FlexAmata: A Universal and Efficient Adaption of Applications to Spatial Automata Processing Accelerators

Elaheh Sadredini[1], Reza Rahimi[2], Marzieh Lenjani[1], Mircea Stan[2], and Kevin Skadron[1]

1. Department of Computer Science, University of Virginia
2. Department of Electrical & Computer Engineering, University of Virginia

{elaheh,rahimi,ml2au,mircea,skadron}@virginia.edu

Abstract

Pattern matching, especially for complex patterns with many variations, is an important task in many big-data applications and maps well to finite automata. Recently, a variety of research has focused on hardware acceleration of automata processing, especially via spatial architectures that directly map the patterns to massively parallel hardware elements, such as in FPGAs and in-memory solutions. We observed that all existing automata-acceleration architectures are designed based on fixed, 8-bit symbol processing, derived from ASCII processing. However, the alphabet size in pattern-matching applications varies from just a few up to billions of unique symbols. This makes it difficult to provide a universal and efficient mapping of this wide variety of automata applications to existing automata accelerators.

In this paper, we present FlexAmata, a compiler solution for efficient adaption of applications with any alphabet size to existing pattern-matching accelerators. We demonstrate that this can increase automata processing efficiency in two ways. First, this improves resource utilization for applications with small alphabets and enables hardware acceleration for applications with very large alphabets (which otherwise would not map to hardware accelerators). Second, this enables the exploration of optimized bitwidth processing for future spatial hardware accelerators. We leverage FlexAmata and investigate the hardware implications of different bitwidth processing rates on the two state-of-the-art spatial accelerators, Cache Automaton (CA) and FPGAs. Our exploration across a wide range of automata benchmarks reveals that 4-bit processing rate on CA and 16-bit processing rate on FPGAs results in higher performance than the default 8-bit processing rate in these existing approaches.

CCS Concepts. • **Computer systems organization** → **Multiple instructions, single data**; • **Hardware** → Emerging architectures; • **Theory of computation** → Formal languages and automata theory.

Keywords. automata processing; memory-centric accelerators; reconfigurable computing; FPGAs; compiler

ACM Reference Format:
Elaheh Sadredini[1], Reza Rahimi[2], Marzieh Lenjani[1], Mircea Stan[2], and Kevin Skadron[1]. 2020. FlexAmata: A Universal and Efficient Adaption of Applications to Spatial Automata Processing Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378459

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378459

1 Introduction

Identifying complex patterns within large datasets is a prominent and time-consuming task in many big-data applications. Finite automata are an efficient computational model for widely used pattern matching languages such as regular expressions. There are many applications in domains such as data mining [9, 35, 46–48], bioinformatics [8, 30, 40], and natural language processing [31, 56] that have been shown to greatly benefit from accelerated automata processing. The automata structure in these applications differs significantly in static structure and dynamic behavior from common regex structures [1, 43]. Moreover, there is a large body of research on using finite automata in formal verification such as satisfiability and model-checking problems in temporal logics [36, 39, 41], and these can benefit from high-performance automata processing, especially for real-time verification.

The growing demand for accelerated automata processing has motivated many efforts in designing *Spatial* automata accelerators, such as processing-in-memory (PIM) architectures [14, 34, 37, 38] and FPGA solutions [18, 20, 50, 53, 54], where they have shown significant speedup over the existing CPU- and GPU-based solutions on a wide range of applications [38, 45]. *Spatial* memory-centric accelerators provide

a reconfigurable substrate to lay out the rules in hardware by placing and routing states and connections onto a pool of hardware units in logic- or memory-based fabrics. This allows a large number of automata to be executed in parallel, up to the hardware capacity, to provide a high-throughput automata computation. If an application is too large to fit in a given hardware capacity, in many cases, multiple passes over the input are required. Liu et al. [25] show that many applications can be partitioned to cold and hot states so that only the most frequently-used (hot) states run on the accelerator, and the infrequent (cold) states are supported on the CPU.

Because regular expressions have most commonly been used for text, packet, and other byte-oriented processing, all existing automata accelerators are designed based on an 8-bit (ASCII) symbol processing scheme, similar to software solutions [6, 12, 19, 42, 49]. This means that the automata structure is based on 8-bit symbols, and an 8-bit input is processed in each cycle. However, real-world automata applications can have a very small or very large alphabets, and we observed that directly adapting these applications to the spatial automata accelerators (1) causes area and throughput inefficiencies, (2) limits the application generality on these accelerators, and (3) diminishes the benefits of automata optimization techniques, such as compression.

In memory-based solutions, symbols are encoded in memory columns, such that each symbol activates a different row of memory. This tends to reinforce designs based on 8-bit symbols because 256 (2^8) is a fairly conventional subarray height. However, this can be extremely inefficient, especially when the application alphabet (symbol-set) size is very small, and the number of rows in a subarray is more than required. For example, in genomics, the alphabets are A, T, C, and G, and a 2-bit automata organization with only 2^2 rows is enough to perform the string matching. This is 64× smaller than what existing spatial architectures provide!

On the contrary, the 8-bit symbol processing architectures can limit the generality of the architecture for applications that have more than 256 symbols. For example, in sequential pattern mining [48], the input database can be quite large, such as market basket analysis datasets from Amazon, and the number of unique items (or symbols) can be very large (on the order of 2^{20} or more). In formal verification problems, the symbols map to the events, and thus, the automata symbol-set size can be extremely large [41, 57]. However, due to delay, power issues, and signal integrity, it is impractical to change the hardware to support 2^{20} rows in each memory subarray. Moreover, simply daisy-chaining multiple states to support larger alphabets results in false report generation. Therefore, an efficient and accurate symbol-size transformation technique is required.

Another problem with the existing 8-bit approach for spatial accelerators is that, if the memory subarray size of the underlying memory technology changes, then there is a need to make sure that the application symbol-set size is still compatible with the memory architecture. For example, Cache Automaton (CA) [38] re-purposes caches in conventional processors for automata processing. If the number of rows in the subarrays of a cache structure changes, then the automata structure and input bitwidth consumption need to be changed for correct functionality and full hardware utilization.

To address these issues, this paper presents FlexAmata, a compiler solution that decouples applications (with any alphabet size) from the details of the memory architecture (e.g., number of rows per subarray). FlexAmata acts as an adjustable wrench and transforms an arbitrary $m-bit$ processing automaton to its equivalent $n-bit$ processing unit, where n can be larger or smaller than m, depending on the target architecture. FlexAmata offers arbitrary bitwidth processing, thus improving efficiency for small alphabets, enabling hardware acceleration for large alphabets that were nearly impossible to process efficiently up till now, and maintains application compatibility with the future automata hardware accelerators. FlexAmata, therefore, improves small-symbol-set efficiency and provides large-symbol-set compatibility even for conventional in-memory solutions such as CA. Thanks to our fine-grain, bit-level optimizations in FlexAmata, state and transition overhead of a transformed automaton is reasonably low.

All the aforementioned problems target the application's alphabet-size incompatibility with the existing automata accelerators. On the other hand, there are unanswered fundamental research questions from the hardware perspective: *what is the best bitwidth size for automata processing on spatial platforms? Does the conventional 8-bit processing offer the highest throughput-per-area?* To the best of our knowledge, this is the first work that addresses all these issues/questions on spatial automata processing architectures.

To answer these questions, we employ FlexAmata and perform a sensitivity analysis for various bitwidths using several automata applications from the ANMLZoo [43], AutomataZoo [44], and Regex [7] benchmark suites on existing memory-centric automata processing models. We apply the necessary changes to the Cache Automaton [38] hardware model (e.g., changing the height of memory subarrays) and re-calculate area and frequency parameters. We find that 4-bit processing as the base processing granularity results in 2.2× higher throughput-per-area than 8-bit processing. This efficiency comes from the fact that 4-bit processing design requires 16× shorter memory subarrays than 8-bit processing design. The area efficiency introduced by FlexAmata provides an opportunity to design denser state-matching resources, which can accommodate more states and results in fewer spatial resources. Moreover, to analyze the effect of different bitwidth processing on FPGAs, FlexAmta is equipped

with a backend HDL generator. We observe that 16-bit processing on FPGAs performs better than the 8-bit processing and has up to 4.9× higher throughput-per-LUT.

In summary, this paper makes the following contributions:

- We identify an inevitable compatibility gap between the applications (which are designed solely based on patterns and independent of the target hardware constraints) and spatial automata processing accelerators, and we observe that this causes performance and feasibility issues. To address this, we present FlexAmata, a compiler solution to bridge the application/hardware gap by transforming automata structure and providing *application compatibility* with existing and future spatial automata architectures, which allows execution efficiency and feasibility of applications with very small or large alphabets, respectively (evaluated in Section 5 using case studies).
- Furthermore, we observe that the 8-bit processing on existing automata hardware accelerators is unwisely derived from software-based computing models, without analyzing hardware-specific parameters. By leveraging FlexAmata, we explore the most efficient bitwidth processing granularity on these accelerators. To do so, (1) we change the architecture of an existing in-memory automata accelerator and (2) we develop an FPGA kernel for processing different bitwidth size. Our exploration on a large number of benchmarks concludes that 4-bit processing on the in-memory architecture and 16-bit processing on FPGAs have higher performance than the original 8-bit processing.
- We present an open-source toolkit for automata simulation, minimization, transformation, performance modeling on in-memory architectures, and performance evaluation on FPGAs.

