Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration with Memory-Centric Architectures

---

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 (Computer Engineering)

by

Reza Rahimi

December 2020

Abstract

The growing need for accelerated pattern recognition and inexact pattern matching has motivated many efforts to design finite-automata-based, in-memory pattern-processing accelerators. However, the lack of a standard, scalable, open-source, and easy-to-modify framework has made it difficult to develop new applications and explore new architectural innovations. Moreover, none of the existing in-memory accelerators is designed to process multiple symbols at once. In addition, none has a reliable, efficient, and scalable reporting architecture to gather and analyze the reporting data.

This proposal outlines four novel software and hardware contributions to improve the effectiveness of pattern processing on big-data applications with real-time processing needs. (1) We propose a robust, easy-to-use/modify automata processing simulation, transformation, optimization, and performance modeling framework to facilitate automata application development and architectural explorations/innovations. (2) Motivated by our application analysis enabled by our framework, we observe a significant resource underutilization in the existing memory-based accelerators. We leverage this observation and propose an area-efficient, high-throughput, and energy-efficient in-SRAM architecture for multi-symbol pattern processing. (3) Inspired by our study on multi-symbol pattern matching in in-memory architectures, we explore temporal multi-symbol matching on FPGA platforms. Our framework enables us to process different bitwidths, which has shown to be beneficial for mapping differnet applications to platforms with different architectural parameters. (4) Finally, we analyze the reporting architecture of the existing state-of-the-art memory-centric solutions, and we find that these reporting architectures are either the major source of the area/performance inefficiency, or they are not scalable and general solutions. To address these issues, we propose a compact, reconfigurable, and easy-to-handle in-situ reporting architecture by re-purposing SRAM subarrays with negligible hardware overhead.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Engineering)

Reza Rahimi

This dissertation has been read and approved by the Examining Committee:

Kevin Skadron , Adviser

Joanne Bechta Dugan

Mircea Stan

Brad Campbell

Ashish Venkat

Accepted for the School of Engineering and Applied Science:

Craig Benson, Dean, School of Engineering and Applied Science

December 2020

DEDICATION

I dedicate this dissertation to my wife, Elaheh, who has been a constant source of support, happiness, and encouragement during the challenges of my life from the first day I met her. I have learned living with passion from you and I am truly thankful for having you in my life.

Acknowledgements

I am grateful to numerous people who have contributed towards shaping this dissertation. I could not have reached the finish line without the influence, advice, and support of many colleagues, friends, and family.

First, I would like to thank my advisor, Prof. Kevin Skadron, who has been a supportive mentor for me and has shaped my life in many profound ways. He always trusted in me and gave me tremendous freedom in pursuing my ideas, and this has made me an independent researcher. At the same time, he continued to contribute valuable feedback and encouragement. I will forever treasure the advice I have received from him.

I also wish to thank the members of my Ph.D. committee; Prof. Mircea Stan, Prof. Ashish Venkat, Prof. Joanne Dugan, and Prof. Brad Campbell for their many insightful discussions and suggestions on my research. I am grateful for having so many friends and collaborators both inside and outside of the Computer Engineering Department. To Tommy Tracy, Jack Wadden, Marzieh Lenjani, Viabhav Verma, Sergui Mosanu, Chunkun Bo, Oluwole Jaiyeoba, and Alif Ahmed; thank you for your help and support. Last but not least, I am deeply thankful to my family, for their unconditional love, support, and sacrifices.

Chapter 3 describes the framework that has enabled several papers [1, 2, 3, 4, 5]. The dissertation author was the primary investigator of this framework.

Chapter 4, in full, is a reprint of the material as it appears in proceeding of FCCM 2020 [2]. Rahimi, Reza; Sadredini, Elaheh; Stan, Mircea; Skadron, Kevin, "Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs", The 28th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2020, **Best paper nominee.** The dissertation author is the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in proceeding of HPCA 2020 [1]. Rahimi, Reza; Sadredini, Elaheh; Lenjani, Marzieh; Stan, Mircea; Skadron, Kevin, "Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching", The 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA), 2020. **Best paper nominee.** The dissertation author is the primary investigator and author of this paper.

Chapter 6, partially, is a preprint of the material as it appears in Computer Architecture Letters, 2020. Rahimi, reza; Sadredini, Elaheh; Skadron, Kevin, "Enabling In-SRAM Pattern Processing with Low-Overhead Reporting Architecture", Computer Architecture Letters, 2020.

The longer version is currently under review. Rahimi, Reza; Sadredini, Elaheh; Skadron, Kevin, "Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration." The dissertation author is the primary investigator and author of these papers.

The dissertation author is the secondary investigator and author of following related papers.

- Sadredini, Elaheh; Rahimi, Reza; Lenjani, Marziyeh; Stan, Mircea; Skadron, Kevin, "FlexAmata: A Universal and Efficient Adaption of Applications to Spatial Automata Processing Accelerators", The 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.
- Sadredini, Elaheh; Rahimi, Reza; Verma, Vaibhav; Stan, Mircea; Skadron, Kevin, "A Scalable and Efficient in-Memory Interconnect Architecture for Automata Processing", Computer Architecture Letters (CAL), 2019.
- Sadredini, Elaheh; Rahimi, Reza; Verma, Vaibhav; Stan, Mircea; Skadron, Kevin, "eAP: A Scalable and Efficient in-Memory Accelerator for Automata Processing", 52th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52), 2019.
- Angstadt, Kevin; Subramaniyan, Arun; Sadredini, Elaheh; Rahimi, Reza; Skadron, Kevin; Weimer, Westley; Das, Reetu, "ASPEN: A Scalable In-SRAM Architecture for Pushdown Automata", 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'51), 2018.
- Sadredini, Elaheh; Guo, Deyuan; Bo, Chunkun; Rahimi, Reza; Skadron, Kevin; Wang, Hongning, "A Scalable Solution for Rule-Based Part-of-Speech Tagging on Novel Hardware Accelerators", 24th ACMSIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2018.
- Sadredini, Elaheh; Rahimi, Reza; Wang, Ke; Skadron, Kevin, "Frequent Subtree Mining on the Automata Processor: Opportunities and Challenges", International Conference on Supercomputing (ICS), 2017.

This dissertation is funded, in part, by the NSF (CCF-1629450) and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's big data era, and with the rise of AI, mobile, and IoT applications, processing and analyzing large data sets drives the demand for more computation and puts even larger demand on the memory and storage infrastructure. The problem is more challenging when a real-time analysis is critical (e.g., in edge devices). However, classic Moore's Law is slowing down, Dennard scaling has stopped; the amount of computation per unit cost and power is no longer increasing at its historic rate. At the same time, the performance gap between processing units and memory is increasing (also known as the *memory wall* [8]), and this is a major source of performance bottleneck for memory-bound, data-intensive applications. A further source of inefficiency is that the raw bandwidth at the edge of data arrays is 1-2 orders of magnitude greater than what can be transmitted off-chip. Therefore, many of the questions we want to explore with growing data remain unanswerable because we lack the computational power to analyze these huge data sets in a timely and cost-effective fashion.

Solving these challenges and enabling the next-generation of data-intensive applications requires computing to be embedded near the data. A recent trend in accelerator design is combining processing logic with memory elements, aka *processing-in-memory (PIM)*, to perform computation as close to where the data are located as possible, in order to exploit massive bandwidth and scalable parallelism [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. This extracts the full compute and storage power from the underlying hardware and allows superior performance and energy-efficiency, especially for memory-bound big-data applications by removing the data movement cost.

Complex and inexact pattern matching kernels in big-data domains, where millions of patterns should be processed at once, are extremely memory-bound [23, 24, 25, 26, 27, 28, 29, 30]. High-throughput and concurrent processing of thousands of patterns on each byte of an input stream is critical for many applications

with real-time and high-throughput processing requirements, such as network intrusion detection, spam filters, virus scanners, and many more. One prominent way of processing patterns is to use regular expressions. They are used in identifying complex patterns and variants of base patterns, potentially the most time-consuming task in many big-data applications.

Regular expressions are a widely used pattern specification language, and they are efficiently implemented via Finite Automata (a form of finite state machines) [31], with important applications in network security [32, 33, 34, 35, 36], log analysis [37], and many more. There are many other applications in domains such as data-mining [27, 24, 38, 25], bioinformatics [39, 23], machine learning [40, 41], natural language processing [26, 29], big data analytics [42], and even particle physics [43], that shown to greatly benefit from accelerated automata processing. The automata structure in these applications differ significantly in static structure and dynamic behavior from existing regular expression benchmarks [44, 45]. These properties, combined with increasing data being collected in big-data domain and time-sensitive applications, make pattern matching even more challenging for these applications.

A regular expression can be represented by either deterministic finite automata (DFA) or non-deterministic finite automata (NFA), which are equivalent in computational power. On CPUs, NFAs and DFAs are represented by tables indicating each state's successor state(s) upon a rule match. DFAs are often the basis for implementing automata on CPUs, because they have predictable memory bandwidth requirements (one lookup per input); while an NFA may have many active states and may require many state lookups to process a single input symbol (potentially leading to a very large memory bandwidth requirement), a DFA requires just one. On the other hand, DFA tables are often too large to fit in the processor caches, because DFAs often suffer an exponential increase in the number of states relative to NFAs [46] (See Chapter 2 for more detail).

Graphics processing units (GPUs) provide a large number of parallel resources, which can help in hiding the DRAM access latency. However, highly-random access patterns in automata processing exhibit poor memory locality and increase branch divergence and need for synchronization [44, 47, 48, 49]. Despite its foundation's importance, the efficient implementation of finite automata processing remains a challenging open research problem and the subject of extensive research.

Researchers are increasingly exploiting memory-centric hardware accelerators to meet demanding real-time requirements as performance growth in conventional processors is slowing down. The growing demand for accelerated automata processing has motivated many efforts in designing regular expression and general automata accelerators on ASIC [50, 37], FPGAs [51, 52, 36, 53, 54], and processing-in-memory (PIM) designs [55, 30, 56, 4, 57]. Spatial memory-centric accelerators, such as FPGAs and in-memory solutions, provide a reconfigurable substrate to lay out the rules in hardware by placing-and-routing automata states and connections onto a pool of hardware units in logic- or memory-based fabrics. This allows a large number of

automata to be executed in parallel, up to the hardware capacity, in contrast to von Neumann architectures such as CPUs that must handle one rule at a time in each core. Moreover, real-world automata benchmarks are often extensive in terms of state count, too big to fit in a single hardware unit, and in current memory-centric architectures, usually need multiple rounds of reconfiguration and re-processing of the data. Therefore, design density plays a vital role in overall performance.

Automata processing research focuses mainly on two dimensions: (1) application development, where obvious applications (e.g., regular expressions) or non-obvious applications (e.g., frequent subtree mining [27], sequential pattern mining [24], part-of-speech tagging [26], entity resolution [42], genome sequencing [39], random forest [58], etc.) are mapped to automata computation and processed on automata processing hardware, and (2) architectural development [4, 55, 59]. Unfortunately, the available tools for both research areas are not comprehensive, parametrizable, and easy-to-modify. Different components are scattered among multiple tools, which make it even harder to explore new research ideas.

In-memory automata processing model has three processing stages, state-matching, state-transition, and report-gathering, and can be combined in a pipeline fashion. In the state-matching stage, the current input symbol is decoded and all the states whose symbols match against it are detected by reading a memory row. In the state-transition stage, successors of active states are determined by propagating signals via an interconnect. In the report-gathering phase, the report data is gathered and analyzed for the final action or decision.

In the existing in-memory automata accelerators, 50%-70% of hardware resources are spent for state-matching [56, 60, 30, 4]. We study the state-matching resource utilization across a diverse set of automata benchmarks, and we found 86% of the time, only 3% of resources are utilized! This is mainly because in all these architectures, each state is modeled with a memory column of size 256, and 8-bit symbols are one-hot encoded in the memory columns to be able to accept a range of symbols (up to 256 symbols) in each state. However, the number of symbols accepted by a state is fewer than 8 symbols 86% of the time. This, in turn, implies that the classic approach of one-hot encoding for matching drastically over-provisions state-matching resources, which incurs significant performance penalties and leads to an inefficient and costly design. Our key observations are: (1) all these architectures are based on 8-bit symbol processing (derived from ASCII), and our analysis on a large set of real-world automata benchmarks reveals that the 8-bit processing dramatically underutilizes hardware resources for state-matching, and (2) multi-stride symbol processing, a major source of throughput growth, is not explored in the existing in-memory solutions.

Moreover, prior work has either neglected the real cost of providing a reporting architecture [30, 1] or incur a significant cost when it is considered accurately. For example, the reporting architecture in the Automata Processor [61] has 40% area overhead and up to 46× performance overhead over the ideal case

due to buffer flush overhead. Other reporting solutions [62] suffer from congested routing, frequent stalling, and lack of control on the hardware for efficient report analysis, and all these make the realistic in-memory pattern processing very inefficient and hard to scale-up.

1.1 Contributions

In this dissertation, **We hypothesize that efficient software/hardware methodology can enable performance, power, and area improvements in memory-centric pattern matching accelerators over CPU/GPU solutions, Cache Automaton, and Micron's Automata Processor.** To evaluate this hypothesis, this dissertation proposes to address several previously unexplored problems in memory-centric automata accelerators by investigating the strength and limits of existing memory-centric automata processing accelerators, and (1) developing an automata processing framework to address the issues of existing tools and provide a flexible, powerful, and easy-to-use/modify structure to explore new research ideas, (2) investigating efficient algorithm/hardware design methodologies for automata processing model to support multi-symbol processing on FPGAs and in-memory solutions, which enables performance, power, and area efficiency, and (3) proposing an efficient, compact, and scalable in-situ reporting architecture to enable a realistic in memory-centric architecture for next generation pattern processing accelerators.

1.1.1 APSim: Automata Simulation and Transformation

In this chapter, we present an open-source automata processing simulator and a compiler that simulates, minimizes, compresses, and transforms a set of automata to different symbol sizes, and efficiently maps them to the hardware resources in memory-centric architectures, such as FPGAs, Cache Automaton [30], eAP [4], etc. APSim provides a parameterizable environment to experiment with bitwidth sizes on various platforms. A set of bitwise and symbol-wise automata transformation and minimization algorithms are developed to enable these explorations.

APSim addresses the state and transition placement in an automaton by converting it to the subgraph isomorphism problem, and applies the genetic algorithm approach to map states/transitions to their physical locations. Moreover, unlike prior works, APSim uses a well-known and actively maintained python graph processing package as its main building block to benefit from its reliability, speed, and massive documentation with examples for further development by other collaborators.

1.1.2 Multi-Symbol Processing on FPGAs

Processing multiple symbols per cycle (also known as symbol striding) in regular expression has been explored in several prior work [63, 64, 65, 36] in academia. However, currently, there is no such tool publicly available for automata processing to be used by an end-user for setting up an FPGA-based multi-symbol pattern matching system.

In this chapter, we present *Grapefruit* (**G**eneral and **R**econfigurable **A**utomata **P**roc**E**ssing **FR**amework **U**sing **I**ntegrated Reporting and In**T**erconnect). We prioritize flexibility, extensibility, and scalability while developing this tool to provide an easy-to-understand interface and easy-to-modify code for other researchers to explore new features and design parameters.

Grapefruit integrates our multi-symbol processing solution on FPGA into our APSim framework to transform any given input automaton (which process one symbol per cycle) to a new automaton that processes multiple symbols per cycle where the number of symbols can be selected by the user. In addition, this tool will provide multiple FPGA design choices such as symbol-width conversion by FlexAmata [66], freedom in BRAM/LUT matching selection for every state in every symbol, BRAM matching sharing for states with the same matching condition on the same input symbol, and alphabet compression.

Our results confirm that we are achieving 9%-80% higher frequency in a single-stride solution than prior works that are not fully end-to-end (including reporting and I/O) and 3.4× higher throughput in a multi-stride solution than a single-stride solution.

1.1.3 Algorithm/Architecture Methodology for In-Memory Multi-Stride Pattern Matching

From studying prior architectures, we observed that: (1) all these architectures are based on 8-bit symbol processing (derived from ASCII), and our analysis on a large set of real-world automata benchmarks reveals that the 8-bit processing dramatically underutilizes hardware resources, and (2) multi-stride symbol processing, a major source of throughput growth, is not explored in the existing in-memory solutions.

This chapter presents Impala [1], a software/hardware design methodology for multi-stride in-memory automata processing by leveraging our observations. The key insight of our work is that transforming 8-bit processing to 4-bit processing exponentially reduces hardware resources for state-matching and improves resource utilization. This, in turn, brings the opportunity to have a denser design, and be able to utilize more memory columns to process multiple symbols per cycle with a linear increase in state-matching resources.

Impala proposes an in-SRAM solution and introduces three-fold area, throughput, and energy benefits at the expense of increased offline compilation time. These three-fold efficiencies are obtained from (1) an

architectural contribution that utilizes *shorter-and-parallel* SRAM-subarrays instead of *longer-and-serial* subarrays, and (2) an algorithmic contribution which efficiently transforms an automaton and maps it to Impala's resources. To the best of our knowledge, this is the first work that observes state-matching inefficiency in memory-centric accelerators and proposes an algorithm/architecture co-design for *multi-stride* automata processing for in-situ computations.

Our empirical evaluations on a wide range of automata benchmarks reveal that Impala has on average 2.7× (up to 3.7×) higher throughput per unit area and 1.22× lower power consumption than Cache Automaton, which is the best performing prior work.

1.1.4 Enabling Low-Overhead Reporting Architecture in Memory-Centric Pattern Matching Accelerators

Prior memory-centric pattern matching solutions have either neglected the real cost of providing a reporting architecture [30, 1] or incur a significant cost when it is considered accurately. For example, the reporting architecture in the Automata Processor [61] has 40% area overhead and up to 46× performance overhead over the ideal case due to buffer flush overhead. Other reporting solutions [62] suffer from congested routing, frequent stalling, and lack of control on the hardware for efficient report analysis, and all these make the realistic in-memory pattern processing very inefficient and hard to scale-up.

To address these issues, we propose Sunder [67], a simple, compact, reconfigurable, and easy-to-implement reporting architecture by re-purpusing SRAM subarrays. Sunder leverages from Impala work, and processes reconfigure number of nibbles (4-bit symbols) in parallel instead of one 8-bit symbol. The key insight of our work is that transforming 8-bit processing to 4-bit processing reduces hardware resources for state-matching exponentially and introduces higher resource utilization. This, in turn, brings the opportunity to have a denser design and be able to utilize memory for reporting data.

Sunder introduces localized report buffers distributed across the chip close to where data (i.e., report states) are located. This approach helps Sunder to share many of the report buffer peripherals with the matching stage, and thus, incurring a negligible hardware overhead (less than 2%). In addition, our solution provides a simple mechanism to easily summarize the reporting data, and selectively read only the portion of the data that the application requires at run-time. We hope that our complete solution for in-memory pattern processing helps to realize its adoption in the industry.

1.1.5 Summary

This dissertation work focuses on developing memory-centric hardware accelerators and the associated software stack to accelerate complex pattern recognition/processing [1, 2, 56, 4, 66, 68, 27, 4, 69, 66, 70, 56, 39], the combination of which forms a hardware/software co-design that enables high-performance and energy-efficient complex pattern processing.

We present an open-source automata processing simulator and a compiler that simulates, minimizes, compresses, and transforms a set of automata to different symbol sizes, and efficiently maps them to the hardware resources in memory-centric architectures, such as FPGAs, Cache Automaton [30], eAP [4], etc. We then present *Grapefruit*, the first open-source, full-stack, comprehensive, and scalable framework for automata processing on FPGAs. Grapefruit provides an extensive set of compiler optimizations, hardware optimizations, and design parameters for design-space exploration on a wide range of emerging applications. We then introduce Impala, a software/hardware methodology for multi-stride in-memory automata processing by repurposing SRAM subarrays. Impala proposes three-fold area, throughput, and energy benefits at the expense of increased offline compilation time. We finally present Sunder, a simple, compact, reconfigurable, and easy-to-implement reporting architecture for memory-centric pattern matching solutions.

This set of works provide solutions to enable performance, power, and area improvements in memory-centric pattern matching accelerators over prior work, which confirms our hypothesis.

1.2 Overview of Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2: Background introduces automata processing, discusses automata-related theory, prior approaches and architectures for automata processing.

Chapter 3: Automata Simulation and Transformation presents an open-source automata processing simulator and a compiler that simulates, minimizes, compresses, and transforms automata.

Chapter 4: Multi-Symbol Processing on FPGAs presents the first open-source, full-stack, comprehensive, and scalable framework for automata processing on FPGAs.

Chapter 5: Algorithm/Architecture Methodology for In-Memory Multi-Stride Pattern Matching introduces a software/hardware methodology for multi-stride in-memory automata processing by re-purposing SRAM subarrays.

Chapter 6: Enabling Low-Overhead Reporting Architecture in Memory-Centric Pattern Matching Accelerators presents a simple, compact, reconfigurable, and easy-to-implement reporting architecture for memory-centric pattern matching solutions.

Chapter 7: Conclusions summarizes the dissertation and discusses the implications of this work and potential future directions of research.

Chapter 2

Background

2.1 Finite Automata

A finite automaton (FA) is a finite state machine (FSM) that accepts or rejects strings of symbols.

An FA is a mathematical model of computing and is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols, Δ is a transition function, q_0 are initial states, and F is a set of final or accepting states. An automaton has one or more start states that initiate computation, and one or more accept states that report a match. The transition function determines the next states using the current active states and the input symbol just read. If an input symbol causes the automata to enter into an accept state, the current position of the input symbol is reported.

A deterministic finite automaton (DFA) allows only one transition per input symbol. A non-deterministic finite automaton (NFA) has the ability to be in several states at once, meaning that transitions from a state on an input symbol can be to any set of states. DFAs and NFAs have equal computational power and can be converted to each other. However, a DFA can have exponentially more states that an equivalent NFA, which greatly increases the memory footprint. On the other hand, an NFA can have many parallel transitions, which is bounded by the limited memory bandwidth in von-neumann architectures.

2.2 Automata Processing on von Neumann Architectures

On CPUs, NFAs and DFAs are represented by tables indicating each state's successor state(s) upon a rule match. DFAs are often the basis for implementing automata on CPUs because they have predictable memory bandwidth requirements; while an NFA may have many active states and may require many state lookups to process a single input symbol (potentially leading to a very large memory bandwidth requirement), a DFA

requires just one. On the other hand, DFA tables are often too large to fit in the processor caches, because DFAs often suffer an exponential increase in the number of states relative to NFAs.

There are several efforts for high-speed regex processing on the CPUs and GPUs [71, 72, 73]. iNFAnt [73] is a parallel engine for regular expressions on GPUs with the support for multi-striding (processing multiple input bytes in each step). Graphics processing units (GPUs) provide a large number of parallel resources, which can help in hiding the DRAM access latency. However, highly-random access patterns in automata processing exhibit poor memory locality and increase branch divergence and need for synchronization [44].

Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [44]. Therefore, even high-throughput off-the-shelf von Neumann architectures struggle to meet today's big-data and streaming line-rate pattern processing requirements.

2.3 Automata Processing on Memory-Centric Architectures

To cope with the memory wall problem in conventional von Neumann architectures, in-memory automata processing hardware accelerators have been proposed and have shown several orders of magnitude speedup compared to CPUs and GPUs [74].

The Automata Processor (AP) [55] and Cache Automata (CA) [30] are two reconfigurable in-memory solutions, both directly implementing NFAs in memory. They can place-and-route automata states in a reconfigurable fabric, eliminating the need to access memory in von Neumann architectures. They also exploit the inherent bit-level parallelism of memory to support many parallel transitions in one cycle. The AP provides a DRAM-based dedicated automata processing chip, and Cache Automata proposes an on-chip solution by re-purposing a portion of the last-level cache for automata processing.

2.3.1 Automata Processor

Micron's Automata Processor (AP) [55] is an in-memory, non-von Neumann processor architecture that computes non-deterministic finite state automata (NFAs) natively in hardware. The AP allows a programmer to create NFAs and also provides a stream of input symbols to be computed on the NFAs in parallel. This is a fundamental departure from the sequential instruction/data addressing of von Neumann architectures. A benchmark repository for automata-based applications is presented in [75].

The AP re-purposes DRAM arrays for the state-matching and proposes a hierarchical FPGA-style programmable interconnect design. Each AP chip consists of two disjoint half-cores. Each half-core has 96 blocks. Each block provides 256 STEs and thus each AP chip supports 48K STEs Micron's current-generation

AP-D480 boards use AP chips built on 50nm DRAM technology, running at an input symbol (8-bit) rate of 133 MHz.

Input and Output The AP takes input streams of 8-bit symbols. The double-buffer strategy for both input and output of the AP chip enables an implicit data transfer/processing overlap. Any type of element on the AP chip can be configured as a reporting element (i.e., accepting state); one reporting element generates a one-bit signal when the element matches the input symbol. If any reporting element reports on a particular cycle, the chip will generate an output vector for all the reporting elements If too many output vectors are generated, the output buffer can fill up and stall the chip. Thus, minimizing output vectors, and hence the frequency at which reporting events can happen, is an important consideration for performance optimization. To address this, we will use the structures that wait until a special end-of-input symbol is seen to generate all of its reports in the same clock cycle.

Programming and Configuration Automata Network Markup Language (ANML), an XML-like language for describing automata networks, is the most basic way to program AP chip. ANML describes the properties of each element and how they connect to each other. The Micron's AP SDK also provides C, Java and Python interfaces to describe automata networks, create input streams, parse the output and manage computational tasks on the AP board.

Placing automata onto the AP fabric involves three stages: placement and routing compilation (PRC), routing configuration and STE symbol-set configuration. In the PRC stage, the AP compiler deduces the best element layout and generates a binary of the automata network. Depending on the complexity and the scale of the automata design, PRC takes several seconds to tens of minutes. Macros or templates can be precompiled and composed later. This shortens PRC time because only a small macro needs to be processed for PRC, and then the board can be tiled with as many of these macros as fit.

Routing configuration/reconfiguration programs the connections, and needs about 5 ms for a whole AP board. The symbol set configuration/reconfiguration writes the matching rules and initial active states for the STEs and takes 45 ms for whole board. A pre-compiled automaton only needs the last two steps. If only STE states change, only the last step is required.

2.3.2 Cache Automaton

Recently, Subramaniyan et al. [30] proposed an in-SRAM automata processing accelerator, Cache Automaton (CA), by re-purposing last-level cache for the state-matching and using 8T SRAM cells for the interconnect. The state-matching phase is based on the AP model. The crossbar interconnect adds 8T SRAM memory arrays and a 2-level hierarchical switch topology with local switches providing intra-partition connectivity

and global switches providing sparse inter-partition connectivity. CA uses a full-crossbar topology for the interconnect to support full connectivity in an automaton. The design has a better state density and throughput compared to the Automata Processor.

To better understand the architecture of memory-centric models, the following example discusses a simplified two-level pipeline architecture of automata processing used in the AP and CA.

2.3.3 Working Example on In-Memory Automata Accelerators

Homogeneous Automaton: The existing in-memory automata processing architectures use the homogeneous automaton representation in their execution model (same as ANML representation in [55]). In a homogeneous automaton, all transitions entering a state must happen on the same input symbol [76]. This provides a nice property that aligns well with a hardware implementation that finds matching states in one clock cycle and allows a label-independent interconnect. Following [55], we call this element that both represents a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE).

Figure 2.1 (a) shows an example on classic NFA and its equivalent homogeneous representation. Both automata in this example accept the language $(A|C)^*(C|T)(G)^+$. The alphabets are $\{A, T, C, G\}$. In the classic representation, the start state is q_0 and accepting state is q_3. In the homogeneous one, we label each STE from STE_0 to STE_3, so starting states are STE_0, $STE1$, and $STE2$, and the accepting state is STE_3. In all the architectures analyzed in this paper, any states can be starting states without incurring any extra overhead.

In-memory Automata Processing Here we presents a simplified two-level pipeline architecture for single-symbol processing of common in-situ automata accelerators, such as CA and the AP. In Figure 2.1 (b), memory columns are configured based on the homogeneous example in Figure 2.1 (a) for STE_0-STE_3. Generally, automata processing involves two steps for each input symbol, *state match* and *state transition*. In the state match phase, the input symbol is decoded and the set of states whose rule or label matches that input symbol are detected through reading a row of memory (*match vector*). Then, the set of potentially matching states is combined with the *active state vector*, which indicates the set of states that are currently active and allowed to initiate state transitions; i.e., these two vectors are ANDed. In the state-transition phase, the potential next-cycle active states are determined for the current active states (*active state vector*) by propagating signals through the interconnect to update the active state vector for the next input symbol operation.

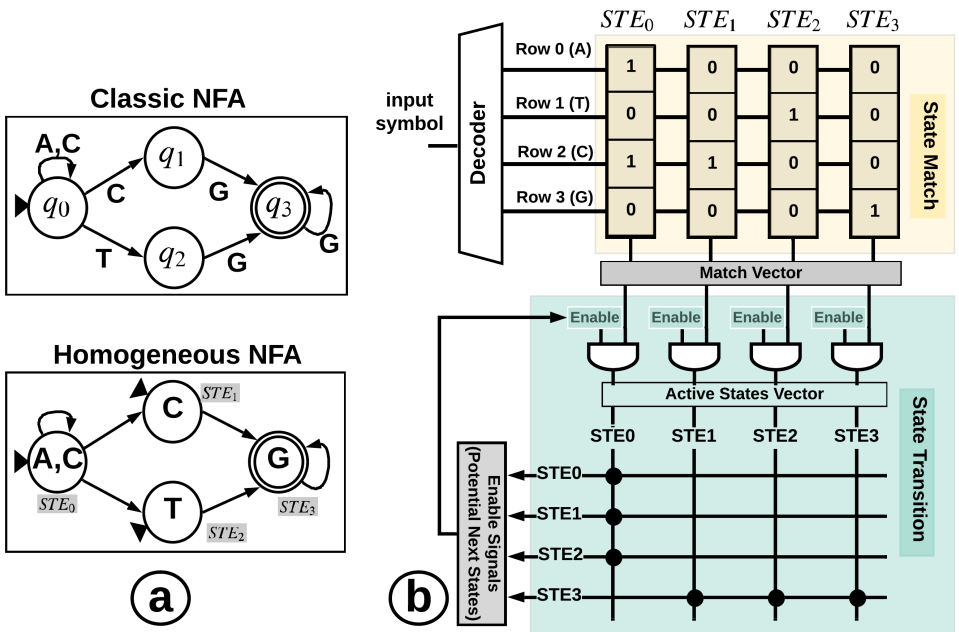

Figure 2.1: (a) Different NFA representation, (b) A simplified in-memory automata processing model

In the example, there are four memory rows, and each is mapped to one label (i.e., A, T, C, and G). Each state in the NFA example is mapped to one memory column, with '1' in the rows matching the label(s) assigned to those STEs. STE_0 matching symbols are A and C (Figure 2.1 (a)), and the corresponding positions have '1', i.e., in the first and third rows (Figure 2.1 (b)). Assume STE_0 is a current active state. The potential next cycle active states (or enable signals) are the states connected to STE_0, which are STE_0, STE_1, and STE_2 (the enable signals for STE_0, STE_1, and STE_2 are '1'). Specifically, if the input symbol is 'C,' then Row2 is read into the *match vector*. Bitwise AND on the *match vector* and *potential next states* (enable signal) determines STE_0 and STE_1 as the current active states.

