Fast Data Analytics with FPGAs

Louis Woods
(supervised by Prof. Dr. Gustavo Alonso)
Systems Group, Department of Computer Science, ETH Zurich
louis.woods@inf.ethz.ch

***Abstract*—The rapidly increasing amount of data available for real-time analysis (*i.e.*, so-called operational business intelligence) is creating an interesting opportunity for creative approaches to speeding up data processing algorithms. One such approach that is starting to become more common is using hardware accelerators for stream processing. Typically these accelerators are implemented on top of reconfigurable hardware, known as *field-programmable gate arrays* (FPGAs). Though the value of FPGAs for data warehouses is gradually recognized by the database community, their true potential for various business analytic tasks is yet unexplored. In this line of research, we investigate FPGA technology in the context of *extreme* data processing looking for opportunities where FPGAs can be exploited to improve over classical CPU-based architectures. We introduce a framework for FPGA-accelerated (real-time) analytics including a query-to-hardware compiler for *static* complex event detection, an XPath engine for *dynamic* query workloads, and templates for high-speed data mining operators in hardware.**

I. Introduction

With the possibility to store terabytes of data, businesses today face the problem of querying these large amounts of data in a timely fashion. This problem is particularly severe when data needs to be searched in unanticipated ways, *e.g.*, as was the case with the collapse of Lehman Brothers when financial institutions needed to search all their data to figure out in what way they were affected by this event.

Another example, where traditional database technology no longer helps, is *stream processing*. High-volume data streams, *e.g.*, as produced by stock exchanges, have to be analyzed in real time so that stakeholders can react in sufficient time. When high-rate data streams are coming from the network, software stream processors tend to suffer from what is known as the *network-memory-CPU bottleneck* and might end up dropping network packets.[1]

FPGAs are reconfigurable digital logic devices consisting of a plethora of uncommitted elementary hardware components such as *flip-flops* (FFs), *lookup tables* (LUTs) and blocks of *fast on-chip RAM* (BRAM) connected via a configurable interconnect. The wiring between these components—what effectively determines the functionality of the implemented circuit—is not fixed but can be changed by downloading a *configuration bitfile* to the FPGA.

The idea of using special-purpose hardware to accelerate database operations has existed since the seventies [1] but never made its way into mainstream databases. The reason is that these *application-specific integrated circuits* (ASICs) could not keep up with general-purpose CPUs in terms of both cost and performance. With the advent of FPGAs the development cost for custom hardware (even in small quantities) has dropped significantly. At the same time, CPU clock frequencies have stagnated and the most promising method to further increase performance remains parallelization.

[1]For latency reasons, stock exchange data streams often use the low-overhead UDP protocol, which gives no guarantees that packets will arrive.

Due to their highly parallel nature and despite their lower clock frequencies, FPGAs can outperform conventional CPUs on certain data processing tasks. This has led the company Netezza, for instance, to make FPGAs integral components of its data warehouse appliance *TwinFin*. The company is so successful that IBM acquired Netezza for 1.7 billion dollars a few month ago. Nevertheless, FPGAs are very new in the database world and fundamentally different to program than commodity CPUs.

In this work, we demonstrate the benefits of using FPGAs to (pre)process data close to the source in the context of analytics on large-volume datasets. As we analyze the following three distinct use cases, we illustrate different design patterns and techniques for different needs:

(a) *FPGAs for real-time analysis of high-rate data streams.* As mentioned above, commodity systems suffer from the network-memory-CPU bottleneck when high-rate packet streams are processed from the network. We propose to insert the FPGA into the data path as to process the data closer to the source. Our work comprises a query-to-hardware compiler that *statically* translates high-level *complex event queries* into fixed hardware circuits.

(b) *FPGAs for dynamic query workloads.* The aforementioned compiler translates queries into rigid circuits meant to run over a long period of time as *standing queries*. This is valid when the query workload is not exposed to frequent changes. Otherwise, a more *dynamic* approach is required since reconfiguring the FPGA takes a substantial amount of time. Using the example of a dynamic XPath engine, we propose a solution to this problem based on runtime-parameterizable circuits for FPGAs.

(c) *FPGAs for data mining.* Whereas the above two approaches target streaming applications, here the focus is on data mining tasks, where data is scanned from some storage. The idea is to invoke fast data mining operators that run in a co-processor fashion on the FPGA close to the storage. As an example, we explore a circuit for computing the *skyline* [2] of a multi-dimensional data set.

