Subset Encoding: Increasing Pattern Density for Finite Automata

Deyuan Guo, Nathan Brunelle, Chunkun Bo, Ke Wang, Kevin Skadron
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
Email: {dg7vp, njb2b, cb2yy, kw5na, skadron}@virginia.edu

***Abstract*—Micron's Automata Processor is an innovative reconfigurable hardware accelerator for parallel finite-automata-based regular-expression matching. While the Automata Processor has demonstrated potential for many pattern matching applications, other applications receive reduced benefit from the architecture due to capacity limitations or routing limitations. In this paper, we present an efficient input encoding method that often results in simplified automata designs and simplified routing by better exploiting the powerful character matching abilities of the Automata Processor. This enables the Automata Processor to more efficiently solve problems for a broad range of new applications. We present Hamming distance, edit distance, and Damerau-Levenshtein distance automata as motivating examples, observing space efficiency improvements up to 192x.**

***Keywords*-Automata Processor; subset encoding; finite automata; Hamming distance; Levenshtein automata**

I. Introduction

Micron's Automata Processor (AP) [1] is an innovative hardware accelerator for parallel finite automata-based regular expression matching. It is a non-von Neumann processor which simulates nondeterministic finite automata (NFAs) mixed with Boolean logic gates and counters. The AP achieves this end by utilizing a bit-parallel technique [2] within DRAM, allowing it to run many NFAs in parallel.

The memory-derived AP can match an input byte stream with a large number of homogeneous-NFAs [1] on the AP in parallel. Homogeneous-NFAs are computationally equivalent to traditional NFAs, the only difference being that homogeneous-NFAs match characters in states instead of on transitions, as shown in Figure 1. Each NFA state (called a state transition element, or STE) in the current design of the AP is realized as a 256-bit memory column to represent the subset of 8-bit characters which that state matches. An STE matches on an input character when the input belongs to that state's character subset. This action intuitively operates as a character-OR in regular expressions. More details about the AP architecture are provided in Section III-A.

Many automata designs on the AP have been developed for accelerating real-world applications, including: network intrusion prevention [1], DNA motif searching [3], Brill tagging in natural language processing [4], association rule mining [5], entity resolution [6], and DNA alignment [7]. These prior works observe the AP achieve more than 100x speedup over CPU implementations.

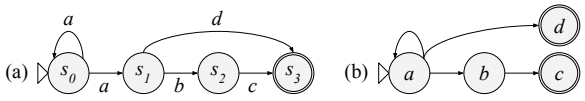

Figure 1. Two equivalent NFA representations for pattern *a+(bc|d)*: (a) Traditional NFA; (b) Homogeneous NFA (the model of the AP).

While working toward these exciting results and other failed ones, we have found that many automata designs on the AP face a few common problems. One problem is the number of patterns that can be represented on the AP at one time and thus checked concurrently. This pattern density is constrained by hardware resources, such as limited STE capacity or inefficient routing. Hardware limitations on the number of states an automaton may have can either limit the amount of parallelism that machine has available (making the application less efficient), or prevent an automaton from fitting on the AP altogether (making the application impossible to accelerate with the AP). Additionally, even some small automata may have a complicated transition topology, making them difficult to route on the architecture. This can cause a poor STE utilization rate. As a consequence, for large problems that exceed the capacity of the AP, one may have to either use more AP hardware, or perform the computation using many passes through the input stream.

Another problem is that it is difficult to fully utilize the powerful character-OR ability of STEs on the AP. One reason is that if we combine many regular expression patterns together using character-ORs, we cannot later distinguish which character was matched. The other reason is that there is a mismatch between common regular expression structures and the matching procedure of the AP, as real world regular expressions typically operate over unions of larger subexpressions rather than using single character-ORs.

We observe that it is possible to overcome these limitations. Since each STE is a 256-bit memory column, it can represent any one of 2^{256} possible subsets of 8-bit characters. This gives plenty of entropy to represent more complex matching behavior than the simple character-OR.

To address these problems, we propose the subset encoding method, which can help the AP to achieve better utilization of its hardware resources for various applications. The subset encoding method encodes both application data and patterns into subsets of characters, illustrated in Figure 2. For example, it can encode a 64-character DNA sequence into a subset of 8-bit characters, so that it only uses one

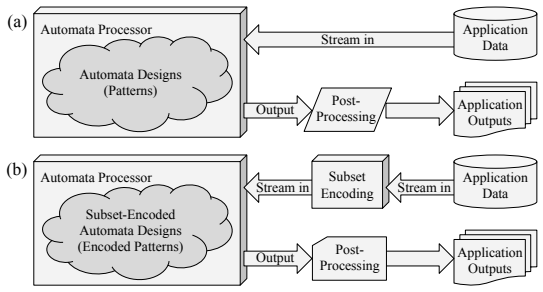

Figure 2. Overview of the subset encoding method: (a) The typical AP execution flow; (b) The execution flow after the subset encoding method

STE to represent this DNA sequence. By matching encoded data with encoded automata we are able to access a richer design space, addressing those problems mentioned above.

The subset encoding method can fully exploit the character-OR ability of STEs by encoding a *sequence* of data into a subset of characters which are then put in a single self-loop STE. Effectively, the subset encoding method encapsulates a short transition history using only one STE, allowing that STE to match on a sequence of characters rather than being limited to matching on just one character. From here, we can adapt the method to find solutions which best fit the characteristics of individual applications by analyzing tradeoffs among input stride, input rate, and alphabet size.

Furthermore, the subset encoding method improves the space efficiency and the degree of parallelism. This results in smaller automata structures with fewer states and connections, enabling more automata to be placed on the AP by reducing routing complexity. For example, we apply the subset encoding to traditional Hamming distance automata and Levenshtein automata. The large structures required for these automata make it very difficult or sometimes impossible to place and route on the AP. We show that after applying the subset encoding method, the automata structures become highly compressed and thus require significantly fewer routing resources on the AP.

Experimental results show that by applying the subset encoding method, the space efficiency of these automata can be improved from 3x to 192x for different application scenarios. We can also use multiple-stride in the subset encoding to increase the input rate. As a result, overall performance on the AP can be significantly improved. This encoding technique will impact future decisions in the design of the AP or other automata-based co-processors.

II. Related Work

The input stride technique was discussed in Becchi's Ph.D. thesis [8]. The input stride technique can compress multiple input characters into a single byte, e.g. compressing four DNA characters to one byte. This technique can increase the input rate, but it may make automata more complicated, thus difficult to route. In this paper, we consider the input stride as one of the design parameters of the subset encoding method, and we combine the stride technique with the subset encoding for more efficient hardware utilization.