When STE_3 in *active state vector* is '1', this means a report is generated and needs to be analyzed for final action (e.g., a malicious network packet is detected and needs to be dropped). The last chapter of this dissertation focuses on how to gather, handle, transfer, and analyze, and summarize the reporting information efficiently at low cost.

2.4 Automata Processing on FPGAs

FPGA solutions for accelerating regex and general automata processing are proposed [51, 65, 36, 52]. REAPR [51] is an FPGA-based implementation of an automata processing engine, and takes advantage of the one-to-one mapping between the spatial distribution of automata states and hardware resources such as lookup

tables and block RAM. REAPR can achieve approximately 2× to 4× higher clock speeds (250-500 MHz) than the AP, but lower than the estimated clock speed for CA. Large FPGA chips have approximately 2× more STE capacity than a single AP chip, but 3-6× less capacity than CA when utilizing 10-20MB of LLC. Moreover, power consumption of FPGA-based engines is higher compared to the AP and CA.

Yi-Hua et al. [36] propose a multi-symbols processing for NFAs on FPGA and utilizes both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicated the resources in each stride. This increases the critical path when increasing the stride value. Yamagaki [65] propose a multi-symbol state transitions solution using temporal transformation of NFAs to construct a new NFA with multi-symbol characters. This approach only utilizes LUTs, and does not scale very well due to a limited number of look-up tables in FPGAs. Besides, in their multi-striding method, the benefit of improved throughput decreases in more complex regexes (with more characters or highly connected automata) mostly due to routing congestion. Moreover, the recent FPGA-based automata processing solutions fail to map complex-to-route automata to the routing resources due to their logical interconnect complexity [52]

2.5 Automata Processing on ASIC

Several ASIC implementations have been proposed [37, 77, 78, 50] to accelerate pattern matching and automata processing. The Unified Automata Processor (UAP) [50] and HARE[37] have demonstrated line-rate automata processing and a regular matching expression on network intrusion detection and log processing benchmarks. HARE uses an array of parallel RISC processors to emulate the AHO-Corasick DFA representation of regular expression rule-sets. UAP can support many automata models using state transition packing and multi-stream processing at low area and power costs. This framework proposes new instructions to configure the transition states, perform finite automata transitions and synchronize the operation of parallel execution. UAP uses an array of parallel processors to execute automata transitions and can emulate any automata (not limited to Aho-Corasick). In general, while ASICs provide high line rates in principle, they are limited by the number of parallel matches and state transitions. HARE incurs high area and power costs when scanning more than 16 patterns, and UAP's line rate drops for large NFAs with many parallel active states.

In general, while ASICs provide high line rates in principle, they are limited by the number of parallel matches, state transitions, and shape of the automata. HARE implements DFA and has limitations on the regex shape, and also incurs high area and power costs when processing more than 16 patterns. UAP's line rate drops for large NFAs with many parallel active states. Therefore, they do not provide general and scalable solutions.

Chapter 3

Automata Simulation and Transformation

3.1 Motivation: The Need for Automata Simulator

Automata processing research focuses mainly on two dimensions: (1) application development, where obvious applications (e.g., regular expressions) or non-obvious applications (e.g., frequent subtree mining [27]) are mapped to automata computation and processed on automata processing hardware, and (2) architectural development. Unfortunately, the available tools for both research areas are not comprehensive, parametrizable, and easy-to-modify. Different components are scattered among multiple tools, which make it even harder to explore new research ideas.

In the application layer, researchers want to have access to a tool that gives them a rich and descriptive interface to define an automaton and examine/debug them with an input stream. Providing extensive observability features of the internal state is an essential need in this aspect. Supporting different accelerators as the target hardware platform is another important feature that makes the programmers life easier by directly generating the final code/configuration for deployment on the accelerator without the need to migrating design across different tools.

Instead, architectural researchers are interested in exploring different designs to pick high-throughput and efficient designs while satisfying the underlying conditions such as power budget and area constraints. One crucial design parameter in the architectural domain of automata processing is the symbol's bit-width (or alphabet size), which is mainly determined by the application. For example, DNA sequencing has only four characters in its alphabet (A, T, C, and G), which can be represented with 2-bit symbols. Conversely, in

frequent sequence mining, the alphabet size is in the order of thousands/millions/billions (depends on the specific application) of independent symbols, which requires a much longer bitwidth. This parameter sets the bitrate processing for a machine that consumes one symbol per cycle.

Hardware researchers are more interested in using a tool that gives them enough flexibility to explore design space parameters comprehensively. In hardware, the implication of symbol bitwidth is also essential as it directly impacts the throughput and hardware cost. Although, the symbol bitwidth is initially determined by application itself, having a set of transformations to change it (while keeping the automaton functionality intact), gives the hardware designer the chance to pick the sweet spot for a specific hardware platform. This sweet spot may vary significantly across different hardware domains such as in-memory processing, FPGA, or Von Neumann machines.

Interconnect design (to model state-transition) for automata processing is another dimension that affects area, power, delay, and throughput in a system. A highly-connected interconnect design (such as full mesh) in hardware leads to a significant area overhead due to the required amount of wiring and switches to be physically realized, with the benefit of easy placement and routing. On the other hand, providing limited connectivity in hardware is area-efficient but hard for state and transition placement and prone to state under-utilization due to routing congestion, especially for a highly connected automaton.

Unfortunately, the most recent tool, VASim [79], developed for automata processing, has been designed only for 8-bit alphabets (256 symbols) and does not provide any freedom to modify the bitwidth. It also does not have any built-in function procedure to handle the bitwidth transformations. In addition, the design space exploration for the interconnect does not exist natively in this tool. There is a side tool focusing on automata routing [80], which utilizes the available FPGA routing tools. However, it is mostly suitable for FPGA-style interconnect pattern. In addition, defining and exploring new interconnect models is not a straight-forward task and needs tedious effort to define the interconnect model. In addition, as this loop is closed in different tools, exploring different interconnect design becomes slow which has negative impact on the covered region of design space.

Further, HDL generation in VASim is not in release mode. Automata generated with VASim needs to be ported by automata descriptor formats such as ANML to other tools such as REAPR [81] to generate HDL code targeting FPGAs, which again works with fixed to 8-bit symbols and does not support other variable bitwidth processing or symbol striding. Finally, we found it hard to modify the VASim to support these functionalities as 8-bit processing was an fundamental assumption in its development process. Therefore, we design our own automata simulator APSim from scratch. We prioritized flexibility and extendability while developing this tool to provides an easy to understand interface for other researchers as well to modify the code and explore new features.

3.2 APSim - A Cycle Accurate Automata Simulator: Current and Proposed Contributions

We present an open-source automata processing simulator and a compiler that simulates, minimizes, compresses, and transforms a set of automata to different symbol sizes, and efficiently maps them to the hardware resources in memory-centric architectures, such as FPGAs, Cache Automaton [30], eAP [4], etc. It receives the input automata based on the standard representations such as ANML and provide a extensive set of transformations/minimizations with its enrich APIs.

Figure 3.1 shows different components of APSim. APSim advantages and contributions over prior automata simulators such VASim are listed below.

- APSim provides a parameterizable environment to experiment with bitwidth sizes on various platforms. A set of bitwise and symbol-wise automata transformation and minimization algorithms are developed to enable these explorations. In the bitwise approach, the input automaton is first flattened to its binary representation (one bit per edge) and traversed by collecting every possible path with the length equal to the target bitwidth. The symbol-wise approach basically does the same traversing as the bitwise approach, but it collects symbols as primitives compared to the bitwise (See details in Section 3.3 and use cases in Chapters 5, 4, and 6).

- APSim addresses the state and transition placement in an automaton by converting it to the subgraph isomorphism problem, and applies the genetic algorithm approach to maps states/transitions to their physical locations. Considering the input automaton as a subgraph S and the supported connectivity model by hardware as a graph G, our solution finds a subgraph in G that is equivalent to S. Subgraph isomorphism is known to be NP-Complete, and conventional solving with exponential timing complexity fails for large scale graphs. APSim uses a genetic algorithm approach to solve this problem with reasonable time and complexity. APSim's simple interface for defining the connectivity pattern of the hardware makes the researcher's life easy to study the effectiveness of different schemes for interconnect support in hardware for automata processing (See details in Chapter 5, Section 5.3).

- Unlike VASim, APSim uses a well-known and actively maintained python graph processing package as its main building block to benefit from its reliability, speed, and massive documentation with examples for further development by other collaborators. In APSim, an automaton is considered as a directed multigraph where the nodes or edges can carry symbol data and NFA metadata. All the automata related algorithms have been implemented on top of this concept. For example, an strided automaton

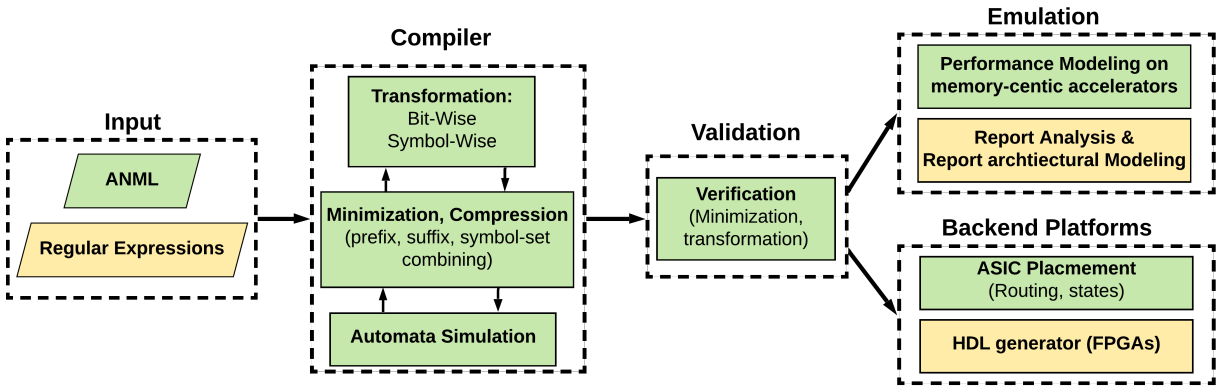

Figure 3.1: APSim Components.

that consumes two symbols per cycle can be achieved by finding every path of length two in the input automaton and replace it with a single edge in the strided automaton.

- An HDL generator targeting FPGAs has been embedded in APSim to produce all the required Verilog files and synthesizing scripts (for Xilinx FPGAs). Having this functionality in APSim makes the hardware designers' life much more comfortable as they can easily access to every data embedded in the automaton. VASim uses REAPR [51] and input the automata with a standard format to generate HDL code, which makes data exchange cumbersome (See details in Chapter 4).

- Stream-based transformation verification is also supported in APSim to check the correctness of transformations. Users can develop their own transformations and easily compare report status of the new automaton with the original automaton by streaming the same input to both of them and check the equivalence of report states status per cycle. This functionality also works with strided automata where a multi strided automaton can be compared against its original representation.

3.3 Automata Transformation

In memory-based solutions, symbols are encoded in memory columns, such that each symbol activates a different row of memory. This tends to reinforce designs based on 8-bit symbols because 256 (2^8) is a fairly conventional subarray height. However, this can be extremely inefficient, especially when the application alphabet (symbol-set) size is very small, and the number of rows in a subarray is more than required. For example, in genomics, the alphabets are A, T, C, and G, and a 2-bit automata organization with only 2^2 rows is enough to perform the string matching. This is 64× smaller than what existing spatial architectures provide!

On the contrary, the 8-bit symbol processing architectures can limit the generality of the architecture for applications that have more than 256 symbols. For example, in sequential pattern mining [24], the input database can be quite large, such as market basket analysis datasets from Amazon, and the number of unique items (or symbols) can be very large (on the order of 2^{20} or more). In formal verification problems, the symbols map to the events, and thus, the automata symbol-set size can be extremely large [82, 83]. However, due to delay, power issues, and signal integrity, it is impractical to change the hardware to support 2^{20} rows in each memory subarray. Moreover, simply daisy-chaining multiple states to support larger alphabets results in false report generation. Therefore, an efficient and accurate symbol-size transformation technique is required.

Another problem with the existing 8-bit approach for spatial accelerators is that, if the memory subarray size of the underlying memory technology changes, then there is a need to make sure that the application symbol-set size is still compatible with the memory architecture. For example, Cache Automaton (CA) [30] re-purposes caches in conventional processors for automata processing. If the number of rows in the subarrays of a cache structure changes, then the automata structure and input bitwidth consumption need to be changed for correct functionality and full hardware utilization.

To address these issues, we develop FlexAmata (and embed it in APSim), a compiler solution that decouples applications (with any alphabet size) from the details of the memory architecture (e.g., number of rows per subarray). FlexAmata acts as an adjustable wrench and transforms an arbitrary $m-bit$ processing automaton to its equivalent $n-bit$ processing unit, where n can be larger or smaller than m, depending on the target architecture. FlexAmata offers arbitrary bitwidth processing, thus improving efficiency for small alphabets, enabling hardware acceleration for large alphabets that were nearly impossible to process efficiently up till now, and maintains application compatibility with the future automata hardware accelerators. FlexAmata, therefore, improves small-symbol-set efficiency and provides large-symbol-set compatibility even for conventional in-memory solutions such as CA. Thanks to our fine-grain, bit-level optimizations in FlexAmata, state and transition overhead of a transformed automaton is reasonably low.

3.3.1 Transformation Methodology

FlexAmata transforms an m-*bit* automaton A to an equivalent n-*bit* automaton B, where n can be larger or smaller than m. This transformation is done in two steps; (1) converting A to a bit-level representation (A_b), and (2) generating automaton B by transforming A_b to process n-*bit* in each cycle. To generate the n-*bit* automaton, we find all the unique paths of size n in A_b and replace each of them as a single edge (or equivalently transition rule) in B. The algorithm performs bit-level minimization on the automata and

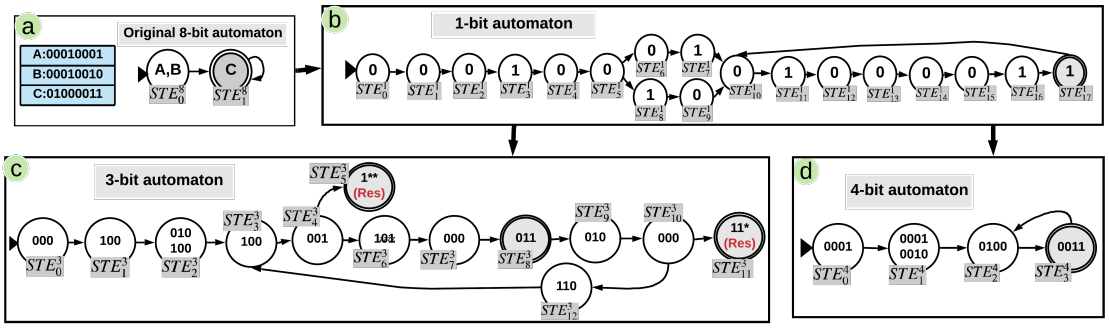

Figure 3.2: An 8-bit automaton (a) is converted to the minimized *1-bit* automaton (b). The *3-bit* (c) and *4-bit* (d) automata are generated from the *1-bit* automaton.

merges the states and transitions in binary paths when applicable. Finally, automaton B is converted to its homogeneous representation to properly be configured on an in-memory platform or FPGAs.

In Figure 3.2, we explain how an 8-bit automaton is transformed into 3-bit and 4-bit automata. In the notation STE_x^y, x is state index and y is the bitwidth size. The original homogeneous automaton (a) has two states and accepts language $(A|B)C^+$. Using FlexAmata, we generate binary automata (b) and minimize the states when possible. For example, the first 6 bits of symbols A and B can be merged. Then, 3-bit (c) and 4-bit (d) are generated from the bit-automaton.

In the 3-bit automaton, STE_0^3 is an start state and STE_5^3, STE_8^3, and STE_{11}^3 are final states. Each state processes one or more 3-bit symbols. STE_{16}^1 in 1-bit-automata is equivalent to reaching the state STE_4^3 in the 3-bit automaton. STE_{17}^1 is a report state, and there is a loop back to the state STE_{10}^1. Assume STE_{16}^1 in 1-bit-automaton is an active state, and the next input character is "1", then it should generate a report. Equivalently in the 3-bit automaton, STE_4^3 is an active state, and because the next input is a 3-bit chunk, then, "1**" should generate a report. In order to address this unalignment in the bitwidths, we generate residual states (Res) to report when a match happens in the middle of a multi-bit input. STE_5^3 is a residual state that reports when the matching happens in the first bit of the 3-bit input. The residual states are used to locate match occurrence in the input stream accurately.

In 4-bit automaton, no residual state is needed as 8 (from original 8-bit automaton) is divisible by 4, and this property avoids mis-alignment in the input symbols. In order to get the correct functionality, we always make sure the input size is divisible by the bitwidth size by padding the input stream.

3.3.2 Transformation Soundness

Correctness of an m-bit to an n-bit automaton can theoretically be proven using contradiction in two parts. First, the equivalence of the m-bit automaton to its 1-bit automaton, and the equivalence of the 1-bit

automaton to the n-bit automaton. First, we show that for every edge in the m-bit automaton, there is one and only one unique path of length m in the 1-bit automaton (with no common edge between paths). Second, we show that for every edge in the n-bit automaton, again, there is a unique path with length n in the 1-bit automaton. Using contradiction, assuming there is an edge in the m-bit automaton that can not be mapped to a path of length m in the 1-bit automatons. However, this is impossible as the algorithm process every edge in m-bit automaton in breadth-first search (BFS) manner and for each edge, creates one path in the 1-bit version (all middle states in the 1-bit automaton are created uniquely and the destination state is added if the destination node has not been previously visited in the m-bit automaton). Similarly, it is not possible to find a path of length m in the 1-bit automaton that can not be mapped into an edge in the m-bit automaton.

In terms of implementation validation, we re-transform a reshaped automaton to the original bitwidth and check for equivalence with the original automaton. In addition, we have verified the correctness of our implementation by streaming input to both automata (original automaton and transformed version) and compared the report information between them. In both analyses, we have not found any inconsistency.

3.3.3 Time Complexity

The offline compilation process for converting an m-bit automaton to its equivalent n-bit automaton has two main steps; (1) converting the m-bit automaton to a 1-bit automaton: assuming the input automaton has a total of *'s'* matching symbols across all the states (m bits each). For every symbol, we need to find its binary representation and add a new chain of states (with length m) for each symbol in the binary automaton. Therefore, the time complexity for this stage is $O(m \times s)$.

(2) Converting a binary automaton with t states to an n-bit automaton: we use dynamic programming with backtracking to solve this problem efficiently. Our converter implementation starts from the start states in the binary automaton and moves forward towards the report states. When a new state is reached, the path from all its parents (ancestors with a maximum distance of n) is added to a dictionary to be reused later again when necessary. In the worst-case, it needs a table of n entries for each of the t states. Each state with n entries keeps the reachable states (from itself as source) with different distance ranging from *1* to n. To calculate the missing entries in the table, we need to iterate through direct neighbors of nodes and reuse data from the existing entries of the table (if the neighbor has an entry in the table) and combine them or recursively go through their neighbors. For a binary automaton with average out-degree of e, assuming combining entries takes $O(1)$ times, the timing complexity is $O(t \times e \times n)$ where $t \times n$ is the table size, and e is the number of times that a combining operation needs to be applied.

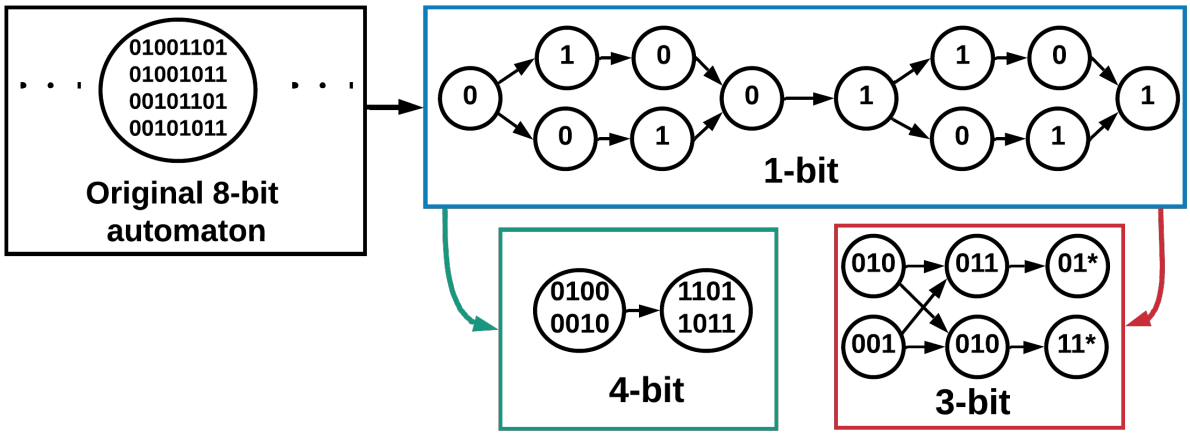

Figure 3.3: State and transition overhead is less in divisible bitwidths (4-bit) than non-divisible bitwidths (3-bit).

3.3.4 Non-divisible bitwidths effect

As Figure 3.2 illustrates, the 3-bit automaton has more state and transition overhead than a 4-bit automaton. From our experiments, we observed that when re-shaping an n-bit automaton to an m-bit automaton, the overhead would be minimum if either m mod n = 0 or n mod m = 0. Figure 3.3 explains the reason with an original state with four 8-bit symbols. First, a 1-bit automaton is generated. Then, 3-bit and 4-bit automata are generated from the 1-bit form. In 3-bit, because 8 mod 3 ! = 0, it needs to generate more states and transitions to consider for combinations of paths when a jump is needed. However, the 4-bit design has a relatively very low state and transition overhead. This observation is later used to find the best bitwidth processing size for spatial accelerators.

3.4 APSim Contribution to the Research

APSim has enabled several architectural research for automata interconnect design [3, 56], universal mapping from application with any alphabet size to hardware accelerator [66], multi-symbol processing exploration on in-memory architectures [69], and multi-symbol processing on FPGAs [2].

Chapter 4

Open-Source, Full-Stack, and Customizable Automata Processing Engine on FPGAs

Researchers are increasingly exploiting hardware accelerators to meet demanding real-time requirements as performance growth in conventional processors is slowing. In particular, several FPGA-based regex implementations for single-stride [51, 35, 84, 52, 53] and multi-stride [36, 65, 64, 63] automata processing have been proposed to improve the performance of regex matching. These solutions provide a reconfigurable substrate to lay out the rules in hardware by placing-and-routing automata states and connections onto a pool of hardware units in logic- or memory-based fabrics. This allows a large number of automata to be executed in parallel, up to the hardware capacity, in contrast to von Neumann architectures such as CPUs that must handle one rule at a time in each core. Most of the current FPGA solutions are inspired by network applications such as Network Intrusion Detection Systems (NIDS). However, patterns in other applications can have different structure and behavior, e.g., higher fan-outs, and this makes it difficult for NIDS-based FPGA solutions to map other automata to FPGA resources efficiently [52, 44, 45].

To enable architectural research, trade-off analysis, and performance comparison with other architectures on the growing range of applications, an open-source, full-stack, parameterized, optimized, scalable, easy-to-use, and easy-to-verify framework for automata processing is required. REAPR [51] is a reconfigurable engine for automata processing, and generates FPGA configurations that operate very similarly to the Micron Automata Processor (AP) style [55] processing model. The RTL generated from the automata graph is a

flat design, which causes a very long compilation time. Due to this flat-design approach, this solution is not scalable and the synthesizer fails to generate RTL for larger designs. Moreover, REAPR only generates the matching kernel and does not provide a full-stack solution or even the automata reporting architecture.

Bo et al. [85] extend REAPR and provide an end-to-end solution on FPGAs using SDAccel for the I/O. However, their I/O design has two issues. First, the input stream should be segmented into limited-size chunks. Second, the reporting structure is very simple; whenever a state generates a report, a long vector (the size of the vector is equal to the number of total reporting states), mostly filled with zeroes, is read and sent to the host. This reporting architecture may become a bottleneck for applications with frequent but sparse reporting, which is a common reporting behavior [62]. Casias et al. [6] also extended REAPR and proposed a tree-shaped hierarchical pipeline architecture. However, their solution generates the HDL code for only the kernel and does not provide a full-stack solution (i.e., broadcasting input symbols to the logic elements and getting the reporting data out the FPGA chip). Furthermore, their source code is not publicly available.

Researchers are interested in using a tool that gives them the flexibility to explore design space parameters comprehensively. For example, in automata processing, symbol size impacts the throughput and hardware cost [66], and none of the prior tools provide support for that. Similarly, reporting architecture can be a performance bottleneck that can reduce the throughput significantly [62] (up to 46X stall overhead in the Micron AP). To improve the performance, Liu et el. [57] propose a hybrid automata processing approach by splitting states between CPU and an automata processing accelerator, and this incurs higher reporting rate on the accelerator. Therefore, an efficient reporting architecture is more critical for the end-to-end performance. In this paper, we present *Grapefruit* (**G**eneral and **R**econfigurable **A**utomata **P**roc**E**ssing **FR**amework **U**sing **I**ntegrated Reporting and In**T**erconnect). We prioritize flexibility, extensibility, and scalability while developing this tool to provide an easy-to-understand interface and easy-to-modify code for other researchers to explore new features and design parameters.

In summary, this paper makes the following contributions:

- We present Grapefruit, **the first open-source, full-stack, comprehensive, and scalable framework for automata processing on FPGAs**[1]. Grapefruit provides an extensive set of compiler optimizations, hardware optimizations, and design parameters for design-space exploration on a wide range of emerging applications.

- We present an optimized pipeline architecture, an adaptive priority-based reporting architecture, and an interconnect model to address the issues in prior tools and support scalability to large numbers of

[1] https://github.com/gr-rahimi/APSim (check temp_scripts/FCCM folder)

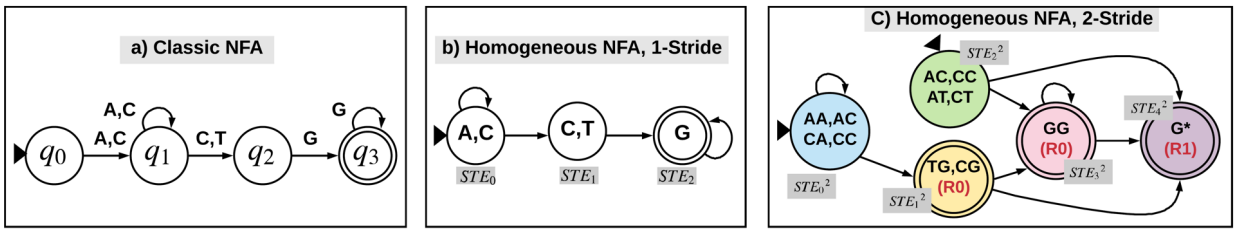

Figure 4.1: (a) Classic NFA, (b) Homogeneous NFA, (c) Equivalent 2-stride automata.

rules/automata. We also investigate LUT-based, BRAM-based, and the combination of both in the design space.

- We present an integrated back-end compiler, with a rich and descriptive interface, to define an automaton and to perform cycle-accurate automata simulation, automata transformation, and automata minimization. An important feature provided in our framework is the support for multi-symbol (multi-stride) processing with variable symbol size, and this directly impacts throughput and hardware cost.

- We perform thorough performance analysis with different optimizations and parameters on a wide range of automata applications on a Xilinx Virtex UltraScale+. Our results confirm that we are achieving 9%-80% higher frequency in a single-stride solution than prior works that are not fully end-to-end (including reporting and I/O) and 3.4× higher throughput in a multi-stride solution than a single-stride solution.

4.1 Background

We use the homogeneous automaton representation in our execution models. In a homogeneous automaton, all transitions entering a state must happen on the same input symbol [76]. This provides a nice property that aligns well with a hardware implementation that finds matching states in one clock cycle and allows a label-independent interconnect. Following [55] and [4], we call this element that represents both a state and performs input-symbol matching in homogeneous automata a *State Transition Element* (STE) (see Chapter 2 for more detail).

Figure 4.1 (a) shows an example of a classic NFA and its equivalent homogeneous representation (b). Both automata in this example accept the language $(A|C)^+(C|T)(G)^+$. The alphabets are $\{A, T, C, G\}$. In the classic representation, the start state is q_0, and accepting state is q_3. In the homogeneous one, we label each STE from STE_0 to STE_3, so starting state is STE_0, and the accepting state is STE_2. Figure 4.1 (c) strides

the NFA in (b) and processes two symbols per cycle, and this provides higher throughput. Details of striding is explained in Section 4.3.2.

4.2 Related Work

A number of multi-stride automata processing engines have been proposed on CPUs and GPUs [73, 86, 64, 87, 71, 47]. Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [44]. Moreover, multi-symbol processing causes more pressure on memory bandwidth, because more states and transitions need to be processed in each clock cycle.

