Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching

Elaheh Sadredini*, Reza Rahimi*, Marzieh Lenjani, Mircea Stan, and Kevin Skadron
University of Virginia, Charlottesville, VA
{elaheh, rahimi, ml2au, mircea, skadron}@virginia.edu

ABSTRACT

High-throughput and concurrent processing of thousands of patterns on each byte of an input stream is critical for many applications with real-time processing needs, such as network intrusion detection, spam filters, virus scanners, and many more. The demand for accelerated pattern matching has motivated several recent in-memory accelerator architectures for automata processing, which is an efficient computation model for pattern matching. Our key observations are: (1) all these architectures are based on 8-bit symbol processing (derived from ASCII), and our analysis on a large set of real-world automata benchmarks reveals that the 8-bit processing dramatically underutilizes hardware resources, and (2) multi-stride symbol processing, a major source of throughput growth, is not explored in the existing in-memory solutions.

This paper presents Impala, a multi-stride in-memory automata processing architecture by leveraging our observations. The key insight of our work is that transforming 8-bit processing to 4-bit processing exponentially reduces hardware resources for state-matching and improves resource utilization. This, in turn, brings the opportunity to have a denser design, and be able to utilize more memory columns to process multiple symbols per cycle with a linear increase in state-matching resources. Impala thus introduces three-fold area, throughput, and energy benefits at the expense of increased offline compilation time. Our empirical evaluations on a wide range of automata benchmarks reveal that Impala has on average 2.7× (up to 3.7×) higher throughput per unit area and 1.22× lower power consumption than Cache Automaton, which is the best performing prior work.

1. INTRODUCTION

The growing performance gap between processor and memory, also known as the memory wall [1], has been a major source of performance concern for many years; this problem is exacerbated in memory-bound applications, such as pattern-matching kernels in big-data domains, where many of patterns must be processed often with high-throughput or even real-time requirements. Pattern matching is an important task in many applications such as network security [2,3,4], bioinformatics [5,6], and data mining [7,8]. These patterns are often enormous in number and complex in static structure and dynamic behavior. This, combined with increasing network bandwidth and real-time stream processing requirements, makes pattern matching the performance bottleneck for these applications.

*Equal contribution

Regular expressions are a widely used pattern specification language, and they are efficiently implemented via Finite Automata (FA) [9]. The growing demand for high-performance pattern matching has motivated several efforts in designing software-based [10, 11, 12, 13, 14, 15, 16, 17] and FPGA-based [4,12,18,19,20] multi-stride pattern processing solutions that can process multiple-symbols per cycle. However, multi-symbol processing forces more pressure on memory bandwidth in CPU/GPU approaches and causes routing congestion for complex patterns in FPGA solutions. Moreover, FPGA solutions are mainly customized for network-based patterns, so current FPGA solutions may not perform well for other applications.

To address memory-wall challenges, recent studies explore in-memory architectures for automata processing, where they perform matching computation exactly where the data is located, and benefit from the massive internal memory bandwidth [21,22,23]. They all allow native execution of Non-deterministic Finite Automata (NFA) by providing a reconfigurable substrate to lay out the rules in hardware. This allows a large number of patterns to be executed in parallel, up to the hardware capacity. If the size of an application exceeds the hardware capacity, either several hardware units or several rounds of re-configurations are required. Many studies have shown that in-memory automata accelerators provide significant speedup over existing software solutions, FPGA implementation, and prior regex accelerators on a wide range of applications [6,7,8,22,24,25,26,27].

To process an automaton in memory-centric architecture models, each input requires two processing phases: state-matching, where the input symbol is decoded, and the states whose symbols match the input are detected through reading a row of memory; and state-transition, where successor states are activated by propagating signals through an interconnect.

In the existing in-memory automata accelerators, 50%-70% of hardware resources are spent for state-matching [22,23,28,29]. We study the state-matching resource utilization across a diverse set of automata benchmarks, and we found 86% of the time, only 3% of resources are utilized! This is mainly because in all these architectures, each state is modeled with a memory column of size 256, and 8-bit symbols are one-hot encoded in the memory columns to be able to accept a range of symbols (up to 256 symbols) in each state. However, the number of symbols accepted by a state is fewer than 8

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00017

symbols in 86% of the time. This, in turn, implies that the classic approach of one-hot encoding for matching drastically over-provisions state-matching resources, which incurs significant performance penalties and leads to an inefficient and costly design. Moreover, real-world automata benchmarks are often extensive in terms of state count, too big to fit in a single hardware unit, and in current memory-centric architectures, usually need multiple rounds of reconfiguration and re-processing of the data. Therefore, design density plays a vital role in overall performance.

To address this problem, we propose to reduce the memory column-height to 16, to take advantage of the common case that few symbols matche any particular state, and at the same time, convert (or *squash*) the automata to process 4-bit symbols instead of 8-bit per cycle without losing any accuracy. With our proposed optimizations, this transformation increases the number of states on average just 1.7×, and in turn, requires 1.7× more memory *columns* to encode the states. However, the total required memory *cells* decreases 9.4× (or $\frac{256}{16\times1.7}$) because the columns are so much shorter. This provides a higher total state capacity, which results in fewer rounds of reconfiguration, and improves the overall utilization and performance.

On the other hand, naively *squashing* 8-bit processing to 4-bit processing halves the throughput and limits the benefits of our solution. We fix this issue by proposing an in-SRAM multi-stride automata processing architecture that can process multiple 4-bit symbols per cycle. Our architecture, Impala, leverages the shorter memory subarrays and seeks a wider parallel solution using multiple combined memory columns to process multiple 4-bit symbols instead of a longer serial memory column that processes only one 8-bit symbol. Impala's compiler pre-processes an automaton and makes it compatible with our hardware design.

A transformed automaton (*squashed and strided*) has a higher number of transitions (edges) than its original automaton, and this makes the placement of transitions to interconnect resources more challenging. To address this issue, our compiler leverages the observation of Sadredini et al. [29] that the real-world automata have a diagonal-shaped connectivity pattern, which then can be efficiently compacted in a hierarchical memory-mapped interconnect architecture. We propose to use a genetic algorithm (GA) to intelligently place the transitions into a compact design to amortize the transition overhead.

In summary, this paper makes the following contributions:

- We propose Impala, an area-efficient, high-throughput, and energy-efficient in-SRAM architecture for automata processing. These three-fold efficiencies are obtained from (1) an architectural contribution that utilizes *shorter-and-parallel* SRAM-subarrays instead of *longer-and-serial* subarrays, and (2) an algorithmic contribution which efficiently transforms an automaton and maps it to Impala's resources. To the best of our knowledge, this is the first work that observes state-matching inefficiency in memory-centric accelerators and proposes an algorithm/architecture co-design for *multi-stride* automata processing for in-situ computations.
- To minimize the number of states in the transformed automata, we exploit Espresso [30], a CAD tool that efficiently reduces the complexity of digital circuits, and maps our state minimization problem to a logic minimization problem. Moreover, we propose an efficient placement algorithm by leverage a genetic algorithm.
- We perform thorough performance analysis on Impala and prior in-memory automata processing architectures. Our sensitivity analysis reveals that the 4-stride design on 4-bit nibbles (16-bit processing per cycle) provides the best overall throughput per area on Impala, which is up to 3.7× and 536× higher than Cache Automaton (CA) [22] and the Automata Processor (AP) [21], respectively. Moreover, Impala's 16-bit design provides 2.8× higher throughput than CA, 1.4× of which comes from our architectural contribution and 2× from the algorithmic benefit that re-shapes an automaton to process larger input size per cycle.
- We also compare Impala with FPGA multi-stride pattern matching solutions. We conclude that Impala provides ~10× higher frequency and ~20× higher throughput than the best performing solutions [4, 18] for 16-bit symbol processing rate. Moreover, Impala's 16-bit has 7.7× higher throughput than these FPGA solutions for a 64-bit processing rate.
- We present APSim, an open-source toolkit for cycle-accurate automata simulation, multi-symbol processing transformation, minimization, and performance modeling on our proposed architecture.

2. BACKGROUND

2.1 Non-Deterministic Finite Automata Primer

An NFA is represented by a 5-tuple, (Q,Σ,Δ,q_0,F), where Q is a finite set of states, Σ is a finite set of symbols, Δ is a transition function, q_0 are initial states, and F is a set of accepting states. The transition function determines the next states using the currently active states, and the input symbol just read. If an input symbol causes the automata to enter into an accept state, the current position of the input is reported.

