Leftmost Longest Regular Expression Matching in Reconfigurable Logic

Kubilay Atasu
IBM Research - Zurich
kat@zurich.ibm.com

***Abstract*—Regular expression (regex) matching is an essential part of text analytics and network intrusion detection systems. The leftmost longest regex matching feature enables finding a leftmost derivation of an input text and helps resolve ambiguities that can arise in natural-language parsing. We show that leftmost longest regex matching can be efficiently performed in a data-flow pipeline by combining a recently proposed regex-matching architecture with simple last-in first-out (LIFO) buffers and streaming filter units, without creating significant back-pressure or using costly sorting operations. The techniques we propose can be used to compute overlapping and non-overlapping leftmost longest and rightmost longest regex matches. In addition, we show that the latency of the LIFO buffers can be hidden by overlapping the processing of subsequent input streams, without replicating the buffer space. Experiments on an Altera Stratix IV FPGA show a 200-fold improvement of the processing rates compared with a multithreaded software implementation.**

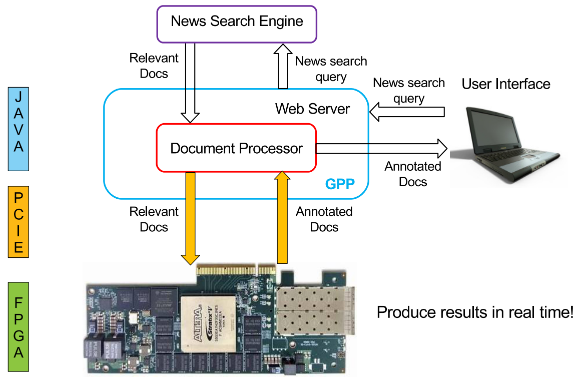


Fig. 1. Using FPGA-based accelerators for text analysis significantly improves the query-processing rates and enables real-time response latencies.

I. Introduction

We live in a data-centric world. Data is driving discovery in many fields, such as in healthcare analytics, cyber-security, weather forecasting, and computational astrophysics etc. The so-called big data has become a new natural resource, and discovering insights in big data will be the key capability of future computing platforms. The explosion in the size of the datasets is leading to a paradigm shift in system design. The need to achieve an efficient integration of massive data and computation is resulting in a major re-thinking of memory hierarchies and computing fabrics in datacentres. Data-centric systems diverge from traditional computer architectures in two main aspects. First, to improve the bandwidth of data access, computation is being moved closer to the data [1]. Secondly, the energy consumption of datacentres is increasing at an alarming rate, and energy costs start to exceed equipment costs [2]. Scaling up datacentre performance simply by increasing the number of processor cores is no longer feasible economically. To improve both performance and energy efficiency and to exploit the data-access bandwidth more efficiently, data-centric systems are increasingly relying on heterogeneous compute resources, such as graphics-processing units (GPUs) and field-programmable gate arrays (FPGAs).

The process of extracting information from large-scale unstructured text is called text analytics and has applications in business analytics, healthcare, and security intelligence. Analyzing unstructured text and extracting insights hidden in it at high bandwidth and low latency are computationally challenging tasks. In particular, text analytics functions rely heavily on regexs and dictionaries for locating named entities, e.g., person and company names, in free text [3], [4]. Typically, these regex and dictionary matching tasks, which are implemented using finite-state machines, dominate the runtime of text analytics systems [5]. The processing of finite-state-machine-based tasks does not map well on general-purpose processors [6]. However, FPGAs are an ideal medium for executing such tasks because of the massive parallelism they offer, which can be exploited at bit-level granularity [7].

Fig. 1 illustrates a use case of FPGA-based accelerators in a business analytics platform that continuously collects news entries from different data sources and indexes them using a local news search engine. When a user submits a news search query that contains a set of keywords, e.g., “IBM” and “Switzerland”, the news search engine retrieves all news entries that contain these keywords from its index. After that, the relevant news entries are parsed word by word to identify phrases that might, for instance, reveal a business expansion strategy of “IBM” in “Switzerland”, e.g., the opening of a new office or the announcement of a new strategic partnership. This second stage acts as a second level of filtering, and only those entries that contain interesting and useful information are transferred to the user, preferably in almost real time. This requires a deeper analysis of the news entries, and thus is computationally much more intensive than a simple keyword lookup in an index. When thousands of users submit news search queries concurrently, this second stage becomes a computational bottleneck. One way of eliminating this bottleneck is to scale up the number of processor cores, which, however, results in higher space and energy consumption and lower reliability. A more promising solution can involve combining an existing processor with a hardware accelerator, which boosts both the performance and the energy efficiency and enables real-time response latencies while preserving reliability.