FPGA implementations of regular expression matching are often inspired by networking applications [35, 88, 36, 84, 52, 51], which typically consist of many independent matching rules. These are often implemented using automata-based (NFA or DFA) computation. Hieu [84] proposes an accelerated automata processing on FPGA using the Aho-Corasick DFA model. DFAs can be easily mapped to BRAM, and reconfiguration in BRAM is cheap. However, a DFA model does not utilize the inherent bit-level parallelism in FPGAs and is better suited to memory-bound von Neumann architectures. Furthermore, DFAs are primarily beneficial for von Neumann architectures, because DFAs only have one active state at any point in time. But combining multiple independent automata or regex rules into a single DFA leads to a rapid blowup in the number of states, while keeping multiple independent automata obviates the main benefit of DFAs. NFAs are thus a better fit for a substrate with high parallelism such as FPGAs or the Micron AP. Karakchi et al. [52] present an overlay architecture for automata processing, which is inspired by the Micron AP architecture. Karakchi's architecture forces fan-out limitation and thus fails to map complex-to-route automata to the routing resources due to its logical interconnect complexity. *Our framework is optimized for general NFA processing, and it provides the user with a variety of parameter and flexibility to optimize the design not only for network regexes but for general and emerging automata applications.*

To improve the throughput of automata processing, some works have investigated multi-striding (multiple symbols per cycle) on FPGAs [36, 65, 64, 63]. Yang et al. [36] proposed a multi-symbol processing solution for regular expressions on FPGA which utilizes both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicates the resources in each stride. This increases the critical path when increasing the stride value. Yamagaki et al. [65] proposed a multi-symbol state transitions solution using a temporal transformation of NFAs to construct a new NFA with multi-symbol characters. This approach only utilizes LUTs and does not scale well due to the limited number of lookup tables in FPGAs. *None of these*

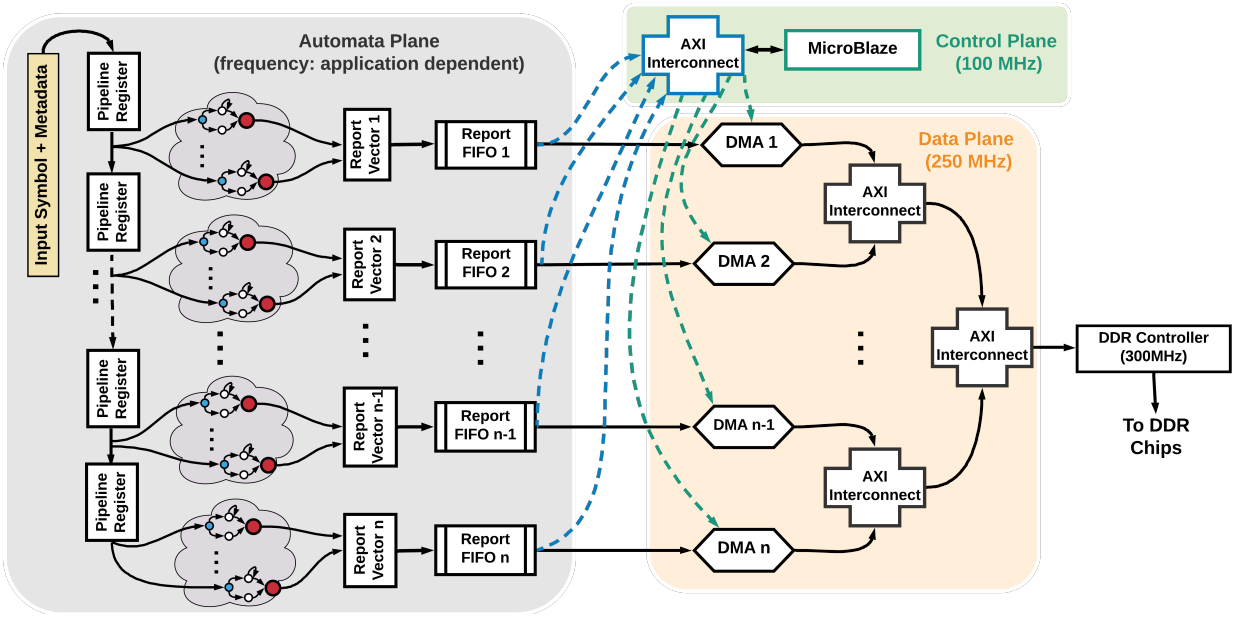

Figure 4.2: Automata processing architecture on FPGAs. The cloud-shaped entities show several connected components.

works provide an open-source tool. Our proposed framework provides a temporal multi-striding solution, and the user can choose to use the LUTs, BRAMs, or the combination of LUT/BRAM for design exploration.

Furthermore, alphabet compression techniques [87, 89, 90, 91, 92] may be employed to reduce memory requirements in CPU/GPU and FPGA-based solutions. Becchi et al. [89] propose to merge symbols with the same transition rules. This reduces the number of unique alphabets in an automaton. However, 8-bit hardware accelerators cannot benefit from the compression techniques if the reduced alphabet size requires fewer than 8-bit symbols. *Our compiler can be utilized to reshape the compressed automaton to 8-bit symbol processing while keeping the benefit of compression. This provides full hardware utilization on the target FPGA and at the same time, increases the processing rate and throughput.*

REAPR [51] is an FPGA implementation of a single-stride NFA processing engine, and takes advantage of the one-to-one mapping between the spatial distribution of automata states and hardware resources such as lookup tables and block RAMs. REAPR uses VASim [79] as its backend compiler. VASim is hard-coded to 8-bits (256 symbols) and does not provide any freedom to modify the bitwidth. It also does not have any built-in function procedure to handle bitwidth transformations. In addition, the design space exploration for the interconnect does not exist natively in this tool. Further, HDL generation in VASim is not in release mode. Automata generated with VASim needs to be ported by automata descriptor formats such as ANML to other tools such as REAPR [81] to generate HDL code targeting FPGAs, which again works with fixed to 8-bit symbols and does not support other variable bitwidth processing or symbol striding.

4.3 Grapefruit Framework

4.3.1 Architecture

This section describes an overview of the system design for an efficient automata processing on FPGAs using Figure 5.4 and explains some of the features and optimizations.

Pipeline design

NFAs for real-world applications are typically composed of many independent patterns, which manifest as separate *connected components (CCs)* with no transitions between them. Each CC usually has a few hundred states. All the CCs can thus be executed in parallel, independently of each other. Grapefruit uses a pipelining methodology to cluster the automata into smaller groups, each group with several CCs (Figure 5.4 - *Automata Plane*). The original automata are first partitioned to multiple sets of automata (the number of automata in each set can be defined by the user), and each set is implemented in the same pipeline stage. Each incoming input symbol first is matched against all the CCs in the first pipeline stage and then travels to the next stage, being processed by the next stage while the first stage processes the second symbol.

Our pipelining structure reduces synthesis time compared to REAPR's flat design [51] because the synthesizer only considers the automata in the same stage, as they share the same input signal. On the other hand, using pipelining reduces the report vector size to the total number of reporting states in CCs of the same stage.

Reporting architecture

The reporting states fire a signal when a match happens. Collecting the reporting data for automata processing is another dimension that affects the area, power, delay, and throughput in a system. In Grapefruit, all the report states in the same pipeline stage create a bit-vector (each vector index is assigned to one report state). If any of the report states gets activated (meaning there is at least one report in that cycle), a snapshot of the report vector is sent to the report buffer allocated for each stage separately. Report buffers are flushed by the *Data Plane* interconnect toward the DDR memory. The *Data Plane* consists of a hierarchy of AXI-4 interconnects running at 250 MHz, which provides a path between every reporting buffer and the memory controller. Each report buffer is equipped with a DMA (Direct Memory Access) module to generate the memory address for storing the report data.

Current results are generated for a Xilinx VCU118 evaluation board, which has two DDR4 memory channels running in 1200 MHz. Grapefruit uses one memory channel for the report data and the other channel for the input symbols. Each memory channel has 5 DRAM chips, each with 16-bit data-width (80

bits in total). However, the Xilinx's IP core for memory controller only uses a 64-bit interface when ECC functionality is disabled. Therefore, in total, each memory channel can provide up to 18.75 GBps bandwidth. Using the hierarchical topology for the interconnect helps to avoid the signal congestion by distributing the complexity across multiple interconnect modules and providing a high-frequency yet area-efficient interconnect design.

In addition, our hierarchical design helps make the system scalable for larger applications, especially when the number of report buffers increases significantly. Grapefruit gives the user the flexibility to specify the desired Data Plane interconnect as an abstract tree data structure and generates the necessary scripts in the backend to implement it in an FPGA. As each of the automata pipeline stages has different report firing rate (which depends on the automata itself and the input characteristics), it is necessary to assign the Data Plane bandwidth wisely to the stages with a higher report rate (and possibly more filled reporting buffer). In addition, a static policy is not suitable as the input symbol pattern may change at run time, and this can lead to dynamic change in the reporting rate of different stages. To solve this issue, the *Control Plane* (shown in Figure 5.4) is used, where a MicroBlaze processor monitors the capacity of buffers to detect the highly reporting stages. The code running in the MicroBlaze reads the buffer size of each pipeline stage in a loop and configures the DMAs of buffers that running low in capacity to flush them into the off-chip DDR memory. The Control Plane uses AXI-Lite standard (no burst transmission) as transactions are simple register-reads/writes (buffer capacity and DMA control registers). In addition, Grapefruit utilizes the on-chip memory as the main memory for the MicroBlaze to make sure its instruction/data traffic does not interfere with the Data Plane memory operations.

Mapping Automata to LUT Resources

This section discusses the mapping of an automaton (represented in *Automata Plane* in Figure5.4) to the LUT resource of the FPGA using an example. In Figure 4.3 (a), a homogeneous automaton is shown that processes two 8-bit symbols (16-bit) per cycle. STE_0^2 is the start state, and STE_1^2 and STE_2^2 are the reporting states. The states are color-coded to represent their equivalent units in the circuit shown in Figure 4.3 (b). Symbol matching is done entirely in LUTs based on the 16-bit symbols. Theoretically, FFs are equivalent to the states that may be activated in the following cycle. Once a state is activated, all of its children are considered as potential active states.

The input signals of the FFs come from an OR gate, which is the OR signal of all the states that have incoming transitions to that specific state. The states that have common parents can share their FFs and save resources. However, in theory, their corresponding states cannot be merged since they are not equivalent

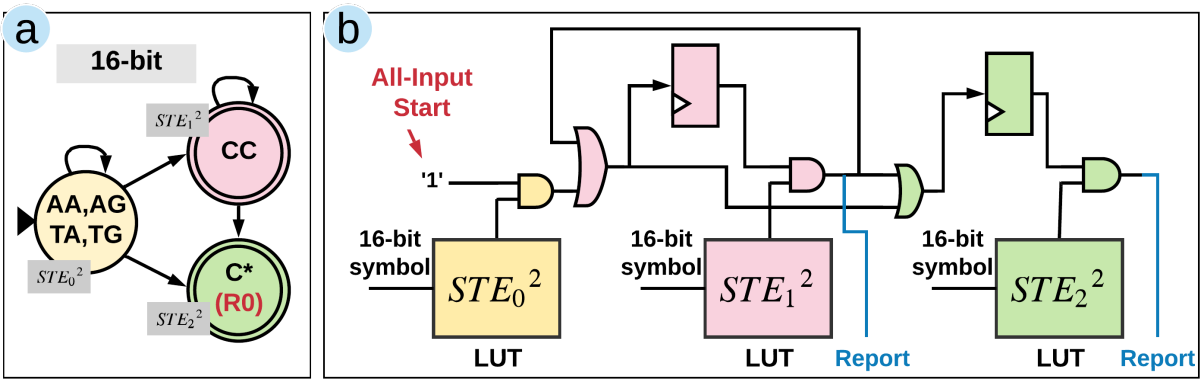

Figure 4.3: Automata mapping in LUT-based design.

states. The report signals of the final states are generated from the AND gate of matching signals and potential active states.

We observed that small bitwidths (smaller than 8) do not utilize FPGA LUT resources well. This is mainly because LUTs in Xilinx FPGAs can implement up to two functions with five inputs or one function with six inputs, and thus, 1, 2, and 4-bit symbols operate inefficiently on LUTs. On the other hand, processing more symbols per cycle leads to a more complex matching with many intervals from different states combined to a single state. This situation makes matching using 6-inputs LUTs inefficient in terms of resource usage and clock frequency (longer critical path), as LUTs need to be combined to implement bigger functions.

Mapping Automata to BRAM Resources

In Addition to LUTs, FPGAs have fast and area-efficient on-chip memory resources such as BlockRAMs (BRAMs) and UltraRAMs (available in newer Xilinx FPGAs). An alternative to LUT-based design is to utilize the on-chip memories to calculate the symbol matching part efficiently. Technically, the symbol matching is responsible for detecting all the states that match against the current symbol, and BRAMs can implement this logic by simply putting each unique matching condition in one column and store ‘1‘ if the matching condition matches the cell location, assuming the input symbol is used as a read address at runtime; otherwise ‘0‘ is stored. In other words, each column is one STE. Because each pipeline stage is processing different symbols, it is not possible to combine matching rules from different stages into one BRAM. To minimize the BRAM waste in cases where the number of unique matching rules in a stage is not enough to fill all the available memory columns, Grapefruit configures the BRAMs in their narrowest configurations with smallest column-count and most-minor-row-size bigger than 256 (36 columns and 512 rows in Xilinx BRAMs).

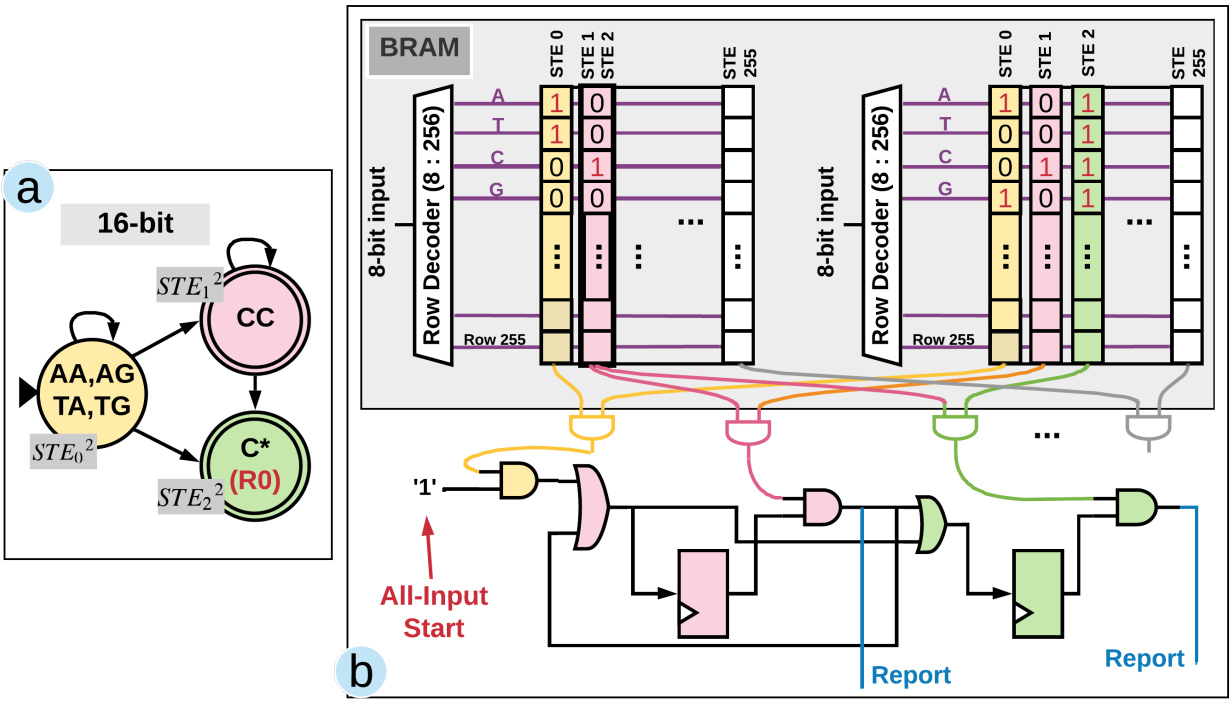

Figure 4.4: Automaton mapping in BRAM-based design.

Matching for higher stride values (processing multiple symbols per cycle) needs more bits to evaluate compared to the one symbol per cycle case. Xilinx BRAMs in their tallest configuration have 14 bits as the address input and one bit data output. This configuration is not even enough to implement the two symbols per cycle case (16-bit symbols). To solve this issue, Grapefruit combines multiple BRAMs (equivalent to the stride value) to implement multi-symbol conditions. Each BRAM is responsible for decoding 8 bits of the input symbol, and the final matching signal is calculated by applying a binary AND operation on the BRAMs output signals. For example, in Figure 4.3, the automaton on the left processes two symbols per cycle. Considering STE_1 as an example, it needs to compare the first and second 8-bit against 'C'. Two BRAMs (shown as two arrays in part b) are used for matching; the left array handles the first 8-bit, and the second array handles the second 8-bit. The character 'C' has been decoded in both these arrays in two separate columns (one column in each array), their output is ANDed, and its result is routed as the final match signal. In this design, columns with the same matching conditions can be merged to save the BRAM resources. For example, in Figure 4.3, both STE1 and STE2 compares the first 8 bits against 'C', so they can share the same column in the left array. However, the second 8-bit needs two separate columns, as their matching conditions are different ('C' versus *). Grapefruit looks for these common matching conditions in each array and tries to save BRAMs as much as possible by assigning the same column to equivalent matching considerations.

Addressing false positive issues for combined BRAMs: combining multiple BRAMs with an AND operation enables utilizing these on-chip memory resources for multi symbol matching, but it needs to be applied with careful consideration. Simply putting every 8-bit matching condition to one column of a BRAM array can potentially lead to a false-positive matching situation. For example, let us assume that one STE has a matching condition AB, CD for an automaton that processes two symbols per cycle. Putting matching symbols A, C of the first input symbol in one BRAM column and B, D in another column of a separate BRAM can mistakenly match against AD *or* CB. Generally, every possible combination of matching symbols in each dimension can also be matched, as we generate the final matching symbol by an AND gate. To solve this issue, we need to detect states with possible false-positive matching conditions and refine the matching with simpler conditions that each can be implemented without regenerating this issue. In our previous example, if we split the matching conditions into two simpler cases, AB and CD, we do not encounter this problem anymore. Grapefruit detects states with bogus matching conditions (if implemented directly in BRAM) automatically and refines them to minimum possible matching conditions and assigns each newly generated rule to a new state and removes the original state. Grapefruit repurposes the Espresso [93] logic minimizer to find the minimum number of refined rules set to implement every matching condition with BRAMs without generating false-positives. More details on this can be found in [69].

Signal Sharing

while LUTs are very flexible in implementing any matching condition (and generally in implementing every boolean function) without generating false positives, it is not the most efficient way to use them for pure LUT-based design. Our approach with BRAMs, to implement the matching condition partially and to combine them with an AND gate, can be emulated with LUTs as well. The benefit of this approach is that the middle 8-bit partial matching can share this signal with other states as well. For example, in Figure 4.5, part (a) shows an NFA that processes two 8-bit symbols per cycle. A pure LUT design without emulating BRAM design leads to the design in part (b). In this case, we rely only on synthesizer to reuse signals without any guide via the generated HDL code. However, following the partial matching approach leads to the design in part (c). In this design, every partial matching condition is implemented in LUT, and each STE picks its required matching condition in different dimensions and combines them with an AND operation. As shown in part (c), STEs can share the partial matching conditions and save the total number of required LUTs. Design (c) needs to go through the false-positive checking procedure, as it is vulnerable to this issue, but it can be resolved using our matching refinement procedure explained for the pure BRAM design.

In summary, Grapefruit supports three matching architectures: pure BRAM, pure LUT, and LUT with signal-sharing. The user can select its preferred architecture at compilation time, and Grapefruit generates

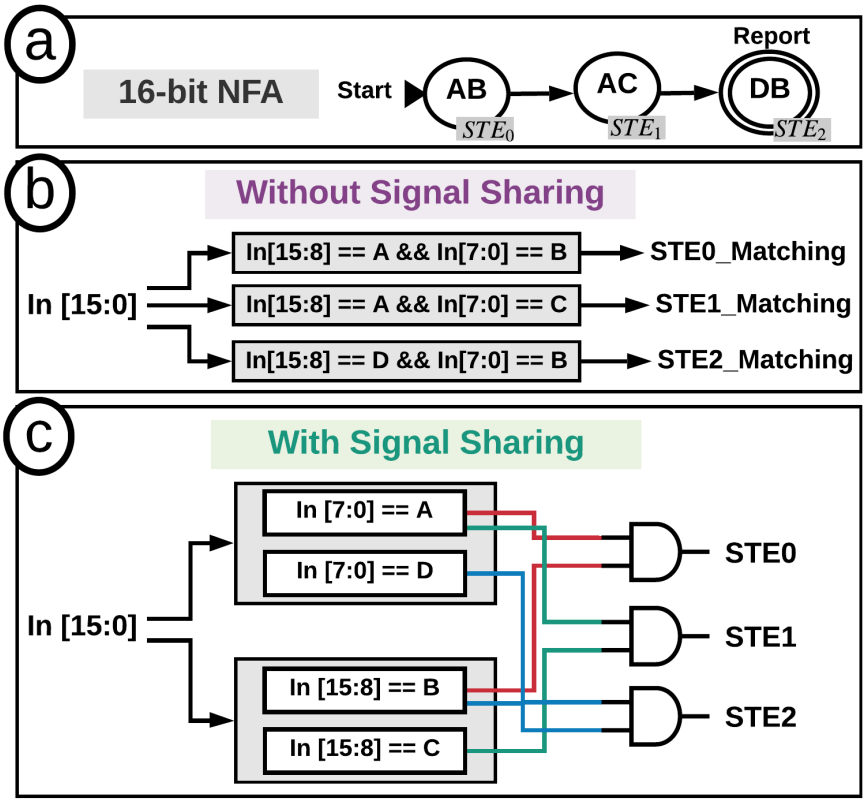

Figure 4.5: An example on signal sharing.

all the HDL codes necessary to implement it. In addition, Grapefruit can also implement a hybrid version of pure BRAM and LUT with signal-sharing by letting the user to freely put partial matching-conditions in BRAMs or LUTs.

4.3.2 Compiler Optimizations

In the application layer, researchers want to have access to a tool that gives them a rich and descriptive interface to define an automaton and examine/debug them with an input stream. Our compiler, called APSim, uses a well-known and actively maintained python graph processing package, NetworkX, as its main building block to benefit from its reliability, speed, and extensive documentation with examples for further development by other collaborators. In APSim, an automaton is considered as a directed multigraph where the nodes or edges can carry symbol data and NFA metadata. All the automata related algorithms have been implemented on top of this concept. For example, a strided automaton that consumes two symbols per cycle can be achieved by finding every path of length two in the input automaton and replace it with a single edge in the strided automaton.

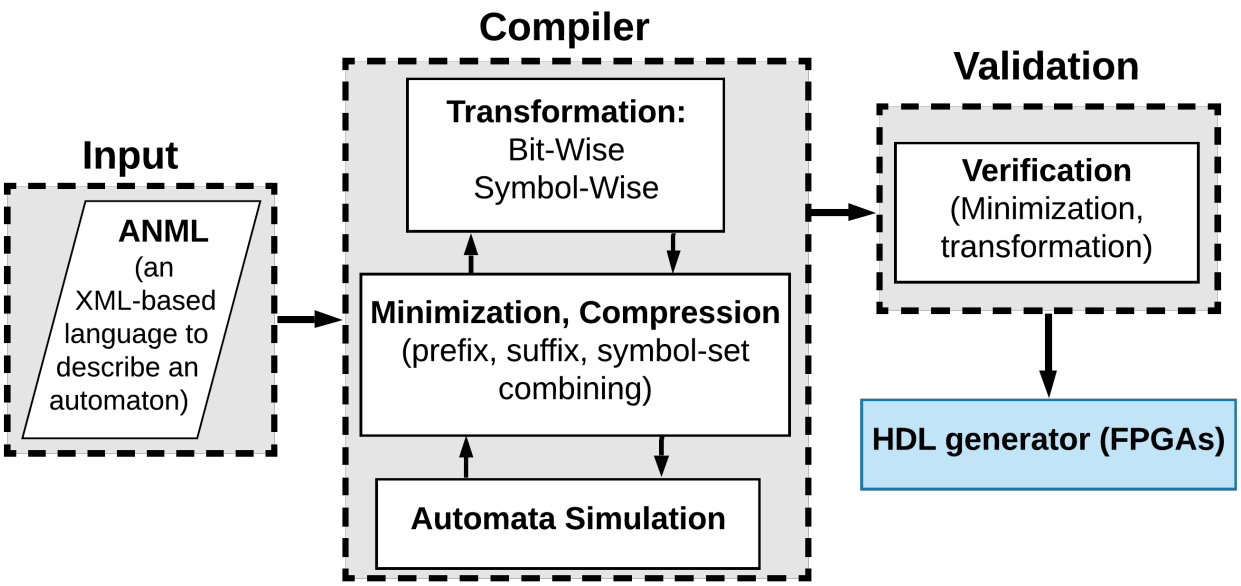

Figure 4.6: Different components in our back-end compiler.

Stream-based transformation verification is also supported in APSim to check the correctness of transformations. Users can develop their own transformations and easily compare report status of the new automaton with the original automaton by streaming the same input to both of them and check the equivalence of report states status per cycle. This functionality also works with striding, where a multi-strided automaton can be compared against its original representation. APSim provides a comprehensive, parameterizable, and easy-to-modify tool. Figure 4.6 shows different components of APSim.

Temporal Striding

researchers are interested in using a tool that gives them enough flexibility to explore design space parameters comprehensively to discover high-throughput and efficient designs. One crucial design parameter in the architectural domain of automata processing is the symbol's bit-width (or alphabet size), which is mainly determined by the application. APSim provides temporal striding functionality, which is a set of transformations to change the bitwidth processing while keeping the automaton functionality intact.

Temporal Striding [91, 89] is a transformation that repeatedly squares the input alphabet of an input automaton and adjusts its matching symbols and transition graph accordingly. The transformed automaton is functionally equivalent to the original automaton, but it processes multiple symbols per cycle, thus increasing throughput. On the other side, the strided automata can cause state and transition overhead. The transformation overhead depends on the properties of automata, such as the in/out degree of states, the number of symbols, and the number of transitions. This creates an interesting performance vs. resource

overhead tradeoff, and our framework supports and facilitates exploration of it and gives the hardware designer the chance to pick the sweet spot for a specific hardware platform. This sweet spot may vary significantly across different hardware domains such as in-memory processing, FPGA, or Von Neumann machines.

For example, DNA sequencing has only four characters in its alphabet (A, T, C, and G) [39], which can be represented with 2-bit symbols. Conversely, in frequent pattern mining [27, 24], the alphabet size can be in the order of up to billions of independent symbols (application dependent), which requires a much longer bitwidth. This parameter sets the bit-rate processing for a machine that consumes one symbol per cycle. In the bitwise approach (Figure 4.6), the input automaton is first flattened to its binary representation (one bit per edge) and traversed by collecting every possible path with the length equal to the target bitwidth. The symbol-wise approach does the same traversing as the bitwise approach, but it collects symbols as primitives compared to the bitwise approach. More details are presented by Sadredini et al. in [69, 66].

4.4 Evaluation Methodology

NFA workloads: We evaluate our proposed claims using ANMLZoo [44] and Regex [94] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. AutomataZoo [95] is mostly an extension of ANMLZoo (9 out of 13 applications are the same), and the difference is that ANMLZoo is normalized to fill one AP chip (with up to 48K states). To provide a fair comparison with prior work, we use ANMLZoo benchmarks because the prior work has used ANMLZoo in their evaluation, and their source code is not publicly available for evaluation with the AutomataZoo benchmark.

We present a summary of the applications in Table 5.2, including the number of states and transitions in each benchmark as well as the average degree (the number of incoming and outgoing transitions) for each state. The higher the degree, the more challenging the benchmark is to map efficiently to the FPGA's underlying routing network.

Experimental setup: all the FPGA results are obtained on a Xilinx Virtex UltraScale+ XCVU9P with a PCIe Gen3 x16 interface, 75.9 Mb BRAM, and 1182k CLB LUTs in 16nm technology. The FPGA's host computer has an eight-core Intel i7-7820X CPU running at 3.6 GHz and 128 GB memory. Designs are synthesized with the Xilinx Vivado v2019.2.

For the interconnect structure, we follow an area-efficient tree structure for Data Plane and Control Plane. In the Data Plane, we construct the tree by starting from the report buffers and assign every five buffers to one AXI interconnect. Every five AXI interconnects are connected to the next level of AXIs. This iterative approach continues until we reach a single AXI interconnect and connect its master port to the DDR

Table 4.1: Benchmark Overview

Benchmark	#Family	#States	#Transitions	Ave. Node Degree
Brill [44]	Regex	42658	62054	2.90
Bro217 [94]	Regex	2312	2130	1.84
ClamAV [94]	Regex	49538	49736	2.0
Dotstar [94]	Regex	96438	94254	1.95
ExactMath [94]	Regex	12439	12144	1.95
PowerEN [44]	Regex	40513	40271	1.98
Protomata [44]	Regex	42009	41635	1.98
Ranges05 [94]	Regex	12621	12472	1.97
Snort [44]	Regex	100500	81380	1.61
TCP [94]	Regex	19704	21164	2.14
EntityResolution [44]	Widget	95136	219264	4.60
Fermi [44]	Widget	40783	57576	2.82
RandomForest [44]	Widget	33220	33220	2
SPM [44]	Widget	69029	211050	6.11
Hamming [44]	Mesh	11346	19251	3.39
Levenshtein [44]	Mesh	2784	9096	6.53

memory controller. The same procedure is applied for the Control Plane by letting ten nodes connected to the interconnects (instead of five), as the Data Plane is running at a slower frequency and a high-bandwidth design is not necessary. Register slicing and AXI datapath FIFOs are enabled for the Data Plane with the burst-size set to its maximum possible value (256), following Xilinx recommendations for high-throughput design. Bus-width in both Data Plane and Control Plane is set to 32 to reduce the routing congestion. All these parameters can be set to different values in compile time thanks to our rich compiler APIs. Moreover, we set the clock constraint to 250MHz to be satisfied by the synthesizer.

4.5 Experimental Results

This section discusses the performance results of Grapefruit with different design choices using sixteen automata benchmarks and compares Grapefruit with prior work (REAPR+).

4.5.1 LUT-based vs. DRAM-based design

Table 4.2 shows different statistics for LUT-based design and BRAM-based design. As expected, benchmarks with a higher number of states and transitions (shown in Table 5.2 have higher LUT, FF, and BRAM usage, and also higher power consumption. However, the frequency is a function of both automata size (number of state/transitions) and automata structure (such as fan in/out or average node degree). For example, EntityResolution has 95,136 states and SPM has 69,029 state. However, SPM frequency is 175.6 MHz while

Table 4.2: Comparing LUT-based design vs BRAM-based design (in single-stride automata or 8-bit processing).

