Parallel Automata Processing Review

Jose Chavez
Computer Science Department, University of Engineering and Technology
June, 2019
jose.chavez@utec.edu.pe, ginger.melo.jc@gmail.com

***Abstract*—A Finite State Machine (FMS) is an important abstraction for solving several problems. This computational model is widely used for many application domains, such as matching regular-expression, tokenizing and Huffman decoding. These embarrassingly sequential applications with irregular memory access patterns perform perform poorly on conventional von-Neumann architectures. The Micron Automata Processor (AP) is an in-situ memory-based computational architecture that accelerates non-deterministic finite automata (NFA) processing in hardware. However, each FSM on the AP is processed sequentially, limiting potential speedups. We explore the FMS parallelization problem in context of the AP. The classical parallelization techniques to NFAs executing on AP is non-trivial because of high state-transition tracking overheads and exponential computation complexity. We take a special look to a work that aims to improve performance by custom parallelization of FSM processing on AP.**

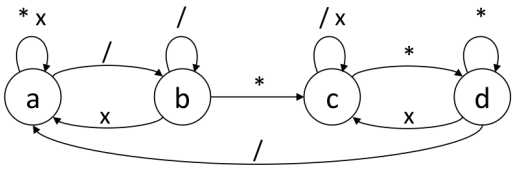
***Index Terms*—Computer systems organization, accelerators, Automata, Finite State Machine**

I. Introduction

WE already stablished that Finite State Machines (FSM) are widely used as a computation model in a number of application domains such as data analytics and data mining [9, 33], network security [13, 21, 24, 38], bioinformatics [11, 28, 36], tokenization of web pages [25], computational finance [1, 4] and software engineering [5, 10, 26]. These applications require processing tens to thousands of patterns for a stream of input data.

NFAs form the core of many end-to-end applications that utilize pattern matching. These pattern matching routines are typically implemented as if-else or switch-case nests in conventional CPUs and contribute to a significant fraction of the overall execution time because of poor branch behavior and irregular memory access patterns.

Figure 1(a) shows an example FSM. The table in Figure 1(b) determines how the FSM states transition on each input symbol, and Figure 1(c) is a straightforward implementation of this FSM that iteratively accesses the transition table to obtain the state after each input symbol. FSMs are difficult to parallelize for two reasons. First, there is a tight dependence between successive loop-iterations making it nontrivial to distribute loops across multiple processors. Second, FSMs perform little computation in each iteration with memory-access patterns that are input-dependent and unpredictable. This makes it difficult for FSM implementations to use parallelism within a processor, namely instruction-level parallelism,



(a)

T:	/	*	x
a	b	a	a
b	b	c	a
c	c	d	c
d	a	d	c

(b)

```
1  state = a;
2  foreach (input in)
3    state = T[in][state];
```

(c)

Fig. 1. AnAn FSM containing four states that accepts C style comments in source code delineated by /* and */. The input x represents all characters other than / and *.

vector (SIMD) capabilitiesand memory-level parallelism. Also the FSM computation, especially Non-Determinstic Finite Automata (NFA) computation is inherently hard to speedup. Modern multicore processors are limited by the number of transitions they can do per thread in a given cycle, limiting the number of patterns they can identify. Their processing capability is also limited by the available memory bandwidth. GPGPUs have limited success with automaton processing because it is inherently dominated by irregular memory access patterns. n comparison, custom architectures which facilitate in-situ com- putation in memories can facilitate highly parallel and energy effi- cient processing of finite state automata in hardware. For instance, Microns Automata Processor (AP) [12] has been shown to accel- erate several applications like entity resolution in databases [9](by 434) and motif search in biological sequences [28] (by 201). Recent efforts from Virginias Center for Automata Processing has demonstrated that AP can outperform GPGPUs by 32, and accel- erators such as XeonPhi by 62, across a wide variety of automata benchmarks [31]. Some key problems in bioinformatics like (28, 12), i.e., matching protein motifs of length 28, within edit distance 12 were previously unsolvable by von-Neumann architectures [28]. The Micron AP is a generalized accelerator supporting many ap- plication domains which can benefit from fast NFA processing and is not limited to regular ex-

