University of Kaiserslautern
Department of Computer Science
Database and Information Systems

Seminar
Optimizing data management on new hardware

Summer Semester 2014

Table of Contents

Data Processing on FPGAs

Stefan Hemmer

Technische Universität Kaiserslautern, Germany

Abstract. Ever growing data sets and rapid improvements in technology together with the demand of faster, or even real-time analytics, have led the database research community to revisit some of their oldest concepts. Of special interests are often those concepts that can be combined with modern technology. An innovative example is the combination of database machines with field programmable gate arrays (FPGAs). The idea of database machines was to use specialized hardware to process and analyze data. FPGAs, as a modern form of specialized hardware, are integrated circuits that offer a series of unique advantages to the data processing world like adaptability, low power consumption, a high degree of parallelism as well as low latency and high throughput rates. For these striking reasons the following report illustrates the structure of FPGAs to demonstrate their advantages. Since there are several strategies of integrating an FPGA into a system, this report focuses on the presentation of data processing scenarios for FPGAs in context of these strategies.

1 Motivation

Technological advancements and enterprise growth have resulted in increasing quantities of business data. In a time of such *big data*, data management is becoming an increasingly difficult task. In order to gain valuable business insight and by that acquire a better market responsiveness, customer satisfaction and overall competitive advantage, companies are looking for ways to optimize online analytical processing (OLAT) of data. Especially ad-hoc analytics queries tend to be rather resource consuming. In light of these requirements a series of innovations has been made, that try to cope with increasing data sizes from different perspectives.

Until a few years back improvements were mostly made in software optimization terms, including new software architectures like column stores [SAB+05] or MapReduce-Style engines [DG08]. While these approaches all improve performance in the context of large data sizes, the hardware side of the solution space was largely ignored over several years.

More recently the database research community has started to look into those hardware oriented solutions. This research includes approaches relying on traditional general-purpose hardware like utilization of multi-core architectures with traditional central processing units (CPUs) [AKN12], SIMD operations [CNL+08] and graphics processing unit (GPU) acceleration [GGKM06]. Over and above there have also been advances with data processing using heterogeneous hardware [MTA09a, CO14, HSM+13] i.e. data processing using hardware were not all cores are equal. These approaches follow the directive to map the hardware to the task at hand, instead of mapping the task to a fixed general-purpose architecture.

The advances in memory technology (i.e. rapidly decreasing memory costs) have made approaches that are apart from CPU-centered solutions especially interesting. They allow a better utilization of memory bandwidth than data processing with solely CPUs. A CPU in a von-Neumann-architecture often has to deal with problems like the *von-Neumann bottleneck* (i.e. data transfer rate between CPU and memory limited by a shared bus) or the *memory wall* (i.e. significant gap between CPU speed and memory speed). By utilizing direct memory access (DMA) specialized hardware

is able to access system memory independently from the CPU. If the state of an operation can be held by a device, specialized hardware does not suffer the effects of the von-Neumann bottleneck or the memory wall and thus can improve the overall memory bandwidth.

Moreover common database workloads do not solely provide methods for analytical processing but are also responsible for the online transactional processing (OLTP) part of business. These OLTP applications provide Service-Level-Agreements (SLAs) for transactional processes that must not be interfered with by analytical processing tasks since they are typically bound to revenue generation. As mentioned above, typical analytics queries involve complex operations like sorting, aggregations or join operations, which can consume a significant amount of CPU time. For this reason businesses that try to utilize the advantages of growing consumer data typically rely on snapshot warehousing. With this technique a copy of the relevant data is analyzed. However in order to realize the maximum benefits of analytical processing, businesses are pushing toward real-time business intelligence. Offloading OLAT tasks to dedicated hardware poses an attractive alternative to realize such real-time analytics without endangering SLAs.

Therefore instead of utilizing additional general-purpose processors (GPPs) the hardware accelerated data processing approach tries to utilize the benefits of heterogeneous hardware. One manifestation of heterogeneous hardware is the field programmable gate array (FPGA). A FPGA is an integrated circuit (IC) that can be tailored to the application scenario. It consists of a large array of logic gates that can be specified dynamically using a hardware description language (HDL), to implement any complex logical function on hardware. By that, FPGAs represent a bridge between the two extremes, the specialized and the general-purpose world. Compared to application-specific integrated circuits (ASICs) and GPPs (CPUs and GPUs), they can provide a higher adaptability, flexbility and scalability.

There are already several industrial solutions like IBM's *Netezza* [Fra11] or xTremeData's dbX [SDCV10] that employ FPGAs in different variations and architectural styles. In research there are several approaches that range from specialized FPGA implementations for certain data operations [TWN13, MVB+, BP05, MSNT11, KT11, HSM+13] over query-to hardware compilers [MTA10, NSJ13, STM+13] to general programming language (GPL) compilers [SG08, HHBR08]. Each product and approach showcases the special benefits that can be realised by the FPGA's low-level granularity parallelism, low latency, flexibility and small power consumption.

The remainder of this work is organized as follows. Chapter 2 provides a hardware-oriented introduction of FPGAs and showcases how logic functions can be implemented on an FPGA. The third chapter initially introduces the architectural alternatives of building a data processing system with FPGAs. In the following sections a number of approaches is presented that utilize those architectural alternatives in various ways.

2 FPGAs - A Hardware Introduction

Field-programmable gate arrays are integrated circuits that can be reprogrammed to fit an application's needs and can implement arbitrary digital logic provided that the chip space is big enough. They consist of logic gates that can be configured and combined to construct digital circuits. To that end they also contain input/output circuitry and routing channels to combine more complex digital functions. They can be (re-)programmed using HDLs. The architectural structure of an FPGA enables high throughput rates, low latency and a high degree of parallelism. Flexibility and adaptability of FPGAs are realized in the process of synthesizing an FPGA.

2.1 Architecture of FPGAs

The FPGA on its coarsest architectural level is a two-dimensional array of *configurable logic blocks* (CLBs, sometimes also called Logic Array Blocks or logic islands) and several Input/Output blocks (IOBs). A CLB itself contains several elementary logic units (also called slices or adaptive logic modules (ALM)) and a switch box that is responsible for the connection of each CLB to the FPGAs *interconnect fabric*. Fig. 1 shows a simplified architectural view of such an FPGA.

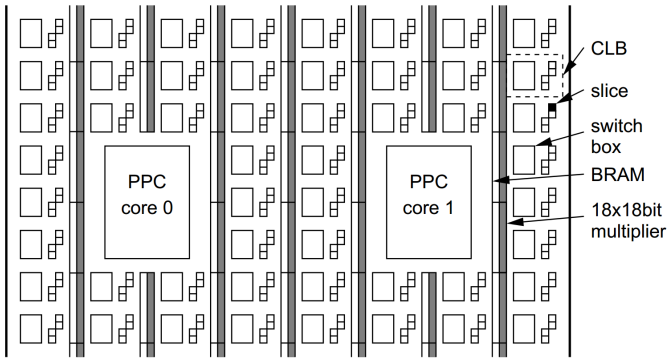


Fig. 1. Simplified architecture of an FPGAs: Array of CLBs, which each consists of 4 slices as well as a switch box [MTA09a].

The interconnect fabric is the routing architecture of the FPGA. It allows communication between arbitrary CLBs by providing bundles of wires that run across the chip. At intersections points there are programmable links that can be configured to route signal in every possible direction.