Benchmark	LUT-based						BRAM-based					
	LUTs	FFs	BRAM18	BRAM36	Power (W)	Clock (MHz)	LUTs	FFs	BRAM18	BRAM36	Power (W)	Clock (MHz)
Brill	144,273	205,162	82	503	6.9	231.3	130,876	203,385	160	503	7.1	258.1
Bro217	31,442	45,552	9	72	4.2	291.0	30,413	45,086	20	72	4.2	264.8
ClamAV	90,747	113,685	23	149	5.7	259.9	89,919	115,729	96	149	5.8	245.9
Dotstar	247,999	324,571	116	710	9.7	237.6	222,694	321,527	292	710	9.5	240.6
ExactMatch	43,442	58,568	13	99	4.6	278.6	41,684	58,262	31	99	4.5	260.8
PowerEN	192,204	278,126	117	779	8.4	166.7	172,119	275,330	233	779	8.5	247.2
Protomata	171,941	230,871	95	591	7.6	170.7	149,861	228,257	367	591	8.0	249.0
Ranges05	44,008	58,527	13	100	4.6	253.2	41,702	58,198	31	100	4.5	260.9
Snort	202,488	279,736	116	763	9.0	163.5	179,862	273,870	243	763	8.6	232.6
TCP	69,685	95,531	31	208	5.3	254.6	64,278	59,822	67	208	5.2	264.1
EntityResolution	122,452	148,907	41	268	6.1	228.8	108,406	148,611	81	268	7.2	248.9
Fermi	145,439	221,904	97	604	6.9	203.4	133,357	221,011	255	604	8.1	208.3
RandomForest	126,562	170,431	69	435	6.4	244.1	80,364	111,977	156	316	6.0	253.1
SPM	179,597	244,101	153	893	9.3	175.6	153,790	242,659	244	893	9.6	201.7
Hamming	29,066	40,703	7	60	4.2	293.3	27,848	40,611	15	60	4.3	271.7
Levenshtein	20,356	27,563	3	41	4.0	303.4	20,719	28,564	4	41	4.0	262.7

Table 4.3: Striding and signal-sharing effect for TCP benchmark.

Design Parameter	Symbol Size	LUTs	FFs	BRAM18	BRAM36	Power (W)	Clock (MHz)	Throughput (Gbps)
BRAM-based	8-bit	64,278	59,822	67	208	5.2	264.1	2.06
	16-bit	64,083	96,862	115	296	5.7	244.5	3.82
	32-bit	73,773	106,138	184	489	6.5	198.6	6.21
LUT-based	8-bit	69,685	95,531	31	208	5.3	254.6	1.99
	16-bit	105,150	96,604	33	362	5.3	225.5	3.52
	32-bit	165,228	97,016	33	362	5.4	217.8	6.81
Signal-Sharing on LUT-based	8-bit	65469	94900	31	208	4.8	245.6	1.92
	16-bit	69,559	97,110	43	296	5.0	243.7	3.81
	32-bit	78,412	106,265	37	489	5.3	223.1	6.97

EntityResolution frequency is 228.8 MHz (30% higher frequency). This is because the average node degree in EntityResolution is 4.6, while the average node degree for SPM is 6.11.

BRAM-based design reaches a higher operational frequency for larger benchmarks, such as PowerEN, Protamata, and Snort, possibly due to the limitations of LUTs. LUTs in Xilinx FPGA's can implement a single 6-bit boolean function. To implement 8-bit functions, multiple LUTs need to be serially connected, which leads to a higher latency. However, BRAMs can implement 8-bit boolean functions with a single memory read operation. For smaller benchmarks, the sparsity of BRAMS across the chip and routing latency associated with it become a disadvantage, and the popularity of LUTs becomes a winning factor. The number of utilized BRAM18s in BRAM-based design is always higher than the number of utilized BRAM18s in LUT-based design because matching logic is implemented in BRAM18s blocks. However, BRAM36 remains the same for both designs as Grapefruit only uses the BRAM18 for matching, and BRAM36 is mainly used by the report interconnect. The number of LUTs in LUT-based design is always higher than BRAM-based design because these resources are utilized mostly for matching. However, a significant fraction of the LUTs are used for the interconnect logic. The number of FFs remains mostly the same as both designs have the same number of STEs.

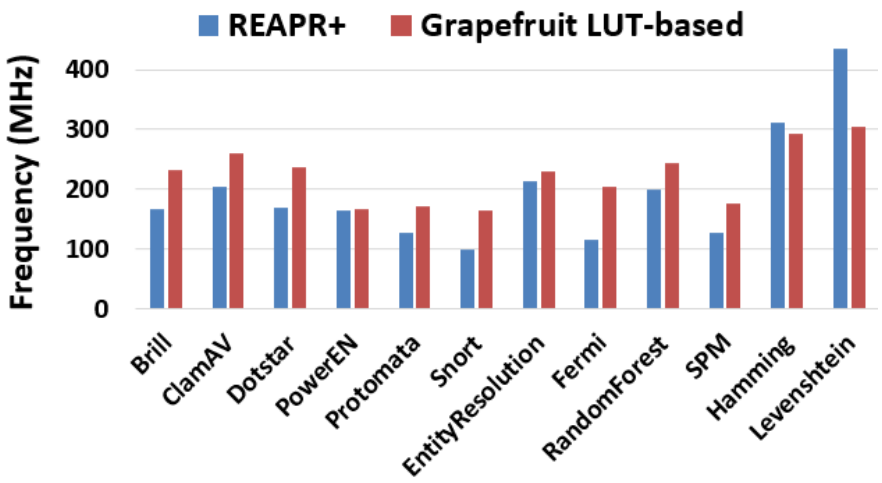

Figure 4.7: Comparing Grapefruit (8-bit) with REAPR+ [6].

4.5.2 Striding and signal sharing effects

Table 4.3 represents various statistics for the TCP benchmark when applying the temporal striding technique to both BRAM-based and LUT-based designs, and when applying signal-sharing optimization to the LUT-based design.

As discussed, temporal striding increases the number of state and transitions, which translates to higher LUT, FF, and BRAM utilization and slightly higher power consumption. However, it significantly increases throughput as it processes multiple symbols in one cycle. In BRAM-based design, increasing processing rate from one symbol per cycle (8-bit processing) to four symbols per cycle (32-bit processing) results in 3× higher throughput at the expense of only 2.37× higher LUTs, 1.7× higher FFs, 2.5× BRAMs, and 1.26× higher power consumption. Similarly, In LUT-based design, increasing the processing rate from one symbol per cycle (8-bit processing) to four symbols per cycle (32-bit processing) results in 3.4× higher throughput at the expense of only 1.15× higher LUTs, 1.01× higher FFs, 1.58× BRAMs, and 1.01× higher power consumption. In summary, striding capability gives the user the ability to explore the area, power, delay, and throughput trade-offs and makes the best design choices based on the target objectives.

The signal-sharing technique is applied to the LUT-based design. This optimization reduces the number of LUTs 1.5× in 16-bit design and 2.1× in 32-bit design. Moreover, signal sharing improves throughput and reduces power consumption.

4.5.3 Comparison with prior work

We compare Grapefruit (in 8-bit processing) with REAPR+ [6]. This tool only generates the kernel and does not consider the I/O. The authors have evaluated their solution on ANMLZoo benchmarks using

Xilinx Kintex-Ultrascale xcku060-ffva1156-2-e FPGA. Figure 4.7 shows that the Grapefruit full-stack solution performs 9%-80% better than kernel results in REAPR+ for the majority of the benchmarks. On average, Grapefruit has 3.8× more LUTs and 5.9× more FFs, and this is mainly because Grapefruit supports I/O.

REAPR+ performs better for Hamming and Levenshtein, and this is because these benchmarks are relatively small, and the designer can force a higher clock constraint. Our clock constraint is set to 250 MHz and increasing the clock constraint can significantly increase the frequency for smaller benchmarks, such as Hamming and Levenshtein.

4.6 Conclusions

This paper presents *Grapefruit*, a publicly available framework for automata processing on FPGAs. Grapefruit consists of two main components: (1) an integrated compiler for automata simulation, minimization, transformation, and optimization, and (2) an HDL generator that produces a full-stack design for a set of automata to be processed on FPGAs. We develop an efficient reporting architecture and pipeline design, along with a set of hardware optimizations and parameters. Our framework allows researchers to investigate frequency and resource-usage trade-offs and provides an easy-to-understand and easy-to-modify code for them to explore new ideas. Grapefruit provides up to 80% higher frequency than prior works that are not fully end-to-end and 3.4× higher throughput in a multi-stride solution than a single-stride solution. An interesting direction for future work would be automatic learning-based parameter tuning using static and dynamic behavior of automata in an application.

Chapter 5

Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching

The growing demand for high-performance pattern matching has motivated several efforts in designing software-based [73, 86, 64, 87, 71, 96, 97, 98] and FPGA-based [36, 65, 64, 88, 63] multi-stride pattern processing solutions that can process multiple-symbols per cycle. However, multi-symbol processing forces more pressure on memory bandwidth in CPU/GPU approaches and causes routing congestion for complex patterns in FPGA solutions. Moreover, FPGA solutions are mainly customized for network-based patterns, so current FPGA solutions may not perform well for other applications with totally different automata structures and behavior.

Our study on the existing PIM accelerators for pattern matching reveals that: (1) all these architectures are based on 8-bit symbol processing (derived from ASCII), and our analysis on a large set of real-world automata benchmarks reveals that the 8-bit processing dramatically underutilizes hardware resources, and (2) multi-stride symbol processing, a major source of throughput growth, is not explored in the existing in-memory solutions.

In the existing in-memory automata accelerators, 50%-70% of hardware resources are spent for state-matching [56, 60, 30, 4]. We study the state-matching resource utilization across a diverse set of automata benchmarks, and we found 86% of the time, only 3% of resources are utilized! This is mainly because in all these architectures, each state is modeled with a memory column of size 256, and 8-bit symbols are

one-hot encoded in the memory columns to be able to accept a range of symbols (up to 256 symbols) in each state. However, the number of symbols accepted by a state is fewer than 8 symbols in 86% of the time. This, in turn, implies that the classic approach of one-hot encoding for matching drastically over-provisions state-matching resources, which incurs significant performance penalties and leads to an inefficient and costly design. Moreover, real-world automata benchmarks are often extensive in terms of state count, too big to fit in a single hardware unit, and in current memory-centric architectures, usually need multiple rounds of reconfiguration and re-processing of the data. Therefore, design density plays a vital role in overall performance.

To address this problem, we propose to reduce the memory column-height to 16, to take advantage of the common case that few symbols matche any particular state, and at the same time, convert (or *squash*) the automata to process 4-bit symbols instead of 8-bit per cycle without losing any accuracy. With our proposed optimizations, this transformation increases the number of states on average just 1.7×, and in turn, requires 1.7× more memory *columns* to encode the states. However, the total required memory *cells* decreases 9.4× (or $\frac{256}{16\times1.7}$) because the columns are so much shorter. This provides a higher total state capacity, which results in fewer rounds of reconfiguration, and improves the overall utilization and performance.

On the other hand, naively *squashing* 8-bit processing to 4-bit processing halves the throughput and limits the benefits of our solution. We fix this issue by proposing an in-SRAM multi-stride automata processing architecture that can process multiple 4-bit symbols per cycle. Our architecture, Impala, leverages the shorter memory subarrays and seeks a wider parallel solution using multiple combined memory columns to process multiple 4-bit symbols instead of a longer serial memory column that processes only one 8-bit symbol. Impala's compiler pre-processes an automaton and makes it compatible with our hardware design.

A transformed automaton (*squashed and strided*) has a higher number of transitions (edges) than its original automaton, and this makes the placement of transitions to interconnect resources more challenging. To address this issue, our compiler leverages the observation of Sadredini et al. [4] that the real-world automata have a diagonal-shaped connectivity pattern, which then can be efficiently compacted in a hierarchical memory-mapped interconnect architecture. We propose to use a genetic algorithm (GA) to intelligently place the transitions into a compact design to amortize the transition overhead.

In summary, this paper makes the following contributions:

- We propose Impala, an area-efficient, high-throughput, and energy-efficient in-SRAM architecture for automata processing. These three-fold efficiencies are obtained from (1) an architectural contribution that utilizes *shorter-and-parallel* SRAM-subarrays instead of *longer-and-serial* subarrays, and (2) an algorithmic contribution which efficiently transforms an automaton and maps it to Impala's resources.

To the best of our knowledge, this is the first work that observes state-matching inefficiency in memory-centric accelerators and proposes an algorithm/architecture co-design for *multi-stride* automata processing for in-situ computations.

- To minimize the number of states in the transformed automata, we exploit Espresso [99], a CAD tool that efficiently reduces the complexity of digital circuits, and maps our state minimization problem to a logic minimization problem. Moreover, we propose an efficient placement algorithm by leverage a genetic algorithm.

- We perform thorough performance analysis on Impala and prior in-memory automata processing architectures. Our sensitivity analysis reveals that the 4-stride design on 4-bit nibbles (16-bit processing per cycle) provides the best overall throughput per area on Impala, which is up to 3.7× and 536× higher than Cache Automaton (CA) [30] and the Automata Processor (AP)[55], respectively. Moreover, Impala's 16-bit design provides 2.8× higher throughput than CA, 1.4× of which comes from our architectural contribution and 2× from the algorithmic benefit that re-shapes an automaton to process larger input size per cycle.

- We also compare Impala with FPGA multi-stride pattern matching solutions. We conclude that Impala provides ~ 10× higher frequency and ~ 20× higher throughput than the best performing solutions [36, 65] for 16-bit symbol processing rate. Moreover, Impala's 16-bit has 7.7× higher throughput than these FPGA solutions for a 64-bit processing rate.

- We present APSim, an open-source toolkit for cycle-accurate automata simulation, multi-symbol processing transformation, minimization, and performance modeling on our proposed architecture.

5.1 Related Work

A number of multi-stride automata processing engines have been proposed on CPUs and GPUs [73, 86, 64, 87, 71]. Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [44] and makes compression techniques [100] less efficient. Moreover, multi-symbol processing causes more pressure on memory bandwidth, because more states and transitions are required to be processed in each clock cycle.

Several FPGA solutions for single-stride [51, 35, 84, 52] and multi-stride [36, 65, 64, 63] automata processing have been proposed. Yang et al. [36] proposed a multi-symbol processing for regular expressions

on FPGA and utilizes both LUTs and BRAMs. Their solution is based on a spatial stacking technique, which duplicates the resources in each stride. This increases the critical path when increasing the stride value. Yamagaki et al. [65] proposed a multi-symbol state transitions solution using a temporal transformation of NFAs to construct a new NFA with multi-symbol characters. This approach only utilizes LUTs and does not scale very well due to the limited number of lookup tables in FPGAs. In addition, in their multi-striding method, the benefit of improved throughput decreases in more complex regexes (with more characters or highly connected automata), mostly due to routing congestion. All the current multi-striding solutions are inspired by networking applications such as Network Intrusion Detection Systems (NIDS). However, patterns in other applications can have different structure and behavior, e.g., higher fan-outs, and this makes it difficult for NIDS-based FPGA solutions to map other automata to FPGA resources efficiently [52].

Accelerators designed specifically for regex processing have been proposed [37, 77, 78, 50] to accelerate pattern matching and automata processing. In general, while these solutions provide high line rates in principle, they are limited by the number of parallel matches, state transitions, and shape of the automata. None of these solutions consider multi-symbol processing, and Subramaniyan et al. [30] show that their in-SRAM solution, Cache Automaton (CA), has higher throughput and lower area consumption than these accelerators.

Unlike FPGA and regex accelerators that are optimized for pattern processing in network applications, several memory-centric automata processing accelerators have been recently proposed to improve the performance of general pattern matching. The Micron Automata Processor (AP) [55] and CA [30] propose in-memory hardware accelerators for single-stride automata processing. They both allow native execution of NFAs by providing a reconfigurable substrate to lay out the rules in hardware. They exploit the inherent bit-level parallelism of memory to support many parallel transitions in one cycle. The AP provides a DRAM-based dedicated automata processing chip, while the CA proposes an on-chip solution by repurposing a portion of the last-level cache for automata processing and has shown higher throughput than previous solutions. Prior work has already shown that the AP is at least an order of magnitude better than GPUs and multi-core processors [74], and CA is at least an order of magnitude better than the AP [30].

To improve throughput, Subramaniyan et al. [101] propose a parallelization technique by replicating an automaton and splitting the input stream among them, and show speedup over the AP (such splitting is only needed when there are not enough regexes to leverage the capacity of the automata hardware). The speedup depends on the input stream, and the upper-bound speedup is equal to the number of hardware replications. High capacity in the automata accelerator is beneficial, and this approach is complementary.

Liu et al. [57] demonstrated that not all the states in an NFA are enabled during execution, and thus, do not need to be configured on the hardware. This reduces the hardware resources for an automaton

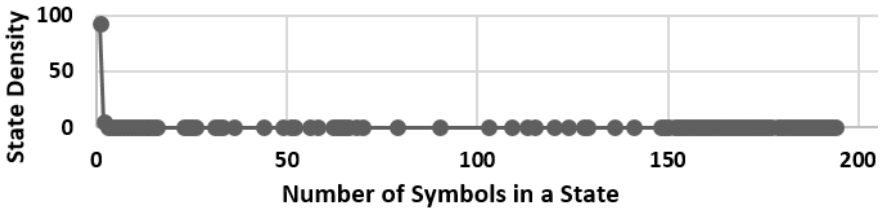

Figure 5.1: Normalized histogram of the states based on the number of accepting symbols. More than 86% of the states accept less than 8 symbols. More than 73% of the states only accept one symbol.

on the in-memory automata accelerators, which in turn increases the performance when the application is very large and requires several rounds of re-configurations. However, the benefits of their approach is input-stream-dependent and cannot be directly compared with prior solutions.

None of the prior in-memory solutions explores multi-symbol processing to increase throughput and overall performance. This work is the first to present a multi-symbol processing architecture co-designed with automata transformation to improve density.

5.2 Algorithmic Design

5.2.1 Motivation

Our key observation across a set of 20 real-world and synthetic automata benchmarks reveals that about 73% of the states only match against a single symbol. This implies that only a single cell in a memory column of 256 cells is set to 1, and the rest are 0. Figure 5.1 demonstrates the frequency of the number of matching-symbols across all the states in these benchmarks normalized to the total number of states. This histogram is highly skewed to the left, where more than 86% of the states are matched against at most eight symbols. This means only 3% of the cells in memory columns are utilized!

While a memory column with 256 cells is powerful enough to implement any boolean function with eight inputs, existing automata computing architectures implement a relatively simple function (e.g., 73% of the time, states are comparing against a single symbol, which is a boolean function with a single product term). In other words, a simpler low-cost matching architecture targeting the dominant case of a small number of matching symbols is more efficient than the existing costly matching architecture targeting infrequent complex matching conditions. To better utilize the state-matching resources, we *squash* the 8-bit symbols to 4-bit, and re-shape the automaton accordingly for correct and lossless functionality. The corresponding hardware change is that state-matching columns now can be designed with shorter subarrays (16 memory cells instead of 256), which reduces the waste significantly.

Generally speaking, as technology shrinks, SRAM arrays are moving from tall to wide structures with fewer rows [102, 103, 104]. This provides a better SRAM energy efficiency at a lower supply voltage. Recently, researchers have started to explore shorter SRAM subarrays to design accelerators in state-of-the-art applications, such as deep neural networks. For example, Lie et al. [105] propose an in-SRAM computation for binary neural networks [106]. Interestingly, they conclude that shorter SRAM subarrays (i.e., shorter memory columns) provide a better classification accuracy due to a smaller quantization error when calculating the partial sum in convolution operation. These support the applicability of Impala design, which relies on short memory subarrays.

A 4-bit automaton halves the processing rate. To increase the throughput, Impala proposes to process multiple 4-bit symbols/cycle versus one 4-bit symbol per cycle. The algorithmic aspects are discussed in this section, and the corresponding architectural solution is explained in Section 5.3.

5.2.2 Vectorized Temporal Squashing and Striding (V-TeSS)

Squashing: As discussed, the default 8-bit processing uses memory columns with 256 cells; however, the matching symbols of an STE are sparse, making this an underutilized matching resource. The squashing step converts the default 8-bit automaton to its equivalent 4-bit automaton. With empirical analysis, we realized that 4-bit conversion is the sweet spot and incurs minimal overhead compared to other squashing sizes (e.g., 2-bit or 3-bit processing - see [107] for more detail). Squashing to 4-bit increases the number of states 2.52× on average. But, the memory column size is exponentially decreased from 2^8 to 2^4.

Figure 5.2 (a) represents an 8-bit NFA in classical non-homogeneous representation. Symbols are represented in hexadecimal (e.g., $\backslash xBD$ represent an 8-bit symbol). Impala's compiler first reshapes the automaton to process 4-bit symbols (Figure 5.2(b)). For simplicity, we picked a simple automaton, and therefore, 8-bit to 4-bit conversion seems straightforward. However, in an automaton with loops and states with ranges of symbols (e.g., $[X-Y]$ notation in regex), we cannot simply break the states into two consecutive states with 4-bit symbols each. Impala's compiler first generates single-bit-automata by replacing each edge in the 8-bit original automaton with a sequence of states of length 8, and then traverses the bit-automaton with paths of length 4 and concatenates edges to generate 4-bit symbols. Due to space limitation, we do not explain the details here, as the algorithm to convert an 8-bit to 4-symbol automaton is not our main contribution.

Vectorized Temporal Striding: As expected, the 4-bit automata processing scheme halves the processing rate compared to the 8-bit automata. To increase the throughput (equals or more than 8-bit version), we again reshape the squashed 4-bit automaton to find its equivalent automaton that processes multiple 4-bit

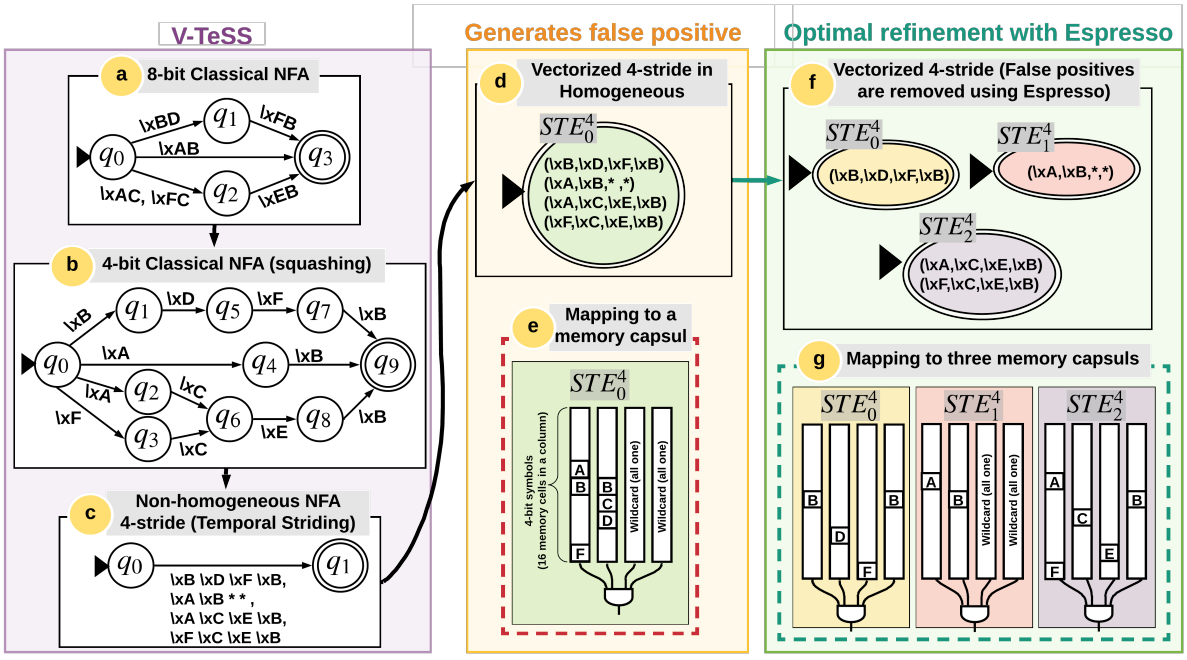

Figure 5.2: (a) Original 8-bit automaton. (b) Squashing the automaton in (a) to 4-bit processing. (c) Striding 4-bit automaton in (b) to process 16 bits/cycle. (d) Converting the automaton in (c) to its homogeneous representation. (e) Mapping STE_0^4 in (d) to one , which produces false positive reports (e.g., $(\backslash xB, \backslash xD, \backslash xE, \backslash xB)$ generates a false report). (f) Espresso solves it with minimal state splitting. (g) The states in (f) are mapped to three s.

symbols per cycle. We call this transformation *Vectorized Temporal Striding*, as the state's matching rules are organized in vectors. Arranging matching symbols in vectors provides a nice property that is aligned with Impala hardware support. Temporal Striding [91, 89] and its vectorized version are transformations that repeatedly square the input alphabet of an input automaton and adjust its matching symbols and transition graph accordingly. The transformed automaton is functionally equal to the original automaton, but it processes multiple symbols per cycle, thus increasing throughput.

Figure 5.2 (c) shows the 4-stride (i.e., processing four 4-bit symbols per cycle) automaton in (b), and (d) converts it to its homogeneous representation. In the notation STE_x^y, x is state index and y is the stride size. We call the resulting automaton vectorized 4-stride because each 4-bit symbol in STE_0^4 represents one dimension in a four-dimensional vector, and each dimension will be mapped to a separate memory column in Impala's area-efficient matching architecture (described in Section 5.3.1). The last two stars in the matching symbol $(\backslash xA, \backslash xB, *, *)$ are wildcards, which can be matched against any 4-bit symbol. Wildcards are used as a padding method to handle the cases where a report happens in the middle of a 16-bit input (the original automaton in (a) reports when the input is $\backslash xAB$).

We use the vectorization concept for efficient hardware mapping. The temporal striding method is already

discussed in prior work [89] and is not a contribution of this paper. Therefore, we do not discuss the details here.

Hardware efficiency: Naively increasing the processing rate to achieve higher throughput in the classical 8-bit in-memory architectures leads to a significant hardware cost. This is because each extra bit will double memory column size (e.g., 9-bit processing requires memory columns of 512 cells). This hardware consumption rate rapidly exceeds the reasonable bitline length due to its exponential growth. In addition, designing long bitlines is impossible without introducing stacked memory subarrays with partial address decoding and costly peripheral hardware. While the exponential hardware cost discourages a naive memory-based solution for multi-symbol matching, Impala redesigns the matching architecture to a set of short and parallel memory columns (16 cell in each column) combined with an AND gate, where every 4-bit increase in processing rate only requires an additional parallel 16-bit memory column. Columns are placed in different subarrays, and each receives 4 bits of the input symbols and processes it independently. This is a linear increment of hardware cost compared to the exponential growth in the naive matching architecture.

We call each of these combined memory columns a . Capsules are suitable for states with a simple matching character-set. For example, a single can easily handle the states with one matching vector, which are frequent in real-world automata benchmarks (see Fig. 5.1). However, there are infrequent matching cases that a single can not precisely implement, and this may lead to generating false-positive reports. Figure 5.2(e) shows how using only one to implement STE_0^4 generates wrong reports when the input is $\backslash xBDEB$. The first column matches $\backslash xB$, the second column matches $\backslash xD$, and the third and fourth columns match $\backslash xE$ and $\backslash xB$, respectively. However, this input is not supposed to be matched by STE_0^4. To address this issue, we exploit Espresso [99] - a tool that was originally developed for logic minimization problems - to find the minimum symbol splits to avoid false positives. In our example, splitting STE_0^4 to three states (Figure 5.2 (f)) and mapping them three capsule (Figure 5.2 (e)) solves the problem. Details of this refinement stage are explained in Section 5.3.1.

State and transition overhead: Temporal striding generally increases the length and cardinality of the alphabet and transition count [89, 65]. We apply squashing, striding, and false-positive matching removal on 21 real-world and synthetic automata benchmarks [44, 94] and observe that 2-stride and 4-stride implementations have a slight state and transition overhead. However, in the 8-stride design, combinatorial growth in the number of symbols (vectors) causes higher state/transition overhead, and this amortizes the architectural density benefits (details are discussed in Section 5.6.1). Our experimental results reveal that 4-stride (16-bit processing) yields the highest throughput per unit area (Section 5.6.4).

Theoretically, the area overhead of Impala's state-matching architecture is in the order of $O(SN)$, where S is the stride value (number of memory columns in a is S) and N is the number of nodes. The interconnect

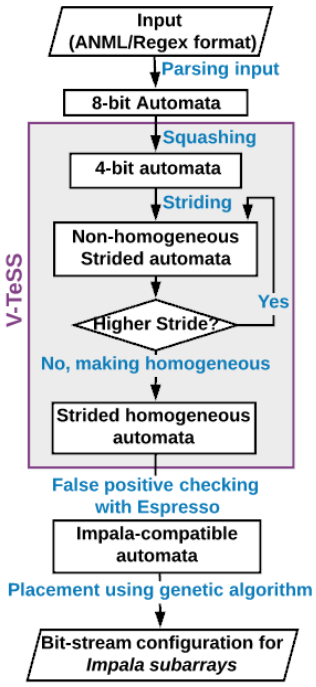

Figure 5.3: Offline pre-processing steps to prepare bit-streams to be configured on Impala's memory subarrays.

Table 5.1: Relative compilation time across architectures.

AP (APCompile [108])	**FPGA (Xilinx tools)**	**CA (APSim)**	**Impala 4-stride (APSim)**
>3 hours	~ *day*	~ *minutes*	~ *minutes*

area usage is in the order of $O(N^2)$ (for a fully connected interconnect), which is independent of stride value. This implies that striding does not directly add any area overhead to the interconnect (except the increase of nodes count). However, the increase in the number of transitions means more switches in a full-crossbar will be utilized. To efficiently map the transitions in the strided automata to the Impala's interconnect resources, we adopt a genetic algorithm, and then generate bit-stream configurations for Impala's state-matching and interconnect subarrays. Figure 5.3 summarizes the offline pre-processing steps to map an automaton to Impala's architecture.

Compile time: Combined algorithmic and architectural benefits of Impala result in high-throughput and area/energy-efficient design at the expense of higher compilation time (due to some stages in Figure 5.3) compared to CA. The compilation is only needed once per automaton before configuring the final bit-stream on the architecture. Table 5.1 shows a relative comparison for the compile/synthesis time for ANMLZoo [44] on different architectures. Impala's pre-processing stage (which includes V-Tess and GA) increases the compilation time compared to CA, but it is still much less than the AP compiler and FPGA synthesis tools.