We use the homogeneous automaton representation in our execution models (similar to ANML representation in [21]). In a homogeneous automaton, all transitions entering a state must happen on the same input symbol [31]. 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 [21], we call this element that both represents a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE). Figure 1 (a) shows an example of a classic NFA and its equivalent homogeneous representation. Both automata in this example accept the language $(A|C)^*(C|T)(G)^+$. The alphabets are $\{A,T,C,G\}$. In the classic representation, the start state is q_0, and accepting state is q_3. In the homogeneous one, we label each STE from STE_0 to STE_3, so starting states are STE_0, STE_1, and STE_2, and the accepting state is STE_3.

2.2 In-memory Automata Processing

To better understand Impala's architecture, this section presents a simplified two-level pipeline architecture for single-symbol processing of common in-situ automata accelerators, such as CA and the AP. In Figure 1 (b), memory columns are configured based on the homogeneous example in Figure 1 (a)

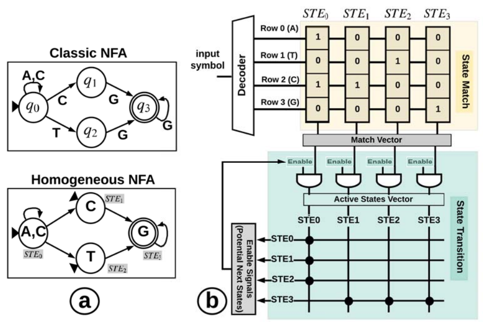


Figure 1: (a) Classic vs homogeneous NFA, (b) In-memory automata processing model.

for STE_0-STE_3. 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 is detected through reading a row of memory (*match vector*). Then, the set of potentially matching states is ANDed with the *active state vector*, which indicates the set of states that is currently active and allowed to initiate state transitions. 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 label (i.e., A, T, C, and G). Each state in the NFA 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 C (Figure 1 (a)), and the corresponding positions have '1', i.e., in the first and third rows (Figure 1 (b)). Assume STE_0 is a current active state. The potential next cycle active states (or enable signals) are the states connected to STE_0, which are STE_0, STE_1, and STE_2 (the enable signals for STE_0, STE_1, and STE_2 are '1'). Specifically, if the input symbol is 'C', then, Row2 is read into the *match vector*. Bitwise AND on the *match vector* and *potential next states* (enable signal) determines STE_0 and STE_1 as the current active states.

3. RELATED WORK

A number of multi-stride automata processing engines have been proposed on CPUs and GPUs [10, 11, 12, 13, 14]. 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 [32] and makes compression techniques [33] less efficient. Moreover, multi-symbol processing causes more pressure on memory bandwidth, because more states and transitions are required to be processed in each clock cycle.

Several FPGA solutions for single-stride [3, 34, 35, 36] and multi-stride [4, 12, 18, 20] automata processing have been proposed. Yang et al. [4] proposed a multi-symbol processing for regular expressions on FPGA and utilizes both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicates the resources in each stride. This increases the critical path when increasing the stride value. Yamagaki et al. [18] proposed a multi-symbol state transitions solution using a temporal transformation of NFAs to construct a new NFA with multi-symbol characters. This approach only utilizes LUTs and does not scale very well due to the limited number of lookup tables in FPGAs. In addition, in their multi-striding method, the benefit of improved throughput decreases in more complex regexes (with more characters or highly connected automata), mostly due to routing congestion. All the current multi-striding solutions are inspired by networking applications such as Network Intrusion Detection Systems (NIDS). However, patterns in other applications can have different structure and behavior, e.g., higher fan-outs, and this makes it difficult for NIDS-based FPGA solutions to map other automata to FPGA resources efficiently [36].

Accelerators designed specifically for regex processing have been proposed [37, 38, 39, 40] to accelerate pattern matching and automata processing. In general, while these solutions provide high line rates in principle, they are limited by the number of parallel matches, state transitions, and shape of the automata. None of these solutions consider multi-symbol processing, and Subramaniyan et al. [22] show that their in-SRAM solution, Cache Automaton (CA), has higher throughput and lower area consumption than these accelerators.

Unlike FPGA and regex accelerators that are optimized for pattern processing in network applications, several memory-centric automata processing accelerators have been recently proposed to improve the performance of general pattern matching. The Micron Automata Processor (AP) [21] and CA [22] propose in-memory hardware accelerators for 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 [41], and CA is at least an order of magnitude better than the AP [22].

To improve throughput, Subramaniyan et al. [42] propose a parallelization technique by replicating an automaton and splitting the input stream among them, and show speedup over the AP (such splitting is only needed when there are not enough regexes to leverage the capacity of the automata hardware). The speedup depends on the input stream, and the upper-bound speedup is equal to the number of hardware replications. High capacity in the automata accelerator is beneficial, and this approach is complementary.

Liu et al. [43] demonstrated that not all the states in an NFA are enabled during execution, and thus, do not need to be configured on the hardware. This reduces the hardware resources for an automaton on the in-memory automata accelerators, which in turn increases the performance when the application is very large and requires several rounds of re-configurations. However, the benefits of their approach is input-stream-dependent and cannot be directly compared

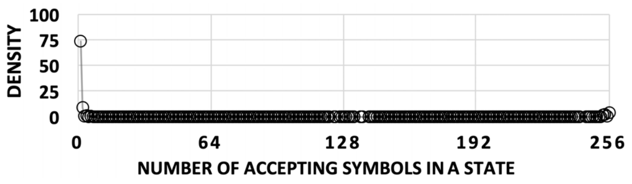


Figure 2: Normalized histogram of the states based on the number of accepting symbols. More than 86% of the states accept less than 8 symbols. More than 73% of the states only accept one symbol.

with prior solutions.

None of the prior in-memory solutions explores multi-symbol processing to increase throughput and overall performance. This work is the first to present a multi-symbol processing architecture co-designed with automata transformation to improve density.

4. ALGORITHMIC DESIGN

4.1 Motivation

Our key observation across a set of 20 real-world and synthetic automata benchmarks reveals that about 73% of the states only match against a single symbol (e.g., STE_1 in Figure 1 (a) that only accepts symbol C). This implies that only a single cell in a memory column of 256 cells is set to 1, and the rest are 0. Figure 2 demonstrates the frequency of the number of matching-symbols across all the states in these benchmarks normalized to the total number of states. This histogram is highly skewed to the left, where more than 86% of the states are matched against at most eight symbols. This means only 3% of the cells in memory columns are utilized!

While a memory column with 256 cells is powerful enough to implement any boolean function with eight inputs, existing automata computing architectures implement a relatively simple function (e.g., 73% of the time, states are comparing against a single symbol, which is a boolean function with a single product term). In other words, a simpler low-cost matching architecture targeting the dominant case of a small number of matching symbols is more efficient than the existing costly matching architecture targeting infrequent complex matching conditions. To better utilize the state-matching resources, we *squash* the 8-bit symbols to 4-bit, and re-shape the automaton accordingly for correct and lossless functionality. The corresponding hardware change is that state-matching columns now can be designed with shorter subarrays (16 memory cells instead of 256), which reduces the waste significantly.

Generally speaking, as technology shrinks, SRAM arrays are moving from tall to wide structures with fewer rows [44,45,46]. 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 state-of-the-art applications, such as deep neural networks. For example, Lie et al. [47] propose an in-SRAM computation for binary neural networks [48]. 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 Impala design, which relies on short memory subarrays.

A 4-bit automaton halves the processing rate. To increase the throughput, Impala proposes to process multiple 4-bit symbols/cycle versus one 4-bit symbol per cycle. The algorithmic aspects are discussed in this section, and the corresponding architectural solution is explained in Section 5.

4.2 Vectorized Temporal Squashing and Striding (V-TeSS)

Squashing: As discussed, the default 8-bit processing uses memory columns with 256 cells; however, the matching symbols of an STE are sparse, making this an underutilized matching resource. The squashing step converts the default 8-bit automaton to its equivalent 4-bit automaton. With empirical analysis, we realized that 4-bit conversion is the sweet spot and incurs minimal overhead compared to other squashing sizes (e.g., 2-bit or 3-bit processing - see [49] for more detail). Squashing to 4-bit increases the number of states 2.52× on average. But, the memory column size is exponentially decreased from 2^8 to 2^4.

Figure 3 (a) represents an 8-bit NFA in classical non-homogeneous representation. Symbols are represented in hexadecimal (e.g., $\backslash xBD$ represent an 8-bit symbol). Impala's compiler first reshapes the automaton to process 4-bit symbols (Figure 3(b)). For simplicity, we picked a simple automaton, and therefore, 8-bit to 4-bit conversion seems straightforward. However, in an automaton with loops and states with ranges of symbols (e.g., $[X-Y]$ notation in regex), we cannot simply break the states into two consecutive states with 4-bit symbols each. Impala's compiler first generates single-bit-automata by replacing each edge in the 8-bit original automaton with a sequence of states of length 8, and then traverses the bit-automaton with paths of length 4 and concatenates edges to generate 4-bit symbols. Due to space limitation, we do not explain the details here, as the algorithm to convert an 8-bit to 4-symbol automaton is not our main contribution.

