

# Query processing on OpenCL-based FPGAs : challenges and opportunities

Paul, Johns; He, Bingsheng; Lau, Chiew Tong

2018

Paul, J., He, B., & Lau, C. T. (2018). Query processing on OpenCL-based FPGAs : challenges and opportunities. Proceedings of the 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), 937-945. doi:10.1109/padsw.2018.8644616

<https://hdl.handle.net/10356/142858>

<https://doi.org/10.1109/PADSW.2018.8644616>

---

© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at:  
<https://doi.org/10.1109/PADSW.2018.8644616>.

*Downloaded on 20 Mar 2024 17:24:28 SGT*

# Query Processing on OpenCL-based FPGAs: Challenges and Opportunities

Johns Paul <sup>1</sup>, Bingsheng He <sup>2</sup>, Chiew Tong Lau <sup>1</sup>

<sup>1</sup> *Nanyang Technological University, Singapore*

<sup>2</sup> *National University of Singapore*

**Abstract**—Traditionally, FPGAs were programmed using low-level Hardware Description Languages (HDLs) like Verilog or VHDL, which made it extremely difficult to design, build and maintain 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. It remains an open question 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. In this paper, we attempt to answer this question under the context of pipelined query execution. For this, we first perform a detailed study of database engines on the latest generation of FPGAs. We then design an FPGA based shared pipeline query execution system (FADE) which exploits the hardware features of FPGAs and minimizes inefficiencies like the high communication reconfiguration overhead. Our experiments show that our design achieves significant performance speedup over existing approaches for pipelined query executions on FPGA. Finally, we also present the challenges and opportunities for query processing on the latest generation FPGAs.

## I. INTRODUCTION

Over the last few decades, there have been significant efforts to accelerate query processing systems [8], [43], [20], [19], [40], [15], [36], [14]. One major field of interest has been the use of accelerators like GPUs and FPGAs to speedup query processing [40], [30], [38], [15], [39]. GPUs are more widely used [40], [30], [15], due to the ease of programming them (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) [5], [1], [2] as a high-level synthesis (HLS) frameworks. These OpenCL SDKs have made it possible for developers to expose the data parallelism of the FPGAs using OpenCL.

Still, it remains an open question 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. This is because, OpenCL is mainly designed for parallel multi-/many-core architectures and the architecture design of FPGAs is significantly different to these. 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.

In this paper, we attempt to answer this question under the context of pipelined query execution, which has been a classic query processing paradigm for data warehouses. Modern data warehouses provide significant opportunities for sharing data and computation among queries. Hence, they need shared execution systems that are capable of taking advantage of these sharing opportunities. Due to this very reason, shared systems like SharedDB [14], CJoin [10] MQJoin [24] have gained significant attention in the recent years. However, none of these existing systems support query execution on FPGAs and their design is inefficient for FPGAs.

Hence, we develop FADE, the first shared execution engine for FPGAs which solves the inefficiencies 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. 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 have conducted experiments using the SSB [6] and TPC-W [7] benchmark on an Intel/Altera Stratix V FPGA. 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 [30] of single query evaluations. Shared pipeline execution can further achieve up to 2.4x speedup over pipelined single query implementation. Finally, we also present the challenges and opportunities for query processing on new generation FPGAs.

The rest of the this paper is organized as follows. In Section II, we introduce the background and review the related work. 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. Finally, we discuss the lessons learnt in Section VI and conclude in Section VII.

## II. BACKGROUND AND RELATED WORK

### 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 [5] 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. However, due to their hardware design these pipes are fixed between a pair of operators and cannot be used for dynamically routing the data between different operators.

### B. 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 [23], [11], [17], [16], [21], [34], [26], [28], [29]. Woods et al. proposed Ibex [39], 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 [36], [37], [33], [35] or other High Level Synthesis (HLS) based [27], [25] 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.

## 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. [30] proposed the current state-of-the-art 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 often lack an on-chip cache and the data in the block RAM cannot be shared across kernels without using primitives like 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 [14], [10], [24] 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 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 and data transfer. Shared pipeline execution is effective in reducing the impact of the PCIe overhead on the overall performance by allowing maximum re-use of data.

