2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines

Centaur: A Framework for Hybrid CPU-FPGA Databases

Muhsen Owaida David Sidler Kaan Kara Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zürich
{firstname.lastname}@inf.ethz.ch

***Abstract*—Accelerating relational databases in general and SQL in particular has become an important topic given the challenges arising from large data collections and increasingly complex workloads. Most existing work, however, has been focused on either accelerating a single operator (e.g., a join) or in data reduction along the data path (e.g., from disk to CPU). In this paper we focus instead on the system aspects of accelerating a relational engine in hybrid CPU-FPGA architectures. In particular, we present Centaur, a framework running on the FPGA that allows the dynamic allocation of FPGA operators to query plans, pipelining these operators among themselves when needed, and the hybrid execution of operator pipelines running on the CPU and the FPGA. Centaur is fully compatible with relational engines as we demonstrate through its seamless integration with MonetDB, a popular column store database. In the paper, we describe how this integration is achieved, and empirically demonstrate the advantages of such an approach. The main contribution of the paper is to provide a realistic solution for accelerating SQL that is compatible with existing database architectures, thereby opening up the possibilities for further exploration of FPGA based data processing.**

I. Introduction

Traditionally the scope of operators and data types in relational databases is limited. The operators are defined by SQL and the data can only be stored in basic and well defined data types. As a result, database engines are able to highly optimize the operator implementations for each data type and CPU architecture. Additionally, most database operators have a relatively low computational complexity. However, the rise in data sizes leads to new challenges and opportunities. There, is in particular, increasing interest in complex analytics operations in the context of machine learning, statistics, and graph analytics. Given that traditional databases are not able to perform well on these complex operators and data types, accelerators offer an alternative way to implement such complex functionality in databases. An important question arising here is how to integrate an accelerator like an FPGA in a database engine.

Databases are throughput oriented, processing thousands to several hundred thousand queries per second. This means the type of operators potentially offloaded to the FPGA are constantly changing. In addition, the database engine should be able to run hybrid queries combining FPGA and CPU operators in any order to harness the benefits of both FPGA accelerators and highly optimized CPU operators.

State-of-the-art research has shown clear benefits from offloading database operators to the FPGA [28], [9], [12], [30]. To avoid the overhead of additional data movement, many approaches place the FPGA on the data path as a bump-in-the-wire accelerator between storage (or network) and the CPU. As a result, only a limited number of operators in the query tree can be pushed to the accelerator. Therefore, the actual integration of hardware accelerators into a database engine is still an open problem.

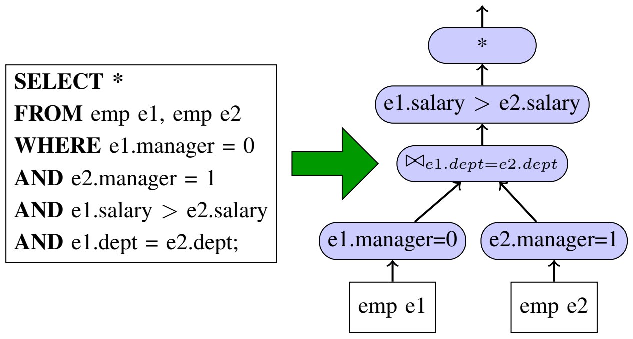
In this paper we present Centaur, a framework bridging the gap between the database engine on the CPU and database operators running on the FPGA. In Centaur, we focus on the system aspects of integrating FPGA operators into a database engine rather than on the implementation of the operators themselves. Centaur provides software abstractions for the operators and a thin hardware layer to facilitate concurrent and dynamic offloading of operators to the FPGA. In developing Centaur, we have three objectives that are essential for enabling FPGA accelerators in databases: i) Seamless integration within the operator tree execution model of databases; ii) Support for hybrid execution with operators running both on the CPU and FPGA; iii) Flexible deployment of operators on the FPGA to support concurrent queries, operator pipelining, and dynamic allocation to different queries.

For a seamless integration of the FPGA operators, Centaur makes use of user defined functions (UDFs) common in databases. This makes deployment of operators either to CPU or FPGA transparent to the database engine. Centaur supports pipelining of software and hardware operators thus enabling hybrid query execution. With Centaur, lightweight operators can remain on the CPU and only compute-intense operators are offloaded to the FPGA. Hardware operators can also be pipelined to reduce data movement between the FPGA and main memory and improve performance further.

Centaur is tailored for CPU-FPGA shared memory platforms where the FPGA operators have direct coherent access to the same memory region as the CPU such as Intel's Xeon+FPGA [18] and IBM's Power8 CAPI [20] architectures. Thanks to coherent shared memory, the FPGA operators are active workers which autonomously issue memory reads and writes. This autonomy enables hybrid query execution. To demonstrate Centaur's concepts and techniques, we have integrated it into the popular open source database MonetDB [23].

The paper makes the following contributions:

- A high-level software abstraction for hardware operators, called FThreads.
- Hardware architecture that facilitates concurrent and dynamic offload of database operators to the FPGA.
- Dynamic pipelining of operators to overlap processing



```
SELECT *
FROM emp e1, emp e2
WHERE e1.manager = 0
AND e2.manager = 1
AND e1.salary > e2.salary
AND e1.dept = e2.dept;
```

Fig. 1: Example query and operator tree

of either hardware and software or multiple hardware operators.