Vectorized Temporal Striding: As expected, the 4-bit automata processing scheme halves the processing rate compared to the 8-bit automata. To increase the throughput (equals or more than 8-bit version), we again reshape the squashed 4-bit automaton to find its equivalent automaton that processes multiple 4-bit symbols per cycle. We call this transformation *Vectorized Temporal Striding*, as the state's matching rules are organized in vectors. Arranging matching symbols in vectors provides a nice property that is aligned with Impala hardware support. Temporal Striding [50, 51] and its vectorized version are transformations that repeatedly square the input alphabet of an input automaton and adjust its matching symbols and transition graph accordingly. The transformed automaton is functionally equal to the original automaton, but it processes multiple symbols per cycle, thus increasing throughput.

Figure 3 (c) shows the 4-stride (i.e., processing four 4-bit symbols per cycle) automaton in (b), and (d) converts it to its homogeneous representation. In the notation STE_x^y, x is state index and y is the stride size. We call the resulting automaton vectorized 4-stride because each 4-bit symbol in STE_0^4 represents one dimension in a four-dimensional vector, and each dimension will be mapped to a separate memory column

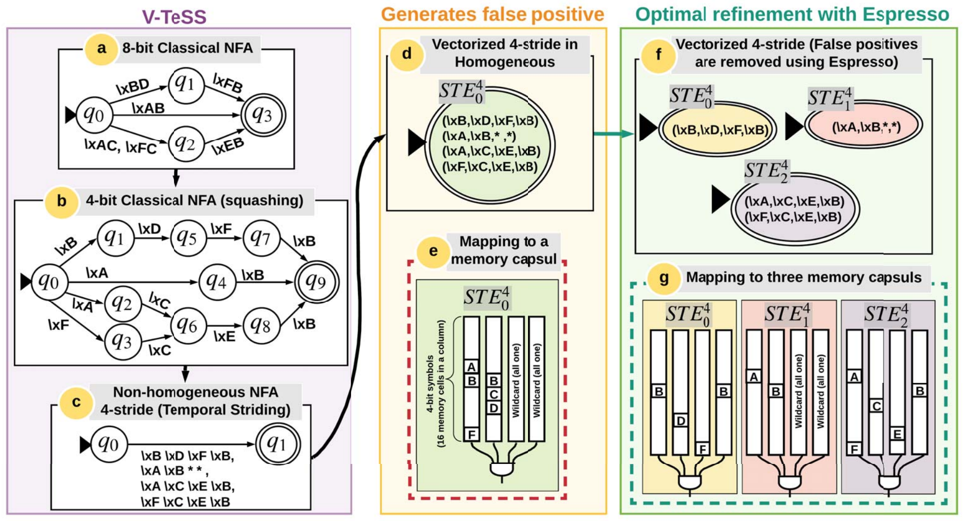

V-TeSS
Generates false positive
Optimal refinement with Espresso
a 8-bit Classical NFA
b 4-bit Classical NFA (squashing)
c Non-homogeneous NFA 4-stride (Temporal Striding)
d Vectorized 4-stride in Homogeneous
e Mapping to a memory capsul
f Vectorized 4-stride (False positives are removed using Espresso)
g Mapping to three memory capsuls
(\xB,\xD,\xF,\xB) (\xA,\xB,*,*) (\xA,\xC,\xE,\xB) (\xF,\xC,\xE,\xB)
4-bit symbols (16 memory cells in a column)
Wildcard (all one)

Figure 3: (a) Original 8-bit automaton. (b) Squashing the automaton in (a) to 4-bit processing. (c) Striding 4-bit automaton in (b) to process 16 bits/cycle. (d) Converting the automaton in (c) to its homogeneous representation. (e) Mapping STE_0^4 in (d) to one capsule, which produces false positive reports (e.g., $(\backslash xB, \backslash xD, \backslash xE, \backslash xB)$ generates a false report). (f) Espresso solves it with minimal state splitting. (g) The states in (f) are mapped to three capsules.

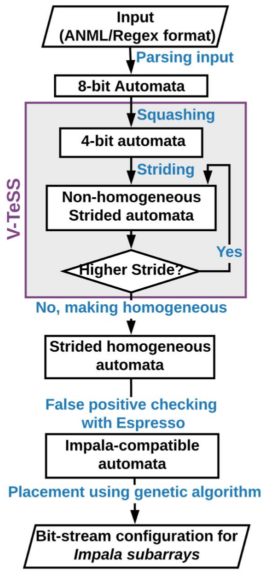

Input (ANML/Regex format)
Parsing input
8-bit Automata
Squashing
4-bit automata
Striding
Non-homogeneous Strided automata
Higher Stride?
Yes
V-TeSS
No, making homogeneous
Strided homogeneous automata
False positive checking with Espresso
Impala-compatible automata
Placement using genetic algorithm
Bit-stream configuration for Impala subarrays

Figure 4: Offline pre-processing steps to prepare bit-streams to be configured on Impala's memory subarrays.

in Impala's area-efficient matching architecture (described in Section 5.1). The last two stars in the matching symbol $(\backslash xA, \backslash xB, *, *)$ are wildcards, which can be matched against any 4-bit symbol. Wildcards are used as a padding method to handle the cases where a report happens in the middle of a 16-bit input (the original automaton in (a) reports when the input is $\backslash xAB$).

We use the vectorization concept for efficient hardware mapping. The temporal striding method is already discussed in prior work [51] and is not a contribution of this paper. Therefore, we do not discuss the details here.

Hardware efficiency: Naively increasing the processing rate to achieve higher throughput in the classical 8-bit in-memory architectures leads to a significant hardware cost. This is because each extra bit will double memory column size (e.g., 9-bit processing requires memory columns of 512 cells). This hardware consumption rate rapidly exceeds the reasonable bitline length due to its exponential growth. In addition, designing long bitlines is impossible without introducing stacked memory subarrays with partial address decoding and costly peripheral hardware. While the exponential hardware cost discourages a naive memory-based solution for multi-symbol matching, Impala redesigns the matching architecture to a set of short and parallel memory columns (16 cell in each column) combined with an AND gate, where every 4-bit increase in processing rate only requires an additional parallel 16-bit memory column. Columns are placed in different subarrays, and each receives 4 bits of the input symbols and processes it independently. This is a linear increment of hardware cost compared to the exponential growth in the naive matching architecture.

We call each of these combined memory columns a *capsule*. Capsules are suitable for states with a simple matching character-set. For example, a single capsule can easily handle the states with one matching vector, which are frequent in real-world automata benchmarks (see Fig. 2). However, there are infrequent matching cases that a single capsule can not precisely implement, and this may lead to generating false-positive reports. Figure 3(e) shows how using only one capsule to implement STE_0^4 generates wrong reports when the input is $\backslash xBDEB$. The first column matches $\backslash xB$, the second column matches $\backslash xD$, and the third and fourth columns match $\backslash xE$ and $\backslash xB$, respectively. However, this input is not supposed to be matched by STE_0^4. To address this issue, we exploit Espresso [30] - a tool that was originally developed for logic minimization problems - to find the minimum symbol splits to avoid false positives. In our example, splitting STE_0^4 to three states (Figure 3 (f)) and mapping them three capsule (Figure 3 (e)) solves the problem. Details of this refinement stage are explained in Section 5.1.2.

State and transition overhead: Temporal striding generally increases the length and cardinality of the alphabet and transition count [18, 51]. We apply squashing, striding, and false-positive matching removal on 21 real-world and synthetic automata benchmarks [32, 52] and observe that 2-stride and 4-stride implementations have a slight state and transition overhead. However, in the 8-stride design, combinatorial growth in the number of symbols (vectors) causes higher state/transition overhead, and this amortizes the architectural density benefits (details are discussed in Section 8.1). Our experimental results reveal that 4-stride (16-bit processing) yields the highest throughput per unit area (Section 8.4).

