The FQP Vision: Flexible Query Processing on a Reconfigurable Computing Fabric

Mohammadreza Najafi[1], Mohammad Sadoghi[2], Hans-Arno Jacobsen[1]
[1]Technical University Munich
[2]IBM T.J. Watson Research Center

ABSTRACT

The Flexible Query Processor (FQP) constitutes a family of hardware-based data stream processors that support dynamic changes to queries and streams, as well as static changes to the processor-internal fabric in order to maximize performance for given workloads. FQP is prototyped on field-programmable gate arrays (FPGAs). To this end, FQP supports select, project and window-join queries over data streams. While processing incoming tuples, FQP can accept new queries, a key characteristic distinguishing FQP from related approaches employing FPGAs for stream processing. In this paper, we present our vision of FQP, focusing on few internal details to support the flexibility dimension, in particular, the segment-at-a-time mechanism to realize processing of tuples of variable sizes. While many of these features are readily available in software, their hardware-based realizations have been one of the main shortcomings of existing research efforts.

1. INTRODUCTION

There is rising interest in accelerating stream processing through FPGAs (*e.g.*, [3, 4, 7, 8].) Many of these approaches are based on "compiling" static queries and fixed stream schemas into hardware designs that are synthesized to configure FPGAs. It is not uncommon for this synthesis step to take on the order of minutes to hours (depending on the complexity of the design), which is the norm for FPGAs, but is too inflexible for modern-day stream processing needs, which require the application to be able to change the query and the schema on the fly, without having to wait an extended period of time for the synthesis computation to be completed.

Furthermore, existing approaches to accelerating stream processing through FPGAs [4, 7, 8] assume that for processing query and stream modifications the arriving stream is halted, the hardware design updates are synthesized into configuration information, and the new information is uploaded onto the FPGA before processing of the event stream can resume. While synthesis and stream processing may overlap, a significant amount of time and efforts are still required to halt, re-configure, and resume the operation, which may take up to several minutes in and onto itself. More importantly, this modus operandi requires logic for buffering, handling of dropped tuples, requests for re-transmissions, and additional data flow controlling tasks, which renders this style of processing difficult in practice. These concerns are often ignored in the approaches listed above, which assumed that processing stops entirely before a new query-stream processing cycle starts.

In this paper, we aim to fill the gap between software solutions which provide the greatest degree of flexibility in query modification needs and hardware solutions which offer massive performance gains by designing an FQP that accepts new queries in an online fashion without disrupting the processing of incoming event streams. While supporting query modifications at runtime is almost trivial for software-based techniques, they are highly uncommon for custom hardware-based approaches, such as FPGAs, and have so far not received much attention in the growing body of work on accelerating data processing with FPGAs.

FQP is comprised of a parameterizable number of "online programmable blocks" (referred to as OPBs) that are inter-connected into a customizable topology. Together with a number of auxiliary components for query and tuple buffering, routing, and dispatching, the OPBs form an instance of the FQP that operates entirely on the FPGA. The inter-connection topology for the OPBs can be chosen in the manner most advantageous for the queries to be processed. For example, if the query workload lends itself for parallelism, a parallel topology can be chosen, whereas for workloads with more data dependency, a pipelined topology can be chosen. The choice of topology is performed statically and an instance of FQP is synthesized that realizes this topology. The OPB is the processing core that implements the actual query operators. It enables online changes to queries based on a number of parameters, including variable tuple size, projection attributes, selection conditions, join conditions, and join-window size.

In the design of FQP, we dealt with a number of challenges: First, a static FPGA-based query processor must over-provision resources to handle the largest expected (intermediate) tuple size, which under-utilizes system resources. Second, the change in tuple size between the join operation's inputs and output adds new challenges, especially when there is the need to use the join result as input for other operations. Third, determining a minimal processing core that can efficiently handle a variety of query operators, some of which are stateless, while others are stateful.