The other benefit of shared execution systems is its ability to address the resource under-utilization encountered by existing single query systems. Our experimental results in Section V-B

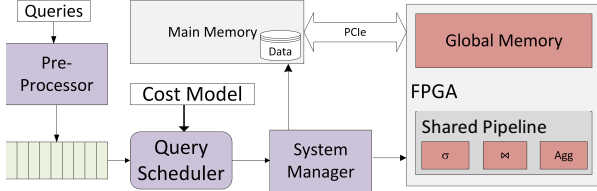


Fig. 1: System Overview of FADE.

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
$Overlap[l]$	Overlap achieved by the $l^{th}$ operator/kernel
$C_l$	Relative weight of the $l^{th}$ operator/kernel

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 are often 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. Such an approach will be further constrained by the limited amount of hardware resources available on FPGAs. 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.

#### IV. SYSTEM DESIGN

In this section we present the design of FADE, the shared pipeline 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) [14]. The system uses a frequency based histogram to estimate the selectivity of each query [16], [32], [31]. After the pre-processing, the queries are added to the *query pool*, waiting for execution.

##### B. Shared Pipeline Design

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. We use a *global query plan* (GQP) to implement shared query execution on FPGAs. The GQP based shared execution system merges the individual query plans (of a group of queries) into a single GQP after identifying opportunities for shared execution.

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 to minimize the communication and reconfiguration overhead.

1) *Operator Design*: The design of our operators are actually rooted at many of the previous pipelined execution engines [30], [13], [22]. 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 [10]. 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 design and implementation of the major operators in FADE, with an emphasis on the modification made to the 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 [30], 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

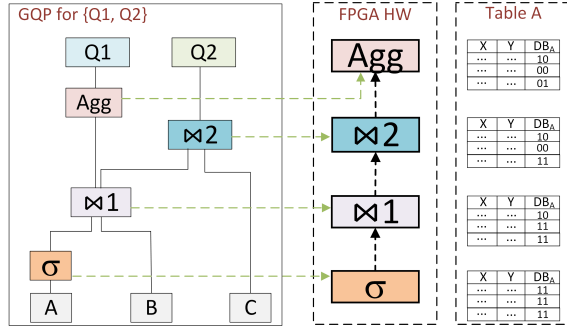


Fig. 2: A valid mapping between GQP and hardware pipeline

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.

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 Intel/Altera and Xilinx allow system designers to tune both the width and the depth of these 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, thus reducing the resources available for performing computations. Hence, in our experiments, we find that the system achieves 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

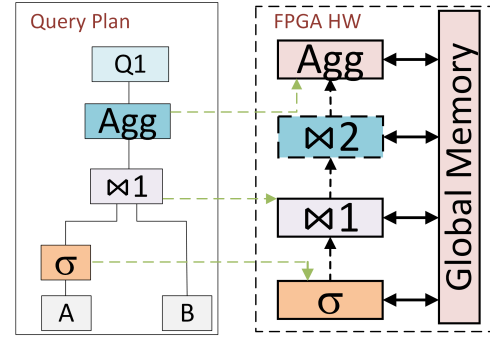


Fig. 3: Use of pass-through mode for operators

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 naive ways to execute the query on the FPGA: 1) by reconfiguring the FPGA with a new bitstream that can execute the given GQP efficiently, leading to costly FPGA reconfigurations and 2) by passing the data from the first join operator to the aggregate operator using the global memory, resulting in an increase in the number of global memory transactions. FADE on the other hand invokes 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 naive approaches.

### 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 to an operator in the hardware pipeline, while keeping the data flow dependency. An example of such a mapping is shown in Figure 2. Due to the support for pass-through mode, the mapping from GQP to the hardware pipeline is in fact more flexible, making it possible for FADE to support a wide variety of queries using a limited number of hardware pipelines.

*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.

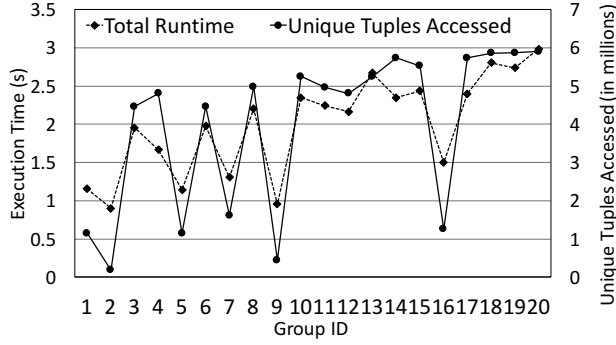


Fig. 4: Tuples accessed vs total runtime for shared query execution.

