiments has a scale factor of 20 and is used for evaluating both single query and shared query execution. Queries 1.1, 1.2 and 1.3 of the SSB benchmark contains filter operations on the fact table and they do not perform any group-by or order-by operations. All the other queries have both group-by and order-by operations and only contain filter operations on dimension tables. TPC-W data set used for experiments has a data size representing 15K emulated browsers and is used for comparing shared execution only. Following the previous study [21], the updates are performed by the CPU and the data in FPGA global memory is updated from the CPU side to reflect the changes. Detailed description of the benchmarks can be found in the benchmark specifications [6], [5].

Experimental Outline. Our evaluation is organized as follows. In Section V-B, we evaluate the consumed by our

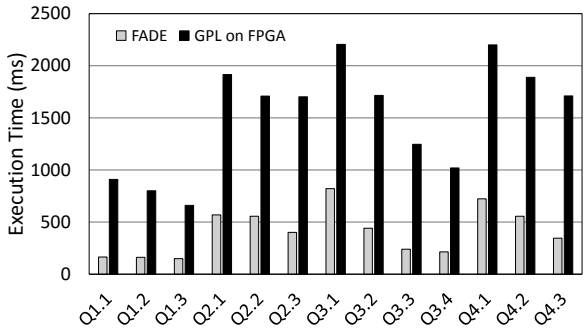

Fig. 6: Performance comparison of single query implementation of FADE and GPL on FPGA.

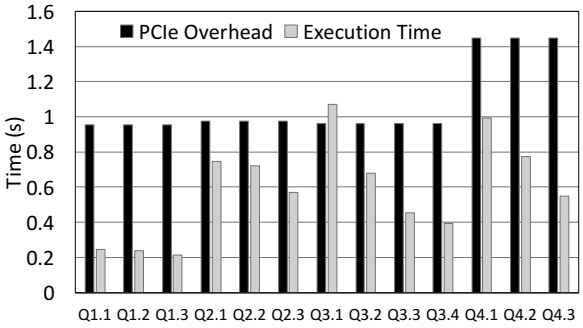

Fig. 7: PCIe overhead and execution time for SSB queries.

fine-grained pipeline by comparing FADE against a version of GPL running on the FPGA. Since both the systems are running on FPGAs, the overhead of the PCIe data transfer is not considered in this set of experiments. In Section V-C, we evaluate the performance of our shared pipelined as well as the greedy heuristic used by the query scheduler. Finally, in Section V-D, we compare the end-to-end performance of FADE against a shared query version of GPL [42] running on GPU.

B. Fine Grained Pipeline Evaluation

Overall Comparison. In Figure 6, we use the SSB queries to evaluate the single query implementation of FADE against a version GPL running on the FPGA. Similar to the GPU based version, the version of GPL running on FPGA moves data between kernels in small chunks. However, due to the lack of a cache on FPGA the data writes and reads associated with each chunk of data actually go to the FPGA global memory. In comparison, FADE is capable of transferring intermediate using hardware interconnections which makes use of the FPGA local memory. This allows FADE to achieve significantly lower communication overhead than GPL. Hence FADE achieves close to 3.6x speedup over GPL, as shown in the figure.

PCIe Overhead. Figure 7 shows amount of time spend on PCIe data transfer and the total execution time for the SSB queries. The results show that, in many cases the PCIe overhead is more than twice of the computation time, making it impossible to even hide the data transfer overhead by

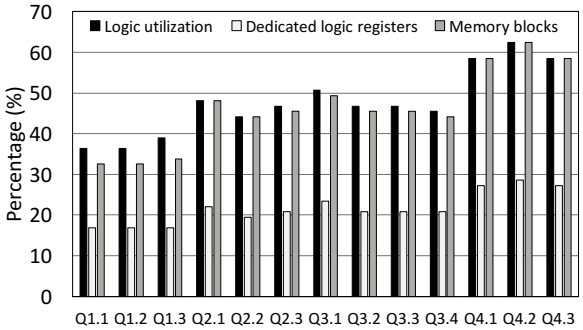

Fig. 8: Resource utilization in single query execution.

overlapping data transfer with computation. This shows the necessity of adopting shared execution, to reduce the PCIe overhead by maximizing sharing of data and operator.