The contributions of this work are manifold: (1) We outline our vision for stream processing on hardware in the context of our proposed FQP architecture. (2) We develop FQP, that unlike the state-of-the-art, enables online changes of queries and stream schema without interrupting query processing over incoming streams and without the need to re-synthesize the design. (3) We unify and share the underlying storage buffer for both data and operator parameters of a query. (4) We support variable tuple sizes by proposing the segment-at-a-time processing model, namely, an abstraction that divides a tuple into smaller chunks that are streamed and processed as a consecutive set of segments. This strategy avoids the need for over-provisioning of hardware resources. (5) We design FQP as a family for instantiating stream processors that are based on the different inter-connection topologies most suitable for the expected workload.

2. RELATED WORK

Over the past few years several projects on accelerating stream processing with FPGAs have been undertaken, many among researchers in the broader data management community. This work effectively demonstrated that FPGAs are a viable option for accelerating certain data management tasks in general, and stream processing in particular (*e.g.*, [3, 4, 7, 8, 5].)

Lockwood *et al.* [3] present an FPGA library to accelerate the development of streaming applications in hardware. Similarly, Meuller *et al.* [4] present Glacier, a component library and compiler, that compiles continuous queries into logic circuits on an operator-level basis. These approaches (including [7, 8]) are characterized by the goal of representing queries in logic circuits to achieve the best possible performance. While performance is a major design goal for us as well, we additionally aim to offer the application flexibility of updating queries at runtime.

The prototype of a hardware stream processor were recently presented [5, 6]. Najafi *et al.* [5, 6] showed the viability of building a stand-alone stream processor with FPGAs. The work presented in this paper builds upon this prototype, elaborating on the full-blown *Flexible Query Processor* systems project. For example, here, details are provided about the *segment-at-a-time processing* mechanism to support the processing of stream tuples with variable tuple sizes. But, more importantly, in the current work, we offer our broader and long-term vision of the FQP project.

3. MOTIVATING FQP VISION

To present our vision of stream processing acceleration, we first need to better understand the design space that must be navigated by resorting to FPGAs in order to complement or to replace general purpose processors. We begin by describing the challenges faced by today's general purpose processors.

Large & complex control units — The design of general purpose processors is based on the execution of consecutive operations on data residing in the system's main memory. The design must guarantee a correct sequential order execution. In presence of such strict execution order, the processor includes complex logic to increase performance, *e.g.*, super pipelining; out-of-order execution; single instruction, multiple data; and hyper-threading. As a result, the performance gain comes at the cost of having to devote resources (*i.e.*, transistors) to large and complex control units, which could occupy up to 95% of chip area [1].

Memory wall & von Neumann bottleneck — The current computer architecture suffers from the limited bandwidth between CPU and memory. This bandwidth is small compared to the rate at which the CPU itself can process. This issue is often referred to as the memory wall, and is becoming a major scalability limitation as the gap between CPU and memory speed increases [2]. To mitigate this issue, processors have been equipped with large cache units. However, the effectiveness of these units depends on the memory access patterns of executing programs. Additionally, the von Neumann bottleneck also contributes to the memory wall by sharing the limited memory bandwidth between instructions and data.

Redundant memory accesses — The current computer architecture enforces that data arriving from an I/O device is first read/written to main memory before it is processed by the CPU, which is a cause for a great deal of memory bandwidth loss. Consider a simple data stream filtering operation that would not require the incoming data stream to be first written to main memory. If the data arrives from the network interface, then in theory the data could stream directly through the processor. However, today's computer architecture prevents this modus operandi. Essentially, any I/O to and from the computer system is channeled through memory, which is a potential bottleneck even for high-speed inter-connects such as infiniBand that introduces latency on the order of microseconds.[1]

These performance limiting factors in today's computer architecture have resulted in a growing interest in accelerating data management and data stream process-

[1]Emerging InfiniBand adaptors, such as Mellanox ConnectX-3 Pro (`http://www.mellanox.com/page/infiniband_cards_overview`), have the potential to enable FPGAs to process data in-line and avoid the data movement overhead between I/O and memory.

ing on FPGAs. By designing custom hardware accelerators tailored for streaming processing, we can afford to get by with simpler control logic and better usage of chip area, *i.e.*, we can achieve higher performance per transistor ratio. Furthermore, we can mitigate memory wall issue, by coupling processor and local memory and instantiating many of these coupled processors and memories as needed. The redundant memory access could be reduced by avoiding copying and reading memory whenever possible.