Each programmable slice consists of a fixed number of lookup tables (LUT) with n inputs and one output, a number of 1-bit registers, arithmetic/carry logic and multiplexers[1]. Figure 2 shows a simplified architectural view of a slice.

A LUT can implement any binary-valued function (e.g. `NAND`, `XOR`, Multiplexing) with n inputs. The number of inputs depends on the manufacturer, but most commonly it is between 4 and 6. In essence, a LUT stores the outcome of an operation for every possible combination of its input vector. Each n-input LUT requires 2^n bits to store the corresponding table. Typically LUTs are realized using static random access memory (SRAM). Main advantage of this configuration is that an LUT can be read asynchronously (i.e. independently from clock signals) in less than a cycle. However writing a 2^n SRAM takes 2^n cycles. Moreover it sacrifices chip space for better read performance since SRAM typically demand more space than dynamical random access memory (DRAM). However, since the SRAM is the main component of an LUT, each LUT can be configured to be used as distributed RAM, to provide more memory space.

An LUT's output can be wired to bypass or incorporate FlipFlops, fast carry paths, feedback to the LUT's input or the switchbox of the surrounding CLB. The FlipFlops serve to cache a signal, to be able to process it in the next clock cycle thereby facilitating pipeline designs. Fast dedicated lines (carry paths) are used between neighboring LUTs. Most commmonly these are used as communication channels called carry chains. Carry chains allow the combination of several LUTs to implement complexer arithmetic functions. These carry chains can also contain

[1] Device that selects one of several inputs and forwards it to a selected output channel

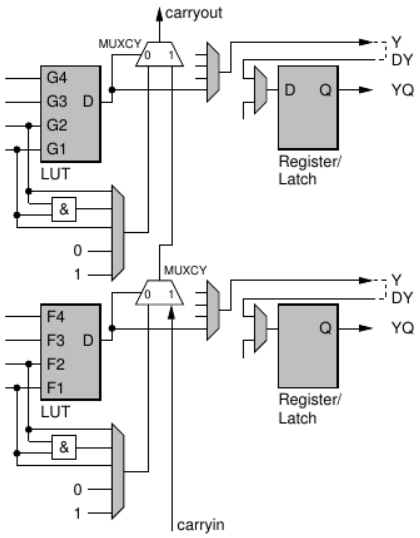


Fig. 2. Simplified architecture of a slice with two LUTs and two FlipFlops. Gray components are programmable [MTA09a].

separate carry logic that helps to implement the building of several arithmetic functions.

An IOB is located at the periphery of an FPGA and is connected to the same interconnect fabric as the CLBs. Therefore the output and input of every CLB can be routed to and from every IOB on the chip. Typically an FPGA supports many I/O standards, classifiable into single-ended (e.g. PCI) and differential IO (e.g. PCI-Express). With a large amount of IOBs, an FPGA can not only support a number of different communication protocols but also very fast communication. High-speed I/O can be supported using fast serial transceivers [2]. As of today the world's fastest transceivers can handle up to 32 Gb/s [Fuj13]. FPGAs usually can contain multiple of such transceivers, so that aggregate bandwidths of terabit per second can be achieved (compare to Serial ATA Revision 3.0's bandwidth of 6 Gb/s).

Commonly FPGAs are often shipped with other auxiliary components ranging from dedicated block RAM (BRAM) structures over multipliers and adders up to full-fledged CPU cores and Ethernet controllers. Figure 1 shows BRAM elements and multipliers between the columns of CLBs and two PowerPC Cores (PPC) as auxiliary elements.

BRAM elements usually can hold a few kilobytes of data and usual FPGAs hold a few hundred of those blocks. Accessing a BRAM block i.e. reading or writing can be done in single clock. Usually an FPGA holds several hundred of those BRAM elements which can all be accessed in parallel. Sometimes they come in dual-ported form i.e. the memory can be accessed from two sides concurrently. BRAMs can be used to store larger data sets than distributed RAM since each block can be combined with another one. On the one hand BRAM units offer advantages in clock domain crossing (i.e. posing as a buffer), but are also prerequisite to leverage FPGAs against the von-Neumann bottleneck or the memory wall.

Multipliers and adders are often added to an FPGA since for a long time FPGA's main application area was digital signal processing (DSP). Exemplary applications of DSP like the Fourier Analysis rely heavily on mathematical operations and therefore many manufacturers include dedicated components in their FPGA that allow

[2] Device consisting of transmitter and receiver

faster multiplication and addition operations. While an FPGA can implement multipliers and adders itself using its, CLBs a hard-wired programmed logic controller is better in performance and space utilization. In terms of data processing, dedicated multipliers become especially useful once databases need to execute hash functions e.g. in a hash join.

Full-fledged CPU cores on the FPGA allow to direct more complex tasks to those auxiliary units. The realization of complex tasks can occupy substantial amounts of CLBs on the FPGA that might be needed to address other operations. To save chip space and reconfiguration time it can be beneficial to give up a little parallelism in order to realise the fastest computation possible.

Separate Ethernet controllers allow the FPGA to circumvent the von Neumann bottleneck. For instance, an application that monitors network activities e.g. high-frequence trading applications, doesn't need to take a detour via system bus and main memory. Using the Ethernet controller it can be directly plugged into the network.

Ultimately the fine-grained configurability of an FPGA allows a truely parallelized execution of operations, high I/O performance and the integration advantages discussed. The architecture of separate LUTs, CLBs or other subareas that are able to communicate with other CLBs and IOBs over fast carry paths and the interconnect fabric enables the user to program independent processing units. These processing units do not share any resource and therefore can execute operations independently from each other. With a number of IOBs the FPGA is able to provide input and output to various independent processing units concurrently, while enabling a better memory bandwidth utilization or even circumvention of system bus and memory.

Furthermore it is noteworthy that FPGAs have significantly less power consumption than traditional GPPs. With the given hardware parallelism it is typically sufficient to solve a given problem with lower clock frequencies resulting in an overall lower power consumption.

Table 1. Specifications of Xilinx's XC5VFX200T

	Virtex-7
Logic Cells	30.720
BRAM	16.416 Kb
Total Transceiver Bandwith	156 Gb/s
Hard Cores	2
DSP Slices (Multipliers, Accumulators, Adders, Subtractors)	384

Table 1 provides an overview of some statistics of the configuration of an FPGA from Xilinx [Xil09], illustrating the size of common FPGAs.