A bounded Hamming distance automaton on the AP was described by Roy and Aluru [3] for solving the DNA motif searching problem. For hamming distance (l, d), that is to match a pattern of length l with at most d substitutions, this traditional design needs $(2d + 1)l - 2d^2$ STEs. Without the subset encoding, the capacity of each STE is not efficiently utilized. In addition, they estimated the capacity assuming the STE utilization efficiency is 80%, but the actual routing results for large l and d might be much worse. In this paper, we introduce new subset-encoded Hamming distance automata which only use $2d + 2$ STEs when the pattern is within the STE capacity, and we show that the new design can achieve higher pattern density.

The Levenshtein Automaton is an elegant solution for computing edit distance. It is based on dynamic programming and uses ε-transitions. However, when implementing the Levenshtein Automaton on the AP, many additional STEs and connections are required for processing *-transitions and ε-transitions, which makes the routing very inefficient. For example, Tracy [7] showed a straightforward implementation of the Levenshtein automata on the AP, but the degree of parallelism was quite limited due to the low routing efficiency. In this paper, we introduce the subset-encoded Levenshtein automata, which allow us to separate the large automata into pieces so as to improve the routing.

III. Subset Encoding Method

In this section, we introduce details of the subset encoding method based on the AP. We discuss the AP architecture, tradeoffs between encoding and matching, the problem reduction, and a model for analyzing tradeoffs among design parameters such as stride, alphabet size, and the way of encoding. The overhead of encoding is also discussed in this section.

A. The Matching Mechanism of the AP

Figure 3 shows the memory-based architecture of the AP. Each 8-bit input character is decoded as a memory row address. Each 256-bit column in memory represents the matching characteristics of an STE. If a memory cell is set to 1, then the input character for that cell's row matches with the STE for its column. STEs are connected by a routing matrix, and some Boolean logic gates and counters are added for extending matching efficiency. Since the matching behavior and routing are represented separately, the user can update the character matching table without reconfiguring the connections. We utilize this feature to gain a performance benefit when the AP has insufficient hardware capacity for all patterns we wish to match against. In this case, we perform our computation by making multiple passes on the input data, each time reusing the same automata structures but with different matching characters.

The current AP generation is defined by the following hardware hierarchy:

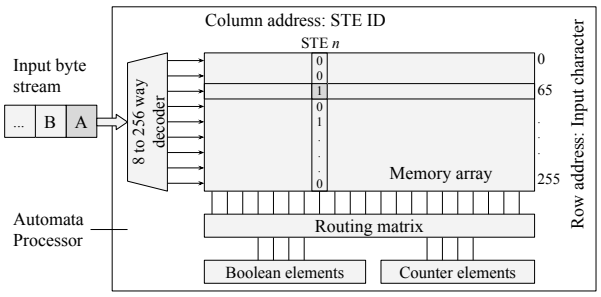

Figure 3. The architecture of one AP block. Each row corresponds to one 8-bit input character, each column corresponds to one STE. If a memory cell is set to 1 then the input character for its row matches with the STE for its column.

- Each board has 32 chips assembled into 4 ranks with 8 chips each.
- Each chip has 2 cores. Automata transitions are not able to span between cores, making the core the limitation on automaton size.
- Each core has 96 blocks. These blocks are connected by an inter-block routing matrix, supporting automata transitions among different blocks.
- Each block has 256 STEs (32 of which can report off board), 12 reconfigurable Boolean elements, and 4 threshold counters. These are all connected by an intra-block routing matrix, supporting automata transitions between components within that block.

In summary, each AP board can match an input byte stream in parallel over 1.5 million STEs at an input rate of 133MB/s.

Each AP board is also equipped with an on-board FPGA. Micron has not yet indicated support for users to place functions on the FPGA, but ideally, this acts as an accelerator for potential preprocessing/postprocessing of AP input/output and provides a way to dynamically interact with the AP without CPU intervention.

B. Redefining Regular Expression Matching Toward Better AP Design

The regular expression matching problem, the general problem which the AP aims to accelerate, seeks to determine whether or not a given input string belongs to the set of strings as defined by a given regular expression. A proper solution to this problem requires answering correctly for *every* possible input string, in other words the regular expression matching problem is a decision problem.

We observe that the encoding function needs not be *onto*, in other words not every string is necessarily a valid encoding of some possible input strings. For such strings, the acceptance behavior of the automaton is undefined. In order to solve this new "promise" problem (where the input is *promised* to be a valid encoding of some strings), the automaton only needs to behave correctly on valid input strings and may behave arbitrarily on all other inputs [9].

The freedom to behave arbitrarily on some input strings results in greater freedom in automaton design.This is responsible for the improved automaton efficiency provided by the subset encoding. The freedom in matching behavior allows for design flexibility in other parameters, such as automaton size, automaton connectivity, or input rate.

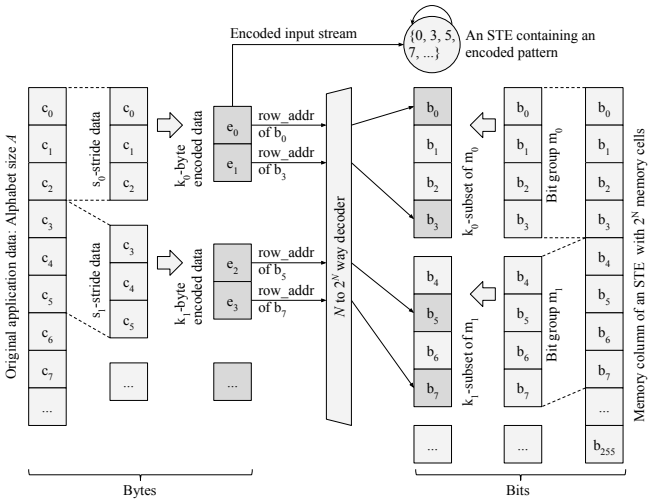

Figure 4. The model of the subset encoding method

C. Modeling Subset Encoding

The core idea of the subset encoding method is to convert subsequences of the input sequence into subsets of characters. This then allows a single STE, whose only operation is to match on a subset of characters, to match subsequences of characters. There are many ways one can do this encoding depending on certain design parameters. We provide a model of our method (Figure 4) which shows how to encode a string into subsets of characters considering tradeoffs among design parameters such as stride, input rate and alphabet size. A taxonomy of design parameters are listed as follows.