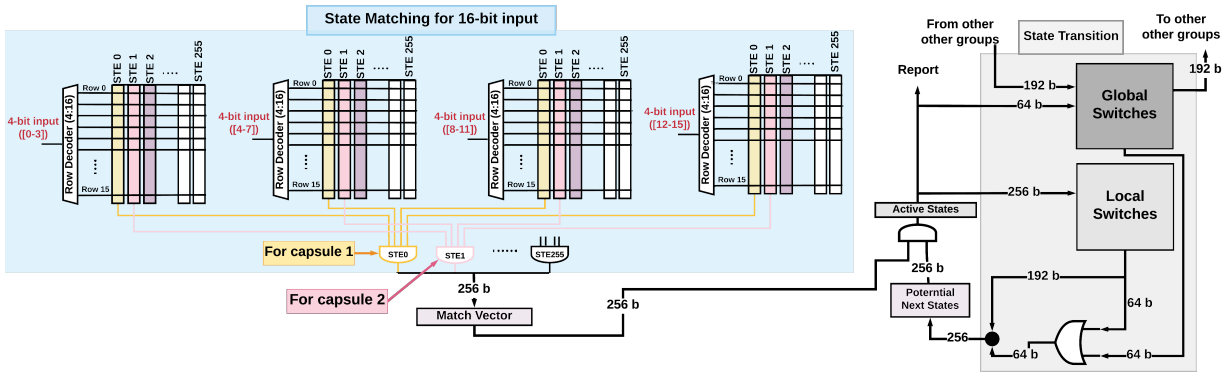

Figure 5.4: A 4-stride (16-bit) automata processing unit with a 2-level switch structure to support a larger automaton.

APSim is our open-source automata transformation and simulator and can be found here [1].

5.3 Architectural Design

Impala has two main stages, *state-matching* and *routing*, which are executed in pipeline. The *matching* compares the current cycle input symbols with all the states in parallel and the *routing* carries signals from the active states in previous cycles into their successors in current cycle using our proposed interconnect.

5.3.1 State Matching

The blue box in Figure 5.4 shows the state-matching architecture, which processes four 4-bit symbols and detects all the states that match against them. The matching operation is performed in a group of memory subarrays to distribute the matching burden among them in parallel. Memory columns with the same index (or the same color in the figure) in subarrays are combined using an AND gate to form a . Matching of each state will be assigned to one of these s. Each memory column in a processes a portion (e.g., 4-bit) of the input symbols by a single memory row access. The final single-bit matching result for each state is the output of the AND gate in each . Any additional 4-bit input processing only adds one more column to each . Now, it is clearer how each dimension in V-TeSS vectors can be mapped to the corresponding memory column index in a . Thanks to our parallel architecture, the latency of the overall matching for every stride value is always equivalent to read-cycle latency of a short bitline memory subarray plus the latency of the AND gate in .

[1] https://github.com/gr-rahimi/APSim

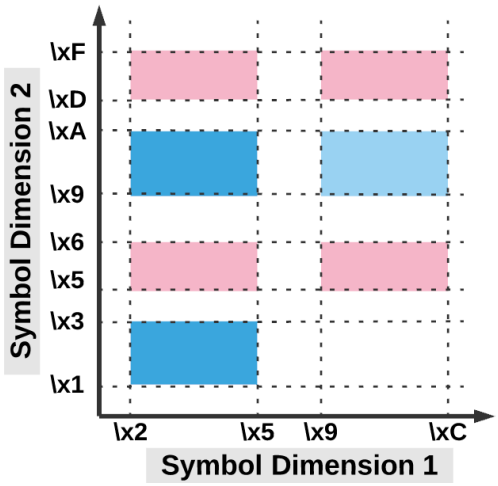

Figure 5.5: An example of an STE matching regions in a 2-stride automaton.

Challenges of Matching with Capsules

As discussed in section 5.2 and shown in Figure 5.2 (d), simply assigning each state to one suffers from a possible false-positive match. Figure 5.5 shows a more general example of when a false match happens in the relatively complex matching regions of a 2-stride automaton using a 2D representation. In this Figure, the X-axis shows the matching symbols of the first dimension (first 4-bit), and the Y-axis shows the matching symbols of the second dimension (second 4-bit). Each colored rectangle represents a matching region.

For example, the bottom-left-most region accepts the symbol range of $[\backslash x2 - \backslash x5]$ in its first dimension and symbol range of $[\backslash x1 - \backslash x3]$ in the second dimension. Impala's compiler also uses a memory-efficient range-based data-structure to store the matching data and this speeds-up automata transformation. Configuring all these colored regions in a single generates false positive matching when the input is from the white rectangular in the bottom-right-most region (i.e., $[\backslash x9 - \backslash xC]$ in first dimension and $[\backslash x1 - \backslash x3]$ in the second dimension).

To address this issue, an easy solution would be to split the seven colored regions to seven new states and assign one to each of them. However, this can be very costly, especially when a state matches against several regions. We observed that splitting into three states with matching regions shown as pink, dark blue, and light blue, and assigning each state to one , would efficiently solve the problem.

However, refining matching regions to a minimal number of sub-regions to increase the total capacity is a challenging problem, especially for high stride values with many matching regions intersecting each other. We observed that this minimization problem is equivalent to the minimization version of set covering problem, which is known to be NP-hard [109]. The next section explains how we solve this problem efficiently using an existing tool.

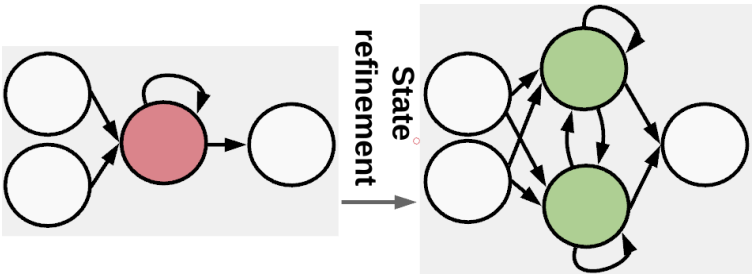

Figure 5.6: Splitting a state to avoid false positives.

Espresso

Espresso [99] is a CAD tool developed at IBM using heuristics for efficiently reducing the complexity of digital circuits. Interestingly, our state-splitting problem is similar to the Sum Of Product (SOP) gate-level logic minimization problem with the support for multi-valued variables [110]. In an SOP minimization problem, Espresso tries to minimize the number of products to use fewer resources in traditional Programmable Logic Arrays (PLA) hardware [111]. Impala can be seen as a PLA with 16-valued input variables, where each memory column in a is a discrete variable with 16 different values. Columns in a are combined with an AND gate which translates to a product term of 4 variables each with 16 values.

State refinement: In our problem, each product term will physically be translated to a new state and implemented using a . The new states will replace the original state by connecting all the parents and children of the original state to all the new states. If the original state has a self-loop, then all the new states should have self-loop as well and must be connected to each other to keep the equivalence to the original automaton. Fig. 5.6 shows an example of how the state in *red* (which causes false positives) is split into two green states. The number of splits is determined by Espresso.

Espresso input/output: The input for Espresso is a text file containing the matching vector of the under process states, represented as multi-valued truth tables. The output of the Espresso is also a text file that specifies the minimum number of required product terms to cover the original matching space. Each of these products is guaranteed to cause no false positive and can be safely configured in Impala's .

5.3.2 Interconnect

The interconnect provides the functionality to move active states forward in time toward the next states. A state S gets activated if (1) the current symbol matches the state S and (2) any of its parents were activated in the previous cycle. The second condition implies that the interconnect should provide the OR-functionality. CA [30] proposes a memory-mapped full-crossbar interconnect based on 8T SRAM memory cells to provide wired-OR functionally on bitlines. The memory blocks are of size 256×256 (local switches), the word-lines

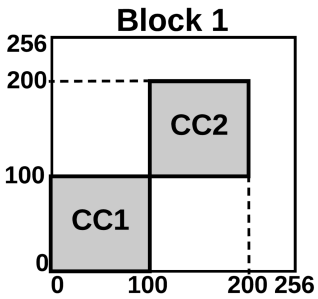

Figure 5.7: Full-crossbar resource utilization.

(rows) are driven by a set of states (one row per state), and bitlines (columns) drive the same set of states (one column per state). This arrangement accommodates connection among every pair of 256 states, as every column intersects with every row. The memory cell at row i and column j stores '1' if there is an edge between state i and state j. To support larger automata with more than 256 states, a two-level interconnect model is proposed to provide inter-block connectivity among local switches using global switches.

NFAs for real-world automata applications are typically composed of many independent rules or patterns, which manifest as separate connected components (CCs) with no transitions between them. The CA crossbar switch is utilized by packing CCs as densely as possible using a greedy approach. Figure 5.7 is an example of how two CCs, each with 100 states, are packed in a local switch block. We found that there are two problems with the state placement to the interconnect resources in CA model. First, the switch resource from index 200 to 256 in Figure 5.7 remains unutilized if the size of CCs are larger then unused portions. Second, if an automaton is larger than 256 states and has long-distance loops, it cannot be handled by CA's placement algorithm. Our general interconnect is based on CA's full-crossbar design. Sadredini et al. [4] showed that the connectivity patterns in real-world automata are diagonal-shaped in a full-crossbar design mapping, and this insight can be used to allow for reduced crossbar switches. We take inspiration from this observation and present a placement solution that automatically addresses the issues in CA placement for reduced crossbars using a genetic algorithm.

The gray box in Figure 5.4 shows the interconnect architecture using the local switch and global switch. Local switches are driven by the currently active states, and the output of the interconnect subarray is combined with the matching signals to compute the active states for the next cycle. Impala expands the connectivity among local switches by letting 64 nodes in a local switch (called *port-nodes (PNs)*) have full connectivity to all the port nodes of three other local switches using a dedicated 256×256 global switch subarray. 64 PNs in local switches are combined with 64 outputs of global switches to provide inter-group connection. We call the set of 4 local switches communicating with each other by their PNs (using a global switch) as *group of four (G4)* (see Figure 5.9 (a)).

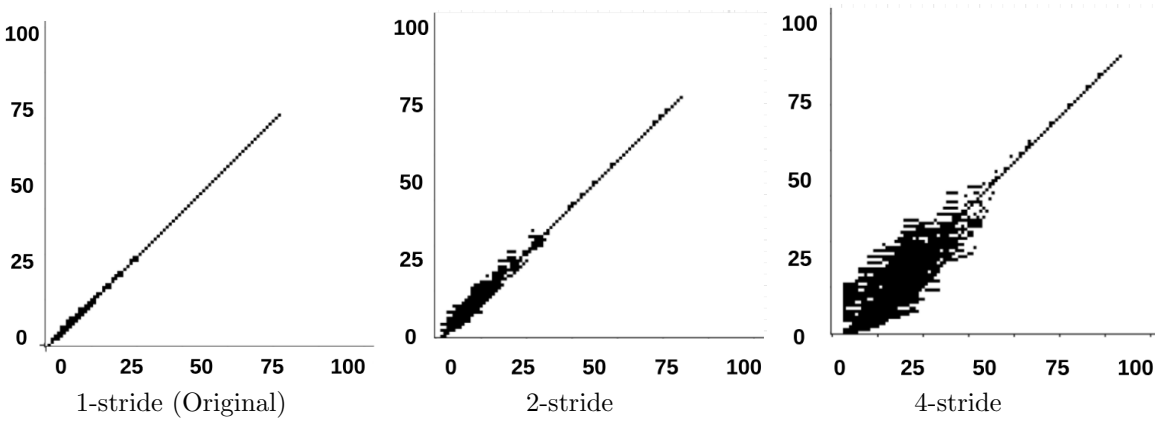

Figure 5.8: Union heatmap of routing switches with BFS labeling for all the connected components in Dotstar06. States are labeled with BFS starting from the start states. Each dark point at (x,y) shows an edge from state y to state x.

To address the placement problems in CA (to be able to efficiently place any large automaton with up to 1024 states in a G4 or break a small CC among two local switches to utilize the unused portion of crossbar), a placement strategy is required to consider (1) the limited inbound and outbound connectivity in PNs and (2) the connectivity pattern of an automaton to assign an integer label to each state, which later is mapped to the interconnect resources in G4. Furthermore, striding makes the connectivity pattern more sophisticated as strided automata have more transitions [89]. Figure 5.8 shows the effect of striding on the interconnect pattern evolution for Dotstar06 benchmark in ANMLZoo[44]. The 4-stride automata have higher transitions than 1-stride, which translates to higher utilization in a full-crossbar interconnect.

G4 Visualization

Figure 5.9 (a) visualizes the expanded layout of all possible connectivity supported in a G4 in a diagonal-shaped structure (inspired from [4]) using all four local switches (gray rectangles) and the global switch (purple rectangles). X axis shows the source index and Y axis shows the destination index. For example, if point (x, y) is in the gray rectangle, it is possible to support connection from state indexed x to state indexed y using local switches, or if it is in any of the purple regions, the global switch can provide the connectivity. Otherwise, if it is not in any of those regions, it is not possible to support that connectivity. The G4 switches can accommodate CCs of up to 1024 states. In our experiments, after striding, all the connected components have fewer than 1024 states. To support even larger automata, a higher-level switch can be used to connect G4 switches. The global switches in the upper-left or bottom-right cover the long-distance loops. Impala's compiler uses a genetic algorithm (GA) to effectively try different combinations of assigned indices to states and find a solution with zero missing connection in G4.

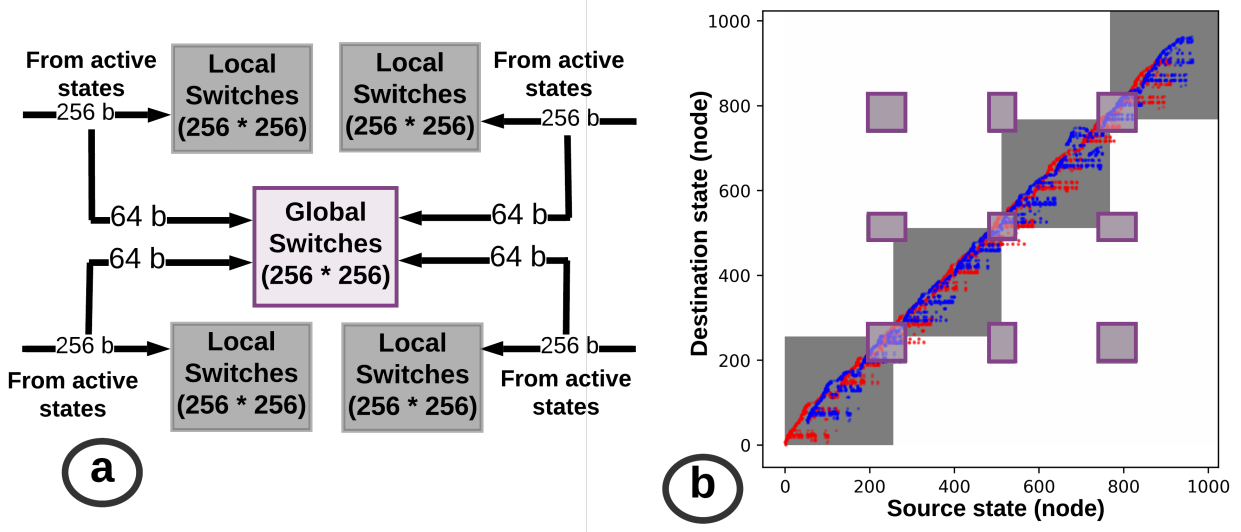

Figure 5.9: (a) G4 switch model and (b) its visualization.

Placement using genetic algorithm

A genetic algorithm (GA) is a computation model inspired by evolution and natural selection. In GA, the original problem model is interpreted as chromosome-like data representation, and evolution happens through three main operations: selection, crossover, and mutation. GA begins with a set of random chromosomes called a population. A population tries to evolve to a better population in each generation by prioritizing the fittest individuals (based on a goodness definition) to mate, mutate, and evolve into a new population of individuals.

In Impala, we encode our placement assignment by defining each individual as an array of unique integers of length 1024 (the number of states that fit in a G4). We initially assign unique labels to each state using the BFS algorithm or random generators to fill the population for the first time. The red dots in Fig. 5.9 (b) show the required connections when states are indexed using BFS in G4. As can be seen, there are many locations where the red dots locate outside of local switches and regions covered by global switches. This situation is anticipated, as BFS labeling assigns indices only based on the automata transition itself and does not consider the G4 connectivity limitations.

Impala's genetic algorithm placement evaluates the goodness of each individual based on the number of necessary switches that are currently available in G4. This evaluation increases the chance of individuals with better answers to survive into the next generation. Crossover combines individuals to generate a new individual by inheriting from their parents. Impala's compiler uses an ordered crossover method which swaps a random interval of two individuals with each other while keeping the order. Mutation happens by randomly swapping numbers inside each individual array. The blue-dots in Fig. 5.9 (b) shows the state mapping in

G4. This indicates that our GA can successfully place all the states within the G4 switch covering regions; therefore, this state labeling is a valid placement solution.

Case Study: EntityResolution from the ANMLZoo benchmark suite is widely used for approximate string matching in databases, and it has been shown that it has a complex connectivity pattern [4]. It has 1000 CCs, and on average, each CC has 95.12 states. We apply V-TeSS to this benchmark to generate the 4-stride automata, where now each CC has 108.9 states on average. Our placement algorithm is able to fit all these CCs in 117 G4 switches (on average, 930.7 states are mapped to each G4). Interestingly, none of these CCs can fit in G4 using a BFS labeling. We run the same experiments for all the benchmark in Table 5.2 and discover that our placement algorithm can successfully map all the states to G4 switch patterns for up to 4-stride automata.

5.4 System Integration

Configuration: Impala can be realized by re-purposing the available on-chip SRAM memory or custom-designed memory arrays. In on-chip SRAM arrays, column height for local subarrays are 256 [30], and Impala can partially utilize the available resources (16 out of 256 rows). However, the original 8-bit automata can be strided, which then can process multiple 8-bit symbols and utilize all 256 rows (evaluated in Section 5.12). On-chip solutions are more suitable for small scale applications where off-chip communication overhead becomes a concern, and low-latency designs are more appropriate. On the other hand, off-chip designs are more desired for large-scale applications where state capacity plays an important role, and I/O becomes less of a concern.

Impala is fully memory-based, and setting memory values configures the system for processing an input stream. After compiling all the automata and applying squashing, striding, and placement, the values for each memory cell in matching and interconnect arrays are determined. Assuming Impala is integrated into a system as a peripheral device, these values can be transferred to Impala using memory-mapped I/O communication such as Linux *mmap* command [112] or PCI commands. Fulcrum [19] proposes to employ CXL [113] interface for in-memory accelerators, which enables another option.

Run-time: Impala has two asynchronous FIFOs to hold the input symbols in the input buffer (IB) and reports in the output buffer (OB). At runtime, the host system communicates with the IB and OB using interrupt triggered memory-mapped IO or DMA while the interrupt service routine (ISR) is responsible to fill in the IB and evict the OB. Assuming 5GHz frequency for Impala and 1 MHz frequency for interrupt, an IB of size 2.5KB can store enough data to feed the Impala until the next IB interrupt. Recently, [62] has characterized the reporting statistics of ANMLZoo's benchmark. The results show that 10 out of 12

benchmarks produce fewer than 0.5 reports per cycle. This investigation motivates us to use 512 entries for the OB (4 bytes each for report meta-data) in order to keep a similar interrupt rate as the IB.

5.5 Evaluation Methodology

NFA workloads: We evaluate our proposed claims and architectures using ANMLZoo [44] and Regex [94] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. AutomataZoo [95] is mostly an extension of ANMLZoo (9 out of 13 applications are the same), and the difference is that ANMLZoo is normalized to fill one AP chip (with up to 48K states). To (i) provide a fair comparison with the AP, and (ii) evaluate on real-size applications, we use ANMLZoo benchmarks, but replicate each benchmark 1000 times to create larger benchmarks. We present a summary of the applications (in the original size) in Table 5.2.

Table 5.2: Benchmark Overview

Benchmark	#Family	#States	#Transitions	Ave. Node Degree	Largest CC Size
Brill [44]	Regex	42658	62054	2.9	67
Bro217 [94]	Regex	2312	2130	1.8	84
Dotstar03 [94]	Regex	12144	12264	2.0	92
Dotstar06 [94]	Regex	12640	12939	2.0	104
Dotstar09 [94]	Regex	12431	12907	2.0	104
ExactMath [94]	Regex	12439	12144	1.9	87
PowerEN [44]	Regex	40513	40271	1.9	52
Protomata [44]	Regex	42009	41635	1.9	123
Ranges05 [94]	Regex	12621	12472	1.9	94
Ranges1 [94]	Regex	12464	12406	1.9	96
Snort [44]	Regex	100500	81380	1.6	222
TCP [94]	Regex	19704	21164	2.1	391
ClamAV [44]	Regex	49538	49736	2.0	515
Hamming [44]	Mesh	11346	19251	3.3	122
Levenshtein [44]	Mesh	2784	9096	6.5	116
Fermi [44]	Widget	40783	57576	2.8	17
RandomForest [44]	Widget	33220	33220	2.0	20
SPM [44]	Widget	69029	211050	6.1	20
EntityResolution [44]	Widget	95136	219264	4.6	96
BlockRings [44]	Synthetic	44352	44352	2.0	231
CoreRings [44]	Synthetic	48002	48002	2.0	2

Experimental setup: We use our Automata compiler and simulator, APSim [114], to perform the pre-processing steps (such as V-TeSS), and emulate Impala and CA [30]. We have verified the functional correctness of APSim with VASim [79], which is an open-source automata simulator. The simulator takes automata in ANML format and processes the input cycle-by-cycle. Per-cycle statistics are used to calculate

Table 5.3: Subarray parameters for state-matching and interconnect (overhead of peripherals are included).

Usage	Cell Type	Size	Delay (ps)	Read Power (mW)	Area (μm^2)
State-matching (Impala)	6T	16×16	180	0.58	453
State-matching (CA)	6T	256×256	220	5.52	9394
Interconnect	8T	256×256	150	6.07	20102

the number of active subarrays, which is then used to calculate energy consumption. To estimate area, delay, and power of the memory subarray in Impala and CA model, we use a standard memory compiler (under NDA) for the 14nm technology node and nominal voltage 0.8V (details in Table 6.2). The global wire-delays are calculated using SPICE modeling in CA. In the memory compiler, the 8T-cell design has wider transistors than 6T; therefore, 8T subarrays are faster and have higher area overhead than 6T subarrays. Moreover, because the AP, CA, and Impala have similar run-time execution models, we can disregard data transfer and control overheads to make general capacity and performance comparisons among these platforms.

Comparison metric: To compare spatial automata processing architectures (the AP, CA, and Impala), we use *throughput per unit area*. Throughput is defined as the number of bits that can be processed in one second ($frequency \times Bitwidth\_size$). If the automata (connected components) in a benchmark cannot fit in one hardware unit (HU), we replicate HUs until all the automata are accommodated. The total area is calculated by multiplying the area of one HU and the number of required HUs for each benchmark.

5.6 Experimental Results

In this section, we evaluate Impala and compare it with the state-of-the-art solutions, such as CA and the AP, as well as multi-striding solutions on FPGAs.

5.6.1 Overhead Analysis of V-TeSS

Squashing to 4-bit design and then striding (V-TeSS) change the shape of automata and increases the number of states and transitions. Table 5.4 shows the number of states and transitions in V-TeSS in different strides normalized to the original 8-bit automata designs across 21 automata benchmarks. We observed that applications with higher average node degree, such as EntityResolution, RandomForest, and SPM (see Table 5.2) result in higher state and transition overhead. This is mainly because more combinations of paths need to be processed in temporal striding. On the other hand, CoreRings and BlockRings have almost no overhead when striding (8-bit and 16-bit). This is mainly because all states have equivalent matching symbols (a single unique symbol) which can effectively benefit from the classic NFA minimization techniques such as

Table 5.4: States and transitions overhead in different strides for V-TeSS normalized to the original 8-bit design. For example, 2-stride processes 4×2 bits of input in each cycle and has 1.12× more states and 1.34× more transitions than the original 8-bit design. However, 4-bit design needs memory columns with 2^4 rows while 8-bit design requires memory columns with 2^8 rows.

	#States Normalized to 8-bit original design				**#Transitions Normalized 8-bit original design**			
Stride	**1**	**2**	**4**	**8**	**1**	**2**	**4**	**8**
Bits per cycle	**4-bit**	**8-bit**	**16-bit**	**32-bit**	**4-bit**	**8-bit**	**16-bit**	**32-bit**
Brill	2.18	1.05	2.03	16.99	1.87	1.07	3.31	42.35
Bro	2.19	1.08	1.17	4.29	2.50	1.18	1.28	4.94
Dorstar03	2.24	1.05	1.19	2.05	2.84	1.15	2.03	3.08
DotStar06	2.33	1.05	1.22	2.38	3.21	1.17	2.16	3.82
Dotstar09	2.45	1.06	1.25	2.53	3.65	1.18	2.30	4.29
ExactMatch	2.05	1.02	1.05	1.40	2.12	1.05	1.07	1.44
PowerEN	2.46	1.08	1.26	6.04	3.66	1.18	1.72	8.77
Protomata	3.08	1.44	1.98	6.22	4.01	2.03	3.63	7.43
Ranges05	2.08	1.03	1.10	3.38	2.24	1.09	1.50	4.14
Ranges1	2.10	1.04	1.15	3.60	2.29	1.13	1.80	4.99
snort	2.79	1.12	1.56	10.25	4.87	1.34	3.73	22.48
TCP	2.47	1.10	1.52	8.55	3.56	1.37	4.08	17.24
ClamAV	2.03	1.00	1.03	7.01	2.06	1.01	1.06	7.42
Hamming	1.99	1.01	1.73	22.97	1.59	1.01	2.65	31.31
Levenshtein	2.66	1.01	2.52	5.35	1.79	1.02	4.19	11.25
Fermi	2.23	1.03	1.06	26.75	2.11	1.04	1.34	30.71
RandomForest	5.07	1.82	3.74	27.75	9.22	3.42	13.97	82.19
SPM	2.60	1.40	5.11	11.88	2.55	2.21	23.44	32.20
EntityResolution	3.45	1.05	1.73	1.08	4.00	1.06	3.11	1.10
BlockRing	2.01	1.00	1.01	3.61	2.02	1.01	1.03	3.98
CoreRing	2.00	1.00	1.00	1.00	2.00	1.00	1.00	1.00
Average	2.52	1.12	1.68	8.34	3.10	1.34	3.97	15.53

prefix merge and suffix merge (both implemented in Impala's compiler). The state and transition overhead of 8-stride automata is higher; this is because the number of symbols (32-bit symbols) increases, which results in more vectors and a higher chance for false positives in each state. Splitting the states extensively in 8-stride causes higher state and transition overhead. This, in turn, surpasses the area benefits of Impala. Therefore, we evaluate Impala for up to 4-stride design for the rest of the paper.

Squashing the 8-bit design to 4-bit increases the number of states 2.52×, and then striding the 4-bit design to 8-bit (2-stride) and applying compiler minimizations reduces the number of states very close to the original 8-bit design. Both 2-stride 4-bit and original 8-bit have similar throughput; however, the 4-bit design requires substantially smaller memory subarrays. On average, the state overhead in 16-bit design (processing four 4-bit symbols per cycle) is only 1.7× compared to the original 8-bit design, but its processing rate is twice and the design is denser. This explains that our V-TeSS method co-designed with Impala architecture

Table 5.5: Pipeline stage delays and operating frequency. The detail implementation of the AP is not publicly available.

Architecture	State Matching	Local Switch	Global Switch	Max Freq. (GHz)	Operating Freq. (GHz)
Impala (14nm)	180 ps	150 ps	170 ps	5.55	5
CA (14nm)	220 ps	150 ps	249 ps	4.01	3.6
AP (50nm)	-	-	-	0.133	0.133
AP (14nm)*	-	-	-	1.69	1.69

* Projected to 14nm

provides a three-fold throughput, area, and energy-efficiency benefits compared to prior 8-bit architectures (details in the following Sections). This also confirms that Espresso has split the false-positive states with minimal overhead. It is important to note that transition overhead translates to higher utilization of the crossbar interconnect and does not impose extra hardware overhead (discussed in Section 5.3.2 and evaluated in Section 5.6.3).

5.6.2 Overall Performance

The overall performance of spatial automata processing architectures is determined by $frequency \times bits/cycle$. The delays and frequencies of different pipeline stages for Impala, CA, and the AP are shown in Table 6.6. Impala's state-matching delay is 180 ps (See Table 6.2) and it is similar for different stride designs. This is because all capsules are processed in parallel, and striding does not increase the pipeline stages delay (except for the minor difference in the 2-input vs 4-input AND gate delay in Impala design, which is less than 4ps in 14nm [115]. This is less than 2% of total delay in the state-matching stage). Both CA and Impala have similar hierarchical interconnect designs, and both local and global switches are evaluated in parallel (Figure 5.4). Following CA design, we assume the SRAM-based CA design slice of $3.19mm \times 3mm$. Therefore, the distance between SRAM arrays and global switch is assumed to be $1.5mm$. From SPICE modeling, the wire delay was found to be $66ps/mm$; therefore, the wire delay for global switches is $99ps$. Global switch delay for CA is 249 ps, which is composed of read-access latency (150ps) and wire delay latency (99ps). Impala state-matching size for 4-stride design is ~ × less; therefore, we assume $20ps$ wire-delay for Impala. Therefore, the global switch delay for Impala is 170 ps (150ps+20ps).

The frequency is determined based on the slowest pipeline stage, which is the global switch delay in both CA and Impala. We assume the operating frequency for them to be 10% less than what we have calculated, to consider potential estimation errors. The AP is designed in 50nm DRAM technology. To have a fair comparison, we project the frequency to 14nm technology, which is an ideal assumption.

In all these spatial architectures, state matching and routing happen in parallel. This, in turn, means

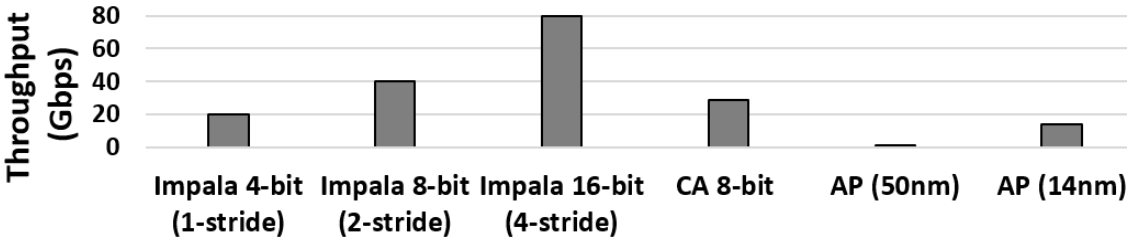

Figure 5.10: Overall performance of different spatial automata accelerators in Gbps.

that they have a deterministic throughput of one input symbol per cycle, and it is independent of the input stream. Figure 6.6 presents the overall achieved throughput for CA, AP, and Impala different stride designs. Impala 4-stride design processes 16 bits per cycle and achieves the highest throughput (5GHz×16-bit=80 Gbps). This implies that if the application fits in the hardware, the Impala 16-bit design provides 2.8× higher throughput than CA. *2× of the benefit comes from the algorithmic contribution, which reshapes the automata to process 16-bit symbols, and 1.4× from architectural contribution in which shorter subarray design results in higher frequency.* Moreover, Impala 16-bit has 5.9× higher throughput than the AP.