2 Background

To better explain the claims and contributions of the paper, this section presents a simplified two-level pipeline organization used in memory-centric automata-processing architectures, such as Cache Automaton [38], the Automata Processor [14], and REAPR (BRAM-based FPGA design) [50].

Figure 1 (right) shows an example of a classic NFA and its equivalent homogeneous representation [16]. In a homogeneous automaton, all transitions entering a state must happen on the same input symbol. This provides a nice property that aligns well with a hardware implementation that finds matching states in one clock cycle and allows a label-independent interconnect. Following [14], we call this element that represents both a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE). Both automata in this example, accept the language $(A|T)(A|G)(C)^+$. The alphabet is $\{A, T, C, G\}$.

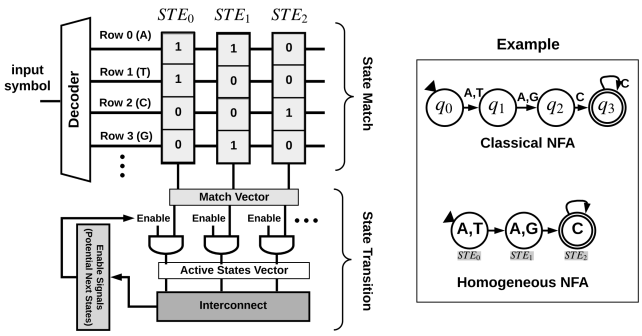

Figure 1. (Left) A simplified in-memory automata processing model. (Right) Classic and homogeneous NFA representations.

Throughout the paper, the states with a black triangle are the start states and the double circled states are final states.

Figure 1 (left) shows the simplified in-memory architecture, where memory columns are configured based on the homogeneous example in Figure 1 (a) for STE_0-STE_2. Each STE is one-hot encoded in a memory column (the "character class"). Generally, automata processing involves two steps for each input symbol, *state match* and *state transition*. In the state match phase, the input symbol is decoded, and the set of states whose rule or label matches that input symbol are detected through reading a row of memory (*match vector*). Then, the set of potentially matching states is combined with the *active state vector*, which indicates the set of states that are currently active and allowed to initiate state transitions; i.e., these two vectors are ANDed. In the state-transition phase, the potential next-cycle active states are determined for the currently active states (*active state vector*) by propagating signals through the interconnect to update the active state vector for the next input symbol operation.

In the example, there are four memory rows, and each is mapped to one symbol (i.e., A, T, C, and G). Each STE in the example is mapped to one memory column, with '1' in the rows matching the label(s) assigned to those STEs. STE_0 matching symbols are A and T, and the corresponding positions have '1'. Assume STE_0 is a currently active state. The potential next cycle active states (or enable signals) are the states connected to STE_0, which is STE_1 (the enable signal for STE_1 is '1'). Specifically, if the input symbol is 'A,' then Row0 is read into the *match vector*. A bitwise AND on the *match vector* and *potential next states* (enable signal) determines STE_1 as the current active state.

3 Prior Work

Data movement is highly expensive, much more expensive than the computation [10, 22–24]. Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and

spatial locality, which often leads to poor cache and memory behavior [43], and this increases the cost of data movement.

Several memory-centric automata processing accelerators have been recently proposed to improve the performance of general pattern matching [14, 33, 38]. The Micron Automata Processor (AP) [14] and Cacue Automata (CA) [38] propose in-memory hardware accelerators for single symbol per cycle (single-stride) automata processing. They both allow native execution of NFAs by providing a reconfigurable substrate to lay out the rules in hardware. They exploit the inherent bit-level parallelism of memory to support many parallel transitions in one cycle. The AP provides a DRAM-based dedicated automata processing chip while the CA proposes an on-chip solution by repurposing a portion of the last-level cache for automata processing and has shown higher throughput than previous solutions. Prior work has already shown that the AP is at least an order of magnitude better than GPUs and multi-core processors [45], and CA is at least an order of magnitude better than the AP [38]. All these architecture are designed based on an 8-bit symbol processing scheme, and we are not aware of any prior work that reshapes an automaton structure to efficiently adapt an application to the in-memory automata accelerators.

To increase the processing rate in regex/automata processing, a number of multi-stride (processing mutiple symbols per cycle) automata processing engines have been proposed on CPUs/GPUs [2, 12, 19, 42, 49] and FPGAs [21, 42, 52, 53]. Yang et al. [53] proposed a multi-symbol processing for regular expressions on FPGA and utilized both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicates the resources in each stride. Yamagaki et al. [52] proposed a multi-symbol state transitions solution using a temporal transformation of NFAs to construct a new NFA with multi-symbol characters. Generally speaking, the prior multi-striding solutions (on CPUs/GPUs/FPGAs) take the original 8-bit processing and increase the processing rate 2x, 4x, 8x, etc. per cycle. The main difference between FlexAmata and multi-striding approaches is that FlexAmata can provide bit-level transformation on an automaton (which is useful to adjust bitwidth based on hardware constraints), whereas multi-striding solutions only provide byte-level transformation and do not consider hardware parameters. The main purpose of FlexAmata is transforming an application to a target spatial automata accelerator architecture for higher efficiency and feasibility (and also to enable portability among architectures, current and future). For example, the BRAM memory height in recent FPGAs can be at least 512 rows [51] (and it may possibly change for future generations), which means that if we directly map an 8-bit processing automaton to FPGA's BRAM resources, at least half of the BRAM block capacity is wasted. FlexAmata can convert an automaton to 9-bit processing units to fully utilize the 512-rows memory blocks. The multi-striding techniques are orthogonal to our contribution and can be applied to process multiple 9-bit symbols per cycle in our example.

Furthermore, alphabet compression techniques [4–6, 11, 27] are employed to reduce memory requirements in CPU/GPU and FPGA-based solutions. Becchi et al. [5] propose to merge symbols with the same transition rules. This reduces the number of unique alphabets in an automaton.

FlexAmata vs. compression techniques: In alphabet-compression, each automaton alphabet is compressed independently of others, and as a result, each automaton will have a different compressed symbol-set size (which is basically a function of an automaton state, connectivity pattern, and accepting symbols of each state). For example, based on our experiment, the length of symbol size can vary from 1 to 6 bits across all the connected components in the Snort dataset after the compression technique from [5] is applied. This variation brings up two main problems for in-memory accelerators when it is used as a technique for reducing the symbol size. First, all the existing in-memory accelerators process the input with a fixed symbol size to allow a fixed rate of N symbols/cycle. However, compression techniques result in variable symbol size across the automata in a benchmark, which makes it unsuitable for spatial accelerators.

Second, assuming that by chance, the destination symbol size of the compression is compatible with the hardware parameters. However, it may potentially impose new challenges in the symbol matching subarrays. As each automaton has its own symbol compression mapping table, a single input-symbol may need to be mapped to different compressed symbols for different connected components. However, if two connected components are placed in the same memory subarray, they can only match against one common input symbol as the memory address decoder input is shared among all the rows. On the other hand, FlexAmata changes the bitwidth processing of all the automata in an application to a fixed target processing rate, which is consistent with the hardware capabilities.

4 FlexAmata

This section explains FlexAmata with a simple example. We then discuss (1) application compatibility implications in Section 5 and (2) bitwidth-size explorations on spatial architectures in Section 6, which both are derived from FlexAmata.

Algorithm: FlexAmata transforms an *m-bit* automaton A to an equivalent *n-bit* automaton B, where n can be larger or smaller than m. This transformation is done in two steps; (1) converting A to a bit-level representation (A_b), and (2) generating automaton B by transforming A_b to process *n-bit* in each cycle. To generate the *n-bit* automaton, we find all the unique paths of size n in A_b and replace each of them as a single edge (or equivalently transition rule) in B. The algorithm performs bit-level minimization on the automata and merges the states and transitions in binary

paths when applicable. Finally, automaton B is converted to its homogeneous representation to properly be configured on an in-memory platform or FPGAs.

In Figure 2, we explain how an 8-bit automaton is transformed into 3-bit and 4-bit automata. In the notation STE_x^y, x is state index and y is the bitwidth size. The original homogeneous automaton (a) has two states and accepts language $(A|B)C^+$. Using FlexAmata, we generate binary automata (b) and minimize the states when possible. For example, the first 6 bits of symbols A and B can be merged. Then, 3-bit (c) and 4-bit (d) are generated from the bit-automaton.

In the 3-bit automaton, STE_0^3 is an start state and STE_5^3, STE_8^3, and STE_{11}^3 are final states. Each state processes one or more 3-bit symbols. STE_{16}^1 in 1-bit-automata is equivalent to reaching the state STE_4^3 in the 3-bit automaton. STE_{17}^1 is a report state, and there is a loop back to the state STE_{10}^1. Assume STE_{16}^1 in 1-bit-automaton is an active state, and the next input character is "1", then it should generate a report. Equivalently in the 3-bit automaton, STE_4^3 is an active state, and because the next input is a 3-bit chunk, then, "1**" should generate a report. In order to address this unalignment in the bitwidths, we generate residual states (Res) to report when a match happens in the middle of a multi-bit input. STE_5^3 is a residual state that reports when the matching happens in the first bit of the 3-bit input. The residual states are used to locate match occurrence in the input stream accurately.

In 4-bit automaton, no residual state is needed as 8 (from original 8-bit automaton) is divisible by 4, and this property avoids mis-alignment in the input symbols. In order to get the correct functionality, we always make sure the input size is divisible by the bitwidth size by padding the input stream.