1) *Characteristics of the input stream*

- A: The size of the application data's alphabet (Σ)

2) *Characteristics of the AP hardware*

- N: The number of bits in each AP character
- 2^N: The number of memory cells in one STE

3) *Encoding Design choices*

- M, m_i: Partition the 2^N memory cells into M *bit* groups, where each group M_i has m_i bits
- k_i: Choose k_i bits from group M_i, which has $\binom{m_i}{k_i}$ different encoding
- s_i: Map s_i-stride characters (consider s_i consecutive characters as atomic) to group M_i
- F: A mapping from s_i-stride application data to k_i-subset of groups M_i, represented as a positive integer. For a sequence of characters the offsets of related bit groups should be added with the final encoding.

$$F = \left(\Sigma^{s_i} \mapsto \binom{\{0, \ldots, m_i - 1\}}{k_i}\right) + \textit{offset}_i \quad (1)$$

where the offset of the i-th bit group is:

$$\textit{offset}_i = \textstyle\sum_{j=0}^{(i \bmod M)-1} m_j \quad (2)$$

In summary, we encode offset information together with the input data so that a specific character will only match with its corresponding bit group. The offset will roll back to zero when it exceeds the length of the longest pattern.

All of the above encoding design choices may be made independently under the following constraints.

4) *Constraints*

- The total number of bits in all bit groups are bounded by the number of memory cells in each STE:

$$\sum_{i=0}^{M-1} m_i \leq 2^N \tag{3}$$

- When picking k_i bits out of m_i bits, maximum value of $\binom{m_i}{k_i}$ occurs when $k_i = m_i/2$. So it must hold that:

$$1 \leq k_i \leq m_i/2 \tag{4}$$

- When mapping from s_i-stride characters to k_i-subset of m_i bits, we need to ensure that the number of subsets exceeds the number of all possible s_i-stride characters:

$$A^{s_i} \leq \binom{m_i}{k_i} \tag{5}$$

5) *Encoding results*

- The capacity of a single STE, i.e. how many original input characters can be encoded in one STE:

$$\textit{Capacity}_{STE} = \sum_{i=0}^{M-1} s_i \tag{6}$$

- The actual input rate after encoding, i.e. how many original input characters can be processed per second:

$$\textit{InputRate}_{Encoded} = \frac{\sum_{i=0}^{M-1} s_i}{\sum_{i=0}^{M-1} k_i} \times \textit{InputRate}_{AP} \tag{7}$$

In practice, we can pick constant numbers for stride s_i, subset size k_i and bit group size m_i. As a result, we can apply the subset encoding method and analyze the encoding performance with the following steps:

1) Determine the characteristics of the input data and the AP hardware, such as the input alphabet size A and the number of bits 2^N in each STE.
2) Choose design parameters, including stride s, bit group size m, and subset size k, such that $A^s \leq \binom{m}{k}$.
3) Design a constant-time encoding function to map each s-stride input character to a k-subset out of m bits, and add bit group offsets to the final encoding.
4) Calculate capacity and input rate. After encoding, an STE can contain $s \times 2^N/m$ original input characters, and the actual input rate will be s/k times the input rate of the AP.

D. Subset Encoding Examples

In this subsection we show two examples of encoding following the above procedure. One example encodes DNA characters with 4 memory cells each, and the other example encodes English lowercase letters with 8 memory cells each. As a result, an 8-bit STE can contain 64 DNA characters or 32 English letters.

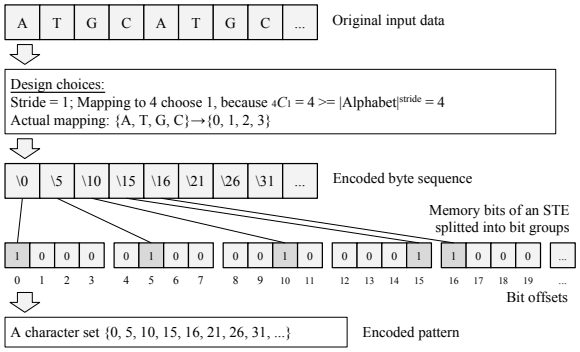

Figure 5. The $\binom{4}{1}$ subset encoding for DNA characters. The DNA sequence "ATGC…" is encoded into a byte sequence "\0\5\10\15…" and a set of characters {0,5,10,15,…}. Matching is done by putting the set of characters in a self-loop STE and streaming the set of characters.

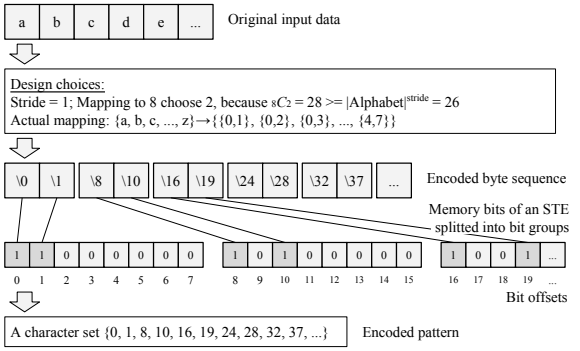

Figure 6. The $\binom{8}{2}$ subset encoding for English lowercase letters. The string "abcde…" is encoded into a byte sequence "\0\1\8\10…" and a set of characters {0,1,8,10,…}.

DNA Characters:

1) The alphabet size of DNA is $A = 4$. Assuming we have a pattern of length 64, we can map a single-stride DNA character to $\binom{4}{1}$ (1 bit is enabled for every group of 4 bits in the STE memory column), as shown in Figure 5.
2) Design choices under the constraint $A^s \leq \binom{m}{k}$: Stride $s = 1$, bit group size $m = 4$, subset size $k = 1$. A single STE can contain $2^N/m = 64$ bit groups, and the pattern has $M = 64$ bit groups.
3) The encoding function F can be a one-on-one mapping: $F = \{\{A, T, G, C\} \mapsto \{0, 1, 2, 3\}\} + \textit{offset}$, where for the n-th DNA character, $\textit{offset} = (n \bmod M) \times 4 = (n \bmod 64) \times 4$.
4) As a result, the capacity of a single STE is $2^8/4$ = 64, and the input rate after encoding is $s/k = 1/1 = 1$x, the same as the input rate of the AP.