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 this 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.

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

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.

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

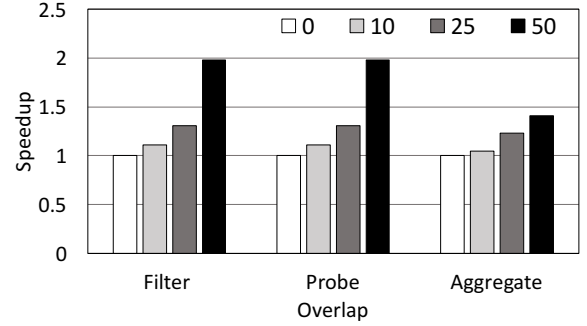


Fig. 5: Speedup achieved by different operators with varying overlap.

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 [18], [41], [42]. The problem has been proved to be NP-Hard [18], when more than two tasks/queries need to be co-scheduled. Further, existing approximation algorithms (e.g., [18]) adds very high runtime overhead when the number of queries 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  $WO$  values of the group of queries in the EL before  $Q$  was added and 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, 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 [9] 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 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, the generation of even simple bitstreams take hours and hence its impossible to perform this stage at runtime. Each bitstreams in FADE will be capable of executing a different set of queries and an 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 and support for partial reconfiguration can potentially reduce this overhead (currently not available on OpenCL FPGAs).

### 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). We use Quartus to measure the power consumption of FPGA.

**Workload.** We evaluate FADE using the following public benchmarks: SSB [6] and TPC-W [7]. The SSB data set used

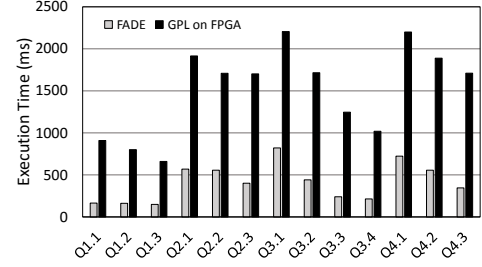


Fig. 6: Performance comparison of single query implementation of FADE and GPL on FPGA.

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 [14], 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 [7], [6].

**Experimental Outline.** Our evaluation is organized as follows. In Section V-B, we evaluate our 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. Note, that more experiments using the TPC-W benchmark and comparison between FPGAs and GPUs can be found in our technical report [4].

#### 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 data 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 overlapping data transfer with computation. This shows the

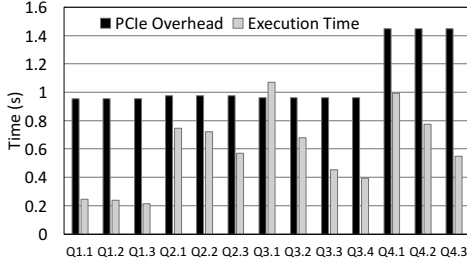


Fig. 7: PCIe overhead and execution time for SSB queries.

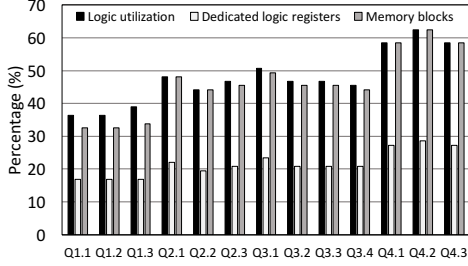


Fig. 8: Resource utilization in single query execution.

necessity of adopting shared execution, to reduce the PCIe overhead by maximizing sharing of data.

**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.

### 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, which enables the flexible design, is small (below 12%).

To further demonstrate the efficiency of our design in reducing the 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 only 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 queries belonging to two separate groups. Figure 10 shows the speedup achieved by  $S1$  when compared to  $S2$ . The results show that when reconfiguration is not required

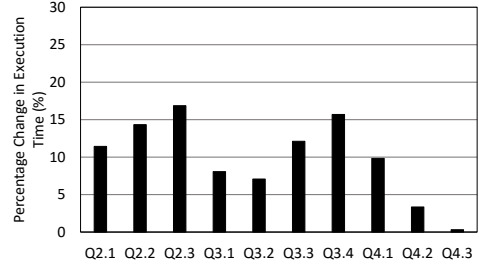


Fig. 9: Overhead associated with the use of unified design.