pressions.The success of AP relies on three factors: massive parallelism, eliminating data movement (between memory and CPU) overheads, and reducing instruction processing overheads significantly. Massive parallelism follows from the fact that all state elements (mapped to columns in DRAM arrays) can be independently activated in a given cycle. An AP chip can support up to 48K transitions in each cycle. Thus it can efficiently execute massive Non-deterministic Finite Automata (NFA) that encapsulate hundreds to thousands of patterns. The technique we first analyze to perform FSM parallelization is just a logical inference. It says that we have to start by partitioning the input string into segments, and processing these segments concurrently. The problem with this approach is that starting states for each segment are unknown except the first segment (which starts from the FSMs designated start states). The starting states for a segment are essentially the ending states of the previous segment.The work [25] has solved this by executing the input segment for every state of the FSM by leveraging classic parallel prefix-sum [22]. This method is referred to as enumerative computation as it enumerates all possible start states. We refer to the sequence of states visited by each enumeration start state as the enumeration path. Once the first segment is finished, we know the correct start states of the second segment and can pick the results of enumerated paths belonging to the correct start states (Section 2.2, and Figure 2 discuss an example enumeration).

While enumerative approach is promising, there were several challenges to realize it in AP. In a conventional processor a SIMD thread can process enumeration paths and threads local variables keep track of the start state for each enumerated path. Tracking the start state of an enumerated path is important for combining the results of individual input segments as discussed above. In the AP, there is no notion of software threads or local variables which can keep track of start states of enumerated paths. A processing unit or half core simply accepts a stream of input symbols and does transitions via a routing matrix (custom interconnect) each cycle. Thus it can be challenging to execute concurrently and keep track of all enumeration paths. Another critical challenge to be solved is taming the enormous computational complexity of enumeration. Enumerations can be highly inefficient because in the worst case each state has to be enumerated. NFAs can have several thousands of states.

The architecture that [**?**]rovides, solves the above problems by leveraging some unique properties of NFAs and unique features of the AP. For in- stance, we utilize the connected components (disconnected subgraphs) in an NFA to merge enumeration paths and thereby take advantage of the massive parallelism of the AP. Furthermore, the range (or all reachable states) of an input symbol can be utilized to prune the enumeration paths. Another NFA property we leverage is based on parent states in an NFA. If the start states of enumeration paths have a common parent state, they can be merged. Similar to prior work [25, 29], we observe enumeration paths converge at run time and implement dynamic convergence checks in AP. To solve the start state tracking problem, we utilize AP flows. The flow abstraction also allows for near-zero overhead convergence checks. Our framework also discusses the details of combining the results from input segments, and hiding these processing overheads by leveraging the asymmetric finish times of input segments.

In summary this review offers the following analysis:

- Parallelization of non deterministic FSMs on the Automata Processor (AP).
- Explore the challenges in parallelizing FSMs in AP.
- Quick look to evaluation of the technique against important benchmarks.

II. BACKGROUND

In this section we provide a brief background on the Automata Processor and enumerative techniques for parallelizing FSM processing.

A. NFA and Automata Processor

A Non-deterministic Finite Automata (NFA) is formally described by a quintuple $\langle$ Q, $\sum$, δ , q_0 , F$\rangle$, where Q is a set of states,$\sum$ is the input symbol alphabet, q_0 is the set of start states and F is the set of reporting or accepting states. The transition function δ(Q,α) defines the set of states reached by Q on input symbol . The non-determinism is due to the fact that an NFA can have multiple states active at the same time and have multiple transitions on the same input symbol.

Conventional compute-centric architectures store the complete transition function as a lookup table in the cache/memory. Since a lookup is required for every active state on every input symbol, symbol processing is bottlenecked by the available memory bandwidth. This leads to performance degradation especially for large NFAs with many active states. With limited memory bandwidth, the number of state transitions that can be processed in parallel is also limited. Converting these NFAs to equivalent DFAs also cannot help improve performance since it leads to exponential growth in the number of states.