- Exploring and demonstrating the functional integration of FPGA accelerators into a database engine through UDFs.

Centaur has been developed on the Intel's HARP v1 prototype machine [18][1], which does not support partial reconfiguration. Accordingly, Centaur does not provide dynamic reconfiguration and the FPGA has to be programed a priori. However, no changes are required to Centaurs software or hardware when switching between different FPGA configurations. Partial reconfiguration will be available in the next version of Centaur, which will run on HARP v2.

II. Background

A. MonetDB

Centaur has been used to integrate FPGA accelerators into MonetDB [23], an open source column store. In a column store, the database tables and their records are partitioned vertically, resulting in one column per attribute. Since many operations in a database are memory-bound, this can lead to a performance improvement depending on the workload. In the meanwhile, most engines supporting analytical queries are column oriented (e.g., SAP Hana [5]). The specific column data layout used in MonetDB are binary association tables (BAT). Each BAT consists of two columns, the first storing an optional object identifier (OID) and the second column storing a value. The value column has a specific data type. The original data and also all intermediate results are stored in BATs.

1) Operator Tree: The task of the query optimizer in a database is to combine the operators used in a query into an operator tree. Figure 1 shows an example query and the corresponding operator tree. As we can see in the first stage, the two select operators, evaluating if an employee is a manager, read data directly from the employee base table identified as *e1* and *e2*. In the next step the records produced from the two select operators are joined based on the department attribute ($e1.dept = e2.dept$) and in the third stage another select operator compares the *salary* of employees *e1* with the managers *e2*. In the last step all records matching the previous criteria are returned as a result. When building the operator tree, the query optimizer considers the dependencies between different operators and also the cost model for each operator to calculate the optimal execution order. Having this flexibility of combining operators at runtime means they need to implement a well-defined interface and follow the same execution model. Centaur's FPGA based operators employ the user defined function (UDF) interface, thereby guaranteeing that hardware operators can be accessed through the same well-defined interface as software operators.

2) User Defined Functions (UDFs): Many databases can be extended with UDFs. For our work, we use the UDF interface to extend MonetDB with operators implemented on the FPGA. Using a UDF is like using any other function in the database, as can be seen from the following query:

```
SELECT COUNT(*) FROM product_sales
WHERE quantity > 1
AND foobar(price) < 10;
```

The UDF called *foobar()* operates on the price attribute and returns a value which is then compared to the value 10. Since UDFs implement the same interface as other operators, the query optimizer can integrate and execute them seamlessly as part of the operator tree.

Most databases only allow UDFs which operate on one tuple at a time which means the database invokes the UDF for each record. For FPGA based UDFs, handing over every record individually is not efficient. However, MonetDB also allows UDFs to operator on a complete BAT, thus minimizing the overhead of invoking an operator on the FPGA. In Centaur, we exclusively use the BAT interface.

III. System Overview

Figure 2 depicts the main components of Centaur. Since Centaur bridges the gap between the hardware operators and the database, it has two main components, one in software and one in hardware. It provides an *Application Interface* (Figure 2) on the software side and the *FThreads Manager* on the FPGA. Centaur defines templates and interfaces that allow developers to extend Centaur with more hardware operators through the UDF and Operator library. Centaur builds on the low level communication interface provided with the Intel's Xeon+FPGA platform. Intel provides an *Accelerator Abstraction Layer*(AAL) and encrypted QPI endpoint. Through AAL, Centaur bootstraps the FPGA, configures the QPI endpoint, and writes the base address of the CPU-FPGA shared memory region into an FPGA register.

A. Software Components

The Application Interface provides software abstractions to create and monitor hardware threads, called FThreads, which map to FPGA operators. FThreads have a similar interface as the software threads in the C++ standard library (std::thread). Through the Application Interface, MonetDB can access the custom memory allocator in Centaur to allocate BATs in the CPU-FPGA shared memory region so the FPGA operators can access them. In addition, it provides concurrency control such that multiple queries can access the FPGA in parallel.

[1]Results in this publication were generated using pre-production hardware and software donated to us by Intel, and may not reflect the performance of production or future systems.

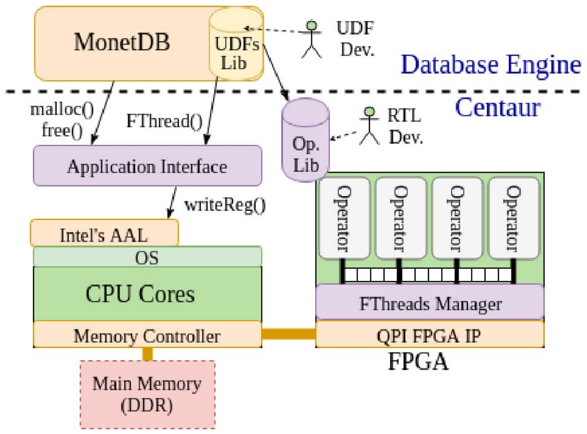


Fig. 2: Centaur overview.