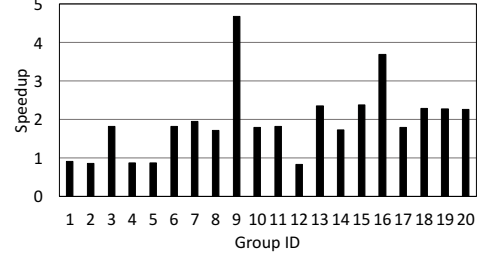


Fig. 10: Speedup achieved by unified design.

(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 system designers when optimizing for FPGAs. On the one hand, we could design pipelines capable of executing only a small set of queries and pay the high reconfiguration overhead. On the other hand, we 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 depends on the system requirements and 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

TABLE III: Shared pipeline resource utilization

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

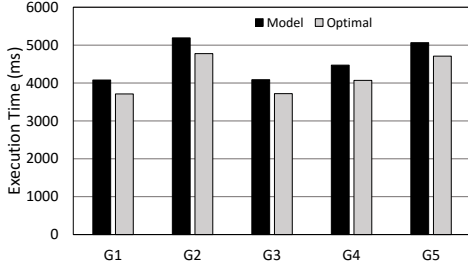


Fig. 11: Comparison of the proposed heuristic against optimal grouping.

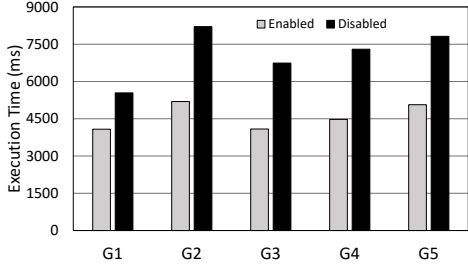


Fig. 12: Impact of using weighted sum for query scheduling

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.

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 the pipeline design of GPL on FPGA (which uses software buffers for

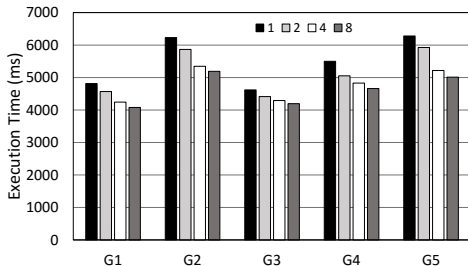


Fig. 13: Impact of using multiple execution list

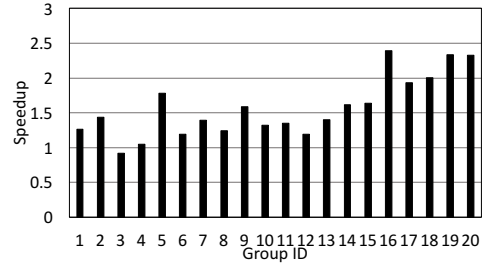


Fig. 14: Speedup of shared query execution when compared to single query execution.

communication). 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 evaluation. 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.

## VI. 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. Further, due to the new memory and computation features that have been introduced to HLS SDK, FPGAs have become a suitable target for implementing data-intensive database applications. Third, the high energy efficiency of FPGAs make them an attractive hardware accelerator for large data warehouses. As a result of this FPGAs are being deployed in the cloud computing environments to reduce the energy footprint of many systems and applications in the cloud [3], [12].

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 based systems could have lower performance when compared to HDL-based implementations. More importantly, this comparison may reveal more opportunities for optimizing the OpenCL-based solutions. However, we find that, despite the abundance of existing [23], [11], [17], [16], [21], [34], [26], [28], 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 work-

loads more efficiently. One example is the high reconfiguration overhead which prohibits many database workloads to execute on FPGAs efficiently. The other is the low memory bandwidth of the test bed, which can be 20 times lower than current GPU 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 use it as a smart controller for disk systems, and some use the FPGA to filter data coming from a network. There lacks a comparison study on these different integrations, 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.

## VII. 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 of shared execution engines on FPGAs. We conduct the experiments on Intel/Altera Stratix V FPGA and demonstrate the efficiency of our pipeline design on FPGAs.

## ACKNOWLEDGEMENT

This work is supported by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) and an NUS startup grant in Singapore.