Traditional regex-matching architectures based on reconfigurable nondeterministic finite-state automata [8], [9], [10] and programmable deterministic finite-state automata [11], [12] do not support features, such as *start-offset reporting*, *capturing groups*, and *leftmost longest matching*. Architectures that can compute a partially sorted stream of leftmost regex matches were given in [13], [14], [15]. This work presents formal algorithms and practical extensions to the architectures proposed in [13], [14], [15] to enable the computation of *overlapping* and *non-overlapping*, *leftmost longest* and *rightmost longest* regex matches. In particular, we show that a fully sorted stream of leftmost longest or rightmost longest regex matches can be produced without performing explicit sorting. Producing sorted results, in turn, enables execution of subsequent text-specific operations in a streaming fashion in a more general text analytics pipeline [16]. Our main contributions are:

1. A baseline architecture based on sorting and filtering for computing leftmost longest regex matches
2. An optimized architecture wherein sorting operations are replaced by simple LIFO operations
3. Adaptations of these architectures to compute non-overlapping and rightmost longest regex matches
4. An FPGA-based implementation that achieves a 200-fold speed-up over a multi-threaded software implementation of equivalent functionality

Section II defines the core concepts and terms used in the remainder of this paper, and Section III covers related work. Section IV introduces a general leftmost longest regex-matching architecture based on sorting and filtering. Section IV presents an optimized LIFO-based leftmost longest regex matching architecture that does not perform any explicit sorting. Section VI describes an extension to our optimized architecture that enables computation of non-overlapping leftmost longest regex matches. Section VII then shows that overlapping and non-overlapping rightmost longest regex matches can be computed by appropriately configuring our leftmost longest regex matching architectures. Section VIII presents our experiments and results, and Section IX our conclusions.

II. Background

State-of-the-art text analytics systems [3], [4] compute a *span* data structure for each regex match that indicates the start-offset position and the end-offset position of the match in the input text. To eliminate ambiguity, text analytics systems use well-defined tie-break heuristics when dealing with overlapping matches. In particular, when several regex matches that end at the same offset position exist, typically only the *span* with the smallest start-offset value will be reported. This technique is called *leftmost regex matching* heuristic. A more common and less trivial heuristic is called *leftmost longest regex matching*, which requires computation of all those regex matches that are *not contained* in other regex matches [17].

Definition 1: Each regex match is associated with a span (s, e), where s is the start-offset position and e the end-offset position of the regex match.

Definition 2: Span $(s0, e0)$ contains span $(s1, e1)$ if $(s1 \geq s0)$ and $(e1 \leq e0)$.

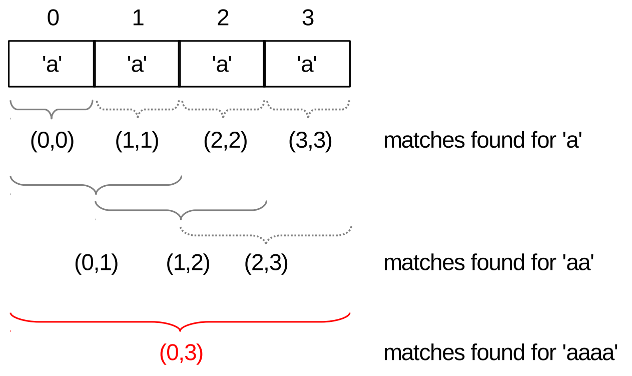


Fig. 2. The matches of the regex `a|aa|aaaa` in the input string `aaaa`. Out of the eight regex matches, only one is a *leftmost longest match*.

Definition 3: The leftmost regex match at offset position i is the regex match with the smallest start-offset position value that ends at offset position i.

Definition 4: A leftmost longest regex match is a regex match that is not contained in any other regex match.

Corollary 1: A leftmost longest regex match is also a leftmost regex match.

Assume that the regex `a|aa|aaaa` is matched against the input stream `aaaa`. Fig. 2 shows the matches associated with the subexpressions `a`, `aa`, and `aaaa`. There are eight distinct matches, each one uniquely defined by a start-offset position and an end-offset position. For instance, span $(0, 1)$ is associated with the match that starts at offset position 0 and ends at offset position 1. Out of these eight matches, only four are leftmost matches, indicated by the solid curly brackets. These are associated with spans $(0, 0)$, $(0, 1)$, $(1, 2)$, and $(0, 3)$. For instance, the leftmost match at offset position 1 is indicated by span $(0, 1)$, which has a smaller start offset than the match associated with span $(1, 1)$. Therefore, a leftmost regex matcher should only report span $(0, 1)$, suppressing the match for span $(1, 1)$. Similarly, at offset position 3, the leftmost match is associated with span $(0, 3)$, and the matches associated with spans $(2, 3)$ and $(3, 3)$ must be suppressed. Finally, only a single *leftmost longest regex match* exists in Fig. 2. It is associated with span $(0, 3)$, which is *not contained* in any other span. Note that all remaining leftmost matches (i.e., $(0, 0)$, $(0, 1)$, and $(1, 2)$) are *contained* in span $(0, 3)$.

III. Related Work

Span-based information-extraction systems [3], [4], [18], [19] have long been popular. The backbone of such systems are regex and dictionary matchers that operate on text documents and produce a sorted stream of leftmost longest matches. To the best of our knowledge, we propose the first fully hardware-based solution that addresses this important problem.