II. Real-Time Analysis of High-Rate Data Streams

Many applications in areas such as finance, network surveillance, supply chain management, or healthcare need to process high-volume event streams in real time [3]. Usually the individual data items (tuples) in those streams only become meaningful when put into context with other items of the same stream. *Complex event processing* (CEP) aims at inferring meaningful higher-level events (complex events) from a sequence of low-level events. Existing complex event detection engines face the problem that the data items of the event stream first need to be brought to the CPU via main memory before the CPU can start processing them. For instance, when events arrive from the network, they are typically wrapped in small UDP packets. Once the packet rate exceeds a certain threshold, CPU-based systems are no longer able to sustain the network load and start dropping UDP packets [4].

A. Compiling Complex Event Patterns to Hardware

We propose to move the complex event detection engine as close as possible to the origin of the event stream (in this case to the network interface) so as to avoid the network-memory-CPU bottleneck. Our contribution is a high-throughput complex event detection system based on regular expression matching in hardware. It can be attached directly to the network interface and operates at gigabit wire speed while guaranteeing constant latencies. We provide a compiler (see [5] for details) to transform complex event patterns expressed in a high-level declarative language into hardware circuits that can be downloaded to the FPGA.

B. Query Language for Complex Event Patterns

We introduce a query language for defining complex event patterns. Our language closely resembles parts of the `MATCH-RECOGNIZE` clause of the current ANSI draft for extending SQL with pattern matching functionality [6]. Nevertheless, since our focus is on Boolean, regular expression-based complex event detection, we have derived a simplified and less verbose version of the language. A sample query in our language is given below:

```
PARTITION BY symbol
PATTERN (A B+)
DEFINE
    A    AS (price > 20 AND volume > 10)
    B    AS (price > 50 AND volume > 10)
```

We use a `PARTITION BY` clause when we want to detect complex events on a sub-stream basis, as in the example above, where we detect the pattern `A B+` for every symbol (e.g. `UBSN` or `ABBN`) separately. The complex event is formulated using regular expression operators over *basic events* that are defined in the query as well, in terms of predicates in a `DEFINE` clause.

C. Evaluation Results

We have built a complex event detection system and evaluated it on a Virtex-5 FPGA[2] from Xilinx. More detailed information on the experiments can be found in [5]. Here we just want to report the most important results. Tuples were transmitted to the FPGA over the network using UDP. Every UDP packet contained a fixed number of 128-bit wide tuples. We saturated a gigabit link with traffic generated from three machines, which we connected to the FPGA via a switch.

In all cases the FPGA could process 100% of the load, which is not surprising as the network traffic is processed directly by dedicated hardware circuits. Using large packets we measured more than seven million tuples processed per second, with small UDP packets (one tuple per packet) we measured nearly 1.5 million tuples processed per second due to the increased network overhead. In both cases these results translate to bandwidth utilization of close to 1Gb/s.

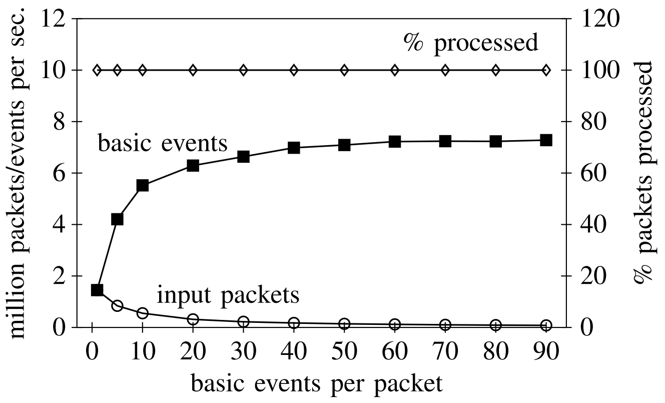

Fig. 1. Performance of FPGA-based complex event detection engine. The FPGA processes 100 % of the input stream independent of packet size and never drops packets.

[2]We used the 700$ XUPV5-LX110T development kit for universities.

III. FPGAs for Dynamic Query Workloads

For many use cases it is valid to reconfigure the FPGA with new queries only every now and then. As *standing queries* they will then run for a longer period of time on some input stream. Nevertheless, applications exist where the time it takes (a few minutes to several hours) to synthesize the circuits and configure the FPGA is not acceptable, *i.e.*, when the query workload changes frequently. We are currently creating the tools to support these kinds of highly dynamic query workloads. As an example, we employ the FPGA as an XML pre-filter for an XQuery backend processor.

A. FPGAs as Data Filters

This use case is different in many respects to the complex event detection example of the previous section. Here, the FPGA is part of a hybrid system and acts as load shedder in front of an XQuery processor running on commodity hardware. Whenever a query is submitted to the XQuery engine a preprocessor extracts filter predicates from the query—so-called *projection paths* as explained shortly—and invokes the FPGA as a co-processor. Hardware reconfiguration for every new query is not an option as this would be simply not fast enough.