Theoretically, the area overhead of Impala's state-matching architecture is in the order of $O(SN)$, where S is the stride value (number of memory columns in a capsule is S) and N is the number of nodes. The interconnect area usage is in the

AP (APCompile [53])	FPGA (Xilinx tools)	CA (APSim)	Impala 4-stride (APSim)
>3 hours	~1 day	~5 minutes	~30 minutes

Table 1: Relative compilation time across architectures.

order of $O(N^2)$ (for a fully connected interconnect), which is independent of stride value. This implies that striding does not directly add any area overhead to the interconnect (except the increase of nodes count). However, the increase in the number of transitions means more switches in a full-crossbar will be utilized. To efficiently map the transitions in the strided automata to the Impala's interconnect resources, we adopt a genetic algorithm, and then generate bit-stream configurations for Impala's state-matching and interconnect subarrays. Figure 4 summarizes the offline pre-processing steps to map an automaton to Impala's architecture.

Compile time: Combined algorithmic and architectural benefits of Impala result in high-throughput and area/energy-efficient design at the expense of higher compilation time (due to some stages in Figure 4) compared to CA. The compilation is only needed once per automaton before configuring the final bit-stream on the architecture. Table 1 shows a relative comparison for the compile/synthesis time for ANMLZoo [32] on different architectures. Impala's pre-processing stage (which includes V-Tess and GA) increases the compilation time compared to CA, but it is still much less than the AP compiler and FPGA synthesis tools. APSim is our open-source automata transformation and simulator and can be found here [1].

5. ARCHITECTURAL DESIGN

5.1 State Matching

The blue box in Figure 5 shows the state-matching architecture, which processes four 4-bit symbols and detects all the states that match against them. The matching operation is performed in a group of memory subarrays to distribute the matching burden among them in parallel. Memory columns with the same index (or the same color in the figure) in subarrays are combined using an AND gate to form a capsule. Matching of each state will be assigned to one of these capsules. Each memory column in a capsule processes a portion (e.g., 4-bit) of the input symbols by a single memory row access. The final single-bit matching result for each state is the output of the AND gate in each capsule. Any additional 4-bit input processing only adds one more column to each capsule. Now, it is clearer how each dimension in V-TeSS vectors can be mapped to the corresponding memory column index in a capsule. Thanks to our parallel architecture, the latency of the overall matching for every stride value is always equivalent to read-cycle latency of a short bitline memory subarray plus the latency of the AND gate in capsule.

5.1.1 Challenges of Matching with Capsules

As discussed in section 4 and shown in Figure 3 (d), simply assigning each state to one capsule suffers from a possible false-positive match. Figure 6 shows a more general example of when a false match happens in the relatively complex matching regions of a 2-stride automaton using a 2D representation. In this Figure, the X-axis shows the matching symbols of the first dimension (first 4-bit), and the Y-axis shows the matching symbols of the second dimension (second 4-bit). Each colored rectangle represents a matching region.

[1]https://github.com/gr-rahimi/APSim

For example, the bottom-left-most region accepts the symbol range of $[\backslash x2-\backslash x5]$ in its first dimension and symbol range of $[\backslash x1-\backslash x3]$ in the second dimension. Impala's compiler also uses a memory-efficient range-based data-structure to store the matching data and this speeds-up automata transformation. Configuring all these colored regions in a single capsule generates false positive matching when the input is from the white rectangular in the bottom-right-most region (i.e., $[\backslash x9-\backslash xC]$ in first dimension and $[\backslash x1-\backslash x3]$ in the second dimension).

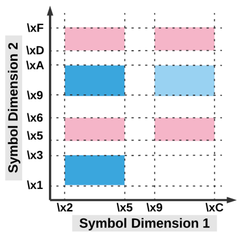


Figure 6: An example of an STE matching regions in a 2-stride automaton.

To address this issue, an easy solution would be to split the seven colored regions to seven new states and assign one capsule to each of them. However, this can be very costly, especially when a state matches against several regions. We observed that splitting into three states with matching regions shown as pink, dark blue, and light blue, and assigning each state to one capsule, would efficiently solve the problem.

However, refining matching regions to a minimal number of sub-regions to increase the total capacity is a challenging problem, especially for high stride values with many matching regions intersecting each other. We observed that this minimization problem is equivalent to the minimization version of set covering problem, which is known to be NP-hard [54]. The next section explains how we solve this problem efficiently using an existing tool.

5.1.2 Espresso

Espresso [30] is a CAD tool developed at IBM using heuristics for efficiently reducing the complexity of digital circuits. Interestingly, our state-splitting problem is similar to the Sum Of Product (SOP) gate-level logic minimization problem with the support for multi-valued variables [55]. In an SOP minimization problem, Espresso tries to minimize the number of products to use fewer resources in traditional Programmable Logic Arrays (PLA) hardware [56]. Impala can be seen as a PLA with 16-valued input variables, where each memory column in a capsule is a discrete variable with 16 different values. Columns in a capsule are combined with an AND gate which translates to a product term of 4 variables each with 16 values.

State refinement: In our problem, each product term will physically be translated to a new state and implemented using a capsule. The new states will replace the original state by connecting all the parents and children of the original state to all the new states. If the original state has a self-loop, then all the new states should have self-loop as well and must be connected to each other to keep the equivalence to the original automaton. Fig. 7 shows an example of how the state

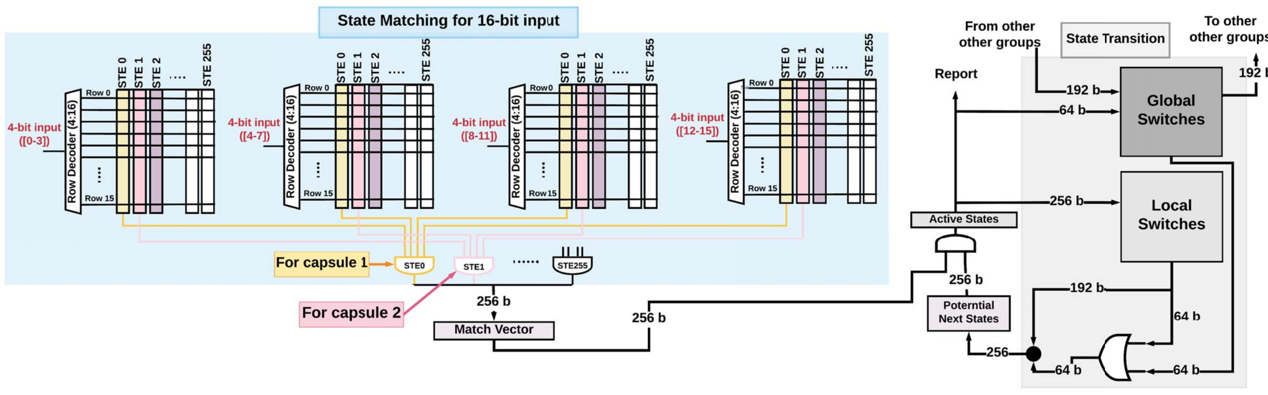


Figure 5: A 4-stride (16-bit) automata processing unit with a 2-level switch structure to support a larger automaton.

in *red* (which causes false positives) is split into two green states. The number of splits is determined by Espresso.

Espresso input/output: The input for Espresso is a text file containing the matching vector of the under process states, represented as multi-valued truth tables. The output of the Espresso is also a text file that specifies the minimum number of required product terms to cover the original matching space. Each of these products is guaranteed to cause no false positive and can be safely configured in Impala's capsule.

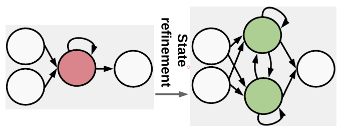


Figure 7: Splitting a state to avoid false positives.

5.2 Interconnect

The interconnect provides the functionality to move active states forward in time toward the next states. A state S gets activated if (1) the current symbol matches the state S and (2) any of its parents were activated in the previous cycle. The second condition implies that the interconnect should provide the OR-functionality. CA [22] proposes a memory-mapped full-crossbar interconnect based on 8T SRAM memory cells to provide wired-OR functionally on bitlines. The memory blocks are of size 256×256 (local switches), the word-lines (rows) are driven by a set of states (one row per state), and bitlines (columns) drive the same set of states (one column per state). This arrangement accommodates connection among every pair of 256 states, as every column intersects with every row. The memory cell at row i and column j stores '1' if there is an edge between state i and state j. To support larger automata with more than 256 states, a two-level interconnect model is proposed to provide inter-block connectivity among local switches using global switches.

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. The CA crossbar switch is utilized by packing CCs as densely as possible using a greedy approach. Figure 8 is an example of how two CCs, each with 100 states, are packed in a local switch block. We found that there are two problems with the state placement to the interconnect resources in CA model. First, the switch resource from index 200 to 256 in Figure 8 remains unutilized if the size of CCs are larger then unused portions. Second, if an automaton is larger than 256 states and has long-distance loops, it cannot be handled by CA's placement algorithm. Our general interconnect is based on CA's full-crossbar design. Sadredini et al. [29] showed that the connectivity patterns in real-world automata are diagonal-shaped in a full-crossbar design mapping, and this insight can be used to allow for reduced crossbar switches. We take inspiration from this observation and present a placement solution that automatically addresses the issues in CA placement for reduced crossbars using a genetic algorithm.