2.2 Programming an FPGA

FPGAs are programmed using hardware description languages (HDL) like Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL). Every VHDL design describes at least one entity/architecture pair or an entity and multiple architectures. In the entity section I/O ports of the IC are defined. The architecture part describes the behavior of an entity. Figure 3 shows an exemplary implementation of an AND-Gate in VHDL.

```
entity AND_ent is
port( x: in std_logic;
      y: in std_logic;
      F: out std_logic
);
end AND_ent;

architecture behav of AND_ent is
begin
  process(x, y)
  begin
       if ((x='1') and (y='1')) then
            F <= '1';
       else
            F <= '0';
begin
       end if;
  end process;
end behav;
```

Fig. 3. Example VHDL Definition of an 'AND'-Operator

Once HDL specifications are defined they are fed as input to a synthesizer. The synthesizer turns HDL specifications into gate-level netlists[3]. These netlists are then input to a translator, that provides a global single netlist that comprises all behavioral aspects of the gate-level netlists and the constraints[4]. The elements of the global netlist are mapped to physical FPGA elements such as CLBs, IOBs or BRAMs. Next step in FPGA design is the the map, place and route processes, which are also the most cost-intensive. The mapping process fits the design onto available resources on the chip, whereas place and route processes map the design to timing constraints. After elements of the global netlist are mapped to physical elements, these elements need to be distributed on the concrete FPGA and then interconnected. Depending on how many timing constraints exist, this task can become the most time-consuming. Once the placement and routing procedure is finished, a bitstream is created. This bitstream is loaded onto the FPGA's configuration SRAM. Some manufacturers provide SRAM that is divided into *frames*, each of which corresponds to a physical site on the FPGA. This technique is the basis of *dynamic partial configuration*, that allows parts of the FPGA to be reprogrammed without interrupting other running parts of the FPGA. In DPC only the frames of a particular *partial reconfiguration region* are updated. Using this process, specialized modules can be loaded during runtime with the result that they do not occupy chip space when they are not needed. However these PRRs need to be configured beforehand, i.e. during the design phase.

3 Data Processing using FPGAs

Before any data processing operation on FPGA can be executed it has to be integrated into a common architecture. There are various different alternatives for such an architectural infrastructure and each of them influences the way actual data processing can be performed. On a high abstraction data processing on FPGAs can be distinguished into two categories, Stream Processing and Co-Processing.

3.1 Architectural integration of FPGAs

In order to employ an FPGA as an accelerator in a database it should be connected to the conventional setup i.e. memory, storage and CPU. In terms of physical placement FPGAs offer three strategies as shown in Fig. 4.

[3] Describes connectivity informations in electronic design
[4] Assignment of special physical FPGA elements or timing specifications

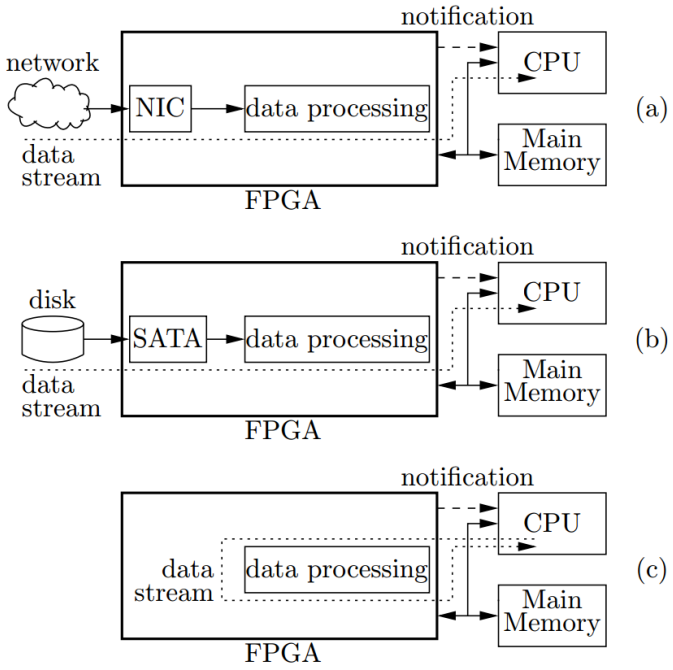


Fig. 4. Different architectural options for placing an FPGA: (a) in the data path between network and CPU (b) between a disk and the CPU (c) as a co-processor [MT09].

To utilize the low-latency of FPGAs, it can be placed between the Network Interface (NIC) and the CPU (see Fig. 4(a)). In this configuration an FPGA is responsible for filtering or aggregating data that comes in over the network such that the CPU has a lower workload and smaller data sets to handle. A related approach by *Netronome* [Net13] uses programmable network flow processors to execute Regular-Expression Matching. These network processors however are designed for a smaller set of applications and thus lack the flexibility of an FPGA.

A similar configuration is shown in Fig. 4 (b). Here the FPGA is placed between the storage unit of an DBMS and the processing unit respectively the memory. This would be the typical use case for a data warehouse. The FPGA again would be able to perform operations like filter, aggregation or even join operations before the data reaches the CPU thus reducing the CPU's workload and increasing overall throughput. IBM's *Netezza* Appliance [Fra11] is a industrial-level product, that implements this architecture. The *FAST* engine ("FPGA-accelerated streaming technology") performs a fixed set of operations[5] on data before it reaches the CPU.

Finally Fig. 4 (c) shows the most easily integrable approach. The FPGA poses as a co-processor to the CPU which can offload tasks on-demand to the accelerator. Communication between CPU and FPGA must be realized through interrupts and shared memory. This approach's problems arise from this communication pattern, since not only does it not circumvent the von-Neumann-bottleneck but can actively increase its severity.

3.2 Network Stream Processing on FPGAs

Since streaming applications readily match the strengths of programmable hardware, big parts of the data processing research community initially concentrated on

[5] Concrete operations are decompression, projection and selection.

these. In these scenarios data arrives in a continuous flow and needs to be processed in real time. This flow oriented paradigm can leverage the specific I/O capabilities of FPGAs along with its low latency and inherent parallelization such that FPGA-based stream processing does not suffer from the von Neumann or memory bottlenecks.

Following the architectural pattern of Fig. 4 (a) and (b) we now take a look at using an FPGA as stream processor. An application for such an architecture are network intrusion detection systems. To detect suspicious patterns in a network's data flow, such systems perform regular expression (RegExp) matching. RegExp is a term coined by theoretical computer science and formal language theory. It describes a sequence characters[6] which can be used in string matching. Network instrusion detection is not the only relevant application area for RegExp matching. It can also be used in semantically more rich stream processing as for example a filtering mechanism.

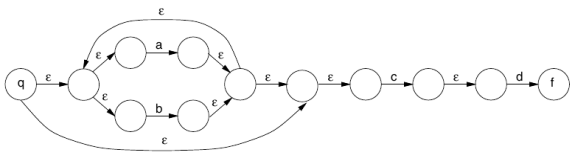


Fig. 5. Non-deterministic finite state automata for the expression $((a|b)*)(cd)$ [SP01]. ε is an empty string, q the initial state and f the final state. a,b,c are literals and $*,|$ are metacharacters.

Given a particular RegExp it's possible to construct a finite-state automaton (FA) from it. As an example Figure 5 shows a non-deterministic FA for the RegExp $((a|b)*)(cd)$. FAs are models that illustrate processes as a series of states and transitions. In this case, the string matching process, edges represent transitions and determine the necessary next character to switch to the next node(state) for each state . Non-deterministic FAs, like the one shown in Fig. 5 are considered inefficient on GPPs since a software implementation has to consider all candidate states and transitions iteratively. For this reason software RegExp implementations turn non-deterministic representations into deterministic FAs. In an FPGA every logical resource operates independently. Thus every candidate state and every transition can be considered in parallel.