Regex matching can be performed by first transforming a regex into a nondeterministic finite-state automaton (NFA) or into a deterministic finite-state automaton (DFA), and then applying the input to the state-machine representation [20]. Sidhu and Prasanna showed that NFA structures can be mapped very efficiently to the programmable logic blocks of Field-Programmable Gate Arrays (FPGAs) [8], where each state stores a 1-bit register that indicates whether the state is active or not, and each state transition is implemented using a wire that is routed from the source state to the destination state.

Methods to reduce the complexity of the next-state computation logic using character-classifier tables are presented in [9]. Dedicated shifter and counter circuits can be used to derive a more compact representation of the next-state computation logic [10], [11], [12]. Similarities between regexs can be exploited to reduce resource consumption via resource sharing [21], [22]. The throughput rate of NFA-based regex matching can be improved by constructing automata that can consume multiple characters per clock cycle [9], [21], [23]. Software-based approaches that combine NFAs and DFAs to explore the trade-offs between memory consumption and computational resources have also been explored [24], [25]. The approach of [26] decomposes complex regexs into a sequence of simple strings that are matched using a DFA-based architecture, and an NFA-based post-processor implemented in reconfigurable logic ensures a correct ordering of the string matches. In [27], a memory-based and programmable NFA architecture that does not require hardware reconfiguration is described. However, none of these architectures support start-offset reporting, leftmost matching, or leftmost longest matching. For the example given in Fig. 2, these architectures would produce a match signal at each offset position, which would reveal only the end-offset positions of the regex matches without reporting the respective start-offset positions.

Supporting start-offset reporting in regex matching is computationally complex, because regex matches that are associated with different start offset and end offset pairs can overlap (see Fig. 2). Therefore, at any point in time, a regex matcher has to keep track of multiple execution paths associated with different start and end offsets. A reconfigurable hardware accelerator for regex matching that supports start-offset reporting and leftmost matching is given in [13]. It exploits the property that only a limited number of NFA states can be active concurrently, and uses this information to build a network of state machines that can perform a breadth-first search on the input text. To support start-offset reporting, each state machine stores a start offset value in its configuration registers, and an optimized network enables the exchange of configuration registers between state machines. Extensions to the architecture of [8] to support start-offset reporting and leftmost matching in a resource-efficient way are proposed in [14], [15]. It is shown that a simple extension of [8] results in a significant redundancy in the number of configuration registers and in the leftmost match computation logic. Therefore, optimizations that can eliminate such redundancies are also presented in [14], [15]. However, a solution for the more widely used and challenging problem of leftmost longest regex match computation is not provided neither in [13] nor in [14], [15].

IV. Baseline Architecture

This section describes our baseline architecture for computing leftmost longest regex matches, which combines a regex matcher that supports *start-offset reporting*, a *sorting unit*, and a *containment filter*, as shown in Fig. 3. The spans produced by the regex matcher are initially unsorted and can be sorted

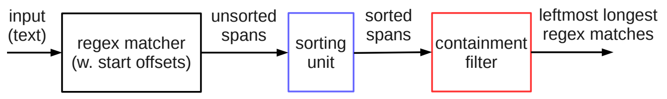
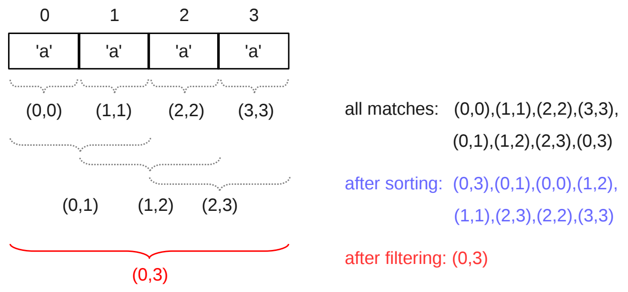


Fig. 3. The baseline architecture incorporates a regex matcher that supports start-offset and end-offset reporting, a sorting unit, and a containment filter.



0
1
2
3
'a'
'a'
'a'
'a'
(0,0)
(1,1)
(2,2)
(3,3)
(0,1)
(1,2)
(2,3)
(0,3)
all matches: (0,0),(1,1),(2,2),(3,3), (0,1),(1,2),(2,3),(0,3)
after sorting: (0,3),(0,1),(0,0),(1,2), (1,1),(2,3),(2,2),(3,3)
after filtering: (0,3)

Fig. 4. The steps that lead to computation of the leftmost longest matches.

in increasing order of the start offsets, where the spans having the same start offset are sorted in decreasing order of their end offsets. Alternatively, the spans can be sorted in decreasing order of the end offsets, where the spans having the same end offset are sorted in increasing order of the start offsets. The sorted spans are then fed into the containment filter, which filters out all spans that are *contained* in others and produces a sorted stream of spans that indicate the leftmost longest regex matches. Fig. 4 shows the application of our sorting- and filtering-based solution to the example given in Fig. 2.

