Parallel Interval Stabbing on the Automata Processor

Indranil Roy
Micron Technology, Inc.
8000 S Federal Way
Boise, ID 83716, USA
Email: iroy@micron.com

Ankit Srivastava
School of Computational
Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: asrivast@gatech.edu

Matt Grimm
Micron Technology, Inc.
8000 S Federal Way
Boise, ID 83716, USA
Email: mgrimm@micron.com

Srinivas Aluru
School of Computational
Science and Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332
Email: aluru@cc.gatech.edu

***Abstract*—The Automata Processor was designed for string-pattern matching. In this paper, we showcase its use to execute integer and floating-point comparisons and apply the same to accelerate *interval stabbing queries*. An interval stabbing query determines which of the intervals in a set overlap a query point. Such queries are often used in computational geometry, pattern matching, database management systems, and geographic information systems. The check for each interval is programmed as a single automaton and multiple automata are executed in parallel to provide significant performance gains. While handling 32-bit integers or single-precision floating-point numbers, up to 2.75 trillion comparisons can be executed per second, whereas 0.79 trillion comparisons per second can be completed for 64-bit integers or double-precision floating-point numbers. Additionally, our solution leaves the intervals in the set unordered; allowing addition or deletion of an interval in constant time. This is not possible for contemporary solutions wherein the intervals are ordered, making the query times faster, but making the updating of intervals complex. Our automata designs exemplify techniques that maximize resource utilization and minimize performance bottlenecks, which may be useful to future application developers on this processor. Their modular design allows them to become constituent parts of larger automata, where the numerical comparisons are part of the overall pattern matching operation. We have validated the designs on hardware, and the routines to generate the necessary automata and execute them on the AP will be made available as software libraries shortly.**

I. Introduction

An *interval stabbing* query identifies the intervals from a list that overlap (or are *stabbed* by) a query point. These queries find usage in a variety of applications such as computational geometry, pattern matching, database management systems, geographic information systems, algorithmic trading, etc. If the intervals are overlapping, state-of-the-art algorithms order the intervals into search-trees and use these trees to generate the solution. However, the runtime of any algorithm is output sensitive, i.e. its run-time complexity is at least dependent on the number of the intervals in the output. Therefore, in the worst case, the runtime may degrade to the order of the number of intervals in the list. Additionally, any modification to the list may entail significant changes to the search-tree and hence may incur severe overheads.

IA^3 2016; Salt Lake City, Utah, USA; November 2016
978-1-5090-3867-1/16/$31.00

The interval stabbing problem falls under a broader category of applications which depend on the execution of one-to-many comparisons quickly. The more such comparisons can be executed in parallel, the less ordered the list needs to be, thereby allowing more expeditious modifications to the list. Towards this end, we present an accelerated solution to the interval stabbing problem, using the Automata Processor (AP) [1], [7]. The AP is a reconfigurable coprocessor which can be programmed to compute a large set of user-defined Nondeterministic Finite Automata (NFAs) in parallel against a single query data-stream. We define one automaton for each interval in the list and execute tens of thousands of such automata in parallel against the query points streamed to the processor.

Our streaming solution has a variety of advantages. First, the fine grained parallelism allows significant acceleration over existing solutions. Second, the addition or deletion of an interval to the unordered list is a constant time operation. And third, the streaming solution is amenable to applications where the query point is embedded in a stream of data. In contemporary solutions, this query point has to be lexically parsed out of the stream before the interval stabbing query can be issued. However, our modular designs can be part of other larger NFA structures which allow the parsing of the data and the answering of the query simultaneously.

Another novelty of our solution is that this is the first known use of the AP for multi-byte integer and floating-point operations in their native formats. Hitherto, the processor has been typically used for string pattern analysis [15], [18], [21], [23] or for graph analysis [16], by conversion to strings. The solution outlined in this paper extends the scope of use of the processor beyond its design intent. Additionally, the automata designs employed by this paper exemplify design techniques which maximize on-board resource utilization and minimize various compile-time and runtime overheads. These design techniques will be useful to anyone developing solutions on this processor.

In this paper, we have outlined the automata designs for 4-byte and 8-byte integers, and single and double precision IEEE floating-point numbers. However, the applicability of this method extends to a variety of other datatypes, e.g. fixed

or variable length strings, multi-dimensional coordinate points, and datestamps or timestamps. Notice that, in the last case, the query point and endpoints may not be expressed in the same format. Therefore, by expressing the endpoints as regular expressions, the scope of acceleration on the AP is amplified. We have also presented insights into our current research regarding potential ways of handling these datatypes.

The rest of the paper is organized as follows. First, we explore the state-of-the-art solutions for the interval stabbing problem in Section II-A. Next, we provide a basic description of the AP architecture in Section II-B, which is required to understand our AP based solution described in Section III. We present results from execution in hardware in Section IV and describe the scope of future work to handle other data formats including higher-dimensional points, variable-length string representation of numbers, or even multiple-format timestamps in Section V. Finally, we conclude in Section VI.