In the interest of space, we skip the details of the algorithm. Instead, we utilize the space to discuss the benefits of this transformation on the universal application adaption and parameter tuning on spatial automata accelerators. Interested readers can find the algorithm details here[1] and its implementation here[2].

Transformation Soundness: Correctness of an m-bit to an n-bit automaton can theoretically be proven using contradiction in two parts. First, the equivalence of the m-bit automaton to its 1-bit automaton, and the equivalence of the 1-bit automaton to the n-bit automaton. First, we show that for every edge in the m-bit automaton, there is one and only one unique path of length m in the 1-bit automaton (with no common edge between paths). Second, we show that for every edge in the n-bit automaton, again, there is a unique path with length n in the 1-bit automaton. Using contradiction, assuming there is an edge in the m-bit automaton that can not be mapped to a path of length m in the 1-bit automatons. However, this is impossible as the algorithm process every edge in m-bit automaton in breadth-first search (BFS) manner and for each edge, creates one path in the 1-bit version (all middle states in the 1-bit automaton are created uniquely and the destination state is added if the destination node has not been previously visited in the m-bit automaton). Similarly, it is not possible to find a path of length m in the 1-bit automaton that can not be mapped into an edge in the m-bit automaton.

In terms of implementation validation, we re-transform a reshaped automaton to the original bitwidth and check for equivalence with the original automaton. In addition, we have verified the correctness of our implementation by streaming input to both automata (original automaton and transformed version) and compared the report information between them. In both analyses, we have not found any inconsistency.

Time complexity: The offline compilation process for converting an m-bit automaton to its equivalent n-bit automaton has two main steps; (1) converting the m-bit automaton to a 1-bit automaton: assuming the input automaton has a total of *'s'* matching symbols across all the states (m bits each). For every symbol, we need to find its binary representation and add a new chain of states (with length m) for each symbol in the binary automaton. Therefore, the time complexity for this stage is $O(m \times s)$.
(2) Converting a binary automaton with t states to an n-bit automaton: we use dynamic programming with backtracking to solve this problem efficiently. Our converter implementation starts from the start states in the binary automaton and moves forward towards the report states. When a new state is reached, the path from all its parents (ancestors with a maximum distance of n) is added to a dictionary to be reused later again when necessary. In the worst-case, it needs a table of n entries for each of the t states. Each state with n entries keeps the reachable states (from itself as source) with different distance ranging from *1* to n. To calculate the missing entries in the table, we need to iterate through direct neighbors of nodes and reuse data from the existing entries of the table (if the neighbor has an entry in the table) and combine them or recursively go through their neighbors. For a binary automaton with average out-degree of e, assuming combining entries takes $O(1)$ times, the timing complexity is $O(t \times e \times n)$ where $t \times n$ is the table size, and e is the number of times that a combining operation needs to be applied.

Non-divisible bitwidths effect: As Figure 2 illustrates, the 3-bit automaton has more state and transition overhead than a 4-bit automaton. From our experiments, we observed that when re-shaping an n-bit automaton to an m-bit automaton, the overhead would be minimum if either $m \bmod n = 0$ or $n \bmod m = 0$. Figure 3 explains the reason with an original state with four 8-bit symbols. First, a 1-bit automaton is generated. Then, 3-bit and 4-bit automata are generated from the 1-bit form. In 3-bit, because 8 mod 3 ! = 0, it needs to generate more states and transitions to consider for combinations of paths when a jump is needed. However, the 4-bit

[1] https://github.com/gr-rahimi/APSim/blob/ASPLOS_AE/supp.pdf
[2] https://github.com/gr-rahimi/APSim/tree/ASPLOS_AE

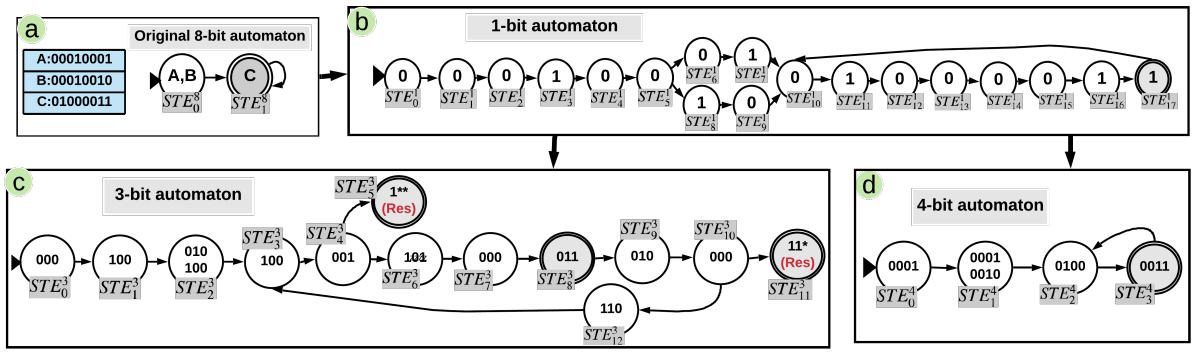

Figure 2. An 8-bit automaton (a) is converted to the minimized *1-bit* automaton (b). The *3-bit* (c) and *4-bit* (d) automata are generated from the *1-bit* automaton.

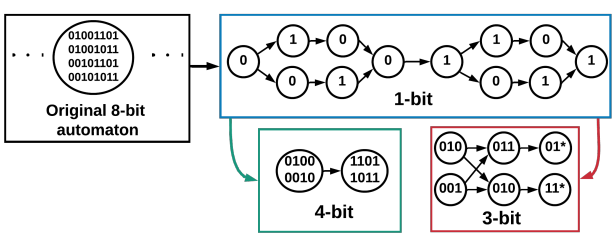

Figure 3. State and transition overhead is less in divisible bitwidths (4-bit) than non-divisible bitwidths (3-bit).

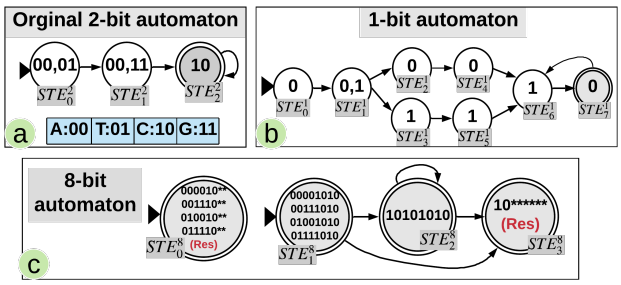

Figure 4. (a) The original automaton has 4 symbols and can be represented with two bits. (b) FlexAmata generates 1-bit from original automaton. (c) Then 8-bit is generated from 1-bit, and can process 4× more symbols.

design has a relatively very low state and transition overhead. This observation is later used to find the best bitwidth processing size for spatial accelerators.

5 Universal Application Adaption

The existing automata processing accelerators are designed based on an 8-bit symbol processing scheme. However, the applications can have a very small or large alphabet (symbol-set). FlexAmata provides application compatibility with the existing automata hardware accelerators, meaning that if the application has a small alphabet size, then FlexAmata can generate the 8-bit automata to increase the processing rate and fully utilize the hardware. On the other hand, if the application has a very large alphabet size, FlexAmata transforms the automaton into an 8-bit automaton, which provides feasibility support for the application.

5.1 Utilization for smaller symbol-sets

Figure 4 shows an example of how FlexAmata improves the utilization and throughput of an application with a small alphabet size on an existing 8-bit automata accelerator, such as CA. Assume the application has four symbols, A, T, C, and G. Therefore, 2 bits are enough to encode the symbols. Directly processing an automaton with only four symbols on an 8-bit accelerator under-utilizes the state-matching resources 64× (i.e., $2^8/2^2$). This is because, in the 8-bit scheme, state symbols are encoded in a 256-bit memory column. However, an application with 2-bit symbols only needs memory columns with four rows. By leveraging FlexAmata, we stride the original 2-bit automaton (Figure 4 (a)) and generate the 8-bit automaton (Figure 4 (C)). In this example, the 8-bit automaton has only 25% more states and 25% more transitions compared to the original 2-bit automaton. However, this transformation results in 4× higher processing rate (i.e., 4× higher throughput) with minimal resource overhead, totally based on a software solution (no hardware modification needed).

Case study: We use Levenshtein automata[3] from the AutomataZoo benchmark suite [44] as a real-world case study to evaluate the benefits of FlexAmata when adapting applications with small alphabet-size on spatial 8-bit automata accelerators. Levenshtein automata are designed to calculate the edit distance between two strings that are useful for some genomic and text-processing applications.

First, we generate a set of Levenshtein automata for different string lengths with 2% edit distance. The strings are

[3]https://github.com/tjt7a/AutomataZoo/tree/master/Levenshtein

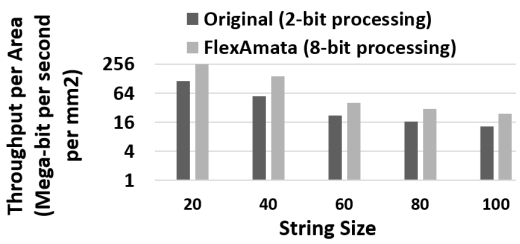

Figure 5. Comparing throughput-per-area for Levenshtein with different string lengths in original 2-bit automata design and optimized FlexAmata 8-bit design.

randomly generated with A, T, C and G symbols to resemble read-alignment in genome sequencing. Clearly, 2 bits is enough to represent the symbols and input characters. Then, using FlexAmata, we transform 2-bit automata to 8-bit automata, which processes 4 of 2-bit symbols in each cycle. Consequently, the generated 8-bit design can be efficiently processed on the existing 8-bit accelerators. To jointly consider the effect of (1) state/transition overhead (resulting from transformation) and (2) increased symbol processing rate, we evaluate the FlexAmata contribution using throughput-per-unit-area metric on CA hardware model [38].

