Nondeterministic Finite Automata in Hardware - the Case of the Levenshtein Automaton

Tommy Tracy II, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, Gabriel Robins
University of Virginia
Charlottesville, VA
[tjt7a, mrs8n, njb2b, jpw8bd, kw5na, ks7h, gr3e]@virginia.edu

ABSTRACT

The Levenshtein Nondeterministic Finite state Automaton (NFA) recognizes input strings within a set edit distance of a configured pattern in linear time. This automaton can be pipelined to recognize all substrings of an input text in linear time with additional use of nondeterminism. In general, von Neumann hardware cannot directly execute NFAs without significant time or space overhead. A von Neumann simulation of the Levenshtein automaton incurs exponential run time overhead in the general case. A common technique to avoid the simulation overhead is to convert the pipelined NFA to a DFA, but at the expense of heavy pre-computation and high space overhead.

In this paper, we introduce a novel technique for executing a pipelined Levenshtein NFA using Micron's Automata Processor (AP), avoiding the run time and space overheads associated with CPU and GPU implementations. We show that run time remains linear with the input while the space requirement of the automaton becomes linear in the product of the configured pattern length and edit distance. These properties allow the AP to execute large instances of the Levenshtein NFA or many small instances in parallel thus making the automaton a viable building block for future approximate string applications on the AP.

Keywords

Automata Processor, Levenshtein Automaton, Nondeterministic Finite Automata, Approximate String Matching

1. INTRODUCTION

The pipelined Levenshtein NFA $A_L(P, d)$ recognizes strings that approximately match a search string pattern P. The closeness of the match is measured by the edit distance d between the input string and P. This edit distance, also known as the Levenshtein distance[2], is determined by the minimum number of primitive character operations required to convert the input string to P. The *primitive character operations* considered are:

$$\begin{aligned} Insertions &: waoo \rightarrow wa\mathbf{h}oo \\ Deletions &: w\mathbf{a}hoo \rightarrow whoo \\ Substitutions &: \mathbf{w}ahoo \rightarrow \mathbf{y}ahoo \end{aligned} \quad (1)$$

Approximate string matching is used in an array of application domains including bioinformatics, e.g. motif search and alignment, and text retrieval, e.g. spell-checking and search engine applications. Utilizing a Levenshtein automaton is a method of determining approximate string matches without explicitly calculating the edit distance between the input string and the search string pattern.

$A_L(P, d)$ accepts inputs within edit distance d of P in linear time with the input. $A_L(P, d)$ represents incurred errors and matches with state transitions. When there is an error, the type of error is ambiguous; choosing which error results in the minimum edit distance is made easier with nondeterminism. Each accept state represents a different edit distance between the input and P. The state representing the minimal edit distance nondeterministicaly reachable on the input gives its edit distance from the pattern.

$A_L(P, d)$ as an NFA is fairly straightforward to pipeline. Other approximate string matching algorithms have to first split the input into k-mers before calculating the edit distance between the k-mers and the search pattern. $A_L(P, d)$ can check all substrings of the input for a near-match in linear time without the need for hashing or indexing!

Micron's Automata Processor (AP) is an NFA engine, which allows software developers to execute NFAs in hardware. This hardware can simulate NFAs as wide as can fit on the hardware, and doesn't have the same scalability limitations as von Neumann simulation techniques like bit wise parallelism or the associated space explosion incurred by a powerset construction [5]. The ability to execute wide nondeterminsm without the associated computation and space overhead makes the AP a promising platform for the pipelined Levenshtein NFA.

In this paper we present our novel technique for executing the pipelined Levenshtein NFA on Micron's Automata Processor (AP). The AP can execute many Levenshtein NFAs with differing search string patterns concurrently processing the same input. The rest of the paper will introduce the Levenshtein NFA, the Automata Processor, the technique

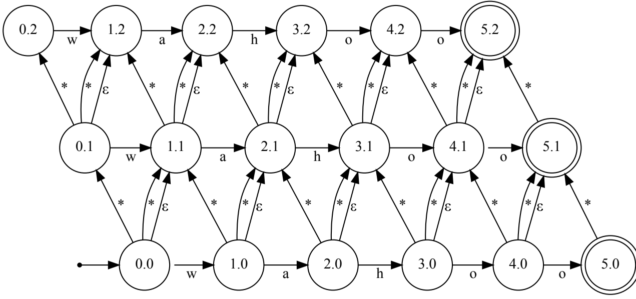

Figure 1: $A_L(\mathbf{wahoo}, 2)$

we developed for executing the Levenshtein automaton on the AP efficiently, results, and finally application-dependent modifications that can be made to the design.

