Automata Processing in Reconfigurable Architectures: In-the-Cloud Deployment, Cross-Platform Evaluation, and Fast Symbol-Only Reconfiguration

CHUNKUN BO, VINH DANG, TED XIE, JACK WADDEN, MIRCEA STAN, and KEVIN SKADRON, University of Virginia, USA

We present a general automata processing framework on FPGAs, which generates an RTL kernel for automata processing together with an AXI and PCIe based I/O circuitry. We implement the framework on both local nodes and cloud platforms (Amazon AWS and Nimbix) with novel features. A full performance comparison of the proposed framework is conducted against state-of-the-art automata processing engines on CPUs, GPUs, and Micron's Automata Processor using the ANMLZoo benchmark suite and some real-world datasets. Results show that FPGAs enable extremely high-throughput automata processing compared to von Neumann architectures. We also collect the resource utilization and power consumption on the two cloud platforms, and find that the I/O circuitry consumes most of the hardware resources and power. Furthermore, we propose a fast, symbol-only reconfiguration mechanism based on the framework for large pattern sets that cannot fit on a single device and need to be partitioned. The proposed method supports multiple passes of the input stream and reduces the re-compilation cost from hours to seconds.

CCS Concepts: • **Computer systems organization** → **Architectures**;

Additional Key Words and Phrases: Automata processing, FPGA, acceleration, cloud-computing

ACM Reference format:

Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. 2019. Automata Processing in Reconfigurable Architectures: In-the-Cloud Deployment, Cross-Platform Evaluation, and Fast Symbol-Only Reconfiguration. *ACM Trans. Reconfigurable Technol. Syst.* 12, 2, Article 9 (May 2019), 25 pages. <https://doi.org/10.1145/3314576>

1 INTRODUCTION

Finite automata have shown their capability in a variety of pattern matching applications, especially when inexact matching is needed (Dlugosch et al. 2014; Zhou et al. 2015; Wang et al. 2015; Bo et al. 2016; Sadredini et al. 2018). Prior work focused on their applicability for regular-expression-specific applications such as network security and natural language processing. For instance, in the case of network security, regular expressions used for deep packet inspection are represented in memory as their equivalent nondeterministic finite automata; RegEx engines such as Intel's

Chunkun Bo and Vinh Dang contributed equally to this work.

This work was supported in part by NSF grant no. CCF-1629450, a grant from Xilinx, and support from CRISP, one of six centers of JUMP, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Authors' address: C. Bo, V. Dang, T. Xie, J. Wadden, M. Stan, and K. Skadron, University of Virginia, 85 Engineer's Way, Charlottesville, VA, 22903, USA; emails: {chunkun, vinhdang, ted.xie, wadden, mircea, skadron}@virginia.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1936-7406/2019/05-ART9 \$15.00

<https://doi.org/10.1145/3314576>

HyperScan (Intel 2018) perform automata transformations to maintain excellent performance in terms of both runtime and memory utilization. Recent research has demonstrated that many modern applications such as bioinformatics, data mining, machine learning, and so forth, which are difficult or impossible to formulate with regular expressions, can also benefit from automata-based approaches (e.g., Wang et al. (2015, 2016a), Zhou et al. (2015), Bo et al. (2015b, 2018), and Sadredini et al. (2018)). In general, the automata-based approach consists of two steps: (i) designing finite automata, either in deterministic finite automaton (DFA) form or nondeterministic finite automaton (NFA) form, to perform pattern matching, and (ii) matching the automata against input string(s). However, this could be computationally intensive, especially for large input streams or complex automata, which requires high memory bandwidth with low-latency access to compute efficiently on von Neumann architectures. Previous research focused on accelerating automata-based computation on multi-core CPUs, as well as GPUs, where data-level parallelism can help explore many possible automata transitions simultaneously (Dang 2017a, 2017b; Intel 2018). However, the performance is still bottlenecked by the random memory access behavior (Nourian et al. 2017).

On the other hand, spatial architectures, such as FPGAs, Micron's Automata Processor (AP) (Dlugosch et al. 2014), and other potential specialized hardware, can be utilized to efficiently process a large number of automata transitions in parallel, by laying out automata graphs directly in hardware, instead of storing states and transition rules in memory, as in von Neumann architectures. Specifically, prior work (Yang and Prasanna 2012; Dlugosch et al. 2014) have shown that digital circuits enable a one-to-one spatial mapping between automata states and circuit components such as registers, logic gates, and wires. In order to accommodate a range of automata-based applications, reconfigurable platforms such as FPGAs and APs are ideal for this purpose. FPGAs provide massive reconfigurable hardware resources (registers, lookup tables, etc.), which can be exploited for mapping state-transition elements. The AP is another reconfigurable fabric specially designed to process NFAs in parallel on DRAM (Dlugosch et al. 2014).

With the fast development of cloud computing and the popularity of FPGAs in the post Moore's law era, many cloud service providers started to provide compute instances with FPGAs. Xilinx has partnered with Nimble (Nimble 2017) and Amazon (AWS 2017) to bring reconfigurable acceleration on FPGAs to the cloud, which allows developers to build, test, and deploy their codes in the cloud-based environment. Users can use Vivado, Vivado HLS, and SDAccel in such cloud platforms without configuring hardware or purchasing licenses. This allows users to "rent" FPGAs (which are often quite expensive) on an as-needed basis, and avoids the need to pay for expensive licenses for the development environment.

Therefore, we aim to develop a high-throughput and scalable engine for enabling direct automata processing on FPGAs on both local computational nodes and cloud platforms, as opposed to prior work which mainly focused on processing regular expressions. To achieve this goal, we previously developed REAPR (Reconfigurable Engine for Automata PRocessing), a flexible, "end-to-end," and parameterizable framework on local computational nodes that generates RTL codes to process finite automata on FPGAs, and tested the feasibility and correctness (Xie et al. 2017). It translates automata representation directly to RTL and adds the appropriate I/O capability to implement the full, end-to-end applications with limited numbers of reports (the max is 8). A *report* in the article refers to the event when a pattern finds a match. We believe that this is the first attempt to create a general direct automata processing framework on FPGAs, and use AXI and PCIe to enable CPU-FPGA communication.

In this article, we extend the previous work to support more reports to accommodate real-world applications, automate the workflow, port the framework to cloud platforms, and further improve the performance. We investigate the "end-to-end" (including both the kernel execution and the I/O cost) comparison of the computational throughput between FPGAs and state-of-the-art automata

processing engines on CPUs, GPUs, and Micron's AP. The performance of REAPR is evaluated in two cloud-based FPGA environments, Nimble Cloud and Amazon EC2 F1, using the ANMLZoo benchmark suite. The ANMLZoo benchmark suite (Wadden et al. 2017) contains a wide variety of automata-related applications (not constrained to regular expressions). To the best of our knowledge, this work is the first effort to provide a complete reconfigurable automata processing engine on FPGAs in the cloud, and to provide a full analysis of various automata engines on a diverse range of workloads other than simple regular expressions. Furthermore, we collect the results of the hardware resource utilization and the power consumption, and find the I/O circuitry consumes the most of the resource and the power, which could be further improved for future automata processing hardware.

One challenge when using spatial architectures is that for large pattern sets or problem sizes, not all automata can fit on one device. This requires a method to partition the automata and support multiple passes of the input stream with fast reconfiguration for each partition. The compilation time on FPGAs could take up to tens of hours, thus compiling all partitions is very time-consuming for large pattern sets. To solve this problem, we propose a fast, symbol-only reconfiguration mechanism for REAPR. The proposed method designs a general automaton structure for the specific application and uses the same structure for all patterns. It only reconfigures the symbols stored in each state, instead of re-compiling automata graphs for every new partition. The proposed method helps process much larger pattern sets, and dramatically reduces the total re-compilation overhead (from hours to seconds).