The containment filter processes the sorted spans in a streaming way, and stores only one span in its local registers. Assume that the spans are sorted in increasing order of their start offsets, where the spans having the same start offset are sorted in decreasing order of the end offsets. When the first span arrives at the containment filter, it gets stored in the local registers without producing any output. When the next span arrives and its end offset is smaller than or equal to the end offset of the span stored in the local registers, the prior sorting operation guarantees that its start offset will be greater than or equal to the start offset of the span stored in the local registers. Thus, the span stored in the local registers *contains* the new span. The new span is filtered out and the span stored remains unchanged. If however the end offset of the new span is greater than the end offset of the span stored, the sorting guarantees that the start offset of the new span will be greater than the start offset of the span stored. Thus, neither span *contains* the other. The span stored in the local memory is written to the output, and the new span gets stored in the local registers. When the end-of-stream signal arrives at the containment filter, the tuple stored in the local registers is written to the output.

In general, the proposed architecture can produce multiple and possibly overlapping spans as output. As an example, assume that spans $(0, 1)$, $(2, 3)$, $(2, 4)$, and $(1, 5)$ are produced by the regex matcher. After sorting, the spans are reordered as follows: $(0, 1)$, $(1, 5)$, $(2, 4)$, and $(2, 3)$. The containment filter stores the first span $(0, 1)$ in its registers. When the second span $(1, 5)$ arrives, $(0, 1)$ is written to output and $(1, 5)$ is written to the registers. $(2, 4)$ and $(2, 3)$ are not leftmost longest matches

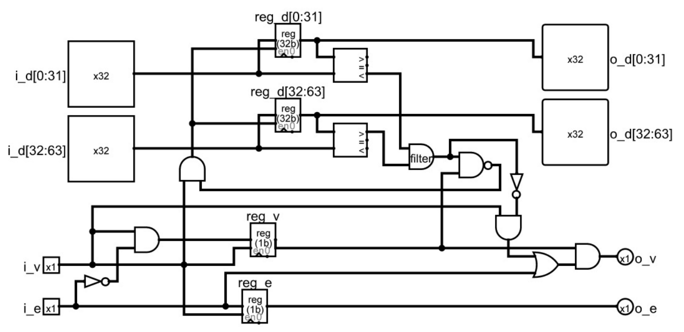


Fig. 5. Architecture of the containment filter: i_v, o_v, and reg_v stand for input valid, output valid, and register valid; i_d, o_d, and reg_d stand for input data, output data, and register data; and i_e, o_e, and reg_e stand for input end-of-stream, output end-of-stream, and register end-of-stream, respectively.

and are filtered out because they are *contained* in $(1, 5)$, i.e., they have a larger start offset and a smaller end offset than $(1, 5)$. When the end-of-stream signal arrives, $(1, 5)$ is written to the output. As a result, the two leftmost longest matches are computed, namely, $(0, 1)$ and $(1, 5)$.

Alternatively, the spans produced by the regex matcher can be sorted in decreasing order of the end offsets, where the spans having the same end offset are sorted in increasing order of the start offsets. In this case, it is sufficient to compare only the start offsets. If the start offset of the new span is larger than or equal to the start offset of the span stored, the new span is filtered out and the span stored remains unchanged. If the start offset of the new span is smaller than the start offset of the span stored, the span stored in the registers is written to the output, and the new span gets registered.

Figure 5 shows the hardware architecture of the containment filter. The architecture uses two 32 bit comparison units to check 1) if the start-offset value of the span registered is smaller than or equal to the start-offset value of the input span, and 2) if the end-offset value of the span registered is greater than or equal to the end-offset value of the input span. Such an architecture functions correctly when the input spans are sorted in either direction, i.e., in the increasing order of start offsets or in the decreasing order of end offsets. However, one of the comparison units and the subsequent *and gate* (labeled *filter*) can be eliminated when the overall design guarantees that the input spans will always be sorted in a single direction.

V. Optimized Architecture

This section describes an optimized leftmost longest regex matching architecture that does not use a sorter. Note that although high-throughput FPGA-based sorter implementations exist [28], they incur a relatively high resource consumption compared with basic regex matcher implementations. Thus, eliminating the sorting operations is crucial for improving the scalability of the leftmost longest regex matching architecture.

We exploit two key properties to eliminate the redundancies that exist in the baseline architecture. Firstly, based on Corollary 1, we observe that to locate the leftmost longest matches, it is not necessary to generate all possible regex matches. It is sufficient to produce the set of leftmost matches and look for the leftmost longest matches inside this set.

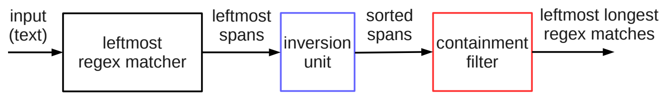


Fig. 6. Leftmost longest regex matching without sorting.