B. XQuery Projection Paths

In our solution, extracting filter predicates from XQuery expressions is based on *XML projection* introduced in [7]. We provide a hardware implementation for XML projection. To understand the idea of XML projection, consider the following query, which is based on XMark [8] data:

```
for $i in //regions//item
  return <item>
           { $i/name }
           <num-categories>
           { count ($i/incategory) }
           </num-categories>
         </item>
```

This query looks up all auction items[3] and prints their name together with the number of categories they appear in. Out of a potentially large XMark instance, the query above will need to touch only a small fraction that has to do with items and their categories. What is more, this fraction can be described using a set of very simple *projection paths*, which are essentially XPath expressions:

```
{ //regions//item,
  //regions//item/name #,
  //regions//item/incategory }
```

Only nodes that match any of the paths in this set are needed to evaluate the above query. All other pieces of the input document can safely be discarded without affecting the query outcome. By default we keep only the matching node itself in the projected document, but discard any descendant nodes that do not match any projection path as well. Whenever the query demands to keep the entire subtree below some matched path, we annotate this path explicitly with a trailing # symbol (consistent with the notation in [7]).

C. Dynamic Projection Path Adaptation

The projection paths extracted from each XQuery expression are often viewed as a set of regular expressions. These regular expressions could be compiled into static circuits and run on the FPGA, as described in Section II. However, as it would be too slow to reconfigure the FPGA every time new XPath expressions are sent from the central processor to the co-processor, we need a more dynamic solution.

In order to explain how our solution works, we first give a brief overview of the main hardware components involved (see Figure 2). First, a SAX-like *XML parser* adds lexical information to the XML stream. Then, this enriched stream passes through a pipeline of *segment matchers* that take care of the path matching. Finally, when one of the projection paths was matched, the *serializer* forwards the appropriately filtered XML to the backend XQuery processor.

The pipeline of segment matchers is used to implement the set of XPath expressions. Each XPath expression is translated to a finite-state automaton. A single segment matcher

[3]xmlgen (the XMark data generator) produces XML documents modeling an auction website.

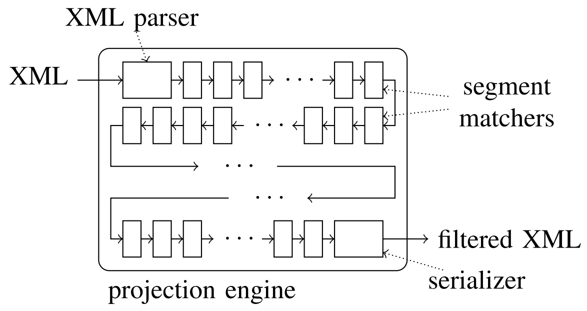

Fig. 2. The sequential structure of the XML projection engine can efficiently be mapped to the two-dimensional chip space.

corresponds to one state of such an automaton. Multiple XPath expressions are sequentially aligned as sets of segment matchers. The symbols (tags in this case) that can activate a given state/segment can be configured at runtime by writing to registers (flip-flops) prepared for this purpose. Additionally, any segment can be marked as start state of some automaton at runtime. Thus, while the pipeline of segment matchers is a static circuit (generated off-line), the specific set of XPath expressions that it implements can be configured at runtime.

IV. FPGAs for Data Mining

In a data warehouse appliance equipped with configurable hardware it makes sense to outsource certain compute-intensive mining tasks to an appropriate co-processor implemented on that hardware. For our data analytics framework, we intend to build a library consisting of suitable templates for mining operators that can be customized and loaded onto an FPGA. A data mining task is typically compute-intensive enough to justify invoking a co-processor and moving the respective computation to a dedicated hardware circuit. For instance, finding frequent items [9] or the computation of the *skyline* [2] would be qualifying tasks.

A. Skyline Computation in Software

Given a set of points $p_1, p_2, \ldots p_n$, the skyline query returns a set of points P, such that any point $p_i \in P$ is not *dominated* by any other point in the dataset. A point p_i *dominates* another point p_j iff p_i is better than or equal to p_j in every corresponding dimension and better in at least one dimension.

Today, a variety of skyline algorithms exist, many targeting different objectives. Here, we have chosen to focus on the block nested loop (BNL) algorithm [2] because it is simple—making it suitable for an implementation in hardware—and yet efficient. BNL is based on the idea of keeping a window of potential skyline points in main memory. Data points reside in this window until they are either evicted by a dominant point or can be output as skyline points. The algorithm iterates over the input data multiple times until all skyline points are found.

Making the window larger reduces the number of runs over the input data but at the same time more comparisons need to be performed for every data point. Reducing runs is only effective to a certain point. Then the cost for the extra comparisons due to the larger window outweigh the reduced number of runs. We conducted an in-memory experiment that confirms this insight. The result is depicted in Figure 3.