2. THE LEVENSHTEIN AUTOMATON

The Levenshtein automaton [7] is a NFA configured for a search string pattern $\boldsymbol{P}$ and max edit distance $\boldsymbol{d}$, that recognizes the set of strings that are within edit distance $\boldsymbol{d}$ of $\boldsymbol{P}$, meaning the input string can be transformed into $\boldsymbol{P}$ by at most $\boldsymbol{d}$ single-character operations: insertions, deletions, and substitutions.

Figure 1 is of a Levenshtein NFA for the search string pattern $\boldsymbol{P}$="wahoo" and edit distance $\boldsymbol{d}$=2. The automaton encodes the running total of single-character errors in the row index. If the current input symbol fails to match on the current search string pattern symbol, the automaton shifts to a state that is one row upward, indicating a single symbol error. As more errors occur, the active automaton states will shift further upward until there are no more rows. The Automaton encodes the current match index of the search string pattern with the column index. For example, the first match transitions from the 1st to the 2nd column of Figure 1 represent a match with the first character of the match string, $'w'$, the second set of transitions for the second character $'a'$.

Each of the states in Figure 1 has a name that corresponds to the state's match index value and the number of errors incurred so far in the automaton$'$s reading of the input. At start state 0.0, the automaton has made no progress towards the acceptance states. By matching on the first character $'w'$, the state 1.0 is activated. If the input string starts with a non-matching character, it will transition up one row, indicating an error on the first character. Finally, once all symbols of the the search string pattern have been accounted for with matches or edits, the last state of each row is an acceptance state. Activating an acceptance state indicates that the input is within the maximum allowed d of P and also provides the edit distance between input and P based on the row the acceptance state resides. The Automaton handles all state transitions with the following rules:

1. All match transitions, are represented as a transition in the rightward direction. Given the example automaton, the input string 'wahoo' would take the following path to an accept state by only traversing matching transitions:

0.0(start) → 1.0 → 2.0 → 3.0 → 4.0 → 5.0(accept)

2. Any character insertions, are represented as automaton transitions in the upward direction. As an example, the following path represents the acceptance of the string 'wahoeo', where there is a single insertion after the first 'o' in 'wahoo':

0.0(start) → 1.0 → 2.0 → 3.0 → 4.0 → 4.1(insert) → 5.1(accept)

3. Any substitutions, where a character in P is replaced by another symbol in the input, are represented as a diagonal '*' transition. Our example Levenshtein automaton would accept the string 'waeoo', where 'e' replaces 'h' using the following path:

0.0(start) → 1.0 → 2.0 → 3.1(substitution) → 4.1 → 5.1(accept)

4. Finally, any deletions are represented in the Levenshtein automaton with ε-transitions. These ε-transitions are used in NFAs to represent 'free' transitions, where a transition is made without the need to consume an input symbol. As an example, the Levenshtein automaton accepts the string 'wah', because it is 2 deletions from 'wahoo'. The automaton would use the following path to accept this string:

0.0(start) → 1.0 → 2.0 → 3.0 → 4.1(delete) → 5.2(delete)(accept)

3. THE AUTOMATA PROCESSOR

Micron's Automata Processor (AP) is a re-configurable non-von Neumann MISD processor that can run multiple Nondeterministic Finite Automata (NFA) concurrently on the same input stream. The automata are designed in a graphical environment called the AP Workbench that generates an XML design file in the Automata Network Markup Language (ANML). This file is then compiled into a bitstream that can be processed by the AP hardware[1]. Alternatively, an AP design can be generated in ANML by a script; we chose this approach with our solution.

Automata on the AP are composed of a connected, directed network of State Transition Elements (STEs). These STEs are activated when the STEs are enabled by a neighboring STE and the input matches the STE's assigned 8-bit symbol class. STEs that represent a starting state are called Start STEs; those that represent accept states are called Reporting STEs. In order to simplify automata designs, the APs design tools include the macro construct. Developers can design small, parameterizable automata as macros [8]. We used this construct to create a set of macros and simplify the design.