II. Background

A. Interval Stabbing

Identifying intervals from a set $I = \{i_1, i_2, \ldots, i_n\}$ which overlap with a query point q is a well studied problem in computational science. If the intervals are non-overlapping, then they can be sorted based on their starting points in $O(n \log n)$ time, the sorted list can be stored in $O(n)$ space, and the queries answered in $O(\log n)$ time. However, if the intervals are arbitrarily overlapping, then the solutions are more involved. We briefly discuss these below.

A brute-force linear search through all the intervals takes $O(n)$ time, which is asymptotically optimal as all the intervals may be stabbed by q. Nonetheless, output-sensitive algorithms have been developed which employ various data-structures to organize the intervals and answer queries in $O(\log n + k)$ time, where k is the number of intervals reported by the query.

One commonly used data-structure is called *interval trees* [8]–[10], [13]. Each node in an interval tree corresponds to a *center point*. All the intervals whose ending point is smaller than the center point are captured in the subtree rooted at the left child of the node, whereas all the intervals whose starting point is greater than the center point are captured in the subtree rooted at its right child. All the intervals stabbed by the center point are captured within the node, and sorted separately using their starting and ending points. Creation of this tree requires $O(n \log n)$ time, a storage complexity of $O(n)$, and an expected run-time of $O(\log n + k)$ per query.

Segment trees [3] are similar to interval trees, and identical in performance except in the storage complexity of $O(n \log n)$. However, the intervals within a node need not be sorted in any order. Schmidt [19] describes a faster data-structure to complete preprocessing in $O(n)$ time and a query in $O(1+k)$ time when the end points of the intervals are within a small integer range, e.g. $\{1, 2, \ldots, O(n)\}$.

All the data-structures described above are generally used in a static setting because addition and deletion of an interval is complex with difficult to estimate run-time. Cormen et al. [5] describe a modification to the interval trees called *augmented trees* wherein the insertion or deletion of an interval can be completed with $O(h)$ complexity, where h is the height of the tree. Another search-tree called *priority tree* [14] requires $O(n)$ space and supports insertion and deletion of intervals. However, the priority search-tree is very complex to implement, especially in its balanced form.

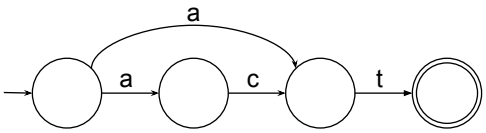
Hanson uses *interval skip list* [12] which is simpler to implement than interval trees and takes an expected time of $O(\log^2 n)$ to insert or delete an interval. Alstrup [2] describes another method to represent each interval as a coordinate in an $n \times n$ matrix, and then iteratively compute the nearest common ancestors in a *cartesian tree* (as shown by Gabow [11]) in $O(1 + k)$ time. However, this procedure has a significant implementation overhead.

B. Automata Processor Basics

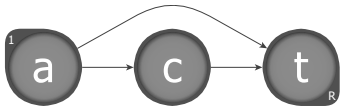
The AP is a Multiple Instruction Single Data (MISD) device which can propagate one symbol from the query data stream to every processing element in the chip on every clock-cycle. Each symbol is a 1-byte word. The multiple instructions are user-defined state machines, realized through connecting the processing elements using a reconfigurable routing network. Since all the processing elements receive every data symbol and all the routing lines can be activated in parallel, the state machines emulate NFAs in hardware which execute in parallel. The intrinsic parallelism in the hardware absolves the programmer from dealing with communication delays, race conditions, etc. and concentrate merely on the design of the NFAs. Given the novelty of the processor, we now briefly describe the processing elements, routing matrix, programming environment, and programmable resources of the architecture.

1) Processing Elements: The representation of automata to be executed on the AP is slightly different from the classical state-diagram representation. We will henceforth refer to the former as ANML-NFA because they are defined using Automata Network Markup Language (ANML, pronounced as animal). The basic processing element is called a State Transition Element (STE) which emulates an edge transition in a traditional state-diagram. The label of the STE is the same as the label of the edge transition and can be any character-class from the 1-byte symbol space. Routing lines between these STEs represent states in a classical state-diagram. All the STEs representing incoming edges into a state are connected to the STEs representing all the outgoing edges of that state. All the STEs representing the outgoing edges of the start state are represented as *start-STE*s and all the incoming edges of the accept states are marked as *reporting-STE*s. More details on converting classical state-diagram to ANML-NFA can be found in [17].

The state-diagram and the ANML-NFA representation of an NFA that accepts words act and at is shown in Fig. 1a and Fig. 1b. The start-STE is shown with an indicator on the top-left corner with the number 1 in the indicator. The reporting-STE has an indicator to the bottom-right with the symbol R placed inside it.