Based on the One-Hot-Encoding Schema, Sidu et al. [SP01] implement a non-deterministic FA by using flip-flops that represent the state of FA (i.e. a flip-flop holds a 1 if its active). This technique, originally developed for deterministic FAs, can be extended to be used in a non-deterministic context. In contrast to a deterministic implementation, the output of a flip-flop in a non-deterministic must be routed to more than one following input requiring multiplexers. Furthermore can non-deterministic FAs contain ε. On FPGAs the implementation of ε edges can be done by simply wiring the output of a flip-flop to the input flip-flop directly. Sidu et al. further observed that if all incoming edges of a state are ε edges the state can be omitted completely. With a reduced automaton at hand, the hardware circuit construction can be done in two steps:

– For each state of the FA a flip-flop needs to be instantiated.

[6] Regular characters with their literal meaning and metacharacters denoting sequences, repetitions etc.

– For each transition of the FA the combination logic that forwards an active bit from flip-flop to flip-flop needs to be instantiated, such that a condition is satisfied. If a state has multiple incoming edges those edges are to be combined with a logical `OR` operator.

Fig. 6 shows the hardware implementation of the non-deterministic FA shown in Fig. 5. The authors propose an algorithm that extracts simple logic structures out of a RegExp, matches them to their hardware implementation and constructs the IC by aligning the simple logic structures.

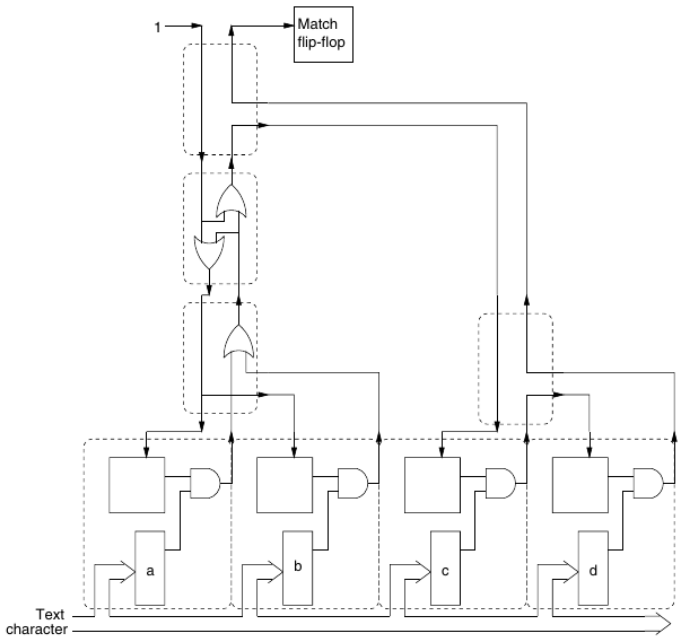


Fig. 6. Hardware implementation of the non-deterministic FA for $((a|b)*)(cd)$ [SP01]. The dashed boxes indicate separate logic structures.

Not only are non-deterministic FAs obviously easier to build (note the structural similiarity between non-deterministic FA and the RegExp in Fig. 5) but they also come more often than not with fewer states. In a hardware implementation these states correspond to precious hardware resources which can be saved using the non-deterministic implementation.

In their experiments Sidhu et al. [SP01] compared finding matches for a RegExp $(a|b) * a(a|b)^8$ in a 2MB file using grep[7] and their FPGA implementation. While grep took between 64.76 ms to 74.76 ms, the FPGA implementation always took 43.44 ms. These tests included an ad-hoc construction of the non-deterministic FA and the FPGA. It is however noteworthy that these times were acquired using a relatively small 800 MHz Pentium III Xeon Processor.

[7] **G**lobally search a **r**egular **e**xpression and **p**rint; a command-line utility of unixoid systems

3.3 Data Stream Processing with FPGAs

The architectural integration presented in Fig. 4 (a) and (b) suggests that FPGAs can also be used in broader application areas than than just network intrusion detection. Moving to such a broader application the goal of the *Glacier* system [MTA09b,MTA10] is to compile queries from the continuous query language (CQL)[8] to VHDL instructions that synthesize an FPGA hardware circuit. When loaded on an FPGA it implements a given query in hardware and guarantees a specific throughput rate. It consists of a component library and a compiler. The component library stores building blocks for each stream processing operator (see Table 2). The compiler takes a given query, instatiates the required modules and describes the connection between those modules such that they can be translated into an FPGA configuration.

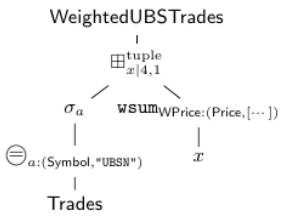
Table 2. Stream Processing Operations adapted from [MTA10]. *a,b,c* are field names, q,q_i are sub plans and x is a parameterized sub plan input

Operator	Semantics
$\pi_{a_1,\ldots,a_n}(q)$	Projections
$\sigma_a(q)$	Selection where a holds true
$\circledast_{a:(b_1,b_2)}(q)$	Arithmetic or boolean operation
$q_1 \cup q_2$	Union
$agg_{b:a}(q)$	Aggregation
$q_1 grp_{x\|c} q2(x)$	Group operation
$q_1 \boxplus^t_{x\|k,l}\ q_2(x)$	Sliding window
$q_1 \propto q_2$	Concatenation; "Join by position"

Glacier operates on the algebra shown in Tab. 2. This algebra assumes tuple structured input events. Each operator can be nested. While it does not represent the complete relational algebra, it supports those aspects of a query that can be realized in a pre-processing step in the sense of the architecture presented in Fig. 4 (a) or (b). Like almost all query processors do, *Glacier* initially turns CQL statements into an internal algebraic plan. Fig. 7 (a) shows an examplary CQL query including a sliding window, aggregation and selection operation. Fig. 7 (b) shows the corresponding algebraic plan.

```
SELECT wsum(Price, [.5,.25,.125,.125]) AS Wprice
FROM (SELECT * FROM Trades
      WHERE SYMBOL = "UBSN")
      [SIZE 4 ADVANCE 1 TUPLES]
INTO WeightedUBSTrades
```

(a) Example query



(b) Algebraic Plan

Fig. 7. Example query and corresponding algebraic plan for a financial trading application [MTA10]. Assuming there is an aggregation function `wsum` the query computes weighted sums of the prices in the last 4 trades of UBS stocks.

[8] An extension of SQL to support DSMS

This algebraic plan can now be fed to the *Glacier* compiler which derives VHDL expressions for every operator in the plan. It is able to do so by assuming that for every algebra expression the wiring interface follows a certain schema. An n-bit wide tuple is represented in hardware by n parallel wires plus an additional `data_valid` that indicates if the signals on the wires should be regarded. To measure the performance of individual operators Mueller et al. [MTA09b] use the units *latency* and *issue rate*. While latency describes the number of clock cycles a circuit needs to produce an output, the issue rate describes the number of tuples that can be processed per clock cycle. Naturally the latter is always ≤ 1.

The definition of the wiring interface makes defining selection and projection operations straight forward. If the signal on the wires does not correspond to the selection condition, the `data_valid` line is set to false thus invalidating the tuple for following operations. Projection operations are done by "cutting" the non-relevant bit data of the wiring interface i.e. not forwarding their signals to any subsequent module. Mueller et al. observed an interesting side effect of this implementation. The interaction between *Glacier* and the compilation stages of an FPGA implements basically a *projection pushdown* for free. For many reasons synthesizers of FPGAs optimize the layout such that there are no "dangling" wires. By leaving non-relevant data bits on such "dangling" wires *Glacier* leaves the actual pushdown to the synthesizer. In the synthesizing process all sub-circuits whose output is not connected to a port are eliminated. Both hardware implementations of these operators have latency and issue rate of one. Arithmetic and boolean operations that are preliminary to selection operations are directly translated into logic operations. Most simple arithmetic or boolean functions produce an output within a single clock cycle. However in case they don't (e.g. multiplication, floating-point arithmetic) the increased latency has to be countered by synchronization methods.