Micron's current generation AP, called the D480, can runs at an input symbol rate of 133 MHz, with each chip supporting two half-cores. Each half-core contains 96 blocks of 256 STEs each. In total, each D480 chip contains 49,152 STEs, and each AP PCIe board can contain up to 48 AP chips. [8]

4. LEVENSHTEIN ON THE AP

The Mealy-type NFA representation of the Levenshtein automaton cannot be executed on the AP in its current form. The architecture requires that the automaton be a Moore-

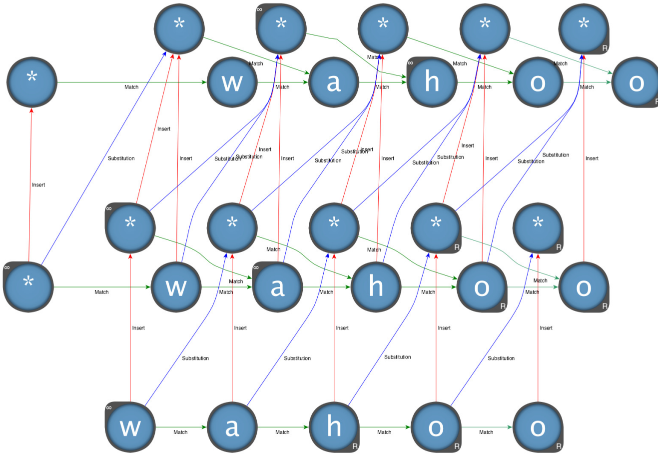

Figure 2: Moore-type Levenshtein Implimentation without ε-transitions

type machine, where transitions are triggered on state values. In addition, the AP cannot recognize ε-transitions (those which consume no input). These limitations required us to develop a mapping technique for converting the Mealy-type Levenshtein automaton with ε-transitions to an NFA that the AP can execute. In this section we present our technique that we have generated a script for and the overhead that we incur to fulfill it. We also introduce a set of macros that our script uses to simplify the design of a Levenshtein automaton.

4.1 Mealy vs. Moore

A Moore-type state machine [4] is a finite-state machine with triggers on the machines's states. A transition is taken from state A to state B, if state A is active and if state B is designated to trigger on the input symbol. A Mealy-type state machine [3] is a finite-state machine with triggers on the transitions. Instead of having a separate state for each possible input symbol, a single Mealy-type state could have multiple transitions into it with differing input triggers. This allows a Mealy-type Machine to be more compact than a Moore-type Machine. Figure 1 is a Mealy-type machine diagram.

The Automata Processor′s architecture can only realize Moore-type Machine designs. We converted the Levenshtein automaton to a Moore-type Machine by creating *super states* which we realize as macros as shown in Table 1. Each STE in a *super state* macro represents both possible input symbols to that state: a search string symbol match, or an error, represented with '*'. The '*' character represents all symbols in the APs alphabet, so these transitions are also taken in the case of a match. This is not a problem because the resulting active set explosion isn't a complication for the AP's architecture. Each *super state* macro has two input and two output ports connected to their respective STEs. In addition, all outgoing transitions from the original Mealy-type state must be duplicated for each STE in the *super state*; therefore each output port will have identical transitions.

Figure 2 shows the resulting *super states* after converting the Mealy-type machine to a Moore-type machine. The bottom-most row and left-most column do not contain super states, because they only have one transition into each of these states. All other states are represented with two STEs with identical outgoing transitions. This figure only contains match, insertion, and substitution transitions to simplify the diagram. ε-transitions cannot be represented as STEs with ε-match symbols; we will address ε-transitions next.

4.2 Epsilon-Transitions

Another limitation of the AP is the lack of support for ε-transitions. Many Mealy-type NFAs use the ε-transition to reduce the complexity of a design. We will present a series of macros to solve this problem.

For internal transitions, we can handle ε-transitions by... Roy and Aluru[6] handle ε-transitions by connecting all states that have outgoing ε-transitions to the states that have the associated incoming ε-transition. This method works for internal states, but does not account for ε-transitions from starting states or ε-transitions to accept states. We broke the Leavenshtein automaton's ε-transitions into three categories: starting state ε-transitions, accept state ε-transitions, and internal ε-transitions.