The memory-centric Automata Processor (AP) accelerates finite state automata processing by implementing NFA states and state transitions in memory. Each automata board fits in a DIMM slot and can be interfaced to a host CPU/FPGA using the DDR/PCIe interface. Figure 2 illustrates the automata processor architecture.

For processing in AP, the classic representation of NFAs is transformed to a compact ANML NFA representation [12] where each state has valid incoming transitions for only one input symbol. Thus each state in an ANML NFA can be labeled by one unique input symbol. ANML NFA computation entails processing a stream of input symbols one at a time. Initially, all the start states are active states. Each step has two phases. In the state match phase, we identify which of the active states have the same label as the current input symbol. In the state transition phase, we look up the transition table to determine the destination states for these matched states. These destination states would become the active states for the next step. In AP, the FSM states (called State-Transition Elements or STEs) are stored as columns in DRAM arrays (256 bits). Each STE is programmed to the one-hot encoding of the 8-bit input symbol (same as it is label) that it is required to match against. For example, for an STE to match the input

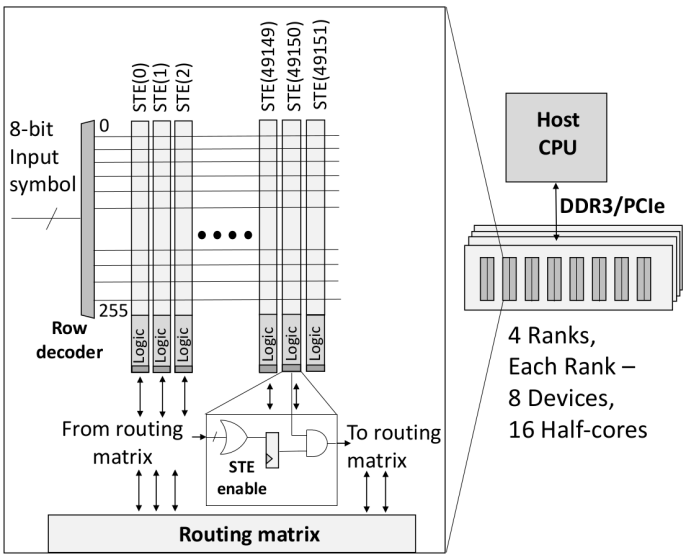


Fig. 2. An Automata proccesor overview, taken from []

symbol a, the bit position corresponding to the 97 th row must be set to 1.

Each cycle, the input symbol (ASCII alphabet) is broadcast to all DRAM arrays and serves as the row address. If an STE has a 1 bit set in the row, it means that the label of the state it has stored matches the input symbol. State match is then simply a DRAM row read operation, with the input symbol as the row address and the contents of the row determining the STEs that match against the input symbol. Thus, by broadcasting the input symbol to all DRAM arrays, it is possible to determine in parallel all the states which match with the current input symbol. State transitions between currently active states to next states is accomplished by a proprietary interconnect (routing matrix) which encodes the transition function. Reconfiguring this interconnect requires a costly recompilation step. Only the states which matched with the current input symbol and are active, undergo state transition.

We see how [**?**]nfasizes how difficult the FMS parallelizing is. Because of the dependencies between every consecutive state transitions.

[**?**]escribes the way of parallelization. To parallelize FSM traversal is by partitioning the input string into segments, and processing these segments concurrently. This is feasible because FSM computation can be expressed as a composition of transition functions [15]. Parallelization is possible because transition function composition is associative. Figure 3 shows an example of parallelizing the FSM with two input segments (I_1 and I_2) each with five symbols. The FSM shown detects the first word in every line. The transition table is shown on right. Both these segments can be executed in parallel to provide a speedup of 2 over sequential baseline. However, the starting states for each input segment are unknown except the first segment (which starts from initial start states). The starting states for a segment are essentially the ending states of the previous segment. These dependencies prevent concurrent execution among threads.This problem can be solved by leveraging classic parallel prefix-sum [22]. The basic idea is to execute the second segment for every state of the FSM. This method is referred to as an enumerative computation as it enumerates all possible start states [25]. In Figure 3 the start state of the first segment is known (S_0) which is the start state of the FSM. However, the start states of input segment I_2 are unknown. Figure 3 shows an example enumeration for the second input segment, I_2 . This example FSM has 3 states, so each segment (except the first) enumerates all 3 states. Once the first segment has finished, it can pick the correct or true paths from the enumerated paths of the second segment and discard false paths. Thus, final results can be obtained by combining the intermediate results of all input segments. The true path for I_2 in Figure 3 starts at S_1 , the remaining two paths are false paths. The final path of the FSM is highlighted.