Glacier introduces two types of synchronization methods. Traditional `FIFO` queues are short-term buffers to handle streams that have varying data rates. They buffer data and emit it at a predictable rate, often the same as the issue rate of upstream sub-circuits. Additionally *delay operators* can block data items for a fixed number of clock cycles. For the aforementioned complex arithmetic operations this type of synchronization is used.

Applications scenario for FIFO queues are `union` operators. To ensure the right synchronization, the union operator implementation uses FIFO queues. These are either implemented using flip-flop registers or BRAM. Since the union operation itself has a latency of one and the FIFO queues add another latency cycle the overall latency of the union operator is two.

To bridge the gap between stream and relational-style data processing CQL provides a `windowing` operator. The operation uses the output of its left-hand subplan and slices it into a set of windows. Subsequently it invokes the right-hand sub-plan in parameterized fashion such that the operations in this sub-plan are applied to all tuples in the current window. The compiler of *Glacier* wraps the right-hand side sub-plans into a template circuit. Additional `eos` signals are introduced to notify a sub-plan of the end of a window-stream. Common use case of this operation are *sliding windows*. The parallelism of FPGAs can be used here. The right-hand subplan is replicated as many times as there may be windows open concurrently (in the definition of Tab. 2 n $= k/l$) plus one. A cyclic shift register CSR1 is used to keep track of open windows. Whenever the end of a window is reached an `adv` signal shifts the register to the right. This closes the oldest window and opens a new one. In parallel to shifting the CSR1 the `eos` signal is send to the sub-plan that processes the oldest window. By this the sub-plan starts to generate output, that is fed to a union operation. To submit the signal to the correct (i.e. oldest) window a second cyclic shift register CSR2 is maintained. This is synchronized with the first one and keeps a bit that identifies said window.

Compared to the previously explained operations, `aggregation` operations assume a finite set of input data. Therefore in DSMS aggregation operations are typically applied to windows. The windowing operation however just streams in tuples. The storage implementation therefore has to be applied by the aggregation operation itself. Since this allows the optimizer to build storage in just the right size for the function this concept is an advantage.

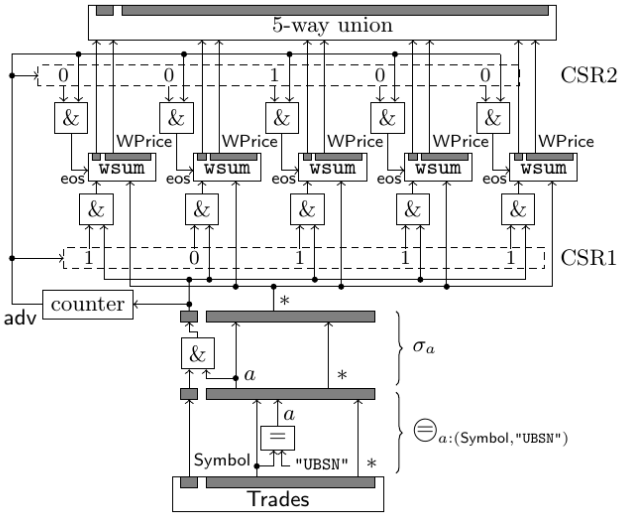


Fig. 8. Hardware implementation of the query presented in Fig. 7 [MTA09b].

Fig. 8 presents a hardware implementation generated by *Glacier* for the examplary query in Fig. 7. The lower parts of the figure present the implementation of projection, selection and arithmetic function of the left-hand side subplan in the algebraic plan. The upper parts represent the windowing clause `SIZE 4 ADVANCE 1 TUPLES`. One can observe that the maximum window size is 4 and 5 copies of the `wsum` sub-plan have been initialized. The counter is responsible for counting the number of tuples and sends the `adv` signal if it reaches the number specified in the `ADVANCE` part of the windowing clause.

Following the architectural pattern presented in Fig. 4 Mueller et al. implemented a soft-core module on the FPGA that is responsible for decoding of UDP datagrams. Result data is written into a FIFO accessible by the CPU via a CPU Adapter. Whenever data is ready, the FPGA raises an interrupt and applications are able to pick up the data. In general Mueller et al. define such (de-)serialization modules under the term *glue logic*. Since every FPGA-based query processor has to assume that the data to consume comes in a specific format, every hardware implementation of data processing needs such logic. This *glue logic* not only requires some chip space but also introduces additional latency. However if build right it should be able to cooperate with a hardware plan in a pipelining fashion.

Alongside the basic layout and functional implementation of a query-to-hardware compiler Mueller et al. also developed several optimization heurisitics. With the observation that not all components are clock bound, the authors integrated an asynchronous execution of sub-plans. By eliminating intermediate registers they reduced the latency and saved a small amount of FPGA resources. Additionally this approach allows a better task parallelism. Fig. 9 illustrates this idea with aid of

two boolean operations in parallel as well as the corresponding selection operation all in one cycle.

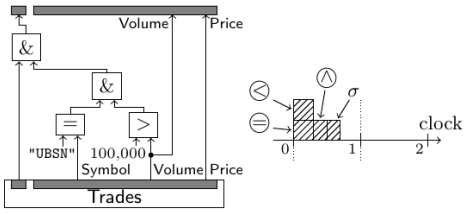


Fig. 9. Reduced latency by asynchronous operations with task parallelism [MTA09b].

3.4 FPGAs as a Co-Processor

The previous chapters of this work concentrated on stream processing applications since those tasks are the most natural way to utilize the hardware structure of FPGAs. However in an OLTP environment the architectural structure of streaming applications interferes with SLA operations in a way that might make this architecture unsuitable. As mentioned earlier not only do the actual operations introduce more or less latency (depending on the complexity of the query) but also serialization and deserialization of the data must be processed. Hence a writethrough protocol in the I/O path of a traditional OLTP system can increase Input/Output operations per second (IOPS). An increased IOPS measure can delay operations of the SLA to unacceptable levels thus rendering the stream processing approaches unsuitable in such systems.

Concurrently FPGAs pose an interesting alternative to support OLAP operations in such a system. Commonly systems that support analytical operations build indexes to speed up those operations. Indexes that merely serve this purpose, however, can have a detrimental effect on OLTP operations. Every insert, update or delete operation can entail index updates which results in heavier CPU and I/O load thus impacting overall throughput. The FPGA as a mechanism to speed up costly analytics operations could make these indexes obsolete.

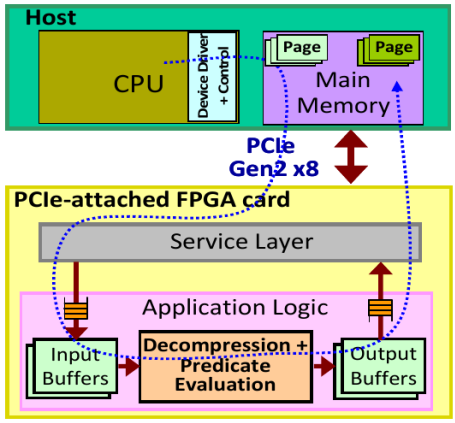


Fig. 10. System architecture with FPGA as a Co-processor [SMT+12].