(a) state-diagram



(b) ANML-NFA

Fig. 1: Automaton to accept words at or act

Before the processing of the data-stream can begin, the user-defined automata have to be compiled and loaded onto the processor. Although the compilation time is dependent on the automata and may take anywhere between seconds to hours, all the compiled automata can be loaded in 50 milliseconds. Once the board has been configured, it is ready to process the data-stream. During the first clock-cycle, only the start-STEs are active and the first byte of the data-stream is broadcasted. If the broadcasted symbol belongs to the character-class stored in the label of an active STE, then all the STEs connected to its outgoing routing lines are activated for the next clock-cycle. In the next clock-cycle, the subsequent byte in the data-stream is broadcasted and the process continues. If a reporting-STE is matched in a cycle, an output is generated identifying the STE and the offset in the data-stream where the match occurred.

2) Programming Resources: The programming elements in the AP are arranged hierarchically as follows. 16 STEs are arranged in a *row*, 16 rows in a *block*, 96 blocks in a *half-core*, and 2 half-cores in a chip. Cumulatively, each chip has a total of $49,152$ STEs. All the STEs in a row can be simultaneously connected to each other, while only 24 routing lines are present to connect the STEs in different rows within a block. STEs within a block can only be connected to the STEs belonging to 8 adjacent blocks. There are no connections between the STEs of the two half-cores.

The AP-compiler completely abstracts the underlying layout while placing the processing elements in user designs to the physical elements on the chip. Therefore, an automata designer may design automata in a completely layout-agnostic manner. However, experienced designers may consider the hierarchical layout while coming up with designs which can be placed by the compiler with higher resource utilization efficiency.

3) Processing Rate: An AP-chip functions at 128 MHz, processing a 1-byte symbol per cycle, thus supporting an input data streaming rate of 1 Gbps. There are 32 chips on a single AP-board which can be organized into *logical-cores* of 2, 4, or 8 chips. All the chips within a logical-core are presented the same data-stream. Separate logical-cores can process different data-streams concurrently. This provides the flexibility of executing a large number of NFAs against a single data-stream using bigger logical-cores, or smaller number of NFAs using smaller logical-cores for a higher combined throughput of up to 16 Gbps.

Output Handling Bottleneck: On the current generation of the chip, the output handling may significantly slow down the overall processing rate. Whenever reporting-STEs generate output in a cycle, an output-vector is created and stored in an output buffer. If the output buffer is sufficiently empty, the vector is stored in the same clock-cycle and the input processing continues unabated from the next cycle. Simultaneously, the output-vector is read out from the output buffer to the main memory of the host processor. This may take between 135 to 494 clock-cycles for each vector. Therefore, if the vectors are generated too frequently, then the output buffer fills up and the input processing has to be stalled till the buffer has been sufficiently emptied for the new output-vector. However, one aspect of the output handling often exploited by many applications [15], [20], [22], [23] is that the output processing rate is not determined by the number of outputs generated, but the number of clock-cycles on which they were generated. Therefore, these applications overcome this bottleneck by batching as many outputs into a single clock-cycle as possible.

III. Methodology

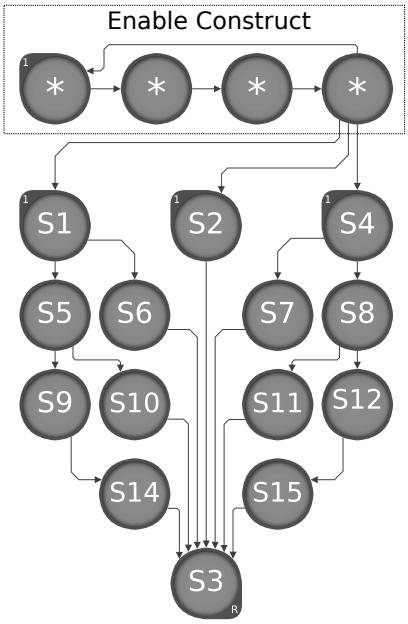
A. Automata Design

In this section, we present our automata designs for checking if the interval $[x, y]$ is overlapped by a query point z. The points x, y, and z are numbers represented as 4-byte or 8-byte integers and floating-point numbers. The numbers can be signed or unsigned. For the sake of brevity, our examples only illustrate the case of closed intervals. Semi-open or open intervals can be handled using the same automata structures by modifying the labeling schemes slightly.

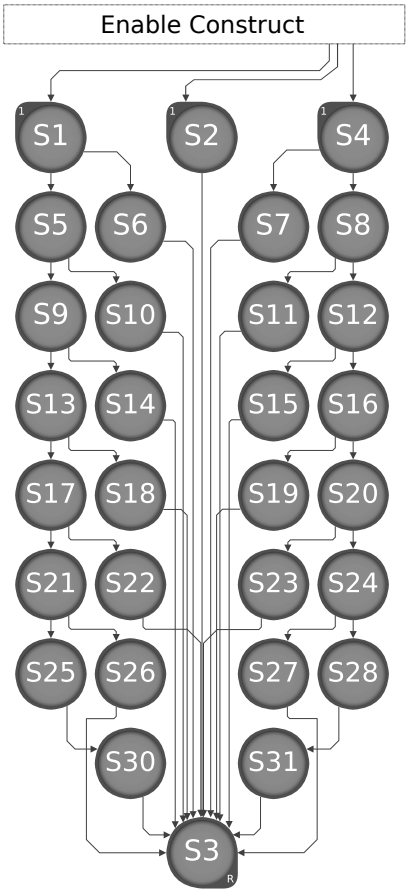
Comparison of numbers represented in binary, using the AP, poses two major challenges. First, as discussed in Section II-B3, the AP processes the input one byte at a time. Therefore, multi-byte numbers have to be compared byte-by-byte. Second, prevalent methods of representing signed binary numbers, such as the *two's complement* form, place negative numbers in a higher lexicographical order than their positive counterparts.

