Critical Analysis and Evaluation of Different Automata Processing Accelerators on Large-scale Datasets

---

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

MARZIYEH NOURIAN

Dr. Michela Becchi, Thesis Supervisor

December 2016

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

CRITICAL ANALYSIS AND EVALUATION OF DIFFERENT AUTOMATA PROCESSING ACCELERATORS ON LARGE-SCALE DATASETS

presented by Marziyeh Nourian,

a candidate for the degree of master of science,

and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Michela Becchi

Professor Prasad Calyam

Professor William Harrison

ACKNOWLEDGEMENTS

First, and foremost, I would like to express my sincere gratitude to my Advisor Dr. Michela Becchi for her continuous support and guidance through my research studies and professional academic experience. I have been extremely lucky to be able to work in a very enthusiastic and professional environment under her lead.

I would like to thank fellow members of NPS Lab for all their help and of course the friendship.

I thank my beloved parents and my brother who has been a source of encouragement and inspiration to me. I also appreciate the heartwarming support that my aunt offered me.

I would like thank my friends and peers for being there for me through the entire program.

This work has been supported through NSF awards CNS-1319748 and CCF-1421765 and equipment donation from Nvidia Corporation.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CRITICAL ANALYSIS AND EVALUATION OF DIFFERENT AUTOMATA PROCESSING ACCELERATORS ON LARGE-SCALE DATASETS

Marziyeh Nourian

Dr. Michela Becchi, Thesis Supervisor

ABSTRACT

Many established and emerging applications perform at their core some form of pattern matching, a computation that maps naturally onto finite automata abstractions. As a consequence, in recent years there has been a substantial amount of work on high-speed automata processing, which has led to a number of implementations targeting a variety of parallel platforms: from multicore CPUs, to GPUs, to FPGAs, to ASICs, to Network Processors. More recently, Micron has announced its Automata Processor (AP), a DRAM-based accelerator of non-deterministic finite automata (NFAs). Despite the abundance of work in this domain, the advantages and disadvantages of different automata processing accelerators and the innovation space in this area are still unclear.

In this work we target this problem. In particular, to allow an apples-to-apples comparison, we focus on NFA acceleration on three platforms: GPUs, FPGAs and Micron's AP. We discuss the automata optimizations that are applicable to all three platforms. We perform an evaluation on large-scale datasets: to this end, we propose an NFA partitioning algorithm that minimizes the number of state replications required to maintain functional equivalence with an unpartitioned NFA, and we evaluate the scalability of each implementation to both large NFAs and large numbers of input streams. Our experimental evaluation covers resource utilization, throughput, and preprocessing cost

Chapter1 Introduction

Many established and emerging applications perform at their core some form of pattern matching, a computation that maps naturally onto finite automata abstractions. In biology, for example, several genomics tasks, such as motif discovery, orthology inference, shotgun and de novo assembly, involve a variety of string matching operations on genomics data. In turn, advances in DNA sequencing technology have led to increasingly large volumes of data available for these applications, resulting in a significant increase in their computational requirements. In the networking domain, several applications such as network intrusion detection, content-based routing, and application-level filtering require inspecting network packets for potentially large sets of predefined patterns, and they typically must perform this operation at the rate of packet arrival on the router interface.

Given the number and relevance of applications requiring efficient pattern matching, there has been a substantial amount of work on high-speed automata processing, and this work has originated from different communities: from the networking, to the reconfigurable computing and computer architecture, to the parallel computing community. These efforts have led to a number of algorithmic [1-9] and architectural solutions targeting different parallel platforms: from CPUs, to GPUs [10-12], to FPGAs [13-16], to ASICs [17-19], to Network Processors [20]. More recently, Micron has announced their Automata Processor [21], a DRAM-based accelerator of non-deterministic finite automata that has been showcased on a variety of applications: motif discovery in biological sequences [22], association rule mining [23], brill tagging [24], high-speed regular expression matching for network intrusion detection [25], graph processing [26], and sequential pattern mining [27].

Despite this abundance of work on high-speed automata processing, there is still lack of clarity as to how existing software and hardware solutions are related to and compare with each other. There are a couple of reasons for this. First, existing solutions are based on different automata models: either non-deterministic or deterministic finite automata (NFAs and DFAs, respectively). While functionally equivalent, NFAs and DFAs have practical differences – in terms of resource requirements and traversal behavior – that are strongly dependent on the characteristics of the underlying pattern set. While there has been substantial body of work proposing automata designs that trade off the advantages and disadvantages of NFAs and DFAs [1-9], no automata model is preferable on all datasets. This makes it hard to provide a fair comparison between automata processors relying on different automata models. Second, some automata processing architectures are designed to optimize the peak performance of a single input stream, while others offer better support for handling stream-level concurrency. Third, most of the existing automata processing solutions have been designed to optimize automata traversal, but different architectures require different amount of preprocessing (in the form of automata compilation, loading to memory, place&route, etc.), which may be significant for some applications.

To target these problems and provide an apples-to-apples comparison, we select automata accelerator designs that rely on the same automata model: NFAs. Since NFAs do not suffer from state explosion, their use allows us to perform an evaluation on large-scale datasets without posing any restrictions on the kind of patterns supported. Specifically, we compare GPU- and FPGA-based NFA accelerator designs with Micron's AP. Micron's AP extends NFAs' functionality with counters and boolean elements. To ensure functional equivalence and the same degree of programmability across the considered platforms, we extend

existing GPU-based designs to support these features and take the extended FPGA design from [33]. In addition, we adopt the same programming interface for all platforms: namely, Micron's Automata Network Markup Language (ANML). Different platforms offer different automata density – to take this into account, we perform an analysis on non-trivial dataset sizes, which require partitioning large NFAs across multiple devices. Besides considering peak performance on a single input stream, we evaluate the scalability of the considered automata processor designs to multiple concurrent inputs. Finally, we evaluate the costs of the different preprocessing steps required by the considered architectures, and study how the size of the automaton and the density of its transitions affect some of the preprocessing stages (for example, place&route on Micron's AP and FPGA).

1.1 Contributions

To summarize, we make the following contributions:

- We extend GPU-based automata processing designs to support Micron's AP counters and boolean elements while the extended FPGA design is taken from [33]. We debug the extended FPGA design using real-world datasets such as Fast-SNAP (a network security application) and PROTOMATA (a bioinformatics application) described in [25]. Additionally, we propose a compiler toolchain to automatically deploy extended NFAs (in ANML form) onto these three platforms.
- We propose an NFA partitioning scheme aimed at minimizing the amount of state replication required to handle large NFAs while preserving functional equivalence with a single unpartitioned NFA.
- For GPU deployment, we explore different state layouts and kernels suited to NFAs with varying characteristics.
- We perform an apples-to-apples comparison between Micron's AP, GPU- and FPGA-based NFA accelerator designs on large-scale datasets. Micron's AP evaluations are the result of collaboration with University of Virginia Tech by partitioning the datasets according to AP resource and routing limitations. Our evaluation covers resource utilization, throughput and preprocessing costs for real-world NFAs used in networking and bioinformatics applications, as well as synthetic datasets covering regular expressions datasets with various characteristics.

Chapter 2 Background and Related Work

2.1 Background on Automata Processing

Regular expression matching has traditionally been implemented by representing the pattern-set through finite automata (FA) [28]. The matching operation is equivalent to a FA traversal guided by the content of the input stream. Worst-case performance guarantees can be offered by bounding the amount of processing performed per input character. However, techniques to keep per-character processing low involve increasing the size of the finite automaton, the basic data structure in the regular expression matching engine. As the size of pattern-sets and the expressiveness of individual patterns increase, limiting the size of the automaton to fit on reasonably provisioned hardware platforms becomes challenging. Thus, the exploration space is characterized by a trade-off between the size of the automaton and the worst-case bound on the amount of per character processing.