To this end Sukhwani et al. and Halstead et al. [SMT+12, HSM+13] employ the architecture shown in Fig. 4 (c) to use FPGAs as processing accelerators. As shown in Fig. 10 the FPGA is connected to the CPU and memory via PCIe (PCI Express). Data access for the FPGA is gained via DMA. Thus FPGA can only work on data that is already in memory. The CPU communicates with the FPGA using a job queue, device drivers and service layer logic. The service layer logic is one of two modules of the FPGA. It provides all logic responsible for DMA, PCIe and job management. The application layer implements the functions required to process queries from the DBMS. Between the two there is a set of well-defined interfaces that include buses for I/O communication, queues for DMA requests and control signals.

In their first attempt to utilize this architecture Sukhwani et al. [SMT+12] implemented predicate evaluation and row decompression on FPGAs for row-based DBMSs. Therefore the authors designed an architecture inside the application layer shown in Fig. 11. A scan tile is the central element that enables the exploitation of the parallelism on FPGAs. It contains a number of decompressors and row scanners as well as buffers for input and output database pages. It also includes logic to extract and write single rows within such a page. Decompression tasks are executed through lookups in a dictionary which is shared by two decompression units. The row scanner houses a number of predicate evaluation units (PE). The whole scan tile can scan one database page at a time. Parallelism can be achieved by simply replicating scan tiles on the chip.

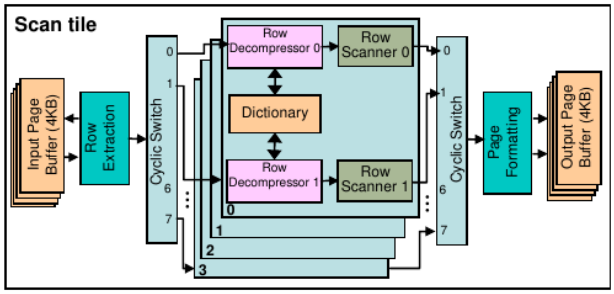


Fig. 11. Row Processing architecture [SMT+12].

To utilize the full bandwidth of 8 lanes of PCIe 2.0 (i.e. 4000 MB/s) two scan tiles are the minimal configuration to process uncompressed database pages. With rising compression coefficients more scan tiles are needed to utilize this bandwidth. The configuration presented allows to trade page parallel computation for query complexity. A scan tile can be configured with differing amounts of row scanners and decompressors thus allowing to utilize the FPGA's space in the best way following the given query.

Suhkhwani et al. observed, that performing decompression on FPGAs provides numerous benefits. Database pages can be sent directly to the FPGA without any need to prefilter or process on host architecture. Further an efficient FPGA implementation can utilize the inherent parallelism such that the decompression algorithm becomes much more efficient. As shown in Fig. 11 the decompression algorithm relies on a dictionary. Basically each compressed row consist of a set of tokens, which get looked up in the dictionary and stitched together. A tokenizer module fetches compressed rows from a compressed row buffer, one token at a time controlled by a controller FSM. This token is fed to a a character decoder or a dictionary

data decoder that decodes the character-type or dictionary-type tokens and write them into the uncompressed row buffer. This algorithm is not purely feed-forward since no new token can be fetched until the previous one is completely decompressed. To enable pipeline-parallelism the authors implemented a FIFO queue in the tokenizer that prefetches 8 tokens, such that a new token is ready for processing as soon as the current one is finished.

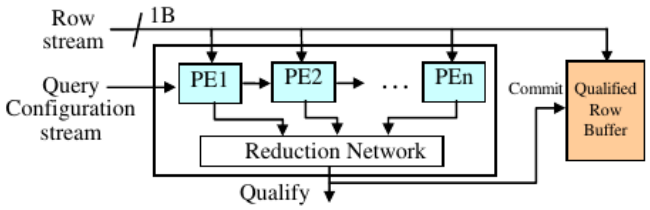


Fig. 12. Logic of the Row Scanner to evaluate predictates on database rows [SMT+12].

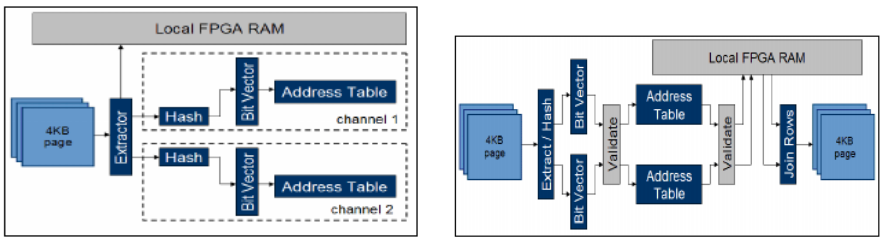
After the decompression of rows, the data is sent to predicate evaluation logic. The corresponding row scanner is shown in Fig. 12. It consists of a series of PE units that each evaluate a single predicate on a column of the database row. A reduction network reduces the result of the evaluation to a 1-bit qualify signal. To address the issue of high synthesis and place-and-route time consumption the authors implemented a hardware image that can be fitted to various queries. The PEs are designed to perform 6 inequality operations or to be excluded from the current query. The reduction network is a binary tree of reduction units each of which is a 2 to 1 reducer performing logical `AND`,`OR`, `NOT` and `PASS` operations. In an initial query load phase the PE is configured in 5 dimensions:

1. Enabled/Disabled
2. Setting predicate values
3. Definition of the inequality operation
4. Determination of the offset of the first byte of the desired field within the row
5. Determination of the field

In the scan phase, rows of the database are streamed over the PEs in a rate of one byte per cycle. Each byte is broadcasted to all PEs and speculatively written into the qualified row buffer. If at the end of the stream of a row the row is qualified, it is written to the page formating logic (see Fig. 10).

In further investigations in the field Halstead et al. [HSM+13] utilized the same architecture to design and prototype an FPGA join operation. The relational join is one of the most CPU intensive tasks of a DBMS, yet it needs to be used commonly. The proposed architecture concentrates on FPGA Hash Joins where columns of table are initially hashed in a build phase. In the following probe phase hashed values of both tables are compared, through which a combination of fields from two tables can be determined. As running example the architecture of many data warehouses is used, where in a star-schema scenario a fact table holds the facts, whereas the smaller dimension tables contain attributes.

