ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures

Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, Kevin Skadron
{wadden,vqd8a,njb2b,tjt7a,dg7vp,es9bt,kw5na,
cb2yy,robins,mircea,skadron}@virginia.edu
University of Virginia, Charlottesville, VA 22903

***Abstract*—High-performance automata-processing engines are traditionally evaluated using a limited set of regular expression-rulesets. While regular expression rulesets are valid real-world examples of use cases for automata processing, they represent a small proportion of all use cases for automata-based computing. With the recent availability of architectures and software frameworks for automata processing, many new applications have been found to benefit from automata processing. These show a wide variety of characteristics that differ from prior, popular regular-expression benchmarks, and these should be considered when designing new systems for automata processing. This paper presents ANMLZoo, a benchmark repository for automata-based applications as well as automata engines for both von-Neumann and reconfigurable dataflow architectures. To demonstrate the usefulness of ANMLZoo, we first characterize diversity in the benchmark suite. We then present insights from five experiments showing how ANMLZoo can be used to expose bottlenecks in both automata-processing software engines and hardware architectures.**

I. Introduction

Regular expressions are a language commonly used to define search patterns in strings. Regular expression pattern matches are usually recognized by converting to equivalent finite automata, and simulating the automata on an input string. While computing a small number of automata that fit in first-level caches can be fast, rulesets of many parallel, complex regular expressions can produce many large automata, requiring massive amounts of memory bandwidth with low-latency access to compute efficiently. For example, large, real-world regular-expression rulesets such as Snort [1] contain many thousands of rules that need to be matched simultaneously against streaming input.

To help accelerate regular-expression pattern matching, researchers have investigated algorithms for parallel computer architectures. Single-instruction multiple-data (SIMD) based graphics processing units (GPUs) and many-core and vector accelerators such as Intel's XeonPhi offer increased parallelism, helping to compute more automata transitions in parallel. Dataflow methods of automata processing on reconfigurable fabrics can be used to implement automata, and efficiently process a large number of automata transitions in parallel. Other hypothetical custom architectures for acceleration of automata processing have also been proposed in the literature [2]. Recently, Micron has released the Automata Processor (AP) [3], a reconfigurable fabric of automata matching elements that can efficiently process complex regular expressions by executing non-deterministic finite automata in hardware.

Because acceleration of automata processing has traditionally been motivated by network intrusion detection, new automata-processing engines on different architectures are evaluated using a small, relatively homogeneous set of existing representative regular expression rulesets [1, 4, 5]. Synthetic benchmark suites such as IBM's PowerEN suite [6] allow scientists to do more controlled studies of regular expression processing. Becchi et al. [7] created a synthetic regular expression rule and stimulus generator to help researchers do even more accurate sensitivity studies on regular expression processing engines.

While these regular expressions applications and benchmarks are valid and important real-world use cases for automata processing, they represent a very narrow range of all useful automata. Regular expressions, as written by humans, tend to be converted by classic algorithms into non-deterministic finite automata with very similar average structure, dynamic behavior, and matching complexity, and thus do not represent a wide range of possible useful automata structures or behavior.

Micron's Automata Processor and accompanying software development kit have made prototyping and development of automata-based (rather than regular-expression-based) pattern matching engines much easier. No good quantitative metric exists to measure the relative merits of either approach, but in our experience, directly constructing finite automata is an easier and more intuitive way for defining complex regular languages and pattern mining tasks. The availability of this software and hardware has led to the development of a large number of new, non-obvious automata-based applications in domains such as big data analysis [8], data-mining [9],

To appear in IISWC 2016. This is the author's version. The authoritative version will appear in the IEEE Digital Library.

bioinformatics [10, 11, 12], high-energy particle physics [13], machine learning [14], pseudo-random number generation and simulation [15], and natural language processing [16] that can differ significantly in static structure and dynamic behavior from existing regular expression benchmarks [1, 4, 5, 6].

This new diversity in automata-based applications, development tools, software recognition engines, and hardware architectures for automata processing, motivates a standard but flexible application and engine repository for fair evaluation of new automata-processing algorithms, architectures, and automata applications.

This paper presents ANMLZoo, a benchmark suite of automata-based applications for prototyping and evaluation of both software and hardware automata-processing engines. ANMLZoo contains 14 different automata-based applications that represent four major classes of automata: regular expression rulesets, string scoring meshes, programmable widgets, and synthetic automata, all of which may stress automata processing algorithms and architectures in different ways. Furthermore, ANMLZoo contains source code for high-performance automata processing engines that we have deployed for CPUs, Intel's XeonPhi, and GPUs, and can accomodate new state-of-the-art high-performance algorithms and architectures as they are developed.

The main contributions of this paper are the following:

- The creation of a public repository for standardized finite automata benchmarks and input stimuli, allowing for easy and fair comparisons of von-Neumann and reconfigurable fabric-based automata processing engines.
- The inclusion of parameterizable automata-generation scripts for some benchmarks, allowing for sensitivity studies of different static and dynamic automata properties of both real applications and synthetic automata.
- The analysis and categorization of automata using both qualitative and quantitative metrics. We consider both well-known metrics (node count, edge count, active set, etc...) and novel metrics (activity compressability, character set complexity) to categorize automata benchmarks. We show how some of these metrics impact performance on different architectures.
- The creation of a public repository for automata processing engines for several architectures for easy and fair comparisons of new algorithms and implementations to prior work. We include high-performance state-of-the-art automata-processing engines that we developed for CPUs (including Intel's XeonPhi), and for GPUs.
- The designation of a common tool for ANMLZoo automata manipulation, optimization, transformation, and analysis. Thus, new automata representations and optimizations can be easily shared among researchers and compared to prior work.

To demonstrate the usefulness of ANMLZoo, we investigate performance of four automata processing engines on four different macro architectures (Intel i7-5820K, Intel's XeonPhi 3120, a Maxwell-based NVidia Titan X GPU, and Micron's first-generation Automata Processor) and identify relative bottlenecks in each software engine and architecture pair.

II. Regular Expressions and Finite Automata

Regular expressions (regex(es)) are a convenient way to describe simple search patterns in text. Each search pattern defines a set of strings that belongs to a corresponding *regular language*.

All regexes can be described by equivalent *finite automata* and vice versa, making regular expressions and finite automata equivalent in power. However, regexes can be much less intuitive and more difficult to define and use in practice. Regexes are said to be *generative*, meaning that they define a set of rules that can be used to produce strings in the corresponding regular language. In practice, this makes them extremely useful for defining patterns in strings where the paths through the grammar are relatively simple. However, when attempting to define generators for more complex regular languages, regexes can quickly become unwieldy and unintuitive as a descriptive medium.

Unlike *generators*, finite automata are said to be *recognizers* and are machines designed to accept all strings in a language. Finite automata are sets of states with transition rules describing transitions between states. A finite automaton computes by following transition rules from start states based on the current input symbol to the machine. An automaton recognizes a string in the corresponding regular language when an input sequence causes it to transition to an accept state.