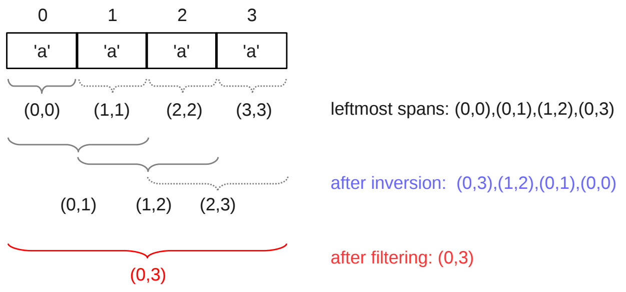

0
1
2
3
'a'
'a'
'a'
'a'
(0,0)
(1,1)
(2,2)
(3,3)
leftmost spans: (0,0),(0,1),(1,2),(0,3)
after inversion: (0,3),(1,2),(0,1),(0,0)
(0,1)
(1,2)
(2,3)
after filtering: (0,3)
(0,3)

Fig. 7. The steps that lead to leftmost and leftmost longest regex matches.

Secondly, the existing architectures that support leftmost regex matching [13], [14] produce partially sorted results. These architectures process the input stream character by character, and at each end offset position that results in a regex match, they report only the regex match with the smallest start offset, i.e., the leftmost regex match. That approach guarantees that the regex matcher will produce the start and end offset tuples in increasing order of the end offsets, and that for each end offset there will be only a single start offset. If we simply invert the order of the spans produced by such a leftmost regex matcher, the spans will already be sorted in decreasing order of the end offsets, and there will be no two spans that have the same end offset, but different start offsets. Therefore, inverting the order of the spans produced by the regex matcher creates a sorted stream of spans that can be consumed directly by a containment filter. Fig. 6 depicts our optimized architecture, and Fig. 7 shows its application to the example given in Fig. 2.

Consider again spans $(0, 1)$, $(2, 3)$, $(2, 4)$, and $(1, 5)$, which can be produced by a leftmost regex matcher. Note that the spans are produced in increasing order of the end offsets. After inverting the order of the spans, we get $(1, 5)$, $(2, 4)$, $(2, 3)$, and $(0, 1)$. The containment filter removes spans $(2, 4)$ and $(2, 3)$, and spans $(1, 5)$ and $(0, 1)$ remain as a result. This result is sorted in decreasing order of both start and end offsets. An optional inversion can be applied on this result to produce a final output that is sorted in increasing order of both start and end offsets, i.e., $(0, 1)$, and $(1, 5)$.

A. A Latency-Hiding Inversion Unit

The inversion unit can be implemented using simple LIFO buffers. However, it will not produce any output until the leftmost regex matcher produces an *eos* (end-of-stream) signal, and the latency of this can significantly reduce the throughput rate of the optimized architecture. To hide this latency, subsequent input streams (i.e., text documents in our setup) can be executed in a pipelined fashion within the processing chain shown in Fig. 6. As an example, as soon as the leftmost regex matcher produces the *eos* signal for stream i, the inversion unit can start accepting new spans associated with stream $i + 1$, while forwarding the inverted spans associated with stream i

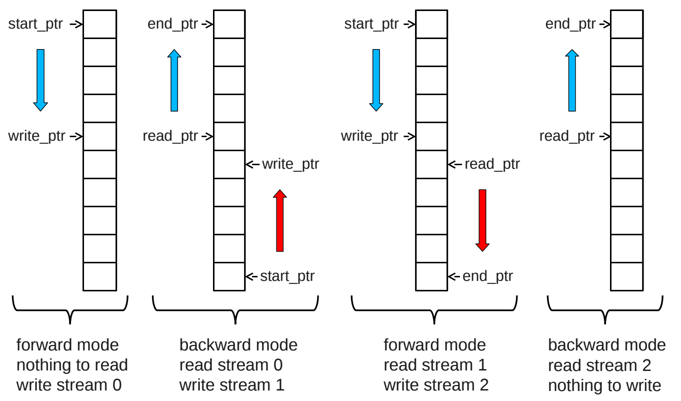


Fig. 8. A latency-hiding inversion unit: overlapping read and write latencies.

to the containment unit. Such an approach can be implemented without doubling the buffer space as shown in Fig. 8.

The architecture given in Fig. 8 alternates between *forward* and *backward mode*. A *start pointer* marks the start position of the data that is being written to the LIFO buffer, and a *write pointer* gets incremented (or decremented, depending on the mode) when a new span arrives from the leftmost regex matcher. The start pointer and the write pointer values are initially equal, and both point to the same end of the buffer space. The end position of the data that is being read from the LIFO buffer is marked by an *end pointer*, and a *read pointer* is decremented (or incremented, depending on the mode) when a new span gets transfered to the containment filter. When an *eos* signal arrives, the inversion unit 1) copies the write pointer into the read pointer and the start pointer into the end pointer, 2) switches from forward to backward mode (or vice versa), and 3) initializes the start pointer and the write pointer.

The architecture shown in Fig. 8 only produces back-pressure (i.e., stalls the processing pipeline) if the write pointer of stream $i+1$ is equal to the read pointer of stream i, which means that there is no room left for new entries in the allocated buffer space. However, this situation happens very rarely because the read buffer typically is flushed very quickly, and the write buffer receives new entries rather infrequently. The entries associated with stream i are transferred to the containment unit very quickly, e.g., at a rate of one span per clock cycle, whereas the leftmost regex matcher typically will not produce a new result per clock cycle for stream $i+1$.