Additionally, the stream processing has a set of unique characteristics that enables us to further exploit underlying hardware. For instance, by its nature, there is a higher chance of repeated execution of the same set of queries over a given potentially unbounded stream with a known stream data schema. Such repetition creates an opportunity to customize data path of our processors for optimal computation (*e.g.*, avoid deep instruction execution pipeline). Essentially, we execute the "data over the instructions" not the other way around, namely, instructions are implemented in hardware as a custom logic. Another distinct feature of streaming application is the I/O nature, where tuples go from input to processing to output without the need for storage in off-chip memory for an extended period of time, *e.g.*, state-less operations such as selections and projections do not require any write to (external) memory; they can simply be processed online in a pure streaming fashion. Also given the custom hardware implementation of queries, there is a greater degree of predictability, essentially eliminating costly branch mispredictions, page misses, and fewer instructions fetching. All of these properties have motivated us to rethink the development of a revolutionary different architecture for a stream processor that avoids today's computer architecture challenges.

4. THE FLEXIBLE QUERY PROCESSOR

The Flexible Query Processor (FQP) is a customized hardware solution we designed for building stream processors that can be specifically tailored to a given set of queries executed over a data stream. Furthermore, new queries can be inserted at run-time without requiring expensive re-synthesis, as is commonplace today in related FPGA-based processing approaches.[2] Essentially, FQP represents a set of components that can be assembled in various ways to give rise to a whole family of stream processors. The basis of FQP are *Online Programmable-Blocks* (OPBs) (*i.e.*, the processing cores.) The OPB itself is a simple stream processing element that supports a number of basic query operators over stream tuples. An OPB can be dynamically programmed by inter-sparsing the input data stream with new or updated queries, represented by a simple instruction set. The structure of input data stream distribution, OPB arrangement, and query result collection components define the *connection topology* of FQP. As a result, aside from the flexibility of FQP to be reprogrammed with new queries, the topology can be tailored to a specific query set (application), to maximize processing performance. OPBs are designed such that they can be connected to each other serially (*i.e.*, a *pipeline* arrangement), in parallel (*i.e.*, a *parallel* arrangement), and in a mixed manner (*i.e.*, *hybrid* arrangement).

[2]This is a unique limitation of FPGA-based approaches not found in software-based approaches, in which dynamic query insertion and update is hardly an issue worth underlining.

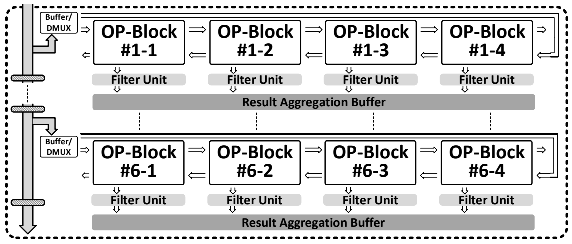

Figure 1: Partially parallel FQP topology.

4.1 FQP Overview

The processing performance of an instance of FQP is determined by its internal connection topology, the performance of each individual OPB, and the assignment of queries to OPBs. Figure 1 shows a small-scale partially parallel FQP topology comprised of 24 OPBs. Each one of the four consecutive OPBs per row are arranged in a pipelined fashion. All rows are arranged in parallel. Other topologies are possible, as determined appropriate by a pre-synthesis-time software-based configuration component that assembles an instance of an FQP, specifically tailored for the given or expected query workload. After synthesis, during query assignment, a query compiler determines a mapping of the input queries onto the given FQP instance. Queries can be inserted dynamically without requiring a resynthesis of the FQP instance, a major differentiation of our work from related approaches.

4.2 FQP Internal Architecture

Data stream distribution circuitry — In FQP instance shown above, we opted for a pipelined data stream distribution architecture, where each incoming tuple is inserted from the top and passes to the next pipeline stage (*cf.*, hashed blocks in the figure) in each clock cycle until it reaches the end of the path.

