Automata Processing: from Application Acceleration to Hardware Design

---

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment
of the requirements for the degree

Doctor of Philosophy

by

Chunkun Bo

December 2019

APPROVAL SHEET

This Dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Author Signature: _____

This Dissertation has been read and approved by the examining committee:

Advisor: Kevin Skadron

Committee Member: Mircea Stan

Committee Member: Samira Khan

Committee Member: Ashish Venkat

Committee Member: Ashish Sirasao

Committee Member: _____

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

December 2019

Abstract

Inexact pattern matching is widely used in many fields, such as machine learning, network intrusion detection, bioinformatics, and many other applications. Inexact pattern matching is usually the most time-consuming phase in such applications, because of the complexity of inexact matching between the input string and a large number of reference patterns; thus, reducing inexact matching time is key to accelerate such applications. Automata processing is a computational model that is widely used for matching regular expressions and can be used for inexact pattern matching. However, because of the memory bottleneck due to the random memory behavior of automata processing on von-Neumann architectures, the performance is not satisfying, especially for large-scale applications, and this requires hardware acceleration. This dissertation studies the potential benefits of spatial architectures, which can process certain computation using reconfigurable processing elements.

To understand better how automata processing could be used to accelerate inexact pattern matching, we use Entity Resolution and searching for potential gRNA off-targets for CRISPR/Cas9 as two case studies. We solve the two problems using automata-based approaches, and evaluate the performance on both von-Neumann architectures (CPUs and GPUs) and spatial architectures (Micron's Automata Processor and FPGAs). We find that automata engines on spatial architectures show a clear advantage over methods on von-Neumann architectures. Therefore, we provide an automated automata processing framework using FPGAs on cloud-based platforms, and propose several novel features to further improve the performance. Finally, to reduce the high compilation overhead on FPGAs, we propose three different methods, including symbol-only reconfiguration, a new workflow using Xilinx Object file, and modular synthesis and reuse of I/O template.

Acknowledgements

During my Ph.D. study and the writing of this dissertation, I have received an enormous amount of assistance. I would not have been able to finish this dissertation without their help.

First and foremost, I would like to express the deepest appreciation to my advisor, Prof. Kevin Skadron, for guiding me through the whole journey. He is a great mentor and teaches me a lot, not only about being a good researcher, but more importantly, being a better person. I could not imagine having a better advisor. He motivates me to explore various ideas, to express myself, to communicate with peer researchers, and to be an independent researcher. He is always ready to help and discuss all kinds of problems/concerns I may have. As an international student, I have some difficult time, especially in the past few months. His tremendous support helps me to overcome all those issues.

I wish to thank the members of my Ph.D. committee from UVa: Prof. Mircea Stan, Prof. Samira Khan, Prof. Ashish Venkat. I would also like to thank Mr. Ashish Sirasao for guiding me during my internship and being part of the committee. Thank you all for the valuable comments and suggestions.

I also want to thank all my colleagues inside and outside CS department. It is a great honor to work and collaborate with you all, Vinh Dang, Ke Wang, Elaheh Sadredini, Reza Rahimi, Ted Xie, Jack Wadden, Tommy Tracy, Kevin Angstadt, Deyuan Guo, Lingxi Wu, and Sergui Mosanu.

This work was supported in part by NSF grant no. CCF-1629450, a grant from Xilinx, and support from Center for Future Architectures Research (C-FAR) and Center for Research on Intelligent Storage and Processing-in-memory (CRISP) , two centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Last but not least, I would like to thank my parents, my older sister, and my family, for their unconditional love and support. My parents received limited education and they do not know a single English word, but they always encourage me to explore the world and learn from the best. My older sister has taken care of me since I was a kid, and now she is taking care of my parents while I am far away from home. I dedicate this dissertation to all of them.

Ever tried. Ever failed. No matter. Try Again. Fail again. Fail better.
-Samuel Beckett

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Inexact pattern matching is a common computational task that identifies patterns with variances relative to a reference pattern. It arises in many different application domains, such as machine learning [1] [2] [3], cyber security [4] [5], bioinformatics [6] [7], anti-virus scanning [8], natural language processing [9], etc. This process is often a major performance bottleneck of the whole application, because of the time complexity to identify the similarity among patterns [10]. Three major factors contribute to the overall complexity: 1). the allowed similarity between patterns, 2). the length of the input stream, and 3). the number of patterns to identify. As we are in the era of "big data", the length of input stream could be hundreds of MB or even GB, and there could be a large number (millions) of patterns to be compared against the input. For example, for Next Generation Sequencing problems, the input is Human Genome (over 3.2GB), and we need to check hundreds of thousand long (>100) DNA sequences against such a huge input stream [11].

One common way of searching for various patterns is to use regular expressions [12]. Regular expressions can describe complex patterns and can recognize derivative patterns of a base pattern (Fig. 1.1). Automata processing is an efficient model that can recognize regular expressions, by directly translating patterns to a set of states and a set of transition rules between states stimulated by the input symbol (Fig. 1.1). An automaton can be represented either as a Deterministic Finite Automaton (DFA) or as a Non-deterministic Finite Automaton (NFA). Regular expressions and automata processing are computationally equivalent, but sometimes it is more convenient and straightforward to represent patterns as automata for many applications. For example, automata processing is used in [13], [14], and [9] to accelerate various applications. However, the performance of existing automata processing

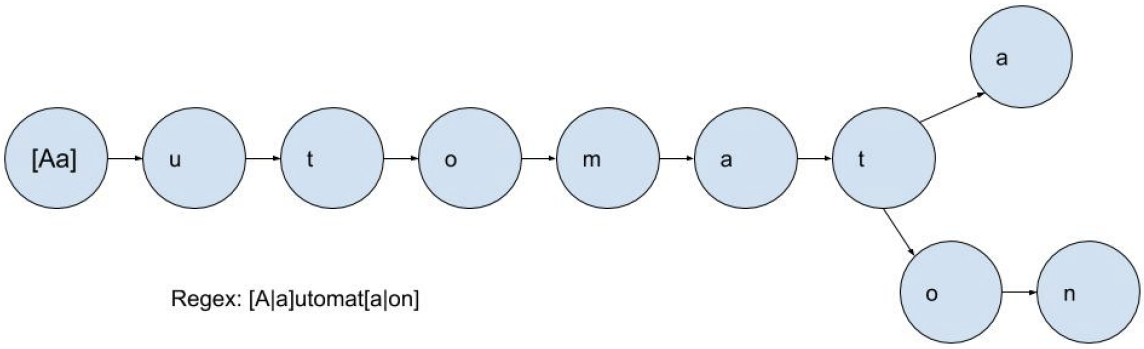

Figure 1.1: Regex and Automaton for singular/plural representation of word "Automaton/Automata".

approaches on von-Neumann architectures cannot meet modern applications' requirements because of the increase of the time complexity as discussed above [9] [15] [16]. One on hand, von-Neumann architectures need to store all states and transition rules in memory. Simulating behaviors of automata processing leads to random memory-access patterns because of the lack of spatial and temporal locality, which makes memory access the bottleneck in von-Neumann architectures. On the other hand, the amount of data is increasing extremely fast currently (exponential growth of data [17]), making the performance issue more severe for modern large-scale applications on von-Neumann architectures. Many efforts have been made to improve the performance of automata processing on von-Neumann architectures. Researchers propose different methods to mitigate the memory bottleneck on von-Neumann architectures (CPUs and GPUs), such as Hyperscan, iNFAnt, iNFAnt2 and DFAge [18] [19] [20] [21]. These methods explore different levels of parallelism to hide the memory access latency. Other researchers propose accelerating automata processing on *spatial architectures* (process certain computation using reconfigurable processing elements), such as Micron's Automata Processor [22] and FPGAs [23] [24] [25] [26]. Using spatial architectures allows users to store a large number of patterns directly on the hardware and match all these patterns against the input stream simultaneously. This helps to increase the parallelism drastically, leading to high throughput.

In this thesis, we hypothesize that using FPGAs for automata processing provides a high-performance, scalable, and user-friendly platform for applications involving inexact pattern matching. To evaluate the hypothesis, this dissertation conducts four major research projects. This dissertation first proposes using automata processing

for two inexact pattern matching applications. 1). Entity Resolution Acceleration using Automata Processing in Knowledge Discovery [15], and 2). Searching for potential gRNA off-targets for CRISPR/Cas9 using Automata Processing in bioinformatics [27]. Automata processing on spatial architectures shows promising results, and we use FPGAs as our primary research spatial hardware. This dissertation then 3). provides an automata processing engine on Cloud-based platforms with new features and improved performance [28], and 4). presents several methods to reduce the high compilation overhead on FPGAs [28].

1.1 Contributions

1.1.1 Entity Resolution Acceleration using Automata Processing

Entity Resolution (ER), also known as Record Linkage, Duplication Reduction or Purging/Merging problems, refers to finding records that store the same entity within a single database or across different databases [29]. ER is an important kernel of many information integration applications in Knowledge Discovery [30]. Determining whether two records represent the same entity is computationally expensive. The time complexity of a naive method is $O(N^2)$, where N is the number of records. Prior work has proposed different algorithms and computation models to improve the performance [31] [32] [33] [34]. However, the performance is still unsatisfying [35].

Micron's Automata Processor (AP) is an efficient and scalable semiconductor architecture for parallel automata processing [22] and can process a large number of complex patterns simultaneously. Therefore, we propose using the AP to accelerate ER. To illustrate how the AP approach works, we present a framework and several different automata designs for various ER applications. We evaluate the suitability of the prototype using several real-world ER problems in different applications. Results show both higher performance and better resolution accuracy using various datasets. This shows that automata processing can help to accelerate the ER problem and the spatial architecture (Micron's AP) can help to further improve the performance.

1.1.2 Searching for Potential gRNA Off-Target sites for CRISPR/Cas9 using Automata Processing across Different Platforms

CRISPR/Cas is an immune system that defends against foreign genetic elements [36]. CRISPR/Cas9 is one version that attracts researchers' paticular interest, because of its ability to edit genomes [37] [38]. Genome editing using CRISPR/Cas9 has been used to cure genetic-related diseases such as amnesia [39] [40]. However, efficiently finding all correct locations to edit the genome, without modifying other locations, is still the bottleneck of using the CRISPR/Cas9 system, because the gRNA sometimes binds to locations with slightly different DNA sequences [41]. This makes the process of finding all potential *off-target* sites (genome locations sufficiently similar to the gRNA targeting sequence) computationally expensive, especially when one allows more differences from the reference sequence. It could take hours/days to search for potential off-target sites when only allowing a few mismatches even with GPUs [42]. Furthermore, existing CPU and GPU tools are also restricted to the number of mismatches or fail to return results due to the computational bottlenecks [42] [43].

To solve the above problem, we propose an automata-based solution to identify potential off-target sites in a reference genome. We present several designs that can recognize different variations of a gRNA, and a general workflow of how to use automata processing to identify potential off-target sites. We evaluate the proposed automata approach across four different platforms (CPUs, GPUs, FPGAs, and Micron's AP), and compare with two state-of-the-art solutions (CasOFFinder [43] and CasOT [44]). The proposed method leads to over 83× speedups on FPGAs compared with CasOFFinder (GPU), and additional speedups can be achieved by using the AP. The automata-based method with iNFAnt2 [45] on GPUs does not confer a clear advantage because automata processing does not map well to the GPU architecture. This again shows the ability of using automata to accelerate applications involving with inexact matching, and shows that spatial architectures are more appropriate for automata processing.

1.1.3 Automata Processing Engine on Cloud-based FPGAs with New Features and Cross-platform Evaluation

The results of the above two applications show that automata processing engines on spatial architectures provides promising performance over von-Neumann architectures, by laying out automata graphs directly on hardware and processing a large number of automata in parallel. Reconfigurable Engine for Automata PRocessing (REAPR) is thus proposed for general automata processing on FPGAs in [25]. It is a framework that can generate RTL codes for automata processing kernel and the I/O circuitry for data transfer between the FPGA and the host CPU.

Though REAPR shows promising results, the performance could be further improved (e.g., the poor performance of the reporting architecture), and there are several limitations of the framework (such as involving many manual efforts for kernel integration and high reconfiguration overhead). Therefore, in this chapter, we provide a more efficient automata processing framework on FPGAs with several new features. We then conduct a cross-platform performance evaluation, and collect the resource utilization and power consumption on cloud platforms.

We will briefly introduce the features, and details can be found in Chapter 5.

1. A new automated workflow for the kernel integration

The original workflow uses the Xilinx SDAccel [46] to generate a dummy kernel and replaces the dummy kernel with actual automata processing kernel. The replacement and integration phase involves a lot of manual work and is prone to errors. Therefore, we provide a set of scripts that automate the replacement and integration of the automata kernel.

2. Processing multiple symbols per cycle

To increase the throughput of REAPR and to compete with other high-throughput automata processing engines, we modify the spatial stack algorithm for processing multiple symbols for regular expressions [24] and support it for general automata processing. We implement the modified algorithm in the original framework and achieve almost linear speedups when stacking multiple copies.

3. New reporting architecture to improve the performance

Preliminary results of REAPR show that the reporting architecture is the major performance bottleneck. This is because, in the original REAPR, we need to transfer the reporting result back to the CPU even when there are no reports in reporting vectors. However, because of the filtering nature automata processing, most

automata-based applications do not report every cycle. Only some cycles have actual reports. Therefore, we propose only transferring the necessary data block back by adding a checker in the framework to check for actual reports. This helps to reduce the transfer time from the FPGA to the host CPU.

4. Simplified I/O integration with one DDR bank

The original REAPR utilizes multiple DDR banks for transferring the results back to the host CPU to maximize the performance. Though this helps to achieve better performance, it makes the integration more complex, because applications with different numbers of reports need different numbers of DDR banks to transfer results and may even use the DDR banks multiple times, leading to difficult integration of the automata kernel. To solve this problem, we propose using just one DDR bank for the output and using a FIFO between the kernel and the DDR bank as a buffer for the output to simplify the integration.

5. Cloud-based platforms implementation

As cloud platforms attract more users' and developers' interest, we deploy the framework on two cloud-based platforms (Amazon AWs EC2 and Nimbix) with a new automated workflow, available on [47].

6. Cross-platform evaluation All the above new features help to provide a better-performance and user-friendly platform for inexact pattern matching applications. We compare the platform against other existing platforms (CPUs, GPUs, and Micron's AP) using ANMLZoo [45]. Results show that using FPGAs provide great potential for automata processing.

1.1.4 Reducing High Compilation Overhead for the Automata Processing Engine on FPGAs

High compilation overhead is a common problem when using FPGAs to accelerate applications. We encounter similar problems when using FPGAs for automata processing. For example, it took almost eighteen hours to finish the compilation for ER. For some inexact pattern matching applications, the rule/pattern sets may change or update periodically. It is costly to configure FPGAs for every single new rule/pattern. Furthermore, for large pattern sets or problem sizes, the automata may not fit on a single device. This requires a method to partition the automata and support multiple passes with fast reconfiguration for each partition. To reduce the configuration overhead, we propose three novel methods (symbol-only reconfiguration [28], a new

workflow using Xilinx Object file, and modular synthesis to reuse compiled components) in the framework.

1. Symbol-only reconfiguration to process large-scale applications

When working with automata processing applications, we find that we can design a general automaton structure for some applications. This allows us to compile the structure once, and reuse the complied structure for new datasets by writing new symbols to be stored in the structure. We name it the symbol-only reconfigure mechanism. Writing new symbols (seconds) is much faster than compiling new structures (hours), thus allowing us to process larger datasets much faster when multiple passes of the input is needed.

2. New workflow using Xilinx Object file

In the original REAPR workflow, to hide all the details from users, we integrate all kernels for the compilation step (including FPGA synthesis, logic optimization, logic placement and routing). We then notice that using the Xilinx Object file can help to reduce the time spent in the compilation phase. The Xilinx Object file stores the information about the RTL kernels. Therefore, we propose a new workflow by using the Xilinx Object(.xo) file in Xilinx SDAccel.

3. Modular synthesis and reuse the I/O communication structure

In the original workflow, we build the I/O communication structure for each application and each different dataset. However, the I/O structure is similar for applications with a similar number of reporting states even though the applications are different, and we can reuse this I/O structure for different applications/datasets. Therefore, we modularize the original RTL kernel and isolate the I/O communication between the CPU and the FPGA from the whole kernel. For new applications, we can use the pre-synthesized I/O part and only synthesize the actual automata kernel, which helps to reduce the synthesis time and placement and routing time.

1.1.5 Summary

This dissertation studies how to use automata processing to accelerate inexact pattern matching applications. We study two applications using automata processing from Knowledge Discovery and Bioinformatics domains. Accelerating these applications provides insights that spatial architectures are better fit for automata processing; therefore we provide a high-performance, scalable, and user-friendly platform on FPGAs. Finally, we propose several methods to reduce the high compilation overhead for the FPGA automata processing engine.

1.2 Organization

The remainder of the dissertation is organized as follows:

Chapter 2: Background introduces automata processing in general and existing platforms for automata processing.

Chapter 3: Entity Resolution using Automata Processing presents a novel method based on automata processing to accelerate the applications involving ER.

Chapter 4: Searching for gRNA Off-target Sites in CRISPR/CAS9 System across Different Platforms presents how we use automata processing to identify potential gRNA off-targets sites. We also evaluate the proposed method across different hardware in order to find a proper platform for such problems.

Chapter 5: Deploy REAPR on Cloud-based Platforms with New Features present the new features we propose and implement on the original REAPR to provide higher performance and make it more user-friendly.

Chapter 6: Ways to Reduce Compilation Overhead presents three different methods to solve the high compilation overhead.

Chapter 7: Conclusion and Future Work summarizes the dissertation and discusses potential future directions of research.

Chapter 2

Background

2.1 Automata Processing

Automata processing can be informally defined as a set of states and a set of transition arrows that connect these states. Each state stores multiple characters, and an input stimulus is streamed into the starting states of an automaton [48]. For each cycle, if a state is activated and the input symbol matches the characters stored in this state, the state will activate all states that connect to the current state. Once a state that is set as a reporting state is activated, it finds a match for that pattern.

A traditional automaton can be represented as a Deterministic Finite Automaton (DFA) or a Non-deterministic Finite Automaton (NFA) [49]. A DFA cannot use empty string (ϵ) transition, and can only be in, and transition to, one state at a time for a given input symbol. However, an NFA can use empty string(ϵ) transition, can transition to and be in, multiple states for a given input symbol. DFA and NFA are computationally equivalent (one example is shown in Fig 2.1), but there are trade-offs between them that affect the performance. For DFA, we only need to compute one transition per cycle, thus requiring less computational power, but the number of states in a DFA is usually much larger than that in an equivalent NFA. This phenomenon is usually referred to as "state explosion" problem. While for NFA, we usually need more computation to process all transitions per cycle, thus leading to higher parallelism, and the automaton structure is smaller. Prior work by Becchi [50] attempted to leverage the best of both types of finite automata (the spatial density of NFA and 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 10x reduction in-memory

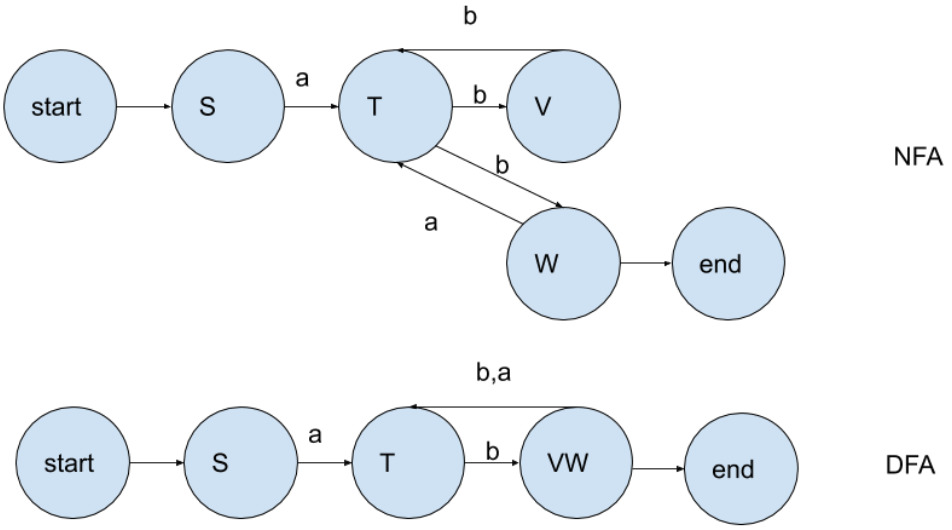

Figure 2.1: An NFA and its equivalent DFA.

transactions. There are also several other work that tries to combine the benefits of DFA and NFA [51] [52] [53].

Merging the transition logic with the state transforms a traditional NFA into a homogeneous finite automaton [54], where all incoming transitions to a state have the same (homogeneous) rule. The major advantage of using homogeneous automata is that we can compute all transition rules simultaneously for the input symbol and broadcast the activation signals to successor states. This feature maps naturally to logic elements in reconfigurable hardware, thus in both Micron's Automata Processor and the REAPR framework use homogeneous automata representation.

2.2 Automata Processing on von Neumann Architectures

2.2.1 CPUs

In an NFA, symbols from the input stream are broadcast to each state simultaneously, and each state connects to several other states, each of which may or may not activate depending on whether a given state matches the incoming symbol [28]. For each

symbol, an NFA engine must determine the next set of activated states, which involves a linear-time scan of the adjacency lists of all states in the current activated set. 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 $O(n \cdot m)$, and for non-trivial data with non-trivial automata ($n = m$), the overall runtime is quadratic. CPU NFA processing 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 perform SIMD vector operations to execute as many state transitions as possible for a given memory transaction. Even so, such optimizations can not escape the fact that sequential von Neumann architectures are *fundamentally* ill-suited for these types of 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 only has one state active for any given symbol cycle and is functionally equivalent to an NFA as we discussed above; this is achieved through a process known as *subset construction*, which involves enumerating all possible paths through an NFA [55]. Converting an NFA to DFA has the benefit of reducing the runtime to $O(n)$ for n symbols (note that now the runtime is independent of automaton size) and only requires one memory access per input symbol, but frequently causes an exponential increase in the number of states necessary. Subset construction for large automata incurs a huge memory footprint, which may cause performance degradation due to memory overhead in von Neumann machines [56].

Many CPU engines for automata processing or regular expression matching were proposed [57] [58]. In this dissertation, we use Hyperscan as the CPU automata processing engine. Hyperscan [19] [59] is an open-source, high-performance automata-based regular expression matching tool on CPUs. It uses hybrid automata techniques to allow simultaneous matching of large numbers of regular expressions against the input stream.

2.2.2 GPUs

GPUs provide highly parallel computational capabilities. Many works have been done to exploit the parallelism on GPUs for various applications and corresponding programming models [60] [61] [62] [63] [64] [65] [66]. Some of them work on high-speed parallel pattern matching [67] [68] [69] [70] [71]. In this dissertation, we use

DFAGE and iNFAnt2 as our DFA engine and NFA engine on GPUs. Both tools are open-source, allowing us to modify the tool when needed.

DFAGE is a prototype framework for running DFA-based matching on NVIDIA CUDA-enabled GPU cards [20]. DFAGE is an optimized version of a DFA engine running on GPUs, which was inspired by the original work in [67] and [72]. Several optimizations are included, such as fast accepting-state recognition by encoding accepting states with negative IDs, DFA transition tables stored in GPU texture memory, simultaneously processing multiple DFAs and multiple packets, etc. All such features enable the execution of large and complex DFAs with high throughput.

iNFAnt2, an optimized version of iNFAnt [18], is a prototype framework for NFA-based automata processing on NVIDIA CUDA-enabled GPU cards [45]. Several optimizations are included, such as processing multiple NFAs simultaneously, utilizing GPU texture memory for storing NFA transition table, multi-byte input symbol fetches, etc. With all such optimizations, iNFAnt2 can process large and complex NFAs with high throughput and low memory consumption [21].

2.2.3 Others

Several past efforts have proposed modifications to existing von Neumann architectures to specifically increase performance of automata processing workloads. HARE (Hardware Accelerator for Regular Expressions) [73] uses an array of parallel modified RISC processors to emulate the Aho-Corasick DFA representation of regular expression rulesets. The Unified Automata Processor (UAP) [74] also uses an array of parallel processors to execute automata transitions and can emulate *any* automaton, not just Aho-Corasick. As processing in memory (PIM) and processing near memory attract more interest, several prior works propose using the memory system directly for automata processing. Subramaniyan et al. propose using last-level cache for automata processing. They also provide a compiler that can map large NFAs to the proposed architecture and achieve over 15× speedups compared to the AP [75]. Sadredini et al. propose a compact, low-overhead, yet flexible in-memory interconnect architecture for autoamta processing. They evaluate the proposed method using SRAM 8T and reduces the interconnect by 7× [76]. The same group then proposes FlexAmata, a compiler that transforms the automata shapes to support different bitwidth processing and map them to in-memory architectures. This provides insights for next-generation automata processing accelerators [77].

Though the above methods help to improve the performance of automata process-

ing on von Neumann architectures, the memory bottleneck still limits the achievable throughput. Researchers start exploring other architectures that may be more efficient for automata processing.

2.3 Automata Processing on Spatial Architectures

Spatial architectures can process certain computation using reconfigurable processing elements. Such architectures usually provide a large number of hardware resources, allowing processing the computational task with massive parallelism [27]. When using spatial architectures for automata processing, we can directly store automata graphs on hardware and process these automata in parallel, which overcomes the memory bottleneck on von Neumann architectures. In this dissertation, we use Micron's Automata Processor and FPGAs as two examples of spatial architectures.

2.3.1 Micron's Automata Processor

Micron's Automata Processor is an efficient and scalable semiconductor architecture for parallel automata processing [22]. It uses a non-Von-Neumann reconfigurable spatial architecture, which directly implements NFA in hardware, to match complex regular expressions. The AP can also match other types of automata that cannot be conveniently expressed as regular expressions, by describing the NFA directly. The ability to efficiently implement regular expressions or NFA processing makes the AP well-suited for inexact pattern-matching problems such as ER. Use of NFAs avoids the exponential growth in automata size that can occur with deterministic FAs (DFAs), thus achieving high density for the number of automata structures that fit on the AP.

Memory-derived Architecture

The AP is a memory-derived architecture, which exploits the bit-level parallelism of the memory structure. The architecture is shown in Figure 2.2. In a traditional RAM, one needs both the row address and the column address to access a memory cell, while the row address for the AP is the input symbol and each column is one State Transition Element (STE–a state and associated transition rules). The 8-bit input symbol is decoded and provided to the memory. The STEs are connected to the configurable Automata Routing Matrix Structure by some logic. Then the AP invokes

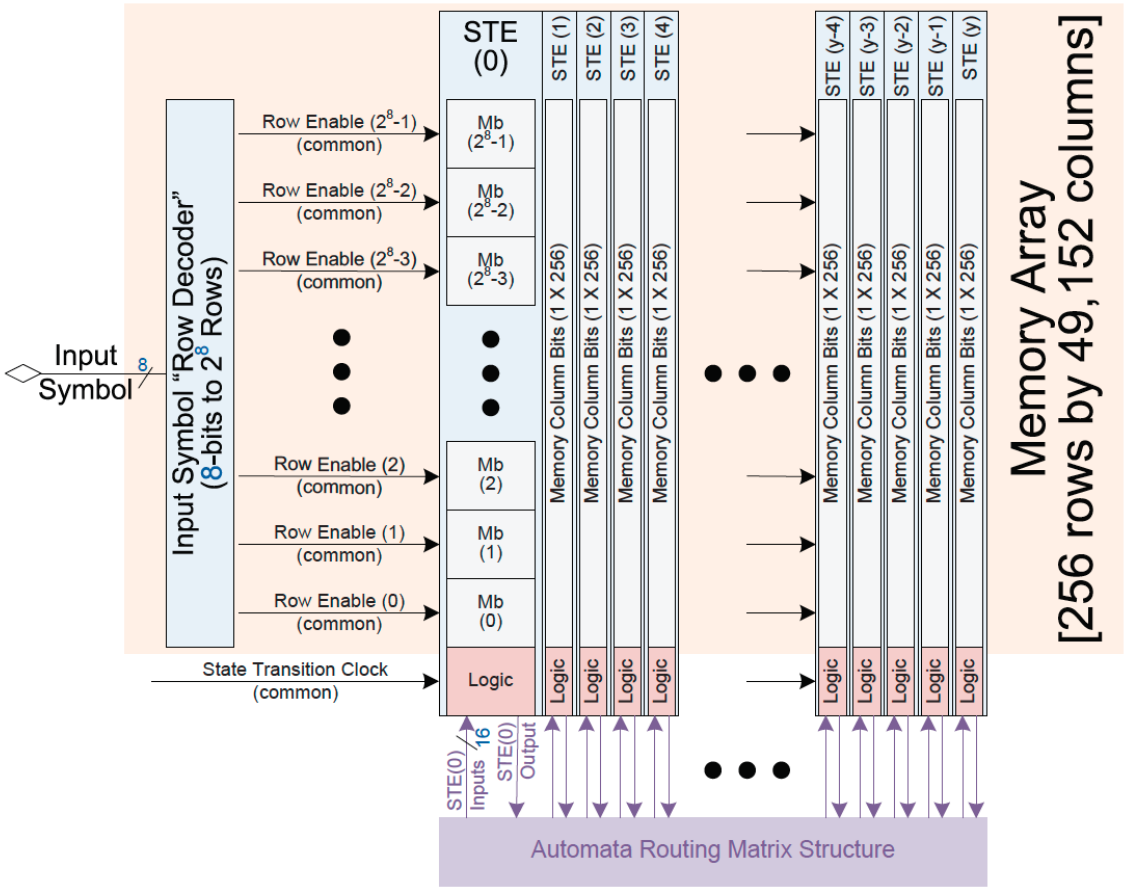

Figure 2.2: Automata Processor Architecture. Each memory column is one STE. [22]

automata activation operations using the routing matrix structure. The details of the AP architecture can be found in [22].

AP Functional Elements

The AP consists of three functional elements: STEs, Counters, and Boolean elements [22]. Each STE can be configured to match a set of any 8-bit symbols and up to 256 different characters can be stored in one STE. The STE activates a set of successor STEs connected to it when the symbols stored in it match the input symbol. STEs can be configured as *start*, *all-input* or *report*, so that they can read symbols from the start of the input sequence, any symbol in the input sequence, or report when a match is found. Counters and Boolean elements are designed to work with STEs to increase the space efficiency of automata implementations and to extend

computational capabilities beyond NFAs. The 12-bit counter will trigger an output event when the accumulated value reaches the pre-defined threshold. The Boolean elements can perform classic logical functions. The AP allows all STEs on the board to inspect the next input symbol in parallel, and it is able to process a new input symbol every clock cycle.

Speed and Capacity

The current generation AP chip (D480) is built on 50nm DRAM technology running at an input symbol rate of 133MHz. The D480 chip has two half-cores and each half-core has 96 blocks. Each block has 256 STEs, 4 counters, and 12 Boolean elements. In total, one D480 chip has 49,152 STEs, 2,304 Boolean elements, and 768 counter elements. Each AP board can have up to 32 AP chips, providing more than 1.5 million STEs.

Programming and Reconfiguration

Automata Network Markup Language (ANML) is an XML language for describing the composition of automata networks. The AP workbench is a graphical user interface tool for quick automata design and debugging. A "macro" is a container of automata for encapsulating a given functionality. The AP SDK also provides C and Python interfaces to build automata, create input streams, parse the output, and manage computational tasks. Furthermore, the symbols that an STE matches can be reconfigured quickly. The replacement time is around 0.24 milliseconds for one block. This feature is helpful when one needs to modify the symbols stored in the AP board without changing the automata structure [13], e.g., for multi-pass algorithms.

2.3.2 FPGAs

NFA

Past implementations of NFAs on FPGA [23] [24] [78] [79] [80] focused on synthesizing *only* regular expression matching circuits for applications such as antivirus file scanning and network intrusion detection. REAPR extends this 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, adapt it for other NFA applications, and detail our process in Section IV.

DFA

Several efforts [81] [82] [83] 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 is easily implemented using BRAM. One benefit to 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 only focuses on the spatial implementation of NFAs.

Results of using spatial architectures for various inexact pattern matching applications [9] [13] [14] [27] [84] show the benefits brought by spatial architectures for automata processing. In this dissertation, we focus on using FPGAs for automata processing. We aim to provide a high-performance, scalable, and user-friendly automata processing engine on FPGAs for inexact pattern matching applications. People can also use this engine as a research platform to explore novel features (as in Chapter 5 and Chapter 6) that could further improve the performance for future automata processing hardware.

Chapter 3

Entity Resolution Acceleration using Automata Processing

Entity Resolution (ER), the process of finding identical entities across different databases, is critical to many information-integration applications. As sizes of databases explode in the big-data era, it becomes computationally expensive to recognize identical entities among all records with variations allowed across multiple databases. Profiling results show that approximate matching is the primary bottleneck. The Automata Processor (AP), an efficient and scalable semiconductor architecture for parallel automata processing, provides a new opportunity for hardware acceleration for ER. We propose an AP-accelerated ER solution, which accelerates the performance bottleneck of fuzzy matching for similar but potentially inexactly-matched names, and use several different real-world applications to illustrate its effectiveness. We compared the proposed method with several conventional methods and achieved both promising speedups and better accuracy (more correct pairs and less generalized merge distance cost) for different datasets.

3.1 Introduction

Entity Resolution (ER), also known as Record Linkage or Purging/Merging problems, refers to finding records that store the same entity within a single database or across different databases [29]. ER is an important kernel of many information-integration applications. For example, Social Networks and Archival Context (SNAC) collects records from databases all over the world to provide an integrated platform for searching historical collections [85]. In such applications, the records of the same

person may be stored with slight differences, because documents come from different sources, with different naming conventions, transliteration conventions, etc. SNAC needs to find the records referring to the same entity despite different representations and merge these records. The intuitive method of solving ER is to compare all possible pair records and check whether a pair represents the same entity.

Determining whether two records represent the same entity is usually computationally expensive. For example, the time complexity of the intuitive method is $O(N^2)$, where N is the number of records. Prior work has proposed different algorithms and computation models to improve the performance [31] [32] [33]. However, the performance is still unsatisfying and the average time used by record comparison is much longer than the cost of simple string comparison [35].

The Automata Processor (AP) is an efficient and scalable semiconductor architecture for parallel automata processing [22] [86] introduced by Micron. It is a hardware implementation of non-deterministic finite automata (NFA) and is capable of matching a large number of complex patterns in parallel. The AP has been used in different fields such as association rule mining [13], bioinformatics [84], string kernel testing [2], natural language processing [14], etc. Such applications need inexact matching and high throughput, just as does ER. Therefore, we propose a hardware acceleration solution to ER using the AP and focus on string-based ER. To illustrate how the AP approach works, we present a framework and evaluate the suitability using several real-world ER problems in different applications.

In summary, we make the following contributions:

1. We propose a novel AP-based hardware acceleration framework to solve string-based Entity Resolution.

2. We present several automata designs for string-based ER, e.g. fuzzy name matching. We apply the proposed approach in SNAC to illustrate the effectiveness of the approach, and generalize it with small modifications for other string-based ER problems.

3. We compare the prototype of the AP approach with several conventional approaches (Apache Lucene, sorting, hashing, and suffix-tree methods) to evaluate the suitability of the proposed method. Results show both higher performance and better resolution accuracy using various datasets from different applications.

3.2 Related Work

Many methods have been proposed to solve ER. One is a domain-independent algorithm [31]. This chapter proposed first computing the minimum edit-distance to recognize possible duplicate records, and then using a union/find algorithm to keep track of duplicate records incrementally. This proposed method achieves around 5x speedup compared with previous methods. Another method sorts the records and checks whether the neighboring records are the same [29]. For approximate duplicates, these researchers define a window size and a threshold of similarity, so that they can find similar records to satisfy application requirements. Apache Lucene is a high-performance search engine and uses a similar method [87]. The difference is that it calculates the score of a document based on the query, and sorts documents instead of every individual record. As databases become much larger, some researchers suggested dividing the original database into small blocks based on prior knowledge and processing smaller blocks, but the method is not always feasible because not all databases are easily divided [32]. However, we are unaware of any implementations of these algorithms using accelerators. Furthermore, we hypothesize that the AP's massive parallelism can achieve much higher performance than these methods. Therefore, we propose an AP-based approach for the Hamming distance-based method.

3.3 Design Details using the AP

In this section, we present how to use the AP to solve string-based ER problems, using the Name Matching problem in SNAC as an example. At the end, we discuss how to generalize the AP approach to other ER problems.

3.3.1 Real-world ER Problems

In ER, *identity attributes* distinguish each entity from one another. *String-based* ER means that the *identity attributes* are strings. In this chapter, we focus on solving real-world string-based ER problems.