Fig. 5 compares throughput-per-unit-area for Levenshtein with different string sizes in the original 2-bit automata design (1 symbol per cycle) and the FlexAmata 8-bit design (4 symbols per cycle). On average, FlexAmata has 2.1× more states than the original design. However, because of its higher processing rate (4 symbols per cycle), it has 2.5× higher throughput per area. This all implies that using FlexAmata as a backend compiler increases the performance of applications with smaller symbol-set sizes without the need to change the existing hardware accelerators.

5.2 Feasibility for very large alphabet-size

Many applications with pattern matching tasks, such as natural language processing with words as symbols or pattern mining with items in market basket as symbols, can have a very large alphabet size. Increasing the memory column size requires long bitlines, which is impossible without introducing stacked memory subarrays with partial address decoding and costly hardware peripherals. Moreover, simply breaking a state with 16-bit symbols to two states with 8-bit symbols (or in other words, daisy-chaining two symbols) could result in false report generations. We explain this problem by using an example in Figure 6. For simplicity, assume the target architecture supports 2-bit symbol processing, but the application has 16 unique symbols and requires support for 4-bit symbol processing. The left-side automaton (a) accepts $(0011|1100)^+$. To process this automaton in a 2-bit architecture, chaining to two states introduces false positives: the automaton in (b) breaks the STE_0^4 into two states (STE_0^2 and STE_1^2), which now can accept $(0011|1100|0000|1111)^+$.

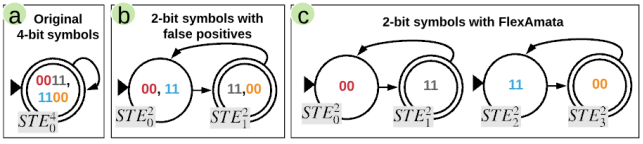

Figure 6. The problem with chaining two symbols to support larger symbol-sets.

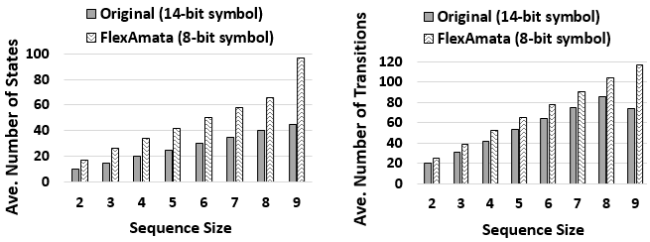

Figure 7. Comparing the number of states and transitions in different sequence sizes for the original and 8-stride automata.

To preserve the correctness of original automata and avoid false positives, FlexAmata performs minimal state splitting on the 1-bit automaton to reduce the overall state and transition overhead.

Case study: To evaluate the generality of FlexAmata, we use the SPM (sequential pattern matching) benchmark[4] from AutomataZoo [44] and generate frequent sequences for BIBLE dataset[5]. BIBLE has 13905 distinct items (symbols); therefore, support for at least 14-bit processing is required.

Fig. 7 shows the average number of states and transitions for the SPM automata in each iteration. The original automata require a 14-bit symbol processing architecture. This means that the memory should provide 2^{14} rows in each subarray, which is very costly and extremely inefficient. The symbol-set size can easily increase in a larger dataset, which makes increasing the memory column size infeasible.

FlexAmata provides a scalable and feasible solution by (1) generating 1-bit automata from the original 14-bit automata and then, (2) creating 8-bit automata from the 1-bit automata. The resulting 8-bit design provides a reasonably low-overhead and feasible solution on the existing 8-bit processing architectures, which requires only up to 2.1× more states and 1.5× more interconnect resources compared to the original automata (Figure 7). This is a very small price to pay for feasibility!

6 Hardware Implications

In this section, we explore the effect of various bitwidth size on the performance of spatial automata accelerators. We leverage FlexAmata to transform the automata across

[4]https://github.com/tjt7a/AutomataZoo/tree/master/SeqMatch
[5]http://www.philippe-fournier-viger.com/spmf

Table 1. Average state and transition overhead for different bitwidths, normalized to the original 8-bit automata.

Bitwidth	1	2	3	4	5	6	7	8	9	12	16
#States	9.9	5.2	11.2	2.3	12.0	5.9	12.9	0.98	13.7	2.9	1.1
#Trans	10.5	6.6	16.1	2.8	19.4	8.3	22.6	0.98	26.7	4.4	1.6

a diverse set of 20 applications from ANMLZoo [43] and Regex [7] benchmark suites into various bitwidth sizes and then map these automata to the spatial hardware accelerators. NFAs for real-world automata applications are typically composed of many independent rules or patterns, which manifest as separate *connected components (CCs)* with no transitions between them. Each connected component usually has a few hundred states. All the connected components can thus be executed in parallel and independent of each other.

Table 1 shows the average number of states and transitions in different bitwidths across the 20 benchmarks, normalized to the number of states and transitions in the original 8-bit automata. These averages mask some important individual application behaviors that are addressed in Section 8.1. 2-bit and 4-bit designs process one-fourth and one-half of an 8-bit symbol in each cycle, and on average, incur 5.2× and 2.4× state overhead, and 6.6× and 2.8× transition overhead, respectively, compared to the 8-bit design. However, the 8-bit design requires memory subarrays with 256 rows (see Section 2), while 2-bit and 4-bit designs only need memory subarrays with 4 and 16 rows, respectively. The significant area saving proposes the potential of more efficient in-memory automata processing in smaller bitwidths than the conventional 8-bit designs.

On average, 16-bit design incurs only 1.2× state overhead and 1.6× transition overhead and permits a higher processing rate. However, it is very costly to encode 16-bit symbols to a memory column with 2^{16} rows. On the other hand, 16-bit symbols can be efficiently stored in look-up-table (LUT) resources on FPGAs. The larger bitwidth processing (e.g., 32-bit, 64-bit, etc.) causes a dramatic increase in the number of symbols (e.g., up to 2^{32} symbols in 32-bit processing), which in turn, causes a very high state/transition overhead. We leave exploration for large bitwidths to future work.

Based on these observations and different properties of automata in each bitwidth, we perform a sensitivity analysis to identify the best bitwidth-size processing on spatial automata accelerators. To do so, we change the subarray sizes in CA to support 1-bit, 2-bit, and 4-bit automata processing (we call them Reduced Bitwidth Designs or RBDs). We then present and evaluate a reconfigurable FPGA solution for different bitwidth processing (2, 4, 8, and 16-bit).

6.1 Reduced Bitwidth Design

We explore different bitwidth sizes in CA architecture, which is an in-memory automata processing accelerator. To do so,

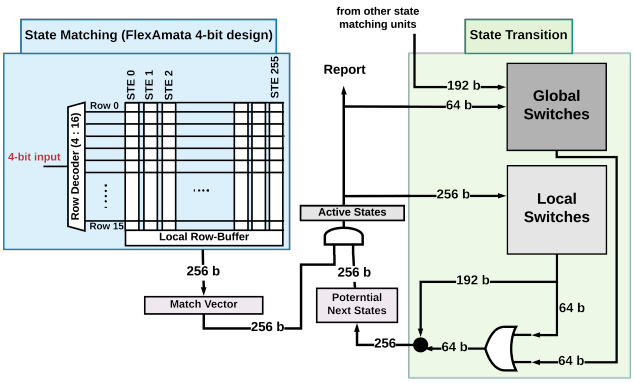

Figure 8. A 4-bit automata processing unit (4-bit CA-RDB).

we change the memory subarray heights in CA design to match the symbol processing rate. We refer to these modified designs as CA-RDBs (reduced bitwidth designs).

Figure 8 represents the 2-stage pipeline architecture of a 4-bit automata processing unit (4-bit CA-RDB), which can process an automaton with up to 256 states and arbitrary connectivity pattern. In the state matching phase, the 4-bit input is decoded as the input of the SRAM-based memory subarray. The states whose symbols match the input is read to the row-buffer and stored in the match vector. In the state transition stage, the potential next states (the states that are connected to the currently active states), are discovered through the local switches. Finally, bitwise AND operation of the potential next states and match vector recognizes the states that (1) are matched with the current input symbol and (2) their parents were active states in the previous cycle.

To support an automaton of a larger size, which is especially needed when processing a different bitwidth automaton, we utilize a hierarchical interconnect to connect local switches through a global interconnect [32, 34]. Both local and global interconnects are full-crossbar and support full connectivity in an automaton, meaning there can be an edge between every two states. A switch in the crossbar is modeled with an 8T SRAM memory cell following prior works [32, 34, 38].

The 1-bit and 2-bit designs are similar to Figure 8, but with state matching subarrays of size 2×256 and 4×256, respectively. Compared to the 8-bit design (original CA), the subarray size decreases 128×, 64×, and 16× for 1, 2, and 4-bit designs, respectively. Moreover, the memory decoder size, the access latency, and energy consumption for accessing state matching subarrays of 1, 2, and 4-bit designs decrease accordingly.