We designed the automata as macros with parameterized labels for the STEs. These macros can be pre-compiled and replicated with different labels for different intervals quickly. The macros for 4-byte and 8-byte numbers are shown in Fig. 2a and Fig. 2b. A b-byte comparator macro contains $4(b-1)+2$ parameterized STEs, which are tagged $S1$ to $S < 4(b-1)+3 >$, with $S < 4(b-1)+1 >$ missing. This tagging scheme simplifies the labeling algorithm. In our figures, the tags are shown inside the STEs.

Although we limit our discussion to 4-byte and 8-byte numbers here, the methodology described is generic and extendable to any b-byte number. Our designs also assume the little-endian representation, i.e. the bytes in the number are streamed from the most significant byte (MSB) to the least significant byte (LSB).



(a) 4-byte comparator macro



(b) 8-byte comparator macro

Fig. 2: Macros for comparing binary representation of numbers

The automata designs work as follows. Automata for the intervals is programmed on the AP and all the query points are concatenated together to form a single query data-stream, which is then streamed to the processor. Each automaton processes one query point at a time and multiple query points in succession. The host application uses the generated reports, and the corresponding offset in the data-stream, to determine which interval(s) are stabbed by each query point.

The algorithms for assigning labels to STEs for intervals and queries of unsigned integers, signed integers, and floating-point numbers are described below.

1) Unsigned Integers: We describe the 4-byte macro first. The *Enable Construct* in the 4-byte macro simply activates $S1$, $S2$, and $S4$ on every fourth byte, after the first byte in the data-stream. This allows these STEs to process the first byte of every query number in the stream. The rest of the macro can be visualized as follows. Each row of STEs in the macro processes one byte of the query number. Since all the unsigned integers are lexicographically ordered, z can be determined to be in the interval $[x, y]$ by comparing the bytes of z to the corresponding bytes of x and y from MSB to LSB. The STEs in the leftmost column are activated successively if the consecutive bytes of z match the corresponding bytes of x. Similarly, the STEs in the rightmost column are activated if the consecutive bytes of z match the bytes of y. If z is determined to be stabbing the interval based on the current byte being processed, then the STE in either the second from the left or the second from the right column, or both, activate $S3$. $S3$ is programmed to generate an output on any input byte, signaling that z overlaps the interval. The same concepts are used to extend the macro to handle 8-byte integers using more rows, and making the Enable Construct activate $S1$, $S2$, and $S4$ after every eight bytes.

Algorithm 1 is used to label the STEs for unsigned integer intervals. In the following discussion, we refer to the j^{th} most significant byte of x, y, and z as x_j, y_j, and z_j, respectively. $S2$ is labeled for matching all the bytes in the open interval (x_1, y_1) and, therefore, checks if z_1 is greater than x_1 and less than y_1. If yes, z overlaps the interval and is reported via $S3$. If $z_1 \notin (x_1, y_1)$, then z_1 must be equal to x_1 or y_1 for z to overlap the interval. These cases are handled by $S1$ and $S4$ respectively. If matched, $S1$ or $S4$ activates the connected STEs in the second row for comparing z_2.

$S6$ and $S7$ handle the case where z can be determined to overlap (x, y) based on z_2. If $x_1 = z_1 = y_1$, then $S6$ and $S7$ are labeled to accept any z_2 in the range (x_2, y_2). Otherwise, if $x_1 = z_1 < y_1$, then z_2 can assume any value larger than x_2 and hence $S6$ is labeled as $(x_2, 255]$. By similar logic, $S7$ is labeled to match the range $[0, y_2)$. If z_2 is matched by $S6$ or $S7$, then $z \in (x, y)$, and $S3$ is activated to generate an output in the next cycle. Otherwise, $S5$ and $S8$ handle the case wherein $z_2 = x_2$ and $z_2 = y_2$ respectively and activate the STEs in the next row to process z_3 in an identical fashion. Finally, $S14$ and $S15$ are labeled based on x_4 and y_4 using logic similar to the labeling of $S6$ and $S7$. However, since they process the last byte of z, the check for equality with x_4 and y_4 is also rolled into these STEs.