NFAs and DFAs are at the two extremes in this exploration space. NFAs have a limited size but can require expensive per-character processing, whereas DFAs offer limited per-character processing at the cost of a possibly large automaton. In Figure 1 we show the NFA and DFA accepting three simple patterns (*a*+*bc*, *bcd*+ and *cde*). In the figure, states active after processing text *aabc* are colored gray. In the NFA, the number of states and transitions is limited by the number of symbols in the pattern-set. In the DFA, every state presents one transition for each character in the alphabet ($\sum$). Each DFA state corresponds to a set of NFA states that can be simultaneously active [28]; therefore, the number of states in a DFA equivalent to an *N*-state NFA can potentially be 2^N. In practice, previous work [2, 5, 29] has shown that this so-called "state explosion" happens only in the presence of complex patterns (typically those containing bounded and unbounded repetitions of large

character sets). Since each DFA state corresponds to a set of simultaneously active NFA states, DFAs ensure minimal per-character processing (only one state transition is taken for each input character).

From an implementation perspective, existing regular expression matching engines can be classified into two categories: memory-based [1-12, 17, 19], and logic-based [13-16]. Within the former, the FA is stored in memory; within the latter, it is stored in combinational and sequential logic. Memory-based implementations can be deployed on various platforms (GPUs, network processors, ASICs, FPGAs); logic-based implementations typically target FPGAs. In a memory-based implementation, design goals are the minimization of the memory size needed to store the automaton and of the memory bandwidth needed to operate it. Similarly, in a logic-based implementation the design aims at minimizing the logic utilization while allowing fast operation (that is, a high clock frequency). Existing proposals targeting DFA-based, memory-centric solutions have focused on designing compression mechanisms to reduce the DFA memory footprint and novel automata to alleviate the state explosion problem [1-9]. Despite the complexity of their design, memory-centric solutions have three advantages: fast reconfigurability, low power consumption, and scalability in the number of input streams. On the other hand, logic-centric solutions allow for easily achieving peak worst-case performance on a single input stream, at the expense of lack of scalability in the number of concurrent inputs.

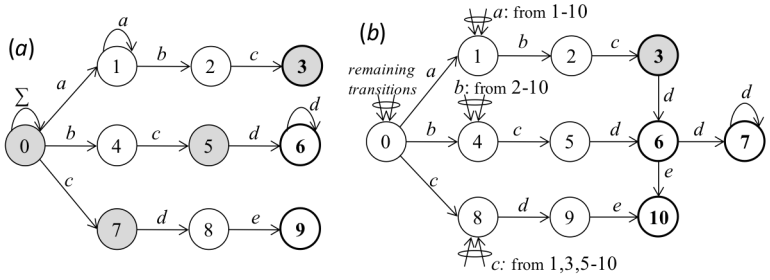


Figure 1: (a) NFA and (b) DFA accepting regular expressions a+bc, bcd+ and cde. Accepting states are bold. States active after processing text aabc are colored gray.

2.2 Micron's Automata Processor Overview

Micron's Automata Processor [21] is a DRAM-based, reconfigurable accelerator that simulates NFA traversal at high speed. The AP includes three kinds of programmable elements stored in SDRAM: *State Transition Elements* (STE), *Counter Elements* (CE) and *Boolean Elements* (BE), which implement states/transitions, counters and logical operators between states, respectively. Each STE includes a 256-bit mask (one bit per ASCII symbol), and symbols triggering state transitions are associated to states (and encoded into STEs) rather than to transitions. Transitions between states are then implemented through a routing matrix consisting of programmable switches, buffers, routing lines, and cross-point connections. The routing capacity is limited by tradeoffs between clock rate, propagation delays and power consumption, and these constraints influence place&route of automata onto the AP hardware. Micron's current generation of AP board (AP-D480) includes 32 *chips* organized into four *ranks* (8 chips per rank). Each AP chip consists of two *half-core*s. There are no routes either between half-cores or inter-chips, which implies that NFA transitions across half-cores and chips are not possible. Programmable elements are organized in *blocks*: each block consists of 16 *rows*, where a row includes eight *groups* of two STEs and one special purpose element (CE or BE). Each chip contains a total of 49,152 STEs, 768 CE and 2,304 BE, organized in 192 blocks and equally residing in both half-cores. Current boards allow up to 6,144 elements per chip to be set as report elements. AP automata can be described in ANML (an XML-based language), and recently proposed high-level programming languages for the AP are mapped and compiled into ANML [30]. Micron's SDK includes a toolchain that parses ANML designs, compiles them into internal objects consisting of subgraphs, places and routes these subgraphs onto the AP hardware,

and finally generates a binary image that can be used to program the AP memory and routing matrix. Once the AP has been programmed, it will be able to simulate the NFA traversal. AP chips can be grouped into logical cores of 2, 4 or 8, each processing a stream of 8-bit input characters [25]. The AP nominally operates at a 133MHz frequency, and, in absence of matches, it processes one input character per clock cycle from all input streams. Once matches occur, AP generates reporting events in vector format and stores them in an output-buffer; reporting matches to the host system requires from 91 to 291 clock cycles.

Chapter 3 Compiler Toolchain

3.1 Overall Design

Figure 2 shows the toolchain designed to deploy ANML specifications on GPU, FPGA and Micron's AP. In the figure, grey boxes represent our software components. The last two modules leading to FPGA and AP are Xilinx and Micron's software development kits used for the final synthesis/compilation, map, and place&route on these two devices.

The input to the toolchain is an ANML file that contains one or more automata networks (each including one or more NFAs). We don't impose any constraints on these networks: in other words, they don't need to be designed to fit a particular device or optimized for it. Once parsed, these networks are stored in our toolchain using an internal representation for later processing and optimization. We distinguish two categories of optimizations: *automata-specific* and *platform-specific*. Since the GPU, FPGA and AP are used as NFA traversal accelerators, optimizations to the automaton apply to all platforms. Becchi and Crowley [14] described several NFA optimizations (*state reduction*, *alphabet compression* and *software striding*) and put them to practice on FPGA; these optimizations apply to GPUs and AP as well. Automata-specific optimizations can be selectively enabled and disabled. Platform-specific optimizations are related to the way the NFA is encoded for the particular target device; these optimizations include compact and efficient memory encodings, logic utilizations, and striding mechanisms that are specific to a particular hardware platform. Since the internals of the operation of the AP hardware and its software stack (including the compilation, map and place&route processes) are proprietary, AP-specific optimizations are deferred to the AP SDK tools (last phase of the toolchain). The *partitioning* step, that takes a potentially large network and breaks it into multiple NFA

partitions so that each of them can fit the target hardware, is performed after the automata-specific optimization step. This allows partitioning to be done on an already optimized NFA. Our partitioning algorithm is platform-independent, but its configuration depends on the target platform. The *code and configuration generation step* produces the files required for the final deployment of the automata network on the hardware. For GPUs, all is needed is a configuration file that includes the information necessary to load the NFA partitions into memory, and a header file with the definition of the boolean connectors in the ANML specification. FPGAs are configured through a Verilog file describing the NFA network and its interface. The AP is configured through an ANML file; this output file differs from the input file in that it contains a partitioned and optimized automata network.