In summary, we make the following contributions.

- (1) REAPR is an end-to-end framework with I/O circuitry that we previously developed for automata processing on FPGAs with a limited number of reports (Xie et al. 2017). We extend the original framework in this article to support a much larger number of reports (making it feasible and practical for a wider range of real-world applications), automate the whole workflow on local nodes to make it more user-friendly, port the framework to cloud platforms to make it more accessible to users, and further improve the performance.
- (2) We evaluate the performance against state-of-the-art automata processing engines on CPUs, GPUs, and Micron's AP, which is the first full analysis of different automata processing engines using a wide variety of applications and computing platforms. Results show that spatial architectures achieve promising results over von Neumann architectures. We also collect resource utilization and the power consumption results, and find that the I/O circuitry is the component consuming most resources. This provides insights for future automata processing architectures.
- (3) We propose a fast, symbol-only reconfiguration mechanism that dramatically reduces the re-compilation overhead for large pattern sets in certain applications.

2 PRIOR WORK

2.1 CPU Automata Engines

In a nondeterministic finite automaton (NFA), symbols from the input stream are broadcast to each state simultaneously, and each state can connect to multiple other states, each of which may or may not be activated depending on whether a given state matches the incoming symbol. For each symbol, an NFA engine must determine the next set of activated states, which involves a scan of the adjacency lists of all states in the current activated set. On a sequential processor, this is linear time relative to the number of active states. In the worst case, the adjacency list may contain nearly all of the states in the automaton; therefore, the runtime on a CPU for simulating an m -state automaton on n symbols is $O(n \cdot m)$. NFA processing on the CPU is additionally hampered by the so-called

“memory wall,” due to the NFA’s pointer-chasing execution model, and therefore it is desirable to drastically reduce the number of memory accesses per input item. In order to mask memory latency, state-of-the-art NFA engines such as Intel HyperScan (Intel 2018) perform SIMD vector operations to execute as many state transitions as possible for a given memory transaction. Even so, such optimizations still cannot escape the fact that sequential von Neumann architectures are *fundamentally* ill-suited for such workloads.

In order to improve the runtime complexity of automata traversals, some regular expression engines transform the NFA into its equivalent deterministic finite automata (DFA). A DFA has at most one state active for any given symbol cycle and is functionally equivalent to an NFA; this is achieved through a process known as *subset construction*, which involves enumerating all possible paths through an NFA. Converting an NFA to a DFA has the benefit of reducing the runtime to $O(n)$ for n input symbols (note that now the runtime is independent of automaton size) and only requires one memory access per input symbol, but this NFA-to-DFA conversion frequently causes an exponential increase in the number of states necessary; this phenomenon is often referred to as *state explosion* (Becci and Crowley 2007). Subset construction for large automata thus typically incurs a huge memory footprint, which may actually cause performance degradation due to the memory overhead in von Neumann machines. However, a spatial architecture completely avoids the need for DFA construction and achieves the full efficiency of NFAs.

Prior work by Becchi (Becci and Crowley 2007) attempted to leverage the best of both types of finite automata (the spatial density of NFA and the temporal density of DFA). By intercepting the subset construction algorithm and not expanding paths that would result in a state explosion, Becchi achieved 98–99% reduction in memory capacity requirement and up to 10× reduction in memory transactions.

2.2 Automata Processing on FPGAs

2.2.1 NFA. Past implementations of NFAs on FPGA (Sidhu and Prosanna 2001; Yang and Prasanna 2012) focused on synthesizing *only* regular expression matching circuits for applications such as antivirus file scanning and network intrusion detection. REAPR extends prior work by focusing on a more diverse set of finite automata to address the fact that the workload for automata processing is much richer and more diverse than regular expressions. We extend the underlying approaches for NFA RTL generation from prior work, and adapt it for other NFA applications. Details of REAPR will be presented in Section 3.

2.2.2 DFA. Several efforts (Hieu and Tran 2013) in accelerating automata processing with FPGAs use Aho-Corasick DFAs as the underlying data structures. A major motivator behind this design choice is the ease of translation between a DFA and a simple transition table, which can be easily implemented using BRAM. One benefit of this approach is that BRAM contents can be hot-swapped easily, whereas a spatial design requires a full recompilation to realize even a single change. Because DFAs do not exploit the native bit-level parallelism in digital hardware and are much better suited to memory-bound CPU architectures, REAPR focuses on the spatial implementation of NFAs. Note that REAPR can support the placement of the automata in logic and/or BRAM.

2.3 GPU Automata Processing and Other Architectures

DFAGE is a prototype framework for running DFA-based matching on NVIDIA CUDA-enabled GPU cards (Dang 2017a). iNFAnt2, an optimized version of iNFAnt (Cascarano et al. 2010), is a prototype framework for NFA-based automata processing on NVIDIA CUDA-enabled GPU cards (Wadden et al. 2017). iNFAnt2 can process large and complex NFAs with high throughput and low memory consumption (Dang 2017b).

Micron's AP is an efficient semiconductor architecture for parallel automata processing (Dlugosch et al. 2014). It directly implements NFAs in hardware, matching complex regular expressions and other types of automata that cannot be conveniently expressed as regular expressions. Following work focused on providing a more convenient programming model for the AP (Angstadt et al. 2016; Yu et al. 2017).

Several past efforts have proposed modifications to existing von Neumann architectures to specifically improve the performance of automata processing workloads. HARE (Hardware Accelerator for Regular Expressions) (Gogte et al. 2016) uses an array of parallel modified RISC processors to emulate the Aho-Corasick DFA representation of regular expression rulesets. The Unified Automata Processor (UAP) (Fang et al. 2015) also used an array of parallel processors to execute automata transitions and can emulate *any* automaton, not just Aho-Corasick. However, because these works are (1) not FPGA-centric (both are ASICs), (2) based on the von Neumann model and not spatially distributed like REAPR, and (3) confined to a limited set of just regular expressions (as opposed to general automata applications), we do not directly compare to them.

3 REAPR DESIGN

3.1 Automata Processing RTL Generation

FPGAs offer a large number of resources and can be configured to process a variety of computations. We adopt a similar approach as in prior work that exploited the reconfigurable nature of FPGAs to lay out regular expressions in reconfigurable logic fabrics using BRAM storing the characters (Yang and Prasanna 2012). However, our work focuses on the hardware synthesis of NFAs. The NFA's highly parallel operation of matching one single datum for many states maps well to the abundant parallelism offered by spatial architectures such as the FPGA. While DFAs can also be implemented on spatial architectures, the argument is less compelling because (1) DFAs only need to perform a single symbol match per cycle, and as previously mentioned are better suited for von Neumann architectures, and (2) DFAs often require a huge area on the FPGA device, due to the potential exponential increase in the number of states relative to the equivalent NFA.