Because they are equivalent, regexes are often converted to finite automata recognizers for evaluation. The following section details current state-of-the-art approaches to automata processing, and the baseline automata processing engines considered on each available architecture.

III. Automata Processing

A. Von Neumann Automata Processing

Von Neumann automata processing involves simulating a finite automaton on an input string of symbols in order to recognize strings that belong to the corresponding regular language. Non-deterministic finite automata (NFAs) allow for an automaton to be in multiple active states at once. For each symbol in the input stream, each active state in the finite automaton must look up its appropriate transition rules in memory and execute those transitions in the automaton. While simple, this algorithm has an extremely low computational intensity per rule fetch, and thus relies on high-bandwidth, low-latency memory accesses for good performance.

Von Neumann architectures therefore perform better when fewer memory accesses are required, and prior work has investigated transformations, optimizations, and new automata models to reduce the number of transitions per cycle necessary to compute [2, 5, 17]. The most drastic of these transformations is the conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA). Because DFAs only allow one transition per cycle, they are optimal for architectures with limited memory accesses per cycle.

However, a DFA may require an exponentially larger number of states to represent all possible combinations of NFA states; this full transformation can often be prohibitively expensive in both time and space.

Many techniques make trade-offs between these two representations [2, 17] and attempt to find reasonable space/time overheads to reduce the total number of average memory accesses, and also support a larger feature set than traditional finite automata. ANMLZoo is a collection of traditional, theoretical non-deterministic finite automata (NFA), as a baseline for comparisons and evaluations of these techniques. Below we describe each baseline von Neumann NFA processing engine used in our evaluations.

VASim (CPU/XeonPhi): VASim is a high-performance, open source Virtual Automata SIMulator for automata processing research. The core execution architecture of VASim is an optimized version of the classic NFA algorithm described above. Each VASim thread is responsible for processing all transitions in a set of automata, but only considers automata states that are currently active. VASim is parametrically multi-threaded in two dimensions: groups of distinct automata can be assigned to parallel threads, and/or different sections of the input symbol stream can be assigned to parallel threads computing the same automata. Therefore, many thread contexts can be launched to take advantage of parallel cores in multi-core CPU architectures. We use VASim as the baseline NFA engine on both server-class CPUs and Intel's XeonPhi many-core co-processor for our evalutions. While other complex hybrid NFA/DFA automata processing methods exist [18, 19], we consider fast, basic NFA emulation as the best baseline for easily examining bottlenecks in architectures for automata processing. Performance of other parallel CPU engines may be considered and catalogued in the future.

VASim can also programatically manipulate and optimize automata, and will act as a public tool for common automata optimizations and transformations. As an example of VASim's usefulness as a transformation platform, we analyze its ability to perform *common prefix merging* on automata. Common prefix-merging attempts to eliminate redundant states (and therefore state matching), and can greatly reduce the number of required memory accesses and size of highly redundant automata [20].

iNFAnt2 (GPU): Modern GPUs offer massive SIMD and task-level parallelism, and thus are a tempting target for acceleration of automata processing. We consider an optimized version of the iNFAnt [21] tool–dubbed iNFAnt2–as our baseline NFA processing engine for the GPU.

In the iNFAnt framework, the NFA transition tables are organized using a symbol-first format with transitions grouped by their incoming symbols. Each SIMD thread is assigned a possible transition on an incoming symbol from a source state to a destination state. Because the number of transitions per incoming symbol is usually not uniform, a supplementary array is used to store the offset in the NFA transition table that indicates the first transition for each particular symbol. All threads fetch the correct offset into this table based on the incoming symbol. They then fetch and accomplish corresponding transitions from the NFA transition table, but only threads with source states that are currently active compute.Both the NFA transition table and the supplementary array are stored in GPU global memory due to their large size. Bit-vectors are used to represent current and future state vectors and are stored in GPU shared memory.

The main disadvantage of iNFAnt is that it assigns individual SIMD threads to compute all *possible* transitions for a particular incoming symbol across every state in the automata. This can be very inefficient when the actual number of active states is small. In contrast, VASim uses only one thread to compute all necessary transitions. For baseline GPU automata processing performance, we present iNFAnt2, which adopts the iNFAnt framework with our own improvements, as well as modifications inspired by Vasiliadis et al. [22]:

- Fast accept-state recognition by encoding accept states with negative IDs
- Multi-byte input symbol fetches
- NFA transition tables stored in GPU texture memory

We used the new instruction level profiling capabilities of Maxwell-based NVIDIA GPUs to pinpoint and relax performance bottlenecks. We also augmented iNFAnt2 to report the cycle and rule ID of each matched rule, an important and previously unimplemented feature. Thus, we consider iNFAnt2 the state-of-the-art GPU NFA processing engine, and a good baseline for evaluation against future research.

In Section VIII, we also evaluate performance of a state-of-the-art GPU DFA engine inspired by the implementation in Becchi et al. [23, 24] and the multi-DFA implementation of Vasiliadis et al. [22]. This engine was also tuned using NVIDIA's new instruction-level profiling tools, and is considered close to state-of-the-art.

B. Data-flow Automata Processing

Knowing that efficient parallel simulation requires large numbers of low-latency rule lookups, researchers have also investigated methods of automata processing on data-flow architectures. Prior work has exploited the reconfigurable nature of field programmable gate arrays (FPGAs) to lay out automata states and transitions in reconfigurable logic fabrics and is summarized in Becchi et al. [17].

Micron, leveraging their experience and IP in memory technology, has developed the Automata Processor (AP) [3], a DRAM-based reconfigurable, native-hardware accelerator for non-deterministic finite automata (NFA). The AP implements an NFA using a reconfigurable network of *state transition elements* (STEs)–analogous to NFA states–that all consume a single input stream of 8-bit symbols. If an STE is enabled, and matches the current input symbol, it activates, propagating enable signals to other STEs via an on-chip routing matrix. All transitions happen in parallel thus AP performance is always linear in the size of the input symbol stream, and independent

of the dynamic activity in the fabric. STEs are capable of single-bit reports, analogous to NFA "accepting states."

Rather than restricting development of applications to regular expressions, Micron's AP software development kit provides a software infrastructure to easily and programmatically create arbitrary automata-based applications. Micron has created both a standard language for defining automata (automata network markup language or ANML) to define networks of automata states, and also a software development kit to easily and programmatically build automata [25]. Furthermore, new high-level programming languages have been developed that can be easily compiled to automata [26].

As a result of this new software infrastructure, many new and non-obvious automata-based applications have been created, which perhaps would never have been developed with regular expressions.

IV. ANMLZoo: An Automata Benchmark Suite

This relatively new diversity in automata-based applications motivates the creation of an automata benchmark suite for prototyping and evaluation of both software and hardware automata-processing engines. We present ANMLZoo, a suite of 14 sets of automata that represent four major classes of automata applications: regular expression rulesets, string scoring meshes, programmable widgets, and synthetic automata, which all have different major properties that may stress automata processing algorithms and architectures in different ways. ANMLZoo also contains the baseline implementations of automata processing engines (discussed in Section III) that can be used as reference points for future research.