When building the SNAC platform, the same person's name may not always be consistent from one record to the next because of typos, mis-spellings, different abbreviation, etc. These differences lead to three major problems. 1) One may miss some correct results when querying a particular record; 2) multiple entries for one entity waste storage space; 3) the duplicated items slow down the search speed. Therefore,

SNAC needs to identify and combine potentially similar records, which is a typical ER problem. Similar problems also exist in DBLP, a website for browsing Computer Science bibliographic information [88]. As DBLP adds records to its database, the contents of the same record may have different representations.

We also present how to identify restaurant records from Fodor's and Zagat's restaurant guides using the AP.

3.3.2 Workflow

The general workflow of using the AP to solve string-based ER problems is shown in Figure 3.1. The CPU first reads the database and extracts the fields of interest from the original database in a pre-processing step, because the database may contain some other information. For example, for the ER problem in SNAC and DBLP, the fields of interest pertain to people's names. We store these fields of interest on the AP board. The input strings are then streamed into the AP and the AP compares the stored contents with the input. If the AP finds a match, it reports back to the CPU. Each record is assigned a number before being streamed into the AP. Based on the reporting STE ID and the offset of the reporting time, the CPU can tell which record finds a match and proceed to combine these records. A reconfiguration phase is needed if the record number exceeds the capacity of the AP. Intuitively, to config the AP for the next batch of names, one can compile new automata structures for the records which have not been processed yet, but this usually takes a long time due to the high cost of routing for these new structures. Instead, we develop canonical matching macros, which new record values can be loaded without reconfiguring the automata structure; the data are then re-streamed. This approach introduces the cost of replacing symbols (milliseconds), but it is usually faster than compiling new structures (minutes). In the following sections, one will see several design choices to take advantage of the fast symbol replacement.

3.3.3 Extracting Name Formats

Intuitively, one can build an automaton for every single record. However, this only works well if the database is small and all records can be stored on one AP board. For large databases, to exploit the fast symbol replacement, we focus on designing a more general automata structure that can be shared by different records [13].

Extracting records formats is a preliminary step for most string-based ER solutions

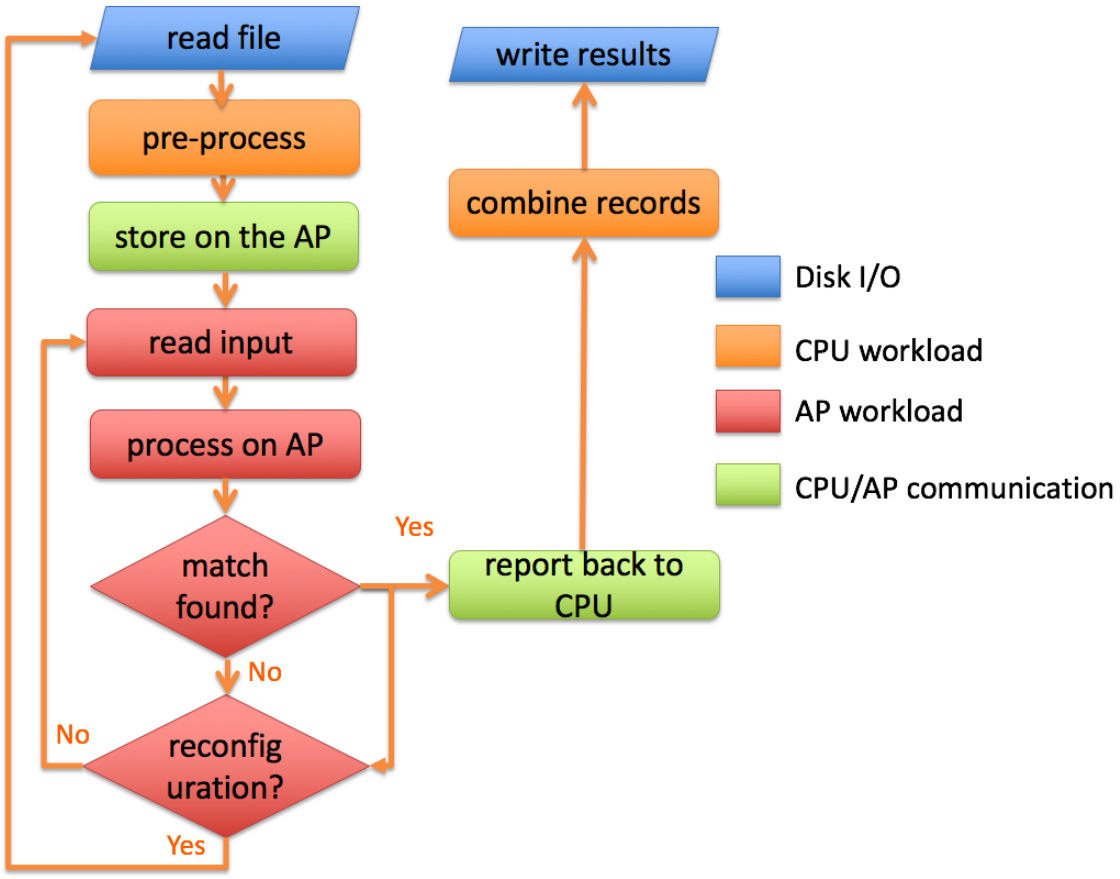

Figure 3.1: General workflow of the AP approach.

using symbol replacement. We start by describing the approach for solving the Name Matching problem in SNAC. One name is usually composed of several sub-names, like family name, middle name, and first name. We only use family name and first name for SNAC, because middle names are both less common and important for correct resolution in SNAC. Family names and first names are sufficient to evaluate the suitability of the proposed approach.

We choose a subset from the whole database randomly as a basis to extract a representative set of formats for family name and first name. Table 3.1 shows common variants of family names and first names. However, not all names can be represented by these formats (fewer than 1%). In this case, we treat it as a failure (no match found). Refinement of these rare cases is left for future work.

Family Name Formats	First Name Formats
Abc (basic)	Bcd (basic)
Abc Bcd	Bcd X.
Abc Bcd Cde	Bcd Cde
Abc II	Bcd X. (Bcd Xyz)
	B. X.
	B. X. (Bcd Xyz)
	B. X. (Bcd X.)
	Bcd Cde (Xyz)
	Bc. Xyz (Bcde Xyz)
	Bcd, Cde
	Bc
	Bcd O. X. (Bcd Opq Xyz)
	Bcd (Bcd X.)
	Bcd Cde Def Efg

Table 3.1: Name formats in SNAC.

3.3.4 Automata for Family/First Name

After extracting name formats, we show how to design automata for these formats. The designs are also important for generalizing the AP approach in other ER problems, because they share similar design ideas and techniques.

Figure 3.2 shows the exact-matching automaton for family names. Though exact-matching automata cannot recognize the same entities with different representations, it is important to understand how the following fuzzy automata work. The three rows correlate with *Abc, Bcd, Cde* in Table 3.1. The first few STEs in each row store the characters in the name to be matched. The subsequent '+' signs are used to pad the remaining positions, so that family names with different lengths can share the same structure. The '$' and the '#' represent spaces and Roman numerals in the database. The ',' STE is configured as a reporting STE. When this STE is activated, it will report. The lengths can be modified according to different dataset characteristics. In this chapter, all the four family name formats in SNAC share this structure, although it may consume more STEs than using different automata for different names. Again, this is to utilize the feature of the fast symbol replacement.

This design may lead to some false positives because it aims to support arbitrary string lengths; all of the STEs after the second STE in a given row are connected to the reporting STE. For example, if the automaton in Figure 3.2 reads *Ada*, it reports a match; but *Ada* is not the name we want. False positives are typically acceptable,

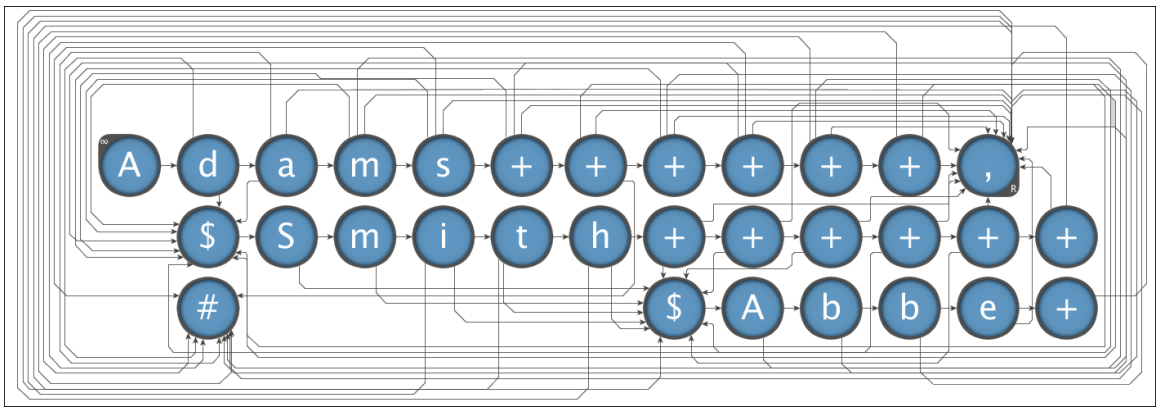

Figure 3.2: Exact-matching automata design for family name (exact match for "*Adams Smith Abbe*").

and we still need to check the first name. The chance that we get a false positive for both family name and first name is small.

However, the exact-matching cannot fully solve the ER problem and we need to execute 'fuzzy' matching. A fuzzy macro in this chapter refers to an automaton that can recognize a string with variances. One fuzzy macro example is shown in Figure 3.3. It matches sequence the $ABCDEF$ and reports when the Hamming distance is ≤ 1. This structure is also used in [84]. Column i corresponds to the ith symbol in the sequence. The STEs in odd rows activate on symbols in the target name and the ones in even rows activate when there are mismatches. The Hamming distance can be extended up to k with more $(2k + 1)$ rows . All macros in this chapter adopt this structure, but with different sequence lengths. For example, in Figure 3.4, the name length of macro *fuzzy* and *fuzzy2* is 11 while the length of macro *fuzzy3* is 5. Furthermore, macro structures are not limited to Hamming distance. One can have macro designs for other distances, like general edit distance in [89].

With these fuzzy macros, we can find the same names with variances (Figure 3.4). The first three rows are used to match the three corresponding parts in family name formats, and we use fuzzy macros in each row to recognize variances of names. The design for the first name (Figure 3.4) is similar to family name, but we need several extra STEs (last row in red rectangle) to process the '.' and parenthesis, which do not exist in family name formats.

The design may produce false negatives if the AP stores a shorter form first. For example, if AP stores '*J.*' first, it will not report a match when it reads '*Janet*', the

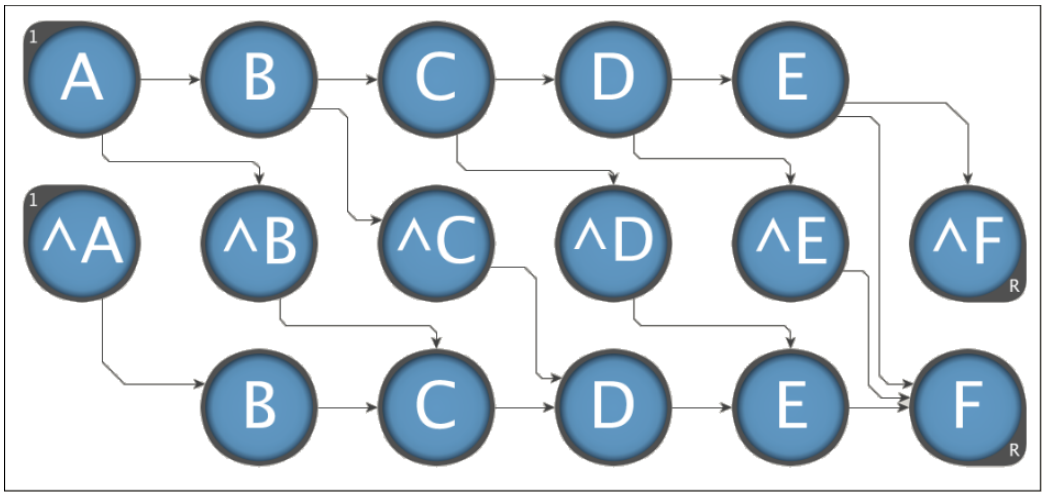

Figure 3.3: Structure of fuzzy macro calculating Hamming distance. It matches sequence *ABCDEF* with Hamming distance = 0, 1.

full form of '*J.*'. This problem causes most of the inaccuracy in our ER outcomes (Section 3.4.3). In this chapter, we consider '.' symbol as a character within a string. However, abbreviations such as '*J.*' often are meant to indicate *J* followed by 0 or more of any character. One possible solution is to use an STE accepting any character when reading '.', but this is left for future work.

3.3.5 Hybrid Version

With these automata designs, we roughly evaluated the approach and found that the STE capacity is the bottleneck. To reduce STE consumption, we use a hybrid version (Figure 3.5) of automata for family name and first name in Figure 3.4. We recognize the names using one single automaton in order to save STEs. The hypothesis is that if the family name is a match, one can compare fewer characters within the first name. The hybrid technique also allows us to use two fuzzy macros instead of three for family name and first name, because it is unlikely that one finds a wrong match for both sub-names. An STE pointing to itself is used to accept all the remaining characters, further reducing STEs consumed. This self-pointing technique is also used when generalizing the AP approach for other ER problems. With all these techniques, we reduce the STEs consumed from 174 to 99 for one name.

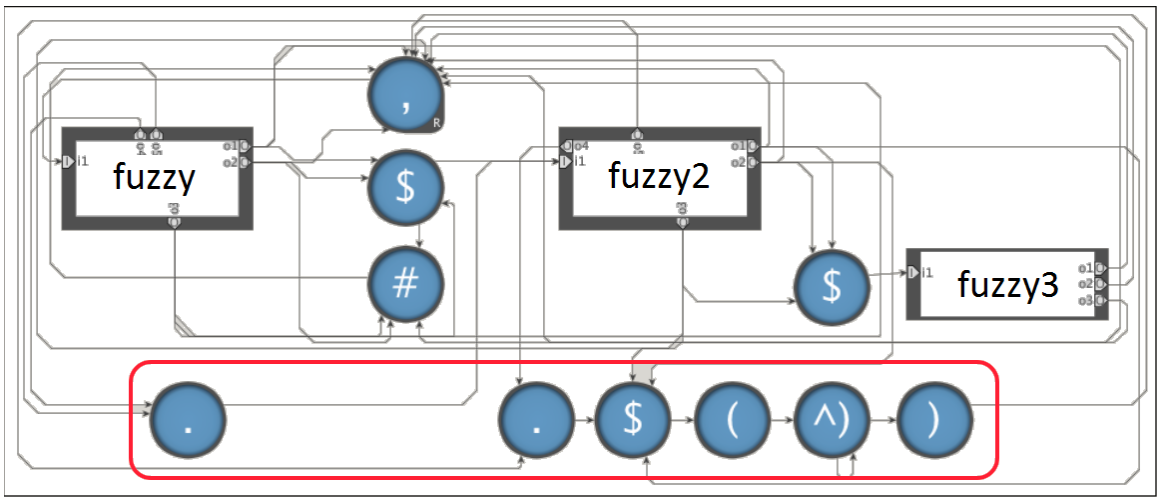

Figure 3.4: Fuzzy-matching automata design for first name (fuzzy macros allow Hamming distance = 0, 1).

3.3.6 Generalizing to Other String-based ER Problems

In this section, we discuss how to generalize the AP approach to other string-based ER problems. As shown in the above discussion, we can build macros that allow different degrees of fuzziness. For example, we can extend the macro in Figure 3.3 to support different string lengths or different Hamming distances. We can also build macro structures for other distances, such as edit distance [89]. Users can generalize the AP approach with these different automata designs to solve their specific ER problems.

First, we show how to generalize the AP approach to solve the ER problem in DBLP. The workflow is the same as in Section 3.3.2. Figure 3.6 shows the automaton design for recognizing similar names in DBLP and middle name is used because most names in DBLP have a middle name. Macro *fuzzy* and *fuzzy1* are used to match first name and family name. These two macros adopt the structure in Figure 3.3 with different lengths (10). The middle part is used to recognize middle name, which is mostly the abbreviation of the full-length middle name. We use three STEs to store characters in middle name. If the middle name is longer, we ignore the remaining part; if the middle name is shorter, similar to what we have discussed in Section 3.3.4, we use '+' to pad the position. The fourth self-pointing STE is used to accept all the characters before the 'space' character. The automaton is similar to the one in Figure 3.5 for SNAC, and they share the same macro structures.

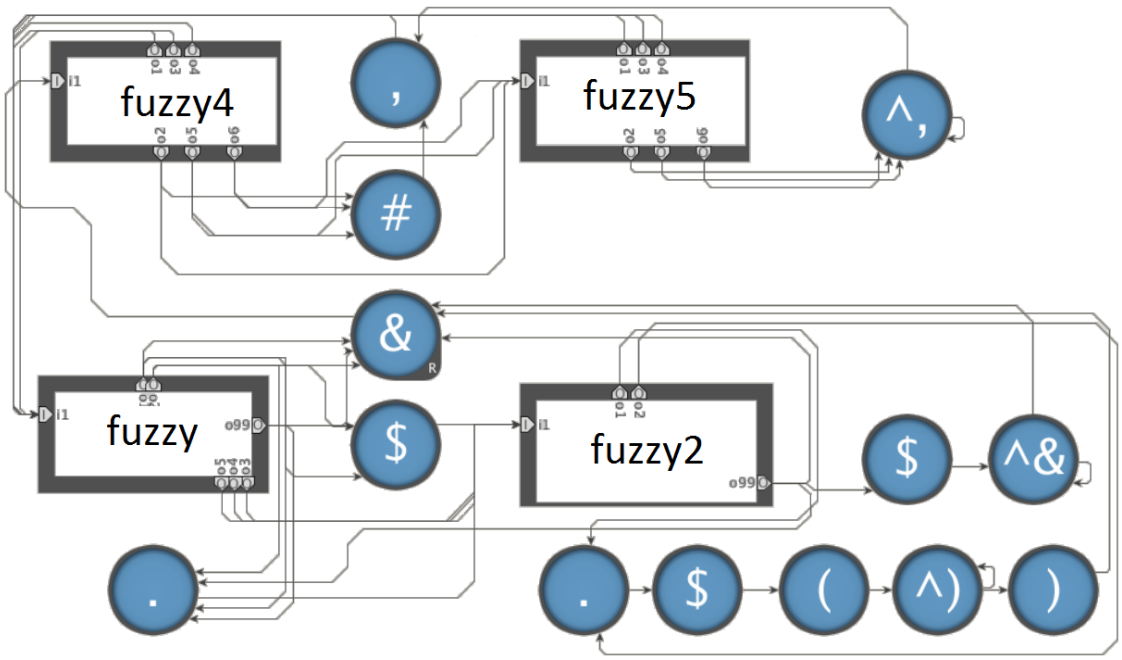

Figure 3.5: Automata design for the whole name in SNAC ('&' is the delimiter of different names).

Secondly, we show how to identify the same restaurant from Zagat's and Fodor's restaurant guides. The workflow is still the same and the automata design is shown in Figure 3.7. We first work on the record formats, finding that most of the variances come from different abbreviations of the same word, like 'deli' vs. 'delicatessens'. This feature makes the automata look more like the exact match design in Figure 3.2 but it can recognize the same word with different lengths. Most of the restaurants' names have fewer than three parts, so we only use three rows to represent different parts inside one name. If the name has more than three parts, we only consider the first three; otherwise, we fill the automaton using the latter rows first. For example, for restaurant 'Carneigie Deli', we store 'Carneigie' in the second row and 'Deli' in the third row. As for the first row, we use '+' as a placeholder as in Section 3.3.4. The second-to-last STE in each row activates itself until it reads the 'space' symbol as discussed in Section 3.3.5.

These two examples show how we generalize the AP approach for SNAC in other string-based ER problems. They all use the same workflow in Section 3.3.2. Even though we need to modify the automata designs to solve the specific problem, the

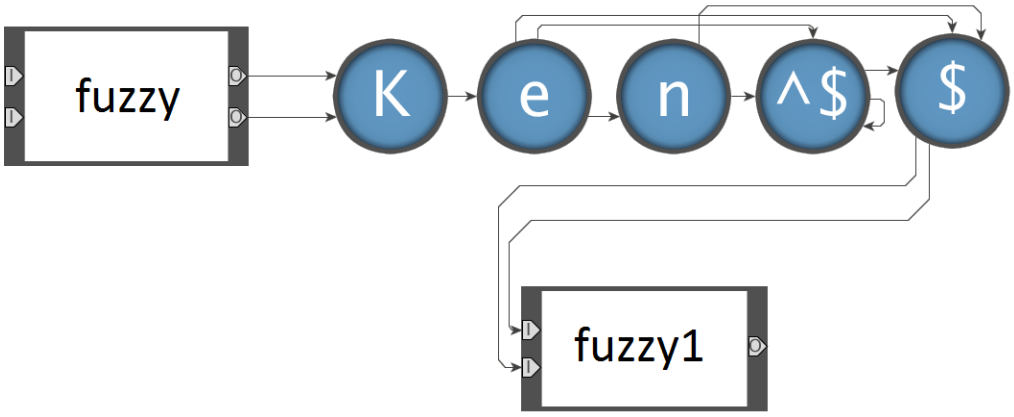

Figure 3.6: Automata design for DBLP ('$' is the delimiter of sub names).

structures are similar and we can re-use many design ideas and techniques, such as macro structures, self-pointing STEs, and using '+' as placeholders.

However, this is still not a universal method. There are some applications for which the proposed method is not suitable. The AP approach did not work well when we tried to resolve consumer-electronics products from online shopping websites. This is because the various representations of products are not due to different spellings; instead, they usually have semantic meanings of words or abbreviations or different descriptions, such as 'PlayStation4' vs. 'PS4' or 'black headphone' vs. 'headphone in black'. In such situations, it is difficult for the AP approach to identify these records, because too many variations are required, and a dictionary of all possible relationships is needed.

3.4 Evaluation

3.4.1 Experiment Setup

To evaluate the suitability of the prototype of the AP approach, we compare both execution time and resolution accuracy of the AP approach with other conventional methods. The experiments are executed on a server with AMD Opteron 4386 Cores (3100MHz). We use an AP simulator to derive the execution time for the AP approach until the real hardware is available. The data used in the following experiments are sampled from different databases, including SNAC [85], DBLP [90], Fodor's and Zagat[91]. We use a random selector to select records from these databases.

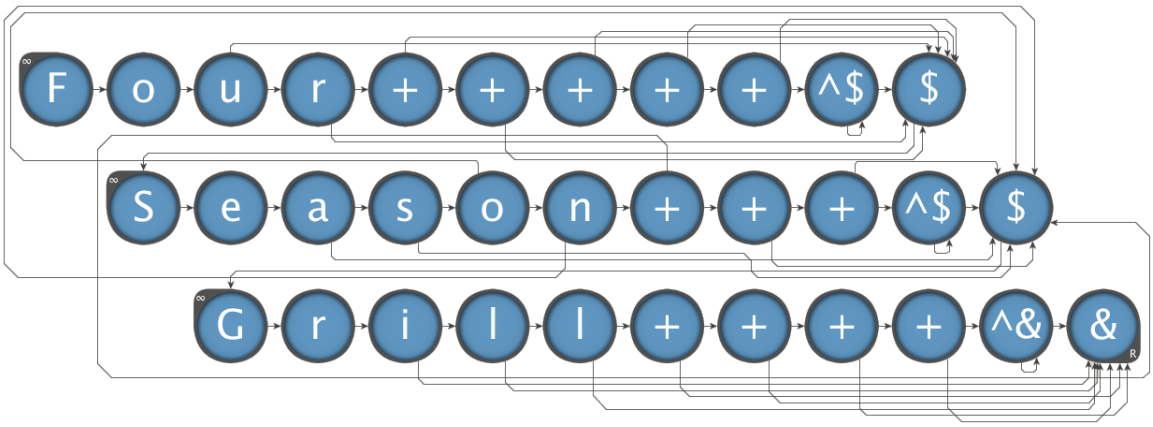

Figure 3.7: Automata design for identifying the same restaurant ('$' is the delimiter of different parts inside a restaurant name and '&' is the delimiter between restaurant names).

3.4.2 Performance

We first compare the performance of the AP approach with conventional methods, including Apache Lucene, a sorting-based method as suggested in [29], a suffix-tree-based method, and a hashing-based method. Apache Lucene is a widely used searching library and supports advanced query types, such as proximity queries, which enables us to execute fuzzy matching [87]. The sorting-based method first sorts the names and then compares the Hamming distance of neighboring names. The suffix-tree-based method builds a suffix tree for the names and searches names against the suffix tree. The hashing-based method builds a hash table for the names and searches the exact names inside the table. We only execute exact matching with these two methods in this chapter. There are some methods which can execute fuzzy matching using suffix tree and hashing-table, but it takes much more time than exact matching.

The matching time is used as the primary metric to evaluate these methods. Because using Apache Lucene involves some other overhead like building indexes for databases, we only collect the time of the search function that executes matching operations. We evaluate these methods with both small datasets and large datasets sampled from SNAC. The results for small datasets are shown in Figure 3.8. When the database is smaller than 10,000, the suffix-tree-based method is the least effective, yet the matching time increases slower than Apache Lucene as the database size increases. The sorting-based method works as well as the hashing-based method

for these datasets. But the sorting-based method achieves better result quality, and the details will be discussed in Section 3.4.3. The AP approach runs faster than all the other four methods for these datasets. The matching time of the AP approach increases almost linearly as the database increases. The slope of the AP approach is nearly flat because the AP can process a new input character every clock cycle. However, the AP can only hold 14,000[1] names at a time; if a dataset is larger than 14,000, a symbol replacement operation is needed to load the next 14,000 names. This shows up in Figure 3.8 as steps in the AP curve. We also compare the AP approach with Apache Lucene with its multiple search support. Apache Lucene supports simultaneously searching multiple records (90 at most), but this does not always work for all datasets, and searching 50 names is the largest number that works for all datasets. The average speedup is calculated over all samples in a given dataset. For example, samples of different numbers of names from SNAC (1,000 to 20,000 names in Figure 3.8) are used to calculate the average speedups against conventional methods (in Table 3.2). Though searching multiple names helps to reduce the matching time, the AP approach still achieves 20.3x speedup compared with searching 50 names simultaneously and up to 373x speedup compared with the suffix-tree-based method.

	Speedup
Sorting	47.9
Suffix-tree	373
Hashing	45.8
Lucene[1]	248
Lucene[10]	75.7
Lucene[50]	20.3

Table 3.2: Average Speedups for small SNAC databases.

To evaluate how the AP approach works for large databases, we then work on larger datasets (from 14,000 names to 140,000 names) from SNAC. Figure 3.9 shows that even when the number of names exceeds the capacity and several reconfigurations are needed, the AP approach still runs at least 8.5x faster than other methods. On average, the proposed method achieves 17x, 33.4x, and 16.9x speedup compared with the sorting-based, suffix-tree-based, and hashing-based methods.

Furthermore, we evaluate the AP approach for DBLP. We can store 45,000 names

[1]The AP capacity in the number of names or other records is a function of the expected record size and complexity (e.g. Hamming distance) of matching.

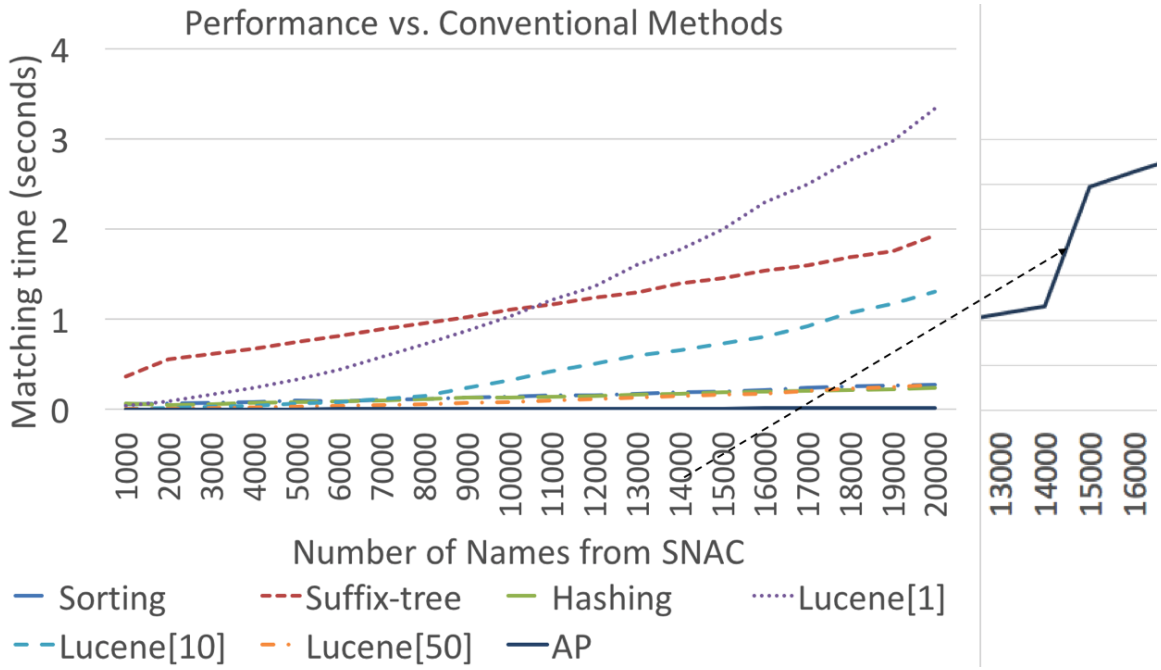

Figure 3.8: Performance vs. conventional methods for small SNAC databases. (X axis represents the number of names, ranging from 1,000 names to 20,000 names.)

with the design in Figure 3.6 on one AP board. We collect the matching time for 1 million to 10 million names, since DBLP has a much larger database. Because the number is larger than the capacity of the current board, we need to replace the symbols and re-stream the input. Figure 3.10 shows that even for much larger datasets, the AP approach still works the best among all the methods. On average, it achieves 3x, 29.7x, and 15x speedup against the sorting-based, the suffix-tree-based, and the hashing-based methods. The speedups are not as high as small datasets when compared with the sorting-based method in SNAC, but we achieve much better resolution accuracy (Section 3.4.3). To achieve similar accuracy as the AP approach does, conventional methods need longer time. For example, we can improve the accuracy of the sorting-based method by increasing neighboring group size. However, the time consumed increases linearly as the size increases. We will discuss how to further improve the AP approach performance for large databases in Section 3.4.4.

In general, the prototype of the AP approach achieves promising speedups compared with conventional methods for both small and large datasets, even when many reconfigurations are needed. The speedups come from the massive parallelism when comparing the records using the AP. We can store a large number of records (14,000

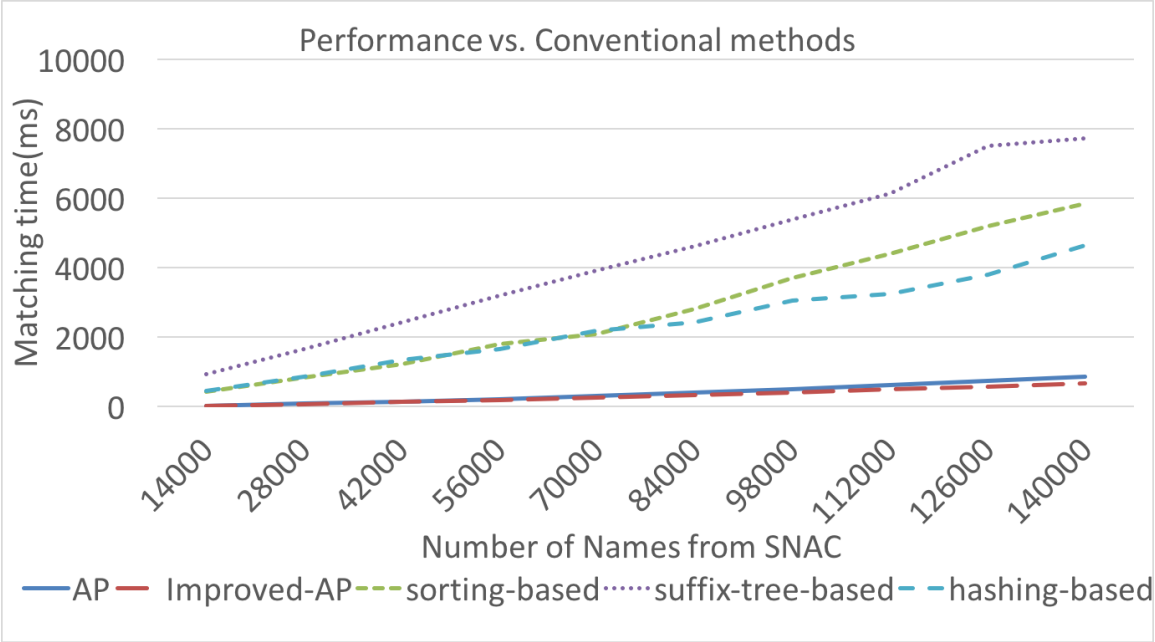

Figure 3.9: Performance vs. conventional methods for large SNAC databases. (X axis represents the number of names, ranging from 14,000 names to 140,000 names. Y axis represents the matching time.)

for SNAC and 45,000 for DBLP) on one AP board and compare these records simultaneously.

3.4.3 Resolution Accuracy

Results quality is also important to study the suitability of the approach; therefore, we use two different metrics (correct pair numbers and generalized merge distance(GMD)) to evaluate the accuracy of the AP approach.

	Correct #	**Pct**	**Merge**	**Split**	**Total**
Apache Lucene	262	80.6%	51	3	54
Sorting	233	71.7%	63	0	63
Hashing	213	65.6%	72	0	72
Suffix-tree	213	65.6%	72	0	72
AP	292	89.8%	30	1	31
Manual	325	100%	0	0	0

Table 3.3: Resolution Accuracy Results for SNAC.

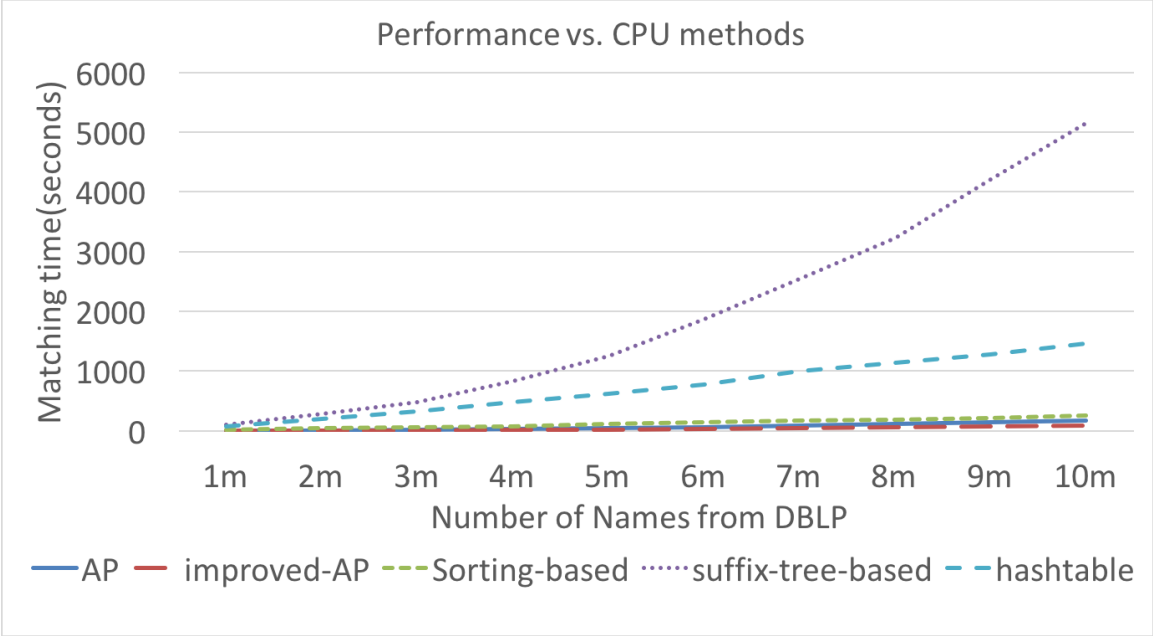

Figure 3.10: Performance vs. conventional methods for DBLP. (X axis represents the database size, ranging from 1 million names to 10 million names. Y axis represents the matching time.)

The first metric is the number of correct pairs (Table 3.3). If there are more than two records in one group, every two records inside the group are counted as one correct pair. The AP approach finds 9.2%, 18.1%, 24.2%, and 24.2% more correct pairs than Apache Lucene, sorting-based, hashing-based and suffix-tree-based methods respectively.