English Lowercase Letters:

1) The alphabet size of English lowercase letters is $A = 26$, so assuming we have a pattern of length 32, we can map a single-stride English letter to $\binom{8}{2}$, where $A^1 = 26 \leq \binom{8}{2} = 28$, as shown in Figure 6.
2) Design choices under the constraint $A^s \leq \binom{m}{k}$: Stride $s = 1$, bit group size $m = 8$, subset size $k = 2$. A single STE can contain $2^N/m = 32$ bit groups, and

Table I
SUBSET ENCODING TRADEOFF EXAMPLES - DNA (${}_mC_k = \binom{m}{k}$)

Stride	Encoding	STE Capacity	Input Rate	Efficiency Factor
1	${}_{256}C_1$	1	1.0x	1.0
1	${}_4C_1$	64	1.0x	64.0
2	${}_{16}C_1$	32	2.0x	64.0
2	${}_7C_2$	72	1.0x	72.0
2	${}_6C_3$	84	0.67x	56.0
4	${}_{256}C_1$	4	4.0x	16.0
4	${}_{24}C_2$	40	2.0x	80.0
4	${}_{13}C_3$	76	1.33x	101.3
4	${}_{11}C_4$	92	1.0x	92.0
125	${}_{256}C_{127}$	125	0.98x	123.0

the pattern has $M = 32$ bit groups.

3) The encoding function F maps each character to a 2-subset out of 8: $F = \{\{a, b, c, \ldots, z\} \mapsto \{\{0,1\},\{0,2\},\{0,3\},\ldots,\{4,7\}\}\} + \textit{offset}$, where for the n-th character in the input stream, $\textit{offset} = (n \bmod M) \times 8 = (n \bmod 32) \times 8$.
4) As a result, the capacity of a single STE is $2^8/8$ = 32, and the input rate after encoding is $s/k = 1/2 = 0.5$x. So we encode 32 English lowercase letters in one self-loop STE in exchange for halving the input rate.

E. Subset Encoding Tradeoff Analysis

There are many tradeoffs among the alphabet size of application data, input stride, input rate, and subset encoding complexity. We can analyze these tradeoffs based on the model of the subset encoding method.

From equation (5), given the alphabet size A of the application data, the stride s is bounded by bit group size m and subset size k. A larger stride s means faster input rate, but we need to choose larger k and m to satisfy the constraint. However, when m is increased, we must put fewer bit groups in an STE. When k is increased, the input rate decreases and the complexity of encoding is increased.

Table I shows how strides and encoding approaches affect the STE capacity and the input rate. Given a stride, we may have many different encoding approaches to use. The efficiency factor, which is the product of STE capacity and the input rate, roughly indicates encoding efficiency. Subset-encoded automata achieve better efficiency than traditional multiple-input stride implementations [8], since traditional designs only map 4-stride DNA characters to $\binom{256}{1}$.

F. Encoding Overhead Analysis

Since we move some computational burden from the matching phase to the encoding phase, there is encoding overhead. To encode each character, we need to do a one-to-one character-to-subset mapping and add the bit group offset to the encoding. For small subset size k, the mapping can be done through simple arithmetic calculation or table lookup. However, if k is large, it may be difficult to efficiently map input data to subsets, and the encoding rate may be slower than the input rate of the AP.

As long as the rate of data encoding is greater than the input rate of the current AP hardware (133 MB/s) then the encoding step is effectively "free" when pipelined with the input stream on either a CPU or FPGA. Our experiments show that we can easily compute the subset encoding of DNA sequences at a rate of more than 140 MB/s with a C program on a 2.7GHz single-threaded CPU. The subset encoding on CPU can be pipelined with the AP by splitting the input sequence into chunks and sending encoded chunks to the AP through PCIe interface. In the case that the same input will be used many time with the same encoding, we can even store the encoded DNA sequences so that we do not need to encode them multiple times.

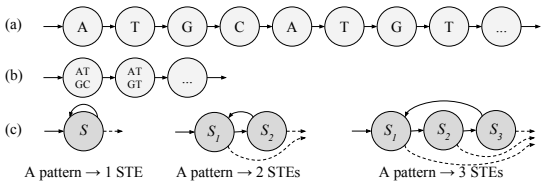

Figure 7. Exact DNA string matching on the AP. (a) Traditional implementation: n STEs for n DNA characters with 1x input rate. (b) 4-stride implementation: $n/4$ STEs for n DNA characters with 4x input rate. (c) Subset-encoded implementation: $n/64$ STEs for n DNA characters with 1x input rate, where S_i is an subset-encoded pattern. The choice of dashed output transition depends on the length of pattern modulo STE count.

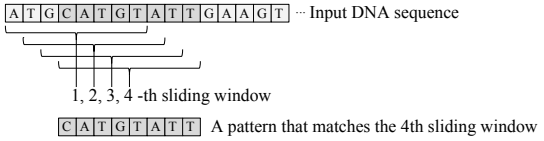

Figure 8. Sliding windows in matching applications. Matches can occur on any sliding window.

IV. SUBSET-ENCODED AUTOMATA DESIGNS

In this section, we introduce some automata designs applying the subset encoding method, including exact matching automata, Hamming distance automata, Levenshtein automata and Damerau-Levenshtein automata.

A. Exact String Matching

The exact string matching problem determines if a given input string is exactly the same as a given pattern. Traditional designs put string patterns in singly linked STE chains and use each STE to match with one character or one stride of characters, as shown in Figure 7 (a) and (b).

In many applications, we need to determine whether a pattern matches with any substring in a long input sequence. In this case, it is important to support a sliding window matching procedure. As shown in Figure 8, a sliding window is a fixed length window which, as it moves over the input stream, defines a substring to match against the given pattern. The AP simulates NFA transitions in lock-step, which means each input character can only match with successors of currently activated STEs, and there are no ε-transitions. Thus, a singly linked STE chain can match with a longer input sequence in a *pipelined* way and produce results on every cycle. In other words, we can get the matching

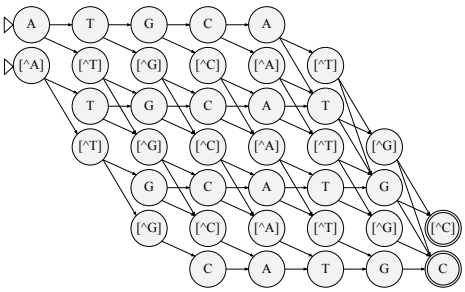

Figure 9. Traditional Hamming distance automaton (8, 3) for matching the 8-character pattern "ATGCATGC" with bounded Hamming distance 3

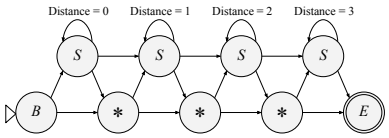

Figure 10. Subset-encoded Hamming distance automaton $(8, 3)$ for matching an input string with a pattern of length 8 within bounded Hamming distance 3. Each mismatch will make the leftmost activated state move one step right. S is the subset-encoded pattern, B is a bit group for controlled beginning, and E is another bit group for controlled reporting. For example, a pattern "ATGCATGC" will be encoded as $S = \{0,5,10,15,16,21,26,31\}$, and B can be the eighth bit group $\{28,29,30,31\}$, and E can be the first bit group $\{0,1,2,3\}$. The '*' is the "any character" wild card symbol used by the AP.

results of every sliding window of the length of the pattern along the entire input sequence.