Merging the transition logic with the state transforms a traditional NFA into a homogeneous finite automaton (Caron and Ziadi 2000), which maps more naturally to reconfigurable hardware, while preserving the benefits of NFAs. The combined state-transition structure is referred to as a state-transition element (STE). The STEs are mapped to FPGA registers and look-up tables/BRAMs, as shown in Figure 1. The register is called the activation state register and is used to store the status indicating whether a state is activated. Activation registers of all starting states will always be set as "high" in order to stimulate the workflow (e.g., the "a" and "b" states in the figure). Each STE's transition logic is one-hot encoded as a 1×256 memory column containing the "character set" (a 256-entry column supports 8-bit input symbols) and is AND'ed with the activation state register. The input symbol is sent to all STEs. Once the input symbol matches the characters stored in the memory column, it will activate the top wire of the AND gate. In the figure, if the input symbol is "a," because the activation state of the blue state is already set, it will trigger the output of the blue AND gate. The output of AND gates is called the "enable" signal, which is then OR'ed with other "enable" signals from other STEs. The "enable" signal will activate the next STEs in the automaton structure. In this example, because the blue state is connected to the green state, the "enable" signal from the blue AND gate will activate the top wire of the OR gate and then set the activation state register of the green state as "high." If the next input symbol matches with "c" or "d" stored in the green state, it will activate the top wire of the green AND gate and will generate a report. With this design, the AND gate of an STE will only output "1" when its activation state is driven high by other STEs or set as starting states, and the current input symbol is accepted in its character set. In

Fig. 1. Mapping automata states to registers and look-up tables (“logic”).

such a way, the whole flow will be triggered by each new input symbol. If an STE is configured as a reporting state, its output will be stored in the reporting vectors. Hence, for each input symbol, all reporting states store its output bits in the corresponding reporting vector.

In REAPR, the “character sets” can be represented in hardware using either lookup tables (LUTs) or BRAMs.

LUT-Based Design: Each state must accept a range of characters corresponding to outgoing transitions in a canonical finite automaton. LUTs are well-suited for this task, due to their proximity to the state registers within a Configurable Logic Block (CLB); a LUT-based flow does not need to use as much long-distance wiring to connect to a far-away BRAM.

BRAM-Based Design: The main disadvantage of using LUTs for storing the character class is the long compilation time; FPGA compilers aggressively minimize logic for LUT designs, which drastically increases compiler effort. Using BRAMs for transition logic circumvents the expensive optimization step and therefore decreases compile time. The AP’s approach to generating NFAs on hardware is similar to the BRAM design, except that the AP stores the 256-bit columns into DRAM banks instead of into BRAMs on FPGAs. This leads to a high state density due to the higher density of DRAM compared to SRAM.

In general, LUTs are better suited for storing “character sets,” because it generally leads to a higher clock frequency because of the shorter distance to the state registers. Therefore, we opt to use the LUT-based method to achieve better performance when evaluating the original REAPR performance. However, when evaluating the performance with the symbol-only reconfiguration feature which needs to write character sets multiple times, we use the BRAM-based design to fully utilize the fast write speed of BRAM. The details of symbol-only reconfiguration will be discussed in Section 3.6 and the results will be presented in Section 5.5.

3.2 I/O Circuitry Integration

A major contribution of this work is the inclusion of I/O circuitry over PCIe and Advanced Extensible Interface (AXI) Interconnect, making REAPR the first work to offer an end-to-end automata

Fig. 2. SDAccel-based approach of AXI and PCIe transactions for automata processing kernel.

processing engine on FPGAs. We adopt a high-level synthesis-centric approach by designing the I/O interface using Vivado HLS and modifying the generated Verilog codes to integrate the real automata processing kernel. We then use Xilinx SDAccel (Inc 2018) to generate AXI Interconnect and PCIe circuitry for the kernel. Evaluating automata engines on real hardware with I/O overhead allows us to obtain true performance results, compared to simulation results in prior work. This approach takes advantage of the productivity of Vivado HLS, enabling developers to describe the I/O kernel in high-level languages instead of using RTL from scratch. The SDAccel-based approach of REAPR with I/O is shown in Figure 2. The execution of REAPR in the SDAccel environment follows a classic offload model. The host CPU sends an input string to the on-board DDR memory; then the FPGA runs the desired automata against the input string while recording matching results (indicating that a pattern of interest has been recognized, i.e., an automata reporting state) to the on-board DDR memory; and finally, the host CPU retrieves these reports. The infrastructure IP provided as part of the FPGA device handles the communication to the host CPU over the PCIe Interconnect and the communication between the automata processing kernel and the on-board DDR memory over the AXI Interconnect.

To ease the integration stage, we design an I/O kernel performing a simple, templated memory-copying operation, which can then be replaced by the automata processing RTL kernel. A simplified code snippet of the dummy I/O kernel is shown in Figure 3. The input symbols are copied from the input buffer (stored in the on-board DDR memory) to the output buffer (stored in the on-board DDR memory) after being added to 0xFA. Loop L0, which performs these operations, is fully pipelined. To maximize the I/O performance, the input buffer and output buffer are located in separate DDR banks. Within loop L0, loop L00 is unrolled to account for concurrently writing the reports to the output buffer. In the corresponding Verilog codes, we search for the dummy addition and substitute the addition operation with the automata RTL kernel. Specifically, the inputs of the automata module are connected to the signals representing the loadAB variable and the outputs of the automata module are connected to the signals representing the resultAB variable. The newly

```

#define FACTOR GroupsOf8bits // GroupsOf8bits = NumberOfReportingStates/8
typedef struct {
    unsigned char data[FACTOR];
} dout_t;
void bandwidth(dout_t *output_port, unsigned char *input_port){
    #pragma HLS DATA_PACK variable=output_port
    L0:for (int i = 0; i < DATA_SIZE; i++) { // read one byte at a time
        #pragma HLS pipeline II=1
        unsigned char loadAB = input_port[i];
        unsigned char resultAB = 0xFA + loadAB;
        L00:for(int k=0; k<FACTOR; k++){
            #pragma HLS UNROLL
            output_port[i].data[k] = resultAB;
        }
    }
}

```

Fig. 3. I/O kernel with dummy computation.

integrated kernel is then compiled using Xilinx SDAccel to generate the hardware binary to be run on FPGA boards.

3.3 Reporting Architecture

“Report” in this article refers to a match found in an automaton. A reporting architecture is included in REAPR as shown in Figure 1. At each cycle (i.e., for each input symbol processed by REAPR), outputs from all reporting STEs (0’s and 1’s) are stored directly in the output buffer in the on-board DDR memory. This requires a reporting vector of which the length equals to the number of reporting states, and the number of reporting vectors equals the count of the input symbol. Therefore, the total size of a result block to be transferred back to the CPU is (length of the input)×(number of reports). For example, if the input is 10MB and there are 1,000 reporting states, the total size of the data to be transferred will be 10GB. The I/O kernel in the earlier work (Xie et al. 2017) only supported an 8-bit output port, thus only supporting eight reporting STEs. However, in most real-world automata-related applications, the number of reporting STEs is much larger. In this work, we support much larger report numbers by defining the output port bitwidths according to the actual number of reporting STEs in the application. We use a “struct” data type, containing an array of groups of 8 bits (because the minimum word width of SDAccel is 8), for the output port. The bitwidth is rounded to 32, 64, 128, 256, or 512, which corresponds to the memory interface bitwidth supported by SDAccel. If the number of reporting STEs exceeds 512, more output ports (i.e., output buffers) will be needed by the I/O kernel. Finally, all reports are offloaded to the host CPU for post-processing after the automata processing kernel completes on the FPGA. This massive amount of data transfer has a significant overhead on the overall throughput. The details will be discussed in Section 5. One potential solution is presented in Wadden et al. (2018a) and more discussion will be presented in Section 5.6.1.