The second metric is GMD [92], which is based on the elementary operations of merging and splitting the records group to correct results. We use a simple version of GMD, where the costs of merging and splitting are equivalent (Table 3.3). The AP approach method only needs 31 operations, while other methods need at least 54. The AP approach needs 50% fewer operations compared with the best conventional method. We observe that most GMD of the AP approach comes from merging operations, which implies that the AP approach may miss some names that should be grouped together instead of recognizing wrong names. To further reduce GMD cost, we can design more complex fuzzy macros in order to identify more similar names. But this may consume more STEs.

We then evaluate the AP approach (Table 3.4) using a subset from DBLP provided

by [90]. The AP approach finds 17% more correct pairs and uses 66% fewer GMD operations than the best conventional method.

	Correct Pair #	**Percentage**	**GMD**
Correct	675	100	0
AP	615	91.4	62
sorting	502	74.4	183
suffix-tree	484	71.7	212
hashing-table	484	71.7	212

Table 3.4: Resolution Accuracy Results for DBLP.

Lastly, we evaluate the design for identifying the same restaurant from Fodors and Zagat. The dataset is achieved from [91] and the matching results are provided. 112 matched tuples should be found and the AP approach finds 105 (93.75%) correct results, while the other three methods only find around 90 (80.36%) correct pairs.

For all three datasets, the AP approach achieves better results quality. This is because the differences in these datasets are usually caused by typos, mis-spellings or different abbreviations, and the fuzzy macros we present in the previous section can recognize these differences and identify the same entities yet with small differences.

3.4.4 Improved AP Approach

When the name number is larger than the AP capacity, we re-stream the whole database. However, we do not need to re-stream all records after reconfigurations. For example, for the ER problem in SNAC, 14,000 names are compared in each round and these 14,000 names do not need to be streamed in the next round since they have already been compared. We improve the algorithm by deleting unnecessary processed names, thus reducing the total number of comparisons on the AP. Figure 3.11 shows that the speedups using the improved algorithm increase from 1.3x to 1.8x compared to the simple AP approach. For the ER problem in DBLP, we also implement the improved algorithm and the results are shown in Figure 3.10 (solid line) and it works 2x faster than the streaming the whole database.

3.4.5 Scalability

For the current generation of the AP, we can compare 14,000 names for SNAC and 45,000 names for DBLP in one pass. The number may not be large enough for some

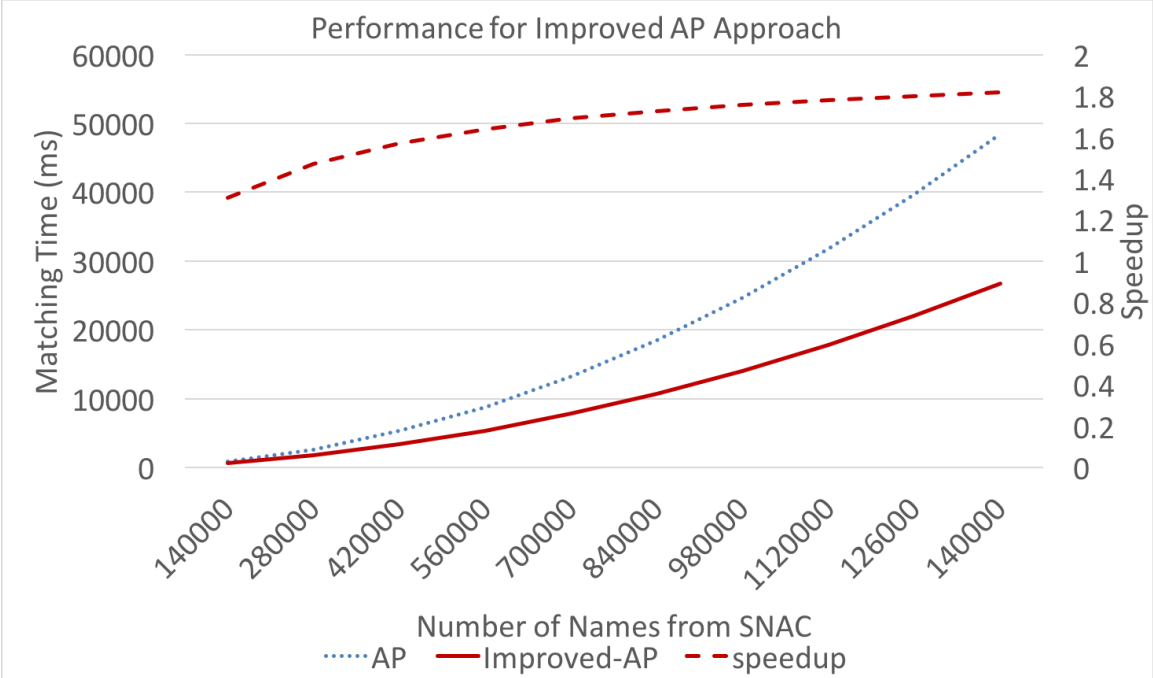

Figure 3.11: Performance for improved AP approach. (X axis represents the database size, ranging from 14,000 names to 140,000 names. Y axis on the left represents the matching time and Y axis on the right represents speedup against original algorithm.)

applications, where databases are much larger, like DBLP. There are several possible ways to address this problem.

The STE capacity is the current bottleneck of the AP approach. Therefore, if we can increase the STE capacity, we can store more records and reduce the number of reconfigurations, thus improving the matching time. In Figure 3.12, we estimate how the matching time varies when STE number increases using the datasets from DBLP. Results show that the speedup increases almost linearly as STE number increases. We achieve 1.97x, 4.91x, and 9.56x speedups if we can have 2x, 5x, and 10x more STEs on one AP board.

The method we use for larger datasets introduces the cost of replacing the symbols and re-streaming the input. We estimate the performance if we can reduce the symbol replacement time or increase the input rate. For relatively small datasets, reducing symbol replacement time helps more to reduce the matching time than reducing re-streaming time, because symbol replacement time takes most of the matching time. Figure 3.13 shows how the matching time in SNAC varies when reducing

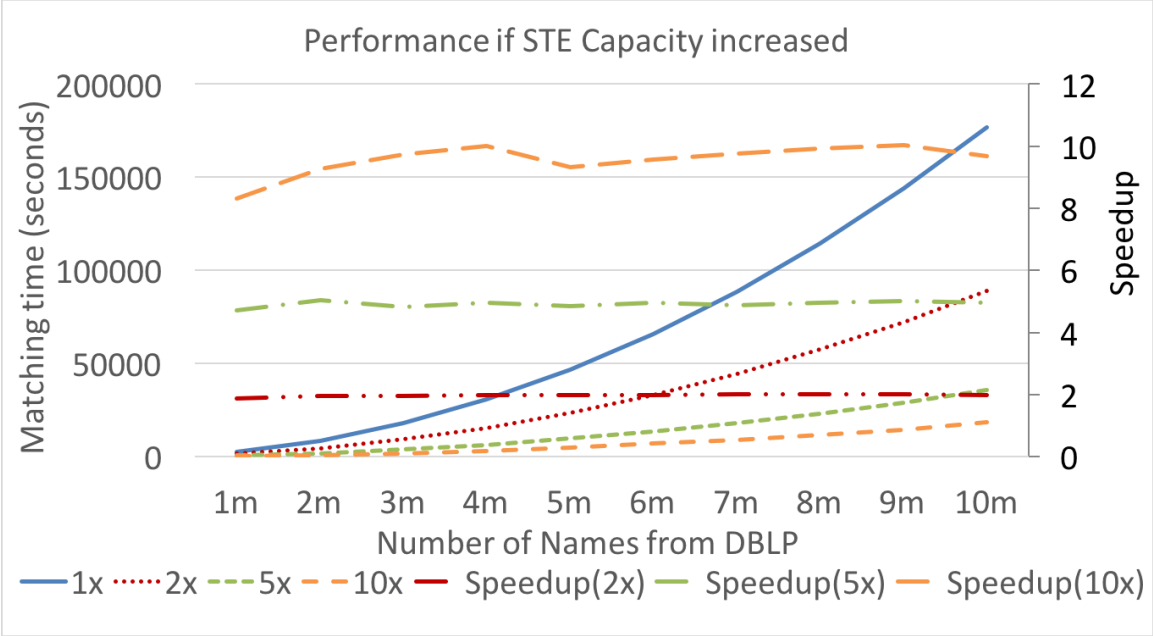

Figure 3.12: Performance if STE capacity increased. (X axis represents the database size, ranging from 1 million names to 10 million names. Y axis on the left represents the matching time and Y axis on the right represents speedup against original performance.)

the symbol replacement time. For the ER problem in SNAC, when the number of names is fewer than 230,000, the symbol replacement consumes more time than re-streaming the input. A reduction of 50% in the symbol replacement time leads to 1.42x to 1.67x speedup and a reduction of 90% leads to 2.16x to 3.63x speedup. The speedup decreases as the number of names increases, which implies that the symbol replacement time becomes less dominant for larger databases. For larger datasets like DBLP, the re-streaming time dominates the matching time. When the input size is 10 million, 94.7% of the matching time is used for re-streaming. In such a case, increasing input rate helps more to reduce the matching time. The speedup increases almost linearly as the input rate increases.

In summary, the AP shows advantages on both performance and result quality compared with different conventional methods. Note that the AP board we use to derive the performance is the first generation. Technology scaling projections and performance estimation suggest that, in the future, we may have a larger capacity and higher frequency, which could lead to even better performance.

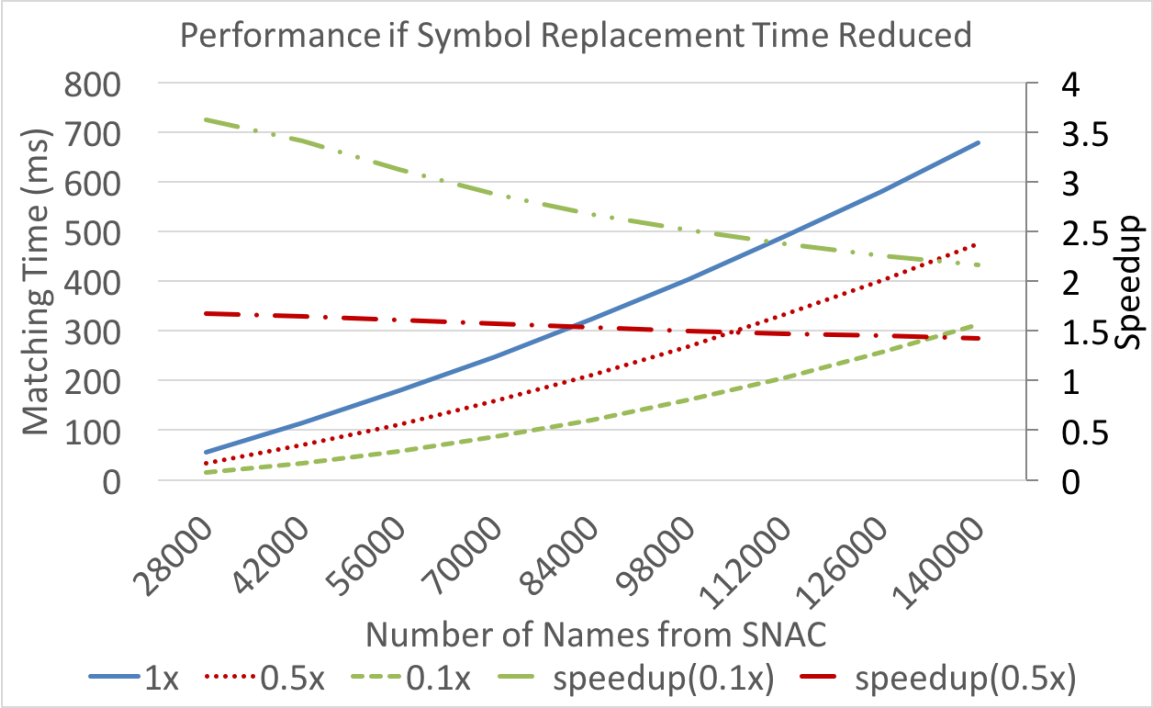

Figure 3.13: Performance if the symbol replacement time is reduced. (X axis represents the database size, ranging from 28,000 names to 140,000 names. Y axis on the left represents the matching time and Y axis on the right represents speedup against original performance.)

3.5 Summary and Future Work

In this chapter, we propose a prototype using the AP to accelerate string-based ER problems. We present the design details of how to solve ER problems in several datasets. The proposed approach makes full use of the massive parallelism of the AP, and compares up to 14,000 names for SNAC and 45,000 names for DBLP in each pass. To evaluate the suitability of the AP approach, we measure both performance and resolution accuracy using various datasets from SNAC, DBLP, Fodors and Zagat. The AP approach achieves promising speeds and also enhances resolution accuracy (more correct pairs with less GMD cost) compared with conventional CPU methods. In summary, the AP shows great potential for accelerating string-based ER problems. Future work includes using the AP to process larger datasets, improve accuracy, and solve other string-based ER problems. This work was published in [93] and [15], and most of the contents are derived from these two papers.

Chapter 4

Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 using Automata Processing across Different Platforms

The CRISPR/Cas system is a bacteria immune system protecting cells from foreign genetic elements. One version that attracted special interest is CRISPR/Cas9, because it can be modified to edit genomes at targeted locations. However, the risk of binding and damaging off-target locations limits its power. Identifying all these potential off-target sites is thus important for users to effectively use the system to edit genomes. This process is computationally expensive, especially when one allows more differences in gRNA targeting sequences. In this chapter, we propose using automata to search for off-target sites while allowing differences between the reference genome and gRNA targeting sequences.

We evaluate the automata-based approach on four different platforms, including conventional architectures such as the CPU and the GPU, and spatial architectures such as the FPGA and Micron's Automata Processor. We compare the proposed approach with two off-target search tools (CasOFFinder (GPU) and CasOT (CPU)), and achieve over 83× speedups on the FPGA compared with CasOFFinder and over 600× speedups compared with CasOT. More customized hardware such as the AP can provide additional speedups (1.5× for the kernel execution) compared with the FPGA. We also evaluate the automata-based solution using single-thread HyperScan (a high-performance automata processing library) on the CPU. HyperScan outper-

forms CasOT by over 29.7×. The automata-based approach on iNFAnt2 (a DFA/NFA engine on the GPU) does not consistently work better than CasOFFinder, and only show a slightly better speedup compared with single-thread HyperScan on the CPU (4.4× for the best case). These results show that the automata-based approach provides significant algorithmic benefits, and that accelerators such as the FPGA and the AP can provide substantial additional speedups. However, iNFAnt2 does not confer a clear advantage because the proposed method does not map well to the GPU architecture. Furthermore, we propose several methods to further improve the performance on spatial architectures, and some potential architectural modifications for future automata processing hardware.

4.1 Introduction

Clustered Regularly Interspaced Short Palindromic Repeats (CRISPR) exist in prokaryotic DNAs. The CRIS-PR/Cas system is an immune system which defends against foreign genetic elements [36]. CRISPR/Cas9 is one version of the system that attracts researchers' interest, because it can be modified to edit genomes [37] [94]. One can deliver the Cas9 nuclease together with a guide RNA (gRNA) into a cell and edit the cell's genome at targeted locations defined by the gRNA. Genome editing using CRISPR/Cas9 has been a popular technique since it was first introduced. For example, researchers are trying to use cells edited by CRISPR/Cas9 to fight cancers [39]. The CRISPR/Cas9 system is also being used to cure other diseases with genetic causes, such as amnesia, muscular dystrophy, etc [40].

However, efficiently finding all correct locations to edit the genome, without modifying other locations, is still the bottleneck of using the CRISPR/Cas9 system, because the gRNA also binds to locations with slightly different DNA sequences [41]. This makes the process of finding all potential *off-target* sites (genome locations sufficiently similar to the gRNA targeting sequence) computationally expensive, especially when one allows more differences from the reference sequence. The time complexity of searching for exact matches is $O(n * l * L)$, where n is the number of gRNA targeting sequences to be searched, l is the length of the query sequence, and L is the length of the reference genome. The time complexity is even worse when differences are allowed. There are several local off-target search tools (running on local workstations), but the performance is not satisfying, even with the GPUs. It could take hours/days to search for potential off-target sites when only allowing a few mis-

matches [42]. The performance of web tools (providing web interfaces for users to provide gRNA targeting sequences and reference genomes) is even worse.

Existing CPU and GPU tools either are restricted to the number of mismatches or fail to return results due to the computational bottlenecks. As Moore's Law slows down, accelerators have attracted more interest. Spatial architectures such as the FPGA and Micron's Automata Processor (AP) provide better performance because they can process automata in massive parallelism by laying out a huge number of automata graphs directly in hardware. The FPGA can be flexibly configured by users to implement a specific algorithm (automata processing in this chapter) with a large amount of logic blocks. REAPR [25] extended prior work on regular expressions to automata processing, allowing us to implement the automata-based method to search for gRNA off-target sites on the FPGA. The AP is an efficient architecture designed for parallel automata processing [22] [95], and has been used in many different domains such as association rule mining [13] [96], sequential pattern mining [16], tree mining [97], entity resolution [15], random forest [1], natural language processing [14], string kernel [2], pseudo-random number generator [98],etc. These applications require inexact matching against many patterns, which is similar to searching for potential gRNA off-targets.

In this chapter, we propose an automata-based solution, to identify potential off-target sites by allowing any Hamming distances in a reference genome. We evaluate the proposed approach across four different platforms (CPU, GPU, FPGA and AP) and compare with two state-of-the-art solutions (CasOFFinder [43] and CasOT [44]). The proposed method leads to over 83× speedups on the FPGA compared with CasOFFinder (GPU) and additional speedups can be achieved by the AP. Furthermore, we evaluate the proposed automata-based method using HyperScan [19], a high-performance automata processing library for the CPU, as an alternative CPU implementation for the proposed approach. The results show that even with single-thread HyperScan, we still achieve promising results (over 29.7×) compared with CasOT (CPU). However, the newly proposed automata-based method with iNFAnt2 [45] on the GPU does not confer a clear advantage because it does not map well to the GPU architecture. These results show significant algorithmic benefits of the automata-based approach, and the potential speedups can be achi-eved by hardware acceleration for automata processing.

In summary, this chapter makes the following contributions:

1. We propose an automata-based approach to search for potential gRNA off-

target sites and implement the approach across four various platforms (CPU, GPU, FPGA, and AP).

2. We present several automata designs for searching for off-target sites with different requirements, such as different Hamming distances or mismatches in different regions of a potential sequence.

3. We evaluate the proposed automata-based method and compare against two state-of-the-art tools (CasOF-Finder and CasOT). Promising speedups are achieved by the FPGA (over 83×) compared with CasOFFinder. Additional speedups could be achieved by more customized hardware such as Micron's AP.

4. We evaluate the automata-based method on the CPU using HyperScan and achieve over 29.7× speedups, showing the algorithmic benefit of the automata approach. However, the proposed method on the GPU does not confer a clear advantage, because it does not map well to the GPU architecture.

5. We discuss how to further improve the performance and support larger datasets on spatial architectures (the FPGA and the AP), and propose several potential architectural modifications for future automata processing hardware.

4.2 Related Work

Several methods have been proposed to search for potential gRNA off-target sites on local workstations [99] [100]. CasOFFinder is a fast off-target search tool [43]. It is written in OpenCL, making it portable across diverse platforms such as CPU and GPU. It supports an unlimited number of mismatches and different PAM sequences. However, it runs much slower as users allow more mismatches. CasOT is another popular potential off-target search tool [44]. It divides the targeting sequence into non-seed and seed regions, and allows different Hamming distances in each region. There are also no limits on how many mismatches are allowed in each region. Similarly, it runs slower as more mismatches are allowed. It could take more than a day to find a relatively large number of mismatches (e.g., larger than five). Some researchers use DNA alignment tools, such as PatMaN [101], Bowtie [102] or BWA-like algorithms [103], to search for potential off-target sites. However, these tools are not designed to solve this specific problem. They cannot recognize PAMs and the performance is not as good as CasOFFinder and CasOT. Furthermore, tools such as PatMaN and Bowtie have limitations on the number of allowed mismatches. Unlike these methods, the proposed method in this chapter solves this search problem using

automata processing, which allows any desired edit distance (we focus on Hamming distances in this chapter) and can benefit from hardware architectures (FPGA and AP) to accelerate such computation.

4.3 Automata Processing on Spatial Architectures

Automata Processing can be defined as a directed graph where the nodes store the states of the automata and the edges store the transition rules between states. To be specific, each node stores the symbols to be matched and reads the symbol from the input stream. All current active states compare with the next input symbol. When the node matches with the input symbol, it will activate all the nodes connected to it by the directed edges. Each automaton has at least one "start" state to initiate the processing and one or more "accept" states to report when a match is found in the input.

Spatial architectures can process certain computation using reconfigurable processing elements. For example, the FPGAs use logic gates and RAM blocks to implement a specific algorithm. Spatial architectures usually provide a large number of hardware resources. For example, the AP A480 board can store up to 1.5 million states. Therefore, the spatial architecture can process automata by placing the automata graphs directly in hardware instead of storing all states and transition rules in the memory as conventional architectures do. It reduces the cost of memory access on each input symbol to search for the next states to be activated. With massive hardware resources, the spatial architecture can process a large number of automata simultaneously.

4.4 CRISPR/Cas9 System

The CRISPR/Cas system was originally found in bacteria immune systems as a way to fight against foreign genetic elements [36]. We list the terminologies used in the chapter in Table 4.1. When a foreign genetic element enters the cell, a Cas complex recognizes it and cleaves it into small fragments and adds a new spacer to the end of the CRISPR. The new array is then transcribed into an RNA, which will direct the Cas complex to the foreign DNA. The targeted DNA will then be destroyed or cleaved. This system was later applied in genome editing as shown in Figure 4.1 [37]. The gRNA contains the targeting sequence and works together with the Cas9 complex

Term	Definition
CRISPR	Clustered Regularly Interspaced Short Palindromic Repeat. A region originally found in bacteria to defend foreign intrusions.
Cas	CRISPR Associated protein, the active enzyme in Type II CRISPR system.
gRNA	Guide RNA, works with Cas9 to target and bind to the specific location. It is also known as single guide RNA or sgRNA.
gRNA targeting sequence	The nucleotides before the PAM sequence, mostly 20 DNA characters.
Target sequence	The genome sequence the gRNA targets at, which can be incorporated into the gRNA and the PAM.
PAM	Protospacer Adjacent Motif, following the gRNA targeting sequence. Necessary for Cas9 to bind target sequence.
Off-target effects	Cas9 cleavages at unwanted locations because the gRNA binds to unintended locations with sequences similar to the gRNA.
Potential off-target sites	Locations in the reference genome that could be the off-target binding sites within a specified edit distance of the gRNA targeting sequence.
Reference genome	The genome the gRNA searches against.
Complementary reference genome	The reverse order of the reference genome, with complementary DNA characters.

Table 4.1: Terminology

to bind to the reference genome. They bind to the specific location (complementary to the gRNA targeting sequence followed by a PAM) and can be further used to edit the target sequence. One major problem is that the gRNA and Cas9 do not bind only to the desired position, but also to other locations where some DNA characters are different from the sequence but close enough to bind, leading to the off-target effects [41]. In this chapter, we aim to accelerate the process of identifying all these potential off-targets.

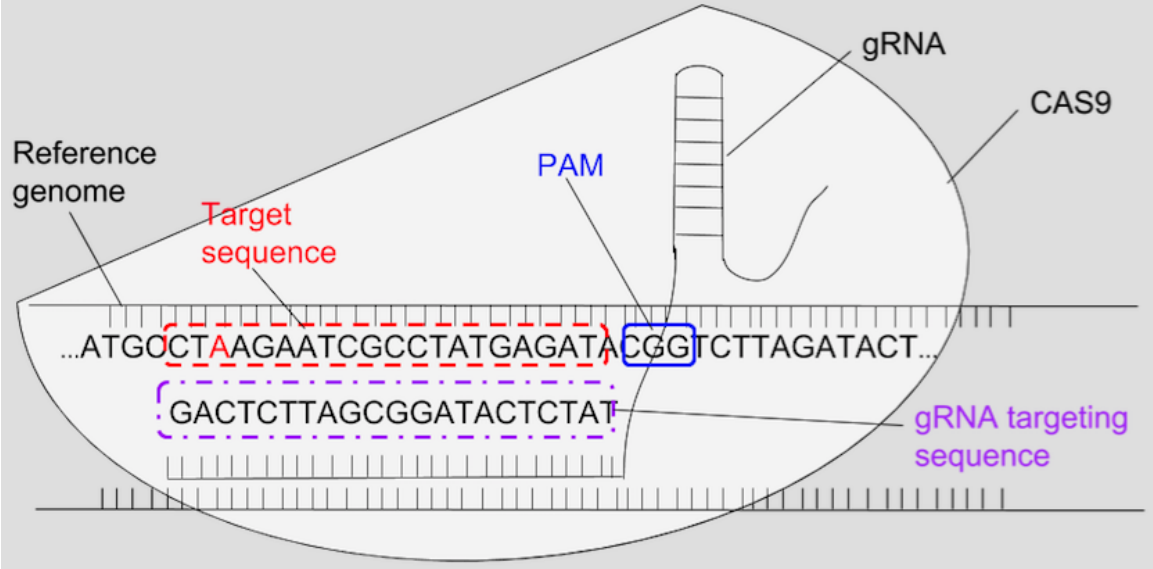

Figure 4.1: gRNA and CAS9 bind to the target sequence [104].

4.5 gRNA Off-target Sites Search using Automata Processing

In this section, we first present several general automata structures recognizing sequences with differences and use these structures to build the automata recognizing potential off-target sites. Later, we will show the general workflow of the automata-based method.

4.5.1 Hamming Distance Automaton

Figure 4.2 shows an automaton design recognizing the sequence "$TAATATAG$" when the Hamming distance is shorter than 3. There are five rows and eight columns in the example. The STEs in the ith column store the ith character in the sequence. The odd rows store the DNA characters one wants to match. Because there is no standard for whether uppercase or lowercase letters are used, this design stores both – both forms can be checked simultaneously, at no extra cost. The even rows match on any other symbol except for the desired character, thus capturing an increase in the Hamming distance. The STEs in odd rows connect to the next STE in the same row and the next STE in the next row. The STEs in even rows connect to the next STE in the next row and the next STE in the third row below them. This allows processing to proceed whether or not the current symbol matched. The structure

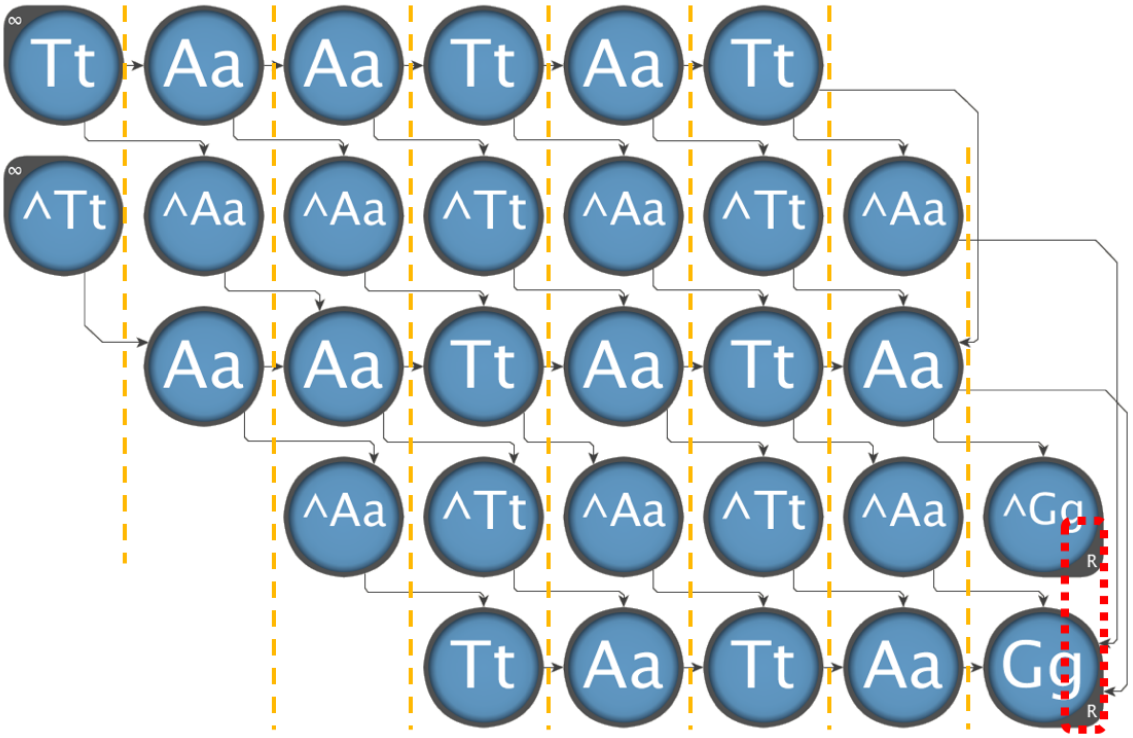

Figure 4.2: Hamming Distance Automaton.

can be extended to recognize sequences with more mismatches, but will consume more STEs. One needs $(2k + 1)$ rows to recognize k mismatches. The STEs in the first column is configured as *all-input*, reading from any position in the input. As indicated in Figures 4.2, the STEs in the last column are configured as *reporting* and they will trigger a report when they are activated. This structure was also used to search for motifs [84]. However, the structure cannot be directly used to search for gRNA target off-target sites. It does not support specifying the regions of differences or identifying PAM sequences, but it can be used as a component to build a larger and more complex automaton to identify gRNA off-targets.

4.5.2 No Consecutive Mismatches Automaton

In Figure 4.3, we present another automata structure for Hamming distance, but it does not allow consecutive mismatches. In this example, we still allow two mismatches, but no two mismatches take place consecutively. This is particularly useful because researchers have discovered that if two mismatches take place consecutively, it is less likely the location containing the consecutive mismatches is a correct off-

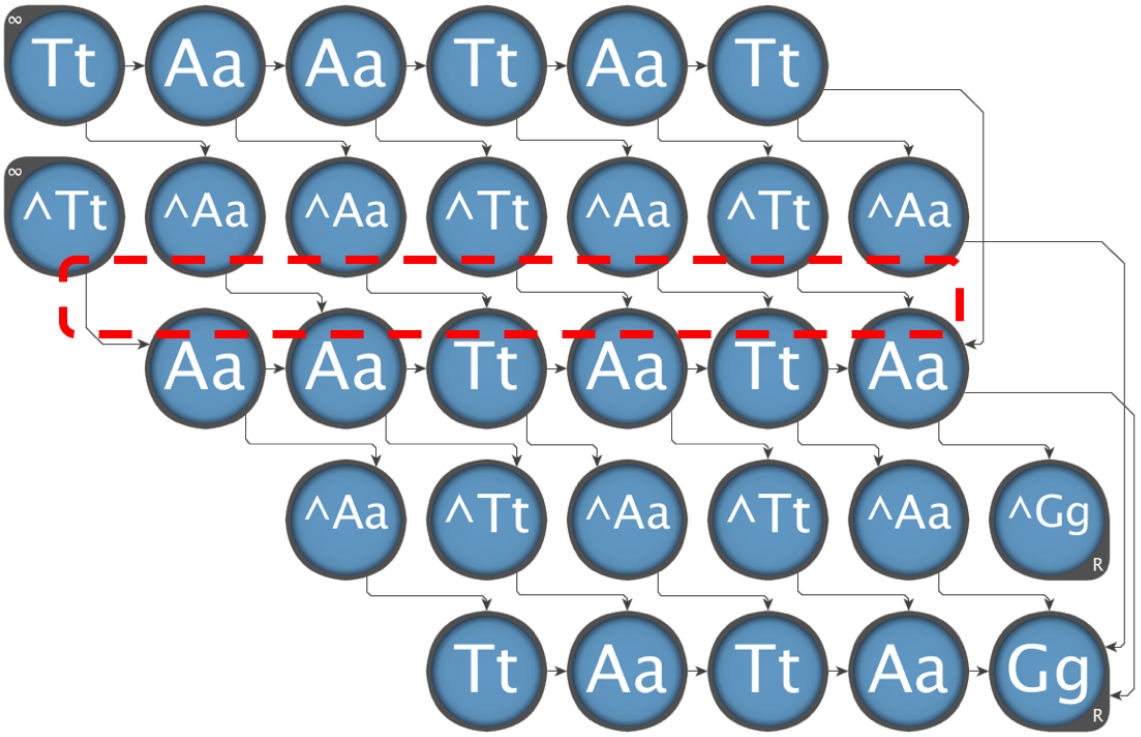

Figure 4.3: Hamming Distance Automaton with no consecutive mismatches.

target site [105]. This design is similar to Figure 4.2, but as highlighted in the center of Figure 4.3, we do not connect the STE in the second row, where the mismatch takes place, to the STE in the fourth row. This assures that one mismatch can only be followed by a matching character. Two more STEs are eliminated at the bottom left corner.

The above examples show that the automata structures for recognizing Hamming Distances are straightforward. With small modifications, we are able to process complex queries. These are just two examples to illustrate how to build these structures. Users can modify these structures or propose a new structure to solve their specific problem. The differences are not limited to Hamming distance. For example, automaton for edit distance can be found in [106] and [107]. The rest of the section presents how we use these structures to build an automaton to identify gRNA off-target sites.

4.5.3 Mismatches in Whole Sequences

Depending on where mismatches take place, different automata designs can be used. For example, CasOFFin-der allows the mismatch to take place in any position in the

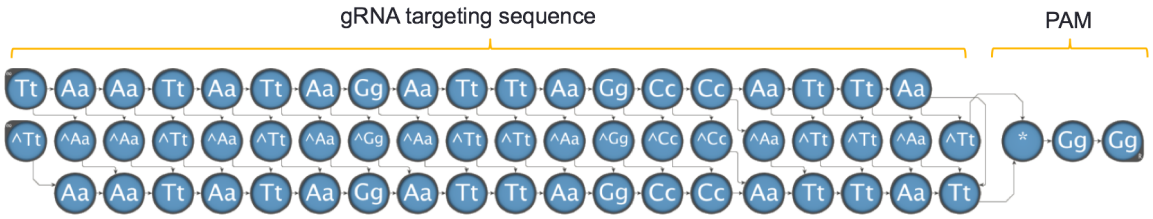

Figure 4.4: Allowing mismatches in any position.

gRNA targeting sequence. Figure 4.4 represents the design for this requirement. The design is straightforward and we use the Hamming distance automata to match the gRNA targeting sequence with one mismatch allowed. We then connect the STEs in the last column in the gRNA targeting sequence to the start of the PAM sequence. No mismatches are allowed in the PAM sequence, so we just connect one STE to another. The STE storing the last character is the reporting STE. The STE storing "*" matches with any input symbol so we can match PAM sequences, such as "NGG" in streptococcus pyogenes, where "N" refers to any DNA character in {AaTtGgCc} in this context.

4.5.4 Mismatches in Different Regions

Some tools propose dividing the sequences into regions and allowing different Hamming distances in different regions. For example, CasOT divides the targeting sequence into a non-seed region (the first 8 characters) and a seed region (the next 12 characters) [42]. Users can specify different numbers of mismatches in these two regions. This may help to improve the quality of potential off-target sites, especially when scoring the potential off-targets. We show one example automaton for CasOT in Figure 4.5. This example allows 2 mismatches in the non-seed region and 1 mismatch in the seed region. Both regions use the Hamming distance automata structure, but with different lengths and different distances. We connect the STEs in the last column in the non-seed region to the STEs in the first column in the seed region. We then connect these regions together. This is just one example of more complex queries. Users can divide the sequence into any number of regions, and apply different automata structures in each region.

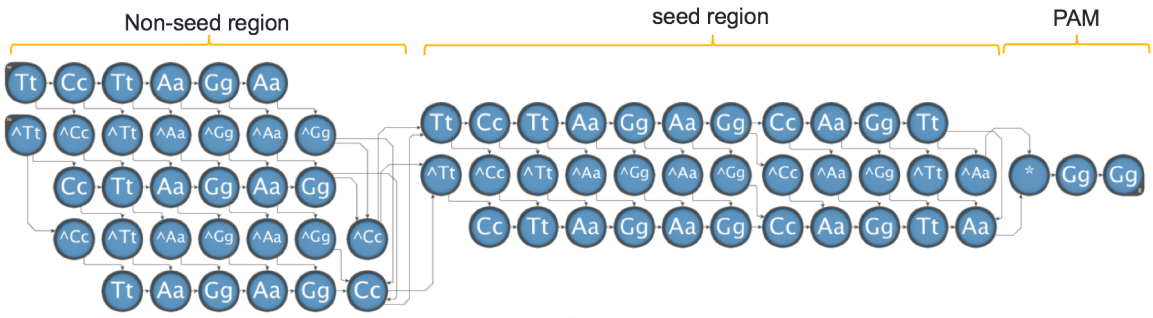

Figure 4.5: Allowing mismatches in different regions.