Generally speaking, as technology shrinks, SRAM arrays are moving from tall to wide structures with fewer rows [15, 17, 55]. This provides a better SRAM energy efficiency at a lower supply voltage. Recently, researchers have started to explore shorter SRAM subarrays to design accelerators in

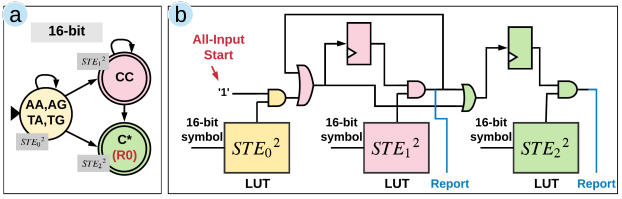

Figure 9. Mapping an automaton to the FPGA resources.

state-of-the-art applications, such as deep neural networks. For example, Lie et al. [26] propose an in-SRAM computation for binary neural networks [29]. Interestingly, they conclude that shorter SRAM subarrays (i.e., shorter memory columns) provide a better classification accuracy due to a smaller quantization error when calculating the partial sum in convolution operation. These support the applicability of CA-RDB designs, which rely on short memory subarrays.

Moreover, memory technologies such as reduced latency DRAMs (RLDRAM) [28] have smaller column sizes in each subarray to achieve a higher memory access rate. The efficiency of *4-bit* automata processing architecture introduces the potential of alternative use of memory technologies (e.g., RLDRMAs) for automata processing.

6.2 FPGA

To study the effect of different bitwidths on FPGAs, we equipped Flexamata with an HDL generator targeting Xilinx FPGAs, and implemented a two-stage pipeline automata processor similar to REAPR [50] but with symbol bit-width length as a parameter. FlexAmata transforms the automata to the target bitwidth, and then generates the HDL code for FPGA backends. This section discusses the automata processing engine on the FPGA to highlight the insights of processing variable bitwidths on this spatial platform.

In Figure 9 (a), a homogeneous automaton is shown that processes two 8-bit symbols (16-bit) per cycle. States have been color-coded to represent their equivalent units in the circuit shown in Figure 9 (b). Symbol matching is done entirely in LUTs, based on the 16-bit symbols. Theoretically, flip-flops (FFs) are equivalent to potential next-state registers in Figure 8, and they represent that a state may be active in the next cycle.

The input signals of the FFs come from an OR gate, which is the OR signal of all the states that have incoming transitions to that specific state. This is compatible with our previous definition, where once a state is activated, all of its children are considered as potential active states. The states that have common parents can share their FFs and save resources. However, in theory, their corresponding states cannot be merged since they are not equivalent states. Just as in Figure 8, the report signals of the final states are generated from the AND gate of matching signals and potential active states.

We observed that small bitwidths (< 8) do not utilize FPGA LUT resources well. This is mainly because LUTs in Xilinx FPGAs can implement up to two functions with five inputs or one function with six inputs, and thus, 1, 2, and 4-bit designs operate inefficiently on LUTs. On the other hand, processing more symbols per cycle leads to a more complex matching with many intervals from different states combined to a single state. This situation makes matching using 6-inputs LUTs inefficient in terms of resource usage and clock frequency (longer critical path), as LUTs need to be combined to implement bigger functions. We observed that the middle-sized bitwidths (e.g., 16-bit) could efficiently utilize the resources and achieve higher performance (explained in Section 8.3).

7 Evaluation Methodology

NFA workloads: We evaluate our proposed claims using ANMLZoo [43], AutomataZoo [44], and Regex [7] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. We present a summary of the applications in Table 2, including the number of states and transitions in each benchmark, as well as the average node degree for each state and symbol density. Symbol density metric represents the average number of symbols per state, and it is calculated by dividing the total number of symbols over the total number of states in each benchmark. We later show that the higher symbol density causes the higher state and transition overhead when transforming an automaton to a higher number of bits per symbol.

Table 2. Benchmark Overview

Benchmark	#Family	#States	#Transitions	Ave. Node Degree	Symbol Density
Brill [43]	Regex	42658	62054	2.90	52.2
Bro217 [7]	Regex	2312	2130	1.84	1.8
Dotstar03 [7]	Regex	12144	12264	2.01	3.1
Dotstar06 [7]	Regex	12640	12939	2.04	4.8
Dotstar09 [7]	Regex	12431	12907	2.07	6.7
ExactMath [7]	Regex	12439	12144	1.95	1
PowerEN [43]	Regex	40513	40271	1.98	5.8
Protomata [43]	Regex	42009	41635	1.98	116
Ranges05 [7]	Regex	12621	12472	1.97	1.2
Ranges1 [7]	Regex	12464	12406	1.99	1.2
Snort [43]	Regex	66466	78315	2.35	13.1
TCP [7]	Regex	19704	21164	2.14	10.1
Hamming [43]	Mesh	11346	19251	3.39	113
Levenshtein [43]	Mesh	2784	9096	6.53	1
EntityResolution [43]	Widget	95136	219264	4.60	47
Fermi [43]	Widget	40783	57576	2.82	7.1
RandomForest [43]	Widget	33220	33220	2	179
SPM [43]	Widget	69029	211050	6.11	26.5
BlockRings [43]	Synthetic	44352	44352	2	1
CoreRings [43]	Synthetic	48002	48002	2	1

Experimental setup: To calculate area, power, and clock cycle for CA-RBDs and CA, we use CACTI 7.0. We assumed a 4MB SRAM-based memory with eight banks on 22nm technology and operating temperature 360K. All FPGA results are obtained on a Xilinx Virtex UltraScale+ XCVU9P with

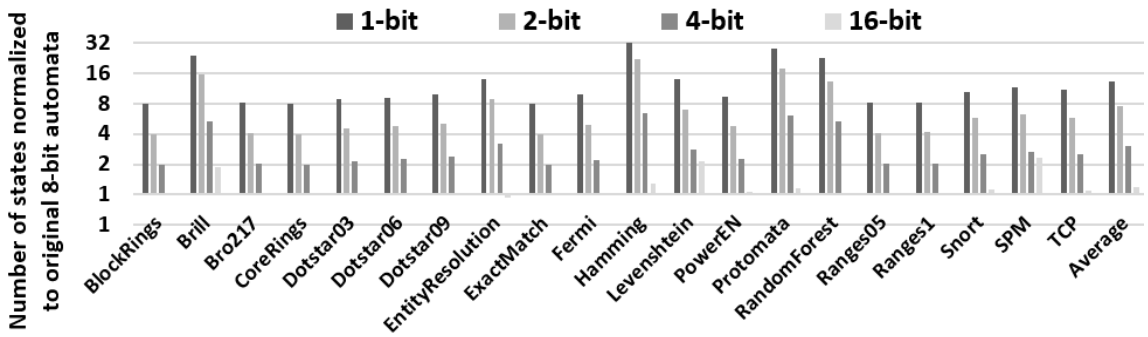

Figure 10. State overhead in different bitwidths normalized to the original 8-bit automata.

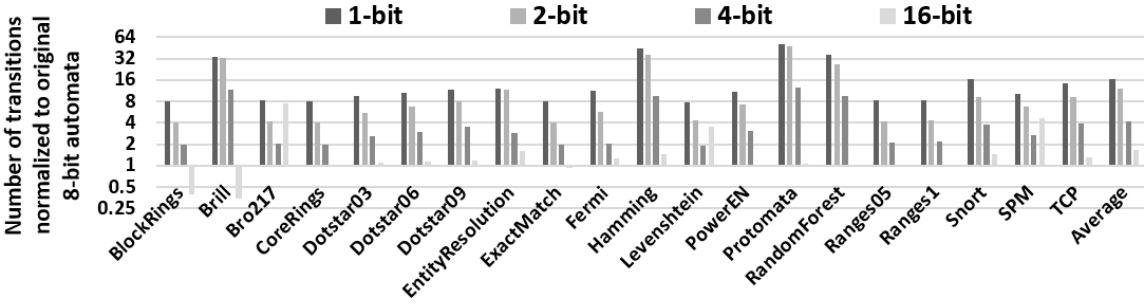

Figure 11. Transition overhead (#edges) in different bitwidths normalized to the original 8-bit automata.

a PCIe Gen3 x16 interface, 75.9 Mb BRAM and 1182k CLB LUTs in 16nm technology. The FPGA's host computer has an eight cores Intel i7-7820X CPU running at 3.6 GHz and 128 GB memory. Designs are synthesized with the Xilinx Vivado v2018.3.

Because the AP, CA-RDBs, CA, and our FPGA solution have similar run-time execution models and all are PCI-Express boards, we can disregard data transfer and control overheads to make general capacity and performance comparisons between these platforms.

Comparison metric: To compare in-memory automata processing architectures (the AP, CA, and RBDs), we use throughput per unit area. Throughput is defined as the number of bits that can be processed in one second ($frequency \times Bitwidth\_size$). We then calculate the throughput per area (the total area used for a benchmark) to consider the effect of consumed spatial resources used in each architecture. If the automata in a benchmark cannot fit in one hardware unit (HU), we replicate HUs until all the automata are accommodated. The total area is calculated by multiplying the area of one HU and the number of required HUs for each benchmark.

ANMLZoo benchmarks are designed to fit into an AP chip (with up to 48K states). However, because of the AP inefficient routing, the ANMLZoo benchmarks cannot utilize all 48K states, and thus, the benchmarks are shrunk and do not indicate a real-size application. We replicate each benchmark 1000 times to create a larger set of automata for each benchmark. This makes sure that all the benchmarks require at least one unit of hardware in the architectures we study.