3.4 Automated Workflow

When working on the original REAPR framework, we find that the I/O circuitry integration requires a lot of manual effort and is prone to errors when applying it to different applications

Fig. 4. Automated workflow of REAPR.

or different datasets. This process ranges from designing the I/O template to accommodating the desired number of reports, to hooking the automata processing kernel to the I/O body inside the complex Vivado HLS codes. In order to mitigate this burden and provide a user-friendly tool, we developed a workflow (as shown in Figure 4) that fully automates the whole framework. The new workflow is open source and is available on Github (Xie et al. 2018). The entry file of the automated workflow is “rtl.st.” Users only need to set certain parameters in the file to use the framework. The automated workflow first takes a file that describes the automata graphs. The automata description file can be either (a) an ANML file: an XML-based description language representing automata, developed for the Micron’s Automata Processor (Micron 2015); or (b) a MNRL file: a JSON-based, open-source description language to describe automata (Angstadt et al. 2018). Compared with ANML, MNRL can be easily extended to support new features, such as up-down

Fig. 5. Original flow for large problem sizes.

counters. Based on the automata description file, the corresponding generator will generate the RTL module, host codes, and the codes for transferring input and output for the dummy kernel (Steps 2, 3, and 4). We then use Vivado_hls to generate the I/O template for the dummy kernel as shown in Figure 3. Step 5 and Step 6 modify the I/O template and hook the real automata kernel generated in Step 2 to the template. In the last phase, the automated workflow will generate the “Makefile” and compile the newly integrated design using SDAccel. This will produce the executable and the binary for the FPGA. With the new automated workflow, all the implementation details are hidden from users, making the framework more user-friendly. Users now can focus on preparing automata description files for their applications, and only need to set certain parameters in the entry file (rtl.sh) according to their FPGA platforms to use the framework.

3.5 Supporting Multiple Streams

In this work, we also extend the original REAPR to support multiple input streams. The input memory bandwidth in previous REAPR’s I/O design is not fully utilized, since the input pointer is defined as an 8-bit data type. Hence, parallel input streams can be handled with this I/O design if the FPGA device has sufficient hardware resources to replicate the automata processing kernel and there are enough memory bandwidth and enough memory ports for writing output reports from these streams. Users can either process the different chunks of the input stream for the same set of automata, or process different input streams for different automata sets. Supporting multiple streams helps to further increase the throughput of REAPR. The performance results will be presented in Section 5.

3.6 Symbol-Only Reconfiguration

Although we have extended REAPR to support a larger number of patterns, it still may not be large enough for some large-scale applications to fit all patterns on one FPGA board. This requires a method to partition the automata and support multiple passes with fast reconfiguration for each partition. For example, for the Next Generation Sequencing (NGS) problem, there could be millions of long (over 100 base pairs) sequences to be matched. Many partitions of such huge pattern set will be needed and the cost of compiling all these partitions will be very high because of the placement and routing phase for each compilation, which jeopardizes the speedups brought by the fast kernel execution. In the original framework, we need to compile the patterns every time we process a new partition as shown in Figure 5. To solve this problem, we propose a fast, symbol-only reconfiguration mechanism for REAPR in this section as shown in Figure 6. Each column in

Fig. 6. Symbol-only reconfiguration.

Fig. 7. Example of general automata structure for SNAC ER problem.

the figure stands for a partition and there are k patterns in each partition. The proposed method needs to design a general automata structure (parallelogram in the figure) that can be shared by all patterns for the specific application.

In this article, we present one example showing how to design the general automata structure. Bo et al. design a general automaton structure for recognizing the same person's names with variances in the SNAC (Social Network and Archival Context) database in Bo et al. (2015a, 2016). They first extract common name formats and then come out with a design for these formats instead of individual names. The design is shown in Figure 7. The left half is the automata structure for the macros on the right half. The right half is the final automaton that can recognize the first name and the family name. All the names in the database can share the same structure on the right. This is one example of a general automaton structure and the general automata structure varies from one application to another. The prior work by Teubner et al. (2012) and Moussalli et al. (2014) propose a similar idea for some database applications, but their ideas are limited to the specific

applications and focus on how to design the automaton structure and map the specific structure to FPGA resources. The proposed method in this article has a broader scope for applications involved with automata processing and can accommodate the applications discussed in the above papers.

To use the new mechanism, we first need to compile the general automata structure, but this is a one-time overhead compared to Figure 5. When a new partition comes, users only need to reconfigure the symbols (write new character sets for the corresponding partition) stored in the states instead of compiling new structures for automata graphs in the new partition. For each state, there are 256 bits (0's or 1's) representing 256 different characters, and the character set file size is very small even when storing a large number of patterns (in the order of KB for hundreds of thousands of states). Writing a new character set is much faster than placing and routing new automata graphs. When evaluating the symbol-only reconfiguration feature, we use the BRAM-based design instead of the LUT-based design because it takes a shorter time to write new character sets in BRAM, and to use the symbol-only reconfiguration, we need to write character sets multiple times.

Not all automata-based applications can adopt the proposed method. The symbol-only reconfiguration only works for the applications in which users can come up with a universal automaton structure that can be shared by all patterns. For some applications, their patterns are so different from each other or their patterns keep changing/updating, it would be difficult or impossible to come out with a general structure. In such case, a general automata overlay on FPGAs that can quickly be re-configured for different patterns would be helpful, which is left for future work.

4 REAPR ON CLOUD PLATFORMS

4.1 Overview of Nimbix and Amazon Web Service

4.1.1 Nimbix. Nimbix is a cloud service provider powered by JARVICE platform (Nimbix 2017). Recently, Nimbix partnered with Xilinx to create a new cloud-based environment for FPGA developing to accelerate various applications. Users can choose different Xilinx FPGA boards to develop and test their FPGA implementations.

4.1.2 Amazon Web Service (AWS). AWS is a popular on-demand cloud service provider. Amazon EC2 F1 is the first AWS instance with FPGAs (16nm Xilinx UltraScale Plus FPGA) (AWS-FPGA 2017). The F1 instance is simple to set up and provide free tools for users to develop, simulate, debug, and compile hardware acceleration codes, including an FPGA Developer AMI (Amazon Machine Image) and a Hardware Developer Kit. There are two different types of the F1 instance: one is equipped with one FPGA (f1.2xlarge) and the other one is equipped with eight FPGAs (f1.16xlarge). Each FPGA has around 2.5 million logic elements, 6,800 Digital Signal Processing engines, a 64GB DDR4 memory, and a PCIe x16 interface.

Both platforms provide the SDAccel environment, allowing users to develop FPGA-based OpenCL applications in high-level languages without purchasing software licenses.

4.2 Workflow on Cloud

The automated workflow of the FPGA automata processing engine on local nodes is described in Section 3.4. In this work, we also implement the framework and evaluate the performance on cloud platforms, namely, the Nimbix Cloud environment and Amazon EC2 F1. For development, we choose the on-premises development on local workstations to avoid paying cloud charges for development. By providing proper information of the desired on-cloud FPGA (e.g., which FPGA board is available on the cloud), one can use the automated workflow to compile an automata application for the hardware binary and the host executable.

Fig. 8. Workflow on F1.

4.2.1 Workflow on Nimblex. To use REAPR on Nimblex Cloud, users first need to run the workflow on local nodes as discussed above. They need to modify the entry file (rtl.sh) to set up correct parameters for the specific FPGA board on Nimblex. The local workflow will generate the hardware binary and the host executable. Then users can submit the hardware binary and the host executable to the cloud, and wait in a queue to actually run on the FPGA board in Nimblex. In general, just one extra submission step is needed for Nimblex compared with the local workflow.