We classified all states that could be reached with ε-transitions from the starting state 0.0 as starting *super states*. Those states that were reachable and had a match transition were called late-start match *super states* to indicate that they serve as starting states that match on P, but after one or more deletions. We created a macro called the Late Start Match Block that represents both states in the *super state*, where the match STE is a starting STE. In the case of starting with an error other than a deletion, we introduced a Starting Error Block macro. In this macro we have the error state serve as the starting STE. Because this error incurs a row penalty, the first Starting Error Block macro is on the second row.

To account for deletions from the end of the search string pattern, we created Simple Error Report Blocks. These blocks report on a symbol or error match, and represent the acceptance of the input string. In the case of Figure 1, only the last states in each row served as accept states. This was fine for a machine with ε-transitions, because deletions were accounted for by these transitions from lower rows. To have the same functionality without ε-transitions we had to collapse the ε-transitions to form report diagonals. Report diagonals represent all possible deletions from the end of the Search String Pattern. For this reason, our resulting AP executable Levenshtein automaton had several more report elements to account for those diagonals; one for each state that was within ε-transitions from an accept state in the original design.

Finally, the last case of ε-transitions that we needed to handle was internal ε-transitions that originated from states that were neither starting states, nor accept states. To do this, we devised an iterative algorithm to account for all deletions possible from each given state. For example, to account for all deletions from the bottom-left most state in Figure 2 we used transitions that skipped one column and matched on the second row to represent a deletion and a match. We then created a transition from that same state

to one column over and an error match on the 3rd row to account for a deletion and a mismatch. The algorithm then continues for larger automata to include multiple deletions followed by a match or error. It should be noted that we did not account for deletions followed by insertions, because that is equivalent to a single substitution.

4.3 Construction Algorithm

In this section we present the technique we used to construct Figure 3, a Moore-type Levenshtein NFA without ε-transitions to execute on the AP given an arbitrary search string pattern P and an edit distance d . The one limitation that our algorithm imposes is that the d be less than the length of P. Intuitively, this seams a reasonable stipulation because if d were the same as the length of P, the empty string would be an acceptable input to the Automaton, effectively producing a useless machine. We break this section into steps to clarify each component of the algorithm.

1. In this step, we construct the bottom row of the Levenshtein automaton. The first block in the bottom row is the Starting Match Block, a starting macro that matches on the first character of P. This is then connected to a chain of Simple Match Blocks until d blocks from the end of the row. All blocks after this index serve as Reporting Match Blocks because they represent early Reports due to potential deletions at the end of an otherwise perfect matching input pattern. As an example, the string "wah" would be accepted by the Automaton because it is two deletions from "wahoo". "waho" and "wahoo" would also be accepted on the bottom-most row of the automaton.

2. These Reporting Match Blocks represent the first block on the deletion diagonals (reachable by ε-transitions). As the row index increases, representing more cumulative errors, fewer deletions can be accounted for at the end of the search string pattern to be within d, therefore reducing the number of Reporting Match Blocks until the top-most row only has a single Reporting Match Block. The right-most Reporting Match Block on the bottom row represents an edit distance of 0, the diagonal originating at the 2nd to last STE from the end represents an edit distance of 1, and so on. We can use this information to assign edit distance values to each of the Reporting Match Blocks.

3. The Late Start Match Blocks are placed in a diagonal from the Starting Match Block. These blocks represent early deletions, where the first 1 to d characters of the Match String are deleted. As expected, as the number of early deletions increases, so does the row index, for errors, and the column count, for the matching P index.

4. The Simple Starting Error Block is placed one column before and one row up from the Starting Match Block. If there are errant insertions before P, it is necessary to account for these with this block. This block is then connected to a vertical chain of Error Match STEs that go up to the last row. This column represents all possible insertions allowed before the Match String.

5. Next, the Starting Error Block is placed on the next row above the Start Match Block. This block represents the first mismatching character (replacement of 'W') in the input string. To account for the possibility of deletions and then mismatches, this block also has a diagonal of Starting Error Blocks to account for these deletions in the upper-right direction.

6. Finally, the rest of the blocks in the design are Simple Error Blocks. These blocks neither serve as starting nodes nor reporting nodes.

The blocks are connected in the same way as the Mealy-type design for all but ε-transitions; we used the previously discussed start STEs, early reporting STEs, and iterative deletion transitions to account for these transitions. To automate this algorithm, we developed a script for generating an AP-compatible Levenshtein NFA given any input string match pattern and max edit distance. We made all starting STEs in the design 'start on all' STEs. This meant that every starting STE would be active for all symbols in this input; this resulted in a pipelined Levenshtein NFA that determines approximate string matches with all substrings of the input, but still in linear time!

