Compiling Text Analytics Queries to FPGAs

Raphael Polig, Kubilay Atasu, Heiner Giefers
IBM Research - Zurich
Rueschlikon, Switzerland
(pol, kat, hgi)@zurich.ibm.com

Laura Chiticariu
IBM Research - Almaden
San Jose, CA, USA
chiti@us.ibm.com

***Abstract*—Extracting information from unstructured text data is a compute-intensive task. The performance of general-purpose processors cannot keep up with the rapid growth of textual data. Therefore we discuss the use of FPGAs to perform large scale text analytics. We present a framework consisting of a compiler and an operator library capable of generating a Verilog processing pipeline from a text analytics query specified in the annotation query language AQL. The operator library comprises a set of configurable modules capable of performing relational and extraction tasks which can be assembled by the compiler to represent a full annotation operator graph. Leveraging the nature of text processing we show that most tasks can be performed in an efficient streaming fashion. We evaluate the performance, power consumption and hardware utilization of our approach for a set of different queries compiled to a Stratix IV FPGA. Measurements show an up to 79 times improvement of document-throughput over a 64 threaded software implementation on a POWER7 server. Moreover the accelerated system's energy efficiency is up to 85 times better.**

I. Introduction

The volume of data on our planet is increasing exponentially. Every day 2.5 billion gigabytes of data is generated of which 80% is unstructured in the form of audio/video records, blog entries, tweets, and sensor data [1]. By using information extraction technologies, scientists and companies are trying to utilize the data to find valuable information. Queries used to pinpoint the desired information are becoming more complex and require more data to be scanned. This requires compute systems that can efficiently execute these queries in terms of to time and power.

Text Analytics (TA) refers to the task of information extraction from unstructured or semi-structured text data. By formulating extraction queries, a user can extract desired information from a set of documents. There are several software frameworks available for text analytics, such as GATE [2], NLTK [3], UIMA [4] or SystemT [5]. To achieve a desired document-throughput rate, these frameworks are usually deployed on compute clusters where each node operates on a subset of documents. Although this scale-out approach achieves the performance goal, it lacks in efficiency by nearly doubling CPU time [6].

Text analytics involves several compute-intensive tasks, which can be categorized into *extraction* and *relational* operators. Extraction operators operate across an entire document and detect string patterns or regular expressions. For each pattern detected, these operators need to report the position within a document, referred to as span. Relational operators operate on these spans, trying to identify a relation between them. Using these operators, information can be created by, e.g., assigning a telephone number to a name. For that, the vicinity of each detected name is inspected for a phone number. Although many improvements have been made to increase the performance of such software frameworks they cannot keep up with the rapid growth of data.

We present an FPGA based accelerator architecture for the SystemT text analytics engine. It consists of a set of configurable hardware modules that can perform the extraction and relational tasks of a text analytics query in a streaming fashion by leveraging the natural ordering in text processing. The framework allows one to build complete queries in hardware and ensures proper operation.

The main contributions of this work are:

1) an operator library for relational tasks in the field of text analytics
2) a compiler that combines the extraction and relational operators in an appropriate way to represent a user specified query
3) evaluation of the performance, power and hardware utilization for different customer queries

The remainder of the paper is structured as follows: Section II gives an overview of related work. A brief introduction to the annotation query language will be given in Section III. We present the hardware operator modules in Section IV and the compilation process in Section V. Section VI contains our experimental results, and we conclude in Section VII.

II. Related work

The *Glacier* [7] project proposes a similar architecture for relational queries. It consists of a component library for various query operators and a compositional compiler for combining them into a full query. But Glacier has limited support for a variable-length string data type which is a crucial feature as extraction matches resulting from the text can be of arbitrary length. Moreover, *Joins* are disallowed in Glacier.

In [8] another component library for SQL queries is proposed for using dynamic partial reconfiguration. This allows on-the-fly compilation of queries to the FPGA without running the full synthesis flow. Although our compilation times can be multiple hours long, they are negligible compared with the runtimes of the compiled system.