In each stage, the tuple is fed to the corresponding chain of OPBs. Depending on the configuration, more OPB-chains could be connected to a single stage. However, the number of attached chains is limited by a maximum fan-out. Attaching more chains to a single stage can result in a decrease of the FQP's clock frequency, leading to a performance degradation of the processor. The maximal fan-out is device (*e.g.*, FPGAs) dependent and has to be determined experimentally for a given FPGA. While feeding tuples through the pipeline of chains, some chains could be busy processing previous tuples. This imposes unwanted stalls in the distribution circuitry, leaving further chains idle. To address this is-

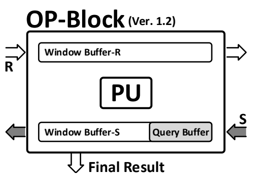

Figure 2: Stream processing element.

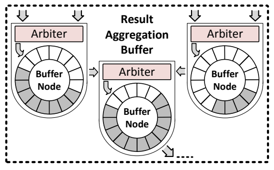

Figure 3: Result aggregation buffer (RAB).

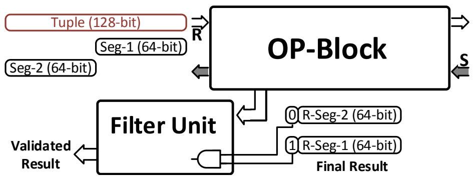

Figure 4: Segment-at-a-time at entry to OPB.

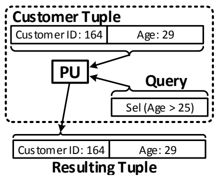

Figure 5: Customer segregation query.

sue, we use a *Buffer/DMUX* component which stores incoming tuples in its internal buffer and feeds them to the connected chain. This component also contains a demultiplexer which splits streams so they pass though the shared data stream distribution circuitry.

Online-programmable block (OPB) — FQP is comprised of a number of OPBs as basic stream processing elements that realize various query operators (*e.g.*, select, project, and join). Figure 2 shows a high-level diagram illustrating the ports of an OPB.

OPB itself is comprised of several components which work in parallel to maximize throughput. Here, we opt to briefly present the *Processing Unit* (PU) and the two window buffers as two important components of our design.

Designed to execute complex operators such as a window-join, the OPB includes two window buffers, with the maximum window sizes as pre-synthesis-time configurable parameters. *Window Buffer-R* is dedicated to input *Stream-R* (R) while *Window Buffer-L* is dedicated to input *Stream-S* (S) and to the reception of dynamically inserted queries, also referred to as *Query Buffer*.

The PU is the actual execution unit of OPB. Upon insertion of a new tuple, the PU fetches instructions from the *Query Buffer* and executes them against the tuples from one or both window buffers (depending on the query semantic). At the end of execution, the resulting tuples are emitted via the *Final Result* port or via the *Stream-R* output port for further processing by neighbouring OPBs (*i.e.*, for larger queries.)

Results at the *Final Result* port are gathered by the *Filter Unit* and after validation are fed to the *Result Aggregation Buffer* (RAB) for transmission to the output port of FQP. Validation includes tasks such as computing validity of an entire result tuple from its constituent parts, produced by the PU (*cf.*, segment-at-a-time mechanism discussed below.)

Result collection circuitry — After tuple processing, validated results are collected by the *Result Aggregation Buffer* (RAB), shown in Figure 3. The RAB is comprised of a structure of connected buffers (*i.e.*, *Buffer Nodes*) that are responsible for collecting results from two sources and guiding them from the OPBs to the output port of the FQP. In this collection step, a fairness granting mechanism makes sure that both sources are treated equally to avoid starvation. Appropriate tuning of *Buffer Node* parameters (*e.g.*, buffer size) and connectivity architecture is important as this affects overall FQP performance. In other words, poor assignment of parameters could result in bottlenecks in the transmission of resulting tuples, while the majority of buffers would be under-utilized.