5. SCALABILITY

Because the Automata processor can run multiple NFAs concurrently on the same input, scalability is important when considering the performance of the AP. It is important that the number of STEs required is minimized so that the AP hardware can execute as many automata concurrently as possible. In this section we will determine the number of STEs required to realize a Levenshtein automaton, and the number of Levenshtein automata we can execute at one time on the AP.

d represents the maximum edit distance.
L represents the length of the match string pattern P.

$$STE_{total} = d + L + (2 * L * d)$$
$$STE_{reporting} = (d + 1)^2$$

These equations indicate that the number of STEs required to construct a Levenshtein NFA scales with the product of the configured pattern length L and the maximum edit distance d, and that the number of reporting STEs is $\mathcal{O}(d^2)$. Considering that for most bioinformatics applications the edit distance is kept relatively low at 5 or less, these results indicate a scalable design. We determined that a single AP core can hold an Automaton configured with a pattern length of size 2730 and edit distance of 4. When considering a relatively modest pattern size of 100 with an edit distance of 4, a single AP core has the capacity for 27 parallel automata.

In the case of an application that requires executing the automaton on many different search string patterns, more than the capacity of the available AP hardware, it is possible to use soft re-configurations (reload match symbols) to swap on new search string patterns. As long as d and the length of P do not change, the same AP design can be used.

Table 1: Levenshtein Macros

Macro Name	Levenshtein Macros	Description
Starting Match Block		- Starting STE - Matches on first character of search string pattern.
Simple Starting Error Block		- Starting STE - Matches on error values. - Accounts for errors in the beginning of an input string.
Simple Match Block		- A single STE matches on a character in the matching string. - This macro is used on the bottom row, where no errors occur.
Reporting Match Block		- This block is the same as the simple match block, except it reports on a match.
Late Start Match Block		- Starting STE - This macro accounts for inputs with beginning deletions, where the input string starts late. - This macro has two states: one that triggers on a matching character, and another on an error.
Starting Error Block		- Starting STE - Matches on all characters. - Used to accept input strings with starting errors.
Simple Error Block		- This block has two STEs: one to match on the match string character, the other to match on an error. - This block neither starts no accepts.
Simple Error Report Block		- This block has two accepting STEs: one for a match, the other for an error.

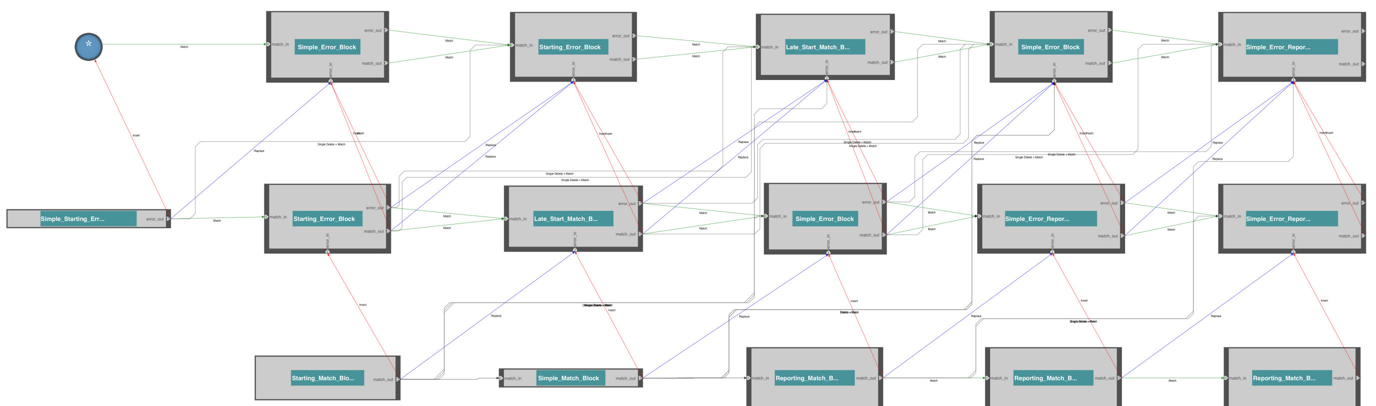

Figure 3: AP-Executable A_L(”wahoo”, 2)