5.6.3 Area Overhead

Impala proposes area benefits coming from its architectural contribution, which is smaller state-matching subarrays and sharing interconnect resources while processing 16 bits per cycle. Figure 6.7 compares the area overhead of state-matching and interconnect of Impala 16-bit processing with CA and the AP (all in 14nm) for 32K STEs. Impala 16-bit has 5.2× and 34.5× smaller state-matching area overhead than CA and the AP (scaled to 14nm), respectively. Moreover, in total, Impala 16-bit has 1.34× and 3.9× smaller area overhead than CA and the AP, respectively. Sadredini et al. [4] show that the AP interconnect incurs routing congestion and limits the state-matching utilization. This implies that the area overhead to accommodate 32K states would be higher in practice for the AP.

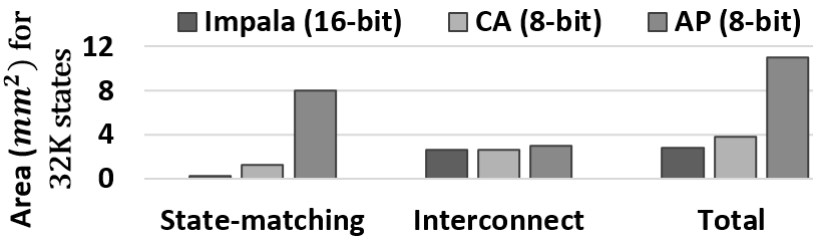

Figure 5.11: Comparing area overhead for 32K STEs.

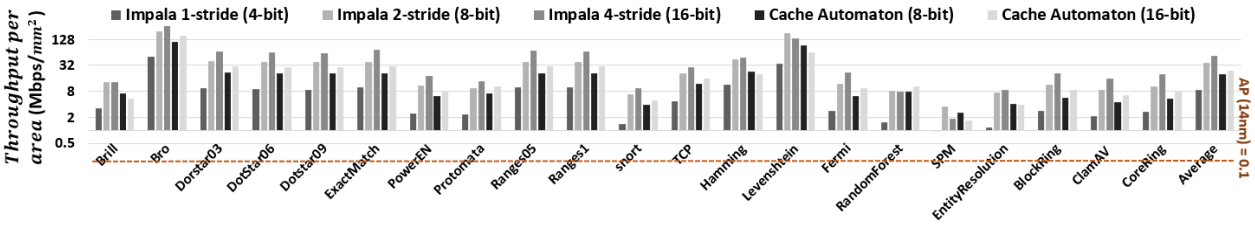

Figure 5.12: Comparing throughput per mm^2 area among Impala 4-bit design in different strides, Cache Automata (original 8bit design) in 1-stride and 2-stride, and the Automata Processor (AP), all in 14nm.

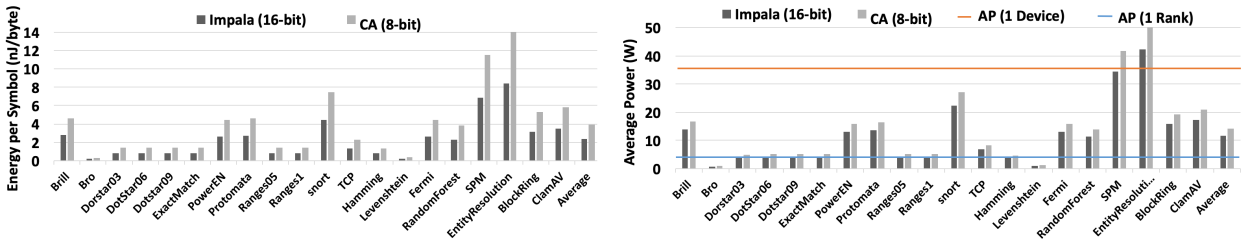

Figure 5.13: (left) Overall energy consumption of Impala compared to CA. (right) Overall power consumption of Impala compared to CA and the AP (reported by Micron [7]).

5.6.4 Throughput per unit area

This section combines throughput, area, and striding overhead effects all together and evaluates throughput per unit area across 21 applications. To have a more comprehensive comparison, we stride the original 8-bit automata using our temporal striding method and evaluate the Cache Automaton for 16-bit processing. This is shown as Cache Automaton (16-bit) in the Figure 5.12. From this figure, the Impala 16-bit design provides on average 2.7× (up to 3.7×) and 371× (up to 536×) throughput per unit area than CA and the AP, respectively. The benefits are calculated as $\frac{Throughput\ benefit \times Area\ benefit}{V\text{-}Tess\ overhead}$. For example, the benefits of Impala 16-bit over CA 8-bit are calculated as $\frac{2.8\times1.34}{1.39}$, where the V-Tess state overhead (1.39×) is calculated for real-size applications. The applications such as Bro, Dotstar03, ExactMatch, snort, and Fermi have smaller striding overhead for 2-stride and 4-stride designs (Table 5.4), and therefore, they present higher throughput per area in Impala 16-bit. On the other hand, SPM has a higher average node degree (Table 5.2), which results in a higher striding overhead (Table 5.4) and lower throughput per area than Cache Automaton 8-bit.

5.6.5 Energy/Power Consumption

This section discusses the energy/power consumption of Impala 16-bit and compares it to prior works assuming 10MB of input. To calculate energy consumption, we need to know (1) the number of active partitions for state-matching and switch blocks, and (2) the number of transitions between local switches to consider for the energy of driving wires.

Table 5.6: Comparison with mutli-stride FPGA solutions.

	Bits/cycle	**Clock rate (GHz)**	**Throughput (Gbps)**
Yang et al. [36]	16	0.212	3.47
Yamagaki et al. [65]	16	0.239	3.91
Impala	16	5	80

Note that it is not possible to power-gate state-matching memory arrays on a cycle-by-cycle basis. In order to power-gate these subarrays, it is necessary to know the potential next states beforehand. However, in the pipeline, the state-matching results and the next potential state are calculated simultaneously, which prevents the power-gating (one can still power-gate an array that is unoccupied). This observation is not considered in CA. We update the energy/power results in CA paper [30] based on this observation and our 14nm technology assumption. All the statistics per cycle are extracted from our compiler.

Figure 5.13 (left) shows the energy per input symbol for Impala and CA (energy details of the AP is not publicly available). We can observe that benchmarks with a larger number of states, such as Entity Resolution, Snort, and SPM consume higher energy. This is because these benchmarks have utilized more state-matching and switch arrays to accommodate a larger number of states. On average, CA consume 1.7× more energy per symbol than Impala. Energy efficiency of Impala comes from its density and compact design, which results in consuming lower dynamic energy due to shorter wires. Figure 5.13 (right) shows the average power consumption across benchmarks. On average, the power consumption of CA is 1.22× more than Impala. This is expected because: $\frac{CA\text{–}Energy}{Impala\text{–}Energy} \times \frac{CA\text{–}Frequency}{Impala\text{–}Frequency} = 1.7 \times \frac{1}{1.39} = 1.22$.

5.6.6 Comparison with multi-stride on FPGA

Yang et al. [36] and Yamagaki et al. [65] propose multi-stride regex processing solutions on FPGA and have evaluated their solutions on Xilinx-Virtex5 LX-220 and Altera Stratix II EP2S180, respectively (details are discussed in the related work in Section 6.2). Table 5.6 compares Impala with these solutions for 16-bit symbol processing rate on Snort dataset. In summary, Impala provides ~ × higher frequency and ~ 20× higher throughput than both of these solutions. Moreover, Impala with 16-bit processing rate has 7.7× higher throughput than these FPGA solutions for 64-bit processing rate. Imapla's compiler (pre-processing and placement) is at least an order of magnitude faster than FPGA synthesis (Table 5.1). The benefit of our approach, i.e., processing 4-bits symbols, can be applied to FPGAs, as we concluded from our preliminary FPGA-based experiments. Further exploration is left for future works.

5.7 Conclusions

This paper presents Impala, an in-memory accelerator for an efficient multi-stride automata processing. Impala is co-designed with our automata transformation algorithm, called V-TeSS, and leverages short and parallel memory columns to implement a dense, high-throughput, and low-power multi-symbol matching architecture. Overall, the benefits of Impala comes from two observations: (1) smaller state-matching subarrays provide higher utilization of memory-cells in memory columns, and this translates to higher density, and (2) V-TeSS transformation provides higher throughput with a linear increase in state-matching resources (or memory columns) relative to the original 8-bit design. *This paper concludes that an in-situ 4-stride automata processing with 16-bit memory columns provides the highest performance, and has up to 3.7× higher throughput per area and 1.22× lower power consumption than Cache Automaton.*

Chapter 6

Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration

In-memory automata processing model has three processing stages; state matching, state transition, and report gathering, and can be combined in a pipelined fashion. In the state matching stage, the current input symbol is decoded and all the states whose symbols match against it are detected by reading the fetched memory row. In the state transition stage, successors of active states are determined by propagating activation signals via a programmable interconnect. In the report gathering phase, the report data are accumulated and eventually analyzed for the final action or decision.

Prior work has mostly overlooked the real cost of providing a reporting architecture and assumed that reporting is not a bottleneck [30, 1, 70, 57] (and evaluated only the first two stages or the kernel). However, reporting incurs a significant cost when it is considered accurately. For example, the reporting architecture in the Micron's Automata Processor (AP) [55] has 40% area overhead [60] and up to 46× performance overhead due to stalls and host communications [62].

To improve the AP reporting architecture, Wadden et al. [62] propose finer-grain reporting buffers to reduce the report vector sparsity. However, their approach (1) needs to store relatively large metadata, (2) requires more complex peripherals to connect the smaller report buffers, (3) the reporting queue gets filled quickly when an application reports frequently, which causes stalls in the execution, and (4) it blindly reads all the generated reports and does not have any control to partially select or summarize the report data when

needed, all of which make their solution inefficient and hard to scale. Moreover, existing in-memory automata accelerators have a single processing rate fixed at design time—typically 8 bits. We study the state-matching resource utilization across a diverse set of automata benchmarks, and we find 86% of the time, only 3% of resources are utilized! This is mainly because each state is modeled with a memory column of size 256, and 8-bit symbols are one-hot encoded in the memory columns to be able to accept a range of symbols (up to 256 symbols) in each state. However, the number of symbols accepted by a state is fewer than 8 symbols 86% of the time. This, in turn, implies that the classic approach of one-hot encoding for directly matching an 8-bit alphabet drastically over-provisions state-matching resources, which incurs significant performance penalties and reduces state capacity. Moreover, real-world automata benchmarks are often extensive in terms of state count, too big to fit in a single hardware unit, and in current memory-centric architectures, usually need multiple rounds of reconfiguration and re-processing of the data. Therefore, design density plays a vital role in overall performance.

To address these issues, we propose Sunder, a highly reconfigurable, in-SRAM automata processing design with a flexible, compact, simple, and low-overhead memory-mapped reporting architecture. Our main observation is that transforming the common, fixed 8-bit automata processing rate (which requires 2^8 memory rows) to a multiple of 4-bit automata or *nibble processing* (which requires groups of 2^4 memory rows) can greatly reduce the required memory elements in memory-based automata processing solutions. We opportunistically utilize the saved memory rows in a subarray to store the reporting data locally and densely near the state-matching data. Our reporting architecture (1) greatly eliminates data movement and stalls due to reporting (zero stall for the 95% of the applications), (2) reuses existing state matching resources, thus, has minimal hardware overhead (less than 2%), (3) provides an easy and flexible mechanism for the host to analyze or summarize any portion of reporting data at any time, and (4) supports both sparse and dense reporting behavior very efficiently. In addition, Sunder presents a reconfigurable processing rate for automata processing, which enables throughput and density benefits for a diverse set of applications.

Liu et al. [57] demonstrated that not all the states in an NFA are enabled during execution, thus, do not need to be configured on the hardware. This reduces the hardware resources for an automaton on the in-memory automata accelerators (by splitting an automaton between the CPU and the AP), which in turn increases the performance when the application is very large and would require several rounds of re-configurations. However, this approach generates more intermediate results (or reports), which needs to be transferred to the CPU. Our proposed reporting architecture is complementary to their technique and can significantly improve the efficiency of reporting when larger intermediate reports are generated.

In summary, this paper makes the following **contributions**:

- We present Sunder, a highly reconfigurable, flexible, and scalable in-SRAM pattern processing architecture using a well-designed algorithm/architecture design methodology. Sunder benefits are achieved by (1) algorithmic transformations to improve the efficiency of in-memory processing, which enable a compact and low-cost in-situ reporting architecture (algorithmic contribution) and (2) a pipeline architecture and reconfigurable design (architectural contribution).

- We thoroughly analyze the behavior of a diverse set of applications to understand the reporting needs and efficiently optimize the architecture based on the different reporting behavior.

- We introduce a compact, simple, and localized memory-mapped reporting architecture to reduce data movement and host communication. Sunder does not incur stalls for 95% of the applications due to reporting during the execution of an application. This simply means that Sunder end-to-end performance is almost equal to the kernel performance, thus, can provide real-time processing with reliable performance.

- For the same state density, Sunder provides 9× larger reporting buffer than the Micron's Automata Processor and at the same time, has 2.2× lower overall area overhead in the same technology size. Moreover, on average, Sunder provides 280× higher throughput compared to the Automate Processor and 4-8× higher throughput compared to the state-of-the-art SRAM-based solutions (Impala [1] and Cache Automaton [30]) assuming AP-style reporting architecture.

- We provide an open-source framework for the algorithm transformations, mapping, and placement.

6.1 Background and Motivation

6.1.1 Existing Reporting Architecture

Micron's Automata Processor (AP) [55] uses a hierarchical reporting architecture with two levels of buffers for offloading reporting state bits. At the system level, the AP contains 32 D480 chips. Each D480 chip (Fig. 6.1) contains two independent half-cores that have independent automata states and edges. Each half-core has three separate reporting regions, where each reporting region is responsible for a maximum of 1024 reporting STEs. Each reporting STE is routed to one if these three reporting regions.

At runtime, if any of the 1024 reporting bits are activated, a full 1024-bit vector, as well as 64 metadata bits, are offloaded to the L1 storage buffer assigned to the triggered report state, as shown in Figure 6.1. When an L1 storage buffer is full, its content is offloaded to one of two L2 buffers shared with the other

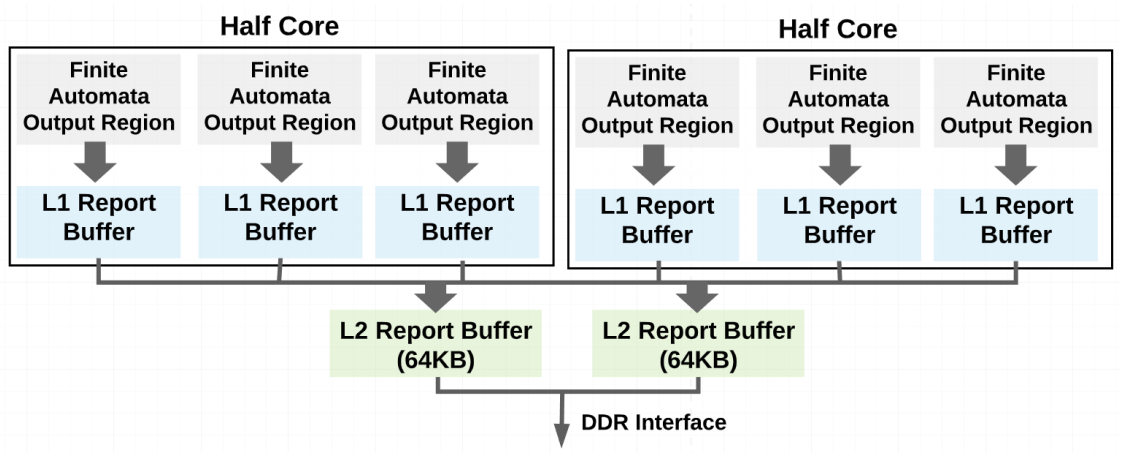

Figure 6.1: The Automata Processor reporting architecture.

half-core for eventual export off-chip. This architecture must stall during this offloading process because they do not support simultaneous push and pop operations. These L2 buffers are then transferred to the host.

Wadden et al. [62] recognized the high cost associated with very sparse reporting workloads on the AP and introduced a new reporting compression scheme for spatial automata architectures. They introduced a technique called Report Aggregator Division (RAD) to break up a large output report vector into smaller packets. This approach reduces the amount of data sent from the spatial accelerator to the host by offloading subsections of the report vector that contain report bits. As expected, this solution improves the AP-style reporting architecture stall rate by using fine-grained report vectors. However, this solution comes with the cost of increased metadata. Moreover, similar to the AP, it still uses the centralized reporting buffers with its negative impact on routing complexity and high propagation delay of reporting signals from the report sates to buffers. AP matching is based on slow DRAM operations, so such a latency could be covered by matching latency, but for our fast SRAM based design, this is no longer acceptable. In addition, it does not provide any summarization functionality in hardware to reduce the off-chip communication size for applications that do not need cycle-accurate report data.

6.2 Related Work

Generally, automata processing on von Neumann architectures exhibits highly irregular memory access patterns with poor temporal and spatial locality, which often leads to poor cache and memory behavior [44, 73, 86, 64, 87, 71], and this increases the cost of data movement.

Unlike FPGA [65, 64, 63, 36] and regex accelerators [37, 77, 78, 50] that are optimized for pattern processing in network applications, several memory-centric automata processing accelerators have been

recently proposed to improve the performance of general pattern matching [55, 30, 1, 101, 2]. The Micron's Automata Processor (AP) [55], CA [30], and Impala [1] propose in-memory hardware accelerators. They all allow native execution of NFAs by providing a reconfigurable substrate to lay out the rules in hardware. They exploit the inherent bit-level parallelism of memory to support many parallel transitions in one cycle. The AP provides a DRAM-based dedicated automata processing chip, while the CA and Impala propose an on-chip solution by repurposing a portion of the last-level cache for automata processing and has shown higher throughput than previous solutions. Impala proposes to transform automata to process 4 4-bit symbols in parallel using smaller subarrays. However, Impala's processing rate is not reconfigurable. Moreover, both Imapal and CA overlook to evaluate the real-cost of reporting.

Prior work has already shown that the AP is at least an order of magnitude better than GPUs and multi-core processors [27, 26, 24, 25, 116, 117], and CA is at least an order of magnitude better than the AP [30]. Liu et al. [47] proposed an optimized GPU solution for NFA processing by identifying the source of data movement and achieved significant speedup over existing GPU solutions, and even outperforming the AP for several applications. On average, Sunder outperforms the AP 280×, and therefore, we do not compare with the GPU solutions.

6.2.1 Motivation

All previous memory-centric implementations for automata processing [55, 30, 4, 1, 51, 50, 70, 85, 57] suffer from three problems. First, they all work with a fixed (mostly 8-bit) symbol processing rate decided at design time. Second, they all have either failed in realizing an efficient report architecture design or overlook the reporting stage. Third, their rudimentary reporting architecture does not provide any support to summarize reporting data in hardware.

Symbol size: existing in-memory automata accelerators have a fixed processing rate set at design time—typically 8 bits. We study the state-matching resource utilization across a diverse set of automata benchmarks, and we find 86% of the time, only 3% of resources are utilized! This is mainly because each state is modeled with a memory column of size 256, and 8-bit symbols are one-hot encoded in the memory columns to be able to accept a range of symbols (up to 256 symbols) in each state. While a memory column with 256 cells is powerful enough to implement any boolean function with eight inputs, existing automata computing architectures implement a relatively simple function (e.g., 73% of the time, states are comparing against a single symbol, which is a boolean function with a single product term). In other words, a simpler low-cost matching architecture targeting the dominant case of a small number of matching symbols is more

efficient and less costly than the existing costly matching architecture targeting infrequent complex matching conditions.

Reporting architecture issues: the reporting architecture module is responsible for collecting per-cycle report information and storing them in a buffer temporarily, to be transferred to the host whenever the host program needs to check for matches. Designing such a hardware module is not straightforward because there are a few concerns that need to be considered. First, report states are generated in different memory arrays and need to be routed toward the global reporting buffers, potentially with high latency. Second, choosing the right buffer bit-width is challenging due to its effect on area cost. A wide buffer solution (e.g., [55]) is attractive for an area-efficient design, as many report states are combined to create a single row of the report buffer, which results in smaller buffer control logic. However, a wide buffer can be more troublesome for applications with sparse and persistent reporting behavior, as the buffer gets filled up frequently, mostly with 0s.

On the other hand, a narrow buffer solution (e.g., [62]) works effectively for applications with sparse reporting behavior and can physically be placed near where the report states are generated. Because buffers are narrow and their capacity is limited, we need many of them to cover all the report states. The cost of the control and access logic of the reporting buffers from the host is not negligible as each needs to be controlled separately. We believe the lack of a feasible and efficient reporting architecture in prior work is one of the main concerns of integrating an efficient automata processing accelerator in a system.

Reporting strategy: None of the prior work on reporting architecture provides report summarization support, which can help to reduce the reporting I/O cost. Instead, they move the entire reporting data from reporting buffers to the host, and have the software to extract the information. For example, if an application only wants to know if a specific state has been triggered since the last time the report buffer was flushed, the host processor must currently first read all the reporting data of the buffer associated with that state and calculate the row-wise logical OR of the reporting cycles.

6.3 Analyzing Reporting Behavior

To motivate our efficient memory-mapped reporting architecture design, we first analyze the reporting behavior of a wide range of real-world automata benchmarks from ANMLZoo [44] and Regex [94] benchmark suites using their associated input streams. We use Virtual Automata Simulator (VASim) [118] to simulate the applications on the 1MB input stream provided with the benchmark suites and track all the reports over the course of automata execution.

Table 6.1: Reporting behavior summary

Benchmark	Static Analysis				Dynamic Behaviour (Input Dependent)				
	#Family	#States	#Report States	#Report States/ States (%)	#Reports	#Report Cycles	#Reports/Cycles	#Reports/ Report Cycles	#Report Cycles/ #Cycles (%)
Brill [44]	Regex	42658	1962	4.6	1092388	118814	1.067	9.19	11.33%
Bro217 [94]	Regex	2312	187	8.1	17219	17210	0.017	1.00	1.64%
Dotstar03 [94]	Regex	12144	300	2.5	1	1	0.000	1.00	≈ 0%
Dotstar06 [94]	Regex	12640	300	2.4	2	2	0.000	1.00	≈ 0%
Dotstar09 [94]	Regex	12431	300	2.4	2	2	0.000	1.00	≈ 0%
ExactMatch [94]	Regex	12439	297	2.4	35	35	0.000	1.00	≈ 0%
PowerEN [44]	Regex	40513	3456	8.5	4304	4303	0.004	1.00	0.41 %
Protomata [44]	Regex	42009	2365	5.6	127413	105722	0.124	1.21	10.08%
Ranges05 [94]	Regex	12621	299	2.4	39	38	0.000	1.03	≈ 0%
Ranges1 [94]	Regex	12464	297	2.4	26	26	0.000	1.00	≈ 0%
Snort [44]	Regex	66466	4166	6.3	1710495	995011	1.670	1.72	94.89%
TCP [94]	Regex	19704	767	3.9	103415	103198	0.101	1.00	9.84%
ClamAV [44]	Regex	49538	515	1.0	0	0	0.000	0.00	≈ 0%
Hamming [44]	Mesh	11346	186	1.6	2	2	0.000	1.00	≈ 0%
Levenshtein [44]	Mesh	2784	96	3.4	4	4	0.000	1.00	≈ 0%
Fermi [44]	Widget	40783	2399	5.9	96127	13444	0.094	7.15	1.28%
RandomForest [44]	Widget	33220	1661	5.0	21310	3322	0.021	6.41	0.32%
SPM [44]	Widget	100500	5025	5.0	47304453	33933	46.19	1394	3.24%
EntityResolution [44]	Widget	95136	1000	1.1	37628	28612	0.037	1.32	2.73%

#Report States: the number of states that are designed to be the report states in the application.
#Reports: the total number of reports when streaming 1MB of input data.
#Report Cycles: the number of cycles that at least one report is generated.

Table 6.1 shows a summary of the automata report statistics and behavior. *#States* represents the number of states in each application. *#Report States* shows the number of states that are labeled as reporting state in the application. As the fifth column represents, minimum 1% and maximum 8.5% of the states in the applications are reporting state. We use this observation to optimize the resources for our reporting architecture.

#Reports shows the total number of generated reports across the entire execution of the application. *#Report Cycle* shows the number of cycles in which at least one report is generated. For example, ExactMatch generates exactly one report in 35 cycles. One input symbol is processed per clock cycle; thus, the total number of cycles for the entire application to run is 1,000,000 for 1MB input stream. The last column shows the percentage of the cycles where at least one report is generated.

Reporting behavior: as Table 6.1 suggests, the reporting behavior varies significantly from application to application. Some applications report very infrequently (i.e., Dotstar03-09, ClamAV, Ranges05, Ranges 1). This is mainly because the automata in these applications are either a set of virus scanning signatures or detecting a bad behavior in a network, and this reporting behavior is expected. Hamming and Levenstein applications are designed for approximate string matching. Their input is generated randomly, and only a few strings within the scoring metrics were identified.

SPM reports nearly every 30 cycles (1,000,000/33,933), and in each reporting cycle, 1394 reports out of 5025 report states are generated on average (i.e., 20% of the reporting states generate a report every 30 cycles). This implies that the reporting architecture should be able to handle the bursty and dense reporting behavior of such applications to avoid significant performance loss.

Snort reports nearly every cycle, and 1.72 reports are generated on average in each reporting cycle. This implies that the reporting architecture needs to efficiently handle frequent but sparse reporting behavior. Other applications, such as Fermi, and RandomForest, report less frequently (e.g., once every 3000 cycles), and generate roughly 7 reports in each reporting cycle. They exhibit infrequent and relatively less sparse reporting behavior. These all imply that when designing a reporting architecture, hardware and application considerations are required to have a stall-free, efficient, and general-purpose solution for a variety of behaviors.

Application-specific report analysis: In addition to understanding the reporting behavior, it is crucial to realize how and when the generated reports for an application are going to be transferred to the host to make a decision or perform an action. For example, in network security applications, the generated reports (which demonstrates a malicious behavior in the network) should be immediately sent back to the host to make a quick decision. Moreover, some applications, such as SPM [24], may need to check if a range of input stream has generated a report or not, and they do not need to know all the reports during the entire execution of an application. Likewise, some applications only need to know if at least one report has happened during a portion or entire execution of the input stream; thus, a summarized reporting would be enough.

A well-designed reporting architecture should be able to transfer the minimal required reporting data (i.e., summarized, a portion, or the entire reports) when the application needs it. To the best of our knowledge, all the existing in-memory automata processing solutions sending the entire reporting data when the allocated buffers are filled as buffer flush instruction and there is no control from the host/application to transfer data selectively. *In Sunder, for the first time, our proposed reporting architecture provides access from the host to request the entire, a portion, or summarized reporting data at any point of time in an efficient and cost-effective manner.* Details are discussed in Section 6.5.2.

Dependency to input stream: the reporting behavior in every automata application changes with changing the size and characteristics of the input stream (i.e., dynamic behavior), and therefore, the underlying architecture should be robust and still efficient in these cases. The sensitivity analysis of reporting is discussed in Section 6.8.4.

6.4 Sunder Algorithmic Transformation

Sunder is based on an algorithm and architecture methodology. Sunder leverages the fact that 4-bit automata consume exponentially fewer memory rows for state encoding than 8-bit automata (2^4 vs. 2^8). The unused memory rows in a standard memory or cache subarray can be used to locally store the reporting data at a minimal cost. This section explains the algorithmic aspects of transforming an NFA with m-bit symbols (m is usually 8 and 2^8 memory rows are required for one-hot encoding of states) to 4-bit symbol automata,

which we call it *nibble processing*. 4-bit symbols only require 2^4 memory rows for one-hot symbol encoding. We then use the temporal striding algorithm presented in Impala [1] to stride the nibbles to our desired processing rate, and configure the Sunder processing rate accordingly.

6.4.1 Transforming to Nibble Processing

We utilize, FlexAmata [5], which is an automata transformation tool and transforms an *m-bit* automaton A to an equivalent *n-bit* automaton B, where n can be larger or smaller than m. This transformation is done in two steps; (1) converting A to a bit-level representation (A_b), and (2) generating automaton B by transforming A_b to process *n-bit* in each cycle. To generate the *n-bit* automaton, all the unique paths of size n in A_b and replace each of them as a single edge (or transition rule) in B.

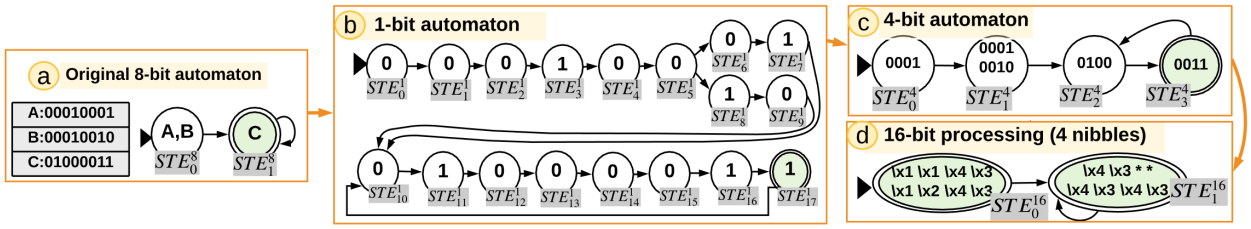

Figure 6.2: An 8-bit automaton (a) is converted to the minimized *1-bit* automaton (b). The *4-bit* automaton (c) is generated from the *1-bit* automaton. Finally, the 4-bit automaton (C) is strided to a 16-bit processing (d) using nibble units.

In our architecture, m is an application-dependent parameter (i.e., depends on the number of unique symbols in the application, and this is usually 8 because of the byte-oriented processing nature of the problems), and n is 4. The main reason to transform to 4-bit automata (instead of 2-bit, 3-bit, 5-bit, etc.) is that 4-bit processing has the lowest transformation overhead (i.e., lowest state and transition overhead).