A. Problems with Existing Rulesets and Generators

Research into fast regular expression processing engines has traditionally been motivated by deep packet inspection, which includes applications in the network intrusion detection system (NIDS) community.

NIDS rulesets such as Snort [1], as well as virus detection rulesets such as ClamAV [4], and synthetic rulesets such as PowerEN [6] have been popular rulesets for benchmarking existing regular expression engines [2, 17]. However, it is desirable to have a common and flexible methodology for benchmarking and conducting sensitivity analysis on regular expression engines with parameterizable rulesets and input stimuli. Becchi et al. [7] constructed a synthetic regular expression generation tool that parameterized regular expression features that make DFA conversion expensive. This tool also includes an automatic trace generation tool, which can tune input streams to induce various levels of activity in any automaton. However, this tool was motivated and designed to generate regular expressions and inputs to better evaluate deep packet inspection engines and architectures, and not for arbitrary automata processing.

Similarly, new architectures for automata evaluation [2] are designed for and evaluated using the above mentioned patterns, or with simple exact match strings, highly compressable binary trees [27], and/or finite automata with a very small number of states.

As our characterization will show, existing regular expression and automata benchmarks are either very similar in static and dynamic properties, or not publicly available, easily accessible, or in a common format. This makes it extremely difficult or even impossible to evaluate improvements over existing state-of-the-art publications. Many state-of-the-art software engines and infrastructures for automata processing and transformation are also not publicly available, again making it difficult or impossible to do fair evaluations of existing automata-processing algorithms and implementations on different architectures.

B. ANMLZoo: an Automata Processing Benchmark Suite

To address the above drawbacks with the current methodology for benchmarking of automata processing engines we present ANMLZoo, a repository for automata benchmarks, input stimuli, and software engines and infrastructures for fair benchmarking of new automata processing engines. Each ANMLZoo benchmark is shown in Table I.

Below we list each application in ANMLZoo, including both existing popular regular expression benchmarks, as well as a new set of recently published automata-based applications that together form a much more diverse starting point for benchmarking of automata-processing engines.

Snort[1] are regular expressions extracted from a snapshot of the snort ruleset, commonly used to benchmark regular expression processing engines.

Dotstar [23] is a combined set of synthetic regular expressions from Becchi et al. [23] containing all variations of the synthetic dotstar rules created from the backdoor Snort rules and the spyware rules used in that evaluation.

ClamAV [4] is a set of regular expression signatures for identifying virus signatures in files. Our benchmark includes ClamAV rules with small (< 64) quantifiers and no ranges.

PowerEN [6] is a combination of over 2000 regular expressions from the PowerEN "complex" regex rule set.

Brill [16] is a set of over 2000 Brill tagging rules.

Protomata [11] is a set of 2340 real and randomly generated protein motif signatures.

Hamming [10] is a set of 93 Hamming distance automata used to calculate the number of mismatches between a randomly generated encoded string and random input sequence.

Levenshtein [12] is a set of 24 Levenshtein automata designed to calculate the edit distance between an encoded DNA sequence and an input DNA sequence.

Entity Resolution [8] is a set of automata designed to identify whether input name sequences match a certain encoded pattern.

SPM [9] or Sequential Pattern Mining, is an automata-based application to identify groups of related items in baskets.

Fermi [13] is a path recognition algorithm that looks for sequential series of ordered coordinates defining a particle path.

Benchmark	Family	States*	Compressability	Node Degree*	Charset Complexity*	Active Set*	Activity Compressability
Snort	Regex	34,480	50.04%	1.13	8.74	98.45	75.71%
Dotstar	Regex	38,951	59.6%	1.01	8.28	3.25	92.79%
ClamAV	Regex	42,543	14.12%	1.02	7.86	4.30	94.78%
PowerEN	Regex	34,495	14.85%	1.06	8.11	31.15	66.20%
Brill†	Regex	26,364	38.20%	1.49	8.75	14.28	99.14%
Protomata†	Regex	38,251	8.95%	1.04	19.44	554.281	63.15%
Hamming	Mesh	11,254	0.81%	1.71	9.89	240.1	15.78%
Levenshtein	Mesh	2,660	4.45%	3.36	8.0	88.02	22.93%
Entity Resolution	Widget	5,689	94.02%	6.38	8.60	10.62	99.11 %
SPM	Widget	100,500	0%	2.1	6.58	6,331.32	0%
Fermi	Widget	39,033	4.29%	1.48	8.18	3854.45	∼0%
Random Forest	Widget	71,574	5.00%	1.053	14.26	968.64	1.26%
BlockRings	Synthetic	44,352	NA	1	8	192	NA
CoreRings	Synthetic	48,002	NA	1	8	2	NA

TABLE I

ANMLZOO BENCHMARK SUITE. † NEWLY PUBLISHED AUTOMATA-INSPIRED REGEX-LIKE RULESETS. RESULTS ARE GATHERED USING REPRESENTATIVE INPUT STREAMS SHOULD BE CONSIDERED BASELINE RESULTS, AND MAY CHANGE WITH NEW ALGORITHMS, IMPLEMENTATIONS, AND ARCHITECTURES.

Random Forest [14] is an encoded and compressed implementation of a random forest ensemble classifier for handwriting recognition.

BlockRings are synthetic automaton rings with deterministic behavior meant to occupy each block (192) on an AP chip.

CoreRings are synthetic automaton rings with deterministic behavior meant to occupy each core (2) on an AP chip.

The difference between automata constructed by regular expressions and other modern automata-based applications in ANMLZoo can be easily quantified by looking at both static and dynamic properties.

Before we perform static or dynamic analysis, all automata are compressed using *common-prefix merging* (CPM) [20]. CPM merges redundant states from the automata in a breadth-first manner, from start states to end states, while preserving automata correctness. This optimization can greatly reduce the size of, and redundant traversals for, automata, and is thus used for baseline static and dynamic evaluation of automata. However, we do not claim that these are optimally minimized automata, and thus they may be compressed further. Dynamic properties can vary greatly depending on the corresponding input stimulus, and so metrics like *active set* should also not be considered inherent properties of the benchmarks, but rather based on the quality of optimization and behavior provided by a representative input.

We considered five metrics to quantify differences in automata applications. Each metric is described below:

- **States:** The total number of states (STEs) in the common prefix-merged (CPM) automata graph. The capacity of an AP chip is 49,152 STEs. State counts lower than this number indicate lower utilization of on-chip resources and a harder routing task for the AP compiler and fabric. State counts higher than this number indicate the Micron compiler was able to identify compression opportunities other than CPM.
- **State Compressability:** The percentage of redundant states removed by CPM. High compressability reduces pressure on reconfigurable resources in FPGAs and the AP, but also may improve cache behavior in von Neumann engines.
- **Node degree:** The average output degree of each node. Higher node degrees and more connectivity indicate a harder place and route task for spatial automata processing engines like FPGAs and the AP.
- **Character set complexity:** We use the Quine-McCluskey algorithm [28] to calculate the minimum number of boolean terms required to compute the boolean match function corresponding to the character set of an automaton transition rule (STE). This metric reflects the average difficulty in building a circuit to compute the match function of a particular STE.
- **Active Set:** The average number of active states. Larger numbers of active states require more transition rule fetches. Thus, this is a proxy metric inverse to performance in von Neumann architectures. Spatial architectures like the AP are unaffected by active set because all transitions are accomplished in parallel in a single cycle if the design can successfully place and route.
- **Activity Compressability:** The average amount of redundant activity removed by CPM. This metric roughly indicates how much performance is gained on von Neumann engines from the CPM optimization.