However, the character-OR ability of STEs are not efficiently utilized, especially when sliding windows are not required, or when the application alphabet size is very small.

If we apply the subset encoding method, we can compress a sequence of data into one STE, as shown in Figure 7 (c). For longer patterns, we can use multiple STEs to construct a loop structure, then put the first character into the first STE, and put the second character into the second STE, etc. On the current generation of the AP up to 64 DNA characters can be encoded into one STE, so we need $n/64$ STEs for n consecutive DNA characters. As a result, for those applications in which sliding windows are not required, we can place 64x more patterns than straightforward implementations and 16x more patterns than 4-stride implementations on the same amount of hardware.

For supporting sliding windows, we need to replicate the subset-encoded automata multiple times. As a result, in some cases the encoded automaton may have more states than the traditional automaton. Even in this case the encoded automaton may still have a routing advantage. We discuss this in detail in Section IV-C.

B. Subset-encoded Hamming Distance Automata

Traditional Hamming distance automata designs [3] use $(2d+1)l-2d^2$ STEs for bounded Hamming distance (l, d). An example of Hamming distance (8, 3) is shown in Figure 9. This automata structure supports sliding windows over the input stream. However, for large l and d, the resulting automata are difficult to route on the AP, achieving only 16% STE utilization for the (60, 10) case.

A subset-encoded Hamming distance automaton for (l, 3) is shown in Figure 10. For Hamming distance d, we construct a d-level ladder structure to match input strings with patterns within distance d. The STEs in upper row are self-loop STEs containing an encoded pattern so that they remain activated as long input characters match with the patterns. The lower row of STEs capture mismatches. As a result, when a substitution occurs, the leftmost activated self-loop STE will be turned off, and the activation state will move one step right. If the number of substitutions is $\leq d$, this automaton will accept the input string. Otherwise, the activation chain will exit this structure, thus all STEs will be shut down before reaching the end of the input and there will be no report at the end.

Compared to the straightforward Hamming distance automata in Figure 9, which needs $(2d + 1)l - 2d^2$ STEs, the subset-encoded Hamming distance automata only use $2d + 2$ STEs for Hamming distance (l, d) when $l \leq 64$. This new subset-encoded automata design can significantly reduce the automata size, improving routability in hardware. For example, for Hamming distance (64, 10), the traditional solution needs 1144 STEs, while the subset-encoded solution only uses 22 STEs, which is a 52x improvement.

The above subset-encoded Hamming distance structure (which only decides whether the input was within distance d of the pattern) can be easily extended for calculating the actual Hamming distance (which finds the number of substitutions required to transform the input into the pattern). To do this, we link extra reporting STEs to the self-loop STEs in the upper row of Figure 10 to find the actual number of substitutions. Alternatively we can use the threshold counters with the subset-encoded automata to count the Hamming distances. The counter solution uses fewer STEs than the former solution. However, since counters are a scarce resource on the AP (there are 768 counters per core) they are best used for large distance d.

C. Hamming Distance Automata with Sliding Windows

Many pattern matching applications, such as DNA alignment, require matching a long input sequence with some short patterns and determine the start positions of all matching subsequences. A single subset-encoded Hamming distance automaton cannot naively support this sliding window approach, since we reuse STEs for a sequence of input data.

To support sliding windows, we replicate the subset-encoded Hamming distance automata, as shown in Figure 11. The key ideas for supporting sliding windows include: 1) Replicate the subset-encoded Hamming distance automata l times. 2) Each replicated automaton contains a unique shift of the given pattern. For example, the result of shifting pattern "$c_0c_1c_2...c_n$" one step right is "$c_nc_0c_1...c_{n-1}$". This shifted pattern can match with string "$c_0c_1c_2...c_n$" starting from different positions. 3) Using characters from a particular bit group in starting and reporting STEs to control when to start matching and when to report.

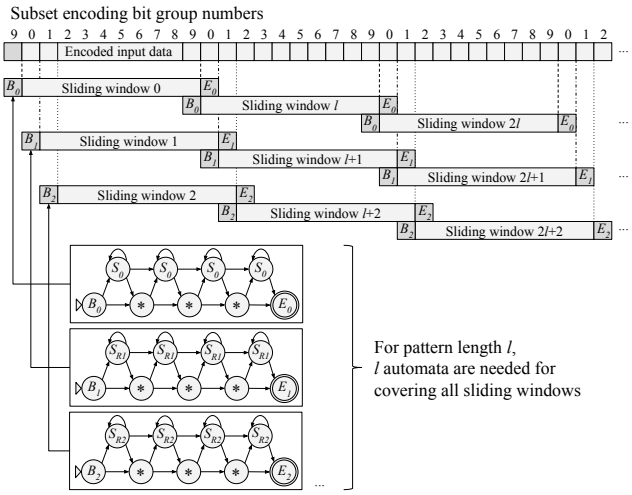

Figure 11. A subset-encoded Hamming distance automaton modified to support a sliding window. This is achieved by replicating a subset-encoded Hamming distance automaton with shifted encodings of the same pattern. S_{Ri}: Encoded shifted patterns. B_i, E_i: Characters from specific bit groups for controlled starting and reporting

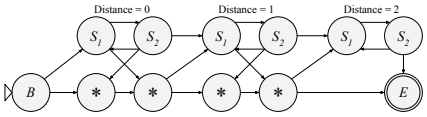

Figure 12. Subset-encoded Hamming distance automata with 1-to-many encoding. Two consecutive mismatches may only increase distance by 1

As shown in Section V-A, we can place up to 192x more subset-encoded Hamming distance automata on the AP. In the sliding window case, we can still place up to 3.2x more patterns than traditional Hamming distance automata, even if we replicate the subset-encoded automata l times.

D. One-to-Many Encoded Hamming Distance Automata

A one-to-many encoding (stride $s = 1$, subset size $k > 1$) will decrease the input rate (since the encoded data is longer than the application data), but it can allow us to encode even more characters into each STE, because according to Equation 5, a larger k will allow us to decrease m. For example, if we map English lowercase letters to 1-subsets of 26 bits, we can only encode 9 English letters into one STE. But if we map them to 2-subsets of 8 bits, we can encode 32 English letters into one STE.

Because each input character is encoded to multiple bytes, multiple mismatches related to the same original input character should only increase Hamming distance by 1. For example, we can encode letter "a" to "0, 1", when we input "0, 2" (the encoding of letter "b"), there is one mismatch, while when we input "1, 2" (the encoding of letter "h") there are two mismatches. However, both cases should increase distance by 1. As a result, we need extra STEs to delay the activations to correctly count the Hamming distance.