VI. Computing Non-overlapping Matches

The architectures presented so far can produce leftmost longest regex matches that overlap. In this section, we therefore present modifications to these architectures that enable computation of non-overlapping leftmost longest matches.

Definition 5: Two spans $(s0, e0)$ and $(s1, e1)$ overlap if $(s1 > s0)$ and $(e1 > e0)$ and $(s1 \leq e0)$ or if $(s0 > s1)$ and $(e0 > e1)$ and $(s0 \leq e1)$.

Definition 6: The non-overlapping leftmost regex match at offset position i is the regex match with the smallest start offset that ends at position i and that does not overlap with any leftmost regex match that ends at a position smaller than i.

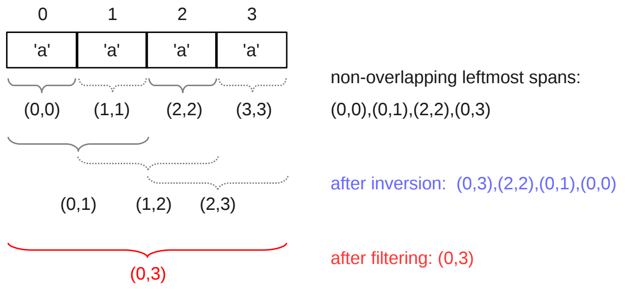


Fig. 9. Non-overlapping leftmost and leftmost longest regex matches.

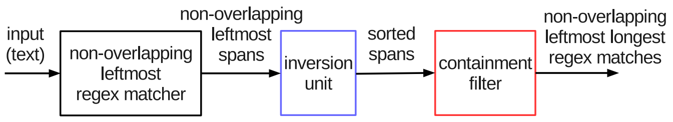


Fig. 10. Non-overlapping leftmost longest match computation without sorting.

Definition 7: A non-overlapping leftmost longest regex match is a non-overlapping leftmost regex match that is not contained in any other non-overlapping leftmost regex match.

One way of computing the non-overlapping leftmost longest matches is to produce all possible regex matches, sort the associated spans, and filter out overlapping spans. However, such a brute-force approach can be avoided by adapting the optimized architecture described in Section V to produce first the non-overlapping leftmost matches and then the non-overlapping leftmost longest matches. Note that the non-overlapping leftmost regex matches can be computed recursively based on Definition 6. Fig. 9 illustrates the non-overlapping leftmost regex matches for the example given in Fig. 2. Span $(0, 0)$ is the only regex match at offset position 0, and it does not overlap with any prior span. Thus, it is a non-overlapping leftmost match. There are two regex matches that end at offset position 1: $(0, 1)$ and $(1, 1)$. Note that based on Definition 6, $(0, 1)$ does not overlap with $(0, 0)$. Therefore, it is a non-overlapping leftmost match, and it suppresses $(1, 1)$. There are two regex matches that end at offset position 2: $(1, 2)$ and $(2, 2)$. $(1, 2)$ must be suppressed because, based on Definition 6, it overlaps with $(0, 1)$. Because $(1, 2)$ is suppressed, $(2, 2)$ does not get suppressed, and is reported as a non-overlapping leftmost regex match. Such a scheme can be implemented 1) by iteratively computing the leftmost matches starting from offset position 0 and incrementing the offset counter after each consumed character, and, in the case of a regex match that ends at offset position i and is associated with a span $(s0, i)$, 2) by deactivating all ongoing searches that have a start offset greater than $s0$.

Once an architecture that produces the non-overlapping leftmost regex matches is available, it can be combined with an inversion unit and a containment filter to produce the non-overlapping leftmost longest matches, as shown in Fig. 10.

VII. Computing Rightmost Longest Matches

The problem of rightmost longest matching arises when parsing the text from right to left rather than from left to right. We show that rightmost longest regex matches can be

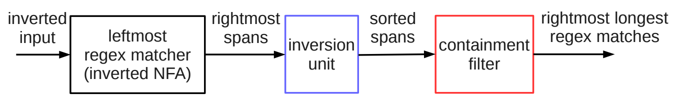


Fig. 11. Rightmost longest matching using leftmost longest matching.

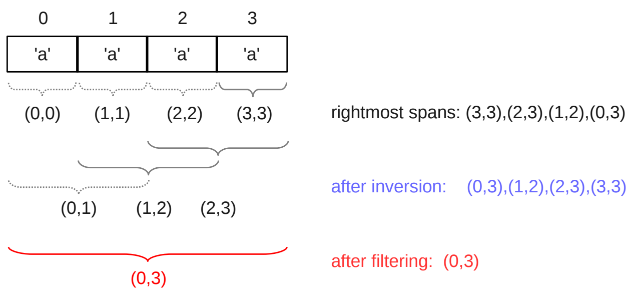

0
1
2
3
'a'
'a'
'a'
'a'
(0,0)
(1,1)
(2,2)
(3,3)
rightmost spans: (3,3),(2,3),(1,2),(0,3)
after inversion: (0,3),(1,2),(2,3),(3,3)
(0,1)
(1,2)
(2,3)
after filtering: (0,3)
(0,3)

Fig. 12. The steps that lead to rightmost and righmost longest regex matches.

computed using a leftmost longest regex matching architecture by making only a small change in the way that architecture operates.