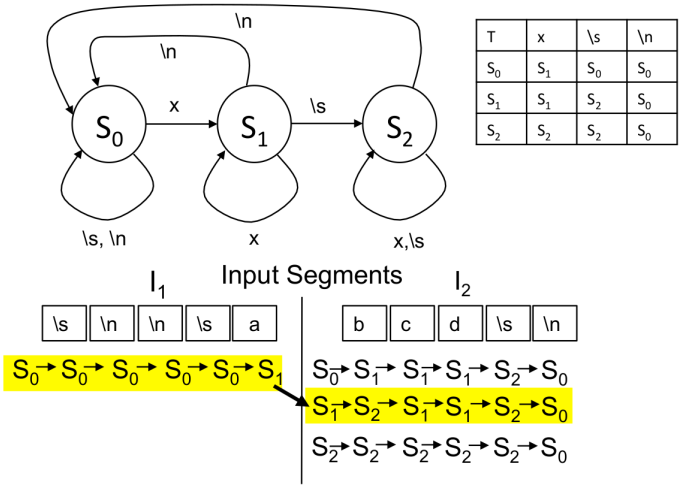


Fig. 3. An FSM example with enumeration, taken from []

III. Parallel Automata Processor

In this section we discuss the proposed framework and architecture [] needed for effective parallelization of NFAs on the Automata Processor(AP).

A. Range Guided INput Partitioning

Enumerating from all states of an FSM will lead to exponential computational complexity. Fortunately, many of these states are impossible start states for the particular input segment. The range of an input symbol is defined as the union of the set of all reachable states, considering transitions from all states in the FSM that have a transition defined for that symbol. During actual execution, the range of the last input symbol in a particular segment determines accurately the subset of start states for the next segment. Any states outside this range are impossible start states. The proposed parallelization framework [] partitions the input such that input segments end at frequently occurring symbols with small ranges to take advantage of minimum range symbols. The symbol chosen for an FSM is obtained by offline profiling. Frequently occurring symbols are chosen to ensure that the size of input segments are roughly equal.

B. Enumeration using Flows

Ideally, one can activate all start states and execute all enumerations simultaneously on one copy of the FSM. We know this is possible and correct, given that AP seamlessly implements any number of simultaneous transitions in a given cycle, and there is no limit on the number of start states that are activated. This seems like a perfect solution, except that we lose all information about enumeration paths. After processing the input segments we know what are the end states for all enumeration paths, but there is no way of knowing which path lead to which end states. Recall that after an input segment finishes execution, it will inform the next input segment which enumeration paths were the true paths and which paths were false paths. The next input segment then must only use the results and end states of true paths and discard false paths. In a conventional processor, enumeration paths are executed on SIMD threads and threads local variables keep track of the start state of each path. In the AP, however there is no notion of local variables or state tracking. The only way to implement state tracking is by propagating the start state via the routing matrix. Routing matrix currently just routes 1 bit per state element pair (which encodes transition between two state elements) and is already known to be a bottleneck in the system, both in determining the cycle time, and area complexity (occupies 30% of the chip) []. Augmenting the routing matrix with state information leads to exponential space complexity. Which is something that ,most of the authors we are reviewing, are not looking for.
[] solution is to leverage AP's flows to solve the above problem of tracking the start states of enumeration flows.
The state vector allows AP to context switch between two independent executions much like the register save/restore that allows tasks to context switch on traditional CPU architectures [3, 14]. The output match events also encapsulate a flow identifier. In our architecture each enumeration path is mapped to an independent flow and time division multiplexed on the same half-core. By association to a flow identifier, we can easily track the enumeration paths that belong to each flow. The host CPU keeps a flow table which tracks which start states (or enumeration paths) are mapped to which flows.