4.2.2 Workflow on AWS F1. For Amazon EC2 F1, users need several extra steps to run the framework on F1 instances. The whole workflow on F1 is shown in Figure 8. A toolkit is available in AWS-FPGA (2017) for F1 developers to use Xilinx FPGAs. Users need to download the toolkit on both local nodes and F1 instances, and need to make sure the local environment uses the correct SDAccel version supporting the FPGA board on AWS (Step 2). The top half in Figure 8 shows the

Table 1. ANMLZoo Details

Benchmark	Family	STEs	Reporting Elements
Snort	Regex	69,029	2,585
Dottar	Regex	96,438	2,837
ClamAV	Regex	49,538	515
PowerEN	Regex	40,513	2,857
Brill Tagging	Regex	42,658	1,962
Protomata	Regex	42,009	2,340
Hamming Distance	Mesh	11,346	186
Levenshtein Distance	Mesh	2,784	96
Entity Resolution (ER)	Widget	95,136	1,000
Sequential Pattern Mining (SPM)	Widget	100,500	5,025
Fermi	Widget	40,783	2,399
Random Forest (RF)	Widget	33,220	1,661

steps on local nodes. Similar to the automated workflow described in the above section, we need to modify the entry file (rtl.sh) and set up the proper platform parameters for the F1 FPGA board. This process will generate the executable and the hardware binary. After this, users need to upload the hardware binary and the host executable to the F1 instance. Then on F1 instances, users can use the tools provided in the toolkit to create the AFI (Amazon FPGA Image). The AFI contains all the information to run a specific application and can be reused across instances. Once the AFI is created successfully, users can run the application just as on local nodes. More details about the new workflow on AWS F1 instances can be found in Bo (2018b).

5 EVALUATION

We evaluate the FPGA automata processing engine performance on two cloud-based FPGA platforms: Xilinx Kintex UltraScale XCKU115 FPGA and Xilinx Virtex UltraScale Plus XCVU9P FPGA. Both FPGA boards are equipped with a single FPGA and four DDR4 banks. The XCKU115 board enables synthesizing kernels at 300MHz clock frequency, while the XCVU9P board allows 250MHz clock speed. The XCKU115 board is accessed through Nimbix Cloud and the VXC9P board is accessed through Amazon EC2 F1. In this work, we use the instance with one FPGA (f1.2xlarge). Using multiple FPGAs for larger automata will be targeted in our future work. We compare the FPGA performance with (i) Intel Hyperscan (Intel 2018) (a highly optimized automata processing engine for the CPU) running on an Intel i7-5820K CPU, (ii) two GPU-based engines, namely, DFAGE (Dang 2017a) (for DFAs) and iNFAnt2 (Dang 2017b) (for NFAs), on a Pascal-based Nvidia Titan X GPU, and (iii) Micron’s AP-based engine. All results are actual runtimes except for the AP (estimated results), because the AP hardware is not yet available on the market.

5.1 Benchmarks

We utilize the ANMLZoo automata benchmark suite developed by Wadden et al. (2017) for the cross-platform comparison. ANMLZoo contains various automata-based applications in three broad categories: regular expressions, mesh, and structured processing elements (or “widgets”). The applications in the benchmark, along with total STE numbers and reporting numbers, are listed in Table 1. The detailed description of these benchmarks can be found in Wadden et al. (2017). Each ANMLZoo benchmark is tailored to provide standardized automata, dubbed “standard candles,” which maxes out the resources of a single first-generation Micron AP chip. This

Fig. 9. CLB utilization.

benchmark suite allows convenient and relatively fair comparisons among different spatial architectures (FPGAs, AP) and von Neumann architectures (CPUs, GPUs). We also use real-world datasets for some of the applications and present the results in Section 5.5.

5.2 Utilization

In this section, we evaluate the CLB utilization on both AWS F1 and Nimble. There are 147,780 and 82,920 CLBs available on AWS F1 and Nimble, respectively. The results are shown in Figure 9, which presents the CLB utilization for both the kernel execution and the whole process including the I/O overhead. For the same application, Nimble and F1 use a similar amount of CLBs for the kernel execution, while F1 consumes more CLBs for the whole process, implying more CLBs are used for the I/O. This is interesting because the FPGA board on F1 (Virtex) is more advanced than the one on Nimble (Kintex). We tried to isolate the CLB usage for the automata processing kernel from CLB usage for the I/O, to determine why more CLBs were used for the Virtex board. However, after synthesizing different sizes for various applications and checking resulting design, we still cannot find the meaningful differences between F1 and Nimble. We suspect that Xilinx tools may use different optimization techniques for the two different boards, leading to better CLB usage on Nimble.

On average, CLBs consumed for the whole process is $2.7\times$ more than the kernel execution on Nimble; while on AWS F1, CLBs consumed for the whole process is $5.3\times$ more than the kernel execution. The results show that on both platforms, the I/O circuitry consumes most of the resources. Further optimization for I/O integration could help to reduce the CLB usage.

Furthermore, we study how different automata graphs affect the CLB usage and find that the CLB usage is mainly a function of two variables: the number of STEs and the number of routing nets. The results are shown in Figure 10 and Figure 11. As shown in these figures, on both cloud platforms, the CLB utilization increases roughly linearly as the number of STEs increases and the number of routing nets increases. Therefore, the larger the automaton is and the more complex the structure of the graph is, the more CLBs will be consumed. When designing automata for an

Fig. 10. CLB vs. STE numbers.

Fig. 11. CLB vs. Routing net numbers.

application, users may think about using a smaller and simpler automaton instead of a large and complex structure in order to save resources on FPGAs. Results for the kernel execution show a similar trend except for Snort, Fermi, and Brill. This is because, the automaton for the above applications is pretty simple, mostly just one symbol connected to another in a stringy way. Therefore, these applications do not consume as many routing nets as other applications with similar scale, leading to a lower CLB utilization.

5.3 Power

Similar to the above section, we present both on-chip power consumption of the kernel execution and of the whole process on Nimble and AWS F1, respectively. The results are shown in Figure 12. The power consumption is estimated using Xilinx Vivado. For different applications in ANMLZoo, the power consumption varies for the kernel execution (from 1.29W to 2.69W on Nimble, and from 2.50W to 3.86W on AWS F1). However, when evaluating the overall power consumption, it is close for different applications (from 16.62W to 20.82W on Nimble, and from 40.67W to 43.34W on AWS F1). The power consumption of the kernel only takes a small portion of the whole process (10% on Nimble and 7% on F1 on average). This concludes that the I/O circuitry dominates the power

Fig. 12. Power consumption for the kernel execution.

Fig. 13. Performance for the kernel execution on von Neumann architectures.

consumption. The I/O circuitry is the component that needs to be further improved if users want to reduce the power consumption.

5.4 Performance

5.4.1 Platforms. We evaluate the performance of each automata processing engine over 12 benchmarks in ANMLZoo using their accompanied 10MB stimuli. Results of the kernel execution and the end-to-end execution are shown in Figures 13–15 and Figures 17 and 18, respectively. The kernel execution denotes the operations within the accelerator boards (i.e., GPU, FPGA, and AP) including the automata processing kernel itself and the communication between the automata processing kernel and the on-board memory. The end-to-end execution encompasses the kernel execution and the overhead associated with the communication between the accelerator board and the host CPU. It is noted that the kernel performance on CPUs is identical to the end-to-end executions.

GPU engines exploit parallelism among automata and among input streams, where distinct automata are clustered into equal groups, and the input stream is divided into equal segments. Details of the two GPU implementations are described in Wadden et al. (2017) and Angstadt et al. (2018).