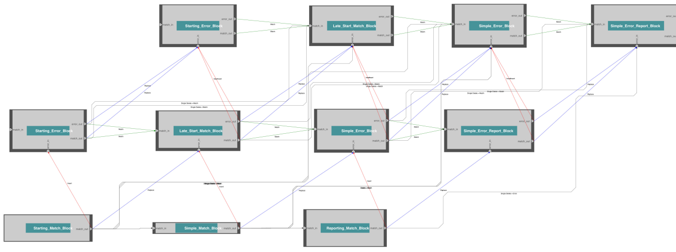

Figure 4: A reduced size Levenshtein Automaton

6. FUNCTIONAL MODIFICATIONS

Our illustrated design determines if any subset of the input string is within edit distance d of P, and reports if so, as well as reporting the indicated edit distance of the matching input string. This design can be extended to include additional functionality like variable scoring, where different primitive operations have differing costs associated with them. This is particularly useful for bioinformatics applications where edit operations occur with differing probabilities. Another enhancement we can make is reducing the size of the Levenshtein NFA by removing early insertion and late deletion states. This allows us to reduce the number of STEs required to instantiate a single Levenshtein automaton, but at the expense of determining the input's Levenshtein score.

6.1 Scoring

Variable edit scoring has potential applications in bioinformatics. It allows a programmer to assign different costs to the edit operations accounted for by the NFA. To explain this enhancement, we will consider insertions. Our design assigns a cost of one error to insertions, deletions, and substitutions. If we wanted to double the cost of an insertion, we could adjust our upward direction transitions from instead of connecting to the next immediate row, connect to the row two rows up. This would account for an insertion cost of 2. This same strategy could be used for any of the other edit operators. The limitation here is that the cost would need to be an integer value.

6.2 Levenshtein Reduction

The Levenshtein NFA recognizes an input string by triggering a reporting STE. This STE has an edit distance associated with it, depending on which diagonal it is on. The bottom-right-most row has an accept state with edit distance 0, because the entire pattern has been matched with no errors; the second row has an edit distance 1 assigned to it, because one error has been discovered in the traversal of the input to the accept state. If the edit distance metric is not of importance it is possible to reduce the size of the Levenshtein NFA to accept on only max edit distance accept states. This works because if a string X is a perfect match of P, X with one deletion is within edit distance 1 of P. In this way, it is possible to account for all possible strings by only accepting the worst-case edit distance and ignoring prefixes and suffixes.

Figure 4 shows the reduced version of $A_L(''wahoo'', 2)$. It is the same automaton but with removed early insertion states and late insertion states. What is clear is that only $d+1$ reporting states are required, because only worst-case reports are necessary. For this reason, the last reporting element diagonal serves as the last state in each of the rows. This modification to the Levenshtein automaton has a small impact on the over-all size of the automaton, but if many are used in parallel, this may have a non-negligible impact on the capacity of the hardware.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a technique for executing a pipelined Levenshtein NFA on Micron's Automata Processor. We found that preserving the automata's nondeterminism resulted in a scalable design on the AP. We also presented several modifications that could be made to the design to account for variable edit costs as well as a modification for reducing the size of the automaton at the expense of determining edit distance.

The Levenshtein automaton has potential in the field of bioinformatics, and with the introduction of the AP, that potential can finally be unlocked. Future work includes comparing the execution of $A_L(P, d)$ on the AP versus the DFA and simulated versions on a CPU and GPU. Finally, we will implement a short read aligner from $A_L(P, d)$ and compare our results to the state of the art CPU and GPU aligners.

8. ACKNOWLEDGMENTS

We would like to thank Micron for their cooperation with this and many other AP projects. We would also like to acknowledge the help that we got from the Center for Automata Processing (CAP) at the University of Virginia.

9. REFERENCES

[1] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(12):3088–3098, Dec 2014.

[2] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR*, 10(8):845âĂŞ848, Feb 1966.

[3] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal, The*, 34(5):1045–1079, Sept 1955.

[4] E. F. Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[5] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[6] I. Roy and S. Aluru. Finding motifs in biological sequences using the micron automata processor. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 415–424, May 2014.

[7] K. U. Schulz and S. Mihov. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85, 2002.

[8] K. Wang, M. Stan, and K. Skadron. Association rule mining with the micron automata processor. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.