C. Merging Flows

The speedup which can be obtained from our parallelization techniques relies on two factors, the number of input segments executing in parallel and the time taken to complete each input segment. In general the speedup obtained is equal to number of input segments divided by slowdown experienced by the slowest input segment. Slowdown of an input segment is simply the time it takes when compared to the time it would have taken had we known the exact start states for that segment. By utilizing flows we have maximized the number of input segments, however time division multiplexing of flows also slows down processing of each input segment. Specifically the processing time of each input segment is proportional to number of flows for that segment. The range guided partitioning method significantly reduces the number of enumeration paths and hence the number of flows needed.

D. Composition of Input Partitions

Once the input segments finish, the final output results can be obtained by combining the results of true paths of each segment. The host CPU reads the final state vector from the AP and then constructs a Boolean array indicating which flow has results for true enumeration paths. This Boolean array is checked when reading the results out of the output buffer. Each output buffer entry has few bits indicating the flow identifier. Only results for the output buffer entries which match with true flows are reported.

E. Put it Together

This section describes our overall framework for parallelizing NFAs on the AP. Figure 7 brings together all the concepts discussed in this section to illustrate our overall framework. The parallelization framework consists of pre-processing steps and dynamic runtime steps. First, the range is computed for all input symbols and a frequently occurring symbol is chosen based on profiling (Section 3.1). This is followed by the merging the states in the range table into flows based on connected components and common parents. This step generate the contents for the State Vector Cache (SVC). The state vector cache is then loaded onto the AP chips. This pre-processing can be augmented to the compilation and configuration process for the AP. Following this, the input is partitioned at boundaries of the chosen range symbol. Each input segment starts getting processed in parallel on AP half-cores. Deactivation and convergence checks occur dynamically to invalidate redundant or unproductive flows (Sections 3.3.4 and 3.3.3). A segment can also receive a flow invalidation vector from the previous segment during its runtime. Once an input segment finishes, the composition of output reports happens in the CPU

IV. Results

We analyze the speedupss obtained by the proposed Parallel Automata Processor Architecture (PAP).

A. Overall Speedup

Figure 4 [] shows the speedups obtained by our proposed Parallel Automata Processor Architecture (PAP), when compared to the baseline AP architecture. We present speedups for both 1 AP rank (8 D480 devices) and 4 AP ranks (32 D480 devices in the current AP generation) and 1 MB and 10 MB input streams. We also exploit the parallelism offered by each of the half-cores in a D480 device when our FSMs can fit in a single half-core. Table 1 details the AP footprint and number of input segments created for each of our benchmark FSMs.

It can be seen from the figure that PAP outperforms the sequential AP baseline for most benchmarks. A noticeable trend is the larger performance gains with the 10 MB stream. This is because the larger stream provides opportunity for creating larger input segments. These larger input segments help in reducing in the number of active flows due to the deactivation and convergence properties of the FSMs and the associated flow switching overhead. Furthermore, large input segments also help amortize the cost of false path invalidation and input composition in the CPU.

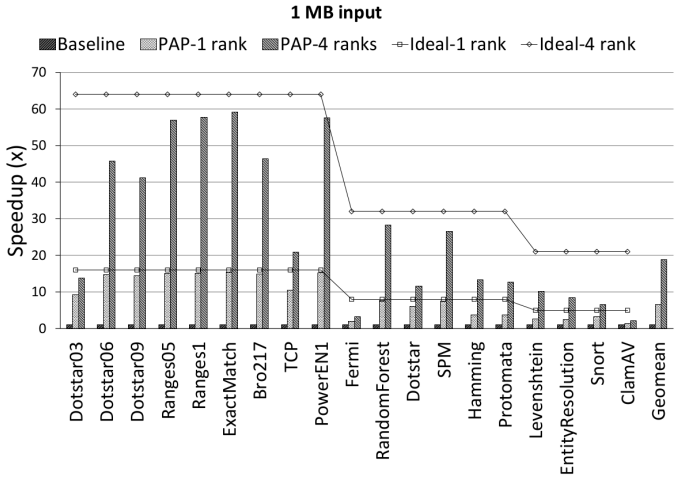


Fig. 4. Speedups obtained by the proposed architecture Parallel Automata Processor Architecture