A commercial product using FPGAs to accelerate queries is IBM's PureData System [9]. The accelerator is directly attached to the disks and prefilters the queried data before sending it to the main memory.

```
create view Last as
        extract dictionary lastnames.dict on D.text
        as name from Document D;

create view First as
        extract dictionary firstnames.dict on D.text
        as name from Document D;

create view Caps as
        extract regex /[A-Z][a-z]*/ on D.text
        as name from Document D;

create view Person as
select S.name as name
from (
        ( select CombineSpans(F.name, C.name) as name
          from First F, Caps C
          where FollowsTok(F.name, C.name, 0, 0))
   union all
        ( select CombineSpans(F.name, L.name) as name
          from First F, Last L
          where FollowsTok(F.name, L.name, 0, 0))
   union all
        ( select *
          from First F )
) S
consolidate on name;
```

Fig. 1. Example of a query written in the annotation query language (AQL).

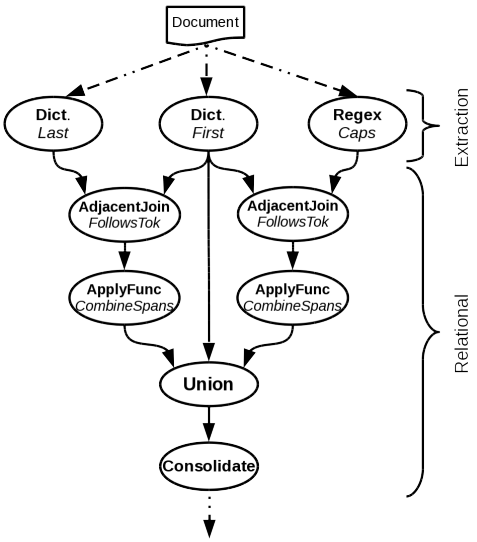

Fig. 2. Example of an annotation operator graph (AOG).

LINQits [10] is a flexible hardware template that can be configured using the language integrated query language LINQ. The compiled accelerators are tightly coupled to the processor and support various operations on data collections such as *Select*, *Aggregate* or *Join*.

Jean et al. [11] present an FPGA design to accelerate queries on databases that contain text and images. It supports *Select* statements that use text keywords or image templates to find appropriate entries in the database. *Joins* are accelerated by using a binary search step on the FPGA which only supports integer keys.

Wu et al. [12] present a GPU-based accelerator to improve the performance of relational database operations. By fusing code bodies of multiple kernels, they reduce the execution time spent on data movement and allow the compiler to perform more optimization. Another approach to query acceleration is the use of systolic arrays as proposed by Kung et al. [13]. It allows the operations to be pipelined in two dimensions: per tuple and per field. Although this is an efficient architecture for many relational operations, it is cumbersome for implementing textual operations.

In our complementary work [14], we show how an accelerator architecture can be integrated into the SystemT runtime environment. In this work, we introduce the hardware compilation process for relational algebra operations and show how they can benefit from the nature of text analytics to be executed in a streaming fashion. This allows the use of *Joins* and complex conditional statements, such as regular expression matching for *Select* operators

III. Annotation Query Language

We base our work on the information extraction software SystemT [5]. SystemT uses a declarative rule language called Annotation Query Language (AQL) to define the information to be extracted from a text source. AQL combines classic text-based tasks, such as regular expression matching or string matching with relational tasks like *Select* or *Join* known from database applications. This permits queries to be written in a modular way using simpler text-based operations and to combine their results later on. Figure 1 shows an example of an extraction rule written in AQL for exctracting full names.

The compiler transforms an AQL query into an annotation operator graph (AOG), which can be executed by the software. The AOG represents the data dependency between the individual operators used in a query. A cost-based optimizer will choose the best execution plan of an AOG based on a set of reference documents. This allows the software to skip whole parts of the AOG if a certain operator does not produce any outputs. Figure 2 shows the AOG representing the AQL query seen in Figure 1.

The main datatype used in SystemT is called *span* and defines a region of interest within a text document. A span is a combination of two integer values comprising a character-based start- and end-offset. All text-based extraction operators produce a set of spans indicating the position of their matches. Relational operators mainly use spans as inputs, but can also refer back to the actual string data defined by the span.

For the hardware implementation, we have extended the definition of a span by two 16-bit integers defining the token-based start and end of a span. Here a token is a predefined segment of the input document, such as a word or a punctuation character.