Fig. 14. Performance for the kernel execution on FPGAs and von Neumann architectures.

Fig. 15. Performance for the kernel execution on spatial architectures.

For GPU engines, the performance varies for different configurations, such as block sizes and grid sizes. The performance of the GPU NFA and DFA engines is achieved using the optimal block and grid size, and thread and stream configuration for each application.

In this work, the “ideal” AP performance and the projected AP performance obtained from the AP simulator (Wadden et al. 2018a) are presented, as denoted by the terms “ideal” and “sim” in Figure 15 and Figure 18, respectively. The “ideal” AP performance is derived from the nominal operating frequency (i.e., 133MHz), regardless of how automata reports are exported from the AP chip. This nominal “ideal” performance is commonly utilized in prior work on the AP (e.g., Zhou et al. (2015) and Wang et al. (2015, 2016b)). The AP kernel performance accounting for reporting overhead is estimated from the AP simulator. The AP simulator uses report traces generated by the Virtual Automata Simulator tool VASim (Wadden and Skadron 2016) to measure the costs associated with exporting these reports from the AP chip to the DDR3 on-board memory, and then generates runtime estimates. Details of the AP simulator can be found in Wadden et al. (2018a). The end-to-end performance should also include the communication overhead (both input and output) between the CPU and the AP board, and we project the communication overhead based on the total amount of the report events returned from the AP simulator and the theoretical PCIe gen3 x8 bandwidth (7,880MB/s). The actual end-to-end results should be a little worse than the projected results because of the difficulty to fully utilize the theoretical PCIe bandwidth.

5.4.2 Cross-Architecture Evaluation. As shown in Figure 13, the GPU NFA engine performs better than the CPU engine in 8 out of 12 benchmarks. This is also true for the end-to-end execution (Figure 16). This highlights the ability of GPUs to hide the latency of memory access by executing a large number of parallel tasks. However, the CPU engine outperforms the GPU NFA engine on Dotstar, ClamAV, PowerEN, and Hamming. This is most likely due to the small active sets in these applications, allowing for better cache behavior on CPUs, and due to the infrequent or not-at-all reporting behavior in these benchmarks. Therefore, for applications with larger activate states and more reports, the GPU NFA engine is a better fit compared with the CPU engine. On the other hand,

Fig. 16. Performance for the total execution on von Neumann architectures.

Fig. 17. Performance for the total execution on FPGAs and von Neumann architectures.

the GPU DFA engine traverses exactly one state per input symbol, while the GPU NFA engine follows a large number of per-symbol state transitions. This makes the GPU DFA kernel much simpler than the GPU NFA kernel. Specifically, the GPU DFA kernel consumes much fewer registers and instructions to process the automata than the GPU NFA kernel does. Unsurprisingly, Figure 13 shows that the GPU DFA engine (when DFAs can be created) outperforms the GPU NFA engine. For example, the GPU DFA engine achieves the best performance on Levenshtein (950MB/s and 115MB/s for the kernel execution and the end-to-end execution, respectively). The drop in its end-to-end execution is due to the substantial overhead of the DFA transition table transfer between the host CPU and the GPU board. This is also true for the NFA engine which we will discuss in Section 5.5 (loading the NFA table dominates the overall runtime). Although Levenshtein has the smallest number of NFA states in ANMLZoo, its complex 2D-mesh topology leads to an exponential increase in the number of DFA states and large DFA transition tables. Overall, if a DFA engine can be built on GPUs, it will produce the highest throughput among von Neumann architectures. A potential way to further improve the performance on GPUs is to minimize the DFA transition table.

For the sake of brevity, we only present the FPGA performance of the XCKU115 board in Figures 14, 15, 17 and 18. A comparison between the performances on Nimble and F1 is presented in Figure 19. Note that Dotstar, PowerEN, and SPM comprise large numbers of STEs, including reporting STEs, which prevents these benchmarks from being synthesized on the targeted FPGAs with one pass of the input. In most applications (except for ClamAV), the FPGA operating on a single input stream (FPGA_single) outperforms von Neumann automata engines, because automata graphs are laid out directly in the hardware so that a large number of transitions can be processed simultaneously, while CPUs and GPUs are bottlenecked by the memory latency for transition-rule look-ups. For ClamAV, as we discussed above, it has small active sets and a small number of reports, leading to a better cache behavior on CPUs. Figure 14 shows that the REAPR throughput varies from 65MB/s (Snort, Protomata, Fermi) to 266MB/s (Hamming, Levenshtein).

Fig. 18. Performance for the total execution on spatial architectures.

Fig. 19. Performance on Nimbix and AWS EC2 F1.

This is because the synthesized clock frequency is controlled by the board MAX frequency, the size of NFAs, the complexity of NFA topology, and the FPGA capacity and the memory interface (i.e., reporting architecture limited by the max. 512 bitwidth). For example, though Brill tagging and Protomata contain similar numbers of states, the structure of Protomata is more complex, leading to a lower throughput. Overall, the above results prove the advantage of using spatial architecture for automata processing. Furthermore, REAPR can handle multiple input streams in parallel (denoted as FPGA_multi in Figure 14) only for Hamming and Levenshtein, due to the limitation of the FPGA's capacity, memory bandwidth, and memory ports. Accordingly, REAPR can handle 6 streams for Hamming and 12 streams for Levenshtein, which leads to $5.4\times$ (1.4GB/s) and $10.6\times$ (2.8GB/s) better performance in the kernel execution, respectively, compared to processing a single input stream. This could help to further improve FPGA performance.

While the "ideal" AP Chip performance remains constant at 133MB/s, using the AP simulator allows us to obtain a more accurate projection, where the AP performance varies as a function of the reporting behavior of each benchmark. For instance, Snort suffers a $22\times$ slowdown (6MB/s) over the "ideal" performance. Because each ANMLZoo benchmark maximizes the resources of a single AP chip, it is fair to estimate the performance of an AP Board (32 chips) as $32\times$ the performance of a single AP chip (for both "ideal" and simulated performance). Similarly to REAPR, the AP shows a much better performance than von Neumann architectures. When comparing with

Fig. 20. Compilation overhead.

FPGAs, although REAPR excels in per-chip capacity, its per-board capacity lags far behind the AP (e.g., 2.8GB/s vs. 4.2GB/s on Levenshtein). This is because an FPGA board, such as the XCKU115, typically contains just one FPGA chip, while the AP board contains 32 AP chips. In an exceedingly large application, users may consider using multiple FPGA boards.

When the end-to-end execution is considered, we notice the performance degradation in REAPR due to its reporting architecture. The current REAPR reporting architecture with massive data transfer of reports (from the on-board DDR memory to the host CPU memory) has a significant impact on the overall performance. Nevertheless, REAPR still shows promising speedups compared to CPUs and GPUs in many applications. When compared to the AP, REAPR outperforms the simulated AP Chip performance in 4 of 12 benchmarks. This is explained by their reporting architectures. The AP only generates a report vector when a match actually occurs, while REAPR records the outputs from all reporting STEs per cycle regardless of whether there is a match. One possible way to improve the current REAPR reporting architecture is to check the reporting vector and only transfer the ones with actual reports.

Figure 19 shows the REAPR performance evaluated on Nimble Cloud and on Amazon EC2 F1. Nimble Cloud outperforms F1 in most benchmarks, due to its higher clock frequency (300MHz on Nimble Cloud vs. 250MHz on F1). F1 shows better performance on Fermi, Protomata, and Snort. In these three benchmarks, the automata graphs are more complex; the Vivado compiler was unable to efficiently place and route for the XCKU115 FPGA (Nimble) because of the limited hardware resource compared to the XCVU9P FPGA (F1), leading to a lower frequency.