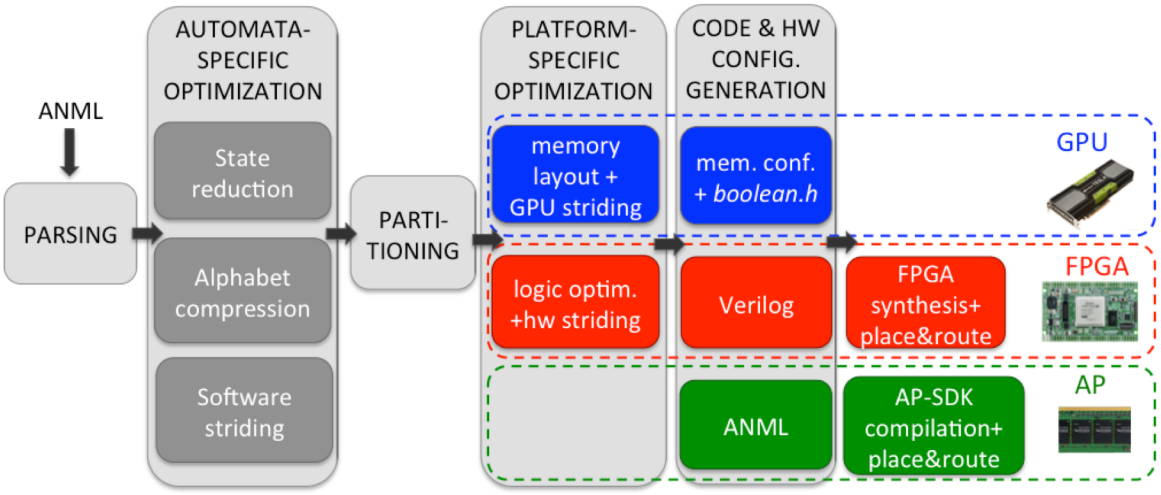


Figure 2: Our toolchain

3.2 GPU Implementation

We reuse and extend iNFAnt [18], an NFA-traversal engine for GPUs. iNFAnt stores the NFA in device memory, and encodes the transition table as set of (source, destination)

pairs indexed by the input character. In order to allow efficient execution, iNFAnt stores the set of active states in shared memory in bit-vector form, along with a persistent vector indicating states that, once they become active, will never be deactivated. For each input character, iNFAnt retrieves from memory all the transitions on that symbol, and, if their source state is active, the engine updates the active state vector with the destination state information. In iNFAnt, each thread-block is assigned an input stream, and threads within a block process the state transitions and update the state vector cooperatively.

We extend iNFAnt with the following functionalities:

Support for multiple NFA partitions – We map each NFA partition to a thread-block, allowing multiple blocks to process the same input stream, each on a different partition. The transition lists corresponding to different NFA partitions are laid out sequentially, and an indexing array maps each partition to the proper set of thread-blocks, each operating on a different input stream.

Traversal kernels based on compressed sparse row (CSR) memory layout – We consider an alternative memory layout where transitions represented as (input symbol, destination) pairs are indexed by the source state. For each input symbol, this layout allows processing only the transitions that originate from active states. We store the identifiers of the active states in a queue in global memory. We consider two variants of this kernel: *CSR-state* and *CSR-tx*, the former mapping active states to threads, and the latter mapping outgoing transitions from active states to threads.

Support for counters and boolean elements – We associate a special state to each counter and boolean element, and store these special states at the end of the state vector. The activation of special states triggers code implementing the operation of the particular

counter or boolean element. Boolean operators are also associated combinational code that is stored in an automatically generated header file.

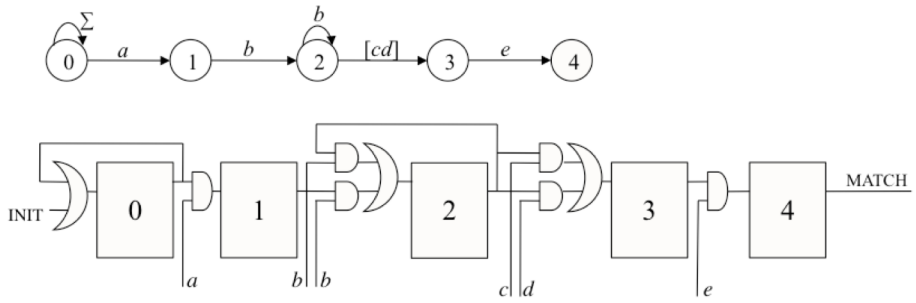


Figure 3: NFA accepting regular expressions ab+[cd]e and corresponding one-hot encoding representation

3.3 FPGA Implementation

On FPGA, NFA processing can be realized in two ways: either by implementing a traversal engine that accesses the NFA stored in memory, or by directly encoding the NFA in logic. Most logic-based NFA implementations are based on the one-hot encoding scheme [13], in which states are represented as flip-flops while transitions are implemented by and-ing and or-ing the outputs of the flip-flops with the decoded input character. For example, Figure 3 shows the one-hot encoding representation of the NFA accepting regular expression $ab^{+}[cd]e$. The main advantage of this scheme is that it limits the processing time to one clock cycle per input character independent of the number of states that are active (this property is shared by Micron's AP). On the other hand, this implementation suffers from two limitations: first, updating the NFA requires reprogramming the device; second, multiple input stream support requires logic replication. The pros and cons of a memory-based FPGA design are comparable to those of a GPU solution: easy support for multiple input streams at the cost of irregular and

unpredictable memory access patterns, leading to dataset dependent performance. In this work we use the optimized logic-based implementation that is described in [14] and the extended form of implementation as described in [33].

3.4 Automata-specific Optimizations

Our toolchain includes three automata-specific optimizations that is described in [14]; namely, state reduction, alphabet compression and software striding. Here, we briefly mention their effect on the considered platforms.

State reduction (which merges duplicate NFA paths) reduces the memory requirements on GPU and AP, and the logic requirements on FPGA. In addition, it reduces the number of states that can be active in parallel, which for GPU is beneficial to the throughput.

Alphabet compression (which consolidates the size of the alphabet according to the symbols appearing on the NFA transitions) reduces the wiring and LUT utilization on FPGA. However, due to the fact that the AP stores a 256-bit mask in each STE, this optimization does not benefit APs unless combined with software striding.

Software striding (which allows processing multiple characters in one step) can be beneficial on all platforms if combined with alphabet reduction. This technique is applicable to the AP only if the alphabet generated by combining alphabet reduction and software striding does not exceed 256 symbols. GPUs and FPGAs offer also platform-specific striding mechanisms [6, 10, 15].

3.5 Partitioning Criteria

An NFA must be partitioned if it exceeds the resources available on a particular device. Here, we indicate the platform-specific partitioning criteria we use. In Section 4, we describe our proposed partitioning algorithm.

GPU: GPU partitioning is required if the shared or global memory capacity is exceeded, or if the state identifier space is exhausted. In this work, we use 16-bit state identifiers (including both standard and special states), leading to a maximum of 65536 states per NFA partition. This constraint is more restrictive than those on the global and shared memory capacity (and, due to thread-block concurrency, is not a limiting factor on performance – see Section 5).

FPGA: The logic design used stores states in flip-flops and transitions in LUTs. We experimentally found flip-flops to be the bottleneck resource (see Section 5). Therefore, we perform NFA partitioning when the number of NFA states exceeds that of available flip-flops.

AP – The AP does not allow transitions across half-cores, and has a limited number of STEs, Counter Elements and Boolean Elements per half-core (see Section 2.2). Thus, the AP NFA partitioning criterion is based on these constraints.

Chapter 4 NFA Partitioning Algorithm

In this section, we describe our NFA partitioning algorithm. For the sake of simplicity, we discuss the algorithm on traditional NFAs: its extension to counters and boolean elements is straightforward. In order to preserve functional equivalence, NFA partitioning requires state replication. For example, let us assume to break the NFA of Figure 4(a) into two partitions to be deployed and operated on two devices: one partition containing states from 0 to 16, and the other containing states from 17 to 24. In order to maintain functional equivalence with the original NFA, the entry state 0, which is shared by the patterns matched in the both partitions, must be replicated into the second partition. In general, very large NFAs may require replication of sets of states shared by several patterns.

The goal of our partitioning algorithm is to split the NFA into a small number of balanced partitions, while minimizing the required state replications. In particular, given a threshold N_{max} on the number of states that can be accommodated on a particular device or hardware component, the algorithm must split the NFA into as few as possible partitions, each with size not exceeding N_{max}. Balanced partitions allow load balancing within (for GPUs and Micron's AP) and across (for FPGAs) devices, which ultimately has a positive effect on throughput.