SystemT operates on a document-per-thread execution model. This means a single thread runs the complete AOG on a full document. The operators in an AOG are therefore executed in a serial fashion. For our hardware accelerator we choose a streaming execution model in which all operators run in parallel forming a large pipeline. Each pipeline operates on an individual document, thus providing a further level of parallelism.

IV. Relational algebra operators

The following section describes some of the hardware operators available and how they can be interconnected. The hardware operators for extraction tasks have been described in

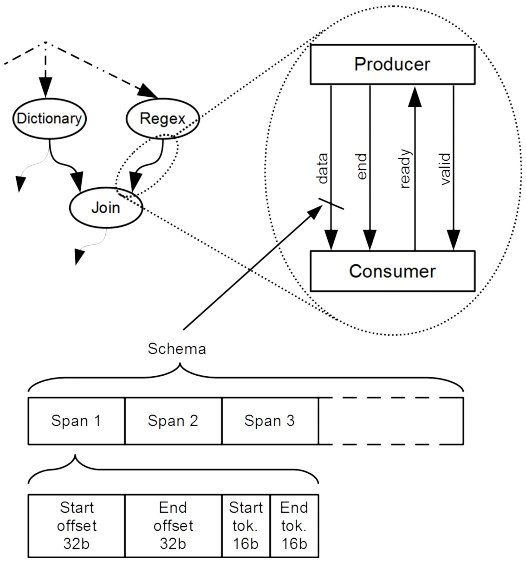

Fig. 3. An elastic interface is used to interconnect the operator modules. The data width depends on the producer's schema and is mainly composed of a number of spans each consisting of a character-based start and end offset and a start and end token index.

[15] and [16]. Our main goal is to achieve a streaming operation for the relational tasks to avoid any backpressure to the extraction operators as they define the document-throughput rate. The relational operator designs leverage the fact that the extraction tasks produce results sorted by either the start or end offset. Configuring the producing operators appropriately or using simple sorting buffers allows the receiving operator modules to get the input in their required ordering.

To connect the various operators, a common communication scheme and interface need to be defined that are flexible in the number of parallel streams and their individual data width. Also bi-directional flow control must be supported to allow back pressure from operators that require multiple cycles of processing. We have chosen an elastic interface to be the top-level interface for all operators.

The interface consists of four signals: *valid*, *ready*, *data* and *end*. For each stream/edge, *valid*, *ready* and *end* are single-bit-wide signals, whereas the width of *data* is determined by the schema of the producing operator. The producer presents new data on the data bus and signals that to the consumer by raising the valid signal. The consumer acknowledges the transmission by keeping the ready signal high. If the consumer's ready signal is low the producer needs to keep valid high and maintain the data steady until the consumer becomes available again.

A. Select

The *Select* operator is a filter operation on a single input stream. It evaluates a condition expression for each tuple it receives and outputs only the ones for which the condition becomes true. This operator works independently of the order of the incoming tuples but needs to be individually compiled as the condition may be a complex expression composed of pattern matches, comparisons and boolean functions. The condition expression is represented in the AOG as a set of secondary nodes attached to the Select operator shown on the top right of Fig. 4.

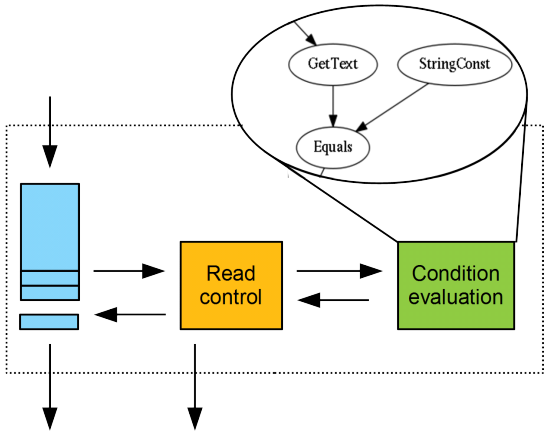

Fig. 4. Select top-level architecture containing a FIFO, a read control unit, and the custom-compiled condition-evaluation module.

The top-level hardware module seen in Fig. 4 is composed of three main components: a FIFO to buffer the incoming tuples, a read control unit, and the custom-compiled condition-evaluation module. Once data is available in the FIFO, the read control unit will read a single tuple and pass it on to the condition evaluation. Processing can take multiple clock cycles and will return two bits indicating that the evaluation has finished and whether the condition is true or false. The read control will then validate or invalidate the output data and continue with the next tuple in the buffer.