Figure 6.2 explains how an 8-bit automaton is transformed into a 4-bit automaton. In the notation STE_x^y, x is state index and y is the bitwidth size. The original homogeneous automaton (a) has two states and accepts language $(A|B)C^+$. FlexAmata generates a binary automaton (b) and minimizes the states when possible. For example, the first 6 bits of symbols A and B can be merged. Then, the 4-bit automaton (c) is generated from the bit-automaton. In the 4-bit automaton, STE_0^4 is a start state and STE_3^4 is the final state. Each state processes one or more 4-bit symbols. STE_{13}^1 in 1-bit-automata is equivalent to reaching the state STE_2^4 in the 4-bit automaton. Although this transformation seems to be intuitive in this simple example, the real-word automata are very complex with loops and different rule properties, which makes the conversion non-intuitive.

6.4.2 Temporal Striding

As expected, the nibble processing scheme halves the processing rate compared to the 8-bit automata. To increase the throughput (equals or more than 8-bit processing), we utilize the *Vectorized Temporal Striding* technique introduced in Impala [1] to reshape the 4-bit automaton and find its equivalent automaton that processes multiple nibbles per cycle. Temporal Striding [91, 89] and its vectorized version are transformations that repeatedly square the input alphabet of an input automaton and adjust its matching symbols and transition graph accordingly. The transformed automaton is functionally equal to the original automaton, but it processes multiple symbols per cycle, thus increasing throughput. Arranging matching symbols in vectors provides a nice property that is aligned with sunder hardware support. Figure 6.2 (d) shows how a 4-bit automaton is temporally strided to a 16-bit automaton with nibble units (a vector of four 4-bit symbols).

There are two reasons for not directly converting an 8-bit automaton to a 16-bit automaton; (1) converting to the bit-automaton can enable the bit-level minimization, and this is especially useful when the application designer has not optimized the automaton and (2) The Sunder hardware is working with nibbles, and for an accurate and sound transformation, the 8-bit symbols should be converted to nibbles to rectify the potential false positives before striding. Transformation to nibble processing does not incur any false positive reports (more in Section 6.5.1). The State and transition overhead of nibble processing transformation is discussed in Section 6.8.1.

6.5 Sunder Architecture

Sunder can be realized by repurposing the last level cache (LLC) of recent processors, such as Intel Xeon with large LLC cache capacity (as Sunder's subarrays have the same size as a conventional L3 cache [119]). This section explains the architectural details of Sunder and discusses how Sunder implements each of the stages (i.e., state matching, state transition or interconnect, and reporting) in automata processing by leveraging the nibble processing transformation.

6.5.1 State Matching

In the state-matching stage, in every cycle, the current input symbol is decoded and all the states whose symbols match against it are detected by reading a memory row. The matching stage is realized by one-hot encoding of symbol sensitivity list of the states in memory columns of a subarray.

Figure 6.3 shows the Sunder architecture for one processing unit (PU) with up to 256 states. Each PU includes the state matching, state transition (interconnect), and reporting units, as well as the global

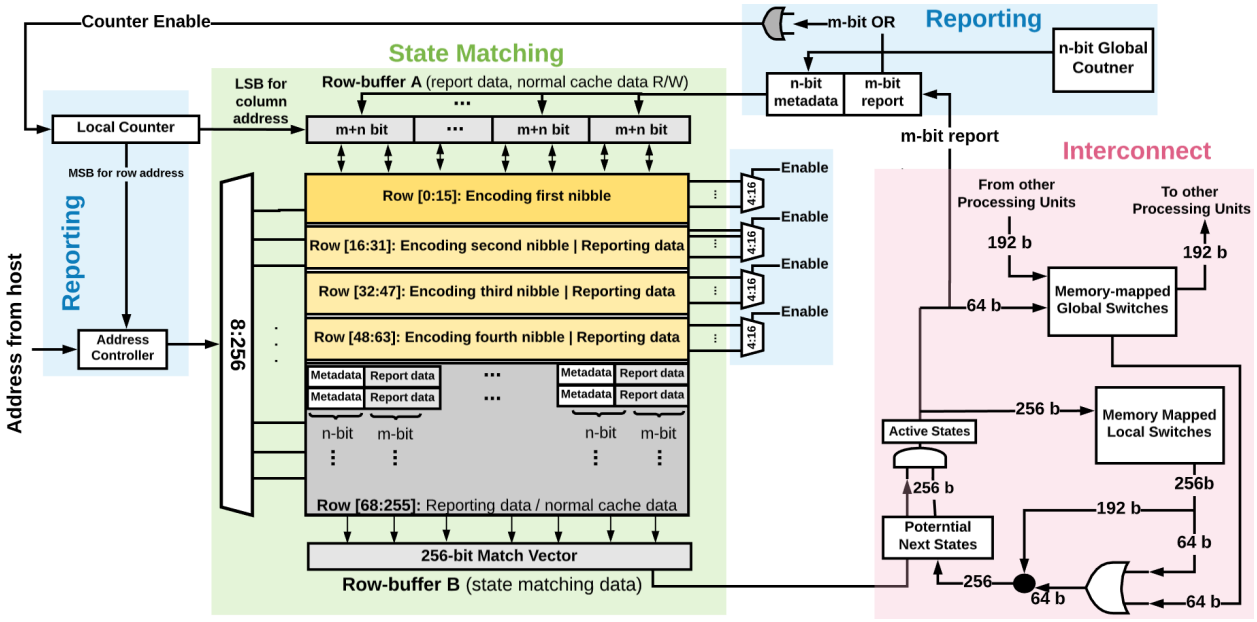

Figure 6.3: Sunder architecture for state matching (green), state transition or interconnect (pink), and reporting (blue). The state matching subarray is repurposed for storing reporting data in addition to storing matching data.

memory-mapped switches to provide inter PU connecting when processing for automata with more than 256 states. The green region depicts one memory subarray of size 256×256 (a conventional subarrays size in an L3 cache [119]), where matching symbols are encoded in the yellow region (upper rows of the memory subarray) and the reporting data is stored in the gray region (lower rows of the memory subarray) or partially in the yellow region (depends on the configured processing rate). *In prior solutions, one memory subarray of size 256×256 was used to encode 8-bit symbols. In this work, we utilize a similar size memory subarray to encode up to 16-bit symbols and to store the reporting data (up to 60Kb)! This is achieved only at the expense of 2% hardware overhead (i.e., the blue regions in Figure 6.3).*

To perform state matching and storing reporting data in the same subarrays in one cycle, we utilize dual-port SRAM subarrays (i.e., two sets of sense amplifiers (SA) and two sets of decoders). The SAs on the bottom are used to read the state matching data from up to the first 64 rows and are working with the four decoders on the right of the subarray. The SAs on the top work with the left decoder (8:256) and are used to read/write the reporting data and write the state matching data in the *Automata Mode (AM)*, and also read/write normal cache data in the *Normal Mode (NM)*.

Reconfigurable Nibble Processing

Different from all prior work, Sunder supports a reconfigurable symbol processing rate (i.e., 4-bit, 8-bit, and 16-bit symbols per cycle). This is unlike Impala [1], where the processing rate is fixed in hardware (i.e., if the

hardware is designed for 16-bit processing, the 8-bit processing is not able to utilize half of the subarrays). Each state is encoded in one memory column by embedding multiple 4-bit symbols. The processing rate can be determined by the user based on the application size and application required throughput. If the application is small, automata can be transformed to process more nibbles in one cycle, which results in higher throughput at the expense of utilizing unused hardware resources. On the other hand, if the application is large and several rounds of reconfiguration would be needed to process the entire set of rules/automata, a smaller processing rate that avoids overhead from extra states can be selected to optimize for space.

In the state matching subarray, Row[0:15] encodes the first nibble, Row[16:31] encodes the second nibble, Row[32:47] encodes the third nibble, and Row[48:63] encodes the fourth nibble of the symbol. When the processing rate is 4 bits per cycle, only the first 16 rows are used to encode the 4-bit symbols using a one-hot encoding scheme; thus, only the associated decoder to the first 16 rows will be enabled. This means the remaining rows (i.e., Row[16:255]) are used for storing the reporting data or normal cache data. Likewise, in 8-bit processing, the first 32 rows are used to encode two nibbles; thus, the first two decoders are enabled, and the remaining rows can be used for reporting data or normal cache data. Finally, when the processing rate is 16 bits per cycle, Row[0:63] will be used for state encoding (for four nibbles) and their associated decoders will be enabled accordingly. In this scenario, Row[64:255] can be used to store the reporting data or normal cache data.

The partial state matching results from nibbles are combined using bitwise operations with multi-row activation of SRAM arrays. For example, for the 16-bit processing, four memory rows are activated (with the four 4:16 decoders), then their matching results are bitwise ANDed to generate the final matching results. Jeloka et al. [120] have shown the stability of simultaneously activating 64 wordlines on SRAM subarrays by lowering the wordline voltage and verified this across 20 fabricated chips.

Memory Cell Structure

The memory cell used in the memory subarray for state matching/reporting and memory-mapped interconnect are 8T SRAM switch cells, following the 8T cross-point design proposed in [30]. An 8T cell consists of a 6T SRAM cell and two additional transistors, which connect the cell to a bitline. This allows a 6T cell to drive the bitlines only when the cell holds '1' and the input signal derived by the row decoder is '1'.

An 8T SRAM cell read operation starts by pre-charging the bitline; then evaluation is done by two serial transistors, one derived by *enable bit* value and the other (i.e., activator) by the wordlines of the 4:16 decoders (Figure 6.3, right-side decoders). The WWL wordlines are derived by the left-side decoder (8:256) for normal read/write operations. If the cell value of an active cell is '1', then the bitline will be discharged to '0' (NOT of the stored value); otherwise, it keeps its charge and reads as '1'. If two or more rows are activated, then

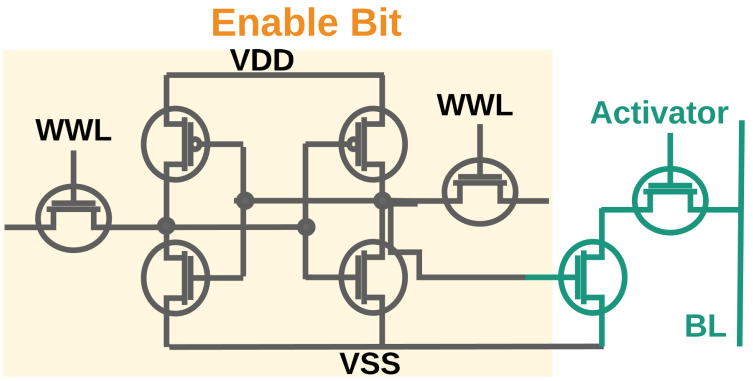

Figure 6.4: 8T SRAM cell

the pre-charged bit-line can be discharged to the ground if any of the active rows stores '1' in its cell. In other words, it computes the NOR of row activated cells in the same column. Sunder benefits from the wired-NOR functionality enabled by 8T cells for the multi-row activation in state matching and interconnect design.

Transforming to Nibble Processing

To make this transformation consistent with the Sunder architecture, preserve the functionality intact and lossless (i.e., avoiding false positives), and be able to process 1-nibble, 2-nibbles, or 4-nibbles in a memory column, we need to take another step and refine the states by splitting them to simpler matching rules. The refining stage (splitting to multiple states if needed) is done by making the parents and children of the original state as parents and children of all the replacing states. If the original state also has a self-loop, all new states will also be connected to each other and self-looped.

To convert conventional 8-bit processing (one hot encoding 8-bit symbols in a column with 256 cells) to 8-bit nibble processing (two 4-bits, each requires 16 cells in memory column), we transform the 8-bit based matching function to a list of two-term AND functions as in Equation 6.1. The g functions work only with the first four bits of the input (first nibble), and the h functions work with the second nibble, and all are combined with OR (sum of products).

$$\begin{aligned} f(x_0, x_1, ..., x_7) = &[g_0(x_0, x_1, x_2, x_3) \wedge h_0(x_4, x_5, x_6, x_7)] \\ &\vee [g_1(x_0, x_1, x_2, x_3) \wedge h_1(x_4, x_5, x_6, x_7)] \\ &\vee ... \vee [g_n(x_0, x_1, x_2, x_3) \wedge h_n(x_4, x_5, x_6, x_7)] \end{aligned} \tag{6.1}$$

This implies that a state with 8-bit symbols is converted to several states (i.e., several product terms), each processing two 4-bit symbols. The number of states is increased to preserve the correct functionality.

Each of the product terms represents a state and is implemented in a single memory column by converting the AND operation to NOR using DeMorgan's laws.

In Sunder's state matching subarray (Figure 6.3), Row[0:15] encodes $\neg g_0(x_0, x_1, x_2, x_3)$ and Row[16:31] encodes $\neg h_0(x_4, x_5, x_6, x_7))$. It then uses the multi-row activation feature of 8T memory cells to implement the NOR operation to compute the final matching signal. At run-time, to process the 8-bit input symbol, the first four bits of the input symbol are used to activate a row from the first 16 memory rows, and at the same time, the second four bits of the input symbol are used to select one row from the next 16 memory rows.

To reduce the overhead of state splitting, it is essential to minimize the number of product terms as much as possible. Sunder uses Espresso [99] - a CAD tool that was originally developed for logic minimization - to find the minimum symbol splits (two-level logic minimization is known to be an NP-complete problem). Espresso not only minimizes functions with boolean variables but also minimizes functions with n-values variables. Sunder uses this feature to express the original 8-bit matching function using two primitive variables, each with 16 distinct values (g's and h's in the previous example). The first variable is associated with (x_0, x_1, x_2, x_3) and the second variable is associated with (x_4, x_5, x_6, x_7). After specifying all the matching points of a state using these two variables in Espresso, it delivers a practically efficient number of Sum Of Product (SOP) terms. The number of product terms shows how many new states are required to split the original state.

6.5.2 Reporting Architecture

Sunder proposes to localize the reporting data within the very same memory subarrays performing the state matching, with minimal hardware overhead. This helps to avoid long wires from report states to buffers and their likely latency and routing congestion. It also helps to share many of the report buffer peripherals with the existing state-matching logic. Thanks to the nibble processing technique, which exponentially saves the memory footprint in the state matching subarrays, and the choice of dual-port 8T cells to isolate read port from write port, Sunder is able to store the reporting data in each cycle at the bottom rows of the state-matching subarrays. Sunder introduces several unique features, which can greatly reduce the overhead of reporting.

Report Storing Mechanism

Assume the processing rate is 16-bit; therefore, the first 64 rows in the memory subarrays in Figure 6.3 are used for encoding the states. We assume m reporting states in each memory subarrays, and we map the reporting states in an automaton to the m-reporting-enabled states in the memory subarrays, which are

the last m memory columns. At run-time, in the automata-mode, after the current active states have been calculated, we check if there is any reporting data is generated. This is done by ORing the m-bit reporting states driven from the *active state vector* (Figure 6.3 - blue region). If at least one report is generated, we then need to store this information along with the current cycle in which the report has happened. The cycle count is generated from a global counter in the hardware. Therefore, we concatenate the cycle number as the metadata to the reporting data, and write it in the reporting region of the subarray in a compact way.

As 8T cells have different ports for read and write, the state matching phase and reporting phase (from the previous cycle) can be pipelined. This approach does not need any additional hardware resources such as an arbiter or global buffer, as report information is locally stored in the same memory array as matching data has been stored. This way, accessing the report information is much easier, as it translates to simply reading data from memory. Reporting data and metadata are written row-wise, starting from row 64 in 16-bit processing (or starting from row 32 in 8-bit processing). To track the currently available location in the reporting region, simply a local counter is used. The counter size is calculated as:

$$\text{Local Counter size} = \lceil \log(\#ReportRows) \rceil + \left\lceil \log\left(\frac{256}{m+n}\right) \right\rceil \quad (6.2)$$

ReportRows is the number of rows configured for storing reporting data (i.e., in the 16-bit processing, up to 192 rows can be preserved for reporting data). m is the number of states in a subarray that can be a reporting state, and n is the global counter size. The MSB ($\lceil \log(\#ReportRows) \rceil$) is used for the address decoder to activate a row, and the LSB ($\left\lceil \log\left(\frac{256}{m+n}\right) \right\rceil$) selects the bitlines for the next available location in a row.

The address controller simply selects the address from the host in the normal read/write of the data to the subarray (either in normal cache mode, or writing the state matching data at the configuration time), or from the local counter in the Automata Mode. It also masks the row address depends on the number of reporting rows at the configuration time.

Reporting Architecture Highlights

Sunder introduces unique and useful features that have never been explored in previous works, and can greatly reduce the overhead of reporting in automata processing applications.

Report summarization: an important concern in the reporting architecture is the I/O cost. We observed that not all the applications required cycle-accurate report information (such as SPM [24]). All the previous accelerators are designed to read bulky cycle-accurate report information and post-process them

on the host. In Sunder, report summarization is achieved by performing the column-wise OR operation among report rows, thanks to wired-OR functionality of the 8T SRAM subarrays. This feature is beneficial for applications that have a very frequent reporting behavior, where the existence of a report in a specific duration matters. In other words, the user does not care about the specific cycle that the report has happened (Evaluation in Section 6.8.4).

Selective reporting: Sunder provides great freedom to the host to read the report status of every state at any cycle with a constant time while the conventional approaches fill the report buffers with report data that might not be interesting at that particular time, and this introduces more stalls to transfer reporting data.

Optimized for different reporting behaviors: when the application has a dense but infrequent reporting behavior, the reporting region has minimal usage; thus, it can be used to store normal cache data. On the other hand, when the application has a sparse but frequent reporting behavior, the reports are compacted in the report-storing subarrays, thus minimizing the need to stall the application.

FIFO strategy for the reporting buffers: our study on real-world applications reveals that they only generate at least one report in less than 12% of total execution cycles (see Table 6.1). This implies that in more than 88% of the cycles, no report is generated, and nothing will be written in the subarray. We take advantage of this observation and start reading the reporting data from the beginning of the reporting region. When the report buffer is full from the end, the reports will be written from the head of the buffer. If the report generation rate is higher than consumption and the report buffer is full, the execution is stalled.

6.5.3 Interconnect

The interconnect provides the functionality to move active states forward in time toward the next states. A state S gets activated if (1) the current symbol matches the state S and (2) any of its parents were activated in the previous cycle. The second condition implies that the interconnect should provide the OR-functionality, which is feasible with 8T SRAM switch cells. For Sunder, we use a memory-mapped full-crossbar interconnect based on 8T SRAM memory cells, as shown in Figure 6.5, to provide wired-OR functionally on bitlines. The left-side blue wordlines are driven by the left-side decoder (and are connected to WWL in 8T cells) for writing the connectivity data into the enable bits at the configuration time (Figure 6.4). The right-side purple wordlines are driven *active state vector* (see Figure 6.3), which determines the currently active states, and are connected to the *activators* in 8T cells. The bitlines (columns) drive the same set of states (one column per state).

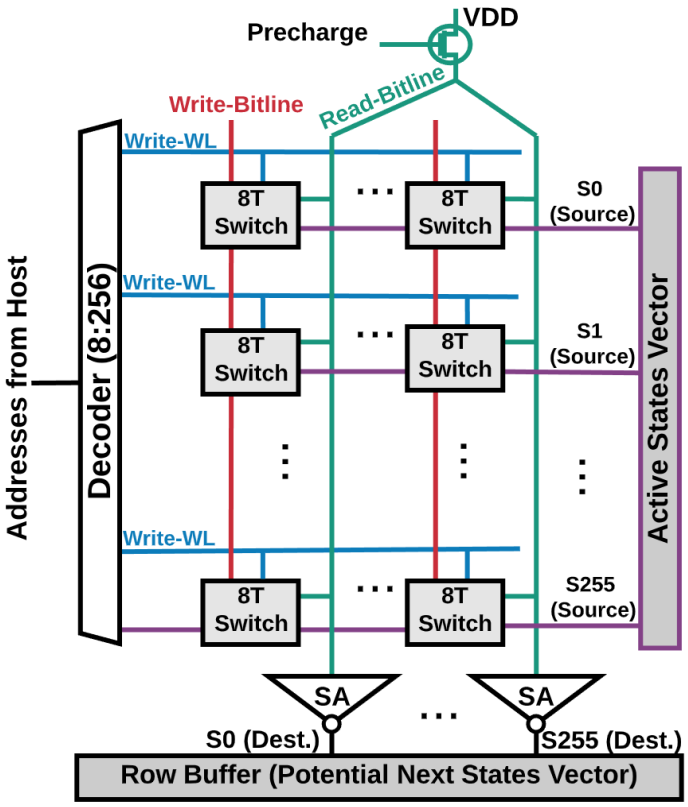

Figure 6.5: Memory-mapped local & global interconnect

This arrangement accommodates connection among every pair of 256 states, as every column intersects with every row, thus, avoids any congestion even for highly connected automaton. The memory cell at row i and column j stores '1' if there is an edge between state i and state j. The global memory-mapped interconnect provides the intra-PU connectivity to support larger automata with more than 256 states.

6.6 System Integration

Sunder can be realized by repurposing the last level cache (LLC) of recent processors, such as Intel Xeon with large L3 cache capacity (as Sunder's subarrays have the same size as a conventional L3 cache [119]). In the Sandy Bridge microarchitecture, the LLC is split into independent slices (usually equal to the number of cores), connected via a ring topology. To access the target slice, the physical address should go through a hashing block. This hash block distributes addresses uniformly across the slices with the granularity of cache lines, so it is possible to have two consecutive memory addresses mapped to two different memory arrays in different slices. However, to configure Sunder, we need flat accesses to certain arrays. Intel has not published the details of the hashing function. To find physical addresses for the slices that are repurposed for Sunder,

we can use the existing efforts to reverse engineer the hash function [121]. Inside a slice, to access the correct memory array in a certain cache way, we use Cache Allocation Technology (CAT) [122] to restrict the ways accessed by the program.

To cover all the addresses in a certain slice, we can set the page size to 1GB using *mmap* at configuration time. To translate the virtual addresses of the arrays to the physical addresses, page table information in */proc/self/pagemap* can be used. Automata are configured in the cache by writing configuration values at these addresses. At runtime, to collect the report information, the host application issues load instruction at the regions assigned to the report arrays for immediate processing or issues *clflush* to store the report data into the DRAM for post-processing. Farshin et al. [123] showed that the latency of accessing LLC slices changes based on the physical placement of the slices and cores in the ring topology. They have introduced a technique to discover the core with the minimum access latency to a given slice. Their approach can be utilized in Sunder integration to pin the host software to the CPU core and to reduce access latency to the target slice.

6.7 Evaluation Methodology

NFA workloads: we evaluate our proposed claims and architectures using ANMLZoo [44] and Regex [94] benchmark suites. They represent a set of diverse applications, including machine learning, data mining, and network security. AutomataZoo [95] is mostly an extension of ANMLZoo (9 out of 13 applications are the same), and the difference is that ANMLZoo is normalized to fill one AP chip. To provide a fair comparison with the AP, we use ANMLZoo benchmarks. A summary of the applications is represented in Table 6.1.

Table 6.2: Subarray parameters for state-matching and interconnect (overhead of peripherals are included) in 14nm technology.

Usage	Cell Type	Size	Delay (ps)	Read Power (mW)	Area (μm^2)
State-matching (Impala)	6T	16×16	180	0.58	453
State-matching (CA)	6T	256×256	220	5.52	9394
Interconnect (CA, Impala, Sunder) State-matching (Sunder)	8T	256×256	150	6.07	20102

Experimental setup: we use our open-source in-house Automata compiler and simulator[1] to perform the preprocessing steps, and simulate Sunder, Cache Automaton (CA) [30], Impala [1], and the Automata Processor (AP) [55], and also to perform the automata transformation for nibble processing. The simulator

[1]The reference to the tool is omitted due to the double-blind review.

takes automata in ANML format and processes the input cycle-by-cycle. Per-cycle statistics are used to calculate the number of active states, the number of reports, and communication overhead. To estimate area, delay, and power of the memory subarray in Sunder, Cache Automaton, and Impala model, we use a standard memory compiler (under NDA) for the 14nm technology node and nominal voltage 0.8V (details in Table 6.2). For example, Impala uses SRAM subarrays of size 16×16 with 6T cells for state matching, or Sunder uses SRAM Subarrays of size 256×256 with 8T cells for both state matching and the interconnect. The global wire-delays are calculated using SPICE modeling in CA. In the memory compiler, the 8T-cell design has wider transistors than 6T; therefore, 8T subarrays are faster and have higher area overhead than 6T subarrays.

Reporting architecture: Impala and CA overlook the real cost of reporting, and they mainly evaluate the matching kernel. To provide a fair and thorough comparison across different solutions, we assume the AP-style reporting architecture for the CA and Impala.

Parameter selection: on average, 3.9% of the states are the reporting states (Table 6.1, fifth column). Therefore, on average, 10 out of 256 states (3.9%×256) are reporting states. Based on this observation, we allocate 12 bits for the reporting data and 20 bit for metadata (i.e., the global counter to count for 1Mb of input data), depicted in Figure 6.3. To allow for capturing the reporting information for larger input, the input stride value is concatenated with all zeros in the reporting data and is written in the *metadata + report data* region.

6.8 Performance Evaluation

6.8.1 State and Transition Overhead

This section discusses the state and transition overhead for different processing rate (i.e., 1, 2, and 4-nibble processing). Figures 6.4 shows the number of states and transitions in each bitwidth, normalized to the number of states and transition in the original 8-bit design. We observed that benchmarks with higher symbol density (i.e., states that accept larger alphabet), such as Brill, EntityResolution, Hamming, Protomata, and RandomForest, have higher state and transition overhead in different bitwidths.

On average, 1, 2, 4-nibble designs have 3.1×, 1.0×, and 1.2× more states and 4.5×, 1.0×, and 1.8× more transitions over the original 8-bit designs. The increase in the number of states translates to utilizing more memory-column resources in in-memory designs. The increase in the number of transitions translates to utilizing more switches in our memory-mapped full crossbar interconnect (Figure 6.5) and does not incur extra resource overhead.

Table 6.3: Number of state and transitions in Sunder normalized to the original 8-bit automata.

Benchmark	Sunder State			Sunder Transition		
	1-nibble (4-bit)	2-nibble (8-bit)	4-nibble (16-bit)	1-nibble (4-bit)	2-nibble (8-bit)	4-nibble (16-bit)
Brill	5.3×	1.0×	1.9×	11.9 ×	1.0×	1.8×
Bro217	2.0×	1.0×	1.0×	2.1 ×	1.0×	7.4×
Dotstar03	2.2×	1.0×	1.0×	2.6 ×	1.0×	1.1×
Dotstar06	2.3×	1.0×	1.0×	3.0 ×	1.0×	1.1×
Dotstar09	2.4×	1.0×	1.0×	3.5 ×	1.0×	1.2×
ExactMatch	2.0×	1.0×	1.0×	2.0 ×	1.0×	1.0×
PowerEN	2.3×	1.0×	1.1×	3.1 ×	1.0×	1.0×
Protomata	6.0×	1.0×	1.2×	12.5 ×	1.0×	1.1×
Ranges05	2.0×	1.0×	1.0×	2.1 ×	1.0×	1.0×
Ranges1	2.1×	1.0×	1.0×	2.2 ×	1.0×	1.0×
Snort	2.5×	1.0×	1.1×	3.8 ×	1.0×	1.4×
TCP	2.5×	1.0×	1.1×	3.9 ×	1.0×	1.3×
Hamming	6.5×	1.1×	1.3×	9.7 ×	1.1×	1.4×
Levenshtein	2.8×	1.1×	2.2×	1.9 ×	1.1×	3.5×
Fermi	2.2×	1.0×	1.0×	2.1 ×	1.0×	1.3×
RandomForest	5.3×	1.0×	1.0×	9.4 ×	1.0×	1.0×
SPM	2.7×	1.1×	2.3×	2.7 ×	1.1×	4.6×
EntityResolution	3.2×	0.7×	0.9×	2.8 ×	0.7×	1.6×
Average	3.1×	1.0×	1.2×	4.5 ×	1.0×	1.8×

This means that compared to the original 8-bit processing, 4-nibble processing (or 16-bit processing) provides 2× throughput benefit only at the expense of 1.2× more memory columns. Moreover, 4-nibble processing requires 64 (4×2^4) memory rows to encode four 4-bit symbols, whereas the original 8-bit processing requires 256 (2^8) memory rows to encode one 8-bit symbol. Sunder opportunistically utilized the 192 (256-64) unused memory rows to store the reporting data with a simple and compact solution. This confirms that our algorithm/architecture methodology provides throughput and area benefits compared to prior works.

6.8.2 Performance Overhead Analysis for Reporting

Table 6.5 summarize the reporting overhead for Sunder (with and without FIFO strategy - see Section 6.5.2) and the AP. For all these architectures, automata matching computations and communication happens within a single cycle. Therefore, the "nominal" time it takes for the matching kernel (i.e., only state matching and transition) to run automata on the input symbol stream is equal to the symbol cycle time of the device multiplied by the number of symbols in the input symbol stream. To have apples-to-apples comparison across different solutions, we assume the AP-style reporting architecture (Figure 6.1) for both Impala and CA (as they overlook the reporting overhead), and add the reporting overhead to the "nominal" kernel execution cycles in CA and Impala.

Performance overhead: the number of flushes is the total number of times an application needs to flush the whole reporting region due to overflow. While a subarrray is flushing out the reporting data, the

Table 6.4: Number of state and transitions in Sunder normalized to the original 8-bit automata.

Benchmark	Sunder State			Sunder Transition		
	1-nibble (4-bit)	2-nibble (8-bit)	4-nibble (16-bit)	1-nibble (4-bit)	2-nibble (8-bit)	4-nibble (16-bit)
Brill	5.3×	1.0×	1.9×	11.9 ×	1.0×	1.8×
Bro217	2.0×	1.0×	1.0×	2.1 ×	1.0×	7.4×
Dotstar03	2.2×	1.0×	1.0×	2.6 ×	1.0×	1.1×
Dotstar06	2.3×	1.0×	1.0×	3.0 ×	1.0×	1.1×
Dotstar09	2.4×	1.0×	1.0×	3.5 ×	1.0×	1.2×
ExactMatch	2.0×	1.0×	1.0×	2.0 ×	1.0×	1.0×
PowerEN	2.3×	1.0×	1.1×	3.1 ×	1.0×	1.0×
Protomata	6.0×	1.0×	1.2×	12.5 ×	1.0×	1.1×
Ranges05	2.0×	1.0×	1.0×	2.1 ×	1.0×	1.0×
Ranges1	2.1×	1.0×	1.0×	2.2 ×	1.0×	1.0×
Snort	2.5×	1.0×	1.1×	3.8 ×	1.0×	1.4×
TCP	2.5×	1.0×	1.1×	3.9 ×	1.0×	1.3×
Hamming	6.5×	1.1×	1.3×	9.7 ×	1.1×	1.4×
Levenshtein	2.8×	1.1×	2.2×	1.9 ×	1.1×	3.5×
Fermi	2.2×	1.0×	1.0×	2.1 ×	1.0×	1.3×
RandomForest	5.3×	1.0×	1.0×	9.4 ×	1.0×	1.0×
SPM	2.7×	1.1×	2.3×	2.7 ×	1.1×	4.6×
EntityResolution	3.2×	0.7×	0.9×	2.8 ×	0.7×	1.6×
Average	3.1×	1.0×	1.2×	4.5 ×	1.0×	1.8×

symbol processing for the whole application is stalled. The reporting overhead (the stalls due to gathering and sending the reporting data to the host) represents the slowdown over the nominal execution cycles. Some benchmarks have little or no reporting overheads, even on the AP-style reporting architecture (e.g., Dotstar, Ranges, ClamAV). This is simply because these benchmarks report infrequently or not at all (Table 6.1). Some benchmarks incur extremely large reporting overheads on the AP-style reporting. For example, Snort incurs a 46× slowdown over ideal performance, and 7 out of 19 benchmarks spend more time processing reporting overheads than processing automata transitions!