Table I shows that many benchmarks derived from regular expressions have similar static properties. Each regex benchmark has an average node-degree of about 1, reflecting the long strings of automata states that are often emitted from typical regular-expression-to-automaton conversion algorithms. In contrast, applications such as mesh automata and Entity Resolution (which uses Hamming automata as a subkernel) have many more output edges and represent a much more complex structure and routing task for spatial automata-processing fabrics. Regular-expression-like automata tend to have a high number of common prefixes, reflected in high state and activity compressability factors. In contrast, automata widgets such as Fermi are designed to compute non-obvious recognition tasks and generally have much less redundancy by design. We explore automata compressability via CPM and its effect on performance in Section V.

ANMLZoo provides the following features:

Diverse Automata Structure and Behavior: ANMLZoo is originally divided into four major automata families: regular-expression-derived automata (Snort, ClamAV, PowerEN, Brill, Protomata), automata meshes for string scoring (Hamming, Levenshtein), structured processing elements or "widgets" (SPM, Random Forest, ER, Fermi), and synthetic automata with exact known properties (BlockRings, CoreRings). All applications are quantitatively diverse in both static and dynamic properties, and reflect real-world uses of automata.

Standard Candles: Each ANMLZoo benchmark provides at least one file that defines a standard set of automata that max out the resources of a single first-generation Micron D480 AP chip. While there is no one "correct" way to standardize trade-offs among both automata state-size, activity, and connectivity, we chose this metric as a compromise to allow easy and fair comparisons between different reconfigurable data-flow architectures and von Neumann automata processing engines. We call these standardized automata the ANMLZoo *standard candles*. Because standard candle automata max out the resources of an AP chip, we can easily and fairly compare application performance on other automata processing architectures against different deployment scenarios of small 4W AP D480 chips. For example, performance of 1 AP rank (8 parallel AP chips consuming 8 parallel input streams), is trivial to deduce via multiplying the performance of one AP chip by 8. This feature is exemplified in Section X. Each *standard candle* benchmark is also accompanied by a corresponding stimulus of 1MB and 10MB, for testing and evaluation respectively. Versions of standard candles and input stimulus may evolve to adapt to the needs of the community, but automata and input stimuli will never be removed from the suite allowing for easy and fair comparisons to prior work.

Written in ANML: Each application is defined using Micron's Automata Network markup language [25] or ANML. ANML allows a standard but flexible method for defining automata networks. ANML is an XML-like language that is used to define automata computation graphs. Applications that are defined as regular expressions can be converted to ANML automata using Micron's SDK [25]. If a benchmark is derived from regular expression rulesets, these rules are also included in the suite.

Parametric Automata Generation Scripts: Where possible, automata generation scripts have been provided to facilitate sensitivity analyses of different automata-processing applications and architectures. For example, Section VI shows how performance of von Neumann architectures is impacted by varying fixed properties of synthetic automata and Section IX shows how AP chip utilization is affected by varying dimensions of mesh automata (Hamming, Levenshtein). These sensitivity analysis cannot be done using the fixed benchmarks like the standard candle automata.

Baseline Automata Processing Engines: ANMLZoo also includes source for baseline CPU and GPU finite automata engines discussed above, and is an open repository for additional high-performance algorithms and implementations for varying computer architectures. ANMLZoo also provides a common software framework to manipulate, optimize, and convert automata written in ANML to other formats required by other engines.

To demonstrate the usefulness of ANMLZoo, the following sections present five different experiments exploring the sensitivity of different engines to different types of automata, exposing bottlenecks in automata processing engines on von Neumann and dataflow architectures, and relative advantages of automata processing on each available architecture. For each CPU experiment, we use a 6-core (12-thread) Intel i7-5870K clocked at 3.3GHz with 32GB of RAM clocked at 2166MHz. This server also acts as the host CPU for the following accelerators. For each many-core CPU accelerator experiment, we use a 57-core (228-thread) Intel XeonPhi 3120p clocked at 1.1GHz. For each GPU accelerator experiment, we use an NVIDIA Maxwell-based GTX Titan X clocked at 1GHz. For each AP fabric utilization experiment, we use Micron's AP SDK version 1.6.5.

V. Parallel Automata Rule Scaling

Many regular expression processing applications are concerned with the number of parallel "rules" or automata that a given engine or architecture can process. For von Neumann architectures, more automata computed in parallel may mean more transitions to compute, and more pressure on the memory hierarchy. For data-flow, reconfigurable fabric automata-processing architectures, more parallel automata lead to a higher number of states, and thus more pressure on the underlying reconfigurable fabric capacity and routing resources. Thus, the more rules an engine is capable of quickly processing, the more desirable the engine.

However, "number of rules" is a poor metric to measure the amount of work being done by an automata engine. As an example, we consider two applications from ANMLZoo (Entity Resolution and Fermi) and vary the number of rules processed by a single thread on the CPU, measuring the sensitivity of performance of VASim to the number of automata being computed. Figure 1 shows the results of our experiment.

We plot normalized performance of common-prefix-merged versions of the automata rules (compressed) and the original (uncompressed) versions of the automata rules for both Entity Resolution and Fermi. The common-prefix-merged version of Entity Resolution, while initially incurring a severe penalty for additional rules, quickly reaches a point where additional rules have little impact on performance. This is due to the high activity compressability of Entity Resolution, as the redundant states removed by common-prefix merging were also responsible for a high amount of redundant activity in the original automata.

Figure 1 also plots the performance cost of adding rules without the benefit of common-prefix merging. The performance penalty of additional rules is much more severe, and

Fig. 1. Sensitivity of VASim performance in response to additional automata rules. Performance of automata with many common prefixes and high activity compressability (ER) are less sensitive to additional rules. This indicates average automata activity after common optimizations, rather than total rule-count, is a better predictor of performance.

does not plateau, indicating new rules require significant additional activity and computation when uncompressed.

For the Fermi application, both the CPM and non-CPM versions have near-identical performance characteristics. This is because Fermi has an extremely small activity-compressability factor. Specially-designed automata like Fermi are therefore extremely important to consider when characterizing new automata engines and optimizations, as they are harder to compress, and therefore pay a larger penalty when computing additional rules.

VI. Visited Set and Active Set Sensitivity

Because automata processing on von-Neumann architectures requires many sequential accesses to memory, performance of automata processing on these architectures has been shown to be limited by access latency in the memory hierarchy [2, 27]. However, this bottleneck and its impact can greatly depend on the underlying automata engine algorithm and implementation. In the previous section, we saw that automata activity, rather than "number of rules," was the main factor hurting performance, but this activity was not measured or controlled for.