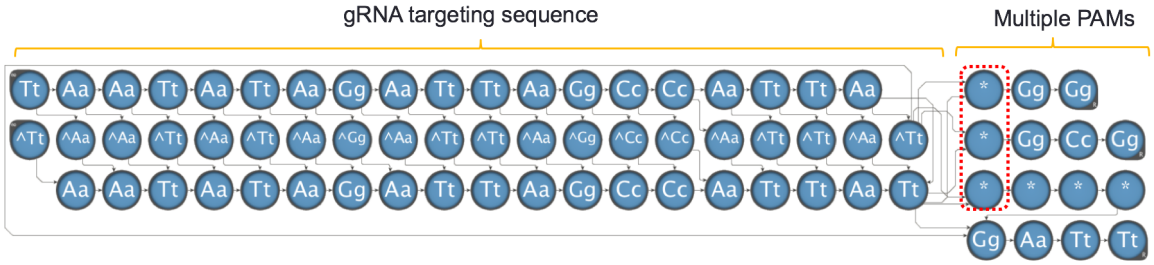

Figure 4.6: Allowing mismatches in any position with multiple PAM sequences.

4.5.5 Multiple PAMs

Users sometimes need to identify potential gRNA targeting sequences with different PAM sequences. This is straightforward using automata processing. We just connect the STEs storing the last DNA character in the gRNA targeting sequence, regardless of which automaton structure is used, to all the starting STEs in the PAM sequences (highlighted in Figure 4.6). For example, similar to what we described in Figure 4.4, we add two more PAM sequences ("NGCG" and "NNNNGATT") in Figure 4.6 and connect the STEs in the last column of the gRNA targeting sequence to the starting STEs in the new PAM sequences. The STEs storing the last DNA character in the PAM sequences are configured as reporting STEs. Because the reporting STEs have different IDs, we can differentiate which PAM sequence finds a match. For the current AP hardware, STEs can have up to 16 output connections, so we can recognize 16 different PAM sequences simultaneously. If one wants to recognize more than 16 PAM sequences, they need to duplicate the structure for the gRNA targeting sequence and connect to the remaining PAM sequences until all PAM sequences are processed.

4.5.6 Workflow

With the above automata, we can identify potential off-targets using automata processing. We will use the AP as an example to illustrate the workflow of the automata-based approach (Figure 4.7). The CPU first extracts the gRNA targeting sequences and PAMs from the database in a pre-processing step, because the original database may contain other information other than gRNA targeting sequences and PAMs. We then store these queries using the automata presented above on the AP board. The reference genome is streamed into the AP afterward and the AP compares the stored queries with the genome. If the AP finds a match, it reports. Based on the reporting STE ID and the offset of the reporting cycle, we can recognize which query finds a potential off-target, and at which position in the genome. Many existing tools support finding potential off-target sites for both strands of the reference genome, because of the double strand structure of DNA. If this is the case, we also need to stream the complementary reference genome to the AP. A reconfiguration phase is needed if the query number exceeds the capacity of the AP. The symbol replacement feature of the AP allows fast replacement of patterns, if the structure of the automata graphs remains unchanged. This feature makes the reconfiguration phase negligible (milliseconds), but one needs to stream in the reference genome again after the reconfiguration.

The workflow of REAPR on the FPGA is similar, but without the fast symbol replacement. Supporting REAPR with symbol replacement is left for future work. Users need to recompile for larger datasets, which may take a longer time to process large datasets. The workflow of iNFAnt2 is also similar, but users need to convert the input patterns to NFA/DFA files before streaming the reference genome to the GPU.

4.5.7 Fast Complementary Genome Processing

In Section 4.5.6, we discussed how to process the complementary reference genome and we used this method for evaluation in Section 4.6. An alternate method can be used to solve this by storing the complementary sequences "A"→"T", "T"→"A", "G"→"C", "C"→"G") of the gRNA targeting sequence in the automaton. This method works faster if one has a small set of gRNA targeting sequences and can store all the queries and complementary sequences on one FPGA board or one AP board, because we do not need to stream the complementary reference genome to

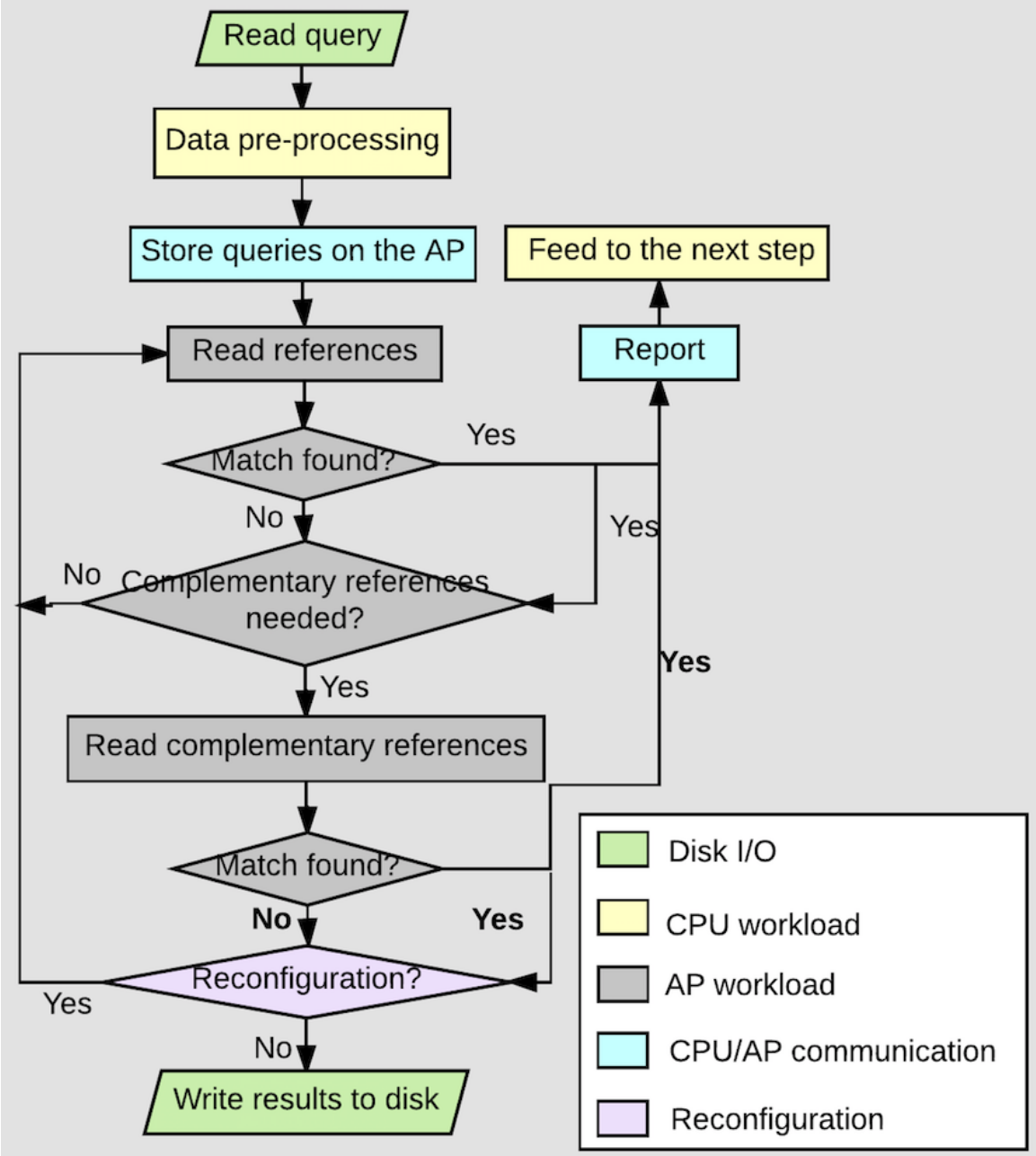

Figure 4.7: Workflow on Micron's AP.

the hardware. The example in Figure 4.8 recognizes the complementary sequence "TCTAGAGC-TCTAGAGCAGTA-NGG") in Figure 4.5. This structure connects the complementary PAM sequence to the complementary seed region and then the complementary non-seed region. The first STE in the complementary PAM sequence is the new starting STE and the last two STEs in the complementary non-seed region are the reporting STEs. All the STEs store the complementary DNA character in

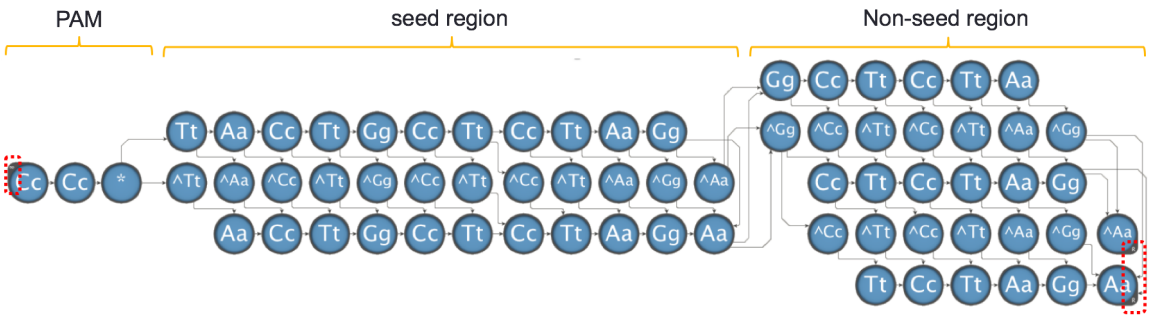

Figure 4.8: Process complementary sequence.

the original sequence in reverse order, so the new sequence to be recognized is "CCN-TACTGCTCTAGA-GCTCTAGA".

All these automata structures described in the above sections can be used by HyperScan on the CPU, REAPR on the FPGA, iNFAnt2 on the GPU, and Micron's AP.

4.6 Performance Evaluation

To evaluate the performance of the proposed approach, we compare with CasOFFinder (GPU) and CasOT (CPU). We implement the automata approach to match the tasks that CasOT and CasOFFinder solve, so they provide the same accuracy. Four different platforms are evaluated: HyperScan on the CPU, iNFAnt2 on the GPU, REAPR on the FPGA, and Micron's AP. The experiments are executed on a server with an Intel Core i7-5820K CPU (3.3GHZ) with 32GB RAM and an NVIDIA Tesla K40c GPU with 12GB memory. For FPGA results, we use an Alpha Data ADM-PCIE-KU3 board equipped with a Xilinx Kintex UltraScale XCKU060 FPGA and two 8GB DDR3 memory banks, hosted in a system with an Intel Core i7-4820K CPU (3.7GHz) with 32GB RAM. All results are actual runtimes except for the AP, because the AP hardware is not yet available. But it is simple to estimate the kernel execution time on the AP, because the input processing rate is fixed at 133MB/s (one character per cycle). The number of queries that can be placed on the board, assuming a 32-chip Micron's D480 AP board, determines how many passes through the input are required.

The human genome is used as the reference genome (3.2 billion base pairs). We develop two different generators to produce gRNA targeting sequences plus PAM sequences, because CasOT and CasOFFinder have different input formats and have

various specifications of where to allow mismatches. We use these generators to study how numbers of queries and mismatches affect the performance. These synthesized datasets are only used to study the performance projections. One can use real datasets to study the biological relationship between the reference genome and the gRNA targeting sequence using the proposed method. More details of the generators will be presented in the following sections.

In this section, we use the total runtime and the kernel execution time (the time used for recognizing off-target sites) as criteria. We compare the performance with CasOFFinder and CasOT separately because they use different input formats and allow mismatches in different regions in gRNA targeting sequences.

4.6.1 CasOFFinder

CasOFFinder is a fast off-target search tool written in OpenCL, portable across the CPU and the GPU. We compare with CasOFFinder on the GPU to achieve its best performance. We use the design in Figure 4.4, because CasOFFinder does not isolate the position in gRNA targeting sequences. The generator for CasOFFi-nder generates the input sequences based on the format specifications. The first line stores the location of the reference genome. The second line is the desired gRNA targeting sequences along with PAM sequences. The sequences are followed by the Hamming distance. The first 20 characters of each query are the gRNA targeting sequence and are selected randomly from DNA character set {AaTtGgCc}, because no real large datasets are available. Without loss of generality, the synthesized datasets help us to study the performance of different approaches on various platforms. The PAM sequence and the number of mismatches are specified by users. We use "NGG" as an example of the PAM sequence to illustrate the suitability of the proposed automata-based method. We presented how to support multiple sequences in Section 4.5.5.

Spatial Architectures

Figure 4.9 and Figure 4.10 show that spatial architectures such as the FPGA, consistently outperform CasOFFind-er by at least 14× for both large number of queries and long Hamming distances.

In Figure 4.9, we present the total runtime and the kernel execution time for REAPR. REAPR works much faster than CasOFFinder for all datasets. The runtime of REAPR depends heavily on the number of queries. Many reports could be

generated for each symbol, and these reports need to be preserved for post-processing on the CPU. This massive data transfer, especially from the reports, has a significant impact on the overall performance on the FPGA. REAPR shows better kernel execution performance than the AP for query numbers (e.g. 30 seconds for 100 queries and 40 seconds for 500 queries). However, for large query numbers (more than 1,000 queries), the AP starts to work faster. We only show the results of queries fewer than 1,000 on the FPGA because of the limitation of the maximum supported memory bitwidth (i.e. 512 bits) of the FPGA system we use. We will discuss how to process large datasets on the FPGA in Section 4.7.1. For the AP, as long as we can store all the queries on one AP board, the kernel execution time stays constant (48 seconds). Processing the reference genome and the complementary reference genome each takes 24 seconds. Since the AP connects to the host CPU by PCIe interface as the FPGA board does, if we assume the AP uses the same report architecture as the FPGA does, we could achieve another 1.9× speedup for the total runtime compared with REAPR. The runtimes of CasOFFinder and single-thread HyperScan increase linearly as query numbers increase. CasOFFinder is only 2.1× faster than single-thread HyperScan for the best case.

In Figure 4.10, we show the runtimes for different Hamming distances. We choose two examples (n = 500 and $1,000$); similar results are found for other query numbers. For REAPR, both the kernel execution time and the total runtime stays constant as we allow more mismatches for a specific number of queries since it only depends on the number of queries. However, the runtimes of CasOFFinder and HyperScan increase as more mismatches are allowed. For a smaller number of mismatches (m = 1 and m = 2), HyperScan is almost as fast as CasOFFinder. CasOFFinder starts to run faster as more mismatches are allowed, but only slightly better. As for the AP, no matter how many mismatches we allow, as long as we can store the queries on one AP board, the kernel execution time stays constant. In our experiments, we allow 5 mismatches at most, because when allowing more than 5 mismatches, CasOFFinder works too slowly for us to obtain runtimes. However, the gRNA also binds to locations with more than 5 mismatches, in which situation CasOFFinder cannot serve its intended purpose. Table 4.2 shows the maximum sequence numbers we can store on one AP board when allowing different mismatches using Micron's AP compiler. The possible speedups are also shown presented assuming the AP use the same reporting architecture as the FPGA. Compared to the total runtime of CasOFFinder, the AP could run over 180× faster. "NA" in the table refers to the

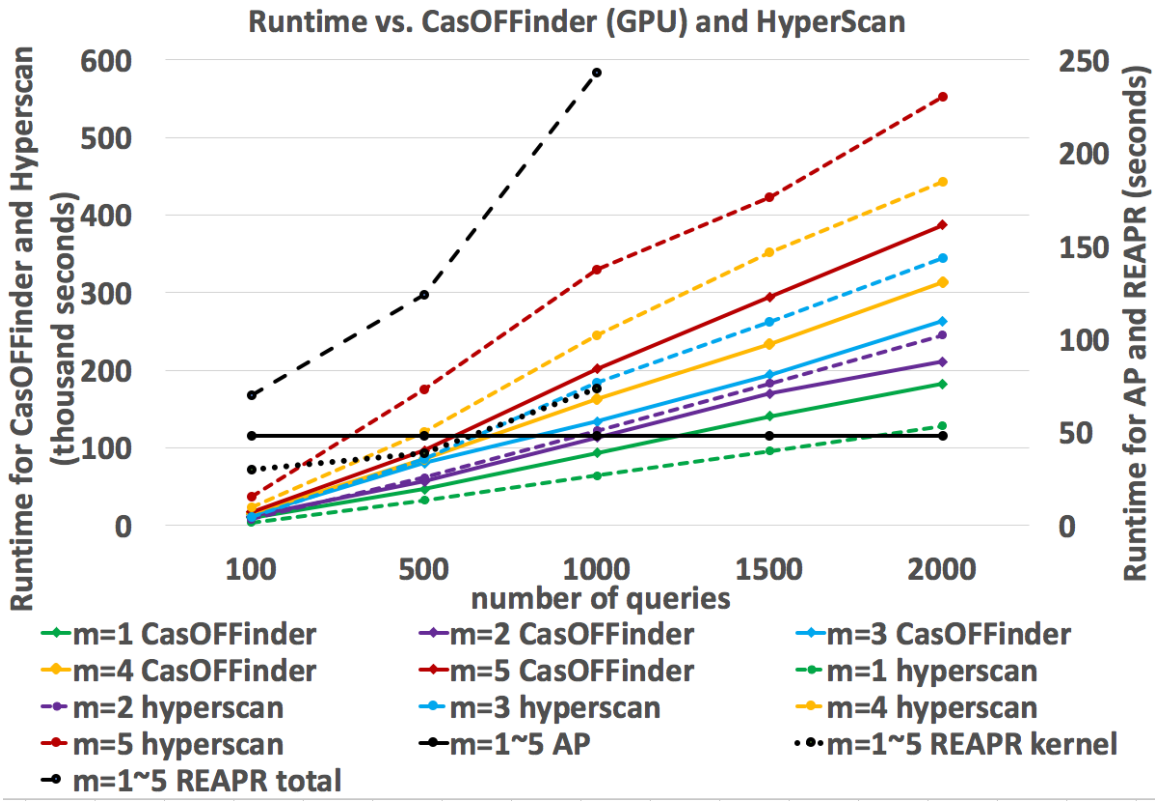

Figure 4.9: Runtimes vs. CasOFFinder for different numbers of queries. m is the number of mismatches. Dotted lines are runtimes of HyperScan and solid lines are runtimes of CasOFFinder. The lines with the same color refer to the same number of mismatches. Black lines represent the results of the AP and REAPR.

cases when CasOFFinder works too slowly and does not return results in ten hours, implying the speedup is even larger. The AP works especially well when comparing a large number of queries with many mismatches allowed.

Mismatches	**1**	**2**	**3**	**4**	**5**	**6**
Max number (K)	22.0	12.6	7.2	5.0	3.4	2.0
Speedup nq=500	53	65	91	98	110	NA
Speedup nq=1,000	73	89	105	127	157	NA
Speedup nq=2,000	88	102	127	150	186	NA

Table 4.2: Max queries stored on one AP board and possible speedups against CasOFFinder for different numbers of mismatches. nq is the number of queries."NA" refers to the cases when CasOFFinder does not finish within 10 hours.

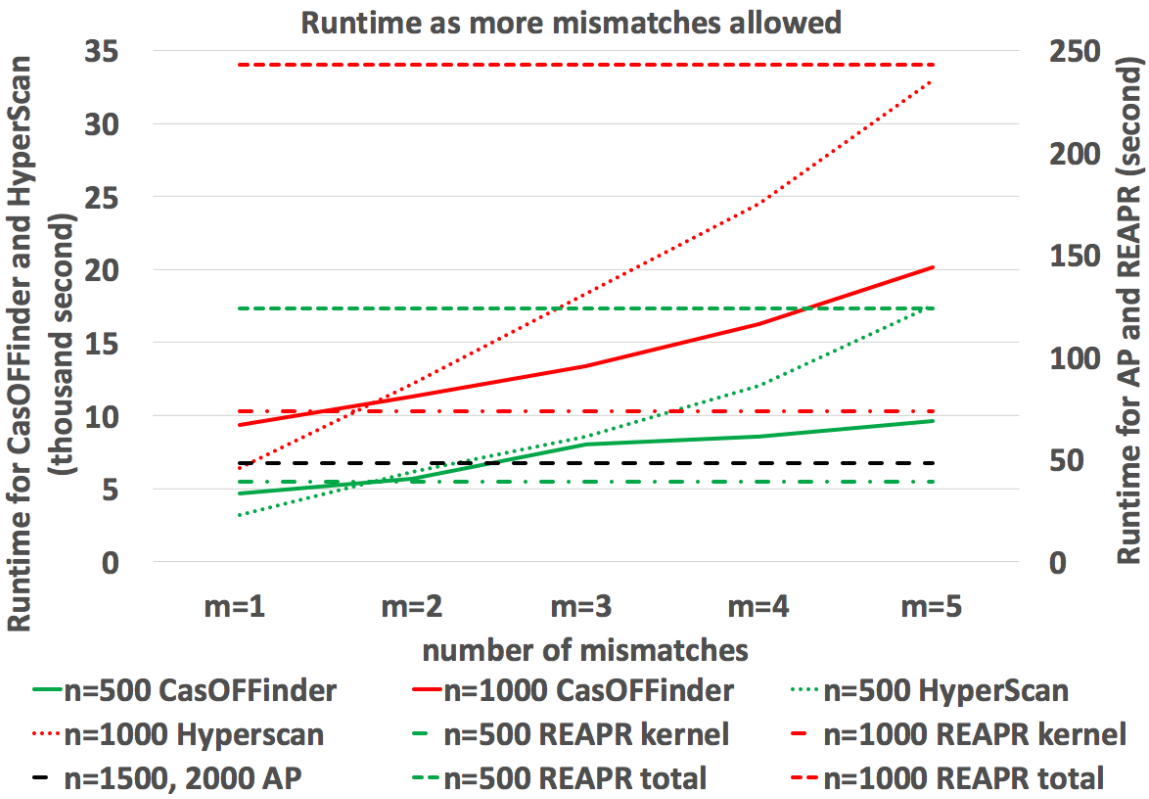

Figure 4.10: Runtimes vs. CasOFFinder for different mismatches (m). Lines with the same color refer to the same query number. The black line represents the runtimes for the AP.

iNFAnt2 on GPU

As Figure 4.11 shows, the automata-based method on the GPU with iNFAnt2 (both the NFA engine and the DFA engine) does not confer a clear advantage compared with CasOFFinder. While the NFA engine does achieve a speedup of 2.4× where n = 500 and m = 1, it is slower by 1.7× where n = 2,000 and m = 5. For smaller query numbers or shorter Hamming distances, the DFA engine works faster than the NFA engine. However, because of the state explosion of the DFA for large automata, the DFA engine does not work for larger pattern numbers or longer Hamming distances. For example, the DFA engine stops working when n = 100, m ¿ 4, because the DFA table size is too large and exceeds the capacity of the GPU memory (12GB).

iNFAnt2 (DFA and NFA) only works faster than single-thread HyperScan on the CPU for some cases. For the best case, iNFAnt2 (DFA) is 4.4× faster (n = 2,000, m = 1), and iNFAnt2 (NFA) is 1.7× faster (n = 500, m = 1). This is because both NFA and DFA engines in iNFAnt2 suffer from the divergence problem. For NFA engine,

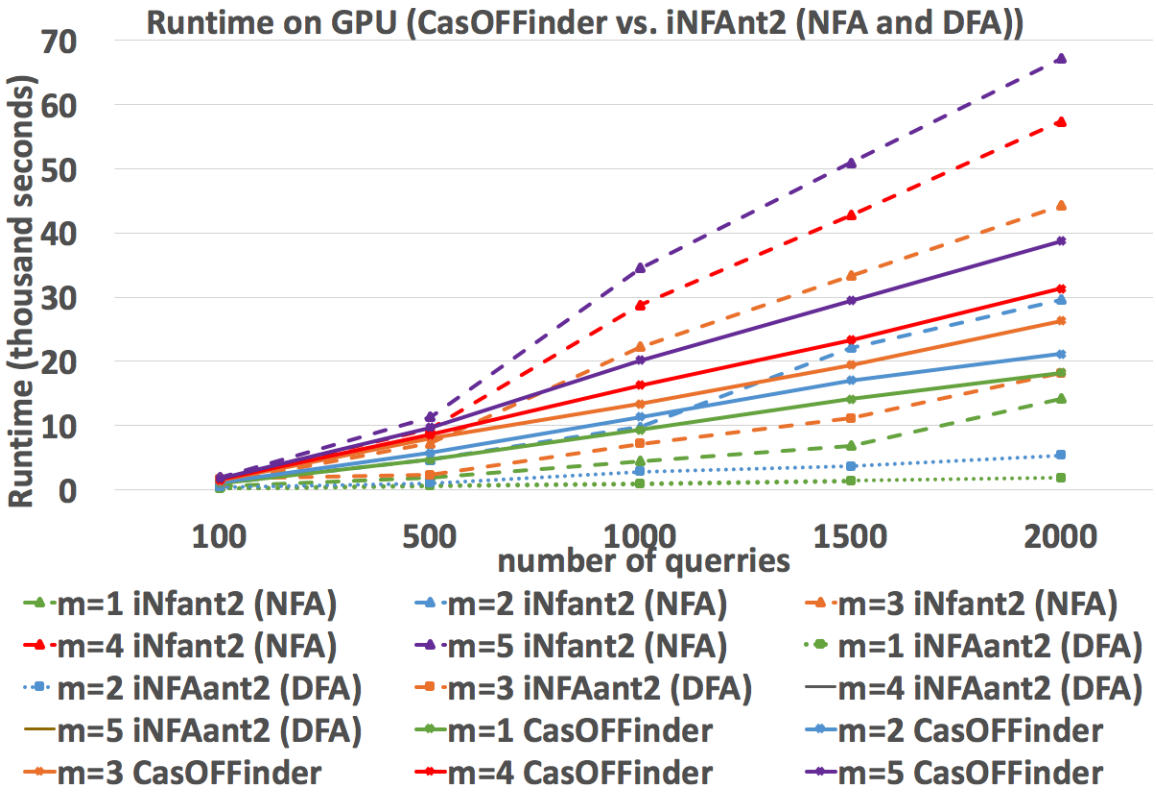

Figure 4.11: Runtimes of iNFAnt2 (NFA & DFA) for different query numbers. m is the number of mismatches. Dotted lines are runtimes of the NFA engine and solid lines are runtimes of the DFA engine. The lines with the same color refer to the same number of mismatches.

for each input symbol, multiple CUDA threads within a thread block are launched to examine all possible transitions corresponding to that symbol across every state in the automata. For the automata used to search for gRNA off-target sites, we only use four DNA characters (both upper and lower cases) plus the "*" character, but the number of transitions on each input character is large, which requires a large number of sequential thread-block iterations. Despite the large number of transitions for each symbol, the number of active states is relatively small (approximate 20-30× smaller than the number of the examined transitions), which potentially causes degraded performance due to thread divergence. For DFA engine, we assign each independent automaton (pattern) to its own CUDA thread. Even if there are enough automata to fill the threads, their memory access for transition lookup and matching behavior are divergent. Therefore, the automata-based approach does not map well to the GPU architecture.

The above results clearly show the suitability and advantage of the proposed

automata-based method using spatial architectures, such as the FPGA and the AP. Furthermore, compared with CasOFFinder (GPU), the method using automata processing with single-thread HyperScan on the CPU is only slightly slower, implying the automata-based method provides substantial algorithmic benefit. iNFAnt2 only provides a limited advantage over CasOFFinder, because the automata-based approach is not well mapped to the GPU architecture.

4.6.2 CasOT

CasOT is an off-target search tool on the CPU. It divides the targeting sequence into non-seed and seed regions and allows different Hamming distances in each region. Therefore, we use the design in Figure 4.5 to compare with CasOT. The generator for CasOT produces two lines for each query. The first line is the name of the sequence and the second line is the gRNA targeting sequence followed by the PAM sequence. We generate the first 20 characters randomly from DNA character set {AaTtGgCc} of each query and use "NGG" as an example for the PAM sequence.

Spatial Architectures

Figure 4.12 and Figure 4.13 show that the automata-based method on the FPGA always works faster than CasOT for different query numbers and Hamming distances (over 29.7× speedups).

In Figure 4.12, the total runtimes of REAPR increase as the query number increases, and REAPR is at least 7.5× faster than CasOT. Higher speedups are achieved for larger datasets. Since the bitwidths of the memory interface currently supported on the FPGA are 32, 64, 128, 256, and 512 bits, the runtime for one particular number of queries will be governed by the I/O bitwidth closest to it (e.g. 500 queries to 512 bits). REAPR shows better kernel execution performance than the AP for small numbers of queries ($\leq$ 512 queries), while the AP outperforms the REAPR for larger numbers of queries. For the AP, since we can store all the queries on one AP board, the kernel execution time stays constant (48 seconds). The total runtimes of CasOT and HyperScan increase linearly as the query number increases. For a small number of queries, CasOT runs faster than HyperScan. However, as the query number increases, the runtime of CasOT increases significantly. Many data points are missing in this figure for CasOT, because it runs too slowly and cannot return results in 24 hours. HyperScan beats CasOT for most of the cases and over 29.7× speedup

is achieved based on the results we can get from CasOT. Figure 4.12 seems to suggest that HyperScan has the highest runtimes, but this is because CasOT cannot even run for larger query numbers. Therefore, speedups of HyperScan are even greater for such cases. We also evaluate the automata-based method on iNFAnt2, but only get 1.88× speedup for the best case compared with HyperScan (n = 700 and m = 1), similar to the results in Section 4.6.1.

Figure 4.13 show the performance for different Hamming distances. We only compare small query numbers (50 and 100), because CasOT runs extremely slowly for larger datasets. The total runtime and the kernel execution time of the FPGA stay constant; while the runtime of CasOT increases almost exponentially as more mismatches are allowed. Compared to what we can get from CasOT, we achieve over 600× speedups on the FPGA. The kernel execution time of the AP also stays constant (48 seconds). Table 4.3 shows the maximum query numbers we can store on one AP board for different numbers of mismatches. We also present the possible speedups assuming the AP shares the same reporting architecture as the FPGA does.

The results prove the great advantage brought by the automata-based method on the FPGA, and potential speedups can be achieved by the AP. If a hardware accelerator is not available, HyperScan on the CPU also provides substantial speedups compared with the existing methods on the CPU. However, iNFAnt2 on the GPU only provides minimal speedups compared with the single-thread HyperScan on the CPU.

In summary, the automata-based method running on spatial architectures, such as the FPGA and the AP, work consistently faster than the state-of-the-art tools, especially when comparing larger numbers of queries or allowing larger numbers of mismatches. Promising speedups are achieved compared with CasOFFinder on the GPU (over 83× on the FPGA) and even better speedups are achieved compared with CasOT on the CPU. The more customized hardware such as the AP could provide additional speedups compared with the FPGA. High speedups are achieved by the spatial architectures because they can implement a large number of automata storing the queries directly in hardware and process these queries simultaneously. Furthermore, the method using single-thread HyperScan also achieves promising results (over 29.7× speedups compared to CasOT) and only works slightly slower than CasOFFinder. This shows that the automata approach confers a significant algorithmic advantage, especially for large numbers of queries and mismatches. The differences in speedups between HyperScan and the FPGA/AP show the benefits of hardware acceleration for automata processing, e.g., 88× faster when searching 200 queries with 6

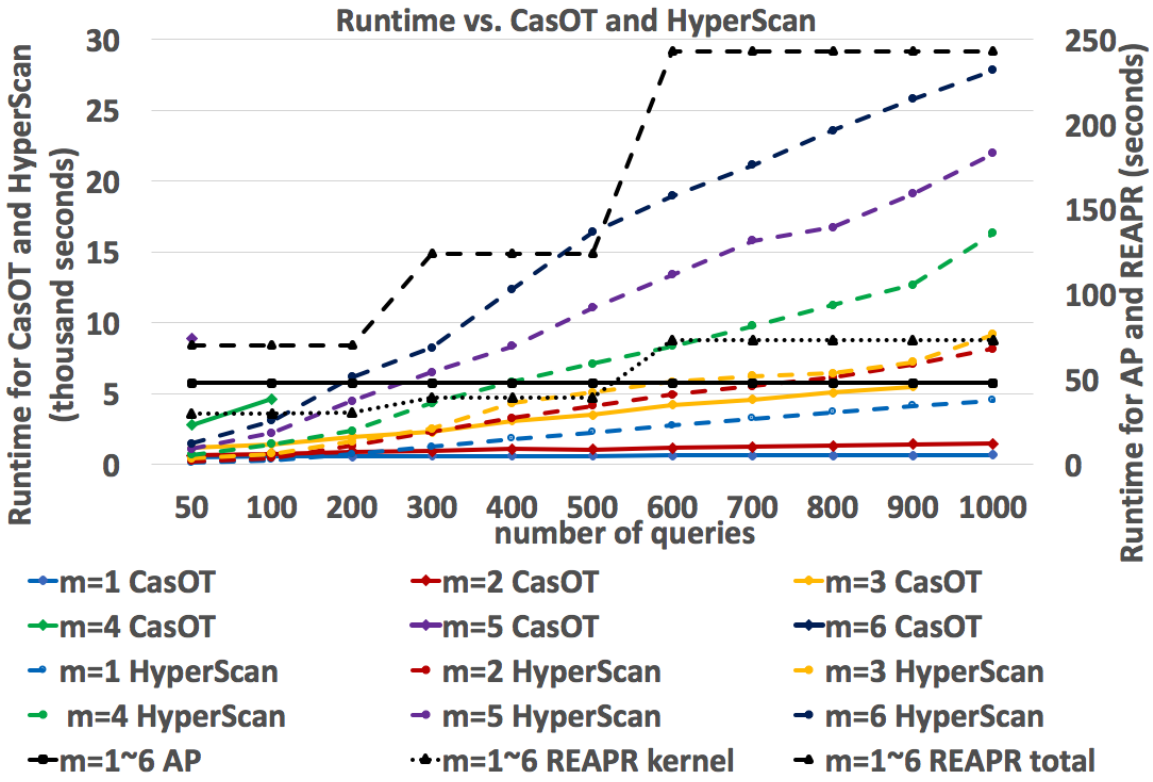

Figure 4.12: Runtimes vs. CasOT for different numbers of queries, where m is the number of mismatches. Dotted lines are runtimes of HyperScan and solid lines are runtimes of CasOT. The lines with the same color refer to the same number of mismatches. Black lines represent the results of the AP and REAPR. Many data points of CasOT are missing because it cannot return the results in 24 hours.

mismatches allowed in the seed region on the FPGA. However, the current automata-based method on the GPU only provides a minimal advantage over the CPU, because the automata processing approach does not map well to the GPU architecture.

Mismatches	**1**	**2**	**3**	**4**	**5**	**6**
Max number (K)	26.2	16.5	11.8	8.3	5.8	3.2
Speedup nq=50	6	7	13	31	101	492
Speedup nq=500	7	12	40	NA	NA	NA
Speedup nq=1,000	5	11	NA	NA	NA	NA

Table 4.3: Max number of queries stored on one AP board and possible speedups for different numbers of mismatches. nq is the number of queries."NA" refers to the cases when CasOT cannot finish within 24 hours.

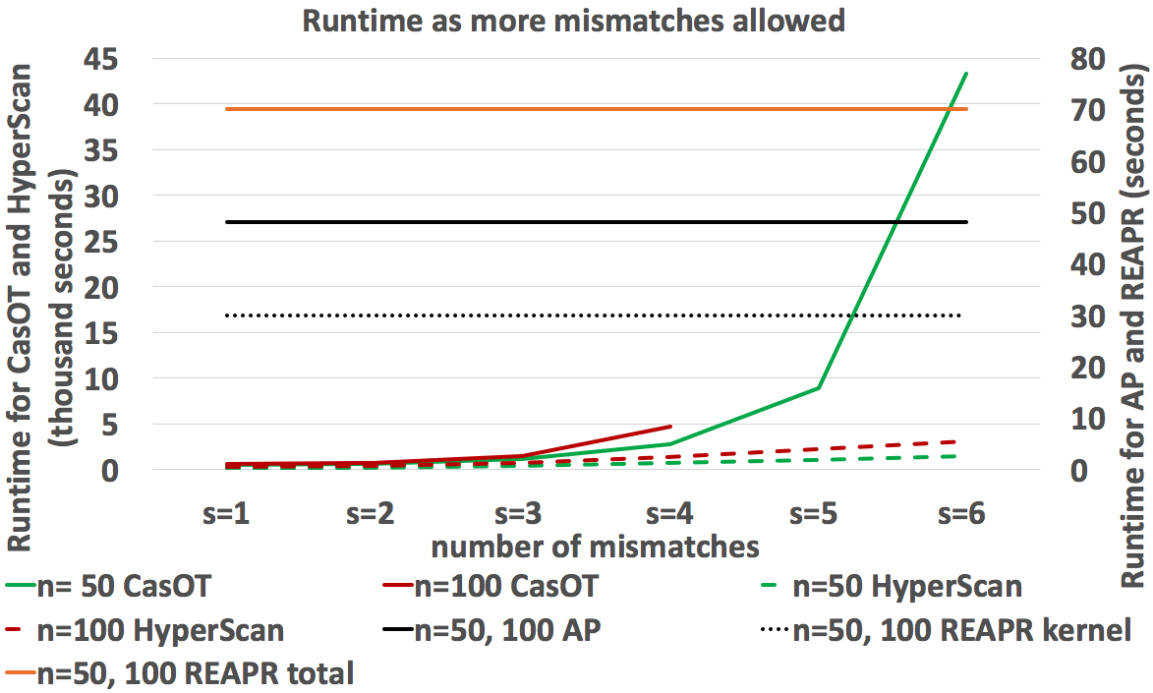

Figure 4.13: Runtimes vs. CasOT for different mismatches(m).