Resource utilization. Figure 8 shows the percentage of LUTs, registers and memory blocks consumed by each individual query of the SSB benchmark, when implemented on the FPGA for optimal performance. It clearly shows that single query execution results in severe resource under-utilization in FPGAs. We observe similar results for TPC-W benchmark as well. However, only SSB results are presented here due to space constraints.

Summary. The experiments presented in this section clearly shows that, traditional optimizations used in OpenCL based systems (e.g., [42]), which are aimed at improving the performance of multi/many-core systems, are not suitable for FPGAs. This is due to the significant differences in FPGA hardware when compared to GPUs or CPUs. Hence, when porting existing OpenCL implementations to FPGAs, we should take advantage of FPGA specific optimizations as well as take the hardware resource utilization into consideration.

C. Shared Query Execution

Overhead of Unified Design Vs Reconfiguration Overhead. Figure 9 shows the percentage increase in execution time when we use a shared pipeline design that can execute multiple SSB queries (2.1 to 4.3), compared with using a custom designed pipeline that can execute only a single query. The results show that, in most cases, the overhead associated with the use of pass-through kernels in a shared pipeline is small (below 12%).

To further demonstrate the impact of reconfiguration overhead to a shared pipeline design, we compare the performance of 20 groups of queries shown in Table II on two bitstream configurations ($S1$ & $S2$). $S1$ consists of a single bitstream that can be shared by any subset of SSB queries from 2.1 to 4.3. $S2$ contains 3 separate bitstreams, each one capable of executing one of the following groups of SSB queries: queries 2.1 to 2.3, queries 3.1 to 3.4, and queries 4.1 to 4.3. Since each bitstream of $S2$ is designed for a small set of queries, each one of them is capable of achieving better performance than the bitstream in $S1$. However, $S2$ has to perform an FPGA reconfiguration to execute a new bitstream. Figure 10 shows the speedup achieved by $S1$ when compared to $S2$. The

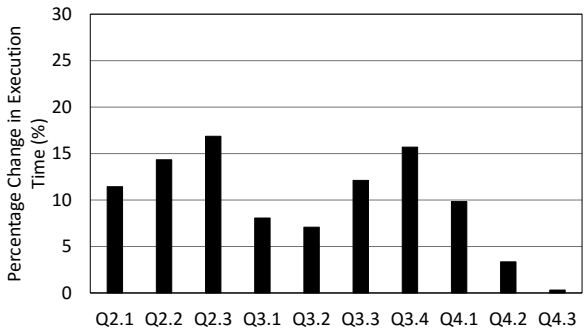

Fig. 9: Overhead associated with the use of unified design.

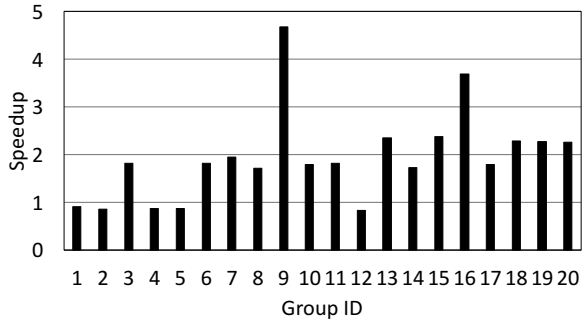

Fig. 10: Speedup achieved by unified design.

speedup is defined as the ratio of the execution times between $S2$ and $S1$. If the speedup is larger than 1, $S1$ is faster than $S2$. The results show that when reconfiguration is not required (Group ID 1, 2, 4, 5 and 12), $S2$ fares slightly better than $S1$. However, for all other groups, $S1$ outperforms $S2$ by up to 4.6x due to the high reconfiguration overhead in $S2$.