In general, it can be difficult or impossible to guarantee certain properties of automata for controlled experiments. It is therefore important to have a set of automata benchmarks (or generation tools) in the benchmark suite that can precisely vary metrics such as the *visited set* (the set of states consistently visited during computation) and the *active set* (the number of active states which need to perform memory accesses per cycle). These synthetic automata and synthetic automata generation tools allow for controlled experiments measuring the specific impact of memory hierarchy latency or throughput on total performance.

We present a parametric synthetic automata design to control for the ratio of active set to visited set. The synthetic automata design is shown in Figure 2.

Each automaton is organized as a ring of stages. Each stage in the ring has a fixed number of states that is always activated

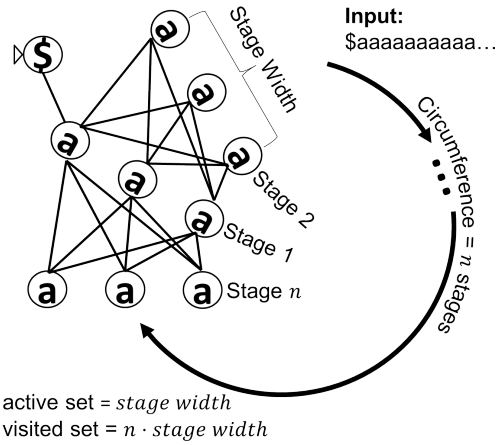

Fig. 2. Parameterizable synthetic automata design. Each ring is guaranteed to have a constant active set and visited set, and is driven by an easy-to-generate input string. This instance has width 3, thus active set 3. Each stage is fully connected with its succeeding stage to form a continuous ring. The circumference, n, is derived using the equation $n = \lceil \frac{visited}{width} \rceil$.

by the previous stage. This property guarantees that at any one cycle, the *active set* in any one ring is equal to the width of the stage. Each ring is also of a fixed circumference (i.e. the number of stages in the ring). Therefore, the total *visited set* of the automaton is the width of the stage times the number of stages in the ring. This design allows us to individually control for both active set and visited set, and isolate the impact of each on performance of different automata-processing engines. Below shows the results of VASim performance while varying active set and visited set independently. We vary the visited set of a single ring by multiples of 10 states from 100 to 100,000 states, and vary the active set of each ring by 1 from 1 to 20.

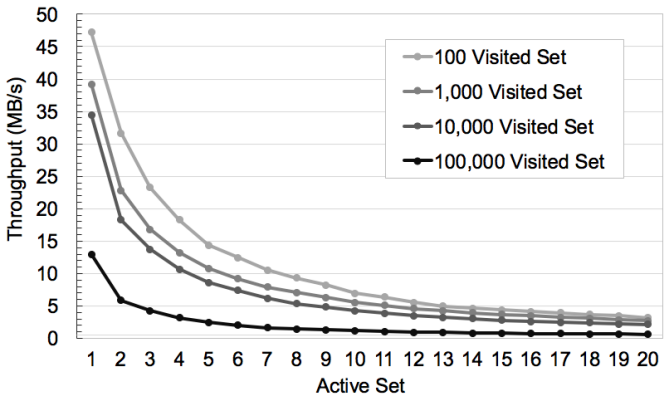

Fig. 3. Sensitivity of automata simulation performance to changes in the active set (number of states considered per cycle) and the visited set (number of states consistently visited). Performance is much more sensitive to increases in active set. The visited set impacts performance when its size grows larger than the size of an available level of cache.

The more average states visited, the larger the pressure on the caches in the memory hierarchy of a von Neumann architecture. Therefore, guaranteeing that the visited set fits into L1 or L2 caches of a CPU can be extremely important for high-performance.

When the number of states is between 100 and 10,000,

increasing the size of the visited set has little impact on performance. This indicates that the entire visited set fits within a single level of the memory hierarchy, and so a larger number of states does not impact the performance of computing transitions for the active set. However, there is a relatively large impact when increasing the size of the visited set from 10,000 to 100,000.

While increasing the size of the visited set does impact performance, slight increases in active set can have extremely large impacts on performance. Because automata structure can be irregular and behavior is often unpredictable, it is difficult to guarantee locality of access. Therefore, to improve automata processing performance on the CPU, VASim must work to reduce the size of the average active set via automata optimizations and transformations, but also maintain an automata visited set size that optimizes performance.

VII. Automata vs Input-level Parallelism Scaling

Because it can be difficult or impossible to reduce the active set of automata, and improving memory latency at the architectural level can be extremely expensive, automata engines often attempt to exploit parallelism among independent automata and among automata input streams to hide the latency of individual transitions and increase throughput of automata engines. This section explores sensitivity of automata engines and architectures to these two dimensions of parallelism– parallel automata and parallel input streams. Distinct automata can be divided into an equal number of groups (G), and the input stream can be divided into an equal number of sections (S). Thus, we can launch $G \times S$ number of CPU threads or GPU thread-blocks to compute in parallel.

We pick three applications: Protomata, Hamming, and Random Forest to illustrate how different families of applications (regex, mesh, and widget) respond to varying automata group and stream parallelism. For each application, we pick 4 sets of groups and vary the number of parallel streams on both the VAsim (CPU) and iNFAnt2 (GPU) baseline engines. Results are presented below.

A. CPU Parallel Scaling

Results from varying automata groups and parallel input streams on VASim, our baseline CPU automata engine, are shown in Figure 4.

Hamming automata seem to favor more parallel automata groups and are not accelerated by increasing the number of parallel packet streams. This is because Hamming automata have relatively little activity compression and so parallel threads computing parallel automata are more likely to be doing distinct, non-redundant work. Thus it is better to have single threads operate on smaller, distinct automata that may have good behavior in an individual CPU's L1 cache.

Protomata is much more responsive to both automata-level and input-level parallelism. This is because Protomata has a small number of automata that have a much greater level of activity than others. Because VASim is not equipped to parallelize work within individual automata, the threads that are responsible for these "problem" automata run much slower and bottleneck performance. While automata-level parallelism cannot accelerate problem automata, stream-level parallelism can. Thus Protomata performs best with eight parallel automata groups (8), but a larger number of packet streams (12).

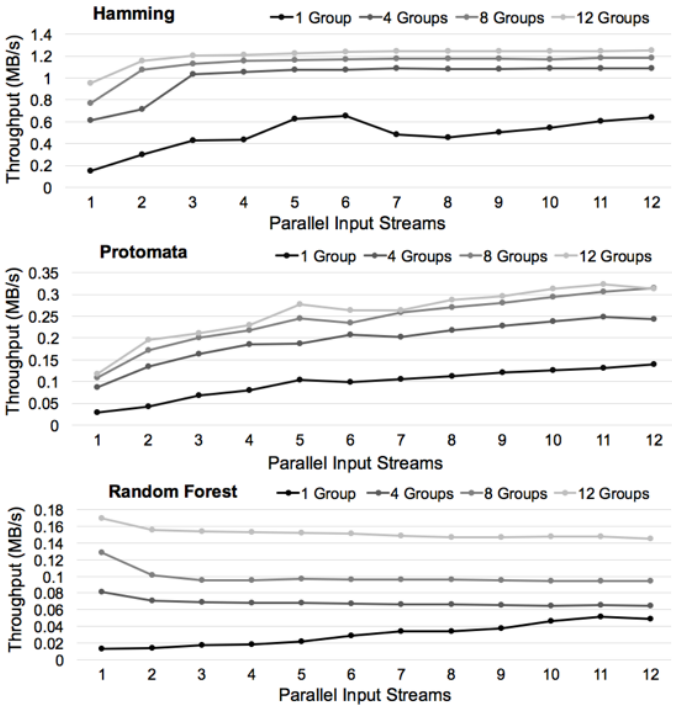

Fig. 4. Hamming automata benefit most from automata-level parallelism. Protomata benefits from parallelism in both dimensions. Random Forest only benefits from automata-level parallelism.