Table 4.4: Runtimes for large datasets when 3 mismatches are allowed.

Query Number(k)	**≤7.2**	**7.2-14.4**	**14.4-21.6**	**21.6-28.8**	**28.8-26.0**	**26.0-43.2**	**43.2-50.4**
Runtime (seconds)	48	96.045	144.09	192.135	240.18	144.355	168.27

4.7 Further Improvement on Spatial Architectures

4.7.1 FPGA

Support Large Datasets

The computation core of REAPR is the automata processing module where state-transition elements are map-ped to registers and lookup tables on the FPGA [25]. REAPR takes advantage of Xilinx SDAccel [108] to generate PCIe and AXI circuitry for the I/O interface to transfer data to and from the automata processing kernel. The entire design, including the automata processing module and the I/O circuitry, is laid out on the reconfigurable fabrics. Hence, the main limitations of REAPR are the FPGA capacity and the memory interface. Because of the maximum memory bitwidth (512 bits) currently supported by SDAccel, we implement the 1024-bit report

architecture using two memory ports, each of which is attached to a DDR bank on the FPGA KU3 board with two DDR banks. Furthermore, when we search for 1,000 queries and allow 5 mismatches, the design already utilizes around 90% of the hardware resource. Due to these two reasons, the current REAPR implementation in this chapter can support up to 1024 queries (i.e. 1024 reporting states in the NFA).

Larger FPGA boards provide more hardware resources, and thus relieve the bottleneck. There are also several other possible solutions. One is using a compressed report architecture to reduce the reports size so that they fit into maximum bitwidth. We profiled the results and found there are only a few reports in the same cycle, making the report vector fairly sparse and could be compressed by a lot. Assuming we compress the report to a quarter of its original size, we can process four times more queries. Another solution is using an on-chip buffer to temporarily store the reports. Instead of transferring the reports back to the CPU immediately, it will not report until the buffer is full. A double-buffering architecture would be useful to improve the throughput for this solution.

Higher Parallelism

Approaches can be applied to further improve the parallelization. The current automata processing kernel on REAPR is synthesized on a single compute unit (CU, the element in the FPGA device on which the kernel is executed). It would be helpful to exploit the parallelism among the independent automata and among the automata input streams on multiple CUs to increase throughput. Specifically, distinct automata can be divided into a number of sub-automata, and the input stream can be divided into chunks properly. We then synthesize multiple CUs on the FPGA device to process these sub-automata and chunks simultaneously. This helps to further reduce the total runtime, as long as there are enough resources on the FPGA device. For the multi-CU implementation, each CU should be assigned to a separate set of memory banks for optimum memory access. Multiple CUs sharing the same memory bank may lead to performance degradation because of memory access contention. This model can be implemented on a multi-CU, multi-FPGA architecture on a local workstation or even in cloud-based FPGA platforms such as the Amazon Web Services F1 [109] or Nimbix Cloud [110]. For instance, we implemented the multi-CU design on a larger board (Xilinx XILACCEL-RD-KU115 board equipped with a Kintex UltraScale XCKU115 FPGA and four DDR4 banks). We access the KU115 board through Nimbix Cloud. The XCKU115 FPGA board is 2× larger than the XCKU60 FPGA board used in our

experiments and enables synthesizing kernels work at 300MHz (250MHz for the KU3 board). Taking all these factors into account, we can achieve another 2.4× speedup compared to the results in the previous section.

4.7.2 AP

Support Large Datasets

For larger datasets, the number of STEs is the bottleneck of current AP board. The number of queries that can be processed increases linearly as the number of STEs increases. In this section, we discuss how to process large datasets on the existing AP board, and use the design in Section 4.5.3 as an example. Table 4.4 presents the kernel execution time when the Hamming distance is 3 and the query number is larger than $12,000$ (maximum queries on one AP board). The symbols stored in the STE can be replaced quickly if the automata graphs stay unchanged, and it takes 45 milliseconds to reconfigure the whole AP board. As shown in the workflow (Figure 4.7), we need to stream in the reference/complementary genome after the symbol replacement. The kernel execution time in the table includes the time of streaming the reference/complementary gen-ome and the reconfiguration time. However, compared with the streaming time, the reconfiguration time is almost negligible. The kernel execution time of the AP increases almost linearly as the query number increases, in steps corresponding to the number of symbol replacement passes. The time will not increase until another reconfiguration and another pass of the input are needed (when the query number is larger than $24,000$). The clock speed, and thus the rate at which the reference genome, is processed is the largest factor in AP performance. Additional performance improvements can be achieved with hardware supporting higher clock speed.

Potential Architectural Modifications

Boards with Fewer Chips: The AP D480 board has 32 chips on one board. The gRNA targeting sequence is usually short (20-30). Therefore, the automata designs proposed to recognize these sequences do not consume many STEs. As a result, the AP can process a huge number of queries simultaneously. However, when users only want to search for a few queries, the AP board is underutilized. It might be more cost-effective for some markets to have smaller boards with fewer chips for small problem sizes.

Multiple Streams to Different Chips: Instead of building smaller boards, allowing different input streams to different chips also helps to better utilize the existing board. For example, if we store different queries on different chips and stream different reference genomes to different chips, we can get the results for different genomes simultaneously. We can also apply a similar method as described in Section 4.7.2: store the same set of queries on different chips, divide the reference genome properly, and stream the divided genome chunks to different chips. This helps to further reduce the runtime and better utilize the hardware.

Flexible Symbol Set Size: The current AP stores 256 different symbols in one STE. However, when searching for gRNA off-target sites, we only use 9 of them (four DNA characters including lowercase and uppercase and '*'). This is a common case in many bioinformatics applications. It is a waste of area and power to operate an STE storage array supporting a large symbol set without actually using it. Therefore, if one can reduce the symbol set size and use the saved area for more STEs, the hardware can store more patterns, thus leading to higher parallelism. This is beneficial for problems such as Next Generation Sequencing.

4.8 Conclusions and Future Work

In this chapter, we proposed an automata-based approach for identifying potential gRNA off-targets, and presented several automata designs to solve this problem. To evaluate the suitability of the proposed method, we compared the automata-based approach on different platforms (CPU, GPU, FPGA and AP) with the state-of-the-art solutions (CasOT and CasOFFinder). Promising speedups are achieved (over 600× on the FPGA) and additional speedups could be achieved by the more customized hardware (AP). Based on the results of single-thread HyperScan (CPU) compared with CasOFFinder and CasOT, we conclude the automata-based method provides significant algorithmic benefits. However, the automata-based approach on the GPU provides a minimal advantage over CasOFFinder because it is not well mapped to the GPU architecture. Finally, we discussed how to further improve the performance (support larger datasets or high parallelism) on spatial architectures and proposed several potential architectural modifications for the future automata processing hardware. This work was published in [27], and most of the contents are derived from that paper.

Future work includes improving the proposed approach on the FPGA, extending to

other applications, implementing proposed architectural modifications and enhancements to automata processing architectures.

Chapter 5

Automata Processing Engine on Cloud-based FPGAs with New Features and Cross-platform Evaluation

REAPR is an FPGA automata processing engine previously developed in the Center for Automata Processing (CAP), UVA. It generates an RTL kernel for automata processing together with an AXI and PCIe based I/O circuitry. In previous chapters, we use REAPR to process different automata-related applications. We compare the performance results using REAPR against state-of-the-art methods, and experimental results show that using FPGAs for automata processing provide promising results. The high performance on FPGAs for automata processing motivates the work in this chapter to further explore how to use FPGAs for automata processing with even higher efficiency. Therefore, we port the framework to cloud platforms (Amazon AWS and Nimbix) with novel features. 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.

5.1 Introduction

Finite automata have shown their capability in a variety of pattern matching applications, especially when inexact matching is needed [9] [13] [14] [15] [22]. 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 HyperScan [19] 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, etc., which are difficult or impossible to formulate with regular expressions, can also benefit from automata-based approaches [2] [9] [13] [14] [27] [95]. In general, the automata-based approach consists of two steps: (i) designing finite automata, either in deterministic (DFA) form or non-deterministic (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 [19] [20] [21]. However, the performance is still bottlenecked by the random memory access behavior [111].

On the other hand, spatial architectures, such as FPGAs, Micron's Automata Processor (AP) [22], 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 [24] and [22] 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 [22].

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 Nimbix [110] and Amazon [112] 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, in this chapter, we aim to develop a high-throughput and user-friendly engine for enabling direct automata processing on FPGAs for 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 [25] [113]. 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 chapter 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 chapter, 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, Nimbix Cloud and Amazon EC2 F1, using the ANMLZoo benchmark suite. The ANMLZoo benchmark suite [45] 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.

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 [25]. We extend the original framework in this chapter 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, simplify the I/O integration, and further improve the performance by using a new reporting architecture and processing multiple symbols per cycle.

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.

5.2 REAPR Design

5.2.1 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 [24] [114]. However, our work focuses on the hardware synthesis of nondeterministic finite automata (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.

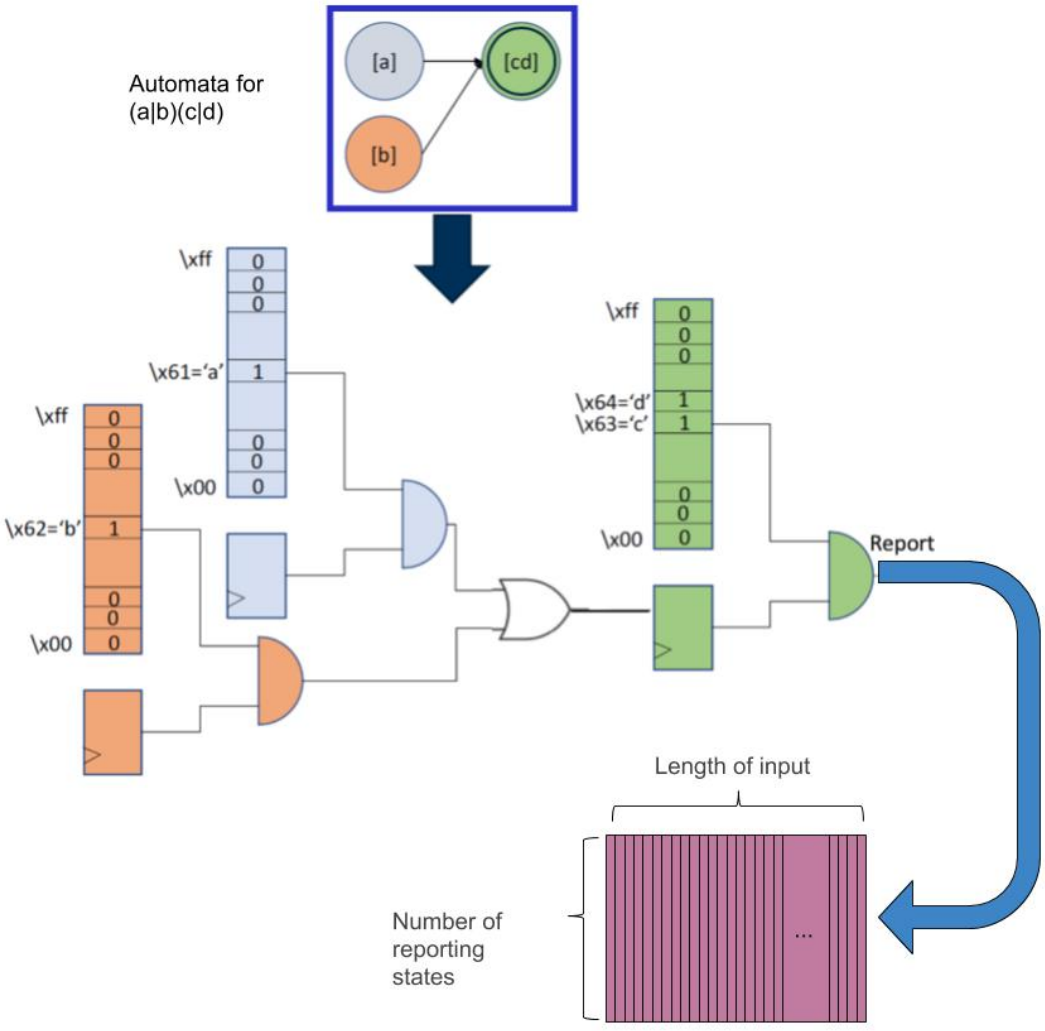

Figure 5.1: Mapping automata states to registers and look-up tables ("logic").

Merging the transition logic with the state transforms a traditional NFA into a homogeneous finite automaton [54], 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 Fig 5.1. 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 1x256 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 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 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 perfor-

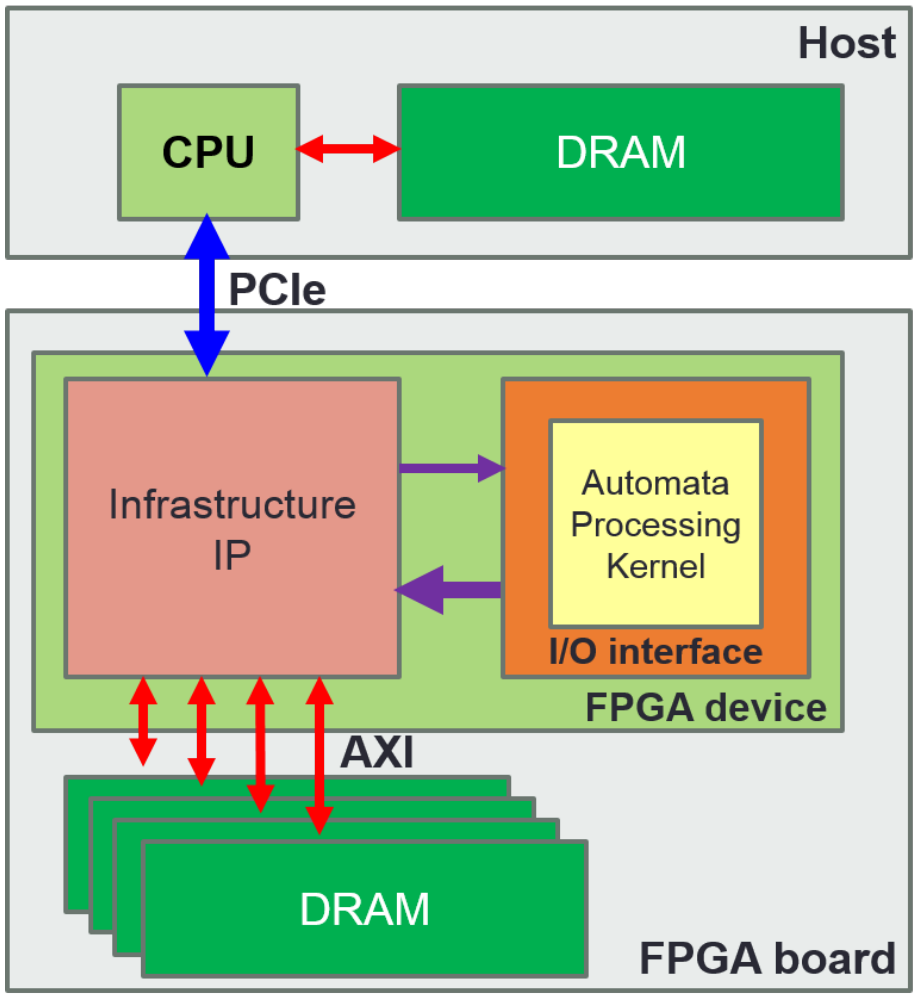

Figure 5.2: SDAccel-based approach of AXI and PCIe transactions for automata processing kernel.

mance 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 the next chapter.

The work described in this section is published in the original REAPR paper [25].

5.2.2 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 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 [108] 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 Fig. 5.2. 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. The inclusion of the I/O circuitry described in this section is published in the original REAPR paper [25].

To ease the integration stage, we design an I/O kernel performing a simple, templatized 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 Fig. 5.3. 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-integrated kernel is then compiled using Xilinx SDAccel to generate the hardware binary to be run on FPGA boards. This new integration work is collaborated with Vinh Dang and Ted Xie. I implement the automata kernel hookup, which replaces the dummy

```
#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;
      }
   }
}
```

Figure 5.3: I/O kernel with dummy computation.

kernel with the actual automata processing kernel.

5.2.3 Reporting Architecture

"Report" in this chapter refers to a match found in an automaton. A reporting architecture is included in REAPR as shown in Fig. 5.1. At each cycle (i.e. for each input symbol processed by REAPR), outputs from all reporting STEs (0s and 1s) 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)$\times$(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 [25] 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. For this work, I collaborate with Vinh Dang. 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 5.5. One potential solution is presented in [115] and more discussion will be presented in Sec. 5.5.5.

5.3 New Features

5.3.1 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 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 Fig. 5.4) that fully automates the whole framework. The new workflow is open-source and is available on Github [116]. 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 [117]; or (b) a MNRL file: a JSON-based, open-source description language to describe automata [118]. Compared with ANML, MNRL can be easily extended to support new features, such as up-down 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 (Step 2, 3 and 4). We then use Vivado_hls to generate the I/O template for the dummy kernel as shown in Fig 5.3. 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-

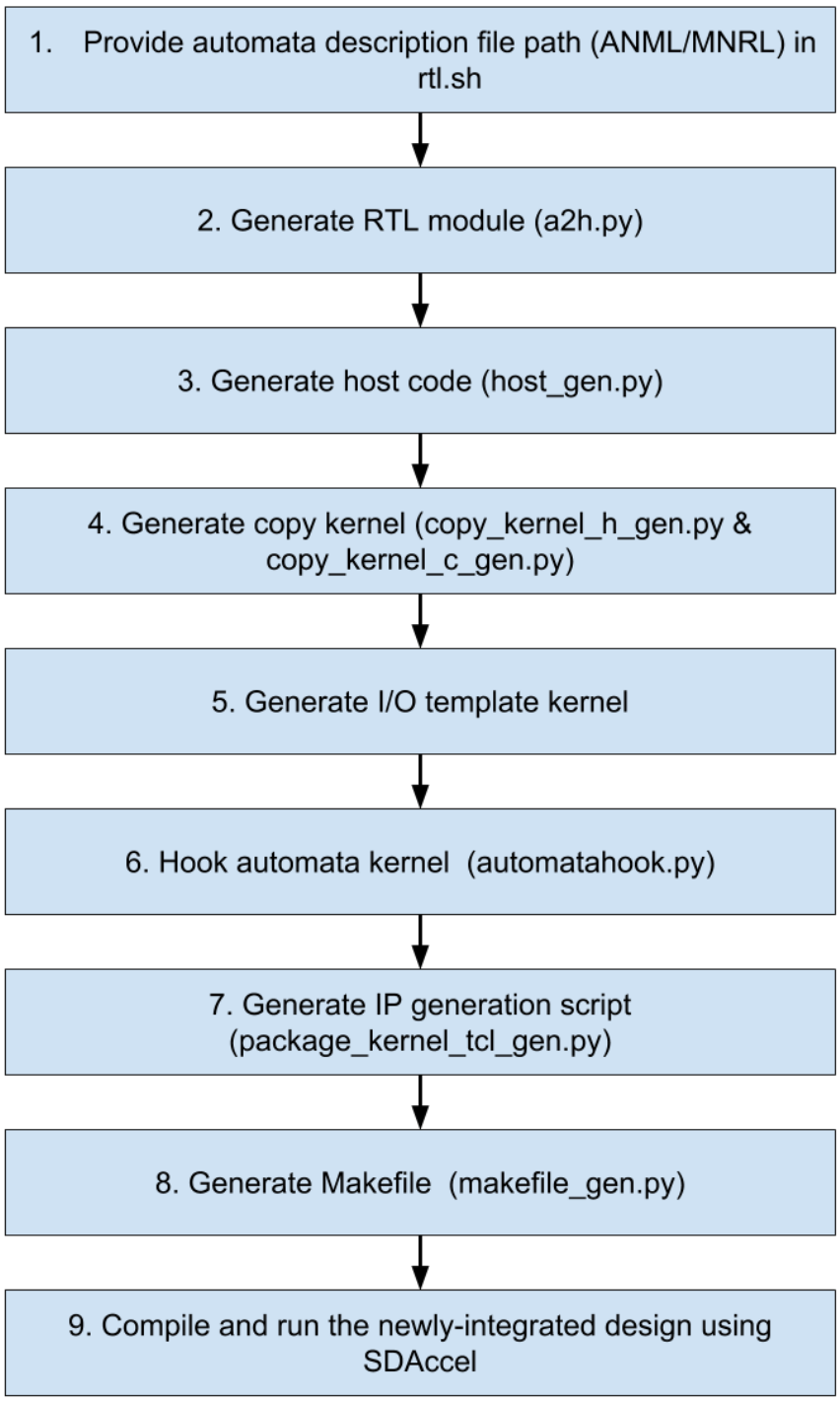

Figure 5.4: Automated workflow of REAPR.

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.

5.3.2 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. Vinh Dang implemented this feature and I took charge of evaluating the new feature against the original REAPR. The performance results will be presented in Section 5.5.

5.3.3 New reporting architecture

Preliminary results of REAPR show that the reporting architecture is the major performance bottleneck. This is because, in the original REAPR, we need to store all reporting states status to the reporting vector every cycle, and transfer the results back to the CPU even when there are no reports in that cycle. For example, in figure 5.5, if there are 1,024 reports in the application (we need to store the results using a vector with 1,024 bits every cycle with 1 indicating a report takes place) and the input size is 10MB, we need a 10 GB data block to store the results. We need to transfer the huge data block back to the host CPU and this is time-consuming. However, because of the nature automata processing, most automata-based applications do not usually see reports every cycle. Only some of the cycles have actual reports, shown as the thin black blocks in the figure. Therefore, I propose only transferring the necessary data block back instead of the whole block by adding a checker in the workflow. This helps to reduce the transfer time from the FPGA to the host CPU.

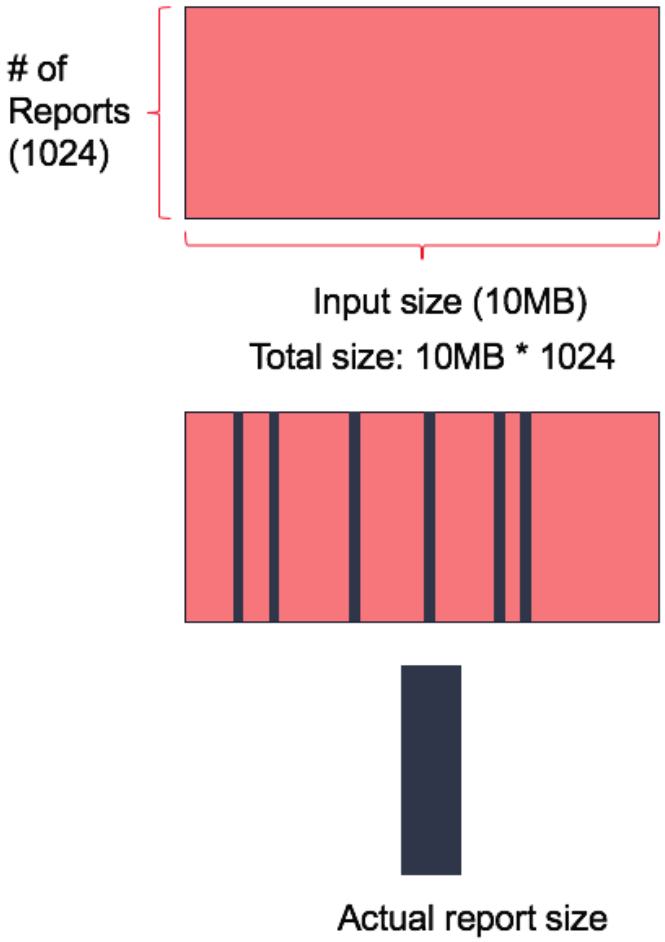

Figure 5.5: New reporting architecture.

5.3.4 Processing multiple symbols per cycle

Though REAPR shows promising results, the highest frequency supported is 250MHz, which limits the highest achievable throughput. Yang et al. propose a method for regular expression matching to increase throughput by processin multiple symbol per cycle. I adopt a similar idea, but I focus on processing homogeneous automata presentation. The proposed method uses the spatial stacking algorithm in order to achieve higher throughput. The algorithm makes multiple copies of the original REAPR design for STE, and use these copies to compose new "super" states. Each "super" states only contains one activation register. Then we connect all the copies in a way such that no copies need to wait for other copies' results and can directly activate successor states if it matches with the input symbol. Therefore, we do not need to stream the input multiple times or process the overlapping part between adjacent

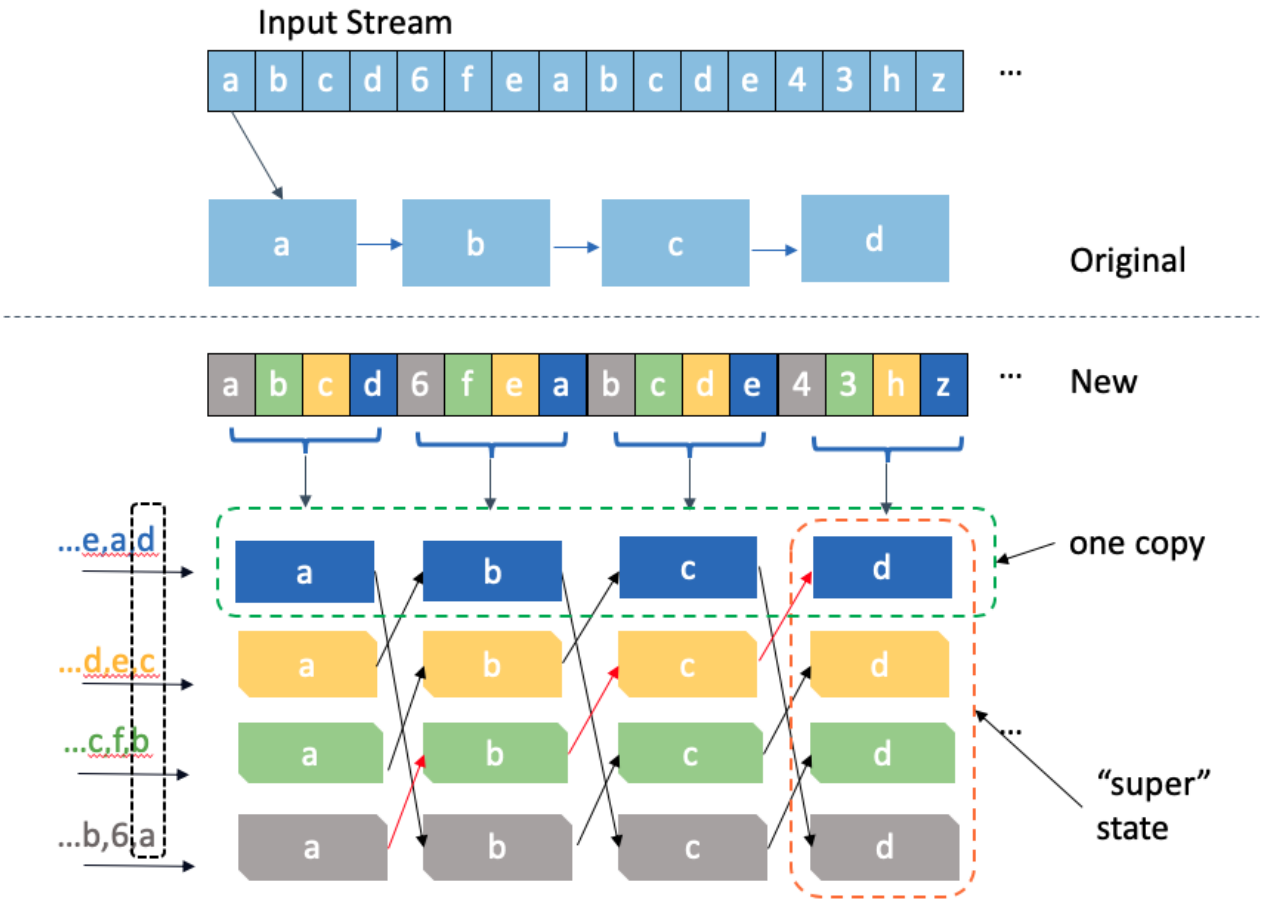

Figure 5.6: Processing multiple symbols per cycle.

windows. At each cycle, multiple symbols are streams to the corresponding character sets in the "super" states (the same color as shown in Fig. 5.6). For each cycle, four symbols are input to the corresponding copy. The input symbol is broadcast to all states in that copy. If we input "abcd" in the first cycle in Fig. 5.6, the first copy in the last column will be activate. This only requires one cycle because the copies are connected by wires and once one is copy is activated the successor copy will be immediately activated if the successor matches the input symbol.

The modification takes places in the translation phase from ANML to RTL codes. Four major steps are involved: 1). Prepare the new starting states; 2). Prepare the new reporting states; 3). Merge multiple copies to new "super" state; 4). Connect these copies.

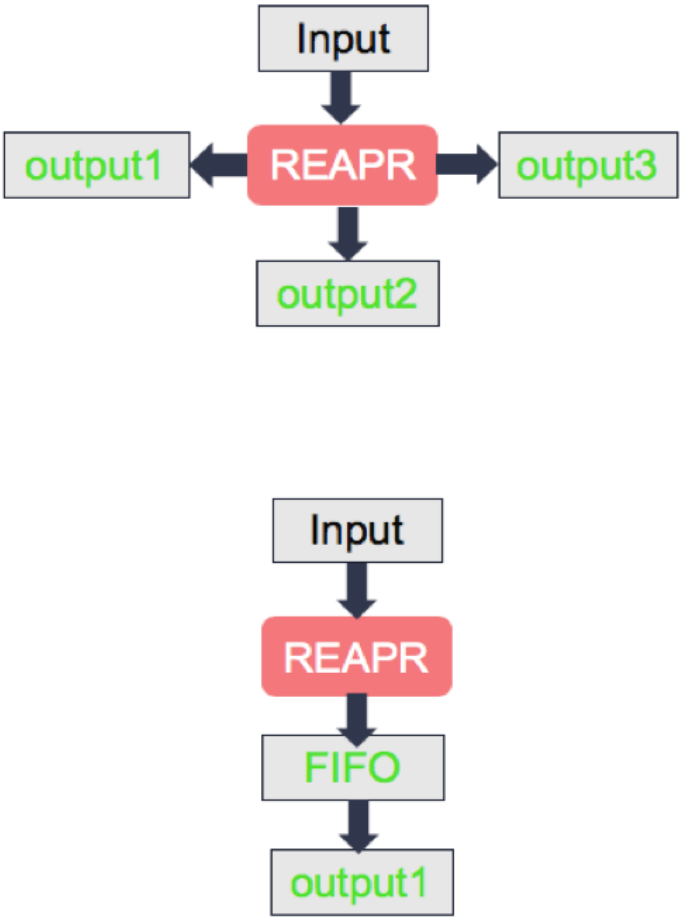

Figure 5.7: Simple I/O.

5.3.5 Simplified I/O integration

The original REAPR utilizes multiple DDR banks for transferring the results back to the host CPU to maximize the performance. For example, for a board with four DDR banks, users use one of them for input and all the remaining three for output. Though this helps to achieve better performance, it makes the integration more difficult because applications with different numbers of reports need different numbers of DDR banks to transfer results and may even use the DDR banks multiple times, leading to the messy integration of the automata kernel as mentioned in the above section. For example, if an application has 600 reports, the first 512 reports are transferred using one bank and the remaining 88 are transferred using another one. To solve this problem, I propose using just one DDR bank for the output and using a FIFO

between the kernel and the DDR bank as a buffer for the output. The new method actually serializes the original output transfer method and may lead to performance degradation. However, because of the automata processing nature we mentioned in the above section, for most of the cycles, we do not see reports and the FIFO can keep transferring reports during these idle cycles. But it needs to be pointed out that there could be cycles that need to wait for the FIFO to finish to process the next symbol.

5.4 REAPR on Cloud Platforms

5.4.1 Overview of Nimbix and Amazon Web Service

Nimbix

Nimbix is a cloud service provider powered by JARVICE platform [110]. 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.

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) [119]. 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 64 GB 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.

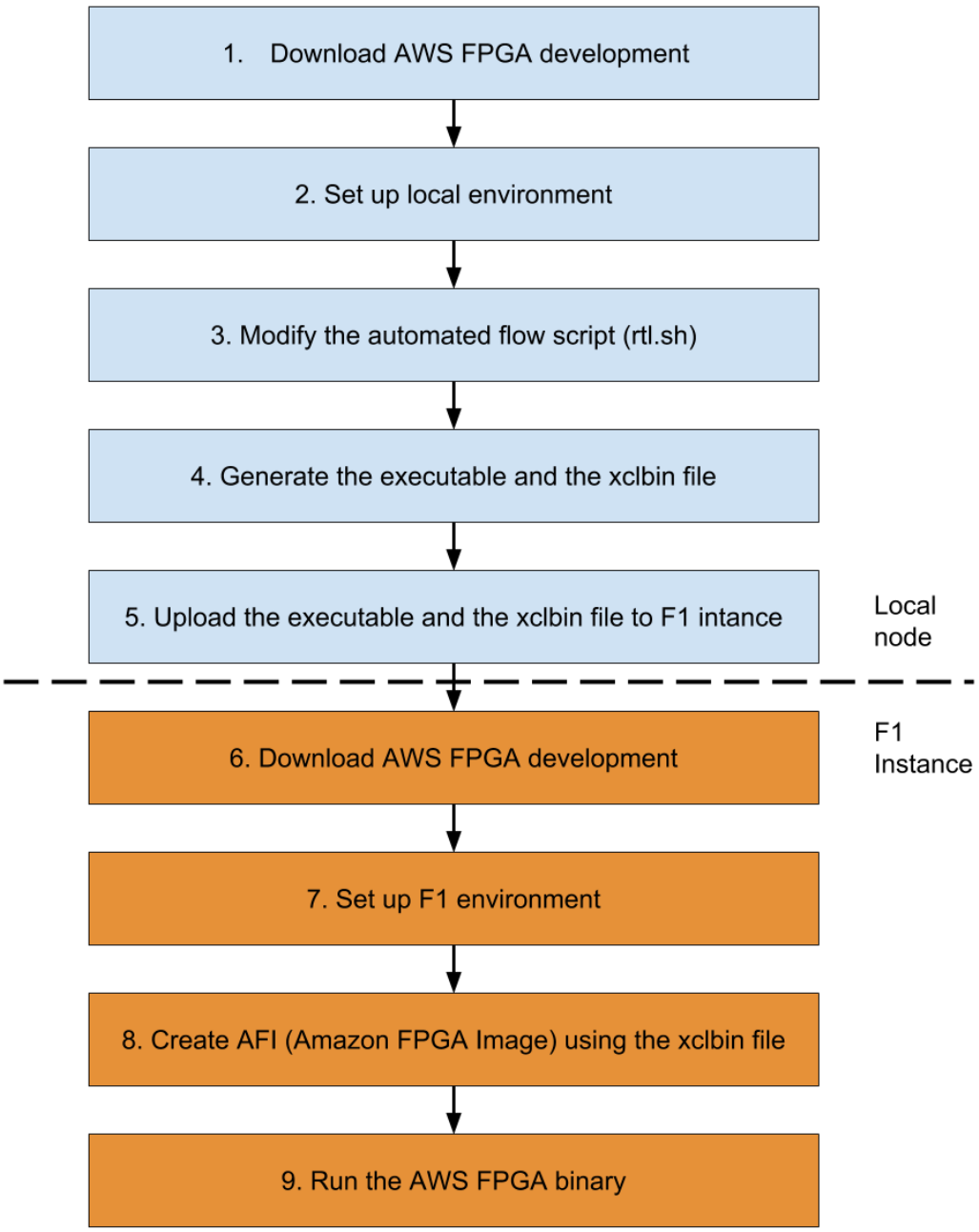

Figure 5.8: Workflow on F1.