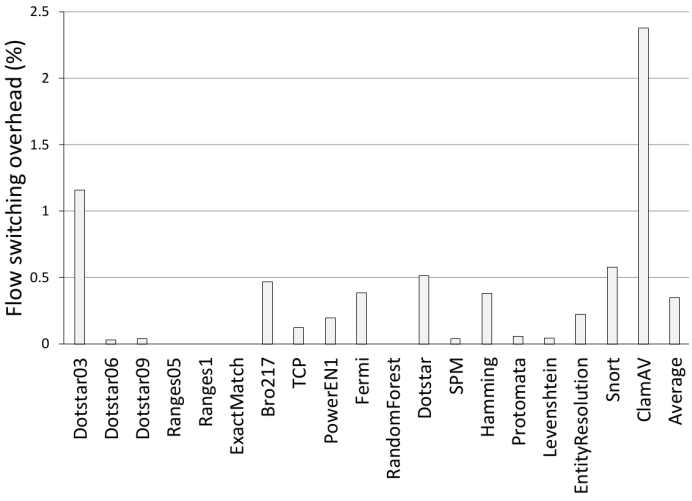


Fig. 5. Costs of flow switching.

B. Overheads

1) Flow Switching: It can be seen from Figure 5 that the overheads of context switching between flows are less than 2% for most benchmarks. As discussed before, since the number of active flows greatly reduces as input symbols are processed, the corresponding convergence and deactivation checking overheads also reduce.

2) False Path Decoding: Figure 6 illustrates the overheads of decoding false paths at the host CPU after an input segment finishes and sending a flow invalidation vector (FIV) to the next segment. On an average most benchmarks see around 2000 symbol cycles overhead. Fortunately, this cost is largely amortized because of two reasons: (1) it can be overlapped with symbol processing in subsequent segments, (2) these invalidations are infrequent since several flows have either already converged or have been deactivated and do not require this invalidation.

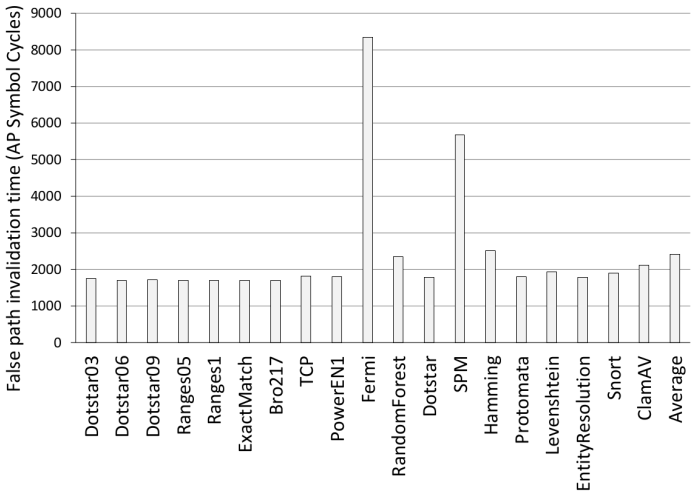


Fig. 6. Costs of decoding false paths at the end of input seg- ment.

V. Conclusion

We saw how the sequential NFA execution bottleneck on the Micron Automata Procesor (AP). We identify two main challenges to applying enumerative NFA parallelization techniques on the AP: (1) high state-tracking overhead for input composition (2) huge computational complexity for enumerating parallel paths on large NFAs. Using the AP flow abstraction and properties of FSMs like small input symbol transition range, connected components and common parents we amortize the overhead of state-tracking and realize a time-mutliplexed execution of enumerated paths. To tackle the computational complexity, we leverage properties of the FSMs like path convergence to dynamically reduce the number of executed enumerated flows.

We can make a conclusion that most of the times , special-oriented hardware outperforms general-oriented if it is well leveraged. Also, we saw that special-oriented hardware with a special management outperforms the same common special-oriented hardware. This conclusion may look trivial , but it's just a reminder of how *Subranamiyan* [] implemented the FSM parallelization in the AP with outstanding results.