Definition 8: The rightmost regex match at offset position i is the regex match with the largest end offset that starts at offset position i.

Definition 9: A rightmost longest regex match is a regex match that is not contained in any other regex match.

Corollary 2: A rightmost longest regex match is also a rightmost regex match.

Definition 10: The non-overlapping rightmost regex match at offset position i is the regex match with the largest end offset that starts at position i and does not overlap with a rightmost regex match starting at a position greater than i.

Definition 11: A non-overlapping rightmost longest regex match is a non-overlapping rightmost regex match that is not contained in any other non-overlapping rightmost regex match.

First, to produce the rightmost regex matches, the input text is parsed from right to left, i.e., the order in which the input text gets processed is inverted (see Fig. 11). Second, an inverted NFA is constructed using standard techniques. For instance, if the regex pattern is defined to be `abc`, the inverted NFA recognizes `cba`. We apply the inverted input text to a leftmost regex matcher that implements the inverted NFA of the regex. This combination produces the rightmost regex matches as shown in Fig. 12. These rightmost regex matches are fed into an inversion unit and then into a containment filter, as usual, to produce the rightmost longest regex matches.

The set of rightmost regex matches is not necessarily equal to the set of leftmost regex matches. However, the set of rightmost longest regex matches is always equal to the set of leftmost longest regex matches. In contrast, the set of non-overlapping rightmost longest regex matches does not have to be equal to the set of non-overlapping leftmost longest regex matches. Therefore, the choice of leftmost vs. rightmost matching and the choice of non-overlapping vs. overlapping matching can have a significant impact on the results produced. The architecture proposed in this work supports all resulting combinations very efficiently in field-programmable logic.

VIII. Experiments

Our designs were synthesized using the Quartus II 12.1 software, for an Altera Stratix IV GX530 FPGA. We used the maximum effort settings of the synthesis tool. The start-offset and the end-offset positions stored in the span data structures were 32 bits wide in our experiments.

The logic resource consumption of the inversion unit and the containment filter was minimal. These two units together consumed approx. 100 combinational ALUTs and approx. 90 dedicated registers (bits of storage), and achieved a clock frequency of approx. 300 MHz. Thus, based on the results given in [14], the clock frequency of the overall design is limited by the leftmost regex matcher. However, an overall clock frequency of 250 MHz appears to be realistic in most practical use cases. Furthermore, based on the results given in [14], the leftmost regex matching logic on average consumes around 300–500 ALUTs and registers for a given regex. Therefore, supporting leftmost longest regex matching on top of an architecture that supports leftmost regex matching results in a 20%–30% increase in logic resource consumption.

In our current design, each regex owns a dedicated inversion unit, and each inversion unit consumes two M9K blocks (256×36 bits each). However, it is possible to design inversion units that are shared across multiple regex engines. Also, our current design does not implement an external memory interface to deal with overflow conditions, but the 256-element-deep LIFO buffers are more than sufficient for typical text-analytics use cases. For instance, the size of Twitter messages does not exceed 140 bytes. Therefore, they can theoretically produce a maximum number of 140 leftmost or rightmost matches. Similarly, the size of news entries is typically in the range of 500 bytes to 4 kB. In such small documents, more than 256 regex matches will rarely be found. Note also that the target FPGA contains 1280 M9K blocks. Therefore, the proposed design can support up to 500–600 regexs with leftmost longest matching or rightmost longest matching support on the target FPGA using the M9K blocks.

Experiments were performed using a regex set with 25 regexs from the text analytics domain. The software throughput measurements were obtained using a scale-out version of the SystemT text analytics library [3] on a 12-core Intel® Xeon® E5-2630 processor, running at 2.6 GHz. Note that the SystemT library is implemented in Java®, and its regular expression matcher uses a backtracking-based algorithm to implement the leftmost longest matching semantics. Fig. 13 shows that the throughput rates achieved by the software library is limited to a few MB/s for the regex set evaluated. We observe that the software throughput rate does not improve beyond 16 threads, and the highest throughput rate achieved by the software library is around 4.3 MB/s.

In contrast, a single instance of our FPGA-based leftmost longest regex matcher can achieve a throughput rate of up to 250 MB/s because our design is clocked at 250 MHz and consumes a single input byte per clock cycle. Multiple instances (threads) of our design can be instantiated on the target FPGA to process multiple independent text documents

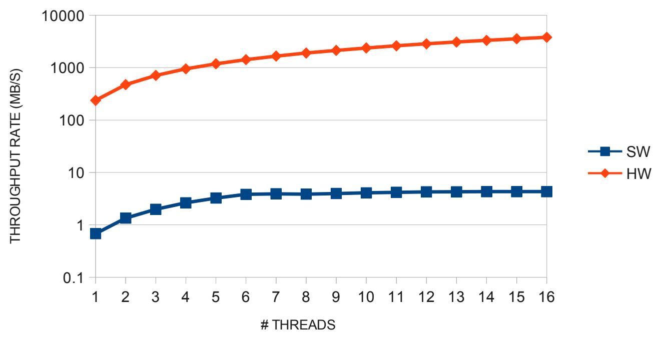


Fig. 13. Comparing software (SW) and hardware (HW) throughput rates.

in parallel. When instantiating four hardware threads, we measured a throughput rate of 0.95 GB/s when using the FPGA, which translates into a more than 200-fold improvement of the processing rates with respect to the SW implementation. Assuming that a 4 GB/s bus interface is available, a simple scaling of the resource consumption figures implies that up to 16 hardware threads can be accomodated on the target FPGA. This translates into a more than 800-fold improvement of the processing rates with respect to the software implementation.