Random Forest has an extremely low level of activity compressability and so benefits most from distributing automata across many threads. Random Forest benefits so much from parallel automata computation, that any additional thread contexts for computing parallel input streams hurts performance, even when cores are underutilized. This indicates that shared per-chip (as opposed to per-core) resources like L2 and L3 cache are over-utilized, and important for performance where active set is high.

These experiments shows that parallelization strategies for CPU-based automata processing depend highly on the automata topography, compressability, and dynamic behavior.

B. GPU Parallel Scaling

Results from varying automata groups and parallel input streams on iNFAnt2, our baseline GPU automata engine, are shown in Figure 5.

Unlike the CPU-based engine, Hamming automata on the GPU overwhelmingly favor more parallel input streams. Hamming performs best when each CUDA thread block operates on all meshes simultaneously and there are more than 560 parallel blocks operating on different sections of the input stream. This highlights the ability of the GPU to hide the latency of any individual memory access by executing an extremely large

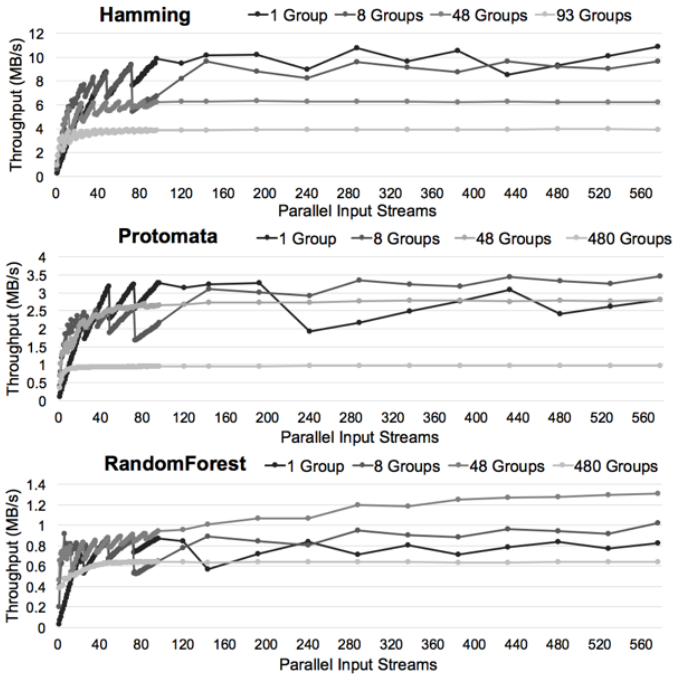

Fig. 5. Protomata, Hamming, and Random Forest all benefit from a massive amount of stream level parallelism, however appropriate care must be taken to tune automata groups to match GPU core resources.

number of parallel tasks. Because there may be a relatively small number of parallel accesses in any one benchmark (e.g. Hamming has an average active set of 240 over 49 distinct automata when prefix merged) it is generally better to exploit input stream-level parallelism for latency hiding on the GPU.

Protomata shows similar performance characteristics to Hamming. However, a single group does not universally perform best. Dividing the automata into eight groups performs better as the number of parallel streams is increased. This reflects sensitivity to utilization of per-GPU stream-processor resources such as shared memory and L1 cache. The total performance of the GPU engine relies on a balance of NFA transition table size and stream-level parallelism that is highly application specific.

Random Forest, as discussed earlier, has a small amount of activity compressability, and therefore favors computation by more parallel groups, with smaller, more efficient NFA transition tables. Random Forest performs best on the GPU when split into 48 distinct groups. However, it is still the case that too many automata groups will limit stream level parallelism, and reduce the ability of the GPU to hide the latency of transition table lookups.

These experiments show that GPU automata processing engines mostly favor parallelization via parallel input streams. If an application allows its input stream to be divided among parallel threads, the GPU can better hide the long latencies associated with SIMD control-flow and memory divergence.

VIII. NFA vs. DFA Engines on the GPU

The variable topography and dynamic parallelism of NFAs can be especially difficult to efficiently map to the GPU's SIMD architecture. Thus, deterministic finite automata (DFA) have been explored as a possible alternative method of automata processing to better exploit the GPU's available resources [22, 23]. DFAs are equivalent automata that are constructed so that only one state can be occupied at any one time. A DFA state therefore represents a particular configuration of NFA states. Because of this relationship, DFAs can be potentially exponentially larger than their equivalent NFAs, and exponentially expensive in time to construct.

We use the DFA generation tool developed by Becchi [5] to convert as many ANMLZoo NFAs to DFAs as was possible. Some ANMLZoo applications took too long, or required too much memory to be converted to a reasonable number of DFAs and were ignored. The GPU DFA engine in iNFAnt2 assigns individual CUDA threads within a thread-block to processes a particular DFA. In contrast, the iNFAnt2 NFA engine maps the computation of entire NFAs to CUDA thread-blocks.

Figure 6 shows the relative performance between our baseline NFA and DFA engines achieved using the optimal block and grid size, and thread and stream configuration for each application. The DFA-based engine traverses exactly one state

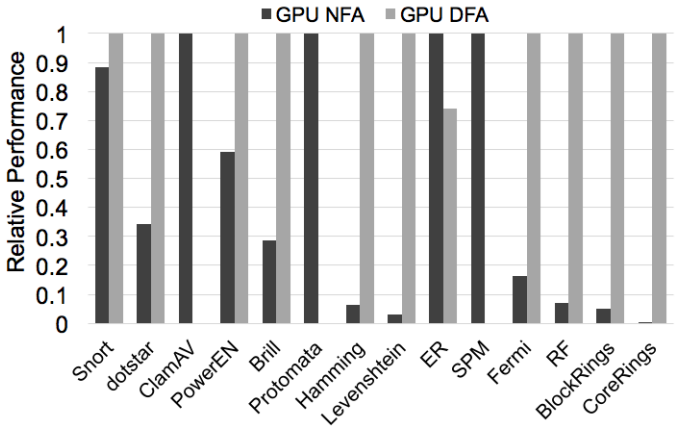

Fig. 6. Relative performance of NFA and DFA engines over all benchmarks in ANMLZoo. DFAs for ClamAV, Protomata, and SPM were too expensive to construct due to space or time costs.

per symbol, independent of the automaton and input stream, while the NFA-based processing engine follows a number of state transitions.