This logical scheme is employed in hardware as shown in Fig. 13. In the build phase (Fig. 13 (a)) the FPGA fetches the pages from a dimension table onto local FPGA storage units in order to perform the row joining and output materialization on FPGA. During this extraction the join attributes are extracted and their values are hashed into a hash table (bit vector). The hashed value, the actual value and the position of the row in local storage are stored in the address table. The collision

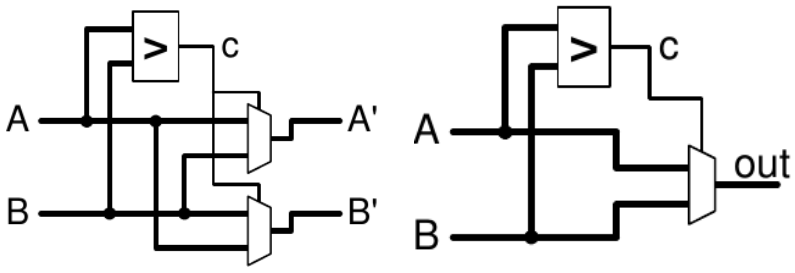


(a) Build Phase (b) Probe Phase

Fig. 13. Logical Scheme of Build and Probe Phase Hardware of a hash join on FPGAs [HSM+13].

of hash values is handled with a chaining scheme whereby the hash table additionally contains `next` and `tail` values. The parallelism of the FPGA allows to handle multiple table joins in a data parallel way. Each channel in Fig. 13 (a) has its own dedicated resources such that hashing and data table generation can be done concurrently. The probe phase (Fig.13 (b)) begins with the streaming and concurrent hashing of relevant columns of the fact table. The hash values are compared with the values from the build phase. If matches are found the row is forwarded to the address table for further processing. There the value of a qualified row of the fact table is again checked against the stored value of the dimension table. With this technique the fact table is simply streamed through the FPGA such that it doesn't need to be stored anywhere on the FPGA. By that the join operator can handle fact tables of arbitrary sizes.

Another join algorithm for relational DBMSs is the sort-merge join. The central concept of this join algorithm is to initially sort relations of a table, such that an interleaved scan of both sets encounters equal values of the join columns at the same time. In practice the initial sorting of the tuples is the most expensive part. Thus the concept of external sort operations were introduced. FPGAs pose an especially interesting alternative to the sorting problem since they can utilize sorting networks or FIFO Merge sorters. In [KT11,MTA12] several approaches to sorting on FPGAs using such techniques were investigated and introduced. A sorting network is a mathematical model of a network that sorts a set of input values through compare-swap elements. It is designed to exploit a high degree of parallelism since each compare-swap operates independently from the others. A FIFO merge sorter uses select-value elements where only one element (the larger resp. smaller one) is selected and forwarded.

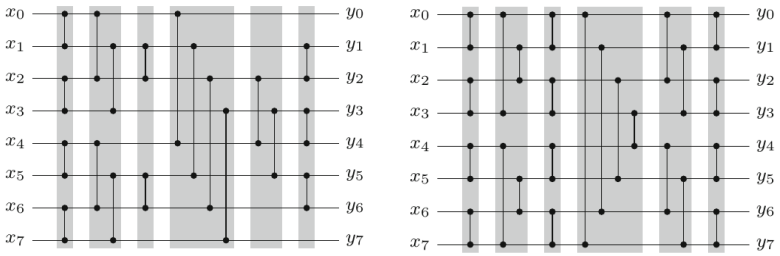


(a) Compare Swap Element (b) Select Value Element

Fig. 14. Circuits diagrams of logical elements for building sorting operations on FPGAs [KT11].

A compare-swap element compares two input values and, if required, swaps the value at the output. Fig. 14 (a) shows such an compare-swap element. It can be built using comparators and multiplexers. Depending on the availability of carry-chain logic on the FPGA comparators for more then 6-bit wide operands can be built in a tree-like structure or using carry-chain logic, whereas carry-chain logic yields the better latency rates. The comparator's one bit output controls the output of a multiplexer which is responsible for routing the correct (smaller or larger) signal on the correct channel. It is noteworthy that the implementation of the multiplexers can take up to 80% of the logic cost of implementing a compare-swap element.

Similarly a select-value element (see Fig. 14 (b)) compares two input values, but instead of swapping the signals it selects the larger (or smaller depending on the sorting task) value and forwards it. The construction of comparators underlies the same limitations as mentioned before. Typically the select-value element is designed in a way so that the not selected value can be kept for comparisons in the next cycle.



(a) Even-odd sorting network (b) Bitonic sorting network

Fig. 15. Different implementations of sorting networks for 8 elements [MTA12].

The idea of sorting networks on FPGAs is to stream unsorted data through a number of compare-swap elements such that at the end of the sorting network the signal paths contain a sorted order of the data. They do not require any control instructions and their simple data flow pattern makes them easy to parallelize. There are several ways to implement such a sorting network. The most popular arrangements are even-odd, bitonic sorting networks (see Fig. 15) and sorting networks based on bubble sort. The circuit of a sorting network consists of horizontal wires and vertical comparator elements. Each wire can hold an m-bit number. Each way has different effects on resource consumption and ease of implementation.

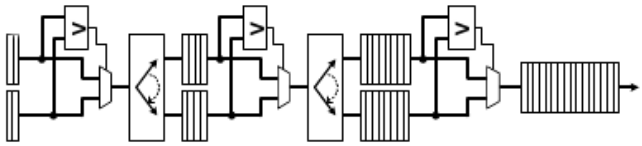


Fig. 16. Cascading structure of a FIFO-based merge sorter [KT11].

Another approach to the sorting problem on FPGAs is a BRAM-based FIFO merge sorter as proposed by Koch et al. [KT11]. This approach is able to handle larger sorting problem sizes inside the FPGA than a sorting network would. At the heart of the FIFO merge sorter a select-value element is utilized to merge to sorted runs into a larger sorted run. Inputs of a select-value element are stored

in FIFO structures that hold the sorted runs. During execution the smaller (resp. bigger) value is selected and written to an output FIFO. In the following cycle the unselected value and another value from the FIFOs are compared and so on. A cascade of such merge sorter can produce larger and larger sorted runs. Fig. 16 shows an examplary implementation of such a merge sorter.

4 Conclusions

In this report we've examined the structure of FPGAs and the usability and advantages of data processing on such chips. The proposals by the research community range from network oriented solutions over data stream processing engines to co-processor units that can be plugged into any existing system provided a software infrastructure is implemented.

At first we discussed the specific structure of FPGA. We showcased the different architectural layers and modules of the FPGA to illustrate how the FPGA can provide a high degree of parallelism, low latency and high throughput rates. Also the programmability of FPGAs was discussed to illustrate the flexibility and adaptability an FPGA can provide.

A network stream processing solution was presented in form of the RegExp matching using FPGAs. The high degree of parallelism allows the implementation of non-deterministic FAs on FPGAs. Those non-deterministic FAs are not only easier to build and more resource efficient, but do also offer better performance values in direct comparison to a software implementation of RegExp matching. Furthermore the proposed strategy is a good example of how specialized hardware can bypass the von-Neumann bottleneck and the memory wall. By putting the RegExp matching in the data path, the process (or for that matter any other process running on GPPs in the system) does not suffer the effects of these problems.

A broader application area in terms of stream processing is covered by the *Glacier* system. It was introduced as full-blown compiler for the data stream processing language CQL that provides a variety of operations for stream processing. Once more this approach circumvents the von-Neumann bottleneck and memory wall by being applied directly in the data path thus eliminating any intra-host communication efforts except the provisioning of result data.