In addition to the Application Interface, Centaur presents an extensible hardware operators library. Although Centaur in itself does not provide a comprehensive set of database operators, the operators library was designed to be easily extended with additional hardware operators.

B. Hardware Components

Centaur's *FThreads Manager* is responsible for mapping hardware operator requests generated by the UDFs to the corresponding Operator Units. An Operator Unit is the hardware implementation of a single database operator (such as select or aggregation). They are connected to Centaur's *FThreads Manager* through a well defined interface as will be discussed in section V.

The *FThreads Manager* time multiplexes access to the QPI interface between the multiple Operator Units. In addition, it can dynamically compose hardware and/or software operators into a pipeline as we will discuss in Section IV-D. Centaur connects adjacent Operator Units with FIFOs to pipeline FThreads running on two adjacent units. Hence, only two or more successive operators can be pipelined as Figure 2 shows.

C. CPU-FPGA Communication

When a MonetDB database instance is started, a handshake is established between the Application Interface and the FThreads Manager. Then the base address of the shared memory region is written to the FThreads Manager using AAL primitives. After this step, both the Application Interface and FThreads Manager have access to the same memory region and can use it for further communication.

The Application Interface passes requests to run hardware operators to the FPGA through a set of concurrent queues which reside in the shared memory. When a UDF requests a hardware operator, the request is packaged in the proper format and then enqueued into the corresponding queue. The FThreads Manager polls on the queues looking for new requests. Whenever there is a request in a queue, it reads it and assigns it to a free Operator Unit of the same type. In the same way, when an Operator Unit finishes execution, it writes its status to a shared memory address monitored by the UDF which initiated the operator. As polling on main memory from the FPGA introduces a memory bandwidth overhead, we tune the polling interval to find a balance between the request processing delay and the bandwidth overhead. The resulting overhead is negligible as shown in section VI-B.

Using shared memory for communication is a platform independent approach that facilitates porting Centaur to different CPU-FPGA shared memory architectures.

IV. Application Interface

A. Shared Memory Management

Due to the fact that not all the main memory region is shared between CPU and FPGA (only 4 GB is shared), the Application Interface implements a custom memory allocator which provides *malloc()* and *free()* functions to allocate/deallocate data chunks in the shared memory region. We modified MonetDB's memory allocation to use Centaur's memory allocator to allocate and deallocate BATs so they are accessible from the FPGA. Allocating BATs directly in the shared memory region avoids copying them from non-shared to shared memory when an FPGA operator is invoked. Further, all data structures used for communication between the Application Interface and the FThreads Manager are also allocated through this allocator. We expect that in future versions of the Xeon+FPGA platform these limitations will disappear and the FPGA will have access to the complete memory address space through the standard operating system memory allocator.

B. FPGA Operators Abstraction

The Application Interface provides an extensible library of FPGA operators which can be invoked from UDFs. Adding an operator to the library involves two things: An RTL implementation of the operator which is designed in compliance with the FThreads Manager I/O interface and a software function which encapsulates the operator *OPCODE* and its arguments in a data structure called *operator descriptor*. The operator descriptor is used to create an FThread instance which is then deployed on the FPGA. Through the FThread interface, users can query the status of the operator while running on the FPGA to detect if it is done or monitor performance metrics.

C. Hardware User Defined Functions

In Figure 3 we show an example of a hardware UDF that invokes the *testcount* operator by first calling the wrapper function *fpga_testcount()* from the operators library. It then passes the returned operator descriptor object to the *FThread* constructor which will create the corresponding FThread request and forward it to the FPGA through the concurrent

```
testcount(test, value, src, dst) {

  // Create Job Request
  FThread tc_fthread(fpga_testcount(test,value,src,dst));

  // Wait for FThread to finish
  tc_fthread.join();
}
```

Fig. 3: Typical hardware UDF structure

queues. The created FThread request (named *tc_fthread* in the UDF) is joinable in the same way a software thread is joined.

D. Composing Operator Pipelines in a UDF

The operator tree model of a query plan (refer to Figure 1) and the column-at-a-time execution model in MonetDB makes operator pipelining a natural match for the operator execution model in Centaur. In software, pipelining operators incurs an overhead due to software thread synchronization. On the other hand, in hardware, there is no synchronization overhead since data is passed through hardware FIFOs. In fact, pipelining of hardware operators reduces the required bandwidth to main memory improving overall throughput. Centaur provides the *PipelineJob* and *FPipe* abstractions which allows the UDF to compose operator pipelines to benefit from data locality on the FPGA and to overlap the execution of two operators to increase performance.

Figure 4 shows an example of a hardware UDF that combines a *regex* operator (regular expression matching operator) and a *testcount* operator. Both reside on the FPGA, which will essentially return the count of matches from the regex operator.

First an *FPipe* object is allocated for the two operators. The *FPipe* data type is passed to the template function `allocate_fpipe<short>()`. In the example, the `short` data type (C++ data type) is used since the data type of *regex* output is 16 bits. Once the *FPipe* object is allocated, the *PipelineJob* request is created by passing the operators descriptors (*regex* and *testcount*) and the *FPipe* object.