The condition may require the operator to access the actual document data again when checking for a string compare or regular expression. If that is the case the operator module is extended with a document buffer that holds a portion of the text document and can be accessed directly from the condition-evaluation module. The incoming span defines the location and amount of text data to be fetched from the document buffer, which leads to arbitrary processing times for the condition evaluation.

Although the condition evaluation is a blocking operation, it is sufficiently fast to keep the tuple buffer from overflowing. Profiling information shows the average number of input tuples to the operator for a set of reference documents. Typical numbers range from less than 0.1 to about 16 tuples for an average document size of 251.5 bytes. This means a tuple enters the operator every 16 characters, on average.

The profiling information is available to the compiler and allows it to set the depth of the tuple buffer appropriately. Furthermore the compiler sets the buffer's width according to the input schema and connects the necessary fields to the condition evaluation unit.

B. Consolidation

Consolidation is another filter operation applied to a single stream of inputs that handles duplicates or overlaps. Each operator instance can have a different consolidation predicate defining its functionality. Examples are *ExactMatch*, which will discard all spans that are equal to a previous one or

ContainedWithin, which discards all spans that are wholly contained within another one. In contrast to the *Select* operator, this requires the evaluation of more than a single input tuple.

The hardware module leverages the fact that the input can be sorted for little or no cost by the producer. For the *ExactMatch* predicate, the module will only validate output data when consecutive spans are not equal or the end of a stream is reached. In this case, the input data can be sorted by end first and then by start offset or vice versa, but needs to be fully sorted. A 64-bit comparator is used to check equality on the character start and end offsets. There is no need to check also the token indices as they are derived by the character offsets.

For the *ContainedWithin* predicate, the input data needs to be sorted by start offset only. In a first stage, the longest span for each unique start offset is detected as this contains all other spans with the same start. The resulting spans are passed to a second stage which removes all spans having an equal or smaller end as they are contained within an earlier span that had a smaller start offset.

The compiler selects the appropriate module and configures it with the according data width. It then identifies the field in the schema the consolidation operates on and wires the input appropriately.

C. Adjacent Join

The *Adjacent Join* operator takes two streams (A and B) of spans as input. It then calculates the distance from a span from A to a span from B either as number of characters or number of tokens. If the distance is within the configured range of the operator the two tuples are joined to form a single new tuple at the output.

The hardware implementation of this operator profits from the ordering of the two input streams where stream A is sorted by end offset and stream B is sorted by start offset. Fig. 5 shows the top-level of the operator module with some example input on the left. The R/W control unit waits until it receives valid inputs from both buffers A and B. It then calculates the distance between the two spans to determine the next action. If the distance is within the configured range the joined tuple is written to the output and the tuple of B is stored to a temporary buffer T. New tuples are read from buffer B and checked until the maximum distance is exceeded. If the distance is exceeded a new tuple from A is read and is checked against all tuples stored in buffer T, discarding all tuples from T that fall below the minimum distance. Once buffer T is fully checked the control unit moves on to buffer B again.

The compiler evaluates the join predicate, which can either be *Follows* or *FollowedBy*. In the case of *FollowedBy*, it swaps the two input streams before wiring them to the input buffers and then swaps the joined output to keep the correct ordering of the fields. It then sets the maximum and minimum distance values and wether the module needs to calculate a character-based or token-based distance.

This architecture is not suitable for a generic *Join*, as the size of buffer T limits the maximum number of *Joins* that can be performed for a single tuple of stream A. Furthermore if one of the input buffers is full the operator may put the entire query into a deadlock situation. If a malformed document causes this situation an exception is signaled to the software to rerun the document in software. To avoid this situation, the compiler uses the profiling information to appropriately size the input buffers.

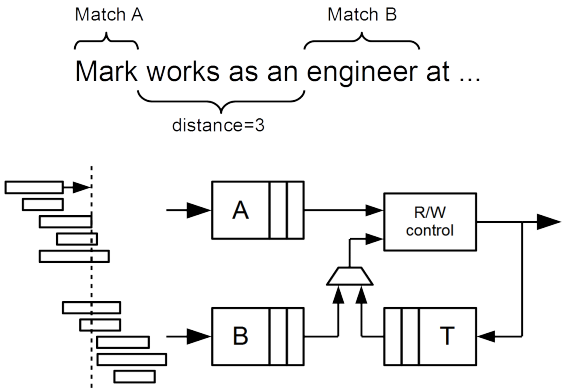

Fig. 5. Adjacent Join can be performed in a streaming fashion if stream A is sorted by end offset and stream B by start offset.