8 Results

In this section, first, we analyze the state/transition overhead in various bitwidths on the automata benchmarks. We then leverage these analyses to evaluate the performance of CA-RDBs and compare them with the AP, CA, and FPGA solutions.

8.1 Complexity Analysis of Different Bitwidths

This section discusses the state and transition overhead in different bitwidths. We use FlexAmata to generate n-bit automata (n=1-16) for the benchmarks. Figures 10 and 11 show the number of states and transitions in each bitwidth, normalized to the number of states and transition in the original 8-bit design. Due to space constraints, we only represent the ones with the lowest overhead (i.e., 1, 2, 4, and 16-bit designs).

We observed that benchmarks with higher symbol density (last column in Table 2), such as Brill, EntityResolution, Hamming, Protomata, and RandomForest have higher state and transition overhead in different bitwidths. Interestingly, the number of states in EntityResolution in 16-bit design is less than the original 8-bit design. This is because, when the original design is converted to 1-bit automata, FlexAmata applies bit-level minimization to merge states. Therefore, the

Table 3. Comparison among 1, 2, 4, and 8-bit RBDs and the AP. RBDs are all based on 4MB SRAM-based memory.

Architecture	Symbol size (bit)	Subarray Size	Number of subarrays	Number of States	State matching delay (ns)	Interconnect delay* (ns)	Frequency (GHz)	Area (mm^2)	Total Dynamic R/W energy (nJ) per access	Total leakage power (mW)**
RBD (22nm)	1	R=2, C=256	65,536	16,384K	0.11	2.7	0.368	13.14	0.53	1895
	2	R=4, C=256	32,768	8,192K	0.11	0.368	2.7	10.29	0.45	1969.24
	4	R=16, C=256	8,192	2048K	0.15	0.368	2.7	7.76	0.39	1664
	8	R=256, C=256	512	128K	0.23	0.368	2.7	5.06	0.35	1396
AP (50nm)	8	R=256, C=256	192	48K	7.5	7.5	0.133	144	N/A	N/A

* In CA-RDBs, critical path is state transition stage (interconnect), which is similar for all CA-RDBs. Therefore, they all have a similar clock frequency.

** The details of energy and power are not available for the AP. The estimated TPD is 4W maximum.

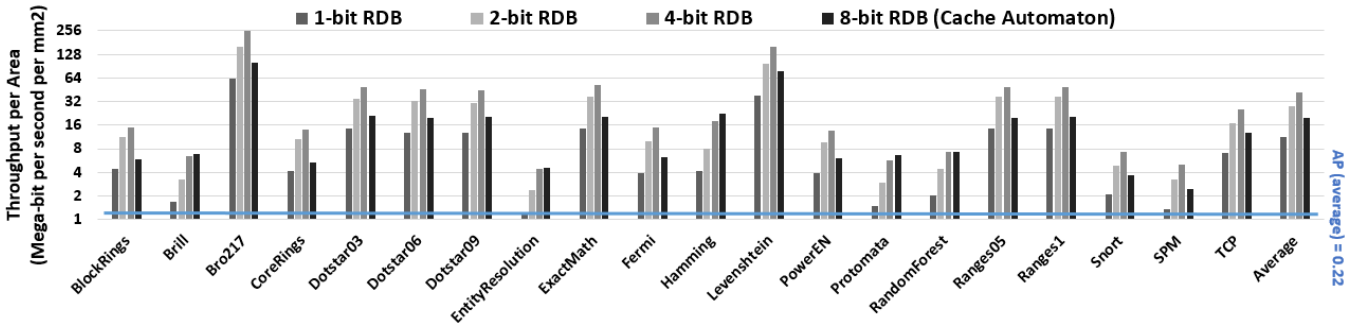

Figure 12. Comparing throughput-per-area (**mega-bit** processing per second per $1mm^2$ area) in RBDs with the AP and CA. RDBs and CA are in 22nm, so we scale the AP to 16nm. On average, 4-bit processing has 2.2× and more than 100× higher throughput-per-unit-area than CA and the AP, respectively.

16-bit designs generated from the optimized bit-automata have fewer states compared to the original 8-bit design. This shows that FlexAmata is useful to minimize an automaton when its original design is not optimized. Moreover, the applications with higher node degree, such as Levenshtein, have higher state/transition overhead.

On average, 1, 2, 4, and 16-bit designs have 13.2×, 7.5×, 3×, and 1.2× more states and 16.2×, 12.1×, 4.2×, and 1.6× more transitions over the original 8-bit designs. The increase in the number of states translates to utilizing more memory-column resources in in-memory designs and LUTs in FPGAs. The increase in the number of transitions translates to utilizing more interconnect resources in FPGAs. However, transitions in RBDs are implemented with a memory-based full crossbar interconnect (Figure 8), which supports full connectivity. This means that the higher transition count in smaller bitwidths utilizes the existing hardware switches in full crossbar and does not incur extra resource overhead.

Using the analysis presented in this section, we calculate hardware-related parameters to identify the best bitwidth-size in spatial accelerators.

8.2 Reduced Bitwidth Designs

This section evaluates reduced bitwidth designs and compares them with the AP and CA models. Table 3 presents the architectural parameters for different CA-RDBs (1, 2, 4, and 8-bit) and the AP. The 8-bit design represents CA model. Generally, in an in-memory automata design, the number of states shows the number of required memory columns, and $2^{biwidth-size}$ shows the number of required memory rows.

All CA-RDBs are designed assuming a 4MB SRAM-based memory. The smaller bitwidth designs have smaller subarrays, and thus, they have a higher state density and smaller read and write access time. To calculate clock frequency, we found that the critical path is the state transition stage, where local and global switch arrays are calculating the potential next states in parallel (see Figure 8). The global switch stage requires 0.368ns composed of 0.125ns due to wire-delay (SPICE modeling) and 0.243ns due to global switch. The distance between SRAM arrays and global switch arrays is estimated to be smaller than 1.9mm assuming maximum state matching dimension of $3.5mm \times 3.75mm$ (for 1-bit design). The pipeline clock frequency is determined by the slowest stage. Thus, the maximum possible frequency is 2.7GHz. We choose to operate at 2.5GHz. The area, total read/write access, and total leakage power of a smaller bitwidth design are higher. This is because more subarrays incur more sense-amplifier and higher wiring overhead.

Figure 12 compares throughput per unit area in CA-RDBs with the original CA design (in 8-bit) and the AP across several benchmarks. The applications with more states, such as Entity Resolution, Snort, and SPM require more hardware resources, and therefore, have lower throughput-per-area. Within each application, the 1-bit design has the lowest throughput-per-area. This is because the state and transition overhead in 1-bit design will not be amortized by the higher

Table 4. Comparing FPGA performance results for different bitwidths and a modified version of REAPR (8-bit) [13].

Benchmark	Number of LUTs					Number of FFs					Frequency (MHz)				
	2-bit	4-bit	8-bit	16-bit	REAPR	2-bit	4-bit	8-bit	16-bit	REAPR	2-bit	4-bit	8-bit	16-bit	REAPR
Brill	118,220	65,589	39,102	87,191	27,621	147,768	72,044	32,441	44,772	27,782	93	141	165	214	166
PowerEN	130,193	88,252	49,526	53,711	35,359	192,418	89,281	38,398	23,427	31,530	153	174	286	279	163
Protomata	127,237	73,745	47,092	46,706	49,791	194,629	90,628	34,491	19,866	36,285	116	196	167	263	126
Snort	75,754	43,829	22,601	31,610	43,061	345,239	148,443	58079	44,456	28,047	97	117	89	162	98
Hamming	31,170	13,876	7,380	9,302	5,602	139,885	17,884	6,702	3,450	6,637	62	118	187	210	312
Levenshtein	14,286	4,209	2,278	9,877	2,538	17,921	6,090	2,346	3,128	2,242	609	514	719	406	434
Entity Resolution	423,515	178,125	65,020	244,925	50,349	412,980	165,450	53,605	61,890	47,102	85	82	175	97	212
Fermi	113,460	44,729	27,804	38,743	36,314	165,495	71,682	29,555	20,127	32,261	183	376	393	225	116
Random Forest	215,066	89,544	41,907	27,971	50,349	321,262	118,944	30,961	15,571	25,769	80	135	205	233	200
SPM	254,038	14,441	87,435	219,014	64,615	381,749	173,161	57,008	84,244	59,106	185	136	264	168	126
BlockRings	110,782	88,507	41,201	22,496	44,446	177,675	88,875	44,367	22,368	44,185	126	53	123	86	256
Average	146,702	75,893	39,213	71,959	37,276	227,002	94,771	35,268	31,209	30,995	163	186	252	213	201

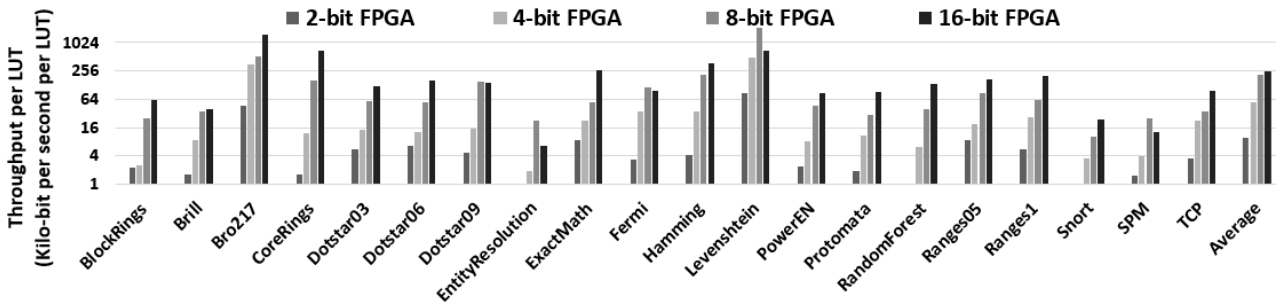

Figure 13. Comparing throughput-per-LUT (**kilo-bit** processing per second per LUT) in FPGA kernels. 16-bit processing has up to 4.9× higher throughput-per-LUT than 8-bit designs.

state density of 1-bit architecture. On average, 2-bit and 4-bit designs have 1.4× and 2.2× higher throughput-per-area than original 8-bit design, respectively. This means that to reach the same throughput, an 8-bit design requires 1.4× and 2.2× more hardware units on average than 2-bit and 4-bit designs.