4.3 Segment-at-a-time Processing

Not only queries change throughout the life of an application, but the streams themselves evolve as well. Their properties such as schema, tuple size, and input rate change continuously. These features are at odds with today's FPGA-based stream processing solutions, which have, for the most part, been tailored to process one specific tuple width before requiring re-synthesis if tuple size changes are permitted at all. This degree of flexibility poses a severe challenge for a hardware-based solution, as opposed to its software counter-part. Our design has been specifically built to afford this flexibility. The OPB-based design of FQP supports varying size tuples, thus, allowing for evolving data streams.

Generally speaking, hardware systems have fixed size input ports, internal communication buses, and output ports. FQP is no exception. However, flexibility in the face of varying size data streams stems from the way an OPB processes incoming tuples. The parametrized design of the OPB allows us to define its ports' width prior to design synthesis. By default, we configure FQP with a 64-bit port width. As a result, for any tuple larger than 64 bits, it is divided into 64-bit segments at the entry-point to FQP. The tuple segments arrive at the input port of an OPB as shown in Figure 4. Then, the OPB processes each segment, one at a time, and hands over the resulting segments to the *Filter Unit* through its *Final Result* port.

Figure 6 shows the segment-at-a-time processing mechanism in more details. Prior to processing a segmented tuple, queries also need to be updated to handle the segments. In our example, the query consists of two segments, of which the first segment corresponds to the first segment of the tuple, while the second segment of the query corresponds to the second segment of the tuple. Segmentation of queries is performed in software outside of FQP.

The PU fetches the first segment of the tuple from the *Window Buffer-R* as well as the first segment of the query. Then, the PU executes the segment of the query and produces a result segment with an additional flag which shows if the first segment of the tuple satisfies the (query) conditions in the first segment of the query. This process is repeated for the second segment of the tuple etc.

All resulting tuple segments are transmitted to and

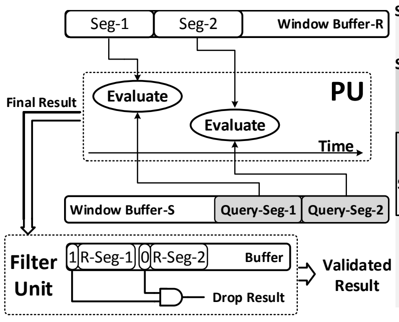

Figure 6: OPB **Segment-at-a-time.**

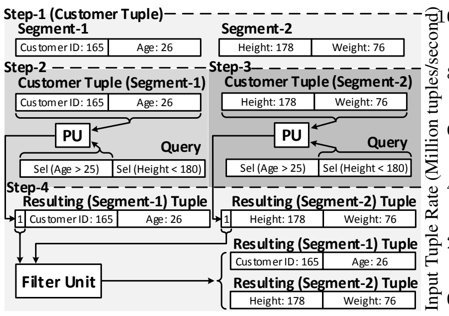

Figure 7: Segregation query.

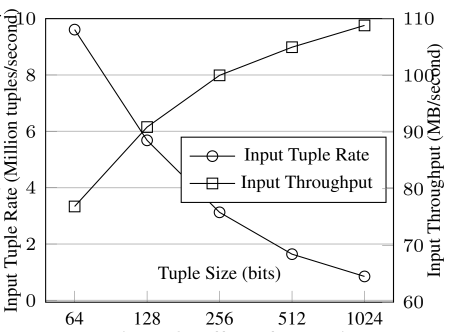

Figure 8: Effect of tuple size.

stored in internal buffers of the *Filter Unit* (FU), which evaluates the validity of the entire resulting tuple. For example, in a selection operator, one of the tuple segments may not pass the selection condition, while others do[3], which would render the entire tuple invalid. After receiving the final segment and positively validating the result, the FU hands the tuple (a segment at a time) over to the RAB to transfer it to the output port of FQP. Otherwise, the FU drops all result segments.

Segment-at-a-time for join operator – Query assignment is a task performed in software that maps the input queries onto the available blocks of the FQP configuration. This task determines the placement of operators, which is not known a priori (*i.e.*, we do not know where a join operator executes). Segment-at-a-time processing is necessary to support the further processing of tuples that result from a join operation as often, the join result is comprised of both input tuples (unless attributes are projected out).