The two experiments presented above demonstrate an important dilemma faced by FPGA optimizations. On the one hand, we could design pipelines capable of executing only a small set of queries and pay the high reconfiguration overhead whenever a reconfiguration is required. On the other hand, they could design more flexible pipelines which are capable of executing a wide variety of queries and pay a small overhead for the execution of each individual query. The optimal choice varies from different FPGAs and depends on the workload (e.g., the mix of different queries). It is our future work to explore different choices in more details.

Improved Resource Utilization. Table III shows the percentage of LUTs, registers and memory blocks consumed by the bistreams containing the shared pipelines used for executing the SSB and TPC-W queries. We measure the resource utilization when these pipelines are implemented on the FPGA for optimal performance. When compared to the resource utilization in single query execution (Figure 8), shared execution achieves significantly higher resource utilization as shown in Table III.

Heuristic Evaluation. We now evaluate the effectiveness of the heuristic proposed in Section IV. Figure 11 shows the average execution time of 5 workloads (G1–G5, each containing 50 random queries from SSB) for two cases: 1) when the queries are grouped together using our heuristic based approach and 2) when we choose the optimal grouping by evaluating all possible groupings offline. The execution time of our proposed approach is very close to the offline (optimal approach). We conduct these experiments using both SSB and TPC-W queries and found similar results. The runtime overhead incurred by the heuristic is usually less than 1% of the total execution time. The results show that the heuristic is capable of finding good enough groups of queries, without adding significant runtime overhead to query processing.

TABLE III: Shared pipeline resource utilization

Pipeline	LUT	Register	Memory Blocks
TPCW	92	46	92
SSB	91	40	92

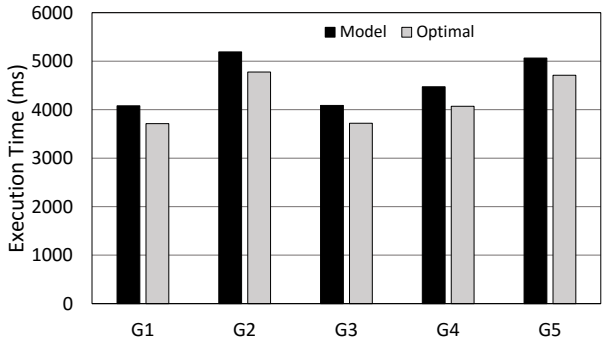

Fig. 11: Comparison of the proposed heuristic against optimal grouping.

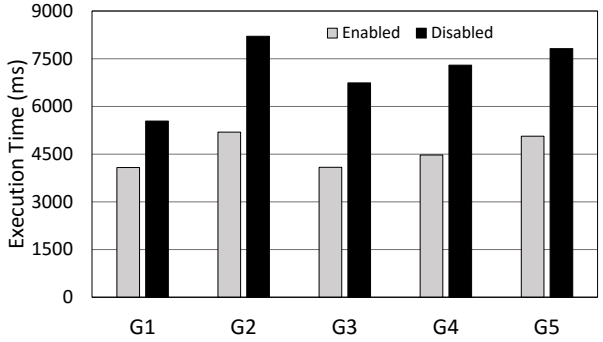

Fig. 12: Impact of using weighted sum for query scheduling

Figure 12 shows the average execution time of 5 group of queries when we enable the use of weighted sum and when we disable it. The result shows that the use of weighted sum has a significant impact on improving the overall performance of the system. This is because, different operators achieve different levels of speedup during shared execution.

In Figure 13, we measure the average execution time of 5 group of queries, when the number of active execution lists is increased from 1 to 8. The results show that the use of multiple execution lists allows the heuristic to achieve better grouping.