A pipeline of two software and hardware operators can be created in a similar way. Instead of the *FPipe* data structure which implements a FIFO in the FPGA, the *FQueue* data structure can be used. The *FQueue* is a single-producer single-consumer concurrent FIFO which resides in the shared memory space such that it is accessible from software and hardware threads. Centaur will allocate the *FQueue* to pipeline two hardware operators if they cannot be connected through an on-chip FIFO. The *FPipe* and *FQueue* abstractions can be considered similar to OpenCL pipes [14].

It is the responsibility of the UDF developer to decide if a pipeline should be created between two operators or not. As Centaur only connects adjacent Operator Units with on-chip FIFOs on the FPGA, it will create an FQueue pipeline resource between two hardware operators if the corresponding Operator Units are not connected through on-chip FIFO on the FPGA.

```
regexcount(src, expr, dst) {

  // Allocate Pipeline resource
  FPipe* pipe = allocate_fpipe<short>(REGEX_OP, TESTCOUNT_OP
      );

  // Create Pipeline Job Request
  PipelineJob regexcount_job(fpga_regex(expr, src, pipe),
                   pipe, fpga_testcount('>', 0, pipe, dst) );

  // Wait for Pipeline Job to finish
  regexcount_job.join();
}
```

Fig. 4: Creation of on-chip pipeline job

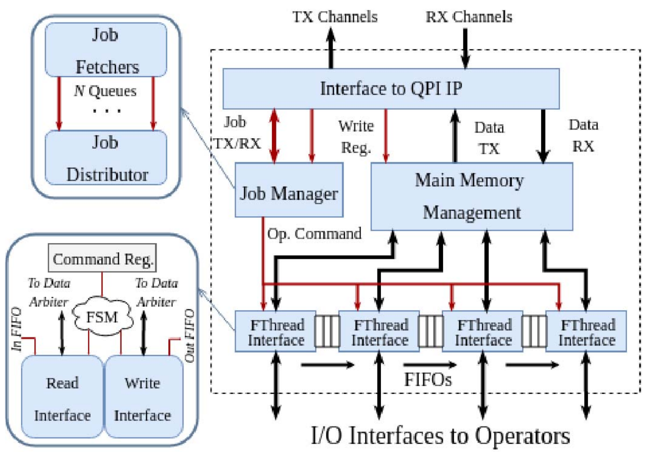


Fig. 5: FThreads Manager architecture.

E. Concurrent Execution of Queries

A key requirement in accelerating database workloads is the ability to execute different queries (from different clients) concurrently. The Application Interface does not provide the notion of FPGA ownership for a single query. MonetDB clients (each running in its own software thread) will compete to insert job requests (UDF calls) into the concurrent job queues. The Application Interface provides a different queue for every operator type. This makes scheduling of operator requests from different types completely independent of each other. This is possible, because the FThreads Manager on the FPGA monitors the job queues and processes queued requests independently of each other.

The FThread is the scheduling unit, where an Operator Unit (Figure 2) executes an FThread to completion before executing the next FThread. Only when executing a pipeline of operators, the pipelined FThreads are scheduled as a group to avoid stalls that could degrade the performance of the pipeline. How the FThreads Manager processes and schedules incoming FThread requests through job queues is discussed in Section V.

V. FThreads Manager

In this section we describe the architecture of the FThreads Manager (Figure 5). It has three main components: a Job Manager which polls on concurrent job queues and fetches job requests; a Memory Management Unit which time multiplexes the I/O interfaces and handles virtual address translation; and an FThread Interface Unit which provides communication abstractions for the operators.

A. Job Manager Unit

The *Job Manager* is responsible for fetching and processing job requests inserted by the Application Interface into the concurrent queues. Initially, when the CPU-FPGA handshake is established, the *Job Fetchers* start polling on the concurrent queues in the shared memory region (see Figure 5). There are as many *Job Fetchers* as there are job queues (the maximum number of job queues is limited by the number of Operator Units on the FPGA).

A job request includes either a request for a single operator (i.e., a single FThread execution) or multiple operators configured as a pipeline. Once a *Job Fetcher* fetches a job request, the *Job Distributor* will try to assign the requested operator(s) to free Operator Unit(s) of that operator type. In case of multiple operators per request, it looks for adjacent Operator Units. If the requested Operator Unit is free, the *Job Distributor* sends a *command* to it, holding all the parameters necessary for the execution. However, if the requested Operator Unit is busy, the job is queued until the required Operator Unit is free, and the corresponding *Job Fetcher* pauses fetching. The *Job Distributor* is able to observe the state of all job queues and processes available requests in a round-robin fashion. The *Job Distributor* spends 1 cycle processing a *Job Fetcher* output, and continues to process the next *Job Fetchers*.

B. Memory Management Units

Since the current version of Intel's QPI IP provides one read and one write channel, a data arbiter multiplexes read and write requests from different operators. The FPGA contains a page table of 2000 entries (addressing 4 GB of memory), to do the address translation from virtual to physical addresses. Each entry in the page is 17-bit wide and represents a 2 MB super page. The page table sits on the path between the QPI IP and the data arbiter. It is implemented in BRAM and is only loaded once during the handshake between the Application Interface and the FThreads Manager and is not updated afterwards.