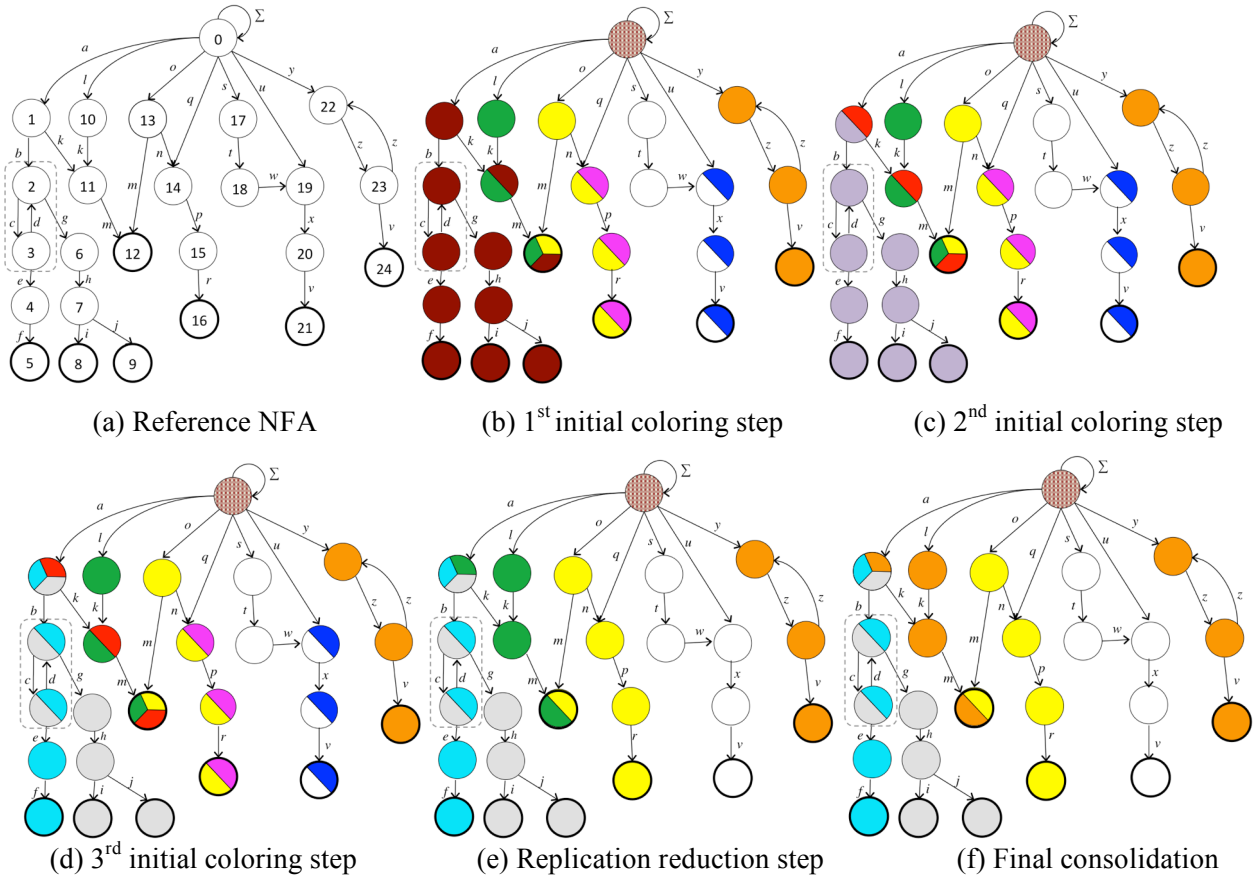
We propose a coloring algorithm that colors the NFA so that each color represents a partition, and states assigned multiple colors are shared across partitions and must be replicated in each of these partitions. In order to meet the requirements above, the algorithm must limit the number of colors and of states with multiple colors, while allowing each color to appear in up to N_{max} states. In the following, we call "color size" the number of states assigned a particular color.

Our algorithm operates in two phases: *initial coloring* and *color consolidation*. In the initial coloring phase, the NFA is traversed from the entry state and recursively colored until the size of each color doesn't exceed N_{max}. The color consolidation phase consolidates multiple colors into one while keeping their size below the given threshold.

We note that sets of states connected by cyclic transitions (e.g., states 2 and 3 in Figure 4(a)) cannot be separated into multiple partitions. We recall that, for partitions to be independent, inter-partition activations (that is, cross-partition transitions) must be avoided. As a consequence, a state belonging to multiple partitions must be replicated along with all the states connected to it in a cyclic fashion. Thus, we group states that are cyclically interconnected into "super-states", and we handle all the states in a super-state together. For example, states 2 and 3 of Figure 4(a) form super-state {2, 3} are handled as a single state. In order to operate, the algorithm requires super-states to include fewer than N_{max} states. If this is not the case, the NFA cannot be split into independent partitions. In the presence of dependent partitions, multiple NFA traversals are required to handle inter-partition activations. Fortunately, NFAs originated from regular expressions datasets tend to have only few super-states of small size. This is because backward-directed transitions in NFAs originate from sub-pattern repetitions within regular expressions (for example, sub-pattern $(dc)^*$ in Figure 4(a), where string *dc* can be repeated zero or more times). Sub-pattern repetitions are not frequent in real-world datasets, and are rarely shared by a large number of patterns.

We now detail the operation of the two phases of the algorithm, and illustrate them in Figure 4. In the example, we assume that the threshold N_{max} is equal to 8.

Initial coloring – The initial coloring procedure starts by assigning distinct colors to the states connected to the entry state (or to the super-state to which it belongs). This is illustrated in Figure 4(b), where the children of the entry state 0 are colored brown, green, yellow, pink, white, blue and orange. The colors are propagated to all the connected states following the transitions. The entry state is then assigned all the colors of its children. As can be seen, this leads to some states (states 11-12, 14-16, 19-21, beside state 0) be assigned multiple colors. This operation must be repeated recursively on all generated NFA partitions until their size does not exceed the threshold N_{max}. As can be seen, after the first coloring step the brown color has size 12 (states 0-9 and 11-12). Therefore, the coloring procedure is repeated starting from state 1. This causes color brown to be split into colors



(a) Reference NFA (b) 1st initial coloring step (c) 2nd initial coloring step

(d) 3rd initial coloring step (e) Replication reduction step (f) Final consolidation

Figure 4: Example of application of our coloring scheme (N_{max} =8).

red and violet, which are again propagated down to the terminal states of the NFA (Figure 4(c)). Since color violet has size 10 (including state 0), the algorithm invokes one additional recursive step on super-state {2,3}, causing color violet to be split into colors cyan and grey (Figure 4(d)). Since the largest color (grey) has now size 8 (equal to N_{max}), the initial coloring phase is terminated.

Color consolidation –While respecting the constraint on the maximum partition size, the partitioning generated by the initial coloring step has two limitations: it includes small and unbalanced partitions, and it leads to significant state replication. In the example, states 2, 3, 11, 14-16, 19-21 must be replicated once, states 1 and 12 must be replicated twice, and state 0 must be replicated 8 times. The coloring consolidation phase aims to combine different colors into one so as to increase the partition size, decrease the number of partitions and the number of state replications required, and achieve more balanced partitions. This phase is broken down into two steps: *replication reduction* and *final consolidation*. The first step aims to reduce the number of state replications required by merging colors. To determine which colors to consolidate, we sort pairs of colors in descending order according to the number of state replications that their consolidation would save. In the example, cyan/grey, yellow/pink and white/blue would save 3 state replications, green/red would save 2, and red/yellow, green/yellow, red/cyan and red/grey would save only 1. We then consider all pair-wise consolidation opportunities in order, and merge the two colors only if their merging doesn't violate the partition size constraint. In the example, we consolidate yellow+pink into yellow, white+blue into white, and green+red into green. Figure 4(e) shows the result of the replication reduction step. In the final consolidation step, we look for opportunities to consolidate colors according to their

size. To this end, we first sort the colors in descending order by size, and then traverse the list and consolidate each color with the next color in the list that doesn't lead to violating the partition size constraint (if such color exists). In the example, colors orange and green are consolidated into orange. Figure 4(f) shows the final coloring, which leads to 5 partitions: two of size 8 (grey and orange) and three of size 6 (cyan, yellow and white).