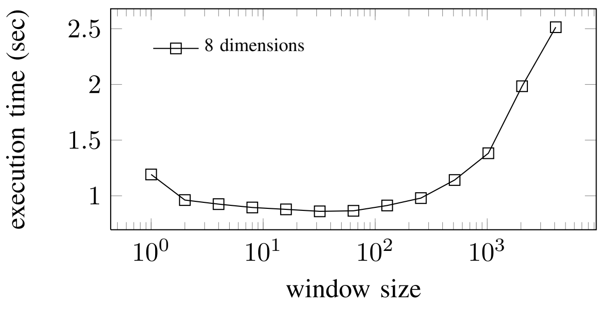

Fig. 3. Skyline computation of 102400 randomly distributed data points, each with eight dimensions. The "skyline" consisted of 8851 points.

B. Skyline Computation in Hardware

The fact that FPGAs are massively parallel architectures can be exploited to our advantage. If we could access all data points in the window in parallel, then multiple data points could be compared against window points concurrently. This can be achieved by building the window of the BNL algorithm as a pipeline of window points. At each pipeline element we store exactly one window point using flip-flops. In addition, each window point is equipped with a comparator for the dominance tests with the data points passing by. This idea is illustrated in Figure 4.

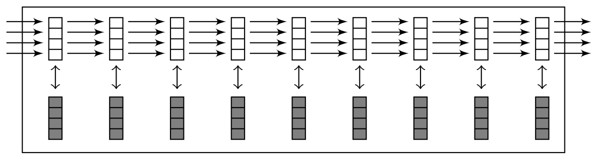

Fig. 4. Pipelined window for the BNL algorithm. The gray boxes represent window points. The white boxes are data points flowing by. In this illustration each data point has four dimensions.

The interesting aspect of this implementation is that it allows us to reduce runs without paying the price for the extra comparisons. Since this circuit is fully pipelined, we can insert a new data point into the pipeline no matter how large the window (pipeline) is. As the comparators are executed concurrently the throughput of this operator will always be the same.

V. Research Directions

There are many more interesting large-scale data analytics problems where FPGA-based solutions promise significant improvements over traditional systems. We plan to gradually extend our framework with tools that tackle these kinds of problems. For example, we are currently developing a SATA *host bus adapter* (HBA) core for direct hard disk access from within the FPGA. With such a core at hand, we could, *e.g.*, compute histograms "for free" as data is moved from disk to memory. Or we could build efficient logging applications, where network traffic is constantly monitored and fragments of the traffic are stored on disk when certain (complex) events of interest occur.

VI. Summary

In this paper, we have presented our initial and on-going research towards a framework for building highly efficient, hardware-accelerated data analytics applications based on FPGA technology. Our framework targets applications where vast amounts of data need to be scanned in a streaming manner and tedious tuning using indexes etc. is not an option. We have given insight into the following three subproblems that we have been investigating so far:

- ▷ Detecting *complex event patterns* on high-rate data streams in the network. We have presented a compiler that turns queries expressed in a high-level, declarative language into circuit specifications in some hardware description language such as VHDL. Placing those circuits close to the network allows us to avoid the network-memory-CPU bottleneck of commodity systems and thus achieve far better results.
- ▷ By employing FPGAs for XML filtering, we have addressed another problem. On the one hand, we have shown how FPGAs can be used effectively for load shedding. On the other hand, we have presented a novel approach for runtime-reconfiguration of FPGA circuits.
- ▷ Finally, we have shifted our focus towards data mining tasks such as *skyline* computation. As this is very recent work, no final results have been presented. However, exploiting the inherent parallelism of FPGAs for these kinds of mining problems is a promising approach and we hope to publish respective results soon.

References

[1] D. DeWitt, "DIRECT—a multiprocessor organization for supporting relational database management systems," *IEEE Trans. on Computers*, vol. c-28, no. 6, Jun. 1979.
[2] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of the 17th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 421–430.
[3] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "Sase: Complex event processing over streams," in *CIDR'07*, Asilomar, CA, USA, 2007.
[4] R. Müller, J. Teubner, and G. Alonso, "Streams on wires - a query compiler for fpgas," in *VLDB'09*, Lyon, France, 2009.
[5] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with FPGAs," *Proc. of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, Sep. 2010.
[6] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby, "Pattern matching in sequences of rows," in *Technical Report ANSI Standard Proposal*, 2007. [Online]. Available: http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf
[7] A. Marian and J. Siméon, "Projecting XML documents," in *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, Sep. 2003.
[8] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A benchmark for XML data management," in *Proc. of the 28th Int'l Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, Aug. 2002, pp. 974–985.
[9] J. Teubner, R. Müller, and G. Alonso, "FPGA acceleration for the frequent item problem," in *ICDE'10*, Long Beach, CA, USA, 2010.