The gray box in Figure 5 shows the interconnect architecture using the local switch and global switch. Local switches are driven by the currently active states, and the output of the interconnect subarray is combined with the matching signals to compute the active states for the next cycle. Impala expands the connectivity among local switches by letting 64 nodes in a local switch (called *port-nodes (PNs)*) have full connectivity to all the port nodes of three other local switches using a dedicated 256×256 global switch subarray. 64 PNs in local switches are combined with 64 outputs of global switches to provide inter-group connection. We call the set of 4 local switches communicating with each other by their PNs (using a global switch) as *group of four (G4)* (see Figure 10 (a)).

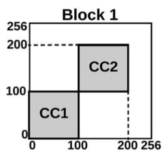


Figure 8: Full-crossbar resource utilization.

To address the placement problems in CA (to be able to efficiently place any large automaton with up to 1024 states in a G4 or break a small CC among two local switches to utilize the unused portion of crossbar), a placement strategy is required to consider (1) the limited inbound and outbound connectivity in PNs and (2) the connectivity pattern of an automaton to assign an integer label to each state, which later is mapped to the interconnect resources in G4. Furthermore, striding makes the connectivity pattern more sophisticated as strided automata have more transitions [51]. Figure 9 shows

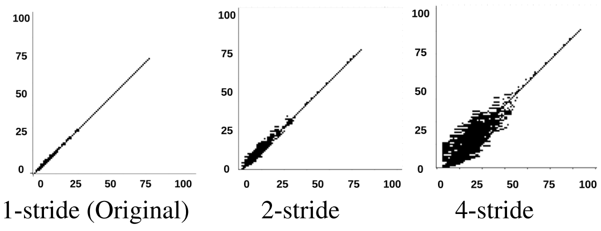


Figure 9: Union heatmap of routing switches with BFS labeling for all the connected components in Dotstar06. States are labeled with BFS starting from the start states. Each dark point at (x,y) shows an edge from state y to state x.

the effect of striding on the interconnect pattern evolution for Dotstar06 benchmark in ANMLZoo [32]. The 4-stride automata have higher transitions than 1-stride, which translates to higher utilization in a full-crossbar interconnect.

5.2.1 *G4 Visualization*

Figure 10 (a) visualizes the expanded layout of all possible connectivity supported in a G4 in a diagonal-shaped structure (inspired from [29]) using all four local switches (gray rectangles) and the global switch (purple rectangles). X axis shows the source index and Y axis shows the destination index. For example, if point (x, y) is in the gray rectangle, it is possible to support connection from state indexed x to state indexed y using local switches, or if it is in any of the purple regions, the global switch can provide the connectivity. Otherwise, if it is not in any of those regions, it is not possible to support that connectivity. The G4 switches can accommodate CCs of up to 1024 states. In our experiments, after striding, all the connected components have fewer than 1024 states. To support even larger automata, a higher-level switch can be used to connect G4 switches. The global switches in the upper-left or bottom-right cover the long-distance loops. Impala's compiler uses a genetic algorithm (GA) to effectively try different combinations of assigned indices to states and find a solution with zero missing connection in G4.

5.2.2 *Placement using genetic algorithm*

A genetic algorithm (GA) is a computation model inspired by evolution and natural selection. In GA, the original problem model is interpreted as chromosome-like data representation, and evolution happens through three main operations: selection, crossover, and mutation. GA begins with a set of random chromosomes called a population. A population tries to evolve to a better population in each generation by prioritizing the fittest individuals (based on a goodness definition) to mate, mutate, and evolve into a new population of individuals.

In Impala, we encode our placement assignment by defining each individual as an array of unique integers of length 1024 (the number of states that fit in a G4). We initially assign unique labels to each state using the BFS algorithm or random generators to fill the population for the first time. The red dots in Fig. 10 (b) show the required connections when states are indexed using BFS in G4. As can be seen, there are many locations where the red dots locate outside of local switches and regions covered by global switches. This situation is anticipated, as BFS labeling assigns indices only

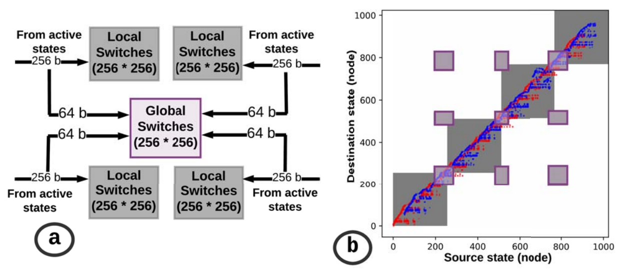


Figure 10: (a) G4 switch model and (b) its visualization.

based on the automata transition itself and does not consider the G4 connectivity limitations.

Impala's genetic algorithm placement evaluates the goodness of each individual based on the number of necessary switches that are currently available in G4. This evaluation increases the chance of individuals with better answers to survive into the next generation. Crossover combines individuals to generate a new individual by inheriting from their parents. Impala's compiler uses an ordered crossover method which swaps a random interval of two individuals with each other while keeping the order. Mutation happens by randomly swapping numbers inside each individual array. The blue-dots in Fig. 10 (b) shows the state mapping in G4. This indicates that our GA can successfully place all the states within the G4 switch covering regions; therefore, this state labeling is a valid placement solution.

Case Study: EntityResolution from the ANMLZoo benchmark suite is widely used for approximate string matching in databases, and it has been shown that it has a complex connectivity pattern [29]. It has 1000 CCs, and on average, each CC has 95.12 states. We apply V-TeSS to this benchmark to generate the 4-stride automata, where now each CC has 108.9 states on average. Our placement algorithm is able to fit all these CCs in 117 G4 switches (on average, 930.7 states are mapped to each G4). Interestingly, none of these CCs can fit in G4 using a BFS labeling. We run the same experiments for all the benchmark in Table 2 and discover that our placement algorithm can successfully map all the states to G4 switch patterns for up to 4-stride automata.

6. SYSTEM INTEGRATION

Configuration: Impala can be realized by re-purposing the available on-chip SRAM memory or custom-designed memory arrays. In on-chip SRAM arrays, column height for local subarrays are 256 [22], and Impala can partially utilize the available resources (16 out of 256 rows). However, the original 8-bit automata can be strided, which then can process multiple 8-bit symbols and utilize all 256 rows (evaluated in Section 11). On-chip solutions are more suitable for small scale applications where off-chip communication overhead becomes a concern, and low-latency designs are more appropriate. On the other hand, off-chip designs are more desired for large-scale applications where state capacity plays an important role, and I/O becomes less of a concern.

Impala is fully memory-based, and setting memory values configures the system for processing an input stream. After compiling all the automata and applying squashing, striding, and placement, the values for each memory cell in matching and interconnect arrays are determined. Assuming Impala is integrated into a system as a peripheral device, these values can be transferred to Impala using memory-mapped I/O communication such as Linux *mmap* command [57] or PCI commands. Fulcrum [58] proposes to employ CXL [59] interface for in-memory accelerators, which enables another option.

Run-time: Impala has two asynchronous FIFOs to hold the input symbols in the input buffer (IB) and reports in the output buffer (OB). At runtime, the host system communicates with the IB and OB using interrupt triggered memory-mapped IO or DMA while the interrupt service routine (ISR) is responsible to fill in the IB and evict the OB. Assuming 5GHz frequency for Impala and 1 MHz frequency for interrupt, an IB of size 2.5KB can store enough data to feed the Impala until the next IB interrupt. Recently, [60] has characterized the reporting statistics of ANMLZoo's benchmark. The results show that 10 out of 12 benchmarks produce fewer than 0.5 reports per cycle. This investigation motivates us to use 512 entries for the OB (4 bytes each for report meta-data) in order to keep a similar interrupt rate as the IB.

7. EVALUATION METHODOLOGY

NFA workloads: We evaluate our proposed claims and architectures using ANMLZoo [32] and Regex [52] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. AutomataZoo [61] is mostly an extension of ANMLZoo (9 out of 13 applications are the same), and the difference is that ANMLZoo is normalized to fill one AP chip (with up to 48K states). To (i) provide a fair comparison with the AP, and (ii) evaluate on real-size applications, we use ANMLZoo benchmarks, but replicate each benchmark 1000 times to create larger benchmarks. We present a summary of the applications (in the original size) in Table 2.