Chapter 5 Experimental Evaluation

5.1 hardware Platform

We conducted almost all our experiments on a machine equipped with a dual 6-core Intel Xeon CPU @ 2.66GHz and 64GB of memory, running CentOS 6.4. Since some of our AP syntheses run out-of-memory on that machine, for our AP experiments we used a server with similar hardware settings but equipped with 256GB memory. For our GPU experiments we used an Nvidia Titan X GPU (Maxwell architecture), equipped with 12GB of global memory and 24 streaming multiprocessors (SMs), each including 128 cores and 96KB of shared memory. We used CUDA 7.0. For our FPGA experiments we used a Xilinx XC5VLX220 device (Virtex-5 family), which includes 34,560 slices (for a total of 138,240 flip-flops and LUTs). We used the Xilinx ISE Design suite v10.1 to perform synthesis, mapping and place&route of our HDL designs. This device was chosen because it is in the same price range (~$3,000) as our GPU. Since Micron's AP is not yet on the market, we don't have pricing information for it.

5.2 Datasets

We selected datasets that allow us to compare the three platforms on different application domains and on NFAs with varying characteristics in terms of number of states and transitions, alphabet size, connectivity and depth. To this end, we used three types on datasets: *small NIDS*, *bioinformatics*, and *synthetic*. Recently proposed benchmark suites for automata processing [31] are not meant for large scale analysis (they include NFAs with up to about 100k states). Table 1 (columns 3-5) summarizes the characteristics of the NFAs for the considered datasets.

Table 1: Dataset characteristics and traversal information. In the traversal information, ranges correspond to traces with p_{forw}=0.5 and p_{forw}=0.9.

Type	*Name*	*NFA Characteristics*			*# Partitions*			*Traversal Information*			
		# states	*# trans.*	*# ANML states*	*GPU*	*FPGA*	*AP*	*Avg. active set*	*Max. active set*	*# Matches*	*Inputs w/ matches %*
Small NIDS	l7-filter	2794	124k	3844	1	1	1	3.0-4.8	5-9	4-10	0-0
	snort534	9514	79k	10k	1	2	1	1.7-22.8	10-37	9-533	0.01-0.2
Bioinformatics	10gene_8k	33k	100k	55k	1	1	4	371.2	617	12286	99.9
	10gene_12k	86k	258k	243k	2	1	16	378.8	713	7443	14.2
	10gene_16k	138k	415k	230k	3	2	14	378.0	713	493	0.1
	10gene_20k	190k	570k	317k	4	3	20	377.4	713	0	0.0
	100gene_8k	137k	413k	229k	3	3	16	670.8	826	81476	100
	100gene_12k	621k	1863k	1035k	14	10	80	742.3	1382	71140	68.6
	100gene_16k	1124 k	3372k	1873k	28	17	149	741.6	1387	4997	1.04
	100gene_20k	1619 k	4858k	2699k	42	25	213	740.1	1386	25	0.01
Synt.	deep-64char	800k	1801k	800k	14	9	76	2.1-2.5	7-7	0-0	0-0
	deep-256char	800k	4855k	800.3k	13	9	72	6.3-6.7	11-11	0-0	0-0
	shallow-64char	800k	1801k	800.1k	15	9	74	3.1-3.5	7-8	2-1904	0-2.9
	shallow-256char	800k	4855k	800.1k	13	9	78	6.1-6.7	12-14	1-2106	0-3.21

Small NIDS (Snort538 and l7-filter) are small network intrusion detection datasets that include 538 and 116 regular expressions, respectively, and have been used in previous work [10]. While small in terms of number of states, l7-filter is dense in transitions, leading to interesting results.

Bioinformatics datasets (*n*gene_k*k*) consist of a set of Hamming distance automata used to address a motif-finding problem [32]. The problem requires identifying all the substrings of length *k* that appear on multiple genes within hamming distance *d*, and can be found in a region of the gene of length *l*. Due to space limitation, here we show only the results for *n* genes from a yeast genome of about 5000 genes, with *n*={10, 100}, *k*={8, 12, 16, 20}, *l*=500, *d*=2 and a 4-symbol alphabet (A, C, G, T). An example of Hamming distance NFA is shown in Figure 5. Each such NFA has $(k+1)(d+1)-d(d+1)/2$ states, and each gene region of length *l* leads to $(l-k+1)$ of these NFAs. The NFAs in Table 1 (used on all three

platforms) have been state-reduced. However, previous work [22, 30] has shown that, on the AP, preprocessing time can be significantly reduced if NFAs with known structure are precompiled. Thus, on the AP we also use a non state-reduced variant of these bioinformatics datasets (see Table 5), leading to networks of $n(l\text{-}k+1)$ small NFAs (each with $(2d+1)k\text{-}d^2$ STEs) of fixed topology.

Synthetic automata exhibit the structure of NFAs accepting sets of regular expressions with shared prefixes: large state outdegrees in the proximity of the entry state, and low state outdegrees as we move deeper in the NFA. Our synthetic NFAs have configurable number of states, alphabet size, entry state outdegree, outdegree decrease factor γ (the outdegree decreases as γ^{depth}), and frequency of wildcards, character sets and their repetitions. We set these parameters so as to generate four 800k-state NFAs with two alphabet sizes (64 and 256), and two structures (*deep* and *shallow,* about 180- and 16-level deep, respectively).

Whenever required, we partition these NFAs with the algorithm described in Section 4. This leads to the number of partitions shown in Table 1 (columns 6-8). In order to simulate the NFA traversal, we use two kinds of input streams. For bioinformatics datasets, we generate traces of length 500,000 (for 1,000 genes) by randomly selecting symbols from the {A, C, G, T} alphabet. For NIDS and synthetic datasets, we generate 256k character

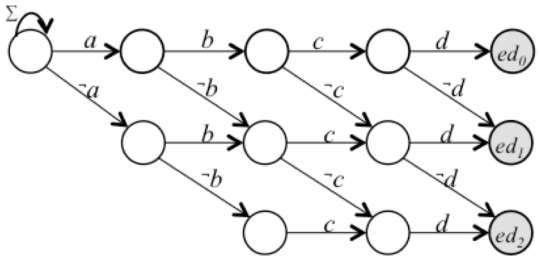


Figure 5:Hamming distance NFA for string abcd and edit distance ≤ 2.
Accepting state ed¬x (grey) recognize patterns at edit distance x.

traces through our trace generator [33], setting the probability to move deeper in the NFA (p_{forw}) to 0.5 and 0.9. The traversal characteristics (average and maximum number of active states per input character, and number and frequency of matches) are reported in Table 1 (columns 9-12).

Table 2: Resource utilization for GPU and FPGA (ranges correspond to the minimum and maximum values across partitions)

Type	*Name*	*GPU Mem. Utilization*			*FPGA % utilization*		
		shared (KB)	*global (MB)*		*FF*	*LUT*	*Slice*
			iNFAnt	*CSR*			
Small NIDS	l7-filter	0.6	0.4	0.5-0.6	1-1	1-1	2-3
	snort534	2.3	0.3	0.4-0.4	6	1	22
Bioinformatics	10gene_8k	8.1	0.4	7.6-47.5	24	17	38
	10gene_12k	13.2	1	13.1-78.1	62	43	90
	10gene_16k	12.3	1.6	13.3-74	36-63	25-43	57-87
	10gene_20k	14.8	2.2	16.6-90.3	68-69	46-47	96-98
	100gene_8k	14.9	1.6	15.4-87.8	44-55	34-42	63-87
	100gene_12k	14.8	7.1	27-99.8	37-68	26-41	52-98
	100gene_16k	15.8	15.5	42.2-121.3	44-69	30-47	65-94
	100gene_20k	15.7	22.5	55-133.4	45-69	31-47	68-96
Synt.	deep-64char	15.5	6.9	13.7-17.3	68-69	1-1	91-92
	deep-256char	15.8	18.5	25.1-27.6	33-69	1-1	55-92
	shallow-64char	15.8	6.8	13.6-17.3	54-69	4-5	86-95
	shallow-256char	15.8	18.5	25.1-27.6	58-68	4-4	99-99

5.3 Results

Resource utilization is reported in Table 2 for GPU and FPGA, and in Table 5 (columns 7-12) for the AP.