Segment-at-a-time tuple size limit – The maximally accepted tuple size is determined by the size of *Window Buffer-(R&L)* in the OPB. From a conceptual point of view, the size of *Window Buffer-R* is not limited for stateless operators (*i.e.*, select and project), while this is not the case for stateful operators (*i.e.*, join). The actual limit depends on the resources available on the FPGA, which is highly device-specific and will only increase in future FPGAs. With today's technology, we have synthesized blocks with window sizes of up to 4K bytes.

5. VARIABLE TUPLE SIZE EXAMPLE

Here, we give an example to illustrate the segment-at-a-time mechanism realized by the OPBs. Assume a `Customer` stream with `Customer ID` and `Age` fields.

```
CREATE STREAM CS_SEL AS
SELECT *
  FROM Customer_Stream
 WHERE Age > 25
```

Furthermore, assume a query to segregate customers into two groups, those who are older and those who are younger than 25 years of age (*e.g.*, a retailer wanting to compute recommendations based on age.)

This query is programmed onto the OPB and executed over the customer tuples as shown in Figure 5. As the `Customer` stream evolves over time, new attributes, such as `Height` and `Weight`, are added (*e.g.*, for the retailer to better differentiate recommendations.)

```
CREATE STREAM CS_SEL AS
SELECT *
  FROM Customer_Stream
 WHERE Age > 25, Height < 180
```

Thus, the query is re-written as follows and through the segment-at-a-time mechanism, the OPB can execute the new query over the larger tuples without any changes as shown in Figure 7.

The processing of the updated (larger) tuple is done in four steps. In Step 1, after the query for the updated tuple schema was re-programmed onto its (target) OPB, the updated (enlarged) tuple is divided into two segments at the entry point of the FQP. In Step 2, that is, after the segments arrive at the target OPB, the *Processing Unit* fetches the first part of the query (`Age` $>$ 25) and executes it on the first segment of the tuple. In Step 3, the same process is repeated for the second part of the query (`Height` $<$ 180) and the second segment of the tuple. Each one of these steps produces a resulting tuple segment together with a validation flag. Finally, in Step 4, the resulting tuple segments are processed jointly using the *Filter Unit*. In case all segments have satisfied the query conditions, they are handed over to the RAB for transfer to the output port of the FQP. In this example, for illustration purposes, we have kept the data stream simple. In practice, segment-at-a-time is applicable to larger tuples with more attribute-value fields.

6. EXPERIMENTAL EVALUATION

We developed all FQP components in VHDL that are configured and synthesized on our Xilinx ML505 development board. In our experiments, the input was generated by a workload generator and passed through an Ethernet component and pipelined reception buffers to the FQP stream processor. The input streams consist of 64-bit long tuples (*i.e.*, 32-bit attribute and 32-bit value).

Raw processing power evaluation – We first present the raw processing power of various queries by focusing on the number of operators on a topology similar to Figure 1, where window size is 16 and clock frequency is 125MHz. For the selection and projection operators, OPB is capable of supporting $|\text{Window Buffer-L}|/2$ independent selection operators or $|\text{Window Buffer-L}|$ independent projection operators. Each OPB is capable of realizing a single join operator. OPBs connected in

[3]*E.g.*, the higher order bits pass the condition, while the lower order ones do not (for two segments).

a chain (OP-Chain) can realize join operators with even larger window buffers. For example, utilizing two, three, or four OPBs increases the window size two, three, or four times, respectively. The processing performance of each OPB for the join operator tightly depends on its window buffers' sizes. For a window size of 16 tuples, the current version of OPB is capable of processing 1.44 million tuples per second. The raw processing power of the topology given in Figure 1 is summarized in Table 1.

Table 1: Tuple processing rate.

Operators	# Operators	Million Tuples/s
Selection	24×8	230.6
Projection	24×16	272.6
Join	24	34.5
Chained Join (4)	6	8.6

Each OPB is capable of processing at the rate of 9.61M, 11.36M, or 1.44M tuples per second for the selection, projection, and join operator, respectively, which translates to 230.6M, 272.6M, or 34.5M tuples per second for the topology in Figure 1. By chaining 4 OPBs we have 6 OP-Chains each with a window size of 4×16 and a total processing rate of 8.6M tuples per second.