Benchmark	#Family	#States	#Transi-tions	Ave. Node Degree	Largest CC Size
Brill [32]	Regex	42658	62054	2.9	67
Bro217 [52]	Regex	2312	2130	1.8	84
Dotstar03 [52]	Regex	12144	12264	2.0	92
Dotstar06 [52]	Regex	12640	12939	2.0	104
Dotstar09 [52]	Regex	12431	12907	2.0	104
ExactMath [52]	Regex	12439	12144	1.9	87
PowerEN [32]	Regex	40513	40271	1.9	52
Protomata [32]	Regex	42009	41635	1.9	123
Ranges05 [52]	Regex	12621	12472	1.9	94
Ranges1 [52]	Regex	12464	12406	1.9	96
Snort [32]	Regex	100500	81380	1.6	222
TCP [52]	Regex	19704	21164	2.1	391
ClamAV [32]	Regex	49538	49736	2.0	515
Hamming [32]	Mesh	11346	19251	3.3	122
Levenshtein [32]	Mesh	2784	9096	6.5	116
Fermi [32]	Widget	40783	57576	2.8	17
RandomForest [32]	Widget	33220	33220	2.0	20
SPM [32]	Widget	69029	211050	6.1	20
EntityResolution [32]	Widget	95136	219264	4.6	96
BlockRings [32]	Synthetic	44352	44352	2.0	231
CoreRings [32]	Synthetic	48002	48002	2.0	2

Table 2: Benchmark Overview

Experimental setup: We use our Automata compiler and simulator, APSim [62], to perform the pre-processing steps (such as V-TeSS), and emulate Impala and CA [22]. We have verified the functional correctness of APSim with VASim [63], which is an open-source automata simulator. The simulator takes automata in ANML format and processes the input cycle-by-cycle. Per-cycle statistics are used to calculate the number of active subarrays, which is then used to calculate energy consumption. To estimate area, delay, and power of the memory subarray in Impala and CA model, we use a standard memory compiler (under NDA) for the 14nm technology node and nominal voltage 0.8V (details in Table 3). The global wire-delays are calculated using SPICE modeling in CA. In the memory compiler, the 8T-cell design has wider transistors than 6T; therefore, 8T subarrays are faster and have higher area overhead than 6T subarrays. Moreover, because the AP, CA, and Impala have similar run-time execution models, we can disregard data transfer and control overheads to make general capacity and performance comparisons among these platforms.

Usage	Cell Type	Size	Delay (ps)	Read Power (mW)	Area (μm^2)
State-matching (Impala)	6T	16×16	180	0.58	453
State-matching (CA)	6T	256×256	220	5.52	9394
Interconnect	8T	256×256	150	6.07	20102

Table 3: Subarray parameters for state-matching and interconnect (overhead of peripherals are included).

Comparison metric: To compare spatial automata processing architectures (the AP, CA, and Impala), we use *throughput per unit area*. Throughput is defined as the number of bits that can be processed in one second ($frequency \times Bitwidth\_size$). If the automata (connected components) 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.

8. EXPERIMENTAL RESULTS

In this section, we evaluate Impala and compare it with the state-of-the-art solutions, such as CA and the AP, as well as multi-striding solutions on FPGAs.

8.1 Overhead Analysis of V-TeSS

Squashing to 4-bit design and then striding (V-TeSS) change the shape of automata and increases the number of states and transitions. Table 4 shows the number of states and transitions in V-TeSS in different strides normalized to the original 8-bit automata designs across 21 automata benchmarks. We observed that applications with higher average node degree, such as EntityResolution, RandomForest, and SPM (see Table 2) result in higher state and transition overhead. This is mainly because more combinations of paths need to be processed in temporal striding. On the other hand, CoreRings and BlockRings have almost no overhead when striding (8-bit and 16-bit). This is mainly because all states have equivalent matching symbols (a single unique symbol) which can effectively benefit from the classic NFA minimization techniques such as prefix merge and suffix merge (both implemented in Impala's compiler). The state and transition overhead of 8-stride automata is higher; this is because the number of symbols (32-bit symbols) increases, which results in more vectors and a higher chance for false positives in each state. Splitting the states extensively in 8-stride causes higher state and transition overhead. This, in turn, surpasses the area benefits of Impala. Therefore, we evaluate Impala for up to 4-stride design for the rest of the paper.

	#States Normalized to 8-bit original design				#Transitions Normalized 8-bit original design			
Stride	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
Bits per cycle	**4-bit**	**8-bit**	**16-bit**	**32-bit**	**4-bit**	**8-bit**	**16-bit**	**32-bit**
Brill	2.18	1.05	2.03	16.99	1.87	1.07	3.31	42.35
Bro	2.19	1.08	1.17	4.29	2.50	1.18	1.28	4.94
Dorstar03	2.24	1.05	1.19	2.05	2.84	1.15	2.03	3.08
DotStar06	2.33	1.05	1.22	2.38	3.21	1.17	2.16	3.82
Dotstar09	2.45	1.06	1.25	2.53	3.65	1.18	2.30	4.29
ExactMatch	2.05	1.02	1.05	1.40	2.12	1.05	1.07	1.44
PowerEN	2.46	1.08	1.26	6.04	3.66	1.18	1.72	8.77
Protomata	3.08	1.44	1.98	6.22	4.01	2.03	3.63	7.43
Ranges05	2.08	1.03	1.10	3.38	2.24	1.09	1.50	4.14
Ranges1	2.10	1.04	1.15	3.60	2.29	1.13	1.80	4.99
snort	2.79	1.12	1.56	10.25	4.87	1.34	3.73	22.48
TCP	2.47	1.10	1.52	8.55	3.56	1.37	4.08	17.24
ClamAV	2.03	1.00	1.03	7.01	2.06	1.01	1.06	7.42
Hamming	1.99	1.01	1.73	22.97	1.59	1.01	2.65	31.31
Levenshtein	2.66	1.01	2.52	5.35	1.79	1.02	4.19	11.25
Fermi	2.23	1.03	1.06	26.75	2.11	1.04	1.34	30.71
RandomForest	5.07	1.82	3.74	27.75	9.22	3.42	13.97	82.19
SPM	2.60	1.40	5.11	11.88	2.55	2.21	23.44	32.20
EntityResolution	3.45	1.05	1.73	1.08	4.00	1.06	3.11	1.10
BlockRing	2.01	1.00	1.01	3.61	2.02	1.01	1.03	3.98
CoreRing	2.00	1.00	1.00	1.00	2.00	1.00	1.00	1.00
Average	2.52	1.12	1.68	8.34	3.10	1.34	3.97	15.53

Table 4: States and transitions overhead in different strides for V-TeSS normalized to the original 8-bit design.

Squashing the 8-bit design to 4-bit increases the number of states 2.52×, and then striding the 4-bit design to 8-bit (2-stride) and applying compiler minimizations reduces the number of states very close to the original 8-bit design. Both 2-stride 4-bit and original 8-bit have similar throughput; however, the 4-bit design requires substantially smaller memory subarrays. On average, the state overhead in 16-bit design (processing four 4-bit symbols per cycle) is only 1.7× compared to the original 8-bit design, but its processing rate is twice and the design is denser. This explains that our V-TeSS method co-designed with Impala architecture provides a three-fold throughput, area, and energy-efficiency benefits compared to prior 8-bit architectures (details in the following Sections). This also confirms that Espresso has split the false-positive states with minimal overhead. It is important to note that transition overhead translates to higher utilization of the crossbar interconnect and does not impose extra hardware overhead (discussed in Section 5.2 and evaluated in Section 8.3).

8.2 Overall Performance

The overall performance of spatial automata processing architectures is determined by $frequency \times bits/cycle$. The delays and frequencies of different pipeline stages for Impala, CA, and the AP are shown in Table 5. Impala's state-matching delay is 180 ps (See Table 3) and it is similar for different stride designs. This is because all capsules are processed in parallel, and striding does not increase the pipeline stages delay (except for the minor difference in the 2-input vs 4-input AND gate delay in Impala design, which is less than 4ps in 14nm [65]. This is less than 2% of total delay in the state-matching stage). Both CA and Impala have similar hierarchical interconnect designs, and both local and global switches are evaluated in parallel (Figure 5). Following CA design, we assume the SRAM-based CA design slice of $3.19mm \times 3mm$. Therefore, the distance between SRAM arrays and global switch is assumed to be $1.5mm$. From SPICE modeling, the wire delay was found to be $66ps/mm$; therefore, the wire delay for global switches is $99ps$. Global switch delay for CA is 249 ps, which is composed of read-access latency (150ps)

Architecture	State Matching	Local Switch	Global Switch	Max Freq. (GHz)	Operating Freq. (GHz)
Impala (14nm)	180 ps	150 ps	170 ps	5.55	5
CA (14nm)	220 ps	150 ps	249 ps	4.01	3.6
AP (50nm)	-	-	-	0.133	0.133
AP (14nm)*	-	-	-	1.69	1.69

* Projected to 14nm

Table 5: Pipeline stage delays and operating frequency. The detail implementation of the AP is not publicly available.

and wire delay latency (99ps). Impala state-matching size for 4-stride design is ~5× less; therefore, we assume $20ps$ wire-delay for Impala. Therefore, the global switch delay for Impala is 170 ps (150ps+20ps).