For GPU, we recall that the NFA is stored in global memory, while the active state information (encoded in a bit vector) is stored in shared memory. In the CSR case, the active state information is also stored (in queue format) in global memory, and therefore the global memory requirement increases with the number of thread-blocks run. However, as can be seen in Table 2, the global memory utilization is very limited even for the CSR format, and even the largest dataset occupies only up to 133MB of the 12GB global memory. We recall that, in our experiments, NFA partitioning is driven by the use of 16-

bit state identifiers. The use of partitions with at most 64k states limits the shared memory utilization to 15.8KB in the worst case, allowing at least 6 blocks to reside on a SM and hide each other's memory latencies.

For FPGA, in order to facilitate the place&route process, we sized the partitions so as to use only up to 70% of the flip-flop capacity. In all cases but the l7-filter dataset, this allowed each partition to fit the FPGA device. As can be seen in Table 2, the LUT utilization is never a limiting factor (it is generally lower than the flip-flop utilization), and the place&route process balances the resources used over the available slices.

Despite its small size and low resource utilization, the l7-level dataset could not be deployed on a single FPGA in a reasonable time. This is due to the large ratio between transitions and states (on average 44), which complicates the wiring process. To allow reasonable preprocessing time, we split l7-filter in two partitions.

For the AP (Table 5), we report both the ideal utilization (the number of blocks and AP cores that a dataset would require based on the number of its STEs and reporting elements), and the utilization numbers reported by the AP's SDK (real utilization). As can be seen, due to the place&route constraints on the routing matrix, the real utilization is significantly worse than the ideal one, and the average STE utilization per core is mostly around 30-40%. The hardest datasets to deploy are again the l7-filter and the shallow synthetic datasets (~20% STE utilization), where non-terminal states have a large outdegree. On the other hand, deep datasets, where most non-terminal states have only one or few outgoing transitions, can be deployed more efficiently, leading to STE utilizations as high as 88%. Due to the generally low STE utilization, we partitioned all the state-reduced NFAs so that

each partition would require 50% (rather than the whole) half-core capacity. This led to the number of AP partitions shown in Table 1 (column 8).

In addition, we experienced that the AP SDK tools run out-of-memory when processing large datasets. To avoid this, for state-reduced NFAs we grouped partitions into batches of 32-64 (depending on the transition density of the dataset), and we run the AP SDK on one batch at a time. In the table, for each state-reduced dataset we report the min-max results across batches and the cumulative results. In case of large non state-reduced datasets (100gene*), which consist of many small NFAs with the same topology, we sized each batch so as to use all 32 cores on the AP. Since the place&route algorithm used by the SDK is proprietary, this was a trial-and-error process. For these datasets, we report the number of batches (which corresponds to the number of AP boards required), and the per-batch data. As can be seen, for small *k* (i.e., small hamming distance NFAs) the place&route is easier and the number of STEs/batch is higher. Since larger hamming distance NFAs are harder to place, the number of STEs/batch decreases as *k* increases. In general, as shown

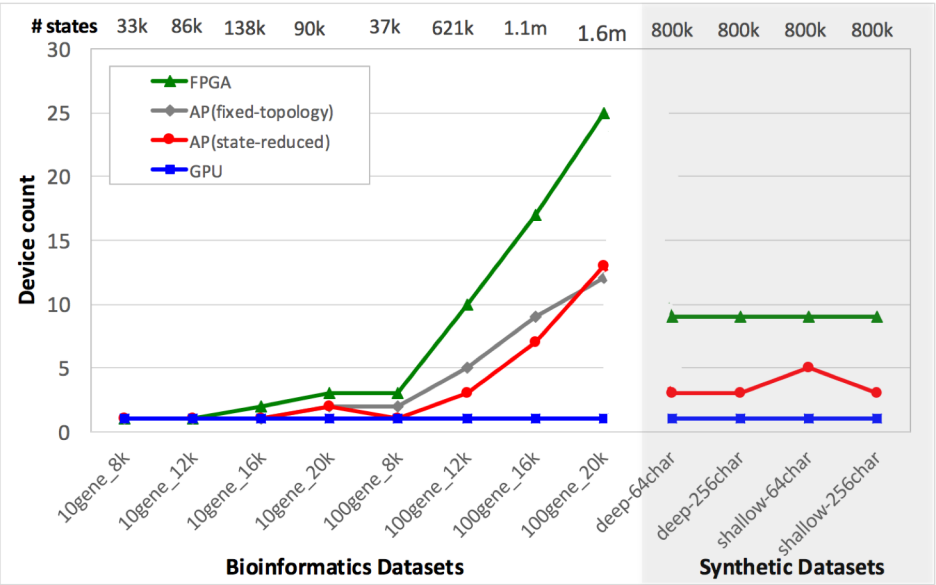


Figure 6: The number of devices required for each platform

in Figure 6, one GPU device can handle all datasets while we need to use multiple AP and FPGA devices for handling large datasets.

Table 3: Throughput (ranges corresponds to different number of streams)

Type	*Name*	*GPU*				*FPGA*
		Stream #	*iNFAnt*	*CSR-state*	*CSR-tx*	*Throughput per device (Gbps)*
			Throughput (Mbps)	*Throughput (Mbps)*	*Throughput (Mbps)*	
Small NIDS	l7-filter	48-4800	73.0-675.7	1.4-9.47	2.2-19.7	52.5-82.5
	snort534	48-4800	60.1-466.5	24.2-39.9	40.9-30.1	13.0
Bioinformatics	10gene_8k	72-480	21.9-32.5	3.1-3.2	0.6-0.8	4.18
	10gene_12k	36-240	7.1-11.9	4.7-4.8	0.6-0.8	1.06
	10gene_16k	24-160	5.3-7.9	8.9-14.0	0.6-0.9	1.1-2.7
	10gene_20k	18-120	3.2-5.5	7.3-12.5	0.6-0.9	0.9-1.0
	100gene_8k	24-160	4.2-7.5	0.5-0.6	0.2-0.3	1.3-1.4
	100gene_12k	6-35	0.8-1.4	0.6-0.6	0.2-0.2	0.9-1.6
	100gene_16k	3-18	0.3-0.5	1.2-2.2	0.2-0.4	1.0-2.1
	100gene_20k	2-12	0.2-0.3	1.0-1.7	0.2-0.3	1.0.1.7
Synt.	deep-64char	6-37	1.5-2.2	1.8-2.7	1.8-2.8	1.4-2.1
	deep-256char	6-37	1.7-2.6	0.5-1.7	1.4-2.7	1.39-1.94
	shallow-64char	5-32	1.2-1.9	0.4-1.8	0.5-2.2	1.22-1.24
	shallow-256char	6-35	1.7-2.7	0.2-1.0	0.3-2.6	1.11-1.18

Throughput data are shown in Table 3 for GPU and FPGA and in Table 5 for the AP, and assume that the matches are reported every 64K inputs (maximum IP packet length) for NIDS datasets, every 500 inputs (length of the relevant portion of a gene) for bioinformatics datasets, and every 1000 inputs for synthetic datasets. As can be seen, while able to fit even large datasets on a single device, GPU reports the lowest throughput data. In the GPU experiments, we configured the thread-block size to 256 and 32 for bioinformatics and NIDS/synthetic datasets, respectively. This is because we expected bioinformatics datasets to have larger active sets (as confirmed in Table 1). We recall that the number of thread-blocks run is equal to the product between the number of partitions and the number of input streams processed. To avoid idle SMs and ensure processing all partitions, we set the number of blocks to be at least equal to the number of SMs and of

partitions. We then increased the number of blocks (and, as a consequence, of streams) until noticeable throughput improvements could no longer be observed. We make two observations. First, GPU resources are better utilized when processing a large number of input streams, leading to better throughput. Second, while the iNFAnt kernel greatly outperforms the CSR kernels on small datasets, the CSR-state kernel reports better performance on bioinformatics datasets with large *k*. On datasets with a large number of partitions, iNFAnt is penalized by looping through a large number of transitions that originate from inactive states.