Unsurprisingly, Figure 6 shows that the DFA engine–when DFAs are able to be created–is the best solution for every benchmark with the exception of ER. This is due to the relative simplicity of the DFA kernel, and the reduced number of total instructions required to compute the automata. We compared profiling information gathered by NVIDIA's profiling tool *nvprof* on the Levenshtein automata. The NFA kernel executed over $5,700$ times more control flow instructions than the equivalent DFA kernel, and 43 times more memory instructions per input symbol.

In some applications (Snort), DFAs do not give significantly better performance compared to NFAs in iNFAnt2. This is because very few NFAs can be combined into single DFAs. Specifically, for the ER benchmark, the large number of

required DFAs causes the iNFAnt2 DFA engine to perform worse than the iNFAnt2 NFA engine.

IX. Mesh Scaling and AP Fabric Utilization

Mesh automata, such as the Hamming and Levenshtein automata, score input strings by positionally keeping track of input mismatches with an encoded string.

Hamming-distance automata have been shown to help accelerate both DNA and protein [10] motif search algorithms. These automata use a simple kernel–match or mismatch–to positionally keep track of the number of mismatches between the input and encoded string using automata states. A mismatch will force a transition to a new row of states that represent mismatches one greater than the previous row. In practice, it is usually only important to keep track of mismatches up to a particular score threshold, and so rows can be pruned from Hamming distance automata to decrease unnecessary states and computation. Because Hamming distance automata only ever considers the match or mismatch kernel, the fan-out and fan-in of any individual state is always less than or equal to 2, no matter the length of the input string, or the number of mismatches the automaton is programmed to compute.

Levenshtein automata use a more complex kernel to keep track of differences between an encoded string and an input string. While Hamming distance only considered matches and mismatches, Levenshtein automata additionally keep track of possible insertions in the input string and deletions from the encoded string, ultimately scoring the number of "edits" (edit distance) required to transform one string to the other up to a certain edit threshold. Because the Levenshtein automata must account for any number of deletions up to the threshold, the maximum fan-out and fan-in of any individual state grows linearly with the size of the threshold.

This increase in the connectivity is not problematic for von Neumann-based automata processing engines, where arbitrary automata networks can be easily stored in memory. However, high connectivity can be problematic for spatial architectures that rely on a reconfigurable routing matrix to lay out all possible datapaths in the automata networks. To show impacts of connectivity in mesh automata on spatial architectures, we vary both encoded string length and a score threshold for Hamming and Levenshtein automata. We then compile the designs for the AP and measure their on-chip routing utilization. Figure 7 plots the resulting routing complexity vs. encoded string length for ten different automata.

Hamming automata are relatively insensitive to increases in both dimensions–the encoded string length and the score threshold–of the automata. This reflects the constant fan-in/fan-out per match/mismatch kernel. While the number of these kernels increases, their routing complexity remains relatively flat.

In contrast, the routing complexity of Levenshtein automata is highly sensitive to changes in the score threshold. This is due to the linear scaling of fan-in/fan-out to account for a number of deletions up to the score threshold. Figure 7 shows that a

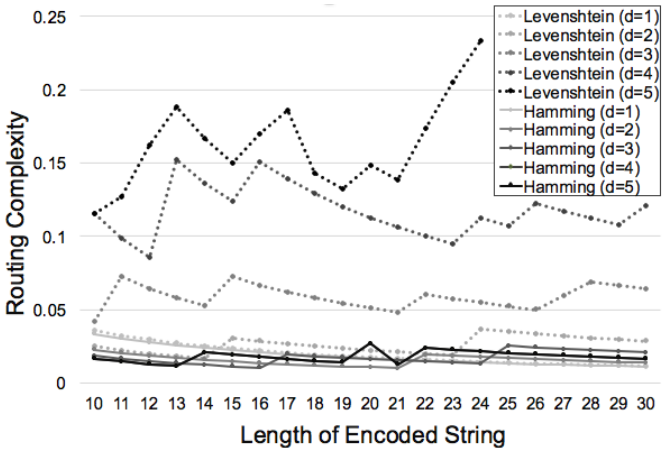

Fig. 7. Hamming automata have a constant fan-in/fan-out per STE and therefore have relatively low routing complexity that is not impacted by the dimension of the mesh. The node degree of Levenshtein automata grows linearly in the size of the encoded edit distance threshold, therefore routing complexity is very sensitive to this dimension.

Levenshtein automaton with length 24 and score threshold 5 takes about 4 times more routing resources than a Levenshtein automaton with a score threshold of 3, and 10 times more routing resources than a Hamming automaton with a score threshold of 5 and encoded string length of 24. Levenshtein automata with a score threshold of 5, and length greater than 24 fail to route on the current generation of the AP architecture and place-and-route tools.

X. Cross-Architecture Application Evaluation

We evaluate the performance of each baseline NFA automata-processing engine over all standard-candle ANML-Zoo benchmarks. Results are shown in Figure 8. While this does not represent an absolute ranking of the performance of each architecture, it does represent the current state of the baseline evaluation engines included in ANMLZoo as compared to the AP. We present the estimated performance of the first generation AP hardware [3].

VASim tends to perform worse on the XeonPhi than the i7 CPU. While the XeonPhi has many more individual cores (57 rather than 6), the CPU's large L3 cache is more important than parallel cores for accelerating the VASim algorithm. A XeonPhi-specific baseline automata-engine to take advantage of its vector units, similar to iNFAnt2, is therefore desirable for a more fair evaluation of these architectures. The XeonPhi performs better than the CPU on the PowerEN and Levenshtein benchmarks, indicating that the VASim algorithm is more bottlenecked by parallelism, and less bottlenecked by rule-transition latency, for these benchmarks.

The GPU NFA engine performs better than our baseline CPU engine in 10 out of 14 benchmarks, indicating that the GPU's massively parallel resources are important for parallel automata-processing. However, the CPU engine outperforms the GPU engine on Brill, ClamAV, BlockRings, and CoreRings. This is most likely due to both a small active set and visited set in these applications, allowing for more ideal cache behavior on the CPU.

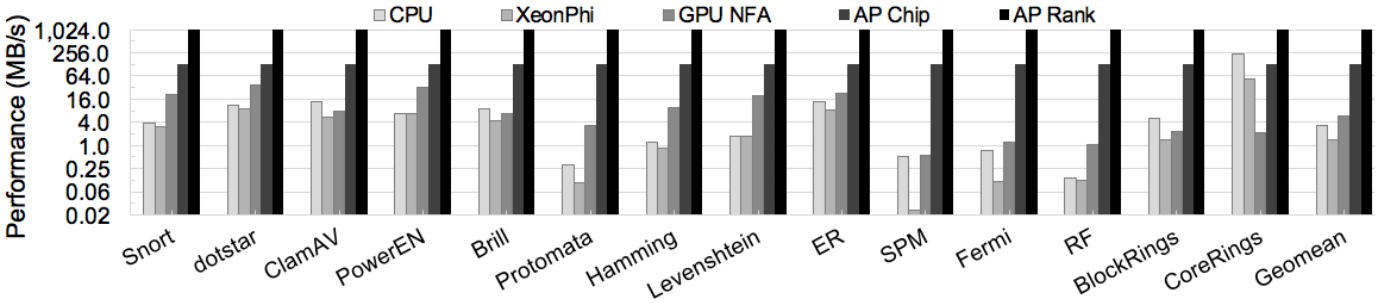

Fig. 8. Performance of all standard candle benchmarks on each available architecture. AP performance is estimated to be 133MB/s, however, we expect to see performance degradations due to output reporting constraints when using the real hardware. We expect to include performance of real hardware in the final version of the paper. Because each ANMLZoo *standard candle* automata maxes out an AP chip, it is easy and fair to estimate the performance of an AP Rank (8 chips) as 8 times the performance of an individual AP chip.