IX. Summary

This paper provides novel theoretical and practical results for improving the efficiency of leftmost longest regex matching on field-programmable devices. First, we present a baseline architecture that relies on sorting and filtering to compute the leftmost longest regex matches. Then, we present an optimized architecture that eliminates the costly sorting operations. We also cover extensions of our optimized architecture to support computation of non-overlapping matches and rightmost longest regex matches. Our experiments on a regex set from the text analytics domain demonstrate a more than 200-fold improvement of the processing rates compared with the multithreaded reference software implementation. Our current and future work includes extending our optimized architecture to support external memory interfaces and shared LIFO buffers, and evaluating the energy-efficiency of our hardware accelerators.

Acknowledgements

I would like to thank C. Bolliger from IBM Research - Zurich for her language-related corrections and comments, and F. R. Reis, L. Chiticariu, and H. Zhu from IBM Research - Almaden for the technical discussions that led to this work.

References

[1] T. Agerwala and M. Perone. Data centric systems: The next paradigm in computing. Keynote at the International Conference on Parallel Processing, Sep. 2014.

[2] J. G. Koomey. Growth in data center electricity use 2005 to 2010. A report by Analytics Press, source: http://www. analytics-press.com/datacenters.html, Aug. 2011.

[3] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: A system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[4] W.-D. Zhu *et al. IBM Watson Content Analytics: Discovering Actionable Insight from Your Content*. IBM Redbooks, 2014.

[5] R. Polig *et al.* Giving text analytics a boost. *IEEE Micro*, 34(4):6–14, July 2014.

[6] K. Asanovic *et al.* A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[7] A. Putnam *et al.* A reconfigurable fabric for accelerating large-scale datacenter services. In *Proc. ISCA*, pages 13–24, June 2014.

[8] R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *Proc. FCCM '01*, pages 227–238, 2001.

[9] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *Proc. ANCS*, pages 30–39, 2008.

[10] I. Sourdis, J. Bispo, J. M. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *J. Signal Process. Syst.*, 51(1):99–121, 2008.

[11] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proc. ANCS*, pages 155–164, 2007.

[12] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *Proc. SIGCOMM '08*, pages 207–218, 2008.

[13] K. Atasu, R. Polig, C. Hagleitner, and F. R. Reiss. Hardware-accelerated regular expression matching for high throughput text analytics. In *Proc. FPL*, pages 289–295, 2013.

[14] K. Atasu. Resource-efficient regular expression matching for text analytics. In *Proc. ASAP*, pages 1–8, 2014.

[15] K. Atasu. Feature-rich regular expression matching accelerator for text analytics. In *Journal of Signal Processing Systems*, to appear, 2015.

[16] R. Polig, K. Atasu, H. Giefers, and L. Chiticariu. Compiling text analytics queries to fpgas. In *Proc. FPL*, pages 427–432, 2014.

[17] C. L. A. Clarke and G. V. Cormack. On the use of regular expressions for searching text. *ACM Trans. Program. Lang. Syst.*, 19(3):413–426, May 1997.

[18] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38:43–56, 1995.

[19] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Spanners: A formal framework for information extraction. In *Proc. 32nd Symposium on Principles of Database Systems, PODS '13*, pages 37–48, 2013.

[20] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.

[21] Z. K. Baker and V. K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *Proc. FCCM '04*, pages 135–144, 2004.

[22] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang. Optimization of pattern matching circuits for regular expression on FPGA. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(12):1303–1310, 2007.

[23] N. Yamagaki, R. P. S. Sidhu, and S. Kamiya. High-speed regular expression matching engine using multi-character NFA. In *Proc. FPL*, pages 131–136, 2008.

[24] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. CoNEXT*, 2007.

[25] Y.-H. E. Yang and V. K. Prasanna. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *Proc. INFOCOM*, pages 1853–1861, 2011.

[26] H. Nakahara, T. Sasao, and M. Matsuura. A regular expression matching circuit based on a decomposed automaton. In *Proc. ARC*, pages 16–28, 2011.

[27] D. Pao, N. Lam Or, and R. C. C. Cheung. A memory-based NFA regular expression match engine for signature-based intrusion detection. *Computer Commun.*, 36(10-11):1255–1267, 2013.

[28] D. Koch and J. Torresen. FPGAsort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proc. FPGA*, pages 45–54, 2011.