C. FThread Interface Unit

The FThread Interface Unit (Figure 5) implements the hardware support for the FThread abstraction in the Application Interface. Every FThread is assigned two memory pointers: One pointer to the location of its status data structure and another pointer to the data structure holding the operator arguments. When the FThread Interface Unit receives a new operator command, it first loads the arguments from main memory and then triggers the Operator Unit to start execution. This initialization of an FThread resembles the process of creating a software thread. Similarly, when the Operator Unit terminates, the FThread Interface Unit updates the status data structure. From the UDF's perspective, the call to the *join()* function of the FThread object returns indicating that the execution on the hardware terminated.

In addition to these control mechanisms, the FThread Interface Unit provides an abstraction for memory access from the operator. On top of raw address based memory access, the operator can use a FIFO abstraction with simple push/pop semantics. This FIFO abstraction can be mapped to hardware FIFOs (512-bits wide) or FQueues (shared memory). Thanks to this abstraction, it is transparent to the operator if it is using a hardware FIFO or an FQueue. Centaur uses this abstractions to dynamically pipeline operators as explained in Section IV-D.

VI. Experimental Evaluation

A. Experimental Setup

We use Intel's Xeon+FPGA platform with MonetDB to deploy Centaur. The Intel's Xeon+FPGA platform is a two-socket machine with a 10-core Intel Xeon E5-2680 v2 CPU (clocked at 2.8 GHz) in one socket and an Altera Stratix V 5SGXEA in the other. The two sockets are connected through QPI. On the CPU socket, 96 GB of main memory are installed, accessible to the FPGA through the QPI link. No DDR memory is attached to the FPGA socket. Memory access from the FPGA is bound by the bandwidth of the QPI link which we measured to peak at 6 GB/s. In contrast, the CPU has direct access to the memory resulting in up to 25 GB/s memory bandwidth.

TABLE I: Database operators for the experiments.

Name	Description
regex	Regular expression matching on a column of strings
testcount	Test entries of column of integers against given condition and counts the matches.
percentage	For a given column of integers it computes the sum of selected entries divided by the sum of all entries.
muladd	Evaluates "$a * X + b$" for column X of integers.

We modified MonetDB version 11.21.19 to integrate the Application Interface of Centaur and allow hardware UDFs to execute on the FPGA. The FPGA architecture of Centaur is configured with 4 Operator Units and clocked at a frequency of 200 MHz.

B. Microbenchmark: Centaur Overhead

To evaluate the overhead of the FThreads Manager on the QPI throughput consumed by the FThreads, we compared a copy operator (i.e., read a cache line and write it back) to the Intel AAL loop-back benchmark provided by Intel. The overhead measured is negligible: 1.6%. The small overhead is caused by the *Job Fetchers* in the Job Manager which poll on the concurrent job queues in shared memory. The QPI throughput consumed by *Job Fetchers* can be tuned against the response time for enqueuing a job request in the job queues. From our experiments we found that the throughput overhead can be minimized to less than 0.2%, at a response time of $12.7\mu s$, which is acceptable when queries run for milliseconds. We chose a point with approximately 3 μs response time and around $100MB$ throughput as the optimal point (1.6% of QPI throughput).

Centaur incurs a 78 μs overhead over the operator execution time. Most of this time is consumed in creating the FThreads, as well as allocating and freeing shared memory regions. A very small fraction of the time ($3\mu s$) is consumed in enqueuing the FThread request, preprocessing on the FPGA, and writing the *DONE* status back. For queries taking a few milliseconds or more to execute, this is an acceptable overhead.

C. Pipelining Operators

In this section we evaluate the benefits of Centaur's pipelining mechanisms between operator units. Table I lists the operators we used in this evaluation. As our objective is to show how different operators can be combined and used in Centaur and not the operators performance, the chosen operators reflect different I/O behavior. The *muladd* is a streaming operator that reads and writes at line rate. The *percentage* operator reads at line rate from multiple sources (two columns) and writes back a scalar value. The *testcount* operator reads from one column at line rate and produces a single scalar value. The *Regex* is

```
Q1:  SELECT regexcount(comment,
         '[7-9]*.(rating|rank|grade)')
     FROM product_sales;

Q2:  SELECT regexpercentage('[7-9]*.(rating|rank|
     grade)', comment, revenue)
     FROM product_sales;

Q3:  SELECT muladdpercentage(2, 1, revenue)
     FROM product_sales;
```

Fig. 6: Queries executed on product sales data using the Combined operators UDF.

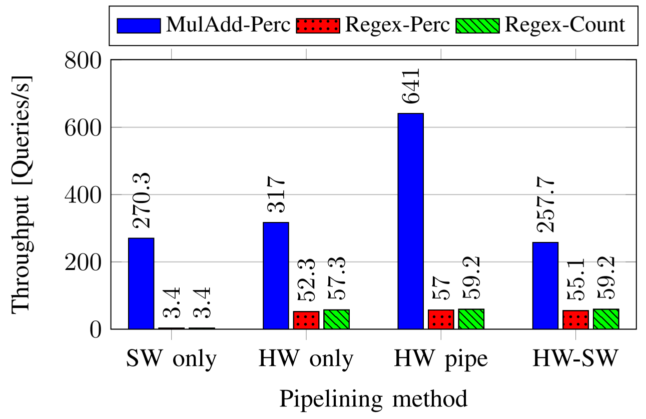


Fig. 7: Throughput for different ways of combining operators.

a more complex operator which alternates between reading a block of string pointers and string data. It consumes almost the complete available memory bandwidth.