Since large datasets require multiple FPGAs and AP boards (or multiple iterations through the same board), for FPGAs and the AP we report the throughput per device. The number of FPGA devices required is equal to the number of FPGA partitions (Table 1/column 7), while the number of AP boards required is reported in Table 5/column 11 for reduced datasets, and in Table 5/column 3 for not reduced ones. For small datasets requiring only a small portion of the device, both platforms can run multiple streams in parallel. In case of

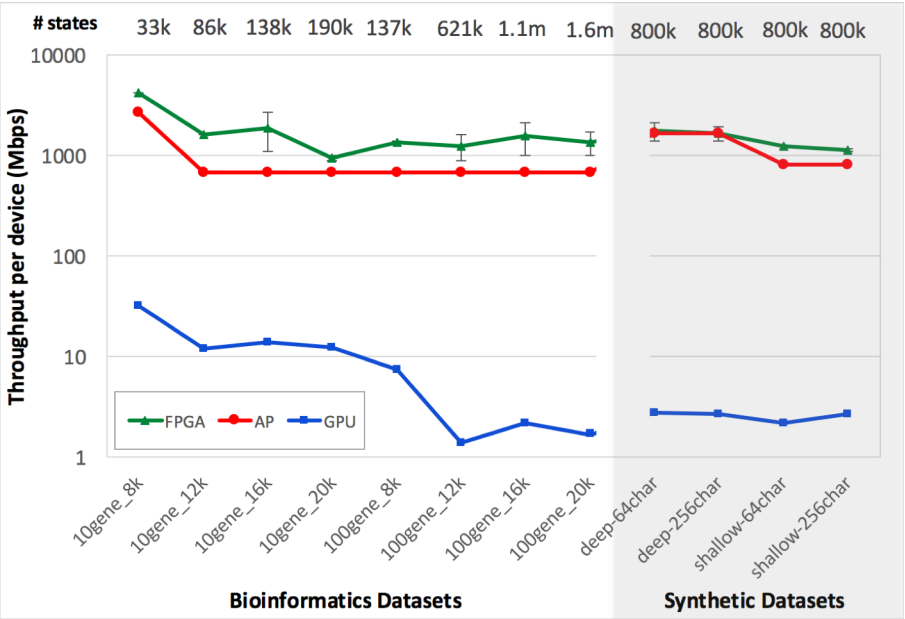


Figure 7: Best throughput per device for each platform

Table 4: Preprocessing overhead (in case of large datasets, we show minimum and maximum per-partition data)

Type	*Name*	*GPU*			*FPGA*		
		Parsing (sec)	*Mem. L. gen. (sec)*	*Loading mem. (ms)*	*Parsing (sec)*	*Verilog gen. (sec)*	*Synt.+ p&r (min)*
Small NIDS	l7-filter	5	0.01	0.004	0.2-0.2	0.2-0.3	7
	snort534	5.1	0.01	0.003	0.7	0.9	2-2
Bioinformatics	10gene_8k	3.2	0.05	3-6	3.8	14.7	293
	10gene_12k	5.8	0.3	8-12	8.7	135.5	1370
	10gene_16k	12	0.5	12-18	4.9-8.6	30.4-124.2	594-1528
	10gene_20k	12.4	0.9	16-26	9.1-9.1	161.2-186.7	1937-2300
	100gene_8k	12.3	0.4	12-18	7.4-9.5	54.8-107.5	771-1264
	100gene_12k	70	8.5	47-76	5.2-8.9	29.5-67.7	498-2175
	100gene_16k	114.8	36.7	99-167	5.8-9.4	40.2-197.1	595-2684
	100gene_20k	193.2	80.8	145-249	5.9-9.0	42.3-180.7	870-2660
Synt.	deep-64char	61.6	7.0	40-80	4.5-4.5	4.7-4.7	448-539
	deep-256char	267.8	8.4	110-150	2.4-4.9	2.6-5.4	99-433
	shallow-64char	76.5	6.1	40-80	5.2-6.6	5.6-7.1	435-738
	shallow-256char	279.5	6.8	120-150	6.3-7.1	12.9-14.2	433-485

FPGA, multiple streams require NFA replication, and we run 25, 4 and 2 streams for l7-filter, snort534 and 10gene_k8, respectively. In case of the AP, we consider that chips can be grouped into logical cores processing streams in parallel (Section 2.2). As can be seen in Figure 7, on large datasets (100ups* and synthetic) FPGAs outperform the AP up to a factor ~2x, while requiring 2-3x more devices than the AP. And GPUs underperform FPGAs by up to a factor of 900x. In order to increase the expressiveness, small NID datasets are eliminated from Figure 7.

Preprocessing cost: In all cases, we assume that the NFA has been optimized and partitioned, and saved into file. As can be seen from Table 4, the GPU preprocessing is mostly related to the parsing of the NFA partition files, and varies from 5 sec to about 4.5 min. For FPGA, synthesis and place&route account for most of the preprocessing time, and preprocessing a large partition may require up to 45 minutes (leading to several hours for the full datasets). Similar preprocessing times are observed on the AP (for example, the

preprocessing time for the shallow-256-char dataset is about 12 hours). In addition, the preprocessing time increases with the transition density (deep datasets are preprocessed must faster than shallow ones), whereas the alphabet size has a lesser effect (since on the AP transition symbols are associated to STEs and stored in memory). As mentioned, the AP preprocessing time can be reduced in case of datasets with known topology (i.e., non reduced datasets) by pre-compilation. However, finding a configuration that fully uses the AP is a trial and error process. Figure 8 represents the comparison between the preprocessing time of each platform and in most of the cases FPGA suffer from highest preprocessing time while GPU has the lowest preprocessing time.