Overall Comparison. To evaluate the overall performance speedup of adopting shared query execution, we conduct experiments on 20 groups of queries shown in Table II. The results presented in Figure 14 show that, shared pipeline execution can achieve up to 2.4x speedup over pipelined single query implementation and up to 10x speedup over running the

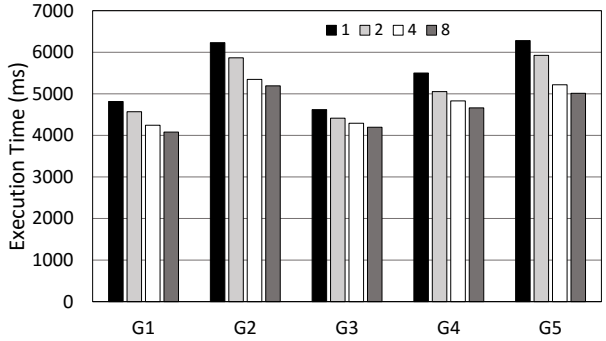

Fig. 13: Impact of using multiple execution list

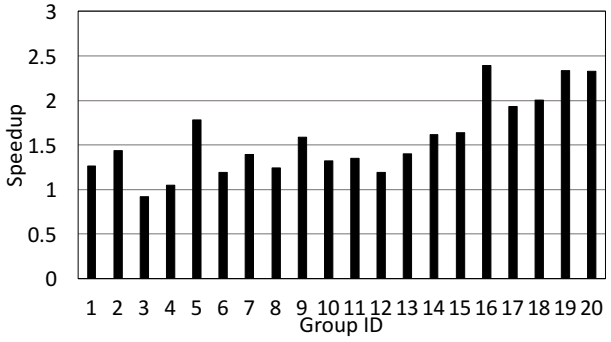

Fig. 14: Speedup of shared query execution when compared to single query execution.

pipeline design of GPL on FPGA. Note that, one benefit of shared pipeline execution is not reflected in this comparison. That is, shared pipeline execution allows more queries to be evaluated before an FPGA reconfiguration. In this experiment, we only need one FPGA reconfiguration at the beginning of the execution. In contrast, single-query executions generates one bitstream for each query, which requires one FPGA reconfiguration per query in the worst case. This overhead is not reflected in the comparison.

Summary. The experiments presented in this section demonstrates the importance of utilizing the sharing opportunities available in modern data warehouses. The pass-through mode of kernel design enables more flexible query sharing as well as reduces the FPGA reconfiguration overhead. The heuristic approach of query scheduling leads to little runtime overhead but still generates very good grouping. All those designs contributes to the improvement of shared pipeline execution on the FPGA.

D. Comparison with GPU

Performance Comparison. To compare FPGAs and GPUs, we implement the shared pipeline version of GPL, *GPL (Shared) on GPU*, for the AMD GPU. We implemented GPL (Shared) on GPU by modifying the existing single query version of GPL to support shared execution.

In Figure 15, we compare the performance (without PCIe) of FADE and GPL (shared) on GPU, in the context of shared execution. The workload is the 20 groups of SSB queries (shown in Table II). The results show that GPU outperforms the FPGA by up to 2x.

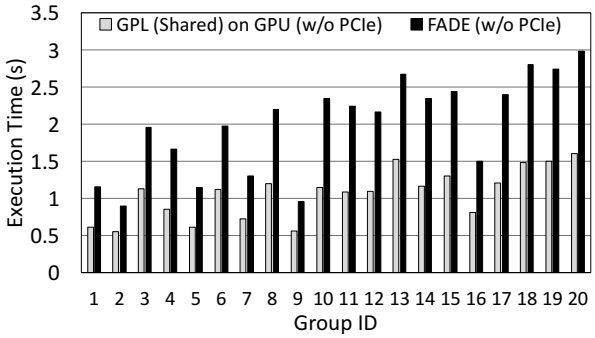

Fig. 15: Execution time comparison between FPGA and GPU (without PCI-e).

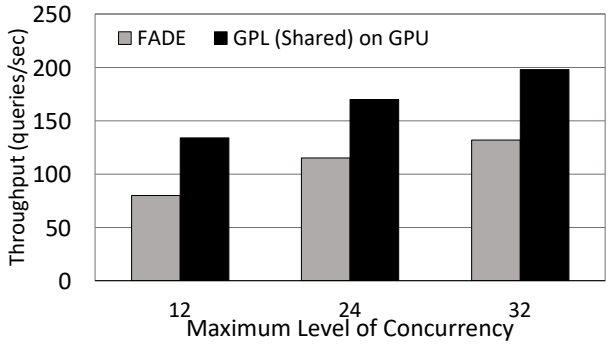

Fig. 16: Performance comparison for TPC-W benchmark.