The frequency is determined based on the slowest pipeline stage, which is the global switch delay in both CA and Impala. We assume the operating frequency for them to be 10% less than what we have calculated, to consider potential estimation errors. The AP is designed in 50nm DRAM technology. To have a fair comparison, we project the frequency to 14nm technology, which is an ideal assumption.

In all these spatial architectures, state matching and routing happen in parallel. This, in turn, means that they have a deterministic throughput of one input symbol per cycle, and it is independent of the input stream. Figure 13 presents the overall achieved throughput for CA, AP, and Impala different stride designs. Impala 4-stride design processes 16 bits per cycle and achieves the highest throughput (5GHz×16-bit=80 Gbps). This implies that if the application fits in the hardware, the Impala 16-bit design provides 2.8× higher throughput than CA. *2× of the benefit comes from the algorithmic contribution, which reshapes the automata to process 16-bit symbols, and 1.4× from architectural contribution in which shorter subarray design results in higher frequency*. Moreover, Impala 16-bit has 5.9× higher throughput than the AP.

8.3 Area Overhead

Impala proposes area benefits coming from its architectural contribution, which is smaller state-matching subarrays and sharing interconnect resources while processing 16 bits per cycle. Figure 14 compares the area overhead of state-matching and interconnect of Impala 16-bit processing with CA and the AP (all in 14nm) for 32K STEs. Impala 16-bit has 5.2× and 34.5× smaller state-matching area overhead than CA and the AP (scaled to 14nm), respectively. Moreover, in total, Impala 16-bit has 1.34× and 3.9× smaller area overhead than CA and the AP, respectively. Sadredini et al. [29] show that the AP interconnect incurs routing congestion and limits the state-matching utilization. This implies that the area overhead to accommodate 32K states would be higher in practice for the AP.

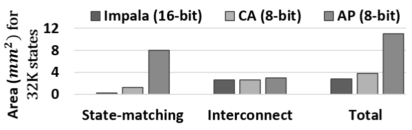


Figure 14: Comparing area overhead for 32K STEs.

8.4 Throughput per unit area

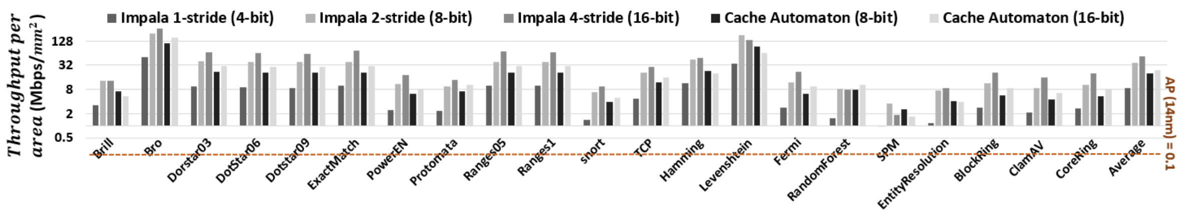
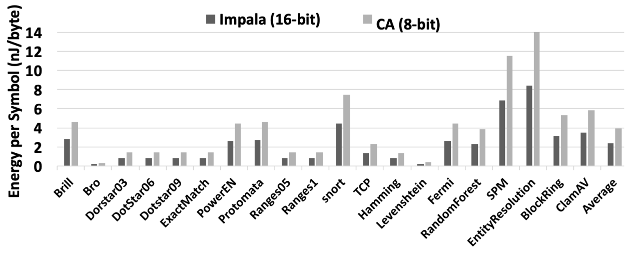


Figure 11: Comparing throughput per mm^2 area among Impala 4-bit design in different strides, Cache Automata (original 8bit design) in 1-stride and 2-stride, and the Automata Processor (AP), all in 14nm.



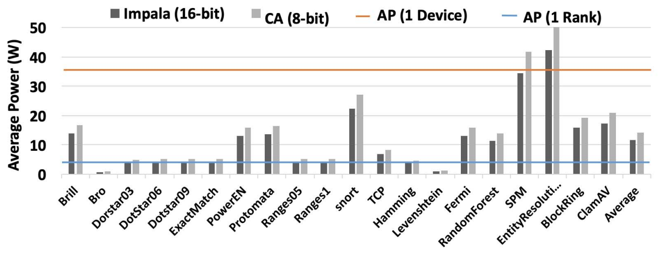


Figure 12: (left) Overall energy consumption of Impala compared to CA. (right) Overall power consumption of Impala compared to CA and the AP (reported by Micron [64]).

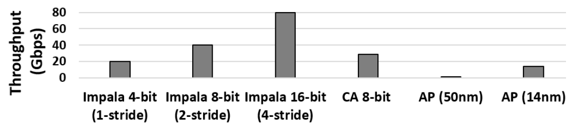


Figure 13: Overall performance of different spatial automata accelerators in Gbps.

This section combines throughput, area, and striding overhead effects all together and evaluates throughput per unit area across 21 applications. To have a more comprehensive comparison, we stride the original 8-bit automata using our temporal striding method and evaluate the Cache Automaton for 16-bit processing. This is shown as Cache Automaton (16-bit) in the Figure 11. From this figure, the Impala 16-bit design provides on average 2.7× (up to 3.7×) and 371× (up to 536×) throughput per unit area than CA and the AP, respectively. The benefits are calculated as $\frac{Throughput\ benefit\ \times\ Area\ benefit}{V-Tess\ overhead}$. For example, the benefits of Impala 16-bit over CA 8-bit are calculated as $\frac{2.8\times1.34}{1.39}$, where the V-Tess state overhead (1.39×) is calculated for real-size applications. The applications such as Bro, Dotstar03, ExactMatch, snort, and Fermi have smaller striding overhead for 2-stride and 4-stride designs (Table 4), and therefore, they present higher throughput per area in Impala 16-bit. On the other hand, SPM has a higher average node degree (Table 2), which results in a higher striding overhead (Table 4) and lower throughput per area than Cache Automaton 8-bit.

8.5 Energy/Power Consumption

This section discusses the energy/power consumption of Impala 16-bit and compares it to prior works assuming 10MB of input. To calculate energy consumption, we need to know (1) the number of active partitions for state-matching and switch blocks, and (2) the number of transitions between local switches to consider for the energy of driving wires.

Note that it is not possible to power-gate state-matching memory arrays on a cycle-by-cycle basis. In order to power-gate these subarrays, it is necessary to know the potential next states beforehand. However, in the pipeline, the state-matching results and the next potential state are calculated simultaneously, which prevents the power-gating (one can still power-gate an array that is unoccupied). This observation is not considered in CA. We update the energy/power results in CA paper [22] based on this observation and our 14nm technology assumption. All the statistics per cycle are extracted from our compiler.

Figure 12 (left) shows the energy per input symbol for Impala and CA (energy details of the AP is not publicly available). We can observe that benchmarks with a larger number of states, such as Entity Resolution, Snort, and SPM consume higher energy. This is because these benchmarks have utilized more state-matching and switch arrays to accommodate a larger number of states. On average, CA consume 1.7× more energy per symbol than Impala. Energy efficiency of Impala comes from its density and compact design, which results in consuming lower dynamic energy due to shorter wires. Figure 12 (right) shows the average power consumption across benchmarks. On average, the power consumption of CA is 1.22× more than Impala. This is expected because: $\frac{CA-Energy}{Impala-Energy} \times \frac{CA-Frequency}{Impala-Frequency} = 1.7 \times \frac{1}{1.39} = 1.22$.