This is mainly because state density in the 2-bit and 4-bit designs is exponentially (64× and 16×, respectively) higher than 8-bit design (column 5 in table 3). This means that more automata can be configured in a similar amount of area, which reduces the total hardware resource requirements. The higher state density can pay for 7.5× and 3× higher state count and larger total area (column 9 in table 3) in the 2-bit and 4-bit designs compared to the 8-bit design. On average, 2-bit and 4-bit designs have 126× and 192× higher throughput per area than the AP.

Using FlexAmata toolchain, we find that the number of states in a connected component does not exceed 1024, which is in compliant with our interconnect model (Figure 8). Moreover, our investigations show that the interconnect of larger connected components can entirely fit into a four 256×256 crossbar switch designs, with allowing up to 64 connections between each with a global switch.

In summary, we conclude that the 4-bit processing design for in-memory automata accelerators results in higher overall performance than the existing 8-bit solutions. Therefore, this paper suggests the 4-bit processing architectures for the next-generation in-memory automata accelerators.

8.3 FPGA Results

Performance results for FPGA-based implementations are presented in Table 4. A modified REAPR is presented in [13] on thirteen benchmarks from ANMLZoo. We compared our results to this implementation from the published results and, thus, limit Table 4 to these thirteen benchmarks. Compared to REAPR (which is an 8-bit design), our 8-bit design has a higher frequency and higher LUTs/FFs usage. Our design uses a staging technique to localize signals and avoid high fan-out wires (e.g., input signal). This reduces the critical path, which increases frequency by 25% at the expense of 4% more LUTs and 12% more FFs.

On average, 2-bit and 4-bit processing require 3.7× and 1.9× more LUTs and 6.4× and 2.7× more FFs compared to the original 8-bit design, respectively. This is mainly due to the higher state and edge overhead in the smaller bitwidths (see Figure 10 and 11). LUTs can have one input and up to two outputs, and, therefore, they cannot accommodate more states in smaller bitwidth designs. They also have 26% and 35% lower frequency and, thus, lower throughput compared

Table 5. Comparison across architectures. Throughput is averaged over 20 benchmarks in Table 2 for the FPGA solution.

Architecture	Technology Size (nm)	Throughput (Gbps)
CA-RDB (4-bit)	22 (original)	10
AP (8-bit)	50 (original)	1
FPGA (16-bit)	16 (original)	3.7
CA-RDB (4-bit)	16 (projected)	13.75
AP (8-bit)	16 (projected)	1.37

to the original 8-bit design. All these confirm that small-bitwidth processing is not suitable for FPGAs.

Compared to 8-bit, a 16-bit design has 1.8× more LUTs, 11% fewer FFs, and 15% lower frequency. However, the input processing rate of the 16-bit design is 2× higher than the 8-bit design. This implies that, for the applications with real-time processing needs, a 16-bit design with 2× higher throughput can be used. In larger bitwidths (e.g., 16-bit), the number of symbols increases, and thus, more LUTs are required. However, in 16-bit design, there are more states with common parents than 8-bit, which can share the FFs. Therefore, more LUTs are used than FFs in larger bitwidths. In smaller bitwidths, more FFs are used than LUTs. This is because there is a higher chance that two states share one LUT when having 2-bit symbols.

Figure 13 compares throughput/LUT for different bitwidths in our FPGA solution. As expected, 2-bit and 4-bit designs have lower throughput-per-LUT than 8-bit design. Benchmarks with higher average node degree, such as Levenshtein and SPM, require relatively more LUTs in 16-bit design than 8-bit design (see Table 2 and Table 4). This decreases throughput-per-LUT in these applications in 16-bit design. On average, the 16-bit design has up 4.9× higher throughput-per-LUT than 8-bit design. Overall, for regular expressions with relatively lower average node degree than mesh and widgets, 16-bit designs perform best on FPGAs.

8.4 Comparison Across Architectures

This section compares the best designs across spatial architectures, i.e., the AP (8-bit), FPGA (16-bit), and an in-memory CA-RDB (4-bit) solutions, in terms of throughput. On average, 4-bit CA-RDB has 3.7× and 10× higher throughput than 16-bit FPGA and the AP solutions, respectively, on the same technology node (16nm).

The 4-bit CA-RDB efficiency is derived from (1) the reduced state-matching subarrays (16× smaller than the 8-bit design) and (2) an efficient and flexible routing architecture, which is a memory-based full-crossbar interconnect that can connect any two states. This results in higher automata density because the state-matching resources are not underutilized due to routing congestion. The automata with more complex routing structures incur routing congestion on the AP and FPGA (FPGAs can handle more complex routing better than the AP), and thus, incur higher area overhead than CA-RDB to accommodate all the automata in a benchmark.

Moreover, our place-and-route algorithm on CA-RDBs is 1-2 orders of magnitude faster than the AP compiler, and the AP compiler is 1-2 orders of magnitude faster than the FPGA tools. With a large application and a limited number of hardware units, the application might need several rounds of reconfigurations on the hardware. This implies that the AP or FPGA will incur a significant performance penalty when an application does not fit on the available hardware resources.

9 Conclusions and Future Work

This paper presents FlexAmata, a compiler solution that reshapes an automaton to process an arbitrary-size input alphabet. This can be done as an offline pre-processing stage. The main two benefits of FlexAmata are: (1) to provide a universal and efficient mapping of any applications with arbitrary alphabet size to any spatial automata hardware accelerators with a built-in fixed-size input processing rate, and (2) to rethink the commonly used 8-bit processing designs in the existing spatial automata accelerators and explore other alternative processing rates that results in higher throughput per unit area. Using real-world automata case studies, we show that FlexAmata provides higher throughput and state-density for applications with small alphabet-size and makes automata acceleration feasible for applications with a very large alphabet size. We then perform a sensitivity analysis for different processing rates on the Cache Automaton architecture (which is the best performing prior work) and our FPGA-based automata processing kernel using a wide range of automata applications, and we discover that the Cache Automaton architecture with a 4-bit processing rate design results in a 2.2× higher throughput-per-area than the default 8-bit processing rate architecture. Moreover, our 16-bit processing design on the FPGA LUT-based implementation results in up to 4.9 higher throughput-per-LUT than the default 8-bit processing engine. These provide insights to design the future automata accelerators more efficiently. Future work will explore striding techniques (or multi-symbol processing) using our architectural findings.

Acknowledgments

We thank the anonymous reviewers whose comments helped improve and clarify this manuscript. This work is funded, in part, by the NSF (CCF-1629450) and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA.

References

[1] Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner. 2013. Hardware-accelerated regular expression matching with overlap handling on IBM PowerEN processor. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 1254–1265.

[2] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. 2016. Scalable algorithms for NFA multi-striding and NFA-based deep packet inspection on GPUs. *IEEE/ACM Transactions on Networking* 24, 3 (2016), 1704–1717.

[3] Michela Becchi and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*. ACM.

[4] Michela Becchi and Patrick Crowley. 2007. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, 145–154.

[5] Michela Becchi and Patrick Crowley. 2008. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 50–59.

[6] Michela Becchi and Patrick Crowley. 2013. A-DFA: A time-and space-efficient DFA compression algorithm for fast regular expression evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)* (2013).

[7] Michela Becchi, Mark Franklin, and Patrick Crowley. 2008. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE.

[8] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 using Automata Processing across Different Platforms. In *24th International Symposium on High-Performance Computer Architecture*. IEEE.

[9] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. 2016. Entity resolution acceleration using the automata processor. In *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 311–318.

[10] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters* 16, 1 (2016), 46–50.

[11] Benjamin C Brodie, David E Taylor, and Ron K Cytron. 2006. A scalable architecture for high-throughput regular-expression pattern matching. In *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 191–202.

[12] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26.

[13] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. 2019. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[14] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on* 25, 12 (2014).

[15] Achiranshu Garg and Tony Tae-Hyoung Kim. 2013. SRAM array structures for energy efficiency enhancement. *IEEE Transactions on Circuits and Systems II: Express Briefs* 60, 6 (2013), 351–355.

[16] Victor Mikhaylovich Glushkov. 1961. The abstract theory of automata. *Russian Mathematical Surveys* (1961).