For the *Regex* operator, we base our implementation on state-of-the-art approaches [22], [29]. Figure 6 shows the queries we run for the combined operators experiments. The experiments run on a table of size 1.25M records. We implement a UDF for each operator combination using either a hardware to hardware pipeline or a hardware-software pipeline, as explained in Section IV-C.

We experimented with 4 different scenarios of combining operators as Figure 7 shows. *"SW only"* runs all operators on the CPU, *"HW only"* deploys both operators in hardware but without pipelining them. *"HW pipe"* uses Centaur's on-chip pipelining mechanism between the two hardware operators. Finally, *"HW-SW"* deploys the first operator (*regex* or *muladd*) in hardware and the second operator on the CPU pipelined through Centaur's *FQueue*.

The experiments demonstrate that Centaur's FThreads abstraction makes it possible to combine operators in all different scenarios. The numbers in Figure 7 show that different scenarios achieve different performance with pipelining and that using hardware operators not always providing a benefit. A noticeable benefit from on-chip pipelining occurs when the two operators have relatively similar cost and produce/consume data at same rate as is the for the pipeline of *muladd* and *percentage* operators. Our objective here is to show that Centaur supports all these scenarios and gives the query optimizer the chance to choose where and how to deploy operators. A subsequent step in developing Centaur is to model and expose these different possible combinations to the query optimizer to choose the proper query plan. This will be done as part of future work (an example of what that entails can be found in [25]).

TABLE II: Workload Description

	W1		W2	
Queries	#records	#attributes	#records	#attributes
Regex	300K, 600K	-	600K, 1.25M	-
Skyline	100K, 1M	4, 5, 6, 7	100K, 1M	4, 5, 8, 12, 16

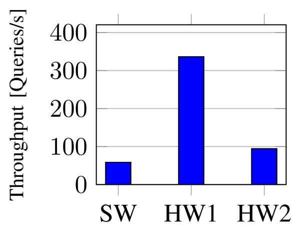
D. Multi-Query Workloads

This experiment demonstrates how Centaur is behaving when executing typical database workloads. In addition to the *Regex* operator we used in the previous section, we use the *Skyline* operator to synthesize mixed workloads of complex and simple operators. Additional operators such as Kara et al. [13] partitioning operator can be used in the same way.

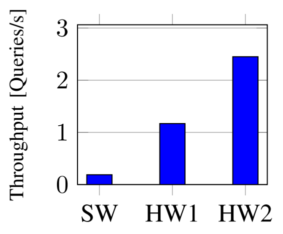
Skyline query. For a given set of data records, for example hotels information (number of stars, users rating, distance to the beach, price, etc.), the skyline operator computes the pareto-optimal front of the data set, where no data record is dominated by any other record. We used Woods et al. [27] implementation of the skyline operator. With increasing number of attributes, the execution time of the skyline operator increases significantly. The skyline operator is an example of iterative compute-bound operators with variable runtime similar to many machine learning operators. It's I/O behavior resembles operators that access many different columns in a burst fashion. The skyline operator we use is not memory bound as it consumes around 1 GB/s of memory bandwidth.

We synthesized two workloads, both run 10 clients, and each client runs 100 queries (table II). The 100 queries are a mix of 90% simple queries and 10% complex queries (5% Regex, and 5% Skyline). The first workload (*W1*) is designed to have moderately complex queries, while the second workload (*W2*) increases the complexity of the skyline queries by increasing the number of attributes. Figure 8 shows the experiment results for the two workloads. We experimented with three different scenarios: 1) All queries are executed in software (*SW*). 2) Offload both Regex and Skyline to the FPGA which is configured with 2 Regex and 2 Skyline operator units (*HW1*). 3) Offload only the skyline operator to the FPGA which is configured it with 4 skyline operator units (*HW2*).

The numbers in Figure 8 demonstrate that depending on the database workloads a different system configuration achieves the highest throughput. In the first workload, the cost of *Regex* and *Skyline* operators is relatively similar, hence offloading both operators to the FPGA delivers better performance. On the other hand, the *Skyline* operator dominates the performance of the second workload because *Skyline* queries that work on 12 and 16 attributes are very expensive (take from 2 to 3 min). Hence using all the FPGA resources to accelerate the *Skyline* operator, we achieve a higher performance.



(a) Workload 1



(b) Workload 2

Fig. 8: Throughput for different ways of combining operators.

The results in this experiment motivates us to integrate partial reconfiguration capabilities in Centaur. Due limitations of the hardware, we could not provide partial reconfiguration capability in the current version of Centaur. However, Centaur provides the environment and abstractions that makes dynamic offloading of operators and managing the FPGA resources feasible, a prerequisite for the ability to dynamically decide which operators unit to reconfigure, load a new bitstream and run on the FPGA. We plan to implement this in the next version of Intel's Xeon+FPGA platform.