Finally, several operator implementations were discussed that utilize the FPGA as a co-processor. Approaches were introduced that use the FPGA to decompress and project data in traditional DBMSs as well as approaches that utilize the FPGA to implement hash join and selection operations. In context of sort-merge joins some approaches on sorting on FPGAs were illustrated. Every one of these approaches was introduced with the goal in mind to enable a OLTP platform to perform OLAP operations without endangering any SLAs. The proposed architecture relies on DMA to be able to bypass von-Neumann bottleneck and memory wall at least in terms of data flow. Considering control flow instructions the FPGA needs to be connected to the same bus system as memory and CPU. If such DMA controllers exist, the FPGA can access memory addresses and initiate read/write cycles without interfering with the GPP's tasks. However this architecture presumes an in-memory database scheme. As soon as database pages have to be picked from a hard drive, the CPU is also involved in data provisioning.

Each of the approaches utilizes the high degree of parallelism available on FPGAs in a different way. For instance the approaches made by Sukhwani et al. or Halstead et al. realize efficient analytical processing on FPGAs using a data parallel approach, whereas the *Glacier* system tends to realize more of a pipeline parallel approach. Most of the comparable approaches document similar if not better performance results for their proposed architectures compared to architectures that solely rely

on GPPs. Even further, since FPGAs do have lower clock rates than traditional GPPs, they tend to considerably reduce overall power consumption of analytical systems.

Up until this point however ASICs could provide the same characteristics while concurrently offering even lower power consumptions, lesser transistors for specific tasks and a higher clock speed. Main advantage of the FPGA compared to those kinds of ICs is their flexibility, scalability and overall higher adaptability. FPGAs offer the possibility to be modified and reconfigured to specific query loads at a fine granular level. This circuit re-compilation time however is a CPU-intensive operation. Several approaches cover these problems for instance through usage of pre-compiled modules and partial reconfiguration. For example Dennl et al. [DZT12] defined such modules for the relational algebra. Others utilize the idea of parameterized circuits, where operators are modelled in such a way that they can be fed parameters to initialize them. *Netezza* [Fra11] uses this concept to initialize projection and selection operations during runtime. Also Teubner et al. [TWN13] used this concept to cover a large subset of *XPath*, a query language for XML data. The definition of the language itself allows the definition of so-called skeleton automata, that can be parameterized to run XPath queries as hardware non-finite state automata.

Further work in the field even includes high-level synthesis approaches such as LiquidMetal [HHBR08] or Kiwi [SG08] both of which are compilation approaches that can turn Java, respectively C# programs, into hardware implemented programs on FPGAs.

References

AKN12. Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012. URL: `http://dx.doi.org/10.14778/2336664.2336678`, `doi:10.14778/2336664.2336678`.

BP05. Zachary K. Baker and Viktor K. Prasanna. Efficient hardware data mining with the apriori algorithm on fpgas. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '05, pages 3–12, Washington, DC, USA, 2005. IEEE Computer Society. URL: `http://dx.doi.org/10.1109/FCCM.2005.31`, `doi:10.1109/FCCM.2005.31`.

CNL+08. Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *Proc. VLDB Endow.*, 1(2):1313–1324, August 2008. URL: `http://dx.doi.org/10.14778/1454159.1454171`, `doi:10.14778/1454159.1454171`.

CO14. Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2554688.2554787`, `doi:10.1145/2554688.2554787`.

DG08. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. URL: `http://doi.acm.org/10.1145/1327452.1327492`, `doi:10.1145/1327452.1327492`.

DZT12. Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '12, pages 45–52, Washington, DC, USA, 2012. IEEE Computer Society. URL: `http://dx.doi.org/10.1109/FCCM.2012.18`, `doi:10.1109/FCCM.2012.18`.

Fra11. Phil Francisco. The netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011.

Fuj13. Fujitsu Laboratories Ltd. Fujitsu achieves world's fastest transceivers of 32 gbps for inter-processor data communications. http://www.fujitsu.com/global/about/resources/news/press-releases/2013/0218-01.html, 2013. Accessed: July 7, 2014.

GGKM06. Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1142473.1142511, doi:10.1145/1142473.1142511.

HHBR08. Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_5, doi:10.1007/978-3-540-70592-5_5.

HSM+13. R.J. Halstead, B. Sukhwani, Hong Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 17–20, April 2013. doi:10.1109/FCCM.2013.17.

KT11. Dirk Koch and Jim Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1950413.1950427, doi:10.1145/1950413.1950427.

MSNT11. Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis J. Tsotras. Massively parallel xml twig filtering using dynamic programming on fpgas. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 948–959, Washington, DC, USA, 2011. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ICDE.2011.5767899, doi:10.1109/ICDE.2011.5767899.

MT09. Rene Mueller and Jens Teubner. Fpga: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1559845.1559965, doi:10.1145/1559845.1559965.

MTA09a. Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, August 2009. URL: http://dl.acm.org/citation.cfm?id=1687627.1687730.

MTA09b. Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: A query compiler for fpgas. *Proc. VLDB Endow.*, 2(1):229–240, August 2009. URL: http://dl.acm.org/citation.cfm?id=1687627.1687654.

MTA10. Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A query-to-hardware compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1159–1162, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1807167.1807307, doi:10.1145/1807167.1807307.

MTA12. Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB Journal*, 21(1):1–23, 2012. URL: http://dx.doi.org/10.1007/s00778-011-0232-z, doi:10.1007/s00778-011-0232-z.

MVB+. Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Walid Najjar, and Vassilis J. Tsotras. Boosting xml filtering with a scalable fpga-based architecture.

Net13. Netronome. 40gbps regular expression matching for network appliances. http://www.netronome.com/wp-content/upload/2013/12/Netronome-40Gbps-RegEx-Matching-for-Network-Appliances-Whitepaper-4-10.pdf, 2013. Accessed: July 7, 2014.

NSJ13. Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Flexible query processor on fpgas. *Proc. VLDB Endow.*, 6(12):1310–1313, August 2013. URL: http://dl.acm.org/citation.cfm?id=2536274.2536303.

SAB+05. Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005. URL: http://dl.acm.org/citation.cfm?id=1083592.1083658.

SDCV10. Todd C. Scofield, Jeffrey A. Delmerico, Vipin Chaudhary, and Geno Valente. Xtremedata dbx: An fpga-based data warehouse appliance. *Computing in Science and Engineering*, 12(4):66–73, 2010. doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2010.93.

SG08. Satnam Singh and David J. Greaves. Kiwi: Synthesis of fpga circuits from parallel programs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '08, pages 3–12, Washington, DC, USA, 2008. IEEE Computer Society. URL: http://dx.doi.org/10.1109/FCCM.2008.46, doi:10.1109/FCCM.2008.46.

SMT+12. Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 411–420, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2370816.2370874, doi:10.1145/2370816.2370874.

SP01. Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society. URL: http://dx.doi.org/10.1109/FCCM.2001.22, doi:10.1109/FCCM.2001.22.

STM+13. B. Sukhwani, M. Thoennes, Hong Min, P. Dube, B. Brezzo, S. Asaad, and D. Dillenberger. Large payload streaming database sort and projection on fpgas. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*, pages 25–32, Oct 2013. doi:10.1109/SBAC-PAD.2013.21.

TWN13. Jens Teubner, Louis Woods, and Chongling Nie. Xlynx an fpga-based xml filter for hybrid xquery processing. *ACM Trans. Database Syst.*, 38(4):23:1–23:39, December 2013. URL: http://doi.acm.org/10.1145/2536800, doi:10.1145/2536800.

Xil09. Xilinx. Virtex-5 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf, 2009. Accessed: July 7, 2014.