Segment-at-a-time evaluation – To evaluate our segment-at-a-time feature of OPB, and to study its influences on the input rate, we utilized a data stream by varying the number of attribute-value pairs per tuple (1 to 16), in which the size of each attribute-value pair is 64 bytes. The clock frequency in this experiment was 125MHz. Figure 8 demonstrates the input tuple rate achieved as we feed larger tuples to an OPB. By feeding larger tuples, the sustainable input tuple rate decreases as expected, since the size of tuple and the number of attribute-value pairs doubles each time. However, interestingly for a double size tuple the processing time does not necessarily double as seen in this figure. This is due to the reduction in the amortized cost of tuple handling that is mostly for the first segment and decreases for the subsequent segments. These results are for the selection operator, but they are applicable for other operators including the projection and join operators.

7. CONCLUSIONS & OPEN PROBLEMS

Our broader vision is to identify key opportunities to exploit the strength of available hardware accelerators given the unique characteristics of stream processing. As a first step towards fulfilling this goal, we have developed FQP, a generic streaming architecture composed of a dynamically re-programmable stream processing elements, (*i.e.*, OPBs) that can be chained together to form a customizable processing topology (exhibiting a "Lego-like" connectable property). We argue that our proposed architecture may serve as a basic framework for both academic and industry research to explore and study the entire life-cycle of accelerating stream processing on hardware. Here we identify a list of important short- and long-term problems that can be tackled within our FQP framework.

• What is the complexity of query assignment to a set of custom hardware blocks (including but not limited to OPBs). Note, a poorly chosen query assignment may increase query execution time, leave some blocks unutilized, negatively affect energy use, and degrade the overall processing performance.

• How to formalize query assignment algorithmically (*e.g.,* develop a cost model), and what is the relationship between query optimization on hardware and classical query optimization in databases. Unlike the classical query optimization and plan generations, we are not just limited to join reordering and physical plan selections, but there is a whole new perspective on how to apply instruction-level and fine-level memory-access optimization (through custom hardware implementation, *e.g.*, different OPB implementations). For example, what is the most efficient method for wiring custom operators to minimize the routing distance? How to collect statistics during query execution and how to introduce dynamic re-wiring and movement of data given a fixed FQP topology?

• What is the best initial topology given a query workload as *a prior*? For example, one can construct a topology in order to reduce routing (*i.e.*, to reduce the wiring complexity) or to minimize chip area overhead (*i.e.*, to reduce the number of OPBs).

• Given the topology and the query assignment formalism, is it possible to generalize from single-query optimization to multi-query optimization, where we amortize executing cost across the shared processing of the n queries and explore inter- and intra-query optimization that are inspired by the capabilities of custom stream processors?

• Finally, how do we extend query execution on hardware to co-processor design by distributing and orchestration query execution over different hardware with unique features such as CPUs, FPGAs, and GPUs? An important design decision arises as to how these various devices communicate and whether or not they are placed on a single board, thus, having at least a shared external memory space, or placed on multi-boards and connected through interfaces such as PCIe.

8. REFERENCES

[1] Symmetric key cryptography on modern graphics hardware. Advanced Micro Devices, Inc., 2008.

[2] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[3] J. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers. A low-latency library in FPGA hardware for high-frequency trading (HFT). In *HOTI*, 2012.

[4] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *VLDB*, 2009.

[5] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *VLDB*, 2013.

[6] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Configurable hardware-based streaming architecture using online programmable-blocks. In *ICDE*, 2015.

[7] M. Sadoghi, H.-A. Jacobsen, M. Labrecque, W. Shum, and H. Singh. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB*, 2010.

[8] M. Sadoghi, R. Javed, N. Tarafdar, R. Palaniappan, H. P. Singh, and H.-A. Jacobsen. Multi-query stream processing on FPGAs. In *ICDE*, 2012.