D. Union

A *Union* operator takes any number of input streams greater than two and combines them into a single output stream. The inputs need to have the same schema as there will be only one output schema. In hardware design this translates into any kind of arbiter that serializes multiple input busses onto a single output bus. If the downstream operators do not require the data to be sorted a round-robin arbitration scheme is used. If a subsequent operator needs the data to be sorted a specialized sorting union is used which keeps a sorted order between different inputs. It will not sort the individual input streams.

E. Difference

The Difference operator is similar to an *ExactMatch* consolidation, but operates on two input streams. It will remove any tuples from input stream A that are also present in stream B. For this operator to work in a single pass, both inputs need to be fully sorted regardless whether first by start or end offset.

The hardware module consists of two input FIFO buffers and a read control unit. As soon as both buffers contain valid data, the read control unit will compare the two tuples. If the tuple from A is smaller than the tuple from B, it gets passed to the output and the next tuple is read from the A FIFO. If the tuples match, the tuple from A gets discarded and FIFO A advances. If A is larger than B, FIFO B advances.

As the *Difference* operator needs to synchronize two streams, it can put the query pipeline into a deadlock situation similar to the *AdjacentJoin* operator. The mechanism to avoid this situation is also the same as for *AdjacentJoin*.

F. Apply Function

The *ApplyFunc* operator receives a single input stream and applies a secondary function to it. It is similar to the *Select* operator but instead of forming a selection criterion, the

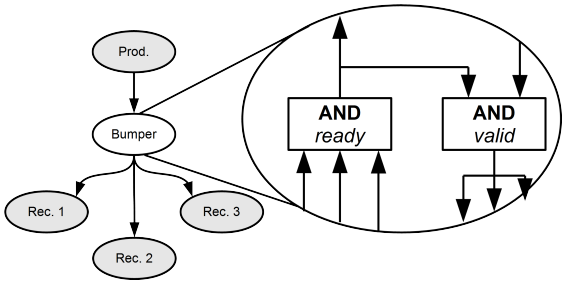

Fig. 6. Bumper node to synchronize the dataflow for multiple receivers.

result of the secondary function is added to the input schema, resulting in a wider output than input schema.

There are many secondary functions, and currently the compiler supports a set of commonly used ones, such as the *CombineSpans* function, which takes two spans and generates a new span by selecting the smallest start offset and the largest end offset. Similarly, *SpanBetween* selects the smallest end offset as a new start and the largest start offset as a new end to create a new span. All functions supported do not require special ordering to work properly.

G. Project

The project operator reorders the schema of an input tuple or selects only a subset of fields from it. This operator is implemented as special wiring between other operators and causes no costs in terms of logic resources. Resulting dangling operator pins from removed fields are not specially handled by the compiler, but we rely on the synthesis tools to remove unnecessary logic.

V. Compilation

The compilation process starts by iterating over all operator nodes and deriving the ordering requirements for the input edges. In a second pass, the compiler detects all edges with the same ordering requirements that are shared as inputs by multiple operators. These edges are replaced by so-called bumper nodes that combine the backpressure signals of multiple receivers. If one of the receivers is not ready to accept new data, the bumper node will invalidate the data for the other receivers and lowers the ready flag for the producer. This is realized using two AND structures to produce the final signaling as shown in Fig. 6.

Once all requirements have been determined the compiler starts generating the individual operators. With the exception of *Project*, all operators are generated as individual Verilog modules. As a last step, the compiler assembles the top-level by instantiating all operator modules and connects them accordingly. At this stage all *Project* operators are applied to the wiring between the modules. The routing code takes into account special interface properties of the operator modules, e.g., a single *Dictionary* module processes four document streams and thus only a single instance is required as opposed to a *Select* module.