E. *Resource Utilization*

Table III lists the amount of resources consumed by the different components on the FPGA. We report the resource utilization for a single FThread Interface Unit (not the 4 units) to give an idea on how the resource utilization will scale when varying the number of operators. The numbers in the brackets are the percentage of the total FPGA resources. We also report the resource utilization by the different Operator Units we used. Note that the *Regex* and *Skyline* are significantly more expensive than the other operators. On the current experimental platform the QPI endpoint is implemented on the FPGA and uses almost 30% of the logic resources.

VII. Related Work

A. *FPGA Programming Frameworks*

Several research efforts have tried to bridge the gap between CPU and FPGA by extending conventional OS concepts of multi-threading and virtualization to FPGA accelerators [15], [1], [2], [24], [10], [4], [3]. ReconOS [15] and HThreads [1] extend the multi-threading programming model to express and manage FPGA accelerators as hardware threads by providing an interface to create hardware threads and synchronize them using hardware mutexes and semaphores. In ReconOS hardware threads are coupled with a delegate software thread which facilitates OS services for the hardware thread. Similarly, FUSE [10] extends posix threads to encapsulate hardware accelerators, but without the need for software delegate threads. In our work, the FThread abstraction follows the same line of imitating software threads. However, a fully fledged multi-threading framework with hardware mutexes is not necessary as operators in the query plan execute sequentially. Synchronization is only required to pipeline operators through on-chip FIFOs or FQueues.

Chen et al. [4] propose a framework for virtualizing FPGA resources in the cloud. They implement a job queue and a job manager to facilitate sharing FPGA resources among multiple users. A single job queue is sufficient when users share the FPGA logic resources not configured functionalities. In Centaur, multiple clients share the same operators, having one job queue per operator type prevents different operators from blocking each other. Dessouky et. al and Asiatici et. al [3] considered the idea of uses on chip page table to assist memory virtualization, in the same way we do.

There has also been recent work in supporting distributed data processing on platforms such as MapReduce and Hadoop. For instance, Wang et al. [26] show how to run MapReduce on a platform developed using OpenCL. Huang et al. [8] describe a system, Blaze, that integrates FPGAs as accelerators into Hadoop YARN as a service (FaaS). These type of frameworks are similar to Centaur only at a high level due to the different kind of data processing being addressed. For instance, in Blaze, a single accelerator is deployed on the FPGA and tasks are scheduled one at a time. Sharing the FPGA between different threads occur by scheduling them on a first come first serve basis executing them one after the other. On the other hand, Centaur targets throughput oriented OLAP/OLTP workloads, and has been built to accommodate concurrent operators such that multiple threads can share the FPGA resources simultaneously. Similarly, the tasks scheduler (Centaurs job manager) resides on the FPGA instead of software to minimize Centaur's overhead on critical CPU cycles used by the database engine. And, of course, Centaur targets database processing on a single machine while both [26] and [8] have considerable additional functionality for distributed data processing.

B. *Database Accelerators*

Many researchers have studied the use of FPGAs to accelerate data processing in database systems [16], [17], [19], [28], [30], [21]. Recent work from Samsung researchers [6], [11] highlights the benefits of near-storage computation in accelerating database analytical workloads. By integrating ARM processors into an SSD, data filtering can be implemented at the storage to reduce the amount of data that has to be transferred to CPU. IBM Netezza [9] and BlueDBM [12] are data processing appliances for complex analytical workloads. Both platforms deploy a pipeline of projection and filtering operators on the data path between storage/network and CPU to filter irrelevant data records early on and thereby boosting database performance. Woods et al. [28] propose an intelligent storage layer implemented with an FPGA attached

TABLE III: Resource Utilization

Module	**ALMs**		**M20k**	
FThreads Manager (total)	23,252	9.9%	315	12.3%
Job Manager	3,695	1.6%	10	0.4%
Memory Management	3,525	1.5%	113	4.4%
FThread Interface	4,011	1.7%	48	1.9%
Regex operator	25,662	10.9%	297	11.6%
Skyline operator	33,358	14.2%	209	8.2%
Muladd operator	830	0.4%	0	0%
Percentage operator	1,296	0.6%	0	0%
Testcount operator	600	0.3%	0	0%
QPI Endpoint	67,288	28.7%	110	4.3%

to a SSD. Supported queries include selection, projection, and aggregation. Masato et al. [30] present an architecture of interconnected FPGA-boards equipped with flash storage and offload UDFs commonly used in OLAP workloads to the FPGA. Sukhwani et al. [21] suggest an FPGA architecture as a pipeline of decompression and predicate evaluation operators connected to main memory through PCIe to accelerate OLAP workloads.

These research efforts show how database operators can be organized in a pipeline to process a stream of data. However, none of them explicitly discusses or explains how an FPGA operator is represented in the software of the database engine, and how the database engine interacts with and invokes the operators on the FPGA. In this paper we focus on the integration of an FPGA accelerator in a database engine by expressing it as hardware thread and having a flexible FPGA architecture that allows for concurrent and hybrid execution of database operators.