Notice that, our macro designs may report a match in the cycle after the last number finishes streaming, because a 4-

Algorithm 1 Labeling STEs for unsigned integer intervals

Input:
- Binary representation of unsigned integers x and y. x_j and y_j denote the j^{th} MSB of x and y.
- Number of bytes in the representation of x and y, b.
- Labels of all the STEs in the b-byte comparator, L. L_t denotes the label of the STE tagged $S < t >$.

Ensure:
- $x \leq y$

```
1:  procedure LABELUNSIGNED(x, y, b, L)
2:      L_2 ← {(x_1, y_1)}
3:      L_3 ← {*}
4:      equalPrefix ← TRUE
5:      for j = 1 to b − 2 do
6:          L_{4(j−1)+1} ← {x_j}
7:          L_{4(j−1)+4} ← {y_j}
8:          if x_j ≠ y_j then
9:              equalPrefix ← FALSE
10:         end if
11:         if equalPrefix == TRUE then
                // x_{j+1} ≤ y_{j+1}, since x ≤ y and
                // x_k == y_k for all k ≤ j.
12:             L_{4j+2} ← {(x_{j+1}, y_{j+1})}
13:             L_{4j+3} ← {(x_{j+1}, y_{j+1})}
14:         else
15:             L_{4j+2} ← {(x_{j+1}, 255]}
16:             L_{4j+3} ← {[0, y_{j+1})}
17:         end if
18:     end for
19:     L_{4(b−2)+1} ← {x_{b−1}}
20:     L_{4(b−2)+4} ← {y_{b−1}}
21:     if equalPrefix == TRUE then
22:         L_{4(b−1)+2} ← {[x_b, y_b]}
23:         L_{4(b−1)+3} ← {[x_b, y_b]}
24:     else
25:         L_{4(b−1)+2} ← {[x_b, 255]}
26:         L_{4(b−1)+3} ← {[0, y_b]}
27:     end if
28: end procedure
```

byte number can be reported in the corresponding 5^{th} cycle. Therefore, the input stream should be padded at the end, with a dummy byte, to ensure that all the intervals report overlaps for the last query point. Since a report can not be generated after processing just one byte of the number, this padding will not generate any spurious output.

2) Signed Integers: We have used the *two's complement* method of signed integer representation in our labeling algorithm as it is the most prevalent method. Alternate, but similar, schemes may be adopted for other methods of representation. In the two's complement representation, the lower lexicographical half of the range is reserved for all the non-negative integers, whereas the upper half is used to store negative integers. However, the relative ordering between any two negative integers or any two non-negative integers is maintained. We use this property in our labeling algorithm for signed integers described in Algorithm 2.

Algorithm 2 Labeling STEs for signed integer intervals

Input:
- Two's complement binary representation of signed integers x and y. x_j and y_j denote the j^{th} MSB of x and y.
- Number of bytes in the representation of x and y, b.
- Labels of all the STEs in the b-byte comparator, L. L_t denotes the label of the STE tagged $S < t >$.

Ensure:
- $x \leq y$

```
1:  procedure LABELSIGNED(x, y, b, L)
2:      if (x_1 ≤ 127 and y_1 ≤ 127) or
           (x_1 > 127 and y_1 > 127) then
            // x and y are either both
            // non-negative or both negative.
3:          LABELUNSIGNED(x, y, b, L)
4:      else
            // x is negative and y is
            // non-negative, since x ≤ y.
5:          L_2 ← {(x_1, 255], [0, y_1)}
6:          L_3 ← {*}
7:          for j = 1 to b − 2 do
8:              L_{4(j−1)+1} ← {x_j}
9:              L_{4(j−1)+4} ← {y_j}
10:             L_{4j+2} ← {(x_{j+1}, 255]}
11:             L_{4j+3} ← {[0, y_{j+1})}
12:         end for
13:         L_{4(b−2)+1} ← {x_{b−1}}
14:         L_{4(b−2)+4} ← {y_{b−1}}
15:         L_{4(b−1)+2} ← {[x_b, 255]}
16:         L_{4(b−1)+3} ← {[0, y_b]}
17:     end if
18: end procedure
```

If the interval $[x, y]$ is either fully negative or fully non-negative, the comparator for the interval is labeled as unsigned integers, using the procedure described in Algorithm 1. In the only other case, i.e. when the interval $[x, y]$ is part negative and part non-negative, x must be negative and y must be non-negative. Hence $x_1 \neq y_1$. Recall that, if $x_1 \neq y_1$ in the algorithm for labeling unsigned integers, the labels of the STEs in the two columns on the left do not depend on the value of y and the labels of the STEs in the two columns on the right do not depend on the value of x. Therefore, we can again label all the STEs as unsigned integers, except for $S2$. $S2$ activates $S3$ only if processing z_1 ensures that z is in the interval (x, y). In the case of part negative and part non-negative interval, z_1 should either be in the interval $(x_1, 255]$ or in the interval $[0, y_1)$ for z to be reported.