## 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](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](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] Query processing on opencl-based fpgas:challenges and opportunities. <https://github.com/johnspaul92/FADE.git>.
- [5] SDAccel Development Environment User Guide. [http://www.xilinx.com/support/documentation/sw\\_manuals/xilinx2015\\_4/ug1023-sdaccel-user-guide.pdf](http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1023-sdaccel-user-guide.pdf).
- [6] Star Schema Benchmark. <http://www.cs.umb.edu/~poneil/StarSchemaB.PDF>.
- [7] TPCW benchmark. <http://www.tpc.org/tpcw/>.
- [8] P. Boncz and et al. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.
- [9] Breßand et al. Robust query processing in co-processor-accelerated databases. In *ACM SIGMOD*, 2016.
- [10] Candea and et al. A scalable, predictable join operator for highly concurrent data warehouses. *VLDB Endow.*, 2009.
- [11] Casper and et al. Hardware acceleration of database operations. In *FPGA*, 2014.
- [12] A. Caulfield and et al. A cloud-scale acceleration architecture. In *MICRO*. IEEE Computer Society, October 2016.
- [13] Cheng and et al. Parallel in-situ data processing with speculative loading. In *ACM SIGMOD*, 2014.
- [14] Giannakis and et al. Shreddb: Killing one thousand queries with one stone. *VLDB Endow.*, 2012.
- [15] Heimel and et al. Hardware-oblivious parallelism for in-memory column-stores. *VLDB Endow.*, 2013.
- [16] Istvan and et al. Histograms as a side effect of data movement for big data. In *ACM SIGMOD*, 2014.
- [17] Z. Istvn and et al. A flexible hash table design for 10gbps key-value stores on fpgas. In *FPL*, 2013.
- [18] Jiang and et al. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, 2008.
- [19] Kallman and et al. H-store: A high-performance, distributed main memory transaction processing system. *VLDB Endow.*, 2008.
- [20] A. Kemper and et al. Hyper: A hybrid oltp amp;olap main memory database system based on virtual memory snapshots. In *ICDE 2011*, 2011.
- [21] Koch and et al. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *FPGA*, 2011.
- [22] Leis and et al. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *ACM SIGMOD*, 2014.
- [23] 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.
- [24] Makreshanski and et al. Mjoin: Efficient shared execution of main-memory joins. *VLDB Endow.*, 2016.
- [25] Malazgirt and et al. High level synthesis based hardware accelerator design for processing sql queries. In *FPGA World Conference*, 2015.
- [26] Mueller and et al. Data processing on fpgas. *VLDB*, 2009.
- [27] Mueller and et al. Glacier: A query-to-hardware compiler. In *ACM SIGMOD*, 2010.
- [28] Mueller and et al. Sorting networks on fpgas. *VLDB*, 2012.
- [29] M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *FCCM*, 2017.
- [30] J. Paul and et al. GPL: A GPU-based Pipelined Query Processing Engine. In *ACM SIGMOD*, 2016.
- [31] Poosala and et al. Improved histograms for selectivity estimation of range predicates. *SIGMOD Rec.*, 1996.
- [32] Poosala and et al. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.
- [33] M. Roomezeh and L. Lavagno. Implementation of a performance optimized database join operation on fpga-gpu platforms using opencl. In *NORCAS*, 2017.
- [34] M. Sadoghi and et al. Multi-query stream processing on fpgas. In *2012 IEEE 28th ICDE*, 2012.
- [35] D. Sidler, Z. Istvan, M. Owaida, K. Kara, and G. Alonso. doppiodb: A hardware accelerated database. In *SIGMOD*. ACM, 2017.
- [36] Wang and et al. Relational query processing on opencl-based fpgas. In *FPL*, 2016.
- [37] Z. Wang and et al. Improving data partitioning performance on opencl-based fpgas. In *FCCM*, 2015.
- [38] Z. Wang and et al. Relational query processing on opencl-based fpgas. In *FPL*, Aug 2016.
- [39] Woods and et al. Ibex: An intelligent storage engine with support for advanced sql offloading. *VLDB Endow.*, 2014.
- [40] Zhang and et al. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *VLDB Endow.*, 2013.
- [41] Zhuravlev and et al. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Archit. News*, 2010.
- [42] Zhuravlev and et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 2012.
- [43] Zukowski and et al. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 2012.