In addition to using FPGAs for data acceleration, researchers have been exploring different accelerator types such as GPUs. He et al. [7] present GDB, a database built on top of a CPU-GPU platform. Their work focuses in particular on query plan optimizations, as well as partitioning of the operators and data between CPU and GPU cores. They extended the query optimizer with cost models for GPU operators. In a recent work [25], the authors applied their approach to build a cost model for FPGA operators. Based on this model the query optimizer can evaluate a wide variety of operator combinations and optimizations. This work nicely complements our own in showing how a query optimizer can take advantage of an accelerator.

VIII. Conclusions

In this paper we have presented Centaur, a framework for accelerating database operators on hybrid CPU-FPGA machines. The main purpose of the paper is to explore the architectural aspects of the system. Using the UDF interface together with the FThread abstraction allows for seamless integration of FPGA operators in the database engine. Allowing the hardware operators to run independently and processing job requests concurrently in the FThreads Manager enables database clients to execute hardware operators concurrently. Further, the ability to run hybrid queries and combine operators in different ways provides plenty of options for the optimizer to find an optimal query plan, but also allows for a wide range of queries to benefit from FPGA acceleration. Supporting partial reconfiguration would be a next step to accelerate a wider pool of queries and provide higher throughput. We make Centaur available as open source to enable both database and hardware developers to conduct further research on integrating FPGAs in database systems[2].

Acknowledgments

We would like to thank Intel for their generous donation of the HARP v1 prototype.

[2]https://www.systems.ethz.ch/fpga/centaur

References

[1] J. Agron and D. Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *CODES+ISSS'09*.

[2] D. Andrews, D. Niehaus, R. Jidin, M. Finley, et al. Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro*, 24(4), July 2004.

[3] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne. Designing a virtual runtime for FPGA accelerators in the cloud. In *FPL'16*.

[4] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. Enabling fpgas in the cloud. In *Computing Frontiers'14*.

[5] F. Färber, S. K. Cha, et al. Sap hana database: data management for modern business applications. *ACM SIGMOD*, 40(4), Dec. 2011.

[6] B. Gu, A. S. Yoon, D.-H. Bae, et al. Biscuit: A framework for near-data processing of big data workloads. In *ISCA'16*.

[7] B. He, M. Lu, K. Yang, R. Fang, et al. Relational query co-processing on graphics processors. *ACM TODS*, 34(4), Dec. 2009.

[8] M. Huang, D. Wu, C. Hao Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale. In *SoCC'16*.

[9] IBM/Netezza. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011. http://www.redbooks.ibm.com/abstracts/redp4725.html.

[10] A. Ismail and L. Shannon. Fuse: Front-end user framework for o/s abstraction of hardware accelerators. In *FCCM'11*.

[11] I. Jo, D.-H. Bae, A. S. Yoon, et al. YourSQL: A high-performance database system leveraging in-storage. In *VLDB'16*.

[12] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankorn, M. King, S. Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *ISCA'15*.

[13] K. Kara, J. Giceva, and G. Alonso. FPGA based data partitioning. In *SIGMOD'17*.

[14] Khronos Group. *The OpenCL Specification: Version 2.0*.

[15] E. Lübbers and M. Platzner. ReconOS: Multithreaded programming for reconfigurable computers. *ACM TECS*, 9(1), Oct. 2009.

[16] R. Müller, J. Teubner, and G. Alonso. Data processing on FPGAs. In *PVLDB'09*.

[17] R. Müller, J. Teubner, and G. Alonso. Glacier: a query-to-hardware compiler. In *SIGMOD'10*.

[18] N. Oliver, R. Sharma, S. Chang, et al. A reconfigurable computing system based on a cache-coherent fabric. In *ReConFig'11*.

[19] D. Sidler, Z. István, M. Owaida, and G. Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD'17*.

[20] J. Stuecheli, B. Blaner, C. Johns, et al. CAPI: A coherent accelerator processor interface. *IBM J. Research and Development*, 59(1), Jan 2015.

[21] B. Sukhwani, H. Min, M. Thoennes, P. Dube, et al. Database analytics acceleration using FPGAs. In *PACT'12*.

[22] J. Teubner, L. Woods, and C. Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *SIGMOD'12*.

[23] M. Vermeij, W. Quak, M. Kersten, and N. Nes. Monetdb, a novel spatial column-store dbms. In *FOSS4G'08*.

[24] Y. Wang, J. Yan, X. Zhou, et al. A partially reconfigurable architecture supporting hardware threads. In *FPT'12*.

[25] Z. Wang, J. Paul, H. Cheah, B. He, and W. Zhang. Relational query processing on OpenCL-based FPGAs. In *FPL'16*.

[26] Z. Wang, S. Zhang, B. He, and W. Zhang. Melia: A MapReduce framework on OpenCL-based FPGAs. *TPDS*, 27(12), Dec. 2016.

[27] L. Woods, G. Alonso, and J. Teubner. Parallel computation of skyline queries. In *FCCM'13*.

[28] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *PVLDB*, 7(11), July 2014.

[29] Y. Yang, W. Jiang, and V. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *ANCS'08*.

[30] M. Yoshimi, R. Kudo, Y. Oge, et al. Accelerating OLAP workload on interconnected FPGAs with flash storage. In *CANDAR'14*.