Figure 12 shows the one-to-many subset-encoded automaton for Hamming distance (32, 1) of English lowercase letter patterns. The ladder structure is similar to Figure 10, except we put the 2-subset encoding of characters into two STEs.

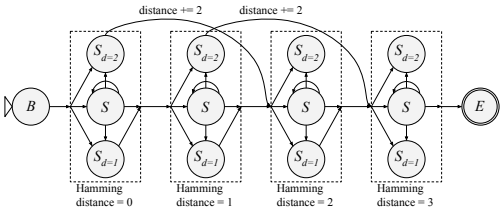

Figure 13. 2-stride subset-encoded Hamming distance automaton. Where S is the encoded pattern and $S_{d=i}$ contains all neighbors with distance i. The 2-stride mapping function can be F={$\{AA, AC, AG, AT, CA, ..., TT\} \mapsto \{0, 1, 2, 3, 4, ..., 15\} + offset$}. For a 2-stride character "AA", S = {AA}, $S_{d=1}$ = {AC, AG, AT, CA, GA, TA}, and $S_{d=2}$ = {CC, CG, CT, GC, GG, GT, TC, TG, TT}

For example, 'a' will be encoded as a 2-subset $\{0, 1\}$. Then we put '0' in the first STE, and '1' in the second.

E. Multiple-Input Stride Hamming Distance Automata

When patterns are too short to fully utilize an STE, we can use multiple-input stride to increase the input rate and encode more data into each STE. As the dual to one-to-many encoding, a single mismatch with multiple-input stride encoding may represent multiple substitutions between the input sequence and the pattern. For example, if we encode 2-stride DNA characters "AC" into a single encoding "1", a mismatched input "2" (encoding of "AG") should increase Hamming distance by 1, while another mismatched input "15" (encoding of "TT") should increase Hamming distance by 2. Thus, when a mismatch occurs, we need to distinguish the actual Hamming distance. The subset encoding method can conveniently support this by separating encoded alphabet into multiple sets according to actual Hamming distances.

Figure 13 shows an example of encoding 2-stride DNA characters into $\binom{16}{1}$. We can encode 32 DNA characters into one STE and achieve a 2x input rate increase. We map the 16 combinations of two DNA characters to numbers from 0 to 15. For each specific combination of two DNA characters, we use two STEs to separate the distance-1 neighbors and distance-2 neighbors. If a distance-1 mismatch occurs, we move one step to the right on the ladder structure, and if a distance-2 mismatch occurs, we move two steps to the right.

F. Subset-encoded Levenshtein Automata

The Levenshtein distance or edit distance is the minimum number of edits that can convert one string to the other. An edit can be an insertion, a deletion, or a substitution. In practice, the edit distance can be used in DNA alignment or spell correction. A classic dynamic-programming-based Levenshtein automaton is shown in Figure 14 (a). When implementing Levenshtein automata on the AP, we need to use extra STEs and more connections to support *-transitions and ε-transitions, and we need to propagate the start and accept states along the ε-transitions, as in Figure 14 (b). As a result, the large automata structures make routing very inefficient, or even impossible [7].

By applying the subset encoding method, we can design smaller Levenshtein automata which can significantly

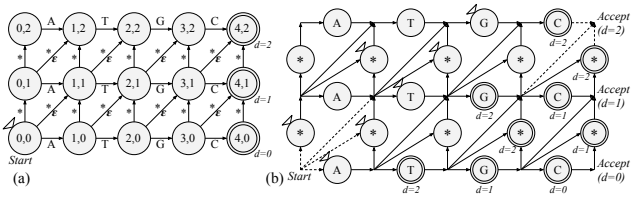

Figure 14. Classic Levenshtein automata for pattern "ATGC" within edit distance 2. (a) Traditional NFA representation. Insertions are captured by vertical *-transitions. Substitutions are captured by diagonal *-transitions. Deletions are captured by diagonal ε-transitions. Tuples in states are (character offset, edit distance). (b) Homogeneous Levenshtein Automata on the AP. Many transitions are merged together for simplifying the figure (though we can not merge them on the AP). Notice that the starting and accepting states are propagated because of the ε-transitions. This structure is routing-intensive due to the large number of transitions.

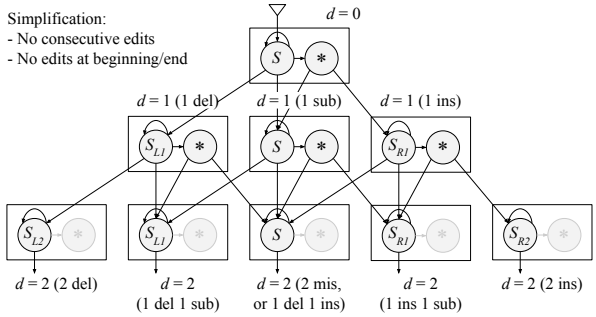

Figure 15. A subset-encoded Levenshtein automaton for edit distance ≤ 2. S, S_{Li}, S_{Ri} are subset-encoded patterns where S_{Li} and S_{Ri} represent automata shifted i characters left and right respectively. A deletion can be captured by jumping from S to its left-shifted pattern S_L, while an insertion can be captured by jumping from S to its right-shifted pattern S_R after a substitution. d represents edit distance; ins represents an insertion; del represents a deletion; sub represents a substitution

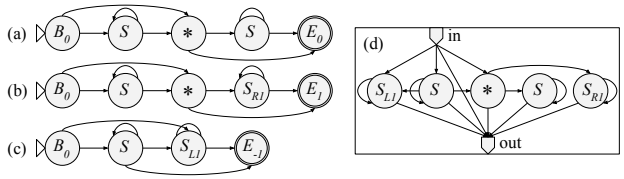

Figure 16. Separated subset-encoded automata for each type of edit. (a) 1 substitution. (b) 1 insertion. (c) 1 deletion. (d) A combined subset-encoded automaton for any 1 edit. We create a subset-encoded automaton for higher edit distance by connecting the output of this widget to three next-level widgets. One each for shift left, shift right, and no shift.

increase the routing efficiency. The subset-encoded Levenshtein automata are adapted from the subset-encoded Hamming distance automata. The key idea is that a deletion can be captured when an activation transfers to a left-shifted pattern, and an insertion can be captured when an activation transfers to a right-shifted pattern after a substitution.

A subset-encoded Levenshtein automaton within distance 2 is shown in Figure 15. For patterns that fit into one STE, the total number of STEs only depends on the distance. As a result, this new design can achieve higher utilization and pattern density on the AP.