The compilation process takes up to a minute to generate the full Verilog description from an AQL query. This includes compiling the AQL to an AOG and running the profiler to obtain the necessary information. Once the Verilog files are generated the synthesis tools are launched to generate the FPGA configuration bitstream. Depending on the complexity of the query, this step may take a few hours. This is acceptable for text analytics queries as they are in place for multiple days or even continuously.

VI. Experiments & Results

In this section we explore the performance and scalability of the system presented, as well as it's power consumption. For this purpose we selected a set of customer queries and processed them with the compiler. Table I summarizes the query profiles showing the various operator and output counts.

TABLE I. Number of operators for different queries.

Operator	Query A	Query B	Query C	Query D	Query E	Query F
Dictionary	1	4	4	5	8	23
RegExp	3	6	6	9	17	26
AdjacentJoin	0	4	9	9	18	0
Select	4	0	0	4	4	0
Consolidate	1	0	2	3	3	0
Union	2	1	5	7	8	0
Difference	2	0	0	2	2	0
ApplyFunc	0	4	9	9	12	0
Project	35	56	88	125	202	32
Total	48	75	123	173	274	81
# of outputs	3	10	1	4	28	55

The Verilog designs were synthesized for an Altera Stratix® IV GX530 FPGA using the Quartus® 13.1.3 tool chain. All queries were implemented to support processing of four documents in parallel. The necessary load/store and control units are included in the numbers reported and remained constant for all queries. Table II shows the hardware characteristics for the different queries and four parallel streams.

The strong use of buffers in many operators resulted in a high utilization of M9K RAMs. Besides the operators themselves, also the output arbiter that serializes the required outputs to the DMA module has a resource utilization. Its size is determined by the number of outputs that need to be reported. The strong impact can be seen when comparing queries E and F. The resource utilization is similar for the two queries although the operator count for query F is less than a third of that of E. But as the output count doubles, the arbiter contributes more strongly to the overall resource utilization. The maximum achievable frequency stays above 200 MHz for queries A through D and then drops to about 187 MHz for query E.

TABLE II. Resource utilization and maximum frequency for different queries when using four parallel streams.

Query	ALUTs (%)	Registers (%)	M9K (%)	f_{max}
Query A	51384 (12)	74694 (18)	374 (29)	219.68 MHz
Query B	52246 (12)	84673 (20)	307 (24)	230.20 MHz
Query C	67700 (16)	119564 (28)	392 (31)	202.35 MHz
Query D	109806 (26)	158086 (37)	519 (41)	206.95 MHz
Query E	181874 (43)	270008 (64)	675 (53)	187.13 MHz
Query F	142703 (34)	203853 (48)	258 (20)	199.16 MHz

For a performance comparison, we ran all queries using SystemT on a POWER7® server running at 3.55 GHz capable of 64 threads. The same machine is used for the FPGA measurements. The query processing pipelines on the FPGA are running at 200 or 166 MHz depending on the f_{max} of the query. The device is attached to the host via a peripheral bus capable of 3.4 GB/s throughput. The host prepares a large set of documents in a buffer and then calls the FPGA to process it.

Table III shows the peak throughput rates measured together with energy efficiency in terms of MB per Joule. Although the software threads operate on different documents independently from each other, the maximum throughput rate is very limited with up to 32.4 MB/s for query A. Also the complexity of the query impacts the software performance as document-throughput degrades with an increasing number of operators. The FPGA is able to reach the peak throughput rates defined by the maximum achievable frequency of the compiled query. The peak throughput rate can always be achieved when sending work packages that are only 1-2 kB large.

TABLE III. MAXIMUM THROUGHPUT AND ENERGY EFFICIENCY NUMBERS FOR DIFFERENT QUERIES.

	1 SW thread		64 SW threads		FPGA	
Query	**MB/s**	**MB/J**	**MB/s**	**MB/J**	**MB/s**	**MB/J**
Query A	3.8	0.02	32.4	0.14	800.0	3.60
Query B	2.4	0.01	26.7	0.11	799.6	3.60
Query C	2.5	0.01	26.4	0.11	800.0	3.60
Query D	1.6	0.008	19.9	0.08	799.6	3.62
Query E	0.4	0.002	8.5	0.036	664.0	2.98
Query F	0.4	0.002	8.4	0.035	664.0	2.98

We have evaluated the system's power consumption by looking at the information provided by the EnergyScale power management system [17]. For the measurements, we have activated dynamic voltage and frequency scaling (DVFS). While idle, the system consumed 199 W without the FPGA accelerator attached. When running the single software thread experiments, we observed a power consumption of 202 W whereas consumption went up to 236 W when running with 64 threads. When plugging in the FPGA card, the baseline power consumption increased to 214-216 W, depending on the query programmed on the FPGA. During processing of the documents the overall system power consumption was about 220-222 W which is 6% lower than during pure software processing.