5.4.2 Workflow on Cloud

The automated workflow of the FPGA automata processing engine on local nodes is described in Section 5.3.1. In this section, we implement the framework on cloud platforms, namely the Nimbix Cloud environment and Amazon EC2 F1. For develop-

ment, 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. the FPGA board model available on the cloud), one can use the automated workflow to compile an automata application for the hardware binary and the host executable.

Workflow on Nimbix

To use REAPR on Nimbix 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 Nimbix. 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 Nimbix. In general, just one extra submission step is needed for Nimbix compared with the local workflow.

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 Fig 5.8. A toolkit is available in [119] 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 use the correct SDAccel version supporting the FPGA board on AWS (step 2). The top half in Fig 5.8 shows the 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 [47].

5.5 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 VXCU9P 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 [19] (a highly optimized automata processing engine for the CPU) running on an Intel i7-5820K CPU, (ii) two GPU-based engines, namely DFAGE [20] (for DFAs) and iNFAnt2 [21] (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.

5.5.1 Benchmarks

We utilize the ANMLZoo automata benchmark suite developed by [45] 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 5.1. The detailed description of these benchmarks can be found in [45]. 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 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 Sec 6.6.1.

5.5.2 Utilization

In this section, we evaluate the CLB utilization on both AWS F1 and Nimbix. There are 147,780 and 82,920 CLBs available on AWS F1 and Nimbix respectively. The results are shown in Fig. 5.9, which presents the CLB utilization for both the kernel

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

Table 5.1: ANMLZoo details

execution and the whole process including the I/O overhead. For the same application, Nimbix 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 Nimbix (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 Nimbix. We suspect that Xilinx tools may use different optimization techniques for the two different boards, leading to better CLB usage on Nimbix.

On average, CLBs consumed for the whole process is 2.7× more than the kernel execution on Nimbix; while on AWS F1, CLBs consumed for the whole process is 5.3× 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 Fig. 5.10 and Fig. 5.11. As shown in these figures, on both cloud platforms, the CLB utilization increases roughly linearly as the number of STE increases and the number of routing nets increases. Therefore, the larger the automaton is and the more complex the structure

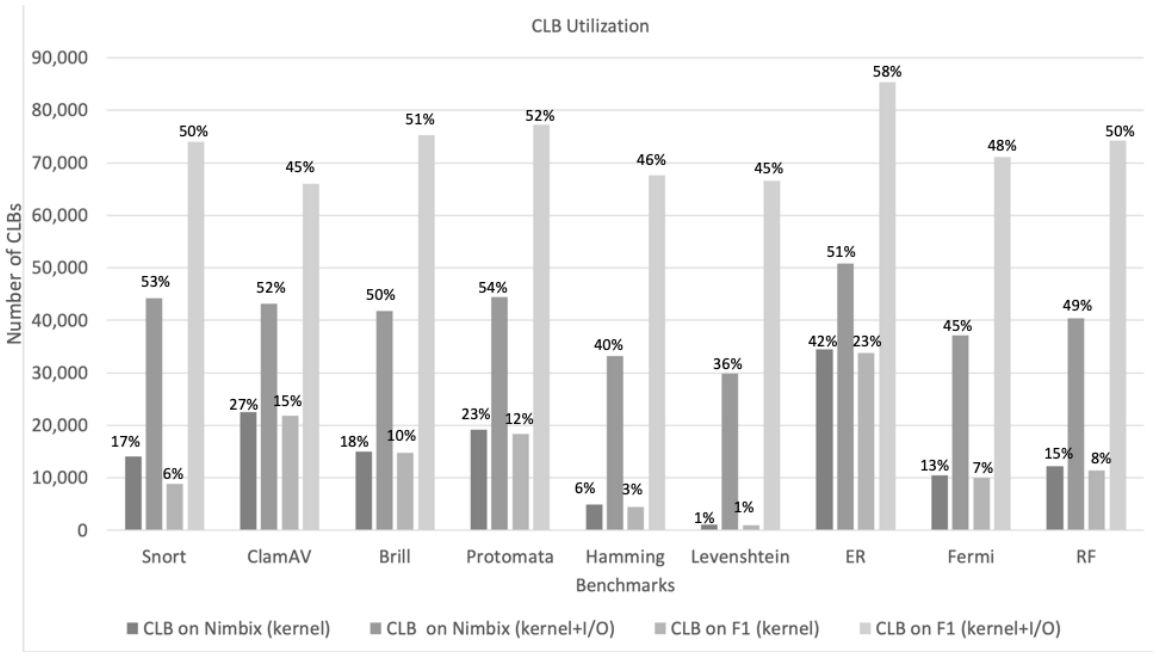

Figure 5.9: CLB utilization

of the graph is, the more CLBs will be consumed. When designing automata for an 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.

We also collect the usage of LUTs and BRAMs on FPGA. In this section, we only provide the results for AWS F1 instances. There are around 1.2 million LUTs on F1. On average, we consume around 29.5% of all LUTs for the applications in ANMLZoo that are runnable on F1. At most, we consume 38.9% of the LUTs for Entity Resolution. For BRAM, there are over 2100 BRAMs on F1. On average, we consume around 26.2% of all BRAMs for all the applications that are runnable on F1. For different applications, the consumption of BRAMs is almost the same. In general, LUTs and BRAMs are not the resource bottleneck on the F1 instance because of the large amount of these two resources available on the VU9P FPGA.

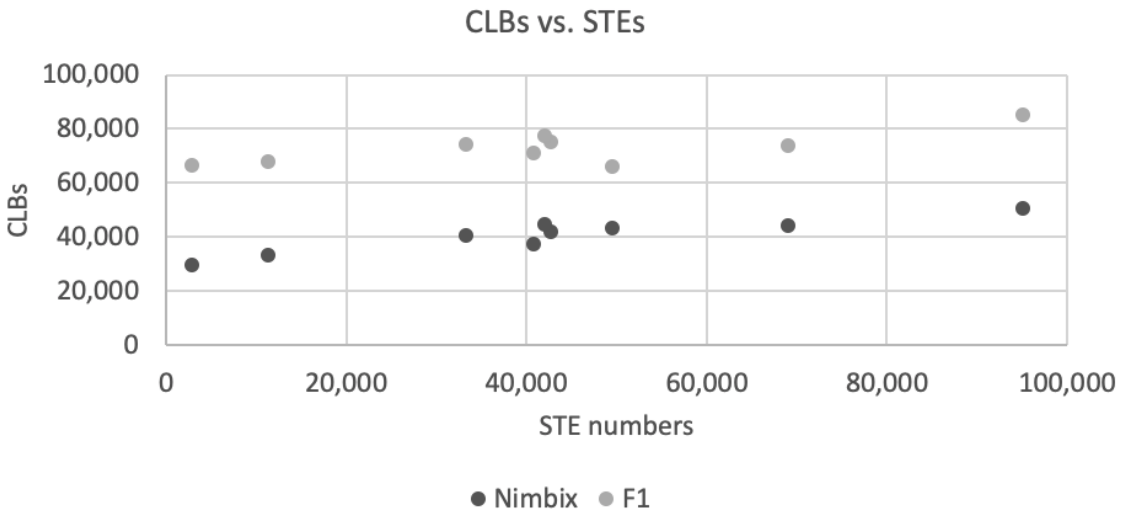

Figure 5.10: CLB vs. STE numbers.

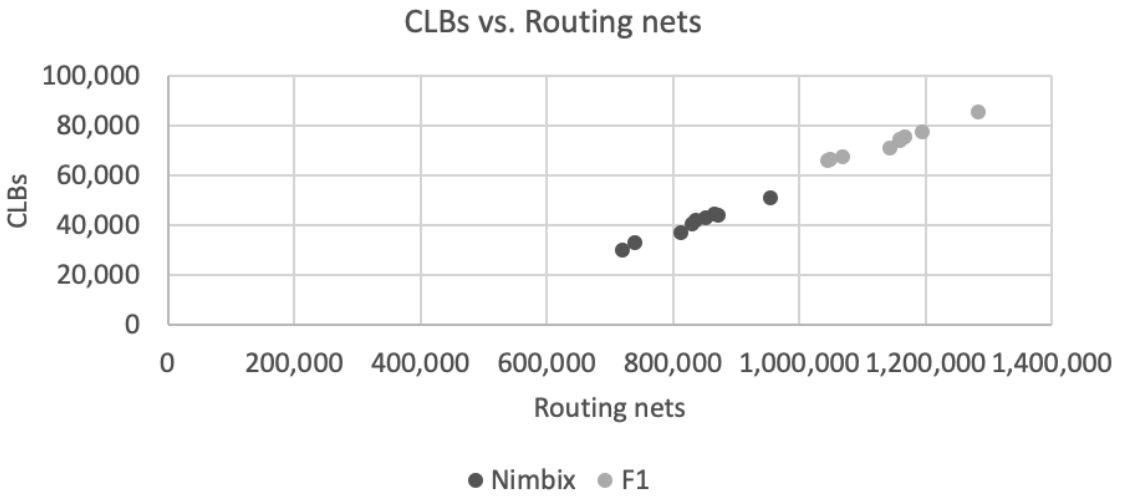

Figure 5.11: CLB vs. Routing net numbers.

5.5.3 Power

Similar to the above section, we present both on-chip power consumption of the kernel execution and of the whole process on Nimbix and AWS F1 respectively. The results are shown in Fig. 5.12. 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 Nimbix, 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 Nimbix, and from 40.67W to 43.34W on

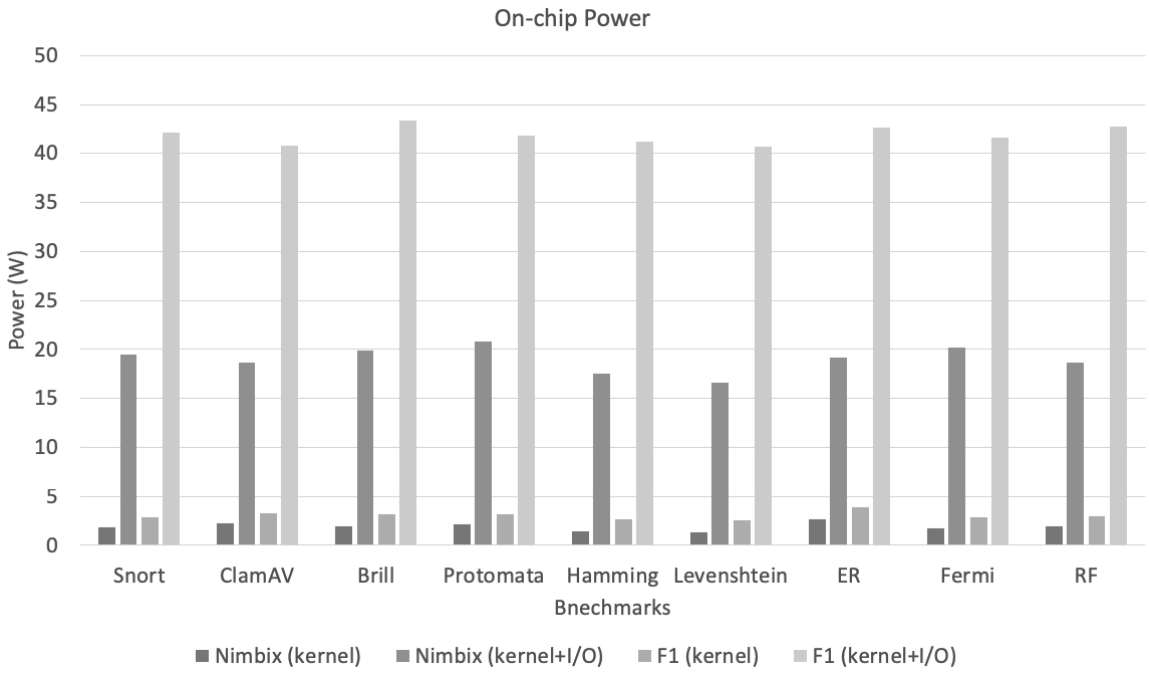

Figure 5.12: Power consumption for the kernel execution

AWS F1). The power consumption of the kernel only takes a small portion of the whole process (10% on Nimbix and 7% on F1 on average). This concludes that the I/O circuitry dominates the power consumption. The I/O circuitry is the component that needs to be further improved if users want to reduce the power consumption.

5.5.4 Performance

Platforms

We evaluate the performance of each automata processing engine over twelve benchmarks in ANMLZoo using their accompanied 10MB stimuli. Results of the kernel execution and the end-to-end execution are shown in Fig. 5.13 to 5.15 and Fig. 5.17 to 5.18, 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.

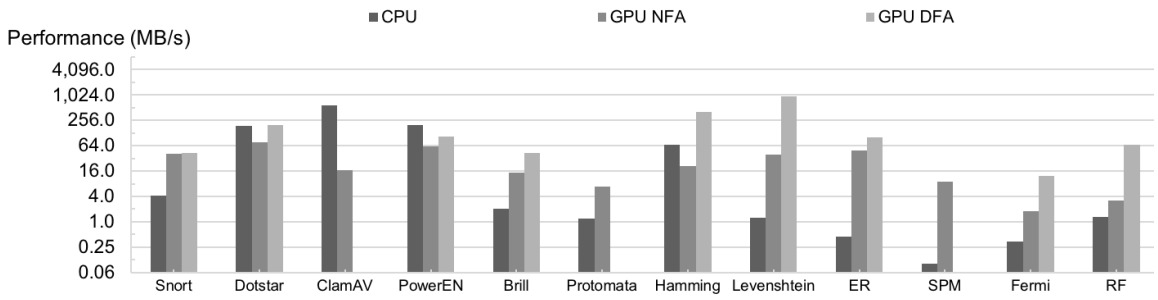

Figure 5.13: Performance for the kernel execution on von Neumann architectures.

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 [45] and [118]. 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 [115] are presented, as denoted by the terms "ideal" and "sim" in Fig. 5.15 and Fig. 5.18, 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. [13] [14] [16]. 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 [120] 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 [115]. 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.

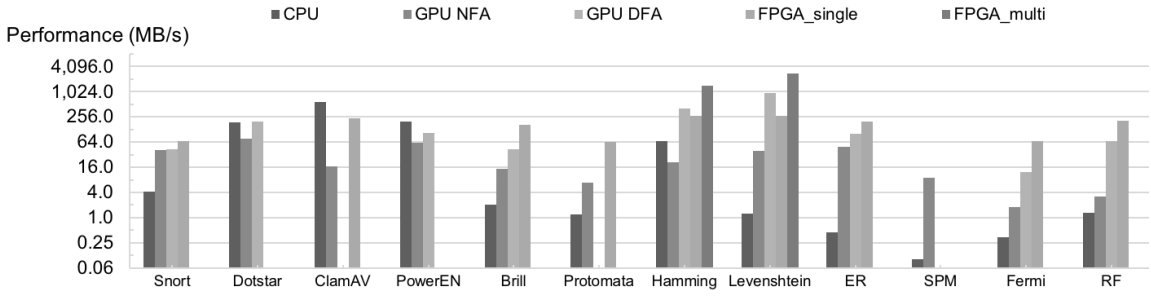

Figure 5.14: Performance for the kernel execution on FPGAs and von Neumann architectures.

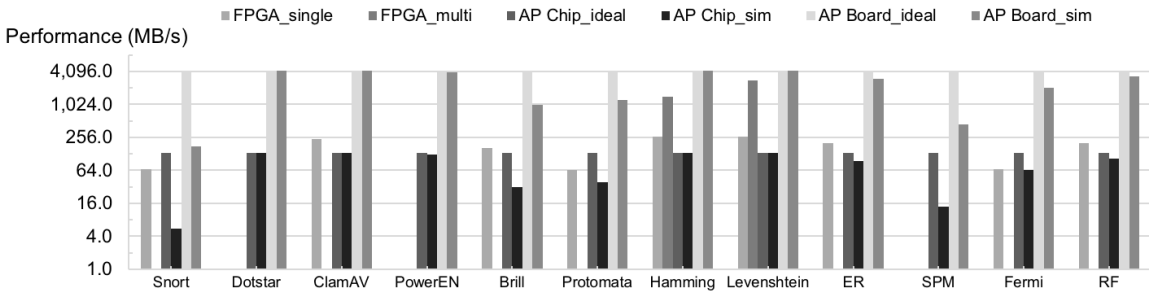

Figure 5.15: Performance for the kernel execution on spatial architectures.

Cross-Architecture Evaluation

As shown in Fig. 5.13, 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 (Fig. 5.16). 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, 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, Fig. 5.13 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 Sec. 6.6.1 (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 Fig. 5.14, 5.15, 5.17 and 5.18. A comparison between the performances on Nimbix and F1 is presented in Fig. 5.19. 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 lookups. 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. Fig. 5.14 shows that the REAPR throughput varies from 65MB/s (Snort, Protomata, Fermi) to 266MB/s (Hamming, Levenshtein). 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 Fig. 5.14) 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

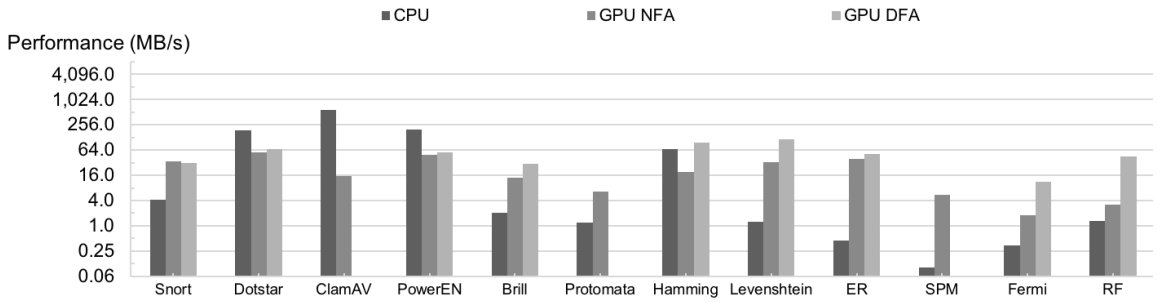

Figure 5.16: Performance for the total execution on von Neumann architectures.

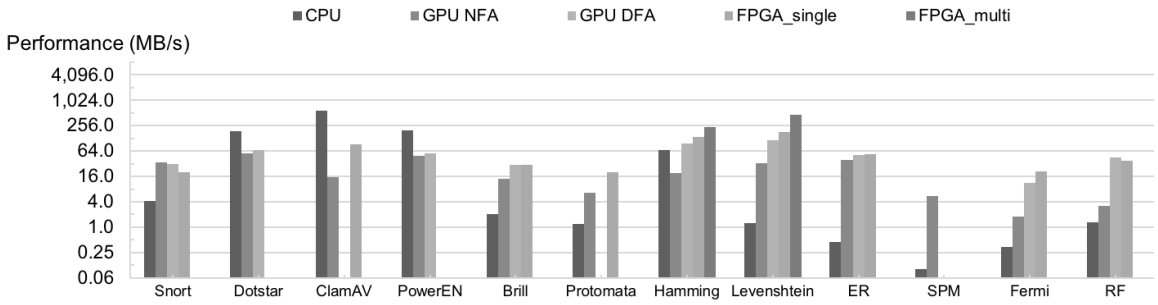

Figure 5.17: Performance for the total execution on FPGAs and von Neumann architectures.

leads to 5.4x (1.4GB/s) and 10.6x (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 22x 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 32x 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 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.

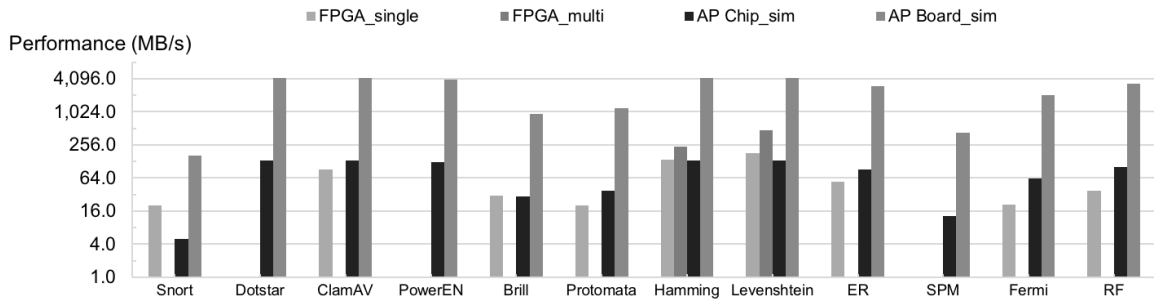

Figure 5.18: Performance for the total execution on spatial architectures.

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 transfers the ones with actual reports.

Fig. 5.19 shows the REAPR performance evaluated on Nimbix Cloud and on Amazon EC2 F1. Nimbix Cloud outperforms F1 in most benchmarks, due to its higher clock frequency (300MHz on Nimbix 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 (Nimbix) because of the limited hardware resource compared to the XCVU9P FPGA (F1), leading to a lower frequency.

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

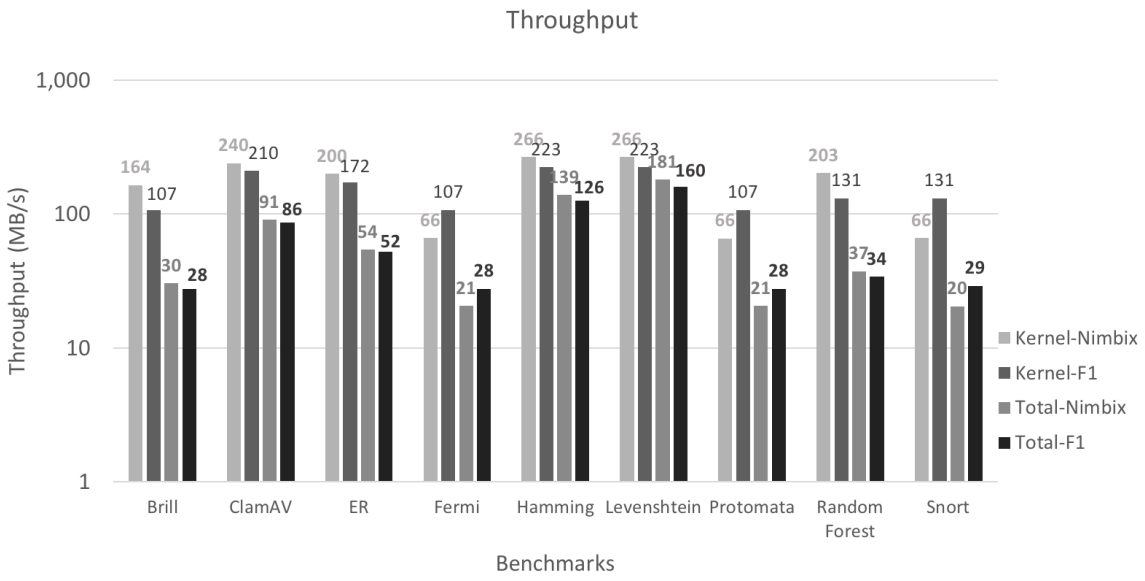

Figure 5.19: Performance on Nimbix and AWS EC2 F1.

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.

5.5.5 Discussion

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. Though 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. We also find that compiling the I/O for the whole application consumes the most time for the overall compilation time, which motivates the work in next chapter that reuse the I/O circuitry in order to reduce the high compilation speed. The details will be presented in Chapter 6.5.

5.6 Conclusions and Future Work

In this chapter, 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. This work was published in [**xie2018reapr**]and [28], and most of the contents are derived from these two papers.

Chapter 6

Reducing High Compilation Overhead for Automata Processing Engine on FPGAs

Long compilation time for large design on FPGAs has been a significant concern since FPGAs were first created. This problem becomes more severe as FPGA densities increase and the scale of FPGA design increases [121]. We encounter similar problems when we use FPGAs for automata processing. For example, the compilation time of ER could take over 17 hours, which is much longer than processing the automata kernel (finishes in seconds). The problem is even worse if patterns update frequently or users need to compile the patterns multiple times because the dataset is too large. For example, when working with large-scale applications, we need to partition the pattern set into small groups and process each group separately. Users need to compile the patterns separately for each group. Therefore, in this chapter, we aim to reduce the high compilation and reconfiguration overhead for automata processing on FPGAs. We propose three different methods to achieve this goal, including symbol-only reconfiguration, a new workflow using the Xilinx Object file, and modular synthesis with the reuse of the I/O architecture.

6.1 Introduction

Using FPGAs has shown advantages in many application domains, but the compilation time, which includes full synthesis, placement, and routing, can take hours or even days, making the compilation phase the bottleneck of many different appli-

cations [121] [122]. When using FPGAs as platforms for automata processing, we encounter similar problems. For example, the compilation using original REAPR takes almost 18 hours for Entity Resolution. Though the compilation is one-time overhead for some applications, it is still expensive; and this can become a problem in applications of which the rule sets may change or update frequently, thus requiring frequent re-compilations.

Furthermore, for large-scale applications (large pattern sizes), a challenge in using spatial architectures is that the patterns may not fit on a single device. This requires a method to partition the patterns into groups and support multiple passes with fast reconfiguration for each group. In this case, we may be able to reuse the compilation results; but if we reload the full structure each time, the high reconfiguration overhead will harm the benefits brought by the fast automata kernel execution on FPGAs.

To reduce the high reconfiguration overhead, we propose three different methods in this chapter.

1). Symbol-only reconfiguration

To process large-scale applications efficiently, we propose a symbol-only reconfiguration method. The proposed method keeps the same automata *structure* for the application and only reconfigures the *symbols* stored in each state, instead of re-compiling automata graphs for new partitions. The proposed method can process much larger problem sizes when the automaton structure is regular, and dramatically reduce the total compilation overhead and loading time.

2). Workflow using the Xilinx Object file

In the original REAPR workflow, to hide implementation details from users, we integrate all kernels in the compilation step (last step in Fig. 5.4, which includes FPGA synthesis, logic optimization, logic placement and routing) to generate the executable and binary container (xclbin file). We then notice that using the Xilinx Object file can help to reduce the time in the compilation phase. The Xilinx Object file stores the information about the RTL kernels and can be utilized by SDAccel for later compilations, which is more efficient than directly integrating all the kernels. Therefore, we propose a new workflow using the Xilinx Object(.xo) file in Xilinx SDAccel to reduce the compilation overhead.

3). Modular synthesis with the reuse of the I/O architecture

In the original workflow, we build the I/O communication structure for every single application and every single dataset. However, the I/O structure only depends on the number of input, and we can reuse the structure for applications with the same

number of reports. Therefore, we modularize the original RTL kernel and isolate the I/O circuitry between the CPU and the FPGA, and reuse the I/O circuitry for different applications with similar numbers of reports. The compilation time for I/O takes a large portion of overall compilation time, and reuse I/O circuitry is 2.3× faster on average across different applications from ANMLZoo.

The above methods help to reduce the high compilation overhead for the automata processing engine on FPGAs. In the following sections, we will present the details of each method and the experimental results. In the last section, we will discuss a potential future work using an overlay for automata processing on FPGAs.

6.2 Related Work

The FPGA compilation process is a time-consuming task. It includes logic synthesis, logic optimization, and logic placement and routing. The long compilation overhead harms the benefit brought by the fast kernel execution. Many applications using FPGAs encounter this problem [121] [122].

Though the compilation overhead is a big concern when using FPGAs, only a few previous works try to reduce the compilation overhead. Ajay Jagtiani et al. propose using parallel compilation in EAD tools [121]. The proposed method aims to combine the top-down approach (requires users to finish the work serially and must have the complete design before compilation, and it provides better results) and the bottom-up approach (allows users to provides smaller portions of the whole design with incremental compilation). They partition the original design into smaller blocks with appropriate constraints file and process each partition in parallel. Lavin et al. accelerate FPGA compilation by using hard macros [123]. Hard macros consist of previously synthesized, placed and routed circuits. They can re-use these hard macros for remaining tasks without recompilation. Lysecky et al. propose a dynamic routing method aiming to reduce the logic routing of the compilation process [124]. They develop a JIT (Just-In-Time) compiler for FPGAs to develop a standard binary that may port across FPGA architectures, such that users do not need to recompile binaries for every new FPGA architecture. We hope our proposed methods in this chapter will provide helpful insights for future research.

Some prior works propose similar ideas to the symbol-only reconfiguration mechanism. Teubner et al. propose a method for XML projection that supports fast reconfiguration [125]. They focus on how to design automata for that specific ap-

plication and map that design to FPGAs. Similarly, [53] focuses on regular expressions by representing regex using new representations of finite automata (ODFA and OD^{2}FA). These works usually target one specific application while our work provides a general automata processing framework for various applications involving inexact pattern matching. Micron's Automata Processor provides a similar method, named symbol replacement [22]. The AP is a memory-derived architecture and it allows users to quickly replace the symbols stored in the memory column with new symbols by writing new symbols to memory cells. The purpose of the symbol replacement feature is to reduce time spent on compiling automata structures on the AP.

6.3 Symbol-only Reconfiguration

In this section, we will first present how the symbol-only reconfiguration feature works and later we use Entity Resolution as an example to show how to use the proposed feature.

Although we have extended original REAPR to support a larger number of patterns in Chapter 5, it still may not be large enough for 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 [11] [126], there could be millions of long (over 100 base pairs) sequences to be matched. Many partitions of such huge pattern sets 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 when processing a new partition, as shown in Fig 6.1. To solve this problem, we propose a fast, symbol-only reconfiguration mechanism for REAPR as shown in Fig. 6.2. Each column in 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.

6.3.1 General Workflow

The symbol-only reconfiguration mechanism can utilize the current REAPR framework. However, the proposed method needs first to design a general automaton

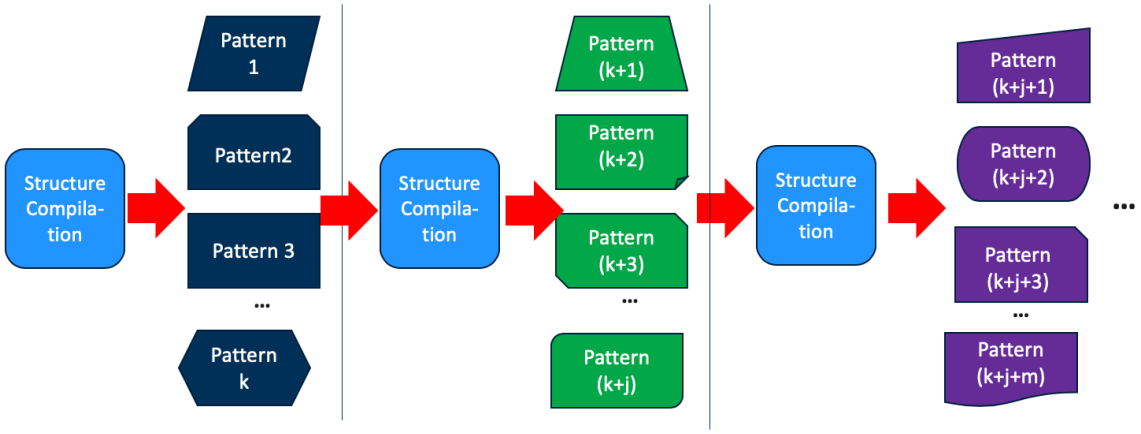

Figure 6.1: Original workflow for large-scale applications.

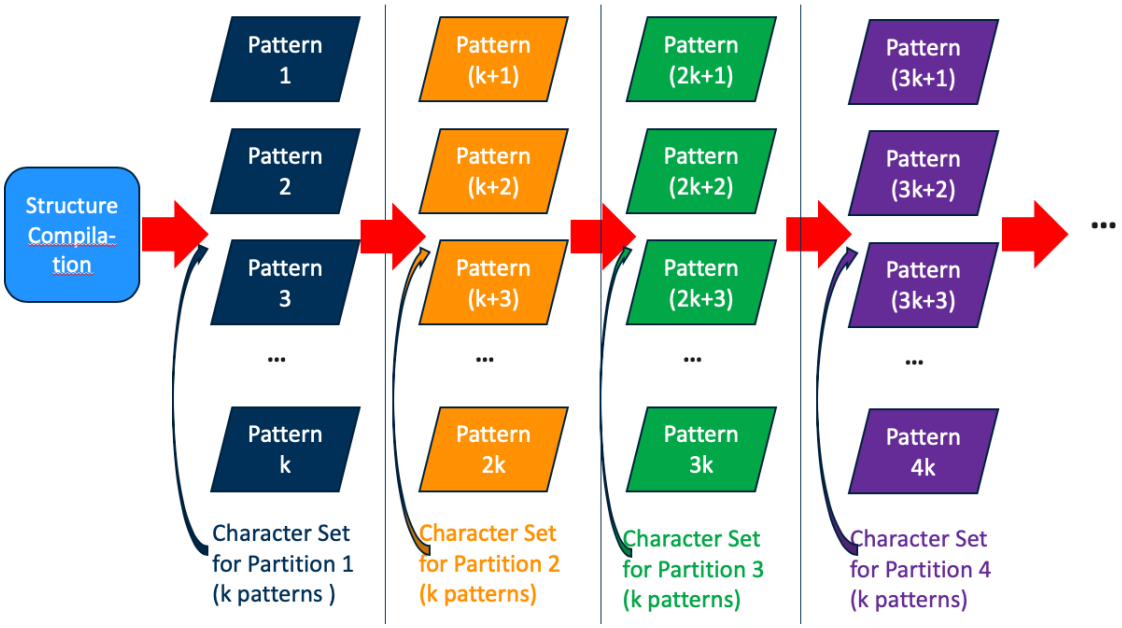

Figure 6.2: Symbol-only reconfiguration workflow.

structure (parallelogram in Fig. 6.2) that can be shared by all patterns in the specific application. This is helpful when the input is scanned for many patterns that all have similar length and structure, e.g., strings of length l with edit distance d. One example of how to design such structures will be discussed in the next subsection. The general automata structure varies from one application to another. After designing the structure, for each partition, we write the corresponding character set to the patterns in the partition. Each line in the character set file stands for one STE, with 256 bits representing 256 different symbols. When a new partition comes, users

only need to reconfigure the symbols (write the new character set for the partition) stored in the states, instead of compiling new structures for automata graphs in the new partition. This allows users to re-write new symbols without reloading the entire structure. Therefore, compared with the original REAPR flow, in addition to the automata description file and the input file, users also need to provide character set files. We provide a script that can generate character sets from an automata description file (ANML/MNRL), so users do not need to generate them manually. Writing new character sets is much faster than placing and routing new automata graphs on FPGAs, leading to the huge reduction of the compilation time. When implementing the symbol-only reconfiguration feature, in order to make the writing of new symbols fast, we choose to use the BRAM-based design of the original REAPR. Using BRAM to store symbol sets allows us to faster over-write previous stored symbols compared against the LUT-based design. More comparison between LUT-based design and BRAM-based design is presented in Chapter 5.2.

However, 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 automata structure that can be shared by all patterns. For example, Entity Resolution, Fermi, Hamming, Levenshtein, Random Forest from ANMLZoo [45] and CRISPR off-target identification [27] using automata processing can adopt this method. Furthermore, some of these applications are actually of broad utility, such as Hamming and Levenshtein. They can be used as building blocks of other automata-related applications.

6.3.2 Case Study: Entity Resolution

In this section, we present one example showing how to design the general automata structure. When using automata processing for Entity Resolution, we design a general automaton structure for recognizing the same person's names with variances in the SNAC (Social Network and Archival Context) database in [15] [93]. We first extract common name formats and then come out a design for these formats instead of individual names. The design is shown in Fig. 6.3. The bottom half is the automata structure (with Hamming distance no more than one) for the macros on the top half. The top 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. Prior work [125] and [127] propose

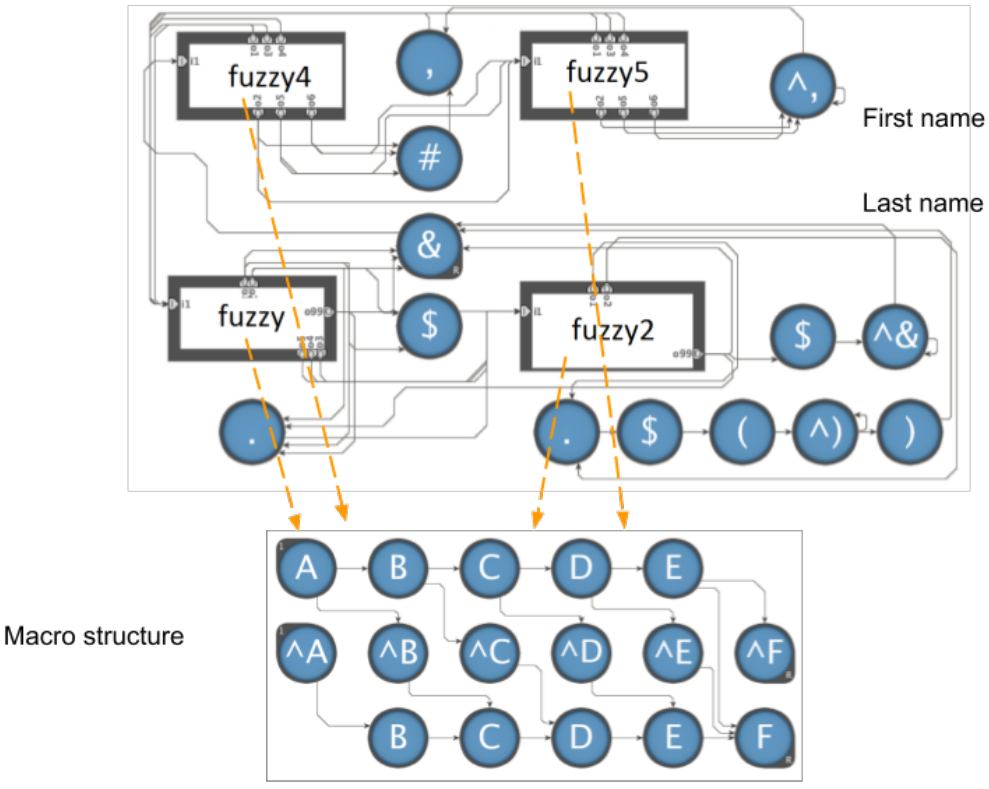

Figure 6.3: Example of general automata structure for SNAC ER problem.