The subset-encoded Levenshtein automata provide two levels of separation: 1) Using separated automata to support sliding windows. 2) Using separated automata to recognize

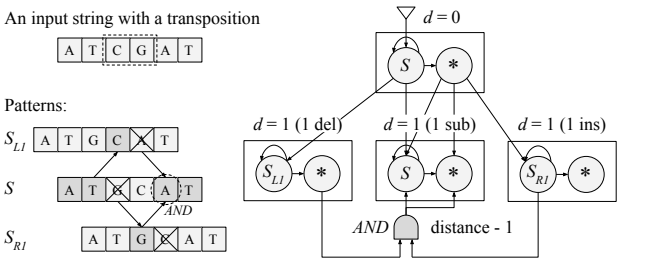

Figure 17. Basic structure of the subset-encoded Damerau-Levenshtein automata. A transposition of adjacent characters should only increase distance by 1. AND gates are used for properly capturing transpositions.

different edits. The three separated subset-encoded Levenshtein automata for one substitution, insertion and deletion are shown in Figure 16 (a), (b) and (c) respectively. We combine these three types of automata together to construct a widget for edit distance 1, as shown in Figure 16 (d). We create a subset-encoded automaton for higher edit distance by connecting the output of this widget to three next-level widgets: one each for shift left, shift right, and no shift. Compiling results for the subset-encoded Levenshtein automata are shown in Section V-C.

G. Subset-encoded Damerau-Levenshtein Automata

The Damerau-Levenshtein distance is an extended version of edit distance that considers the distance of transposition of adjacent characters as a singular edit. We modify the the subset-encoded Levenshtein automaton to support Damerau-Levenshtein distance by adding AND gates. The idea here is that if a single edit can be considered as any of an insertion, deletion, or substitution then that edit could also be considered a transposition. The basic structure of the subset-encoded Damerau-Levenshtein automata is shown in Figure 17. While traditional Levenshtein automata will count transposition as distance 2, this new structure uses AND gates (supported on the AP using Boolean elements) to increase distances by only 1 for transposition.

H. Supporting General Regular Expressions

Besides the automata designs shown in previous sections, there are many other ways to design automata with the subset encoding method, which provides a huge potential to improve the space efficiency. For example, the subset encoding method can support general regular expressions with a large alphabet size, such as 4-byte characters, because we can map these characters of large alphabet size to subsets which can be put in one STE. In addition, the wild card character "." (or "*" on the AP) can be represented by setting all bits in a bit group to 1. The OR operation in regular expressions is implemented by setting unions of subsets to 1, though ambiguities arise from one-to-many encoding.

V. Experimental Results

To evaluate the efficiency and performance of the proposed subset encoding method, we show both compiling results and running time of DNA k-mer searching using real-world data. The compiling results of automata are collected

Table II
COMPILING RESULTS OF HAMMING DISTANCE AUTOMATA

D	SE_d	$C_{30,d}$	$X_{30,d}$	$X_{1/30}$	$C_{60,d}$	$X_{60,d}$	$X_{1/60}$
1	3072	120	25.6x	0.9x	60	51.2x	0.9x
2	1536	50	30.7x	1.0x	16	96.0x	1.6x
3	1536	24	64.0x	2.1x	12	128.0x	2.1x
4	960	24	40.0x	1.3x	8	120.0x	2.0x
5	960	16	60.0x	2.0x	6	160.0x	2.7x
6	768	12	64.0x	2.1x	6	128.0x	2.1x
7	768	10	76.8x	2.6x	4	192.0x	3.2x
8	576	9	64.0x	2.1x	4	144.0x	2.4x
9	480	9	53.3x	1.8x	4	120.0x	2.0x
10	480	6	80.0x	2.7x	3	160.0x	2.7x

– D: Hamming distance
– SE_d: # of subset-encoded Hamming-dist automata per core
– $C_{30,d}$, $C_{60,d}$: # of traditional Hamming-dist automata (30, d) and (60, d) per core
– $X_{30,d}$, $X_{60,d}$: Speedup of the subset encoding method on (30, d) and (60, d)
– $X_{1/30}$, $X_{1/60}$: Speedup of the subset encoding method when replicate l times to support sliding window

Table III
DNA HAMMING DISTANCE ON 200MB INPUT AND 20,000 25-MERS

D	$C_{25,d}$	$Time_1$	$SE2_d$	$Time_2$	X	PatMaN	Bowtie
1	144	4.5s	1536	4.5s	1.0x	106s	353s
2	64	7.5s	1536	4.5s	1.7x	1080s	841s
3	32	15.0s	1008	6.8s	2.2x	>2hr	1731s
4	32	15.0s	672	9.8s	1.5x	>9hr	–
5	24	21.0s	480	12.8s	1.6x	–	–

– D: Hamming distance
– $C_{25,d}$: # of traditional Hamming-dist automata (25, d) per core
– $Time_1$: Solving time of traditional automata
– $SE2_d$: # of 2-stride subset-encoded Hamming-dist automata per core
– $Time_2$: Solving time of 2-stride subset-encoded automata
– PatMaN, Bowtie: Solving time of PatMaN and Bowtie

from the Micron AP SDK compiler. The AP execution time is derived using the 133MB/s input rate and the compiling results. Single-threaded CPU experiments are run on servers with 3.3GHz Intel(R) i7-5820K CPU and 32GB RAM.

A. Compiling Results for Hamming Distance Automata

To demonstrate the benefit of the subset encoding method, we first compare the compiling efficiency between the traditional Hamming distance automaton and the subset-encoded Hamming distance automaton on $(30, d)$ and $(60, d)$ (Table II). Traditional implementations use the structure shown in Figure 9. The subset-encoded Hamming distance automata use the more succinct ladder structure shown in Figure 10.

For Hamming distance (l, d), we need to replicate the subset-encoded automata to support the sliding windows, as shown in Figure 11. Compiling results show that even if we replicated the subset-encoded automaton l times (thus divide the number of patterns by l), we can still place up to 3.2x more patterns on the AP. If the sliding window is not required, scaling is unnecessary, thus subset encoding can place up to 192x more patterns than traditional solutions.

B. DNA k-mer Searching

The problem of matching DNA/RNA k-mers against reference sequences to identify regions of similarity is ubiquitous, as it is essential to many biological applications. In this subsection we demonstrate the contribution of the subset encoding technique for a 2-stride Hamming distance. We estimate the AP's execution time for matching 20,000 25-mers with 200 million DNA sequences in Table III.