We also evaluate FADE using the TPC-W benchmark, by comparing the throughput achieved by GPL (Shared) on GPU and FADE. We present the results by varying the maximum number of queries that can grouped together (for shared execution) from 12 to 32. The throughput of both FADE and GPL (Shared) on GPU increases with increasing level of concurrency. Overall, GPL (Shared) on GPU significantly outperforms FADE just like in the case SSB benhcmark.

Energy Consumption. We also evaluate the energy consumption FPGAs and GPUs for both SSB and TPC-W benchmarks. In Figure 17, we present the energy consumed by the FPGA and GPU when executing the 20 groups of SSB queries. We also compare the average energy consumed per TPC-W query by FADE and GPL (Shared) on GPU in Figure 18. In

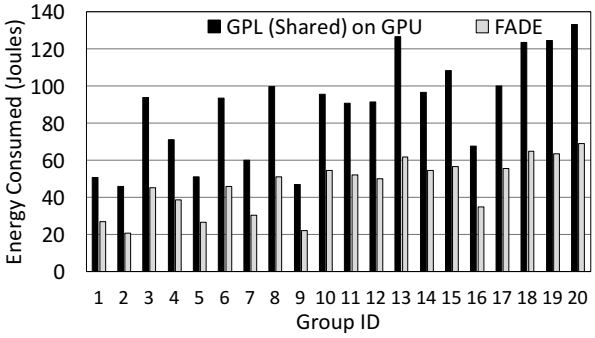

Fig. 17: Energy consumption of FPGA and GPU while executing SSB queries.

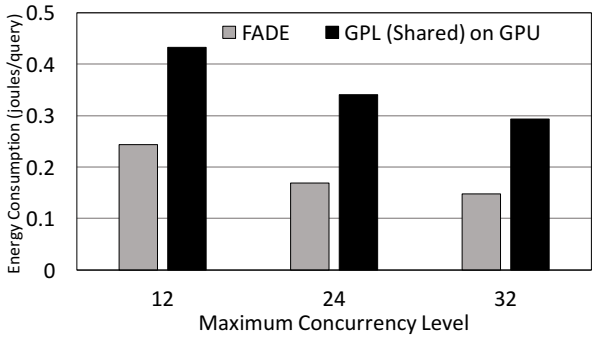

Fig. 18: Energy consumed by GPU and FPGA while executing the TPC-W Benchmark.

both workloads, the FPGA consumes much less energy than the GPU by up to 2x.

Summary. Under the context of OpenCL-based pipeline execution, the GPU offers a better end-to-end performance than FPGA, due to better PCI-e bus and better query processing performance on the accelerator. Our further comparison on accelerator only reveals that the FPGA has much better energy consumption than the GPU, and the FPGA is still slower than the GPU. Further, in terms of cost efficiency, the AMD GPU used in this study costs about 700$ while the FPGA costs about 5000$. Hence, the GPU is significantly more cost-efficient than the FPGA. Note that, the FPGA price can significantly drop for massive adoption. Therefore, the comparison among query co-processors like FPGAs and GPUs is inconclusive in terms of performance and energy consumption. We acknowledge that this study is based on specific CPU/GPU/FPGA architectures, and it is our future work to extend our study on other architectures.

VI. Related Work

Query Porcessing on GPUs. Recently general purpose GPUs have received tremendous attention from database researchers. The availability of large number of parallel cores and their huge memory bandwidth makes GPU a very attractive co-processor for accelerating database applications. In the last decade, a number of database systems have been designed specifically for GPUs [25], [23], [24], [52], [56], [44]. GPL [42] proposed a pipelined query processing engine for GPUs which makes efficient use of hardware resources available within the GPU and reduces communication overhead across kernels. But, the tile based pipeline approach is not suitable for FPGAs, which do not have on-chip caches. Similar to GPL, the pipeline design of FADE can improve the memory bandwidth utilization. However, the pipeline design of FADE was done with the consideration of FPGA hardware features and is significantly different from the pipelined implementation on GPUs or CPUs.