[17] Fatih Hamzaoglu, Yibin Ye, Ali Keshavarzi, Kevin Zhang, Siva Narendra, Shekhar Borkar, Mircea Stan, and Vivek De. 2000. Dual-V/sub T/SRAM cells with full-swing single-ended bit line sensing for high-performance on-chip cache in 0.13/spl mu/m technology generation. In *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No. 00TH8514)*. IEEE, 15–19.

[18] Tran Trung Hieu and Ngoc Thinh Tran. 2013. A memory efficient FPGA-based pattern matching engine for stateful NIDS. In *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*. IEEE, 252–257.

[19] Intel. [n.d.]. https://github.com/01org/hyperscan.

[20] Rasha Karakchi, Lothrop O Richards, and Jason D Bakos. 2017. A Dynamically Reconfigurable Automata Processor Overlay. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE, 1–8.

[21] Vlastimil Košar and Jan Korenek. 2014. Multi-stride NFA-split architecture for regular expression matching using FPGA. In *Proceedings of the 9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*.

[22] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R. Stan, and Kevin Skadron. 2020. Fulcrum: a Simplified Control and Access Mechanism toward Flexible and Practical in-situ Accelerators. *The 26th IEEE International Symposium on High-Performance Computer Architecture* (2020).

[23] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, M Arif Rahman, and Mircea R. Stan. 2019. An Overflow-free Quantized Memory Hierarchy in General-purpose Processors. *IEEE International Symposium on Workload Characterization* (2019).

[24] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. 2016. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 173.

[25] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. 2018. Architectural Support for Efficient Large-Scale Automata Processing. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 908–920.

[26] Rui Liu, Xiaochen Peng, Xiaoyu Sun, Win-San Khwa, Xin Si, Jia-Jing Chen, Jia-Fang Li, Meng-Fan Chang, and Shimeng Yu. 2018. Parallelizing SRAM arrays with customized bit-cell for binary neural networks. In *Proceedings of the 55th Annual Design Automation Conference*. ACM, 21.

[27] Tingwen Liu, Yifu Yang, Yanbing Liu, Yong Sun, and Li Guo. 2011. An efficient regular expressions compression algorithm from a new perspective. In *2011 Proceedings IEEE INFOCOM*. IEEE, 2129–2137.

[28] Micron. 2019. RLDRAM Memory. https://www.micron.com/products/dram/\rldram-memory.

[29] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[30] Indranil Roy and Srinivas Aluru. 2016. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM transactions on computational biology and bioinformatics* 13, 1 (2016), 99–111.

[31] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. 2018. A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 665–674.

[32] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching. *The 26th IEEE International Symposium on High-Performance Computer Architecture* (2020).

[33] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient In-Memory Accelerator for Automata Processing. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019), 87–99.

[34] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. Scalable and Efficient in-Memory Interconnect Architecture for Automata Processing. *IEEE Computer Architecture Letters* (2019).

[35] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. 2017. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing (ICS)*. ACM.

[36] Fabio Somenzi and Roderick Bloem. 2000. Efficient Büchi automata from LTL formulae. In *International Conference on Computer Aided Verification*. Springer, 248–263.

[37] Arun Subramaniyan and Reetuparna Das. 2017. Parallel automata processor. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 600–612.

[38] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *50th Annual IEEE/ACM International Symposium on Microarchitecture*.

[39] Cong Tian and Zhenhua Duan. 2007. Model checking propositional projection temporal logic based on SPIN. In *International Conference on Formal Engineering Methods*. Springer, 246–265.

[40] II Tommy Tracy, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabriel Robins. [n.d.]. Nondeterministic finite automata in hardware-the case of the Levenshtein automaton. ([n. d.]).

[41] Moshe Y Vardi and Pierre Wolper. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society, 322–331.

[42] Lucas Vespa, Ning Weng, and Ramaswamy Ramaswamy. 2010. Ms-dfa: Multiple-stride pattern matching for scalable deep packet inspection. *Comput. J.* 54, 2 (2010), 285–303.

[43] Jack Wadden et al. 2016. ANMLZoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *IISWC*. IEEE.

[44] Jack Wadden et al. 2018. AutomataZoo: A Modern Automata Processing Benchmark Suite. In *IISWC*. IEEE.

[45] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An overview of micron's automata processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*. IEEE, 1–3.

[46] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea R Stan, and Kevin Skadron. 2015. Association rule mining with the Micron Automata Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 689–699.

[47] Ke Wang, Elaheh Sadredini, and Kevin Skadron. [n.d.]. Hierarchical Pattern Mining with the Micron Automata Processor. In *International Journal of Parallel Programming (IJPP)*. 2017.

[48] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *International Conference on Computing Frontiers*. ACM.

[49] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 631–648.

[50] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *International Conference on Field Programmable Logic and App.* IEEE.

[51] Xilinx. 2019. UltraScale Architecture Memory Resources. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf.

[52] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. 2008. High-speed regular expression matching engine using multi-character NFA. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 131–136.

[53] Yi-Hua Yang and Viktor Prasanna. 2012. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Trans. Comput.* 61, 7 (2012).

[54] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. 2006. Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*. IEEE, 93–102.

[55] Kevin Zhang, Uddalak Bhattacharya, Zhanping Chen, Fatih Hamzaoglu, Daniel Murray, Narendra Vallepalli, Yih Wang, B Zheng, and Mark Bohr. 2005. SRAM design on 65-nm CMOS technology with dynamic sleep transistor for leakage reduction. *IEEE Journal of Solid-State Circuits* 40, 4 (2005), 895–901.

[56] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. 2015. Brill tagging on the micron automata processor. *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)* (2015), 236–239.

[57] Shufang Zhu, Geguang Pu, and Moshe Y Vardi. 2019. First-Order vs. Second-Order Encodings for LTLf-to-Automata Translation. *arXiv preprint arXiv:1901.06108* (2019).

A Artifact Appendix

A.1 Abstract

Our artifact contains all the source code for FlexAmata. FlexAmata is an automata transformation tool, which transforms an n-bit processing automaton to an m-bit processing automaton. This transformation provides compatibility between automata application alphabet-size and target hardware accelerator constraints. FlexAmata is a part of a larger simulator, APSim (Automata Processing Simulator), developed by the authors of this paper. We provide the source code for APSim, the benchmarks we used, and scripts to regenerate Table 2, Table 4, Figure 10, and Figure 11.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Finite automata bitwidth transformation and minimization.
- **Program:** python 2.7, Xilinx Vivado, swig
- **Compilation:** g++
- **Transformations:** Finite automata transformations
- **Data set:** ANMLZoo benchmark, which is publicly available.
- **Run-time environment:** Ubuntu > 12.04 with necessary python packages installed
- **Hardware:** x86/64 CPU for the transformation part. Partial hardware evaluation was done on Xilinx Virtex UltraScale+ XCVU9P.
- **Execution:** python scripts
- **Metrics:** Number of states and transitions in an automaton, and hardware frequency and resource usage in FPGA
- **Output:** transformation statistics and auto generated HDL code to be synthesized for FPGA.
- **Experiments:** Three main experiment scripts and one demo are provided in the "Examples" folder of the repository
- **How much disk space required (approximately)?:** less than 1GB for the automaton transformation part. For FPGA, the generated code can grow up significantly based on the dataset size (e.g., 50GB).
- **How much time is needed to prepare workflow (approximately)?:** 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** A few hours for a small portion of the dataset. It increases significantly for full dataset. In the scripts, you can reduce the number of automata (or connected components) to decrease the processing time.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Yes.

https://doi.org/10.5281/zenodo.3612777

A.3 Description

A.3.1 How delivered. The source code, benchmarks, and scripts are available on Github:
https://github.com/gr-rahimi/APSim/tree/ASPLOS_AE

A.3.2 Hardware dependencies. Xilinx Virtex UltraScale+ XCVU9P

A.3.3 Software dependencies. Pyhton, swig, g++

A.3.4 Data sets. All datasets are either publicly available in ANMLZoo benchmark suite, or taken from [3]. For convenience, we have included all the benchmarks in a github repository (https://github.com/gr-rahimi/ANMLZoo.git) and the setup instructions are in the README file.

A.4 Installation

Installation procedures has been explained in the README of the repository.

A.5 Experiment workflow

Three main scripts are provided to replicate the experiments in the paper.

A.6 Evaluation and expected result

"Table2.py" script should replicate the results in Table-2 in the paper. "Table4.py" script should generate HDL codes to be synthesized by Xilinx Vivado. Frequency and look-up table usage should proportionally be compatible with the results in Table-4 based on the dataset size. For the interest of synthesis time, the first 200 connected component (CC) of benchmarks (if the benchmark has more than 200 CC) are considered. For these benchmarks, the final LUT/FF usages are scaled (with respect to the original CC count) to estimate the original benchmark resource usage. "Fig10-11.py" generates the number of states and transitions in different bitwidths (1-bit, 2-bit, 4-bit, 8-bit, and 16-bit) for the selected benchmark (the current script is set for ExactMatch). Figures 10 and 11 show the number of states and transitions in different bitwidths normalized to the number of states and transitions in the original 8-bit design. The script output shows these normalized numbers for the selected benchmark, which can evaluate Figure 10 and Figure 11.

A.7 Experiment Customization

Running experiments for the whole dataset is very time-consuming. It is possible to change the size of the dataset in the scripts (comments in the scripts can guide the user to change the dataset size).

A.8 Notes

To know more about APSim, send feedback, or file issues, please visit our github page:
https://github.com/gr-rahimi/APSim/tree/ASPLOS_AE