Since we can encode 64 DNA characters into one STE using a $\binom{4}{1}$ encoding and 25-mers are relatively short, they can not fully utilize all memory cells in each STE. Thus, using the subset encoding method, we make a tradeoff among input rate and the length of patterns that can be encoded into one STE. We use a 2-stride subset encoding, which maps pairs of DNA characters to subsets of $\binom{16}{1}$. As a result, we can encode up to $256/16 * 2 = 32$ characters into one STE and get 2x input rate at the same time.

The capacity and processing time of the AP are shown in Table III. The AP processing time is calculated by 200MB/133MB/s×*pass*, where *pass* is the total number of 20,000 patterns divided by the number of patterns that can be placed on an AP board. Results show that the 2-stride subset-encoded Hamming distance automata can be up to 2.2x faster than traditional Hamming distance automata solutions.

The AP solution is more suitable for application scenarios in which reference sequences are frequently changed, because it takes DNA reference sequence as input. For comparison, we list the single-threaded running times of two well-established CPU DNA aligning tools (PatMaN [10] and Bowtie [11]) on this same problem.

Table IV
COMPILING RESULTS OF LEVENSHTEIN AUTOMATA

Levenshtein	d=1	d=2	d=3	d=4
Traditional (64, d)	24/core	10/core	6/core	failed
SE (64, d)	1536/core	288/core	128/core	48/core
SE (64, d) sliding	24/core	4.5/core	2/core	0.75/core

C. Compiling Results for Levenshtein Automata

Traditional Levenshtein automata on the AP have very low STE utilization efficiency. According to the results from the Micron AP SDK compiler, we can only compile 24 traditional instances of (64, 1) in an AP core with 25.8% STE utilization, 10 instances of (64, 2) with 17.5% utilization, and 6 instances of (64, 3) with 13.3% utilization. In addition, on the current AP generation, the fan-in of each STE is bounded by 16. As a result, traditional Levenshtein automata for edit distance ≥ 4 fail to compile due to the intensive transitions caused by ε-transitions.

The subset-encoded Levenshtein automata mitigate this routing problem by factoring the large automaton structure into smaller pieces. Table IV shows the compiling results of traditional Levenshtein automata versus the subset-encoded Levenshtein automata. We can see that the subset encoding method can provide capacity for up to 64x more Levenshtein automata for applications without sliding windows. It also makes edit distance $d \geq 4$ feasible on the AP, which was not possible to route using the traditional structure.

Table V
TRADEOFF BETWEEN AP SYMBOL SIZE AND THE NUMBER OF STES

c	4	6	8	10	12	14	16
n	393216	98304	24576	6144	1536	384	96
n'	3145728	786432	196608	49152	12288	3072	768

– c: The symbol size of the AP
– n: The number of STEs on one current AP core
– n': The number of STEs on one AP core with 8x semiconductor technology scaling

VI. FURTHER DISCUSSION

A. Technology Scaling of the AP

The current AP generation we are working on is based on 50nm technology, while the state-of-the-art DRAM is using 20nm technology, which is three generations ahead (37nm, 25nm, 20nm). Thus, if we normalize the AP to the state-of-the-art semiconductor technology and assume there is a 2x improvement on capacity per node, we can place 8x patterns on a future generation of the AP with 20nm technology.

B. Impact on the Architecture of the AP

This new subset encoding technique brings up interesting considerations for the design of future AP architectures. There is a tradeoff between the native symbol size (i.e., how large the alphabet is) and the number of states. In Table V, we show how the AP symbol size affects the number of STEs in one AP core. We assume the total number of memory cells are constant, and consider two scenarios: the current generation, and projecting an 8x increase in capacity after semiconductor technology scaling. Since Micron has not published details of its routing matrix, we assume the fraction of area of the routing matrix is fixed. We can see from the table that small symbol sizes benefit applications with small alphabet size by providing more STEs, such as DNA applications. Large symbol sizes benefit applications that have larger alphabet sizes or long patterns without requiring sliding windows since they allow for a larger character set to be represented in one STE, and thus longer sequences by subset encoding. Note that the aspect ratio of the DRAM structures must also be taken into account. DRAM banks with high aspect ratios are not efficient, so the two ends of the scale in Table V may not be practical, because they would lead to very wide or very tall banks.

In addition, the subset encoding method can take advantage of the on-board FPGA for preprocessing or post-processing. We can stream in the original data to the FPGA on the AP board, so that we do not need to preprocess the application data on CPU.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we present the subset encoding method and related automata designs for improving the pattern density on the Automata Processor. The proposed method is a general method that can take advantage of the character-OR ability of STEs on the AP, and it relieves the problems of limited hardware capacity and inefficient routing. Experimental results show that after applying the subset encoding method on Hamming distance automata, we can place up to 3.2x more patterns on the AP if sliding window is required, and 192x more if sliding window is not required. For Levenshtein distance, the subset encoding splits the Levenshtein automata into small chunks and makes them routable on the AP, allowing higher edit distance to be supported. In addition, the subset encoding method influences future decisions in the design of the AP or other automata-based co-processors. This idea of encoding sequences of data into subsets before doing NFA matching can also be applied to CPU, GPU or FPGA regular expression matching implementations. By doing problem reductions, we may be able to utilize those hardware accelerators more efficiently.

ACKNOWLEDGMENT

This work was supported in part by the Center for Future Architecture Research (C-FAR) and NSF grant no. EF-1124931.

REFERENCES

[1] 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 & Distributed Systems*, no. 12, 2014.

[2] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, vol. 16, no. 5, 1961.

[3] I. Roy and S. Aluru, "Finding motifs in biological sequences using the Micron Automata Processor," *IEEE IPDPS*, May 2014.

[4] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the Micron Automata Processor," *IEEE ICSC*, February 2015.

[5] K. Wang, M. Stan, and K. Skadron, "Association rule mining with the Micron Automata Processor," *IEEE IPDPS*, May 2015.

[6] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using Micron's Automata Processor," *ASBD workshop, with ISCA*, June 2015.

[7] T. Tracy II, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins, "Nondeterministic finite automata in hardware - the case of the Levenshtein automaton," *ASBD workshop, with ISCA*, June 2015.

[8] M. Becchi, *Data structures, algorithms and architectures for efficient regular expression evaluation*. PhD thesis, Dept. of CSE, Washington University, 2009.

[9] S. Even, A. L. Selman, and Y. Yacobi, "The complexity of promise problems with applications to public-key cryptography," *Information and Control*, vol. 61, no. 2, 1984.

[10] K. Prüfer, U. Stenzel, M. Dannemann, R. E. Green, M. Lachmann, and J. Kelso, "PatMaN: rapid alignment of short sequences to large databases," *Bioinformatics*, vol. 24, no. 13, 2008.

[11] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome biol*, vol. 10, no. 3, 2009.