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 paper 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. This is a one-time overhead compared to Fig. 6.1. 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 (0s or 1s) 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 thousand 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 characters sets multiple times.

6.4 New workflow using the Xilinx Object File

The Xilinx OpenCL Compiler (XOCC) [128] is a separate command-line utility for compiling kernel functions and linking these kernel functions with SDAccel environment supported platforms. In the previous automated workflow introduced in Chapter 5, we wait until the last step to compile the whole design to generate the executable and the binary container (xclbin) file. In this step, we provide the platform information and all kernel files to XOCC. XOCC will then link all kernels into the platform to create the binary container.

When working with the original workflow, we find that we can further split the last step into two steps, and this helps reduce the overall compilation overhead. The new workflow is shown in Fig. 6.4. The first seven steps are similar to the steps in the original automated workflow, and we can reuse some scripts we develop. However, when generating the executable and the binary container, we use the Xilinx Object file instead of directly using RTL kernel files. We first package the RTL IP and the kernel XML file together to generate a Xilinx Object (.xo) file. This can be achieved by using the package_xo command [129]. All kernel information is stored in the Xilinx Object file. Then we use XOCC and the previous generated Xilinx Object file to create the binary container. One or more Xilinx Object files can be used in this step [128]. Though this step still involves logic synthesis, logic optimization, logic placement and routing, it is faster than generating directly using RTL kernel files. Detailed results will be presented in Chapter 6.6.2.

6.5 Modular Synthesis and Reuse of I/O Templates

In the original workflow, we build the I/O communication structure for every single application and each different dataset. We use the high-level synthesis-centric approach by designing the I/O interface using Vivado HLS. We then use Xilinx SDAccel to generate the AXI interconnect and the PCIe circuitry for the automata kernel. The specific I/O circuitry depends on the number of reports in a particular dataset (usually equals to the number of patterns/rules). Different applications can share the same I/O circuitry as long as they have the same number of reports. When generating

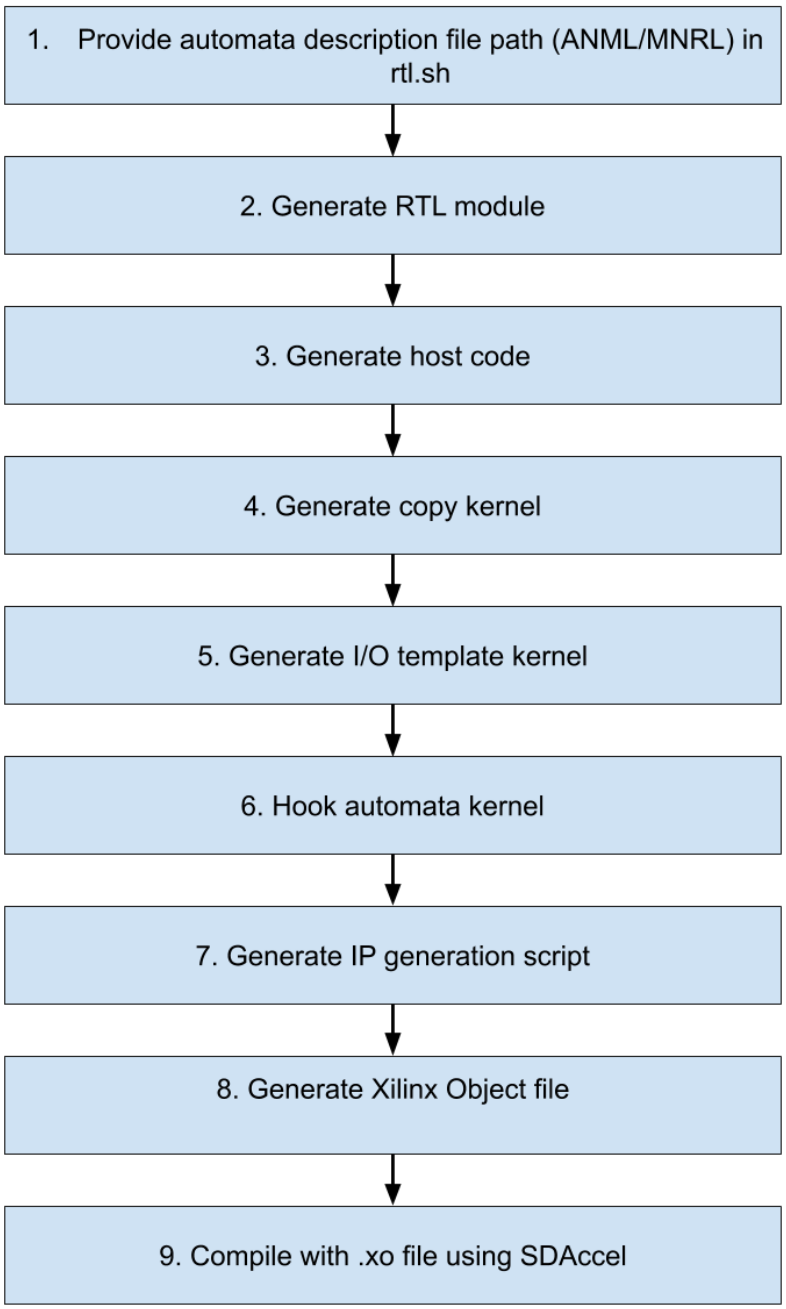

Figure 6.4: A new workflow using Xilinx Object files.

the I/O circuitry in original REAPR, we need to know the exact number of reports so that we can create correct interfaces. For example, when using the automated

REAPR workflow, we need to provide the number of reports in the entry file(rtl.sh).

There are four DRAM banks on the AWS F1 FPGA board, and the bidwidth of each bank is 512. As mentioned above, the I/O circuitry only depends on the number of reports. Therefore, we can round up the actual number of reports to 512, 1024, 1536, 2048, etc. For example, even if an application has only 439 reports, we still use the I/O circuitry for 512. In such a way, even if the applications have different numbers of reports, we can use the same I/O circuitry template that has already been pre-compiled. In the new design, we separate the I/O circuitry from the whole design and pre-compile the I/O circuitry for 512, 1024, 1536, etc. We then choose the corresponding I/O circuitry for different applications and datasets. By reusing the I/O circuitry, for a new application or a new dataset, we only need to synthesize the automata processing kernel instead of the whole design, which is much faster.

The detailed results will be presented in Section 6.6.3. However, though SDAccel uses Vivado in the back-end, it does not support using pre-compiled IP, and it has to redo the synthesis, the optimization, the placement and routing. The results we present in Section 6.6.3 are simulated by using Vivado directly, but we expect to achieve similar results when SDAccel supports similar functions.

6.6 Performance Evaluation

In this section, we evaluate the three proposed methods against the original REAPR for the compilation phase. We use applications from the ANMLZoo benchmark to show the feasibility of these proposed methods and how much potential benefits can be achieved. The applications include Entity Resolution, Random Forest, Brill, ClamAV, Hamming Distance, and Levenshtein Distance. We use FPGAs (Xilinx Virtex UltraScale Plus XCVU9P FPGA devices) from the AWS F1 service. Each FPGA is equipped with four DDR4 banks, and the max frequency is 250MHz. In this chapter, we use the instance with one FPGA (f1.2xlarge). Using multiple FPGAs for larger automata can be interesting future work.

6.6.1 Symbol-only Reconfiguration

In this section, we evaluate the symbol-only reconfiguration method against the original REAPR without this feature. As mentioned in Sec. 5.2.1, we use the BRAM-based design for symbol-only reconfiguration to utilize the faster writing speed of BRAM.

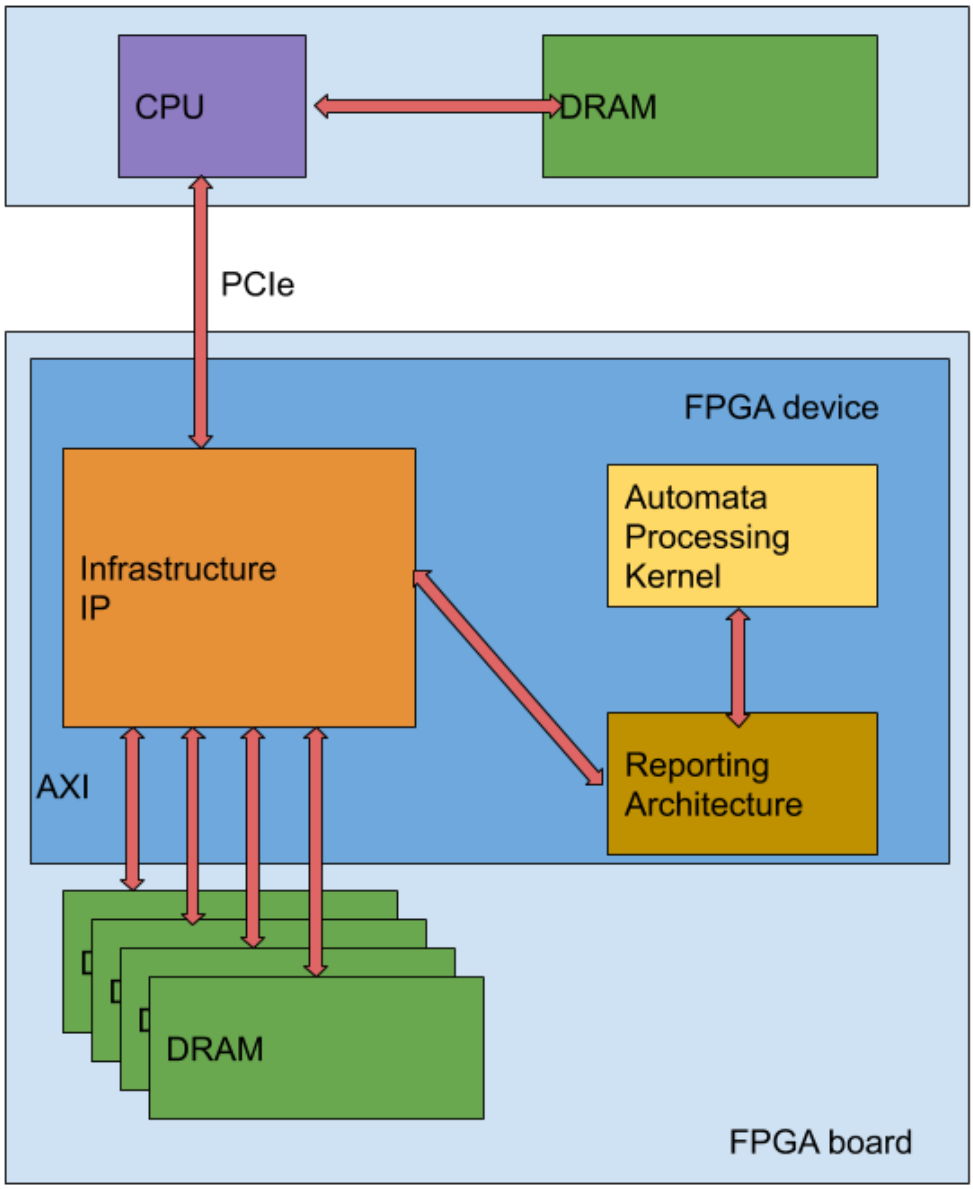

Figure 6.5: Modular synthesis.

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 Fig. 6.6.

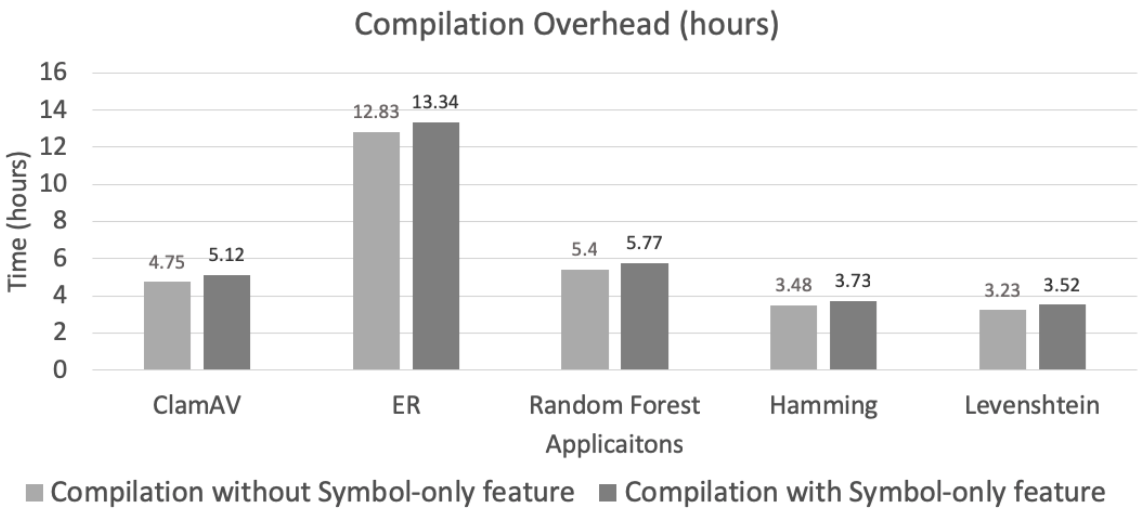

Figure 6.6: Compilation overhead.

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 symbol-only reconfiguration feature. On average, it consumes 6.5% longer time for compilation. Therefore, 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 6.1. 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 one second. This is a huge reduction from hours of re-compilation.

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

Table 6.1: Character sets writing overhead

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 6.2. 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 CRISPR/Cas9 system [130] [131]. 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% percent 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 much 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 compilations time 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.

Apps	**states #**	**input**	**GPU(total)**	**GPU(loading)**	**CPU**	**FPGA**	**Comp.**
CRISPR	346k	3.2GB	1,579	278.7	44,257	481	6h23m
ER	13,314k	3.1MB	NA	NA	NA	51.24	17h50m
SPM	201.5k	0.3MB	183	181.5	34	0.18	4h3m
RF	992k	4.6MB	9,697	9,686.4	216	9.9	5h24m

Table 6.2: Full-size application performance. (The performance results on of Hyperscan, GPU, and FPGA are in seconds. NA refers the automata is too large to run with GPU/Hyperscan.)

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 complication overhead and improves the overall performance.

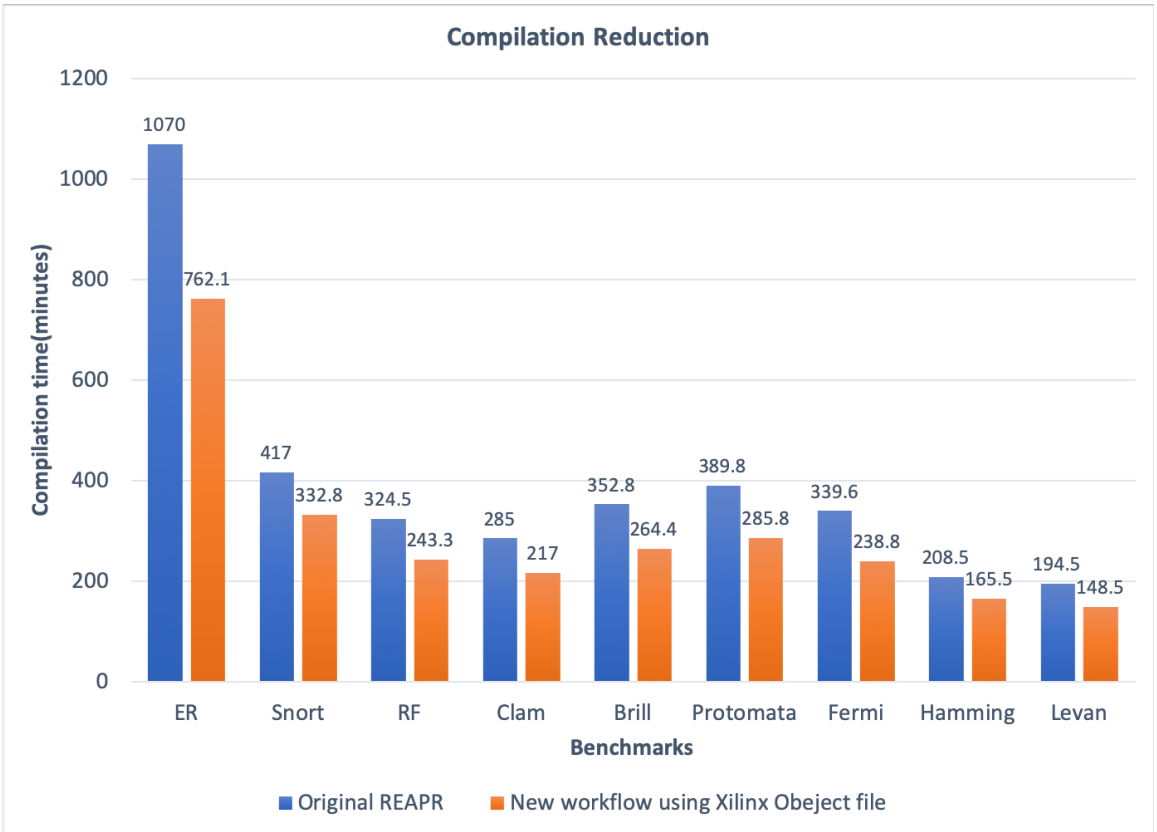

Figure 6.7: Compilation reduction using Xilinx Object file.

6.6.2 Workflow using the Xilinx Object file

In this section, we evaluate the newly proposed workflow using Xilinx Object files. Because we focus on the compilation overhead, we only compare the compilation time with the original REAPR workflow. The overall results are presented in Fig. 6.7 and Table 6.3. The new workflow reduces the overall compilation time for all nine applications selected from ANMLZoo. The reduction is from 20% to 30%, and on average, we achieve 24.8% reduction compared to the original workflow. This shows the benefits brought by using Xilinx Object files instead of directly compiling all kernel files.

To further study how using Xilinx Object file helps to reduce the overall compilation time, we present the detailed results in Fig. 6.8 and Table 6.4. For all four phases (the logic synthesis, the logic optimization, the placement, and the routing), the new workflow consumes less time than the original flow does. But for logic optimization and placement, these two phases only take around 17% of the total time on average, and thus the reduction is relatively small. Most of the reduction comes

Benchmark	**Original**	**new**	Reduction
ER	1070.0	762.1	28.77%
Snort	417.0	332.8	20.19%
RandomForest	324.5	243.3	25.02%
ClamAV	285.0	217.0	23.86%
Brill	352.8	264.4	25.06%
Protomata	389.8	285.8	26.68%
Fermi	339.6	238.8	29.68%
Hamming	208.5	165.5	20.62%
leven	194.5	148.5	23.65%

Table 6.3: Compilation reduction using Xilinx Object file. Time is in minutes.

from the synthesis and the routing phase, which take around 73% of the total time on average. Furthermore, the portion of the synthesis phase and routing phase increases as the overall time increases. For example, for Entity Resolution, the synthesis phase and routing phase consume 86.6% of the overall compilation time. Therefore, using the Xilinx Object file helps to reduce the synthesis time and the routing time, thus leading to the overall compilation time reduction.

6.6.3 Modular synthesis and reuse of I/O templates

In this section, we evaluate the potential benefits brought the proposed modular synthesis and reuse I/O circuitry method. Since the SDAccel does not support modular synthesis, we use the Vivado out-of-context(OOC) feature to simulate the proposed method [132]. As discussed in the previous section, we isolate the I/O template from the whole design. For each of the benchmark, we modify the top module. The modified top module now does not run the kernel. It only generates the I/O circuitry for the application. We collect the compilation overhead of the I/O circuitry in each application. By reusing pre-defined I/O circuitry, we do not need to re-synthesize the I/O template and only need to synthesize the kernel of the application. The results are shown in Fig. 6.9. As shown in the figure, in most of the benchmarks, compiling the I/O circuitry takes more than 50% of the overall compilation time. This is the major reason why using I/O templates can help to reduce the compilation overhead. But for Hamming Distance and Levenshtein Distance, the I/O compilation only takes a small portion of the overall compilation time. This is because, in these two benchmarks, there are only small numbers of reports (186 in Hamming and 96 in Levenshtein), and thus the I/O circuitry compilation only takes a small portion

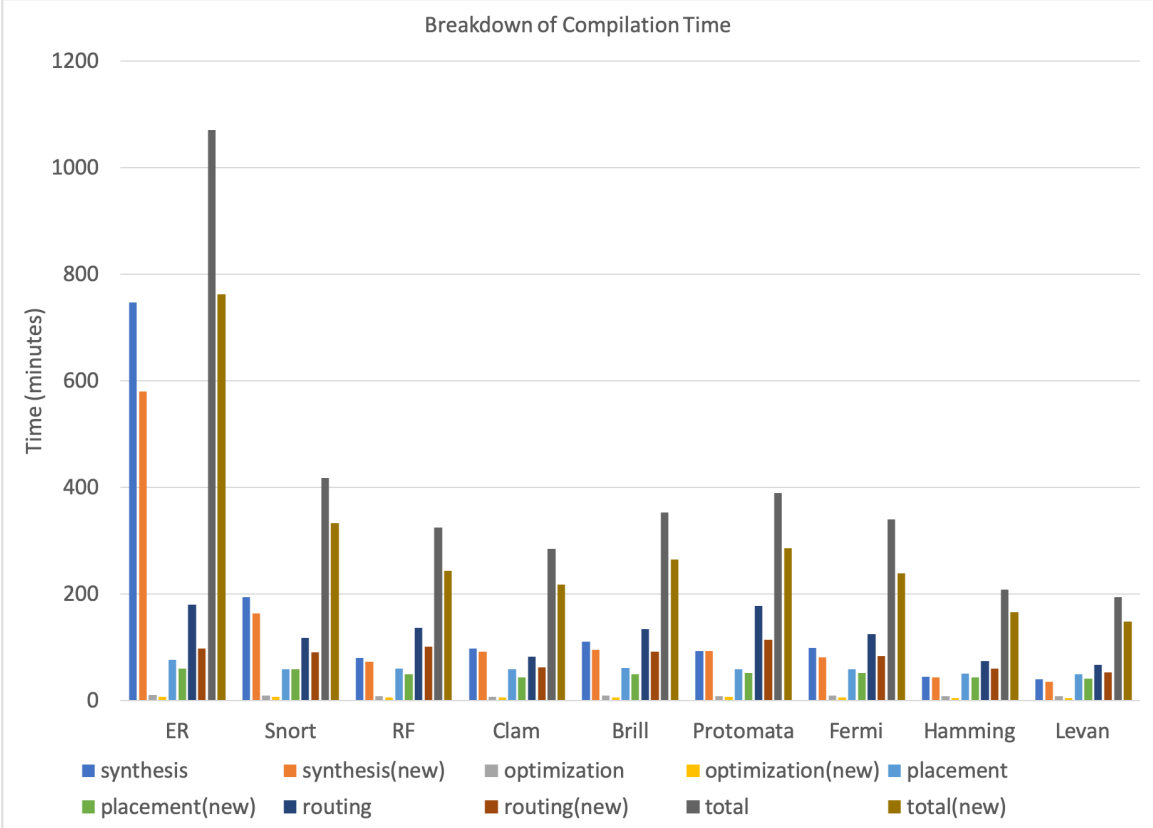

Figure 6.8: Compilation time breakdown using Xilinx Object file.

of the overall compilation. Removing the I/O part only conveys a minor speedup in such applications. However, for most real-world applications, the report number is much larger, and the I/O circuitry compilation dominates the overall compilation overhead. The grey line shows the speedups brought by removing the I/O compilation and using pre-defined I/O templates. We achieve up to 4× speedups and 2.3× speedup on average.

In Table 6.5, we present the compilation overhead for the I/O template. This is a one-time overhead and can be compiled in advance. We then present the kernel compilation overhead in Table 6.6. From the table, we can see the kernel compilation is much faster compared to I/O circuitry. Though the OOC is not supported in SDAccel, SDAccel uses Vivado in the backend. We expect to achieve similar results when OOC is supported in SDAccel.

The above results show that using I/O circuitry templates helps to reduce the compilation time. But it may also consume more hardware resources on FPGAs, because we need to use extra resources for some applications or datasets that do not

Benchmark	Synthesis	Syn(XO)	Optimization	Opt(XO)	Placement
ER	746.5	580.4	10.5	7	76.5
Snort	193.5	163	9.5	6.5	58.5
RandomForest	80.5	72.3	12	5.3	60
ClamAV	98	92	7	6	58.5
Brill	110.1	95.2	9.5	6.3	61.4
Protomata	93.4	92.5	8.6	6.5	58.8
Fermi	98.4	80.7	9.5	6.2	58.6
Hamming	44.5	43.5	8.5	5	51
leven	39.5	35.3	8	5	58.5
Benchmark	**PLM(XO)**	**Routing**	**RT(XO)**	**Total**	**Total(XO)**
ER	60	180	97	1070	762.1
Snort	58.2	117.5	90.5	417	332.8
RandomForest	48.8	136	101.3	324.5	243.3
ClamAV	43.5	82.5	61.7	285	217
Brill	49.8	133.6	91.7	352.8	264.4
Protomata	51.5	177	114.5	398.8	285.8
Fermi	51.6	124.8	83	339.6	238.8
Hamming	43	74.5	60	208.5	165.5
leven	41.5	67	53.3	194.5	148.5

Table 6.4: Compilation breakdown using Xilinx Object file. Time is in minutes. XO stands for the time consumed by the workflow using Xilinx Object file.

Report Number	Time (minutes)
1-512	189
512-1024	242
1024-1536	321
1536-2048	386
2048-2560	437

Table 6.5: Compilation overhead for I/O templates.

need as many reports as the template provides. For example, if an application has 400 reports and the template for 512 reports is used, I/O circuitry for 112 reports are consumed by the I/O template but not actually used by the application. In this section, we collect the resource utilization of CLBs and LUTs using Xilinx tools (Vivado and SDAccel), which are the major resources we utilize on FPGAs for automata processing. The results are presented in Fig. 6.10 and Fig. 6.11. The blue bar represents the resources used by the original design without I/O templates and the orange bar represents the resources used by the new design with I/O templates. For both

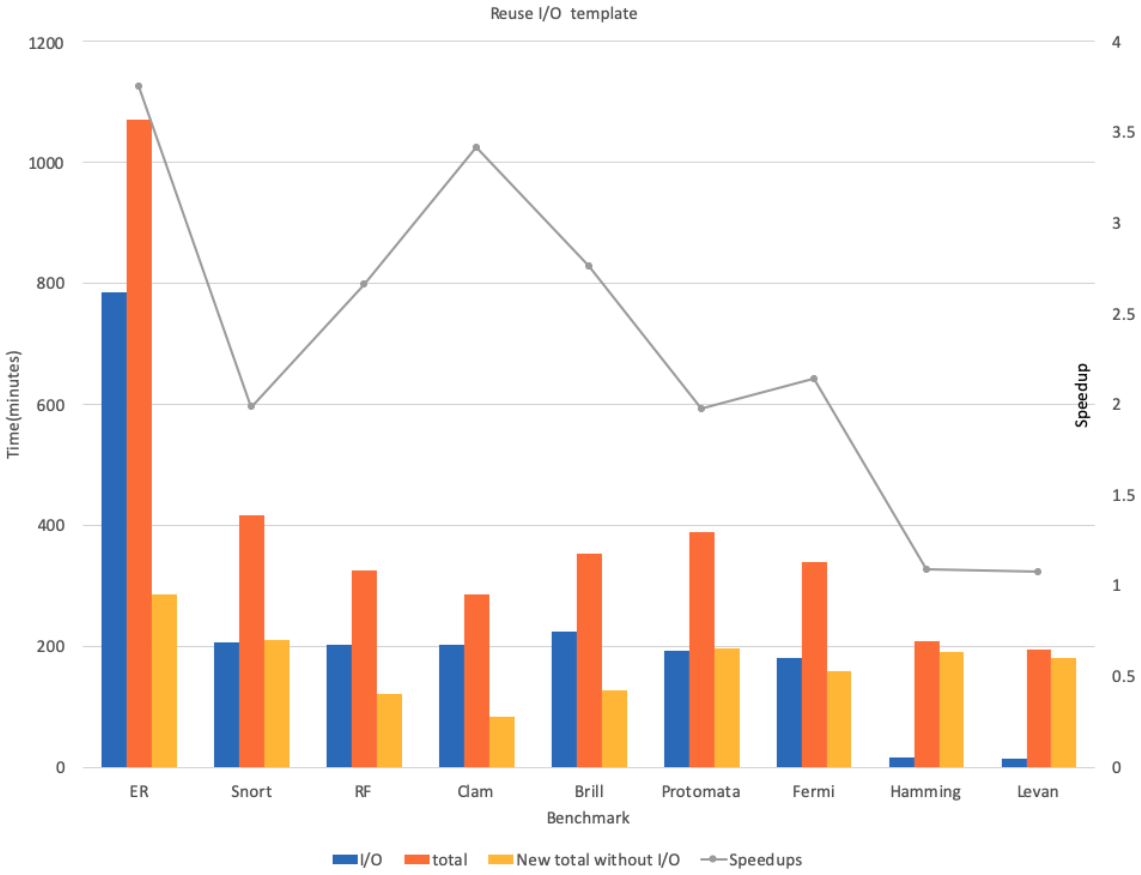

Figure 6.9: Overhead of I/O compilation and speedups using pre-synthesized I/O template.

Report Number	**Time (minutes)**
ER	17.2
Snort	12.6
RF	14.5
ClamAV	11.4
Brill	10.1
Protomata	12.4
Fermi	10.1
Hamming	9.9
Levenshtein	15.9

Table 6.6: Kernel compilation overhead

CLB and LUT utilization, using I/O templates will consume more resources than the original design as we expect. The amount of extra resources needed depends on the specific application and datasets. For the datasets in ANMLZoo, on average, we

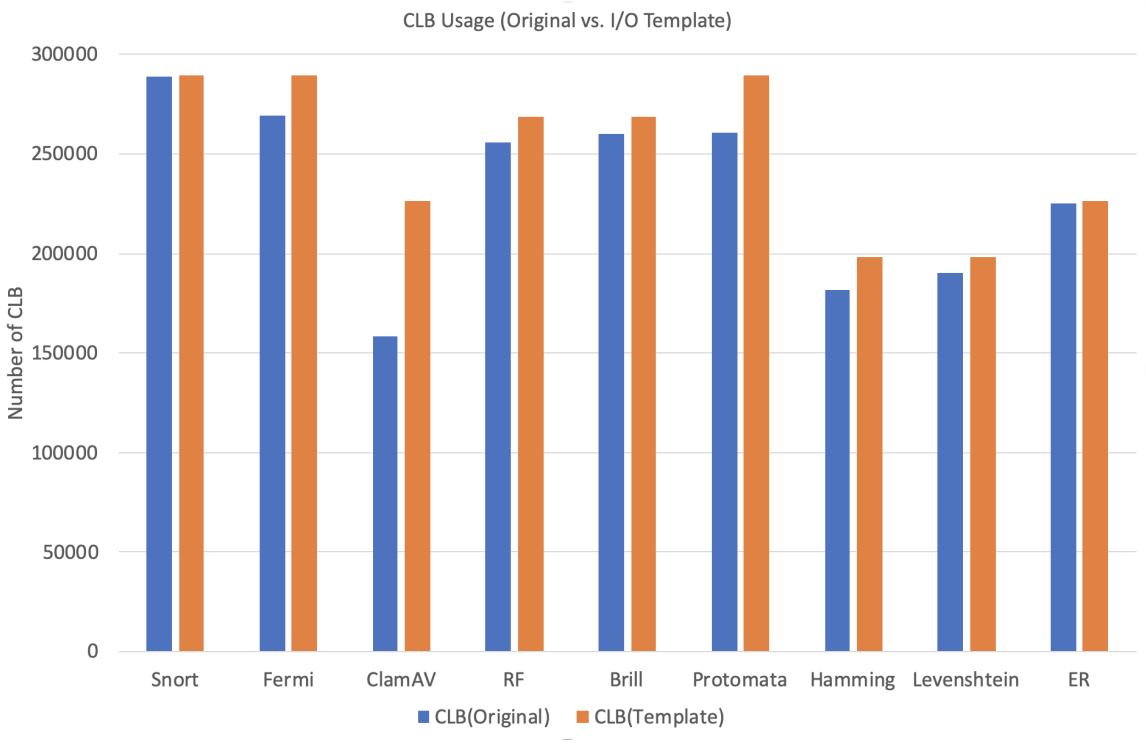

Figure 6.10: CLB usage comparison between the original I/O circuitry and using I/O templates.

consume 9.5% and 8.5% more CLB and LUT for the I/O template. However, for the ClamAV dataset provided in ANMLZoo, it consumes 43% more CLBs and 18% more LUTs because of the big gap between the actual number of reports (515) and the number of reports provided by the template (1024). In general, the amount of CLB and LUT is not the bottleneck of using FPGAs for automata processing. However, if the application consumes a lot of hardware resources, users should be careful when using I/O templates.

6.7 Discussion: Hybrid Methods

In previous sections, we present three methods that can help to reduce the high compilation overhead when using FPGAs for automata processing. We provide the details and experimental results of each method separately. However, these methods are not mutually exclusive. We can combine these new methods and further reduce the overall compilation time. In the following sections, we will present two hybrid methods.

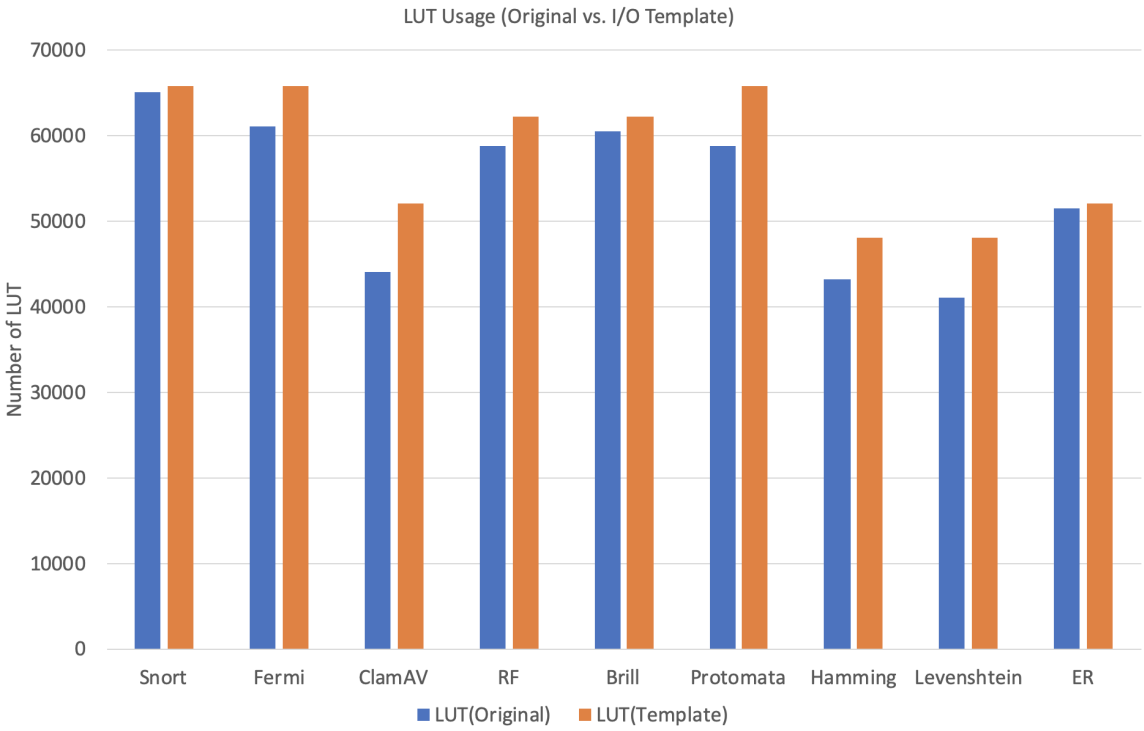

Figure 6.11: LUT usage comparison between the original I/O circuitry and using I/O templates.