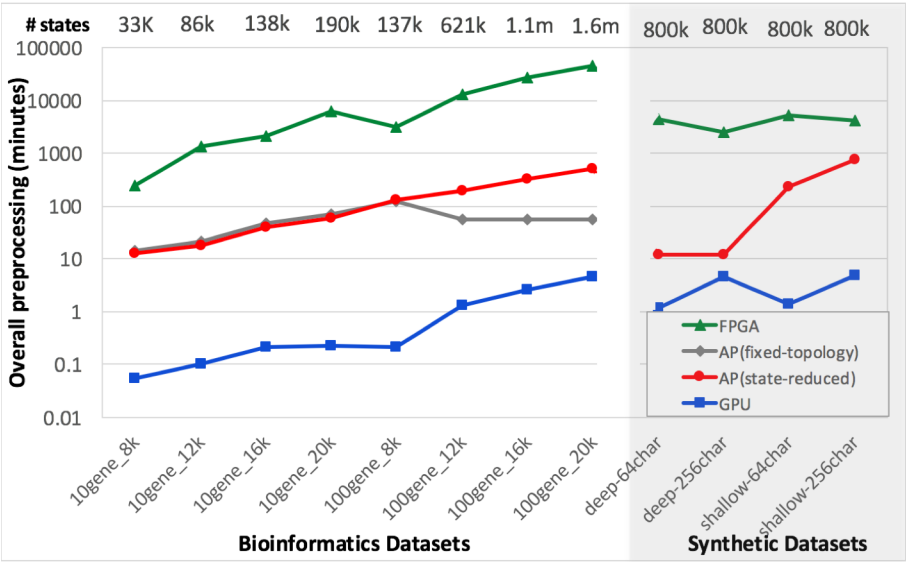


Figure 8:Preprocessing time for each platform

Table 5:AP results

Type		Name		# batches	ANML-NFA characteristics			Ideal utilization		Resource utilization from SDK profiling				SDK preprocessing time			Through-put per board (Mbps)
					# states/ STEs	# start states	# report states	# cores	# blocks	# cores	# blocks	# AP boards	STE util. %	Comp (sec)	p&r (sec)	total (min)	
Small NIDS		L7-Filter		1	4k	78	489	1	16	1	99	1	22.6	1	732	16	16946
		Snort534		1	11k	24	582	1	43	1	135	1	41.1	2	2261	38	16946
Bioinformatics	not reduced	10gene_8k		1	177k	10k	25k	4	694	6	1089	1	40.9	77	714	14	2690
		10gene_20k		1	462k	10k	24k	10	1804	37	7055	2	35.1	1587	2412	68	672
		100gene_8k		2	1300k	72k	181k	27	5079	32	6051	1	33.8	2109	5022	122	672
		100gene_12k		5	597k	21k	53k	13	2333	32	6123	1	40.8	1383	1861	55	672
		100gene_16k		9	436k	12k	29k	9	1704	32	6141	1	36.9	1226	2161	57	672
		100gene_20k		12	406k	9k	21k	9	1587	32	6118	1	35.4	1156	2086	55	672
	state-reduced	10gene_8k		1	56k	8	21k	2	218	6	971	1	33.1	68	674	13	2690
		10gene_20k		1	317k	8	22k	7	1239	35	6625	2	34.3	1030	2390	58	672
		100gene_8k		1	230k	32	136k	5	899	30	5643	1	21.2	1696	6098	131	672
		100gene 12k	Overall	2	985k	162	201k	21	3850	77	15k	3	45.6	4537	12k	192	672
			batches min-max		202k-783k	128-34	39k-162k	5-16	790-3060	16-61	3k-12k	1-2	45.3-45.8	328-4209	5517-6098	28-164	
		100gene 16k	Overall	5	1746k	300	197k	38	6822	153	29k	7	40.1	5273	13k	331	672
			batches min-max		248k-390k	44-64	27k-45k	11-16	968-6822	21-36	3940-6768	1-2	39.6-40.8	460-1428	1309-3214	46-76	
		100gene 20k	Overall	7	2592k	428	201k	56	10k	256	49k	13	36.5	9132	21k	502	672
			batches min-max		262k-410k	44-64	20k-32k	6-9	1026-1602	25-42	4644-8054	1-2	35.6-37.7	518-1617	2270-3376	40-84	
Synthetic	Deep	64char	Overall	3	800k	84	4447	18	3128	20	3623	3	86.2	324	300	12	1648
			batches min-max		102k-358k	16-33	566-1987	3-8	397-1400	3-9	456-1631	1-1	85.4-86.7	23-154	40-134	2-5	
		256char	Overall	3	752k	130	5096	17	2940	19	3300	3	88.6	404	249	12	1648
			batches min-max		78k-352k	31-52	528-2381	2-8	305-1374	2-9	342-1545	1-1	88.6-88.7	23-200	23-119	1-6	
	Shallow	64char	Overall	3	779k	101	382k	18	3045	79	15k	5	20.7	2725	11k	224	824
			batches min-max		104k-351k	12-47	51k-172k	3-8	407-1370	11-35	1949-6681	1-2	20.4-21	161-1403	1832-4740	27-103	
		256char	Overall	3	801k	193	317k	17	3128	66	12k	3	25.9	2258	43k	757	824
			batches min-max		90k-367k	29-79	36k-145k	2-8	352-1433	8-30	1411-5652	1-1	24.8-26.3	110-1136	4761-19k	82-338	

Chapter 6 Conclusion

To summarize, large datasets with more than 100-200 thousand states must be partitioned in order to be deployed on GPUs, FPGAs and Micron's AP. But, while for GPUs partitioning is required only to effectively use the GPU resources (e.g., on-chip memory), FPGAs and the AP require splitting large NFAs onto multiple devices. On these large datasets, logic-based FPGA designs can outperform the AP by a factor ~2x, while requiring 2-3x more devices to accommodate the dataset; GPUs underperform FPGAs by up to a factor 900x. GPUs in general offer low performance on a single input stream, but their cumulative throughput scales up to thousand input streams. GPUs offer the advantage of limited preprocessing time (up to a few minutes on million state NFAs), while FPGAs and AP can take several hours to preprocess the same datasets. Precompiling the NFA can hide the AP's preprocessing time, but this is possible only if the topology of the NFA is known a priori (e.g. Hamming or Levenshtein distance NFAs). Finding an NFA configuration that uses all 32 AP cores is a trial and error process that can require about one hour per experiment. Finally, due to routing constraints, AP's SDK can keep STE utilization down to 30-40% and as low as 20%, while the FPGA utilization is more predictable given the NFA size.

REFERENCES

[1] S. Kumar *et al.*, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in Proc. of SIGCOMM 2006.

[2] S. Kumar *et al.*, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in Proc. of ANCS 2007.

[3] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in Proc. of ANCS 2006.

[4] M. Becchi, and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in Proc. of ANCS 2007.

[5] M. Becchi, and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in Proc. of CoNEXT 2007

[6] M. Becchi, and P. Crowley, "Extending finite automata to efficiently match Perl-compatible regular expressions," in Proc. of CoNEXT 2008.

[7] R. Smith *et al.*, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in Proc. of SIGCOMM 2008.

[8] A. X. Liu, and E. Torng, "An overlay automata approach to regular expression matching," in Proc. of INFOCOM 2014.

[9] X. Yu, B. Lin, and M. Becchi, "Revisiting State Blow-up: Automatically Building Augmented-FA while Preserving Functional Equivalence," Journal on Selected Areas in Communications, vol.32, no.10, pp.1822-1833, Oct. 2014.

[10] N. Cascarano *et al.*, "iNFAnt: NFA pattern matching on GPGPU devices," *SIGCOMM Comput. Commun. Rev.,* vol. 40, no. 5, pp. 20-26, 2010.

[11] Y. Zu *et al.*, "GPU-based NFA implementation for memory efficient high speed regular expression matching," in Proc. of PPOPP 2012.

[12] X. Yu, and M. Becchi, "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," in Proc. of CF 2013.

[13] R. Sidhu, and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in Proc. of FCCM 2001.

[14] M. Becchi, and P. Crowley, "Efficient regular expression evaluation: theory to practice," in Proc. of ANCS 2008.

[15] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in Proc. of ANCS 2008.

[16] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in Proc. of ANCS 2007.

[17] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in Proc. of ISCA 2006.

[18] J. Van Lunteren *et al.*, "Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator," in Proc. of MICRO 2012.

[19] Y. Fang *et al.*, "Fast support for unstructured data processing: the unified automata processor," in Proc. of MICRO 2015.

[20] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in Proc. of ANCS 2009.

[21] P. Dlugosch *et al.*, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *IEEE Transactions on Parallel and Distributed Systems,* no. 99, pp. 1-1, 2014.

[22] I. Roy, and S. Aluru, "Finding Motifs in Biological Sequences Using the Micron Automata Processor," in Proc. of IPDPS 2014.

[23] K. Wang *et al.*, "Association Rule Mining with the Micron Automata Processor," in Proc. of IPDPS 2015.

[24] K. Zhou *et al.*, "Regular expression acceleration on the micron automata processor: Brill tagging as a case study," Proc. of Big Data 2015.

[25] I. Roy *et al.*, "High Performance Pattern Matching Using the Automata Processor," Proc. of IPDPS 2016.

[26] I. Roy, N. Jammula, and S. Aluru, "Algorithmic Techniques for Solving Graph Problems on the Automata Processor," in Proc. of IPDPS 2016.

[27] K. Wang, E. Sadredini, and K. Skadron, "Sequential pattern mining with the Micron automata processor," in Proc. of CF 2016.

[28] J. E. Hopcroft, and J. Ullman, *Introduction to automata theory, languages, and computation*: Addison-Wesley, Reading, Massachusetts, 1979.

[29] F. Yu *et al.*, "Fast and memory-efficient regular expression matching for deep packet inspection," in Proc. of ANCS 2006.

[30] K. Angstadt, W. Weimer, and K. Skadron, "RAPID Programming of Pattern-Recognition Processors," in Proc. of APLOS 2016.

[31] J. Wadden *et al.*, "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures," in Proc. of IISWC 2016.

[32] A. Todd *et al.*, "Parallel Gene Upstream Comparison via Multi-Level Hash Tables on GPU," in Proc. of ICPADS 2016.

[33] X. Wang, "Techniques for Efficient Regular Expression Matching Across Hardware Architectures," M.S. thesis, ECE. Dept., University of Missouri ,Columbia ,MO ,2014.