3) Floating-Point Numbers: IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) [4] describes the represen-

Algorithm 3 Labeling STEs for floating-point intervals

Input:

- IEEE standard binary representation of floating-point numbers x and y.
  x_j and y_j denote the j^{th} MSB of x and y.
- Number of bytes in the representation of x and y, b.
- Labels of all the STEs in the b-byte comparator, L.
  L_t denotes the label of the STE tagged $S < t >$.

Ensure:

- $x \leq y$

```
1:  procedure LABELFLOATS(x, y, b, L)
2:      L_3 ← {*}
3:      if (x_1 ≤ 127 and y_1 ≤ 127) then
4:          LABELUNSIGNED(x, y, b, L)
5:      else if (x_1 > 127 and y_1 > 127) then
            // If x and y are both negative,
            // interchange x and y.
6:          LABELUNSIGNED(y, x, b, L)
7:      else
            // If x and y have different
            // signs, use two intervals.
8:          L' ← duplicate(L)
9:          LABELFLOATS(x, −0, b, L)
10:         LABELFLOATS(+0, y, b, L')
11:     end if
12: end procedure
```

tation of fractional numbers in binary. In this representation, the first bit is the sign bit, followed by a standard-defined number of *exponent* and *trailing significand* bits. This representation of floating-point numbers can be interpreted as *sign-magnitude* representation of signed integers for the purpose of comparison [6]. In the sign-magnitude representation, the first bit (sign bit) is 1 for negative numbers and 0 for positive numbers. The other bits of the number determine the magnitude of the number. As in the two's complement representation, the positive numbers are represented lexicographically in the lower half of the range, and the negative numbers are shifted to the upper half. Unlike the two's complement representation, the ordering of the negative numbers is reversed.

Algorithm 3 labels the STEs for floating-point intervals as follows. If both x and y are non-negative, i.e. their sign bit is 0, then the labeling is identical to unsigned integers. On the other hand, if the sign bit is 1 for both, i.e. the interval is fully negative, then x and y are interchanged since the lexicographic ordering of negative numbers is reversed. The STEs of the comparator are again labeled using the procedure for labeling unsigned integers. However, if the upper bound and the lower bound have different sign bits, then the interval $[x, y]$ is broken into two intervals: $[x, -0]$ and $[+0, y]$, where -0 and $+0$ denote the two representations of zero with sign bit set to 1 and 0, respectively. The two intervals can then be programmed as discussed earlier and z is reported to overlap the complete interval if it stabs any of the two partial intervals.

4) Reducing Output Frequency: Notice that, the macro designs shown in Fig. 2 can generate a report in any cycle after the first byte of a query number has streamed. Therefore, when a query data-stream is checked against multiple intervals programmed on the processor, a report can potentially be generated in every cycle after the first. As discussed in Section II-B3, this can lead to stalls, thus lowering the overall processing rate. We modified our automata designs so that a query point is reported to overlap an interval exactly after all the bytes of the query number have streamed. This reduced the output generation frequency to a maximum of once every b cycles when comparing b-byte numbers.

The updated design for the 4-byte macro in Fig. 2a is shown in Fig. 3. As described in Section III-A1, in the original automaton design, $S3$ is activated to generate an output in the subsequent cycle after a query point is determined to overlap an interval. However, for ensuring that the match is reported exactly after processing the fourth byte, we added two STEs in the central column and labeled them to match all the 1-byte symbols. These STEs process the remaining bytes of the query number, after it is determined to overlap the interval, and activate $S3$ for generating a report after processing the last byte. Further, in the original design, any matches determined by the STEs labeled $S14$ or $S15$ are reported by $S3$ while processing the first byte of the next query in the stream. For handling this case, we made $S14$ and $S15$ reporting-STEs, and removed the connections between these two STEs and $S3$.

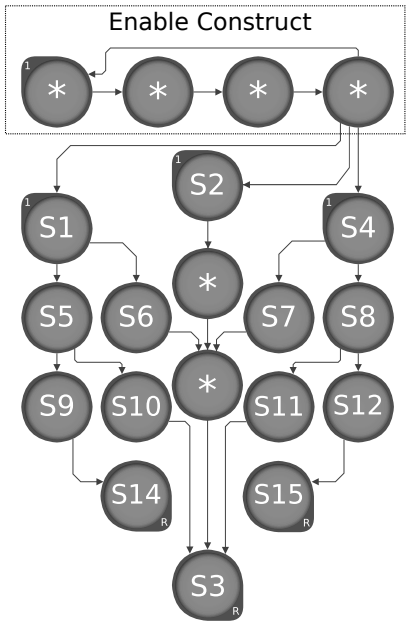


Fig. 3: Modified 4-byte macro for reducing output generation frequency

B. Analysis

Creating and loading the automata for all the intervals requires a preprocessing time of $O(n)$ and a storage complexity of $O(n)$. Addition or deletion of an interval can be done incrementally with constant overhead, a significant advantage

over contemporary methods. For each query, the presence of overlapping intervals can be ascertained in $O(1)$ time, but identifying them requires $O(k)$ time. Additionally, in many streaming applications, the query point is embedded within the data-stream.