Query Processing on FPGAs. A number of prior studies have tried to accelerate database systems using FPGAs. Most of these works used HDLs like Verilog or VHDL for designing the database operators [34], [14], [27], [26], [32], [39], [48], [38], [41]. Woods et al. proposed Ibex [55], which is a storage engine that supports offloading of certain query operators. In Ibex the FPGA is plugged in between the SSD and the operators are executed while the data is read from the SSD. Our implementation on the other hand looks into using FPGA to accelerate operators in an in-memory database engine where the data is already loaded into the memory. The development of database systems using HDLs takes considerable amount of time and these systems are not as flexible as OpenCL [53], [54] or other High Level Synthesis (HLS) based [40], [36] approaches. The development of database systems using OpenCL is a recent phenomenon and most of these systems are still in their early stage and none of them are capable of shared query execution. To the best of our knowledge, this paper is the first pipeline shared query execution system on OpenCL-based FPGAs. Also, this study offers a detailed study on the performance and energy efficiency among CPU, FPGA and GPU under the context of pipeline execution.

Accelerator comparisons. There have been some studies comparing the performance and energy efficiency of FPGAs and GPUs. Thomas et. al. [51] studied the energy efficiency of FPGAs, GPUs and CPUs for generating random number. Betkauoi et. al. [11] evaluated the energy efficiency of FPGAs and GPUs in high productivity computing. There have also been a number of studies comparing the performance of FPGA and GPU based systems. Fowers et al. [20] conducted an in-depth study of FPGAs, CPUs and GPUs for sliding windows applications. Che et al. [16] compared the performance of GPUs and FPGAs for compute intensive applications. Studies have also looked into the performance of FPGAs and GPUs for real time applications [43], [29]. None of these studies have systematically compare OpenCL-based pipeline executions among CPUs, GPUs and FPGAs.

VII. Discussions

Through designing and implementing relational query processing on OpenCL-based FPGAs, we have identified a number of opportunities and challenges for using FPGAs as a database query co-processor.

The following are the major opportunities. First, the FPGA programmability in terms of HLS support has been improving greatly. FPGA vendors have offered OpenCL SDKs as well as debugging and performance optimization tools for better programmability. Second, the advantage of FPGAs for data processing is in the fine-grained hardware parallelism. As more memory and computation features that have been introduced to HLS SDK, new generation FPGAs can better realize the data parallelism of databases. Third, energy efficiency is a key selling point of FPGAs. As FPGAs are being deployed in the cloud computing environment, they can be used to reduce the energy footprint of many systems and applications in the cloud.

We also identified a few limitations as well as open problems of FPGAs for performing relational query processing.

First, the performance comparison between OpenCL HLS and HDL based solutions is challenging. HLS with OpenCL

could have performance loss compared to HDL-based implementation. More importantly, this comparison may reveal more opportunities in optimizing the OpenCL-based solutions. However, we find that, despite many previous studies have been published [34], [14], [27], [26], [32], [39], [48], [38], [41], none of them are open-sourced, to our best knowledge. Without proper open-sourced implementation of the previous studies and due to the resource constraints, we have to leave this comparison as a future work.

Second, as a co-processor, the FPGA requires advanced hardware and software techniques to support complex workloads more efficiently. One example is on the high reconfiguration overhead which prohibits many database workloads to execute on FPGAs efficiently. The other example is the low memory bandwidth of the test bed, which can be 10 times lower than current CPU-based systems.

Third, the further integration of database query processing techniques into FPGAs still has a large unexplored space. The techniques such as dividing a query into separate segments, multiple query execution, sharing among multiple queries, and sharing the same input stream have been studied extensively in RDBMSs and data warehouses. It is an open question on how to integrate them in the best way so that the FPGA-based systems can be optimized.