5.5 Symbol-Only Reconfiguration

In this section, we evaluate the symbol-only reconfiguration method against the original REAPR without this feature. As mentioned in Section 3.1, we use the BRAM-based design for symbol-only reconfiguration to utilize the faster writing speed of BRAM. We first evaluate the compilation overhead of some applications in the ANMLZoo to show the feasibility of the proposed method and how much potential benefit can be achieved. The applications include Entity Resolution, Random Forest, ClamAV, Hamming Distance, and Levenshtein Distance. The results are presented in Figure 20. Results show that adopting symbol-only reconfiguration (dark bars) leads to longer one-time compilation time. But the increase is small compared to the ones without the symbol-only reconfiguration feature. On average, it consumes a 6.5% longer time for compilation. Therefore,

Table 2. Character Sets Writing Overhead

Benchmark	Writing overhead (millisecond)
ClamAV	335
Hamming Distance	46.7
Levenshtein Distance	35.7
Entity Resolution (ER)	293
Random Forest (RF)	408

Table 3. Full-Size Application Performance

Benchmark	# of states	Input size	GPU (total)	GPU (loading)	Hyperscan	FPGA	SOR?	Compilation
CRISPR	346,000	3.2GB	1,579	278.7	44,257	481	Yes	6h 23m
ER	13,314,000	3.1MB	NA	NA	NA	51.24	Yes	17h 50m
SPM	201,500	0.3MB	183	181.5	34	0.18	Yes	4h 3m
RF	992,000	4.6MB	9,697	9,686.4	216	9.9	Yes	5h 24m

The performance results on of Hyperscan, GPU, and FPGA are in seconds. NA refers to the automata that is too large to run with GPU/Hyperscan. SR stands for symbol-only reconfiguration.

for smaller pattern sets that can be fit on one device, users should use the original REAPR (LUT-based). However, for larger pattern sets, REAPR without symbol-only reconfiguration incurs a large amount of time on the compilation for every new partition, which is not needed by the symbol-only reconfiguration approach. Users only need to compile the automata graph for the symbol-only reconfiguration once and simply write new character sets to the FPGA for new partitions. The overhead of writing new character sets is shown in Table 2. The writing overhead varies among these applications, because the automata structure is different and the number of states to be written also varies for each application. However, all of the writing overhead is under 1 second. This is a huge reduction from hours of re-compilation.

We use a few example applications from ANMLZoo and evaluate them with real-world datasets. The real end-to-end performance results of the full-size application runtime on CPUs, GPUs, and FPGAs are presented in Table 3. The AP performance is not presented here because its performance is estimated. CRISPR is an application used as an example of Hamming distance, which uses Hamming distance automata as its major component to identify gRNA off-target sites for the CRISPR/Cas9 system (Wadden et al. 2018b; Bo 2018a). For the GPU performance, we run iNFAnt2 with the NFA representation. Even with the NFA representation (smaller than the DFA representation), the loading time of the NFA file takes most of the total time. For example, for Random Forest, it takes over 99% of the total time; and for Entity Resolution, after 24 hours it still cannot load the NFA file, so we assume this is too slow and use “NA” in the table. Every time users run a new dataset, the GPU needs to reload the NFA file. Therefore, the size and complexity of the NFA file is the major factor for the performance on GPUs. However, the CPU automata engine (Hyperscan) is more sensitive to the input size compared with the GPU engine. For example, when running CRISPR, it is way slower than iNFAnt2 because the input of CRISPR is very large (Human Genome, 3.2GB); while for other smaller inputs, it is close to the GPU performance sometimes even better. The performance on FPGA does not include the compilation times and the one-time compilation time is provided in a separate column. The compilation can be finished before actually using FPGAs to run these applications. The FPGA automata engine with symbol-only reconfiguration works much faster than both Hyperscan and iNFAnt2, especially when the automata size is

very large such as the dataset in RF. It can also process very large-scale applications when GPU and CPU engines could not serve its intended purpose as in ER.

The above results show that if an application (especially the ones with large pattern sets) can adopt the symbol-only reconfiguration approach, the feature helps to drastically reduce the compilation overhead and improves the overall performance.

In summary, automata processing on GPUs only shows minor improvement over CPUs, and GPU DFA engines usually provide a better performance compared to GPU NFA engines. However, processing automata on FPGAs achieves much better performance than using von Neumann architectures and shows nice scalability for large-scale applications. More specialized hardware accelerators such as Micron's AP could provide even better performance by integrating many chips on a single board. The performance on spatial architectures could be further improved by a more efficient reporting architecture. Furthermore, CLB usage and power consumption show that I/O circuits dominate hardware utilization on FPGAs. Simplifying I/O circuitry could help to reduce resource consumption and store more states.

5.6 Discussion

5.6.1 I/O Overhead. The end-to-end performance results in the previous section show that the I/O plays an important role in the overall runtime, which is ignored by prior work. Although the current reporting architecture in REAPR works for most of the applications in ANMLZoo, it is not the optimal one and only works for the off-load model. The results are not transferred back to the host CPU until the whole input string is processed. There are several possible ways to improve the current reporting architecture, for example, using a double buffer to store reports and transfer the results back to CPU when the buffer is full instead of transferring all the data until the kernel finishes. Another solution is to get rid of the unnecessary data transfer in the data block. A checker could be added in the reporting architecture and only transfers the vectors back to the host CPU when a match actually occurs in that cycle.

5.6.2 Automata Overlay. Although the symbol-only reconfiguration helps to improve the performance for applications with larger pattern sets, it only works for certain applications. How to reduce the high compilation overhead on FPGAs for other automata-based applications is still an open question. Karakchi et al. propose an overlay as an alternative to Micron's AP, but it does not apply for most of the applications in ANMLZoo because of the constraints on max fan-in and fan-out (Karakchi et al. 2017). One possibility is to design a generalized automata processing overlay on FPGAs, so that even for different applications, users do not need to pay the high cost of compilation. However, the various topologies of automata graphs make it difficult to design a generalized automata overlay for all applications. One potential solution is to design a few different automata overlays, and use the one that fits the specific application best. For example, we may design a 2D mesh automata overlay for applications involving Hamming distance or Levenshtein distance, and the applications involving Hamming/Levenshtein distance can adopt this overlay.

6 CONCLUSIONS AND FUTURE WORK

In this article, we presented a framework that generates RTL and I/O circuitry for automata processing on FPGAs. We implement the framework on both local nodes and cloud-based platforms. We present a full analysis of various automata engines (CPU, GPU, FPGA, and AP) on a diverse range of workloads. CPUs and GPUs for automata processing are generally bottlenecked by the memory latency for rule lookups. CPUs perform well if the average active sets in the automata are small, or if the automata application has few reporting activities. GPUs perform well with DFA representation, hiding the latency of memory access by executing a large number of parallel tasks.

However, DFA-based engines suffer from the state-explosion problem, which prevents them from processing very large or complex automata. When evaluating full-size applications, the GPU NFA engine also degrades when the NFA is large. Spatial architectures (FPGAs and APs) outperform von Neumann architectures in the automata processing domain, because the FPGA and AP can process automata with massive parallelism brought by laying out automata graphs directly in hardware. However, FPGA and AP performance depends heavily on the reporting activity and the reporting architecture. We then study the CLB utilization and power consumption and find that the I/O circuitry consumes most of the resource. Finally, we propose a symbol-only reconfiguration mechanism to process large pattern sets via multiple passes and evaluate it with real-world datasets. Future work will target more efficient reporting architectures, techniques to achieve higher clock speeds, as well as the capability of processing extremely large applications on multiple FPGAs.