Within the limits of the AP-board, the intervals, in the form of their corresponding automata, can be stored in any arbitrary order. If the number of intervals are larger than what can be fit inside an AP-board, then they can be partitioned into buckets and handled iteratively. The choice of which bucket an interval is placed in can be made using one of the end points. The ordering of intervals within a bucket is arbitrary, and irrelevant to the determination of query time and storage complexity.

In spite of the theoretical advantages mentioned above, programming a board incurs substantial latency which cannot be amortized through the performance gained using a single query. Therefore, the methodology described here best serves applications where a large number of stabbing queries are serviced with high throughput.

IV. Results

The AP is in advanced stages of qualification and production. The authors have access to pre-production prototypes which were used to evaluate the performance reported in this paper. To the best of our knowledge, these are the first hardware results to be published for the processor.

We used the Python API provided with the AP-SDK for creating the macros, combining them into ANML-NFA networks, compilation of ANML-NFA, and substitution of the labels. The designs were validated using the simulator provided with the AP-SDK.

Comparing the AP implementation of the interval stabbing problem to the state-of-the-art CPU-based implementations is unfair because of the following reasons. We are using AP-board hardware and software which are in *bring-up* phase. There are known overheads which slow down the processing by a factor of 20 times or more than that of the production hardware and software. The current generation of the chip itself is fabricated using *memory process* technology which is a couple of generations behind the state-of-the-art. The next generation of the chip, currently in design, is supposed to be orders of magnitude faster than the current chip. Finally, using the interval stabbing problem as a generic representation of a class of problems requiring comparison of a query element against an unordered list of other elements is not accurate. As discussed in Section II-A, the list in the interval stabbing problem can be partially ordered using *interval trees*, *segment trees*, etc. The run-time of algorithms using these structures are output-sensitive in $O(log\, n + k)$ time, where n is the total number of intervals, and k is the number of intervals *stabbed* by the query. When k is small, CPU-based algorithms run faster, but as k grows, AP-based solutions become more attractive. This is not the case with all the problems.

Instead, we developed a simple metric similar to the widely adopted FLoating-point Operations Per Second (FLOPS) metric. We simply calculate the number of FLOating-point Comparisons per Second (FLOCS). This can be calculated using the following formula:

$$\texttt{FLOCS} = \texttt{c} \times \frac{\#\texttt{automata per board} \times \texttt{f}}{\#\texttt{cycles per query}}$$

Here, `c` is the number of comparisons per automaton and `f` is the operating frequency. In our case, `c` equals 2, because each automaton compares the query point against the limits of the interval, and `f` is 128 MHz. Using this formula, Table I describes the FLOCS for the 4-byte and the 8-byte comparator macros. In the best case, where output handling does not become a bottleneck, the number of cycles per query are 4 and 8, respectively. However, this may degrade to 494 cycles for both the automata due to the output handling bottleneck in the worst case. The observed FLOCS on the prototype hardware has also been reported. This is in line with the expected behavior as the software and hardware are currently under testing and validation phase. Multiple stages are yet to be pipelined, and some are supposed to be parallelized. Some of the resources are yet to be programmable, and the system software is to be improved with respect to output handling.

	#Automata per board	#Cycles per query	giga-FLOCS
Maximum theoretical throughput			
4-byte macro	43008	4	2752.5
8-byte macro	24576	8	786.4
Minimum theoretical throughput (output regulated)			
4-byte macro	43008	494	22.3
8-byte macro	24576	494	12.7
Observed throughput: on prototype hardware			
4-byte macro	43008	-N/A-	0.95
8-byte macro	24576	-N/A-	0.55

TABLE I: Theoretical and observed FLOCS for the 4-byte and 8-byte macros

V. Future Research

Handling Fixed-Length Strings: The automata designs described in Section III-A compare points denoted by numbers represented as four or eight byte strings. The alphabet of these strings contains all the symbols in the 1-byte symbol set. However, the same designs can be used for comparing strings of any fixed-length defined over any alphabet, as long as the symbols in the alphabet can be lexicographically ordered. The labeling algorithm can be modified accordingly for this purpose. Such string based intervals are used to index into entries of large databases such as those used in the fields of bioinformatics and biometrics.

Handling Variable-Length Formats: Until now, we have only discussed fixed-width formats, where the number of bytes used to represent x, y, and z are identical. This enables the design of pre-compilable automata structures which have a fixed shape for any choice of x and y. However, many applications need the support for variable-width formats, e.g. numbers represented as ASCII strings. A common automaton structure for all values of x and y is not feasible for such

representations. As a continuation of our current work, we will explore the design of automata for such representations using a few constituent sub-automata structures of fixed shapes. For example, we have identified that using only four such sub-automata structures, the required automaton for any interval $[x, y]$ can be generated when dealing with decimal numbers in the ASCII string representation. These numbers may be signed or unsigned, whole or fractional. The compilation of such automata is not significantly slower than the automata described in this paper. However, the performance evaluation of such automata is currently under research.