Fourth, it seems that there is still no real consensus on the integration of the FPGA into the system architecture. Some systems have used FPGAs as a co-processor, some using it as a smart controller for disk systems, and some using the FPGA to filter data coming from a network. There lacks a comparison and a study on those different integrations, and to identify their pros and cons.

Lastly, OpenCL can be recompiled and target CPUs, GPUs and FPGA architectures as well. We have witnessed that "one size does not fit all. Although OpenCL targets different architectures, architecture-aware optimizations have to be developed to unleash the power of target architectures. It is uncertain whether we can develop some self-tuning optimizations for OpenCL across different architectures.

VIII. Conclusion

The recent release of OpenCL based HLS frameworks by FPGA vendors like Xilinx and Intel/Altera have significantly improved the programmability of FPGAs and has brought new research opportunities for FPGA-based query processing systems. As a start, we design FADE, the first OpenCL based shared execution system. We then use FADE to study the performance and energy consumption of shared execution engines on FPGAs. We conduct the experiments on Intel/Altera Stratix V FPGA, in comparison with Intel Xeon E5-1650 CPU and AMD R9 Fury X GPU. Our experiments show that FADE achieves up to 10x performance speedup over an existing pipelined single query executions on FPGA and 2x lower energy consumption than GPU based shared execution systems. Still, FADE is 2x slower than its GPU-based counterparts. Our preliminary studies show that the comparison among query co-processors like FPGAs and GPUs are inconclusive in terms of performance and energy consumption. We conjecture that this study will open up further studies and optimizations of query co-processing on accelerators and their comparison. For example, cloud service providers like Microsoft [15] and Amazon [3] have already started moving towards FPGAs, as an alternative to accelerate specific tasks. The comparison between GPU and FPGA at a large-scale system could be interesting for future work.

Finally, although OpenCL-based FPGAs have demonstrated some promising results in this study, there is a number of limitations for FPGA in terms of high compilation time and high reconfiguration overhead. That leads to some limitations of FADE on supporting *ad hoc* queries. More work has to be done to resolve those limitations of FPGA to make it more general.

References

[1] Altera SDK for OpenCL Getting Started Guide. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf.

[2] Altera SDK for OpenCL Programming Guide. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[3] Amazon EC2 F1 instances, howpublished = https://aws.amazon.com/ec2/instance-types/f1/.

[4] SDAccel Development Environment User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1023-sdaccel-user-guide.pdf.

[5] Star Schema Benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[6] TPCW benchmark. http://www.tpc.org/tpcw/.

[7] Ailamaki and et al. Dbmss on a modern processor: Where does time go? In *VLDB*, 1999.

[8] S. Arumugam and et al. The datapath system: A data-centric analytic processing engine for large data warehouses. In *ACM SIGMOD*, 2010.

[9] C. Balkesen and et al. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

[10] P. Boncz and et al. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.

[11] B. Brahim and et al. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. *FPT*, pages 94–101, 2010.

[12] Breßand et al. Robust query processing in co-processor-accelerated databases. In *ACM SIGMOD*, 2016.

[13] Candea and et al. A scalable, predictable join operator for highly concurrent data warehouses. *VLDB Endow.*, 2009.

[14] Casper and et al. Hardware acceleration of database operations. In *FPGA*, 2014.

[15] A. Caulfield and et al. A cloud-scale acceleration architecture. In *MICRO*. IEEE Computer Society, October 2016.

[16] S. Che and et al. Accelerating compute-intensive applications with gpus and fpgas. In *SASP*, 2008.

[17] Cheng and et al. Parallel in-situ data processing with speculative loading. In *ACM SIGMOD*, 2014.

[18] Cieslewicz and et al. Cache-conscious buffering for database operators with state. In *DaMoN*, 2009.

[19] Diao and et al. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 2003.

[20] Fowers and et al. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In *FPGA*, 2012.

[21] Giannikis and et al. Shareddb: Killing one thousand queries with one stone. *VLDB Endow.*, 2012.