New workflow using Xilinx Object files + Symbol-only reconfiguration

We can use the new workflow using Xilinx Object files for applications that could use symbol-only reconfiguration. When using the symbol-only reconfiguration feature, we only need to compile the general structure once, and the hybrid method could further reduce the one-time compilation overhead by using Xilinx Object files. For example, if we use the new hybrid method for Entity Resolution, the one-time compilation time reduces from 17h50m to 11h23m. For CRISPR, the one-time compilation time reduces from 6h23m to 5h11m. Similar to the requirements of using the symbol-only reconfiguration method, we need to design a general automaton structure for the application.

Symbol-only reconfiguration + I/O template reuse

We can use pre-compiled I/O templates for applications that support the symbol-only reconfiguration feature to reduce the one-time configuration time. By using I/O templates, we do not need to compile the I/O circuitry for every new application or dataset. Instead, we directly bind the kernel with the I/O templates. For example,

if we use the I/O template with 1,024 reports for Entity Resolution, we could save around 10 hours that are used for compiling the I/O circuitry for the original datasets. If we use the I/O template with 2,048 reports for CRISPR, we could save around 3 hours that are used for compiling the I/O circuitry for the original datasets.

Still, we need to design a general automata structure for such applications in order to use the symbol-only reconfiguration feature. But for most of automata-based applications except for regex-based applications (e.g., Snort), we are able to design a general structure. Therefore, the above two hybrid methods are very helpful for reducing the high compilation overhead for many automata-based applications.

Users have the flexibility to utilize any of the new proposed methods in this chapter or combine these methods. In this section, we only provide two possible hybrid examples to show the potential benefits of using hybrid methods. Results indicate that these hybrid methods can help to further reduce the compilation overhead compared to the separate method.

6.8 Conclusion and Future Work

6.8.1 Conclusion

In this chapter, we present three different methods to reduce the high compilation overhead for automata processing engine on FPGAs. These three methods explore the possibilities from different perspectives. Symbol-only reconfiguration is proposed to solve the challenge when multiple passes of the input stream are needed. We design a general automata structure for the applications, and compile the structure using the BRAM-based design. For each new partition, we only need to write symbols to be stored in the states instead of compiling for the new partition. The symbol-only reconfiguration work in this chapter was published in [28], and most of the contents are derived from that paper. Secondly, we propose a new workflow that uses Xilinx Object files. This new workflow helps to reduce the overall overhead by 25% on average. At last, we propose a modular synthesis mechanism and reuse the I/O circuitry. This helps to reduce the overall compilation by 2.3× on average.

These methods provide insights on how to reduce compilation overhead on FPGAs for other applications or computational kernels, such as reusing the communication circuitry between the FPGA and the host CPU.

Finally, we will talk about automata processing overlay, which could be a promis-

ing future direction to further reduce the compilation overhead with certain tool support.

6.8.2 Automata Overlay

We propose three methods in this chapter to reduce the compilation overhead for automata processing on FPGAs and achieve some promising results. Some other methods could also help to reduce the compilation overhead. One possibility is to design a generalized automata processing *automata overlay* (a pre-configured virtual automata processing architecture that overlays on top of the physical configurable fabric) on FPGAs, which only requires loading configuration files (e.g. storing status of logic gates) when launching a new automata application. In such a way, 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. Karakchi et al. propose an overlay (by fully connecting states) as an alternative to Micron's AP, but it does not apply for most of the applications in ANMLZoo [26]. This is because fully connecting states requires very large fan-in and fan-out of the states, and there are constraints on max connections of logic cells on FPGA(such as logic gates).

We propose another potential solution for automata overlay on FPGAs. We can design a few different automata overlay templates, and use the one that fits the specific application best. Fig 6.12 shows one feasible general automata overlay design. CAPB stands for Configurable Automata Processing Block. Each CAPB can be configured with different topology. For example, we present a ring structure, a mesh structure, a linear structure, and a star structure in the figure. A mesh structure can be used for Hamming distance or Levenshtein Distance designs. A linear structure can be used for regular expressions. A star structure can be used for the design with many "hot" states (which connect to many different other states). A ring structure can be used for the design with many back-traces. These structures can be pre-configured; thus we void paying the compilation cost on these structures. Ideally, each CAPB can choose any of the pre-configured structures. However, in this project, each CAPB uses the same pre-configured structure.

Furthermore, the reuse of I/O templates and modular synthesis presented in previous sections could be another direction of implementing automata overlay. Though this is not the overlay architecture for the whole automata processing procedure, it avoids re-compile the I/O circuitry, which is the most time-consuming part of the

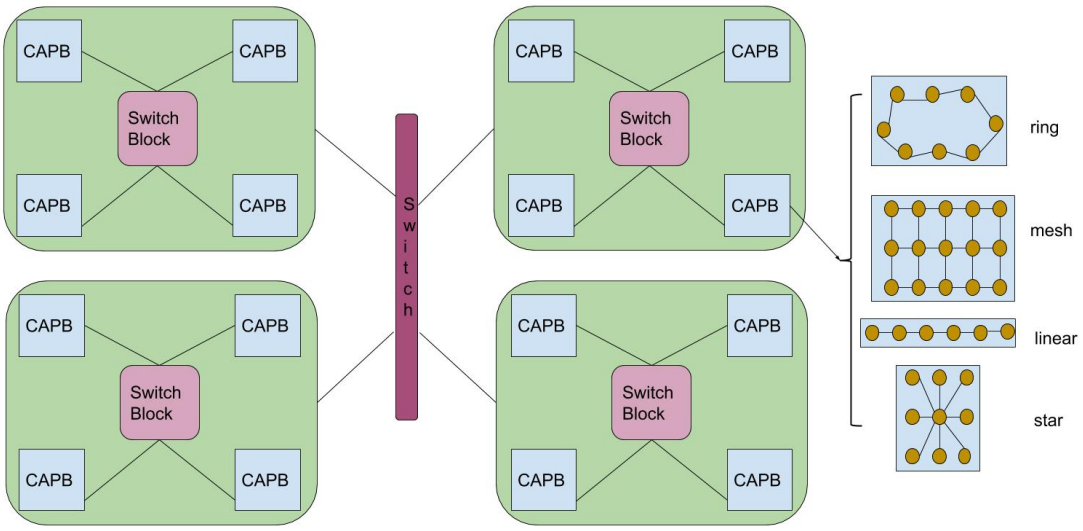

Figure 6.12: Flexible automata overlay structure.

compilation phase. Results show that it also helps to reduce the overall compilation time.

Automata overlay could help to reduce the high configuration overhead on FPAGs and allow fast reconfiguration when multiple passes of the input stream are needed. However, we do need support in FPGA development tools that allow users to pre-compile different CAPB structures and various switch blocks in order to support general automata processing. For example, Xilinx partial reconfiguration feature could be helpful for this task [133] [134] [135]. Furthermore, using pre-compiled structure usually consumes more hardware resources on FPGAs, which could be an interesting trade-off study. The overlay could be a very promising future work, which could potentially provide helpful insights for other applications using FPGAs when high compilation overhead is encountered.

Chapter 7

Conclusions and Future Work

7.0.1 Conclusions

This dissertation focuses on using automata processing for inexact pattern matching applications and providing a high-performance, scalable, and user-friendly automata processing engine on FPGAs. Inexact pattern matching is a widely-used kernel existing in many domains, such as machine learning [2] [97], cyber security [136] [137] [138], bioinformatics [27][84], anti-virus scanning [139], natural language processing [9], etc. This process is computationally expensive and is usually a bottleneck of the application.

This dissertation first proposes accelerating Entity Resolution using automata processing, which is a new application domain (Knowledge Discovery). ER is an important kernel of many information integration applications in Knowledge Discovery. Micron's Automata Processor (AP) is an efficient and scalable semiconductor architecture for parallel automata processing [22] and can process a large number of complex patterns simultaneously. Therefore, we propose using the AP to accelerate Entity Resolution. To illustrate how the AP approach works, we present a framework and several different automata designs for various ER applications. We evaluate the suitability of the prototype using several real-world ER problems in different applications. Results show both higher performance and better resolution accuracy using several datasets.

This dissertation then proposes searching for gRNA off-targets for CRISPR/Cas9 using automata processing across different platforms. Efficiently finding all correct locations to edit the genome, without modifying other locations, is the bottleneck of using the CRISPR/Cas9 system, because the gRNA sometimes binds to locations with slightly different DNA sequences [41]. To solve the above problem, we propose

an automata-based solution to identify potential off-target sites in a reference genome. We present several designs that can recognize different variations of a gRNA, and a general workflow of how to use automata processing to identify potential off-target sites. We evaluate the proposed automata approach across four different platforms (CPUs, GPUs, FPGAs, and Micron's AP) and compare it with two state-of-the-art solutions (CasOFFinder [43] and CasOT [44]). The proposed method leads to over 83× speedups on the FPGA compared with CasOFFinder (GPU), and additional speedups can be achieved by using the AP. Results also show that the automata-based method on the GPU does not confer a clear advantage because it does not map well to the GPU architecture. The above results indicate the potentials of using automata to accelerate applications involving inexact pattern matching, and spatial architectures is more appropriate for automata processing.

This dissertation then presents a new automata processing engine on cloud-based FPGAs with new features. Reconfigurable Engine for Automata PRocessing (REAPR) is a framework that can generate RTL codes for automata processing kernel and the I/O circuitry for data transfer between the FPGA and the host CPU [25]. Though REAPR shows promising results, the performance could be further improved (e.g., poor performance of the reporting architecture), and there are still several limitations of the framework (such as involving many manual efforts and high reconfiguration overhead). We extend the prior framework with several new features to improve the performance and make it more user-friendly. New features include 1). new automated workflow; 2). processing multiple symbols per cycle; 3). new reporting architecture; 4). simplified I/O integration. We implement the framework on Cloud-based platforms and conduct a cross-platform evaluation. Results show that using FPGAs provide great potential for automata processing.

This dissertation at last aims to reduce the high compilation overhead on FPGAs. High compilation overhead is a common problem when using FPGAs to accelerate applications. We encounter similar problems when using FPGAs for automata processing. It is costly to configure FPGAs for every single new rule/pattern. Furthermore, for large pattern sets or problem sizes, the automata may not fit on a single device. This requires a method to partition the automata and support multiple passes with fast reconfiguration for each partition. To reduce the configuration overhead, we propose three novel methods (symbol-only reconfiguration, a new workflow using Xilinx Object file, and modular synthesis to reuse compiled components) in the framework.

1. Symbol-only reconfiguration

To use the symbol-only reconfiguration mechanism, we need to design a general automaton structure for the application. This allows us to compile the structure once and reuse the complied structure. We do not need to compile for new partitions, and only need to update the symbols stored in the states, which is much faster.

2. New workflow using Xilinx Object file

In the original REAPR workflow, we integrate all kernels in the last step for compilation (including FPGA synthesis, logic optimization, logic placement and routing). We then notice that using Xilinx Object files can help to reduce the time in the compilation phase. Therefore, we propose a new workflow by using the Xilinx Object(.xo) file in Xilinx SDAccel.

3. Modular synthesis and reuse the I/O communication structure

We modularize the original RTL kernel and isolate the I/O circuitry between the CPU and the FPGA from the whole kernel. The I/O structure is similar for those applications with a similar number of reporting states, and we can pre-synthesize and reuse the structure for different applications.Results show that up to 4× speedup can be achieved for the overall compilation time.

The work in Chapter 3 and Chapter 4 proves that automata processing can help to accelerate applications in which the inexact pattern matching is the major performance bottleneck. The work in Chapter 5 and Chapter 6 proves that using FPGAs for automata processing provides a high-performance, scalable, and user-friendly platform for inexact pattern matching applications. All the work in the above chapters confirms the dissertation hypothesis in Chapter 1.

7.0.2 Future Work

Future directions of the dissertation include, but are not limited to:

1. Explore other inexact pattern matching applications that may benefit from using automata processing. This dissertation uses Entity Resolution in Knowledge Discovery field and gRNA off-targets identification in Bioinformatics as two example applications to show how to use automata processing to accelerate inexact pattern matching applications. We present how to design automata for recognizing various patterns and the general workflow of using automata to accelerate such applications. Many other inexact pattern matching applications in other application domains may adopt similar methods.

2. Add new features to the FPGA automata processing engine. We add several new features to the original REAPR, which makes the FPGA automata processing

engine more efficient, scalable, and user-friendly. Users may add other features for general automata processing into the platform or implement new customized features for the specific application they want to accelerate. As we mentioned in the previous chapter, the FPGA automata processing engine can not only be used for processing different automata-based applications, but can also be used as a research platform for exploring new hardware design for automata processing.

3. Design and implement automata processing overlays. We propose a couple of possible ideas on how to design and implement an automata overlay. A general automata processing overlay provides a general architecture that can process various automata-based applications, without spending a long time on the compilation and reconfiguration. Future work could explore other automata processing overlay ideas on FPGAs.

Bibliography

[1] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. "Towards machine learning on the automata processor". In: *International Conference on High Performance Computing*. Springer. 2016, pp. 200–218.

[2] Chunkun Bo, Ke Wang, Y Qi, and Kevin Skadron. "String kernel testing acceleration using the Micron Automata Processor". In: *Workshop on Computer Architecture for Machine Learning*. 2015.

[3] Vincent T Lee, Justin Kotalik, Carlo C Del Mundo, Armin Alaghi, Luis Ceze, and Mark Oskin. "Similarity search on automata processors". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 523–534.

[4] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms". In: *IEEE Communications Surveys & Tutorials* 18.4 (2016), pp. 2991–3029.

[5] Ioannis Sourdis and Dionisios Pnevmatikatos. "Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system". In: *International Conference on Field Programmable Logic and Applications*. Springer. 2003, pp. 880–889.

[6] Jeremy Leipzig. "A review of bioinformatic pipeline frameworks". In: *Briefings in bioinformatics* 18.3 (2017), pp. 530–536.

[7] Khaled Benkrid, Ying Liu, and AbdSamad Benkrid. "A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17.4 (2009), pp. 561–570.

[8] Christopher R Clark and David E Schimmel. "Scalable pattern matching for high speed networks". In: *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2004, pp. 249–257.

[9] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, and Kevin Skadron. "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)*. 2. 2018.

[10] Karl Abrahamson. "Generalized string matching". In: *SIAM journal on Computing* 16.6 (1987), pp. 1039–1051.

[11] Stephan C Schuster. "Next-generation sequencing transforms today's biology". In: *Nature methods* 5.1 (2007), p. 16.

[12] Benjamin Langmead, Kenneth M MacKenzie, Steven K Reinhardt, and Richard A Lethin. *System, Apparatus, And Methods For Pattern Matching*. US Patent App. 11/766,704. 2008.

[13] Ke Wang, Yanjun Qi, Jeffrey J Fox, Mircea R Stan, and Kevin Skadron. "Association rule mining with the micron automata processor". In: *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2015, pp. 689–699.

[14] Keira Zhou, Jeffrey J Fox, Ke Wang, Donald E Brown, and Kevin Skadron. "Brill tagging on the micron automata processor". In: *Semantic Computing (ICSC), 2015 IEEE International Conference on*. IEEE. 2015, pp. 236–239.

[15] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. "Entity resolution acceleration using the automata processor". In: *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE. 2016, pp. 311–318.

[16] Ke Wang, Elaheh Sadredini, and Kevin Skadron. "Sequential pattern mining with the Micron automata processor". In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM. 2016, pp. 135–144.

[17] InsideBigdata. *The Exponential Growth of Data*. URL: `https://insidebigdata.com/2017/02/16/the-exponential-growth-of-data/`.

[18] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. "iNFAnt: NFA pattern matching on GPGPU devices". In: *ACM SIGCOMM Computer Communication Review* 40.5 (2010), pp. 20–26.

[19] Intel. *High-performance regular expression matching library*. URL: https://github.com/intel/hyperscan.

[20] Vinh Dang. *A Deterministic Finite Automata GPU-based Engine*. URL: https://github.com/vqd8a/DFAGE.

[21] Vinh Dang. *iNFAnt2*. URL: https://github.com/vqd8a/iNFAnt2.

[22] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. "An efficient and scalable semiconductor architecture for parallel automata processing". In: *IEEE Transactions on Parallel and Distributed Systems* 25.12 (2014), pp. 3088–3098.

[23] Reetinder Sidhu and Viktor Prosanna. "Fast regular expression matching using FPGAs". In: *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2. 2001.

[24] Yi-Hua Yang and Viktor Prasanna. "High-performance and compact architecture for regular expression matching on FPGA". In: *IEEE Transactions on Computers*. 2. 2012.

[25] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. "REAPR: Reconfigurable engine for automata processing". In: *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE. 2017, pp. 1–8.

[26] Rasha Karakchi, Lothrop O. Richards, and Jason D. Bakos. "A Dynamically Reconfigurable Automata Processor Overlay". In: *Proceedings of International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2017.

[27] Chunkun Bo, Dang Vinh, Elaheh Sadredini, and Kevin Skadron. "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)*. 2. 2018.

[28] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. "Automata Processing in Reconfigurable Architectures: In-the-Cloud Deployment, Cross-Platform Evaluation, and Fast Symbol-Only Reconfiguration". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.2 (2019), p. 9.

[29] Mauricio A Hernández and Salvatore J Stolfo. "The merge/purge problem for large databases". In: *ACM Sigmod Record*. Vol. 24. 2. ACM. 1995, pp. 127–138.

[30] Lise Getoor and Ashwin Machanavajjhala. “Entity resolution: theory, practice & open challenges”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 2018–2019.

[31] Alvaro Monge and Charles Elkan. “An efficient domain-independent algorithm for detecting approximately duplicate database records”. In: (1997).

[32] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. “Entity resolution with iterative blocking”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pp. 219–232.

[33] Lars Kolb and Erhard Rahm. “Parallel entity resolution with dedoop”. In: *Datenbank-Spektrum* 13.1 (2013), pp. 23–32.

[34] Omar Benjelloun, Hector Garcia-Molina, Hideki Kawai, Tait Eliott Larson, David Menestrina, Qi Su, Sutthipong Thavisomboon, and Jennifer Widom. *Generic entity resolution in the serf project*. Tech. rep. Stanford InfoLab, 2006.

[35] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven Euijong Whang, and Jennifer Widom. “Swoosh: a generic approach to entity resolution”. In: *The VLDB JournalThe International Journal on Very Large Data Bases* 18.1 (2009), pp. 255–276.

[36] Philippe Horvath and Rodolphe Barrangou. “CRISPR/Cas, the immune system of bacteria and archaea”. In: *Science* 327.5962 (2010), pp. 167–170.

[37] Feng Zhang, Yan Wen, and Xiong Guo. “CRISPR/Cas9 for genome editing: progress, implications and challenges”. In: *Human molecular genetics* 23.R1 (2014), R40–R46.

[38] Woong Y Hwang, Yanfang Fu, Deepak Reyon, Morgan L Maeder, Shengdar Q Tsai, Jeffry D Sander, Randall T Peterson, JR Joanna Yeh, and J Keith Joung. “Efficient genome editing in zebrafish using a CRISPR-Cas system”. In: *Nature biotechnology* 31.3 (2013), p. 227.

[39] Randall J Platt, Sidi Chen, Yang Zhou, Michael J Yim, Lukasz Swiech, Hannah R Kempton, James E Dahlman, Oren Parnas, Thomas M Eisenhaure, Marko Jovanovic, et al. “CRISPR-Cas9 knockin mice for genome editing and cancer modeling”. In: *Cell* 159.2 (2014), pp. 440–455.

[40] *11 Crazy Gene-Hacking Things We Can Do with CRISPR*. http://www.popularmechanics.com/science/a19067/11-crazy-things-we-can-do-with-crispr-cas9/.

[41] Yanfang Fu, Jennifer A Foden, Cyd Khayter, Morgan L Maeder, Deepak Reyon, J Keith Joung, and Jeffry D Sander. "High-frequency off-target mutagenesis induced by CRISPR-Cas nucleases in human cells". In: *Nature biotechnology* 31.9 (2013), p. 822.

[42] *CasOT - CRISPR/Cas system (Cas9/gRNA) Off-Targeter*. http://casot.cbi.pku.edu.cn.

[43] Sangsu Bae, Jeongbin Park, and Jin-Soo Kim. "Cas-OFFinder: a fast and versatile algorithm that searches for potential off-target sites of Cas9 RNA-guided endonucleases". In: *Bioinformatics* 30.10 (2014), pp. 1473–1475.

[44] An Xiao, Zhenchao Cheng, Lei Kong, Zuoyan Zhu, Shuo Lin, Ge Gao, and Bo Zhang. "CasOT: a genome-wide Cas9/gRNA off-target searching tool". In: *Bioinformatics* 30.8 (2014), pp. 1180–1182.

[45] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, et al. "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures". In: *Workload Characterization (IISWC), 2016 IEEE International Symposium on*. IEEE. 2016, pp. 1–12.

[46] Loring Wirbel. *Xilinx sdaccel whitepaper*. 2014.

[47] Chunkun Bo. *REAPR on F1*. URL: https://github.com/chunkunbo/REARP-on-Amazon-F1.

[48] Pinaki Chakraborty, Prem Chandra Saxena, and Chittaranjan Padmanabha Katti. "Fifty years of automata simulation: a review". In: *ACM Inroads* 2.4 (2011), pp. 59–70.

[49] Harry R Lewis and Christos H Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 1997.

[50] Michela Becci and PatricK Crowley. "A hybrid finite automaton for practical deep packet inspection". In: *ACM CoNEXT conference*. 2. 2007.

[51] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. "Algorithms to accelerate multiple regular expressions matching for deep packet inspection". In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. 4. ACM. 2006, pp. 339–350.

[52] Randy Smith, Cristian Estan, and Somesh Jha. "XFA: Faster signature matching with extended automata". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 187–201.

[53] Alex X Liu, Eric Torng, Alex X Liu, and Eric Torng. "Overlay automata and algorithms for fast and scalable regular expression matching". In: *IEEE/ACM Transactions on Networking (TON)* 24.4 (2016), pp. 2400–2415.

[54] Pascal Caron and Djelloul Ziadi. "Characterization of Glushkov automata". In: *Theoretical Computer Science* 233.1-2 (2000), pp. 75–90.

[55] J-M Champarnaud. "Subset construction complexity for homogeneous automata, position automata and ZPC-structures". In: *Theoretical Computer Science* 267.1-2 (2001), pp. 17–34.

[56] Ted Leslie. "Efficient approaches to subset construction." In: (1993).

[57] Jian Chen and Xinyu Hu. *Regular expression matching method and system*. US Patent 8,756,170. 2014.

[58] Davide Pasetto, Fabrizio Petrini, and Virat Agarwal. "Tools for very fast regular expression matching". In: *Computer* 43.3 (2010), pp. 50–58.

[59] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. "Hyperscan: a fast multi-pattern regex matcher for modern CPUs". In: *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 2019, pp. 631–648.

[60] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. "SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU". In: *ACM SIGPLAN Symp. Principles Practice Parallel Program. (PPoPP)*. 2019.

[61] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. "Fast Segmented Sort on GPUs". In: *Proceedings of the 2017 International Conference on Supercomputing*. ICS '17. Chicago, IL, USA: ACM, 2017.

[62] Kaixi Hou, Hao Wang, and Wu-chun Feng. "GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs". In: *Proceedings of the ACM Conference on Computing Frontiers*. CF '17. Siena, Italy: ACM, 2017.

[63] K. Hou, H. Wang, W. Feng, J. Vetter, and S. Lee. "Highly Efcient Compensation-based Parallelism for Wavefront Loops on GPUs". In: *IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS)*. 2018.

[64] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. "An evaluation of unified memory technology on nvidia gpus". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 1092–1098.

[65] Xuewen Cui, Thomas RW Scogland, Bronis R de Supinski, and Wu-Chun Feng. "Directive-based pipelining extension for openmp". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2016, pp. 481–484.

[66] Xuewen Cui, Thomas RW Scogland, Bronis R de Supinski, and Wu-chun Feng. "Directive-based partitioning and pipelining for graphics processing units". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 575–584.

[67] Xiaodong Yu and Michela Becchi. "GPU acceleration of regular expression matching for large datasets: exploring the implementation space". In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM. 2013, p. 18.

[68] Ciprian Pungila and Viorel Negru. "A highly-efficient memory-compression approach for GPU-accelerated virus signature matching". In: *International Conference on Information Security*. Springer. 2012, pp. 354–369.

[69] Lei Wang, Shuhui Chen, Yong Tang, and Jinshu Su. "Gregex: Gpu based high speed regular expression matching engine". In: *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE. 2011, pp. 366–370.

[70] K. Hou, H. Wang, and W. c. Feng. "Delivering Parallel Programmability to the Masses via the Intel MIC Ecosystem: A Case Study". In: *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 273–282.

[71] Kaixi Hou. “Exploring Performance Portability for Accelerators via High-level Parallel Patterns”. PhD thesis. Virginia Tech, 2018.

[72] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. “Parallelization and characterization of pattern matching using GPUs”. In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2011, pp. 216–225.

[73] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D’Antoni, and Thomas F. Wenisch. “HARE: Hardware accelerator for regular expressions”. In: *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2. 2016.

[74] Yuanwei Fang, Tung T. Hoang, Michela Becci, and Andrew A. Chien. “HARE: Hardware accelerator for regular expressions”. In: *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2. 2015.

[75] Arun Subramaniyan, Jingcheng Wang, Ezhil RM Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. “Cache automaton”. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2017, pp. 259–272.

[76] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. “A Scalable and Efficient in-Memory Interconnect Architecture for Automata Processing”. In: *IEEE Computer Architecture Letters* (2019).

[77] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. “eAP: A Scalable and Efficient in Memory Accelerator for Automata Processing”. In: *Proceedings of the 52th Annual IEEE/ACM International Symposium on Microarchitecture* (2019).

[78] Yun Qu, Yi-Hua E Yang, and Viktor K Prasanna. “Large-scale multi-flow regular expression matching on fpga”. In: *2012 IEEE 13th International Conference on High Performance Switching and Routing*. IEEE. 2012, pp. 70–75.

[79] Johnny Tsung Lin Ho. “PERG-Rx: an FPGA-based pattern-matching engine with limited regular expression support for large pattern databases”. PhD thesis. University of British Columbia, 2009.

[80] Ioannis Sourdis, João Bispo, Joao MP Cardoso, and Stamatis Vassiliadis. “Regular expression matching in reconfigurable hardware”. In: *Journal of Signal Processing Systems* 51.1 (2008), pp. 99–121.

[81] Neil Duxbury. *Aho-Corasick methodology for string searching*. US Patent 7,769,788. 2010.

[82] Tran Trung Hieu and Ngoc Thinh Tran. "A memory efficient FPGA-based pattern matching engine for stateful NIDS". In: *The Fifth Annual IEEE Conference Ubiquitous and Future Networks (ICUFN)*. 2. 2013.

[83] Ciprian Pungila. "Hybrid compression of the Aho-Corasick automaton for static analysis in intrusion detection systems". In: *International Joint Conference CISIS12-ICEUTE 12-SOCO 12 Special Sessions*. Springer. 2013, pp. 77–86.

[84] Indranil Roy and Srinivas Aluru. "Finding motifs in biological sequences using the micron automata processor". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, pp. 415–424.

[85] *Social Networks and Archival Context*. `http://socialarchive.iath.virginia.edu`.

[86] H Noyes et al. "Microns automata processor architecture: Reconfigurable and massively parallel automata processing". In: *Proc. of Fifth International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*. 2014.

[87] *Apache Lucene*. `http://lucene.apache.org`.

[88] Michael Ley. "The DBLP computer science bibliography: Evolution, research issues, perspectives". In: *International symposium on string processing and information retrieval*. Springer. 2002, pp. 1–10.

[89] II Tommy Tracy, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. "Nondeterministic finite automata in hardware-the case of the Levenshtein automaton". In: *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[90] *Fuzzy document finding in Ruby*. `https://github.com/brianhempel/fuzzy_tools`.

[91] *Duplicate Detection, Record Linkage, and Identity Uncertainty: Datasets*. `http://www.cs.utexas.edu/users/ml/riddle/data.html`.

[92] David Menestrina, Steven Euijong Whang, and Hector Garcia-Molina. "Evaluating entity resolution results". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 208–219.

[93] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. "Entity resolution acceleration using Microns Automata Processor". In: *Proceedings of Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[94] Zhang Tengyu and Melissa A Mefford. "Gene editing in yeast cells using the CRISPR/Cas9 system". In: (2019).

[95] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea Stan, and Kevin Skadron. "An overview of micron's automata processor". In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. ACM. 2016, p. 14.

[96] Ke Wang, Kevin Skadron, and Elaheh Sadredini. *Disjunctive rule mining with finite automaton hardware*. US Patent App. 15/475,819. 2018.

[97] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. "Frequent subtree mining on the automata processor: Challenges and opportunities". In: *Proceedings of the International Conference on Supercomputing*. ACM. 2017, p. 4.

[98] Jack Wadden, Nathan Brunelle, Ke Wang, Mohamed El-Hadedy, Gabriel Robins, Mircea Stan, and Kevin Skadron. "Generating efficient and high-quality pseudo-random behavior on Automata Processors". In: *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE. 2016, pp. 622–629.

[99] Dimitre R Simeonov and Alexander Marson. "CRISPR-based tools in immunity". In: *Annual review of immunology* 37 (2019), pp. 571–597.

[100] Adrian Pickar-Oliver and Charles A Gersbach. "The next generation of CRISPR–Cas technologies and applications". In: *Nature Reviews Molecular Cell Biology* (2019), p. 1.

[101] Kay Prüfer, Udo Stenzel, Michael Dannemann, Richard E Green, Michael Lachmann, and Janet Kelso. "PatMaN: rapid alignment of short sequences to large databases". In: *Bioinformatics* 24.13 (2008), pp. 1530–1531.

[102] Ben Langmead, Cole Trapnell, M Pop, and SL Salzberg. "Bowtie: An ultrafast memory-efficient short read aligner". In: *Genome Biol* 10.3 (2009), R25.

[103] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows–Wheeler transform". In: *bioinformatics* 25.14 (2009), pp. 1754–1760.

[104] Xiao-Hui Zhang, Louis Y Tee, Xiao-Gang Wang, Qun-Shan Huang, and Shi-Hua Yang. "Off-target effects in CRISPR/Cas9-mediated genome engineering". In: *Molecular Therapy-Nucleic Acids* 4 (2015).

[105] Ritambhara Singh, Cem Kuscu, Aaron Quinlan, Yanjun Qi, and Mazhar Adli. "Cas9-chromatin binding information enables more accurate CRISPR off-target prediction". In: *Nucleic acids research* 43.18 (2015), e118–e118.

[106] *Levenshtein in ANMLZoo*. `https://github.com/jackwadden/ANMLZoo/tree/master/Levenshtein`.

[107] Xiaodong Yu, Kaixi Hou, Hao Wang, and Wu-chun Feng. "Robotomata: A framework for approximate pattern matching of big data on an automata processor". In: *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE. 2017, pp. 283–292.

[108] Xilinx Inc. *SDAccel development environment*. URL: `https://www.xilinx.com/products/design-tools/softwarezone/sdaccel.html`.

[109] *Amazon EC2 F1 Instances*. `https://aws.amazon.com/ec2/instance-types/f1/`.

[110] Nimbix. *Xilinx FPGAs on the Nimbix Cloud*. URL: `https://www.nimbix.net/xilinx/`.

[111] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu chun Feng, and Michela Becchi. "Demystifying Automata Processing: GPUs, FPGAs or Micron's AP?" In: *Proceedings of the International Conference on Supercomputing*. ACM. 2017, p. 1.

[112] AWS. *Amazon EC2 F1 Instances*. URL: `https://aws.amazon.com/ec2/instance-types/f1/`.

[113] Ted Xie, Vinh Dang, Jack Wadden, Mircea Stan, and Kevin Skadron. "An End-to-End Reconfigurable Engine for Automata Processing". In: *Government Microcircuit Applications adn Critical Technology (GOMACTech), 2018 50th Conference on*. 2018.

[114] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. "Optimization of regular expression pattern matching circuits on FPGA". In: *Proceedings of the Design Automation & Test in Europe Conference*. Vol. 2. IEEE. 2006, pp. 1–6.

[115] Jack Wadden, Kevin Angstadt, and Kevin Skadron. "Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures". In: *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2. 2018.

[116] Ted Xie, Vinh Dang, Chunkun Bo, Jack Wadden, Kevin Skadron, and Mircea Stan. *Reconfigurable Engine for Automata Processing*. 2018. URL: https://github.com/ted-xie/REAPR.

[117] Micron. *ANML Documentation*. 2015. URL: http://www.micronautomata.com/documentation/anml_documentation/c_intro.html.

[118] Kevin Angstadt, Jack Wadden, Vinh Dang, Ted Xie, Dan Kramp, Westley Weimer, Mircea Stan, and Kevin Skadron. "MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem". In: *IEEE Computer Architecture Letters* 17.1 (2018), pp. 84–87.

[119] AWS-FPGA. *Official repository of the AWS EC2 FPGA Hardware and Software Development Kit*. URL: https://github.com/aws/aws-fpga.

[120] Jack Wadden and Kevin Skadron. *VASim: An open virtual automata simulator for automata processing application and architecture research*. Tech. Rep. CS2016–03, University of Virginia. University of Virginia, 2016.

[121] Ajay Jagtiani. *Reducing FPGA Compile Time Using Parallel Compilation Methodology*. URL: https://www.eetimes.com/document.asp?doc_id=1276054.

[122] James Coole and Greg Stitt. "Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts". In: *IEEE Micro* 34.1 (2014), pp. 42–53.

[123] Christopher Lavin, Marc Padilla, Jaren Lamprecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. "HMFlow: accelerating FPGA compilation with hard macros for rapid prototyping". In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2011, pp. 117–124.

[124] João MP Cardoso and Horácio C Neto. "Compilation for FPGA-based reconfigurable hardware". In: *IEEE Design & Test of Computers* 20.2 (2003), pp. 65–75.

[125] Jens Teubner, Louis Woods, and Chongling Nie. "Skeleton automata for FPGAs: reconfiguring without reconstructing". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 229–240.

[126] Michael L Metzker. "Sequencing technologiesthe next generation". In: *Nature reviews genetics* 11.1 (2010), p. 31.

[127] Roger Moussalli, Mariam Salloum, Robert Halstead, Walid Najjar, and Vassilis J. Tsotras. "A Study on Parallelizing XML Path Filtering using Accelerators". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4 (2014), p. 93.

[128] Xilinx. *XOCC (Xilinx OpenCL Compiler) Command Line Utility*. URL: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/wrj1504034328013.html.

[129] Xilinx. *Package RTL Kernel into Xilinx Object File*. URL: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/lfy1504034326094.html.

[130] Jack Wadden, Tommy Tracy II, Elaheh Sadredini, Lingxi Wu, Chunkun Bo, Jesse Du, Yizhou Wei, Matthew Wallace, Jeffrey Udall, Mircea Stan, and Kevin Skadron. "AutomataZoo: A Modern Automata Processing Benchmark Suite". In: *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*. 2018.

[131] Chunkun Bo. *Identify gRNA off-targets for CRISPR/Cas9 using Automata Processing*. 2018. URL: https://github.com/chunkunbo/CRISPR.

[132] Xilinx Inc. *Vivado Design Suite User Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug901-vivado-synthesis.pdf.

[133] Kizheppatt Vipin and Suhaib A Fahmy. "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications". In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 72.

[134] Xilinx Inc. *Vivado Design Suite User Guide*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug909-vivado-partial-reconfiguration.pdf.

[135] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. "Case for fast fpga compilation using partial reconfiguration". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2018, pp. 235–2353.

[136] Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, and Tsern-Huei Lee. "Using string matching for deep packet inspection". In: *Computer* 41.4 (2008), pp. 23–28.

[137] Brad L Hutchings, Rob Franklin, and Daniel Carver. "Assisting network intrusion detection with reconfigurable hardware". In: *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2002, pp. 111–120.

[138] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. "Compiling pcre to fpga for accelerating snort ids". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM. 2007, pp. 127–136.

[139] Martin Roesch. "Snort: Lightweight intrusion detection for networks." In: *Lisa*. Vol. 99. 1. 1999, pp. 229–238.