Handling Points in Higher Dimensions: In this paper, we have demonstrated handling stabbing queries in one-dimensional space. However this methodology can be extended to handling higher dimensions points. For example, in the case of two-dimensional space, the region covered by an axis-parallel rectangle can be represented by two endpoints x_1, x_2 and y_1, y_2, where $x_1 \leq y_1$ and $x_2 \leq y_2$. A query point z_1, z_2 lies within this rectangle if and only if $x_1 \leq z_1 \leq y_1$ and $x_2 \leq z_2 \leq y_2$; i.e. by executing two interval queries in series, and declaring a positive result only if both the queries are satisfied. In fact, generating a unified automaton to do so by concatenating two instances of the automaton described above, one each for each dimension, is trivial. However, handling more generalized cases, e.g. rectangles without axis-parallel edges, any 2-dimensional polygon, any n-dimensional hyper-rectangles, and finally any n-dimensional polygon is a subject of ongoing research.

Handling Multiple Formats: In some applications, the data may be represented using multiple formats. For example, a datestamp could be represented as *mm/dd/yy*, *mm/dd/yyyy*, *mm-dd-yyyy*, etc. We are currently investigating automata structures which allow the representation of points in one of the many permissible formats. In fact, the performance benefits of using this architecture is amplified in such cases as the conversion of the points to a specific format can be avoided.

VI. Conclusion

In this paper, we have provided a streaming solution to the interval stabbing problem using the Automata Processor. To the best of our knowledge, this is the first use of the processor to execute numerical comparisons on multi-byte integers and single and double precision IEEE floating-point numbers. Our implementation is based on defining one automaton for every interval in the list, and executing tens of thousands of such automata in parallel using the resources of a single AP-board. Not only does this provide significant performance benefits in answering the stabbing queries, but also leaves the interval list unordered. This allows expeditious modifications to the list, which is required by many applications. Additionally, the principles of these solutions can be extended to other datatypes like fixed and variable length strings, multidimensional coordinates, datestamps, timestamps, etc. The automata designs presented in this paper are modular and extendable to larger composite automata. These designs exemplify techniques to overcome significant resource and performance bottlenecks which may be useful to prospective application designers working on this processor.

Acknowledgments

We acknowledge funding from NSF Exploratory Grant CCF-1448333 and Micron Technology, Inc.

References

[1] The Micron Automata Processor Developer Portal. http://www.micronautomata.com/. Accessed: Sept, 2016.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 198–207. IEEE, 2000.

[3] J. L. Bentley. Solutions to Klee's rectangle problems. *Unpublished manuscript*, pages 282–300, 1977.

[4] I. S. Committee et al. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008, 2008.

[5] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.

[6] B. Dawson. Comparing floating point numbers, 2008.

[7] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems*, 99:1, 2014.

[8] H. Edelsbrunner. *Dynamic data structures for orthogonal intersection queries*. Technische Universität Graz/Forschungszentrum Graz. Institut für Informationsverarbeitung, 1980.

[9] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *International Journal of Computer Mathematics*, 13(3-4):209–219, 1983.

[10] H. Edelsbrunner. A new approach to rectangle intersections, Part II. *International Journal of Computer Mathematics*, 13(3-4):221–229, 1983.

[11] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143. ACM, 1984.

[12] E. N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Algorithms and Data Structures*, pages 153–164. Springer, 1991.

[13] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical report, 1980.

[14] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[15] I. Roy and S. Aluru. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *IEEE 28th International on Parallel and Distributed Processing Symposium*, pages 415–424, May 2014.

[16] I. Roy, N. Jammula, and S. Aluru. Algorithmic techniques for solving graph problems on the automata processor. In *IEEE 30th International Parallel and Distributed Processing Symposium*, pages 283–292, May 2016.

[17] I. Roy, A. Srivastava, and S. Aluru. Programming techniques for the Automata Processor. In *45th International Conference on Parallel Processing*, pages 205–210, Aug 2016.

[18] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and A. Srinivas. High Performance Pattern Matching Using the Automata Processor. In *IEEE 30th International Parallel and Distributed Processing Symposium*, pages 1123–1132, May 2016.

[19] J. M. Schmidt. Interval stabbing problems in small integer ranges. In *Algorithms and Computation*, pages 163–172. Springer, 2009.

[20] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of the 31st International Conference on High Performance Computing (ISC)*.

[21] K. Wang, E. Sadredini, and K. Skadron. Sequential pattern mining with the micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 135–144, New York, NY, USA, 2016. ACM.

[22] K. Wang, M. Stan, and K. Skadron. Association Rule Mining with the Micron Automata Processor. In *IEEE 29th International Parallel and Distributed Processing Symposium*, May 2015. To appear.

[23] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron. Brill tagging on the micron automata processor. In *IEEE 9th International Conference on Semantic Computing (ICSC)*, pages 236–239, 2015.