REFERENCES

- Kevin Angstadt, Jack Wadden, Vinh Dang, Ted Xie, Dan Kramp, Westley Weimer, Mircea Stan, and Kevin Skadron. 2018. MNCaRT: An open-source, multi-architecture automata-processing research and execution ecosystem. *IEEE Computer Architecture Letters* 17, 1 (2018), 84–87.
- Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID programming of pattern-recognition processors. *ACM SIGOPS Operating Systems Review* 50, 2 (2016), 593–605.
- AWS. 2017. Amazon EC2 F1 Instances. Retrieved October 2017 from <https://aws.amazon.com/ec2/instance-types/f1/>.
- AWS-FPGA. 2017. Official Repository of the AWS EC2 FPGA Hardware and Software Development Kit. Retrieved October 2017 from <https://github.com/aws/aws-fpga>.
- Michela Becci and Patrick Crowley. 2007. A hybrid finite automaton for practical deep packet inspection. In *ACM CoNEXT Conference*.
- Chunkun Bo. 2018a. Identify gRNA Off-Targets for CRISPR/Cas9 using Automata Processing. Retrieved February 2018a from <https://github.com/chunkunbo/CRISPR>.
- Chunkun Bo. 2018b. REAPR on F1. Retrieved January 2018b from <https://github.com/chunkunbo/REAPR-on-Amazon-F1>.
- Chunkun Bo, Dang Vinh, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for potential gRNA off-target sites for CRISPR/Cas9 using automata processing across different platforms. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*.
- Chunkun Bo, Ke Wang, Jefferey J. Fox, and Kevin Skadron. 2015a. Entity resolution acceleration using Micron's automata processor. In *Proceedings of Architectures and Systems for Big Data (ASBD'15), in conjunction with ISCA (2015)*.
- Chunkun Bo, Ke Wang, Jefferey J. Fox, and Kevin Skadron. 2016. Entity resolution acceleration using the automata processor. In *Proceedings of IEEE International Conference on Big Data (Big Data'16)*.
- Chunkun Bo, Ke Wang, Yanjun Qi, and Kevin Skadron. 2015b. String kernel testing acceleration using the Micron automata processor. In *Proceedings of Workshop on Computer Architecture for Machine Learning*.
- Pascal Caron and Djelloul Ziadi. 2000. Characterization of Glushkov automata. *Theoretical Computer Science* 233, 1–2 (2000), 75–90.
- Niccolo Cascarano, Pierluigi Rolando, Fulvio Rizzo, and Riccardo Sisto. 2010. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review* 40, 5 (2010), 20–26.
- Vinh Dang. 2017a. A Deterministic Finite Automata GPU-Based Engine. Retrieved May 2017a from <https://github.com/vqd8a/DFAGE>.
- Vinh Dang. 2017b. iNFAnt2. Retrieved May 2017b from <https://github.com/vqd8a/iNFAnt2>.
- Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An efficient and scalable semi-conductor architecture for parallel automata processing. In *Proceedings of IEEE Transactions on Parallel and Distributed Systems*.
- Yuanwei Fang, Tung T. Hoang, Michela Becci, and Andrew A. Chien. 2015. HARE: Hardware accelerator for regular expressions. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*.
- Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.
- Tran Trung Hieu and Ngoc Thinh Tran. 2013. A memory efficient FPGA-based pattern matching engine for stateful NIDS. In *Proceedings of the 5th Annual IEEE Conference Ubiquitous and Future Networks (ICUFN'13)*.
- Xilinx Inc. 2018. SDAccel Development Environment. Retrieved January 2018 from <https://www.xilinx.com/products/design-tools/softwarezone/sdaccel.html>.

- Intel. 2018. High-Performance Regular Expression Matching Library. Retrieved January 2018 from <https://github.com/intel/hyperscan>.
- Rasha Karakchi, Lothrop O. Richards, and Jason D. Bakos. 2017. A dynamically reconfigurable automata processor overlay. In *Proceedings of International Conference on ReConfigurable Computing and FPGAs (ReConFig'17)*.
- Micron. 2015. ANML Documentation. Retrieved August 2015 from http://www.micronautomata.com/documentation/anml_documentation/c_intro.html.
- Roger Moussalli, Mariam Salloum, Robert Halstead, Walid Najjar, and Vassilis J. Tsotras. 2014. A study on parallelizing XML path filtering using accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 13, 4 (2014), 93.
- Nimbix. 2017. Xilinx FPGAs on the Nimbix Cloud. Retrieved October 2017 from <https://www.nimbix.net/xilinx/>.
- Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu chun Feng, and Michela Becchi. 2017. Demystifying automata processing: GPUs, FPGAs or Micron's AP? In *Proceedings of the International Conference on Supercomputing*. ACM, 1.
- Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, and Kevin Skadron. 2018. A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*.
- Reetinder Sidhu and Viktor Prosanna. 2001. Fast regular expression matching using FPGAs. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*.
- Jens Teubner, Louis Woods, and Chongling Nie. 2012. Skeleton automata for FPGAs: Reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 229–240.
- Jack Wadden, Kevin Angstadt, and Kevin Skadron. 2018a. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*.
- Jack Wadden, Tommy Tracy II, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Matthew Wallace, Jeffrey Udall, Mircea Stan, and Kevin Skadron. 2018b. AutomataZoo: A modern automata processing benchmark suite. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC'18)*.
- Jack Wadden and Kevin Skadron. 2016. VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research. Technical Report CS2016–03, University of Virginia.
- Jack Wadden, Dang Vinh, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Robins Gabriel, Mircea Stan, and Kevin Skadron. 2017. ANMLzoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC'17)*.
- Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016a. An overview of Micron's automata processor. In *Proceedings of Workshop on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'16)*.
- Ke Wang, Yanjun Qi, Jefferey J. Fox, Mircea R. Stan, and Kevin Skadron. 2015. Association rule mining with the Micron automata processor. In *Proceedings of IEEE International Conference on Parallel and Distributed Processing Symposium (IPDPS'15)*.
- Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016b. Sequential pattern mining with the Micron automata processor. In *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 135–144.
- Ted Xie, Vinh Dang, Chunkun Bo, Jack Wadden, Kevin Skadron, and Mircea Stan. 2018. REAPR: Reconfigurable Engine for Automata Processing. Retrieved August 2018 from <https://github.com/ted-xie/REAPR>.
- Ted Xie, Dang Vinh, Jack Wadden, Mircea Stan, and Kevin Skadron. 2017. REAPR: Reconfigurable engine for automata processing. In *Proceedings of IEEE International Conference on Field Programmable Logic and Applications (FPL'17)*.
- Yi-Hua Yang and Viktor Prasanna. 2012. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Transactions on Computers* 61, 7 (2012), 1013–1025.
- Xiaodong Yu, Kaixi Hou, Hao Wang, and Wu chun Feng. 2017. Robotomata: A framework for approximate pattern matching of big data on an automata processor. In *Proceedings of the 2017 IEEE International Conference on Big Data (Big Data'17)*. IEEE, 283–292.
- Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Brill tagging on the Micron automata processor. In *Proceedings of IEEE International Conference on Semantic Computing (ICSC'15)*.

Received July 2018; revised December 2018; accepted February 2019