VII. CONCLUSION

We have presented a framework to compile and run text analytics queries on FPGAs. By leveraging the natural order of results produced by extraction tasks processing a document, we show that many relational operations can be performed in a streaming fashion. We have shown that our framework is capable of compiling and running real-life queries while achieving up to 79 times throughput improvement over a two socket POWER7 server running with 64 parallel threads. Moreover the overall power consumption of the accelerated system is 6% less during processing, resulting an up to 85 times higher energy efficiency for processing text analytics queries.

We will continue our work by adding support for more relational operators such as aggregation functions. Furthermore we will analyze the effective utilization of the operator modules during operation. Also we will investigate the possibility and effect of operator clustering and hardware sharing.

REFERENCES

[1] "IBM: What is Big Data? Bringing Big Data to enterprise." http://www-01.ibm.com/software/data/bigdata/, accessed: 2014-03-21.

[2] H. Cunningham, "GATE, a general architecture for text engineering," *Computers and the Humanities*, vol. 36, no. 2, pp. 223–254, 2002.

[3] S. Bird, "NLTK: The natural language toolkit," in *Proceedings of the COLING/ACL on Interactive Presentation Sessions*. Association for Computational Linguistics, 2006, pp. 69–72.

[4] D. Ferrucci and A. Lally, "UIMA: An architectural approach to unstructured information processing in the corporate research environment," *Natural Language Engineering*, vol. 10, no. 3-4, pp. 327–348, 2004.

[5] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, "SystemT: A system for declarative information extraction," *ACM SIGMOD Record*, vol. 37, no. 4, pp. 7–13, 2009.

[6] V. Tablan, I. Roberts, H. Cunningham, and K. Bontcheva, "GATECloud.net: A platform for large-scale, open-source text processing on the cloud," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1983, 2013.

[7] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: A query compiler for FPGAs," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 229–240, 2009.

[8] C. Dennl, D. Ziener, and J. Teich, "On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library," in *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2012*. IEEE, 2012, pp. 45–52.

[9] Datasheet, "IBM PureData system for Analytics N2001," 2013. [Online]. Available: http://www-01.ibm.com/software/data/puredata/analytics/

[10] E. S. Chung, J. D. Davis, and J. Lee, "LINQits: Big data on little clients," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 261–272.

[11] J. S. Jean, G. Dong, H. Zhang, X. Guo, and B. Zhang, "Query processing with an FPGA coprocessor board," in *Proc. 1st Int. Conf. Engineering of Reconfigurable Systems and Algorithms*, 2001.

[12] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili, "Kernel weaver: Automatically fusing database primitives for efficient GPU computation," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 107–118.

[13] H. Kung and P. L. Lehman, "Systolic (VLSI) arrays for relational database operations," in *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*. ACM, 1980, pp. 105–116.

[14] R. Polig, K. Atasu, L. Chiticariu, C. Hagleitner, H. P. Hofstee, F. R. Reiss, E. Sitaridi, and H. Zhu, "Giving Text Analytics a Boost (to be published)," *IEEE Micro*, vol. 34, 2014.

[15] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss, "Hardware-accelerated regular expression matching for high-throughput text analytics," in *23rd International Conference on Field Programmable Logic and Applications (FPL), 2013*. IEEE, 2013, pp. 1–7.

[16] R. Polig, K. Atasu, and C. Hagleitner, "Token-based dictionary pattern matching for text analytics," in *23rd International Conference on Field Programmable Logic and Applications (FPL), 2013*. IEEE, 2013, pp. 1–6.

[17] H. Y. McCreary, M. A. Broyles, M. Floyd, A. Geissler, S. P. Hartman, F. Rawson, T. Rosedahl, J. Rubio, and M. Ware, "EnergyScale for IBM POWER6 microprocessor-based systems," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 775–786, Nov 2007.