The simulated AP's chip's data-flow style architecture generally outperforms all von-Neumann-style NFA automata engines. One notable exception is the synthetic CoreRings benchmark, where VASim is capable of achieving upwards of $230MB/s$. CoreRings has an extremely low activity relative to the number of states. Section VI showed that the CPU can achieve $50MB/s$ per core per active state. Thus, the impressive performance for the full multi-threaded version of CoreRings is unsurprising. This motivates heterogeneous automata-processing engines and architectures with both von Neumann and data-flow engines operating on portions of automata that best suit them.

Because each standard-candle automata benchmark maximizes the resources of an individual AP chip, we can easily, and fairly, estimate performance of other deployment scenarios of AP chips. Because each AP chip can operate on a separate, parallel portion of the input stream, AP performance is expected to scale perfectly linearly in the real hardware. As an example of this feature of the benchmark suite, we also include estimated AP Rank performance in Figure 8.

For each architecture, we explored both dimensions of automata parallelism to attempt find the best-performing configuration. However, exploring every possible configuration was not feasible and so optimal performance is not guaranteed. Data-flow architectures such as the AP do not have this dimension of complexity, and thus guarantee deterministic performance with *no* dynamic performance tuning. This property is extremely desirable for real-time applications such as deep-packet inspection and on-line machine learning.

Moving forward, new algorithms, automata-representations, automata-processing engines and new automata-processing architectures can be easily evaluated and compared using ANMLZoo. We encourage researchers to contribute any of these components as they are developed, so that new research can fairly and easily compare to prior work.

XI. Conclusions and Future Work

This paper presented ANMLZoo, a diverse benchmark suite of finite automata for easy and fair evaluation of automata processing engines. ANMLZoo benchmarks are quantitatively diverse in both static structure and dynamic behavior and represent a wide range of well known and new applications for automata processing. ANMLZoo also acts as a repository for automata-processing engines for a wide range of new architectures, allowing easy access to prior work for quick and fair evaluations of new automata-processing engines and old engines on new hardware architectures.

Using ANMLZoo, we were able to show bottlenecks in von Neumann computer architectures for automata processing. CPUs perform well when the average activity in an automata is small, and the average number of visited automata states is small. GPUs and Intel's XeonPhi can perform well, but exploiting the computational power of SIMD units to compute the irregular parallelism of automata is difficult. Thus, these architectures generally benefit from input stream parallelism, rather than automata-level parallelism. The AP is the fastest automata-processing hardware but its capacity is very sensitive to automata topography and cannot place-and-route automata states with large fan-in/out.

Future work may include more in-depth analysis of micro-architectural bottlenecks to automata processing within individual hardware architectures. Future work may also explore automata processing engines on other architectures such as FPGAs and changes in performance of automata applications over multiple generations of CPU, GPU, FPGA, and AP architectures. Future work may also evaluate relative power efficiency, instead of pure performance, of each architecture.

XII. Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank contributors of ANMLZoo benchmarks including public sources, Michela Becchi, and Micron Technologies. This work was partly funded by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, Achievement Rewards for College Scientists (ARCS), Micron Technologies, and NSF grant no. EF-1124931.

References

[1] M. Roesch, "Snort: Lightweight intrusion detection for networks.," in *Proceedings of the USENIX Systems Administration Conference*, LISA '99, 1999.

[2] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: the unified automata processor," in *Proceedings of the ACM*

International Symposium on Microarchitecture, Micro '15, pp. 533–545, 2015.

[3] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[4] "ClamAV." Available at https://www.clamav.net/.

[5] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, pp. 30–39, 2009.

[6] K. Atasu, F. Doerfler, J. van Lunteren, and C. Hagleitner, "Hardware-accelerated regular expression matching with overlap handling on ibm poweren processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, pp. 1254–1265, 2013.

[7] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '08, pp. 79–89, IEEE, 2008.

[8] C. Bo, K. Wang, J. J. Fox, and K. Skadron, "Entity resolution acceleration using Micron's Automata Processor," *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[9] 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*, CF '16, 2016.

[10] I. Roy and S. Aluru, "Finding motifs in biological sequences using the Micron Automata Processor," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '14, pp. 415–424, 2014.

[11] I. Roy, *Algorithmic Techniques for the Micron Automata Processor*. PhD thesis, Georgia Institute of Technology, 2015.

[12] T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, and G. Robins, "Nondeterministic finite automata in hardware - the case of the Levenshtein automaton," *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA*, 2015.

[13] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda, "Using the Automata Processor for fast pattern recognition in high energy physics experimentsa proof of concept," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 832, pp. 219 – 230, 2016.

[14] T. Tracy II, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the Automata Processor," *ISC High Performance Computing*, 2016.

[15] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron, "Generating efficient and high-quality pseudo-random behavior on Micron's Automata Processor," in *Procceedings of the IEEE International Conference on Computer Design, to appear*, ICCD '16, 2016.

[16] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the Micron Automata Processor," in *Proceedings of the IEEE International Conference on Semantic Computing*, ICSC '15, pp. 236–239, 2015.

[17] M. Becchi, *Data structures, algorithms and architectures for efficient regular expression evaluation*. PhD thesis, Washington University in St. Louis, 2009.

[18] Google, "Re2." https://github.com/google/re2.

[19] Intel, "Hyperscan." https://github.com/01org/hyperscan.

[20] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of Architectures for Networking and Communications Systems*, ANCS '08, pp. 50–59, 2008.

[21] 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.

[22] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Parallelization and characterization of pattern matching using gpus," in *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '11, pp. 216–225, 2011.

[23] X. Yu and M. Becchi, "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, p. 18, 2013.

[24] X. Yu and M. Becchi, "Exploring different automata representations for efficient regular expression matching on GPUs," in *ACM SIGPLAN Notices*, vol. 48, pp. 287–288, 2013.

[25] "Micron's AP SDK." http://micronautomata.com/.

[26] K. Angstadt, W. Weimer, and K. Skadron, "RAPID programming of pattern-recognition processors," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pp. 593–605, 2016.

[27] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 529–542, 2014.

[28] S. Banerji, "Computer simulation codes for the Quine-McCluskey method of logic minimization," *arXiv preprint arXiv:1404.3349*, 2014.