[22] Harizopoulos and et al. Qpipe: A simultaneously pipelined relational query engine. In *ACM SIGMOD*, 2005.

[23] J. He and et al. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *VLDB Endow.*, 2013.

[24] J. He and et al. In-cache query co-processing on coupled cpu-gpu architectures. *VLDB Endow.*, 2014.

[25] Heimel and et al. Hardware-oblivious parallelism for in-memory column-stores. *VLDB Endow.*, 2013.
[26] Istvan and et al. Histograms as a side effect of data movement for big data. In *ACM SIGMOD*, 2014.
[27] Z. Istvn and et al. A flexible hash table design for 10gbps key-value stores on fpgas. In *FPL*, 2013.
[28] Jiang and et al. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, 2008.
[29] R. Kalarot and et al. Comparison of fpga and gpu implementations of real-time stereo vision. In *CVPR*, 2010.
[30] Kallman and et al. H-store: A high-performance, distributed main memory transaction processing system. *VLDB Endow.*, 2008.
[31] A. Kemper and et al. Hyper: A hybrid oltp amp;olap main memory database system based on virtual memory snapshots. In *ICDE 2011*, 2011.
[32] Koch and et al. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *FPGA*, 2011.
[33] Leis and et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *ACM SIGMOD*, 2014.
[34] K. T. Leung and et al. Exploiting reconfigurable fpga for parallel query processing in computation intensive data mining applications. In *In UC MICRO Technical Report*, 1999.
[35] Makreshanski and et al. Mqjoin: Efficient shared execution of main-memory joins. *VLDB Endow.*, 2016.
[36] Malazgirt and et al. High level synthesis based hardware accelerator design for processing sql queries. In *FPGA World Conference*, 2015.
[37] Manegold and et al. Optimizing database architecture for the new bottleneck: Memory access. *VLDBJ*, 2000.
[38] Mueller and et al. Data processing on fpgas. *VLDB*, 2009.
[39] Mueller and et al. Fpga: What's in it for a database? In *ACM SIGMOD*, 2009.
[40] Mueller and et al. Glacier: A query-to-hardware compiler. In *ACM SIGMOD*, 2010.
[41] Mueller and et al. Sorting networks on fpgas. *VLDB*, 2012.
[42] J. Paul and et al. GPL: A GPU-based Pipelined Query Processing Engine. In *ACM SIGMOD*, 2016.
[43] K. Pauwels and et al. A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features. *TC*, 2012.
[44] H. Pirk and et al. Waste not; efficient co-processing of relational data. In *ICDE*, 2014.
[45] Poosala and et al. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 1996.
[46] Poosala and et al. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.
[47] Psaroudakis and et al. Sharing data and work across concurrent analytical queries. *VLDB Endow.*, 2013.
[48] M. Sadoghi and et al. Multi-query stream processing on fpgas. In *2012 IEEE 28th ICDE*, 2012.
[49] A. Silberschatz and et al. Operating system concepts, 2012.
[50] Tan and et al. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, 2015.
[51] Thomas and et al. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *FPGA*.
[52] Wang and et al. Concurrent analytical query processing with gpus. *VLDB Endow.*, 2014.
[53] Wang and et al. Relational query processing on opencl-based fpgas. In *FPL*, 2016.
[54] Z. Wang and et al. Improving data partitioning performance on opencl-based fpgas. In *FCCM*, 2015.
[55] Woods and et al. Ibex: An intelligent storage engine with support for advanced sql offloading. *VLDB Endow.*, 2014.
[56] Yuan and et al. The yin and yang of processing data warehousing queries on gpu devices. *VLDB Endow.*, 2013.
[57] Zhang and et al. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *VLDB Endow.*, 2013.
[58] H. Zhang and et al. In-memory big data management and processing: A survey. *IEEE TKDE*, 2015.
[59] Zhuravlev and et al. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News*, 2010.
[60] Zhuravlev and et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 2012.
[61] Zukowski and et al. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 2012.