8.6 Comparison with multi-stride on FPGA

Yang et al. [4] and Yamagaki et al. [18] propose multi-stride regex processing solutions on FPGA and have evaluated their solutions on Xilinx-Virtex5 LX-220 and Altera Stratix II EP2S180, respectively (details are discussed in the related work in Section 3). Table 6 compares Impala with these solutions for 16-bit symbol processing rate on Snort dataset. In summary, Impala provides ~20× higher frequency and ~20× higher throughput than both of these solutions. Moreover, Impala with 16-bit processing rate has 7.7× higher throughput than these FPGA solutions for 64-bit processing rate. Imapla's compiler (pre-processing and placement) is at least an order of magnitude faster than FPGA synthesis

(Table 1). The benefit of our approach, i.e., processing 4-bits symbols, can be applied to FPGAs, as we concluded from our preliminary FPGA-based experiments. Further exploration is left for future works.

	Bits/cycle	**Clock rate (GHz)**	**Throughput (Gbps)**
Yang et al. [4]	16	0.212	3.47
Yamagaki et al. [18]	16	0.239	3.91
Impala	16	5	80

Table 6: Comparison with mutli-stride FPGA solutions.

9. CONCLUSIONS

This paper presents Impala, an in-memory accelerator for an efficient multi-stride automata processing. Impala is co-designed with our automata transformation algorithm, called V-TeSS, and leverages short and parallel memory columns to implement a dense, high-throughput, and low-power multi-symbol matching architecture. Overall, the benefits of Impala comes from two observations: (1) smaller state-matching sub-arrays provide higher utilization of memory-cells in memory columns, and this translates to higher density, and (2) V-TeSS transformation provides higher throughput with a linear increase in state-matching resources (or memory columns) relative to the original 8-bit design. *This paper concludes that an in-situ 4-stride automata processing with 16-bit memory columns provides the highest performance, and has up to 3.7× higher throughput per area and 1.22× lower power consumption than Cache Automaton.*

10. ACKNOWLEDGEMENTS

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.

11. REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[2] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 30–39, ACM, 2009.

[3] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pp. 93–102, ACM, 2006.

[4] Y.-H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA," *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 1013–1025, 2012.

[5] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "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, 2018.

[6] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 13, no. 1, pp. 99–111, 2016.

[7] E. Sadredini, R. Rahimi, K. Wang, and K. Skadron, "Frequent subtree mining on the automata processor: challenges and opportunities," in *International Conference on Supercomputing (ICS)*, ACM, 2017.

[8] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the micron automata processor," in *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 135–144, ACM, 2016.

[9] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[10] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA pattern matching on gpgpu devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.

[11] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: a fast multi-pattern regex matcher for modern cpus," in *16th USENIX Symposium on Networked Systems Design and Implementation*, pp. 631–648, 2019.

[12] L. Vespa, N. Weng, and R. Ramaswamy, "MS-DFA: Multiple-stride pattern matching for scalable deep packet inspection," *The Computer Journal*, vol. 54, no. 2, pp. 285–303, 2010.

[13] M. Becchi and P. Crowley, "A-DFA: A time-and space-efficient DFA compression algorithm for fast regular expression evaluation," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[14] Intel. `https://github.com/01org/hyperscan`.

[15] J. Qiu, Z. Zhao, and B. Ren, "Microspec: Speculation-centric fine-grained parallelization for fsm computations," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques*, ACM, 2016.

[16] Z. Zhao and X. Shen, "On-the-fly principled speculation for fsm parallelization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2015.

[17] Z. Zhao, B. Wu, and X. Shen, "Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation," *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[18] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 131–136, IEEE, 2008.

[19] M. Avalle, F. Risso, and R. Sisto, "Scalable algorithms for NFA multi-striding and NFA-based deep packet inspection on gpus," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1704–1717, 2016.

[20] V. Kosar and J. Korenek, "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*, 2014.

[21] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 12, 2014.

[22] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[23] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "A scalable and efficient in-memory interconnect architecture for automata processing," *IEEE Computer Architecture Letters*, 2019.

[24] K. Wang, E. Sadredini, and K. Skadron, "Hierarchical pattern mining with the micron automata processor," in *International Journal of Parallel Programming (IJPP)*, 2017.

[25] E. Sadredini, D. Guo, C. Bo, R. Rahimi, K. Skadron, and H. Wang, "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*, pp. 665–674, ACM, 2018.

[26] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pp. 737–748, IEEE, 2018.

[27] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *International Conference on Semantic Computing (ICSC)*, IEEE, 2015.

[28] L. Gwennap, "New chip speeds NFA processing using DRAM

architectures," in *In Microprocessor Report*, 2014.

[29] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eAP: A scalable and efficient in-memory accelerator for automata processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 87–99, ACM, 2019.

[30] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 1987.

[31] V. M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, 1961.

[32] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, *et al.*, "Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2016.

[33] M. Lenjani, P. Gonzalez, E. Sadredini, M. A. Rahman, and M. R. Stan, "An overflow-free quantized memory hierarchy in general-purpose processors," *IEEE International Symposium on Workload Characterization*, 2019.

[34] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pp. 1–8, IEEE, 2017.

[35] T. T. Hieu and N. T. Tran, "A memory efficient FPGA-based pattern matching engine for stateful nids," in *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*, pp. 252–257, IEEE, 2013.

[36] R. Karakchi, L. O. Richards, and J. D. Bakos, "A dynamically reconfigurable automata processor overlay," in *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pp. 1–8, IEEE, 2017.

[37] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[38] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, "Hawk: Hardware support for unstructured log processing," in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pp. 469–480, IEEE, 2016.

[39] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 461–472, IEEE Computer Society, 2012.

[40] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pp. 533–545, IEEE, 2015.

[41] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron, "An overview of micron's automata processor," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*, pp. 1–3, IEEE, 2016.

[42] A. Subramaniyan and R. Das, "Parallel automata processor," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 600–612, IEEE, 2017.

[43] H. Liu, M. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 908–920, IEEE, 2018.

[44] F. Hamzaoglu, Y. Ye, A. Keshavarzi, K. Zhang, S. Narendra, S. Borkar, M. Stan, and V. De, "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*, pp. 15–19, IEEE, 2000.

[45] K. Zhang, U. Bhattacharya, Z. Chen, F. Hamzaoglu, D. Murray, N. Vallepalli, Y. Wang, B. Zheng, and M. Bohr, "Sram design on 65-nm cmos technology with dynamic sleep transistor for leakage reduction," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 4, pp. 895–901, 2005.

[46] A. Garg and T. T.-H. Kim, "Sram array structures for energy efficiency enhancement," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 6, pp. 351–355, 2013.

[47] R. Liu, X. Peng, X. Sun, W.-S. Khwa, X. Si, J.-J. Chen, J.-F. Li, M.-F. Chang, and S. Yu, "Parallelizing sram arrays with customized bit-cell for binary neural networks," in *Proceedings of the 55th Annual Design Automation Conference*, p. 21, ACM, 2018.

[48] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[49] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators," in *The International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2020.

[50] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *ACM SIGARCH computer architecture news*, vol. 34, no. 2, pp. 191–202, 2006.

[51] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 50–59, ACM, 2008.

[52] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 79–89, IEEE, 2008.

[53] K. Angstadt, W. Weimer, and K. Skadron, "Rapid programming of pattern-recognition processors," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 593–605, 2016.

[54] Wikipedia contributors, "Set cover problem — Wikipedia, the free encyclopedia," 2019. [Online; accessed 28-June-2019].

[55] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006.

[56] Wikipedia contributors, "Programmable logic array — Wikipedia, the free encyclopedia," 2019. [Online; accessed 28-June-2019].

[57] "Linux programmer's manual for mmap." `http://man7.org/linux/man-pages/man2/mmap.2.html`. Accessed: 2019-07-15.

[58] M. Lenjani, P. Gonzalez, E. Sadredini, S. Li, Y. Xie, A. Akel, S. Eilert, M. R. Stan, and K. Skadron, "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.

[59] "Compute express link." `https://www.computeexpresslink.org/`.

[60] J. Wadden, K. Angstadt, and K. Skadron, "Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 749–761, IEEE, 2018.

[61] J. Wadden et al., "AutomataZoo: A modern automata processing benchmark suite," in *IISWC*, IEEE, 2018.

[62] `https://github.com/gr-rahimi/APSim`.

[63] J. Wadden and K. Skadron, "VASim: An open virtual automata simulator for automata processing application and architecture research," tech. rep., Technical Report CS2016-03, University of Virginia, 2016.

[64] Micron, "Micron automata processor." `www.cs.virginia.edu/~skadron/grab/Skadron-Micron_AP_Briefing_Deck_13032014a.pdf`.

[65] V. B. Kleeberger, H. Graeb, and U. Schlichtmann, "Predicting future product performance: Modeling and evaluation of standard cells in finfet technologies," in *Proceedings of the 50th Annual Design Automation Conference*, ACM, 2013.