As expected, Sunder reporting architecture incurs negligible overhead, i.e., less than 1.06× slowdown with no FIFO design, and this can even further decrease to less than 1.03× when applying the FIFO strategy (which reads from the beginning of the report array during the application execution). This simply means that Sunder's end-to-end performance is almost equal to the kernel performance, and minimizing the data movement overhead between CPU and memory is an ultimate mission in processing-in-memory architectures! Technically, Sunder does not incur stalls during the execution of an application due to reporting, can be used for real-time processing with a reliable and predictable performance.

SPM has extremely high-frequency reporting behavior and is the only application that has reporting overhead in Sunder architecture (3% reporting overhead - 6th column in Figure 6.5). Interestingly, the SPM application mostly requires to know if a single report has happened for specific input intervals with no interest in knowing the exact cycles that report events have occurred. This means that our report summarizing

Table 6.5: Reporting overhead for four nibble processing.

		Sunder w/o FIFO		Sunder w/ FIFO		AP
Benchmark	**#Subarrays**	**#Flushes**	**Reporting Overhead**	**#Flushes**	**Reporting Overhead**	**Reporting Overhead**
Brill	167	666	1.04×	0	1×	7.07×
Bro217	16	0	1×	0	1×	1.6×
Dotstar03	47	0	1×	0	1×	1×
Dotstar06	49	0	1×	0	1×	1×
Dotstar09	49	0	1×	0	1×	1×
ExactMatch	49	0	1×	0	1×	1×
PowerEN	288	0	1×	0	1×	1.1×
Protomata	197	0	1×	0	1×	5.8×
Ranges05	49	0	1×	0	1×	1×
Ranges1	49	0	1×	0	1×	1×
Snort	347	1	1.01×	0	1×	46×
TCP	77	0	1×	0	1×	3.8×
ClamAV	194	0	1×	0	1×	1×
Hamming	44	0	1×	0	1×	1×
Levenshtein	11	0	1×	0	1×	1×
Fermi	200	0	1×	0	1×	2.3×
RandomForest	138	0	1×	0	1×	1.6×
SPM	419	9212	1.06×	3870	1.03×	9.7×
EntityResolution	372	0	1×	0	1×	2.25×

technique can further reduce the reporting overhead for the applications with extremely high reporting behavior. For application with low reporting frequency (e.g., Hamming, Levenshtein, ClamAV, Dotstar - Table 6.1), the reporting region can be mostly used for the normal cache data (i.e., row 16 forward in Figure 6.3).

6.8.3 Comparison with Prior Work

Overall performance: The delays and frequencies of different pipeline stages for Sunder, Impala, CA, and the AP are shown in Table 6.6 (derived from Table 6.2). Sunder uses memory subarrays with 8T cells for both state matching and interconnect. Sunder, CA, and Impala have similar hierarchical interconnect designs, and both local and global switches are evaluated in parallel (Figure 6.3). Following CA design, we assume the SRAM-based CA design slice of $3.19mm \times 3mm$. Therefore, the distance between SRAM arrays and global switch is assumed to be $1.5mm$. From SPICE modeling, the wire delay was found to be $66ps/mm$; therefore, the wire delay for global switches is $99ps$. Global switch delay for CA and Impala is 249 ps, composed of read-access latency (150ps) and wire delay latency (99ps). Impala state-matching subarray is ~5X less; therefore, we assume $20ps$ wire-delay for Impala. Therefore, the global switch delay for Impala is 170 ps (150ps+20ps).

The frequency is determined based on the slowest pipeline stage. We assume the operating frequency to be 10% less than what we have calculated to consider potential estimation errors. The AP is designed in

Table 6.6: Pipeline stage delays and operating frequency. The detail implementation of the AP is not publicly available.

Architecture	State Matching	Local Switch	Global Switch	Max Freq. (GHz)	Operating Freq. (GHz)
Sunder (14nm)	150 ps	150 ps	249 ps	4.01	3.6
Impala (14nm)	180 ps	150 ps	170 ps	5.55	5
CA (14nm)	220 ps	150 ps	249 ps	4.01	3.6
AP (50nm)	-	-	-	0.133	0.133
AP (14nm)*	-	-	-	1.69	1.69

* Projected to 14nm

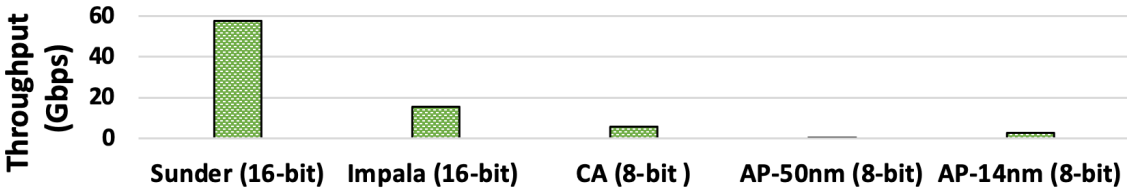

Figure 6.6: Throughput of different spatial automata accelerators.

50nm DRAM technology. To have a fair comparison, we project the frequency to 14nm technology, which is an ideal assumption.

The overall performance (i.e., throughput) of in-memory automata processing architectures is determined by $\frac{frequency \times (bits/cycle)}{reporting-overhead}$. This is unlike prior work that calculates $frequency \times (bits/cycle)$ as the overall performance and overlooks the reporting overhead. The $reporting - overhead$ is the average reporting overhead in Table 6.5, and is equal for CA, Impala, and AP. Impala has a fixed 16-bit per cycle processing rate, whereas Sunder has a reconfigurable 16-bit per cycle processing rate. CA and AP design only work with 8-bit per cycle rate. Figure 6.6 shows that Sunder achieves 280×, 22×, 10×, and 4× higher throughput than the AP (50nm), AP (14nm), CA, and Impala, respectively. This benefit comes from the fact that Sunder has almost no reporting overhead (i.e., reporting does not cause a slowdown in performance), which can provide a deterministic throughput of one input symbol per cycle!

Area Overhead: Figure 6.7 compares the area overhead of state-matching, interconnect, and reporting of Sunder with Impala, CA and, and the AP (all in 14nm) for 32K STEs. Impala uses four 16×16 SRAM subarrays (6T cells) for the state matching, thus, has the minimum area overhead for this stage. Impala and CA reporting overhead are modeled after the AP reporting architecture. In Sunder, the reporting architecture is infused in the state matching subarray, and there is only an additional 2% overhead for the addition circuitry (i.e., blue area in Figure 6.3). Both state matching and interconnect switches in Sunder are designed with 8T SRAM subarrays, which is 2.1× larger than the 6T subarrays (Table 6.2). Overall, Sunder has 2.1×, 1.6×, and 1.5 lower area overhead than AP, Impala, and CA, respectively. This benefit comes from the compact and in-place reporting architecture enabled by algorithmic transformation. Moreover, Sunder incurs

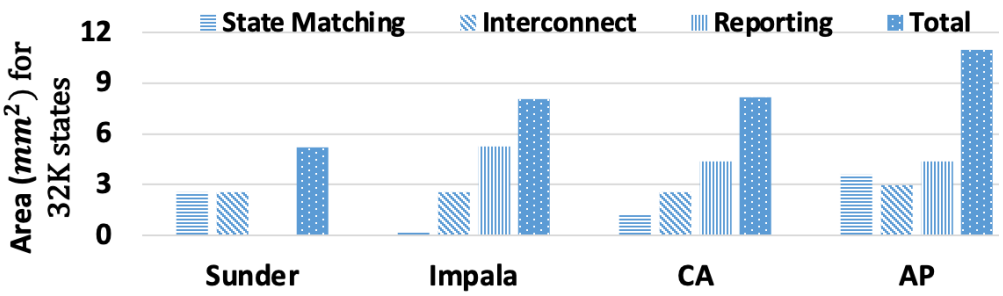

Figure 6.7: Comparing area overhead for 32K STEs.

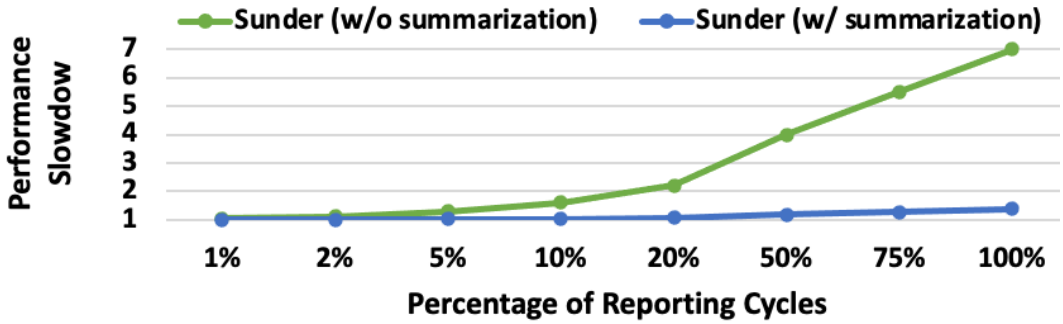

Figure 6.8: Performance slowdown for various reporting rates.

almost no performance penalty for the reporting, while the other solutions cause up to 46× slowdown due to stalls for reporting.

6.8.4 Input Stream Sensitivity Analysis

Variations on the input steam change the reporting behavior. To evaluate this, we perform a sensitivity analysis on the percentage of reporting cycles, sweeping from 1% to 100%. We assume 12 reporting states in each subarray (based on the analysis in Table 6.1). Figure 6.8 represents the performance slowdown with and without summarization technique (Section 6.5.2). As expected, Sunder reporting architecture incurs negligible performance overhead when the reporting cycles are less than 5%. In the absolute worst-case scenario, which is reporting 100% of the times, Sunder with no summarizing incurs only 7× performance overhead. However, if the application only needs to know the occurrence of the report, then Sunder can summarize the reporting region in 16-row batches, which improve the performance overhead to only 1.4×. However, the AP-style reporting incurs up to 46× slowdown with only 3.24% of report cycles (SPM in Table 6.1).

6.9 Conclusions

We introduce Sunder, a fully reconfigurable, efficient, and low overhead in-SRAM pattern processing accelerator. Sunder integrates our analysis of the prior architectures and sources of inefficiencies, and our study of the static structure and dynamic behavior of real-world applications, to implement the next-generation of in-memory automata processing. Transforming an automaton for better hardware utilization exponentially reduces memory usage and increases information density. This frees up space in the memory subarrays and creates an opportunity to store the reporting data locally in each subarray to significantly reduce the host communication and stabilize the processing throughput across the execution of an application. Sunder's reporting architecture incurs less than 2% hardware overhead (as the reporting data are co-located in the state matching subarrays and use shared resources). On average, Sunder has two orders of magnitude higher throughput than Micron's AP and one order to magnitude higher throughput than the state-of-the-art SRAM-based solutions.

Our flexible and compact memory-mapped interconnect solution avoids congestion, even for highly connected components. SRAM implementation (1) directly enables a compact memory-mapped interconnect by allowing non-destructive multi-row activation with 8T SRAM cells, (2) facilitates multi-symbol processing to increase the throughput, and (3) increases the processing frequency up to 4GHz. As a result, our software/hardware methodology enables three orders of magnitude higher throughput per unit area compared to the Micron's AP, and this low-cost, high-throughput solution hopefully shows a path toward commercial viability and unlocks the full potential of automata processing by making it accessible to an increasing set of pattern processing applications with real-time requirements.

Chapter 7

Conclusions and Broader Impacts

7.1 Dissertation Conclusion

This dissertation focuses on the design and evaluation of efficient and low-cost memory-centric accelerators for finite automata processing. Accelerating finite automata processing benefits regular-expression workloads, such as network intrusion detection and virus scanning [33], and a wide range of other applications that do not map obviously to regular expressions, including pattern mining [27, 25, 24, 124, 42], bioinformatics [39, 125], natural language processing [126, 29], and machine learning [40].

This dissertation first proposes, APSim, an open-source automata processing simulator and a compiler that simulates, minimizes, compresses, and transforms a set of automata to different symbol sizes, and efficiently maps them to the hardware resources in memory-centric architectures, such as FPGAs, Cache Automaton [30], eAP [4], etc. It receives the input automata based on the standard representations such as ANML and provide a extensive set of transformations/minimizations with its enrich APIs. APSim provides a parameterizable environment to experiment with bitwidth sizes on various platforms. A set of bitwise and symbol-wise automata transformation and minimization algorithms are developed to enable these explorations. APSim addresses the state and transition placement in an automaton by converting it to the subgraph isomorphism problem, and applies the genetic algorithm approach to maps states/transitions to their physical locations. APSim has enabled several architectural research for automata interconnect design [3, 56], universal mapping from application with any alphabet size to hardware accelerator [66], multi-symbol processing exploration on in-memory architectures [69], and multi-symbol processing on FPGAs [2].

This dissertation then introduces *Grapefruit*, a publicly available framework for automata processing on FPGAs. Grapefruit consists of two main components: (1) an integrated compiler for automata simulation,

minimization, transformation, and optimization, and (2) an HDL generator that produces a full-stack design for a set of automata to be processed on FPGAs. We develop an efficient reporting architecture and pipeline design, along with a set of hardware optimizations and parameters. Our framework allows researchers to investigate frequency and resource-usage trade-offs and provides an easy-to-understand and easy-to-modify code for them to explore new ideas. Grapefruit provides up to 80% higher frequency than prior works that are not fully end-to-end and 3.4× higher throughput in a multi-stride solution than a single-stride solution. An interesting direction for future work would be automatic learning-based parameter tuning using static and dynamic behavior of automata in an application.

This dissertation then identifies two observtions in the existing memory-centric architectures for automata processing: (1) all these architectures are based on 8-bit symbol processing (derived from ASCII), and our analysis on a large set of real-world automata benchmarks reveals that the 8-bit processing dramatically underutilizes hardware resources, and (2) multi-stride symbol processing, a major source of throughput growth, is not explored in the existing in-memory solutions. To address these issues, this dissertation then presents Impala, an in-memory accelerator for an efficient multi-stride automata processing by leveraging our observations. The key insight of our work is that transforming 8-bit processing to 4-bit processing exponentially reduces hardware resources for state-matching and improves resource utilization. This, in turn, brings the opportunity to have a denser design, and be able to utilize more memory columns to process multiple symbols per cycle with a linear increase in state-matching resources. Impala thus introduces three-fold area, throughput, and energy benefits at the expense of increased offline compilation time. Our empirical evaluations on a wide range of automata benchmarks reveal that Impala has on average 2.7× (up to 3.7×) higher throughput per unit area and 1.22× lower power consumption than Cache Automaton, which is the best performing prior work.

This dissertation then presents Sunder, a fully reconfigurable, efficient, and low overhead in-SRAM pattern processing accelerator. Sunder integrates our analysis of the prior architectures and sources of inefficiencies, and our study of the static structure and dynamic behavior of real-world applications, to implement the next-generation of in-memory automata processing. Transforming an automaton for better hardware utilization exponentially reduces memory usage and increases information density. This frees up space in the memory subarrays and creates an opportunity to store the reporting data locally in each subarray to significantly reduce the host communication and stabilize the processing throughput across the execution of an application. Sunder's reporting architecture incurs less than 2% hardware overhead (as the reporting data are co-located in the state matching subarrays and use shared resources). On average, Sunder has two orders of magnitude higher throughput than Micron's AP and one order to magnitude higher throughput than the state-of-the-art SRAM-based solutions.

Our flexible and compact memory-mapped interconnect solution avoids congestion, even for highly connected components. SRAM implementation (1) directly enables a compact memory-mapped interconnect by allowing non-destructive multi-row activation with 8T SRAM cells, (2) facilitates multi-symbol processing to increase the throughput, and (3) increases the processing frequency up to 4GHz. As a result, our software/hardware methodology enables three orders of magnitude higher throughput per unit area compared to the Micron's AP, and this low-cost, high-throughput solution hopefully shows a path toward commercial viability and unlocks the full potential of automata processing by making it accessible to an increasing set of pattern processing applications with real-time requirements.

The main insight is that the conventional 8-bit automata model does not provide the most efficient computation on memory-centric architecture for real-world automata applications. There are applications with smaller character-set, and their automata do not utilize the 8-bit supported hardware accelerators. On the other hands, there are applications with very large character-set, which cannot be mapped to the existing 8-bit automata accelerators. Moreover, to design the next generation of automata processor, 8-bit designs will not provide the best implementations for general automata processing. This research provides toolset and insights on (1) how the existing automata applications can fully utilize the existing architectures and (2) how to efficiently design future automata accelerators.

7.2 Potential Long-Term Impact

This section summarizes several potential long-term impacts of this dissertation.

7.2.1 Arithmetic Operations with Automata Accelerators

This dissertation enables a low-overhead and efficient in-memory automata processing solution. The computation of automata processing consists of iterative comparisons and routing, which naturally maps to memory-compatible processing and can leverage the extreme density and parallelism benefits intrinsic to memory. A potential broader impact of this work is to implement arithmetic operations using only memory lookups to take advantage of this simple yet powerful computation paradigm. These arithmetic operations can be laid out as automata on memory arrays to calculate more complex operations such as convolutions or a linear layer in a Deep Neural Network (DNN). Such a mapping has great potential for performance, area, and energy benefits for DNN accelerators (due to the simpler computation), especially for DNN processing in edge devices.

7.2.2 Re-evaluation of Rule-Based Methods

Traditional rule-based methods in various applications, such as NLP, image processing, and data mining, have proven expensive if the ruleset is large, but capturing corner cases often leads to many rules. However, rule-based methods can be re-evaluated with the existence of a high-throughput, flexible, and efficient automata accelerators. The main idea is that rules can be converted to regular expressions, which in turn, can be represented with finite automata. This dissertation presents a high-throughput and low-cost PIM solution, where increasing the number and complexity of the learned rules does not incur any performance overhead (up to the capacity of the hardware), and the ability to process many rules can improve the accuracy of rule-based methods and make them competitive with the state-of-the-art machine learning solutions [126].

7.2.3 Shifting Computing Paradigm

Moving from a compute-centric to a memory-centric processing paradigm requires both software (SW) and hardware (HW) considerations. With PIM (or any other specialization), we need to work with data structures that are closer to HW. This dissertation shows how re-thinking the SW data structure and data representation (i.e., moving from conventional SW-friendly 8-bit processing to PIM-friendly 4-bit processing) can greatly improve the PIM benefits. This observation can help other researchers when developing PIM architectures.

7.2.4 Logic Optimization

Moving processing functions into memory arrays to take advantage of their efficient lookup to perform computation is not a straightforward task, as we need to keep the same functionality while using less memory. Lookup-based computation has been studied before, but this dissertation shows how classic logic minimization CAD tools such as Espresso can be employed for this task. While we use this tool in the automata processing domain, this approach can be deployed for minimizing the representation in the lookup tables in different domains.

7.2.5 Open-Source Framework

Our APSim modeling framework[1] provides a comprehensive set of features, including automata simulation, transformation, minimization, compression, verification, mapping, and placement for PIM solutions. Moreover, it supports FPGA backends (e.g., our Grapefruit work [2] was selected as a best paper nominee in FCCM'20). The design is modular and parameterizable, which facilitates design-space exploration, and it is based on

[1] https://github.com/gr-rahimi/APSim

the standard NetworkX library, which provides an easy-to-modify and easy-to-enhance solution for other researchers.

7.3 Impacts on Industry

The idea of automata processing in memory was initiated at Micron in 2013 when Micron developed the Automata Processor (AP) [55] by re-purposing DRAM arrays. We believe this product did not get traction into the market for several reasons: (1) the choice of DRAM, which resulted in a relatively low processing frequency (133MHz), due to high DRAM access latency compared to FPGA solutions with frequency ranges between 160MHz to 300MHz, (2) inefficient state matching design, which is based on a fixed 8-bit processing rate and one-hot encoding scheme that causes low information density and waste of resources, (3) an inefficient hierarchical interconnect design which causes routing congestion, and in turn, decreases resource utilization, and (4) costly reporting architecture that decreases execution performance up to 46× [62].

This dissertation integrates our analysis of the AP architecture and sources of inefficiencies, and our study of the static structure and dynamic behavior of real-world applications, to implement the next-generation of in-memory automata processing. Key insights that we believe have more general applicability are the importance of optimizing the data structure for in-memory processing and transforming an automaton for better hardware utilization, which exponentially reduces memory usage and increases information density. This frees up space in the memory subarrays and creates an opportunity to store the reporting data locally in each subarray to significantly reduce the host communication and stabilize the processing throughput across the execution of an application. Our flexible and compact memory-mapped interconnect solution totally avoids congestion, even for highly connected components. We also realized that the choice of SRAM in place of DRAM (1) directly enables a compact memory-mapped interconnect by allowing non-destructive multi-row activation with 8T SRAM cells, (2) facilitates multi-symbol processing to increase the throughput, and (3) increases the processing frequency up to 5GHz. As a result, our software/hardware methodology enables more than 500× higher throughput per unit area compared to the Micron's AP, and this low-cost, high-throughput solution hopefully shows a path toward commercial viability and unlocks the automata processing full potential by making it accessible to an increasing set of pattern processing applications with real-time requirements, such as genome sequencing, network security, database queries, and virus scanning.

Bibliography

[1] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 86–98. IEEE, 2020.

[2] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020.

[3] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. Scalable and efficient in-memory interconnect architecture for automata processing. *IEEE Computer Architecture Letters*, 2019.

[4] 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 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 87–99. ACM, 2019.

[5] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[6] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. Debugging support for pattern-matching languages and accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[7] Micron. Micron automata processor. www.cs.virginia.edu/~skadron/grab/Skadron-Micron_AP_Briefing_Deck_13032014a.pdf.

[8] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[9] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 85–98, 2014.

[10] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[11] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[12] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348. IEEE, 2015.

[13] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[14] Fabrice Devaux. The true processing in memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–24. IEEE Computer Society, 2019.

[15] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 100–113, 2019.

[16] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287. IEEE, 2017.

[17] J Thomas Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot chips 23 symposium (HCS)*, pages 1–24. IEEE, 2011.

[18] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301. IEEE, 2017.

[19] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R Stan, and Kevin Skadron. Fulcrum: a simplified control and access mechanism toward flexible and practical in-situ accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 556–569. IEEE, 2020.

[20] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[21] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 383–396. IEEE, 2018.

[22] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492. IEEE, 2017.

[23] Indranil Roy and Srinivas Aluru. Discovering motifs in biological sequences using the micron automata processor. *IEEE/ACM transactions on computational biology and bioinformatics*, 13(1):99–111, 2016.

[24] 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*, pages 135–144. ACM, 2016.

[25] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Hierarchical pattern mining with the micron automata processor. In *International Journal of Parallel Programming (IJPP)*. 2017.

[26] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. 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*, pages 665–674. ACM, 2018.

[27] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. Frequent subtree mining on the automata processor: challenges and opportunities. In *International Conference on Supercomputing (ICS)*. ACM, 2017.

[28] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 737–748. IEEE, 2018.

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

[30] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[31] John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 2008.

[32] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463, 1999.

[33] Cong Liu and Jie Wu. Fast deep packet inspection with a dual finite automata. *IEEE Transactions on Computers*, 62(2), 2013.

[34] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2009.

[35] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102. ACM, 2006.

[36] Yi-Hua Yang and Viktor Prasanna. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Transactions on Computers*, 61(7):1013–1025, 2012.

[37] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. Hare: Hardware accelerator for regular expressions. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[38] Ke Wang et al. Association rule mining with the micron automata processor. In *IPDPS*. IEEE, 2015.

[39] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. Searching for potential gRNA off-target sites for CRISPR/Cas9 using automata processing across different platforms. In *24th International Symposium on High-Performance Computer Architecture*. IEEE, 2018.

[40] 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*, pages 200–218. Springer, 2016.

[41] Mateja Putic, AJ Varshneya, and Mircea R Stan. Hierarchical temporal memory on the automata processor. *IEEE Micro*, 37(1):52–59, 2017.

[42] 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*, pages 311–318. IEEE, 2016.

[43] Michael HLS Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. Using the automata processor for fast pattern recognition in high energy physics experiments—a proof of concept. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 832:219–230, 2016.

[44] Jack Wadden et al. ANMLZoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *IISWC*. IEEE, 2016.

[45] Kubilay Atasu, Florian Doerfler, Jan van Lunteren, and Christoph Hagleitner. Hardware-accelerated regular expression matching with overlap handling on ibm poweren processor. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1254–1265. IEEE, 2013.

[46] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*. ACM, 2007.

[47] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. Why gpus are slow at executing NFAs and how to make them faster. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 251–265, 2020.

[48] *iNFAnt2*. https://github.com/vqd8a/iNFAnt2.

[49] *DFAGE*. https://github.com/vqd8a/DFAGE.

[50] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. Fast support for unstructured data processing: the unified automata processor. In *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*, pages 533–545. IEEE, 2015.

[51] 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*, pages 1–8. IEEE, 2017.

[52] Rasha Karakchi, Lothrop O Richards, and Jason D Bakos. A dynamically reconfigurable automata processor overlay. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*, pages 1–8. IEEE, 2017.

[53] Xiang Wang. *Techniques for efficient regular expression matching across hardware architectures*. University of Missouri-Columbia, 2014.

[54] Ioannis Sourdis, João Bispo, Joao MP Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[55] Paul Dlugosch, Dean Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25(12), 2014.

[56] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. A scalable and efficient in-memory interconnect architecture for automata processing. *IEEE Computer Architecture Letters*, 2019.

[57] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. Architectural support for efficient large-scale automata processing. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–920. IEEE, 2018.

[58] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*, pages 200–218, 2016.

[59] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. In *International Symposium on Microarchitecture*, pages 259–272, 2017.

[60] Linley Gwennap. New chip speeds nfa processing using DRAM architectures. In *In Microprocessor Report*, 2014.

[61] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, 2014.

[62] Jack Wadden, Kevin Angstadt, and Kevin Skadron. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 749–761. IEEE, 2018.

[63] Vlastimil Košar and Jan Korenek. Multi-stride NFA-split architecture for regular expression matching using FPGA. In *Proceedings of the 9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, 2014.

[64] Lucas Vespa, Ning Weng, and Ramaswamy Ramaswamy. MS-DFA: Multiple-stride pattern matching for scalable deep packet inspection. *The Computer Journal*, 54(2):285–303, 2010.

[65] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-speed regular expression matching engine using multi-character NFA. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 131–136. IEEE, 2008.

[66] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *Submitted to 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[67] Elaheh Sadredini, Reza Rahimi, and Kevin Skadron. Enabling in-SRAM pattern processing with low-overhead reporting architecture. *IEEE Computer Architecture Letters*, 2020.

[68] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, , Reza Rahimi, Kevin Skadron, Weimer, and Reetu Das. Aspen: A scalable in-sram architecture for pushdown automata. In *51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'51)*. IEEE, 2018.

[69] Reza Rahimi, Elaheh Sadredini, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *Submitted to 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[70] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. Aspen: A scalable in-sram architecture for pushdown automata. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[71] Intel. https://github.com/01org/hyperscan.

[72] Google. Re2. https://github.com/google/re2.

[73] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: NFA pattern matching on gpgpu devices. *ACM SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[74] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of micron's automata processor. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on*, pages 1–3. IEEE, 2016.

[75] Jack Wadden et al. Anmlzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *in Workload Characterization (IISWC)*. IEEE, 2016.

[76] Victor Mikhaylovich Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 1961.

[77] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. Hawk: Hardware support for unstructured log processing. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 469–480. IEEE, 2016.

[78] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 461–472. IEEE Computer Society, 2012.

[79] Jack Wadden and Kevin Skadron. VASim: An open virtual automata simulator for automata processing application and architecture research. Technical report, Technical Report CS2016-03, University of Virginia, 2016.

[80] Jack Wadden, Samira Khan, and Kevin Skadron. Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 180–187. IEEE, 2017.

[81] Ted Xie, Vinh Dang, Chunkun Bo, Jack Wadden, Mircea Stan, and Kevin Skadron. An end-to-end reconfigurable engine for automata processing. In *50th Conference on Government Microcircuit Applications and Critical Technology (GOMACTech)*, 2018.

[82] Shufang Zhu, Geguang Pu, and Moshe Y Vardi. First-order vs. second-order encodings for ltlf-to-automata translation. *arXiv preprint arXiv:1901.06108*, 2019.

[83] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society, 1986.

[84] Tran Trung Hieu and Ngoc Thinh Tran. A memory efficient FPGA-based pattern matching engine for stateful nids. In *Ubiquitous and Future Networks (ICUFN), 2013 Fifth International Conference on*, pages 252–257. IEEE, 2013.

[85] 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. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(2):9, 2019.

[86] 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)*, pages 631–648, 2019.

[87] Michela Becchi and Patrick Crowley. A-DFA: A time-and space-efficient DFA compression algorithm for fast regular expression evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[88] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. Scalable algorithms for NFA multi-striding and NFA-based deep packet inspection on GPUs. *IEEE/ACM Transactions on Networking*, 24(3):1704–1717, 2016.

[89] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59. ACM, 2008.

[90] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154. ACM, 2007.

[91] Benjamin C Brodie, David E Taylor, and Ron K Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH computer architecture news*, 34(2):191–202, 2006.

[92] Tingwen Liu, Yifu Yang, Yanbing Liu, Yong Sun, and Li Guo. An efficient regular expressions compression algorithm from a new perspective. In *2011 Proceedings IEEE INFOCOM*, pages 2129–2137. IEEE, 2011.

[93] Wikipedia contributors. Espresso heuristic logic minimizer — Wikipedia, the free encyclopedia, 2019.

[94] Michela Becchi, Mark Franklin, and Patrick Crowley. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 79–89. IEEE, 2008.

[95] Jack Wadden et al. AutomataZoo: A modern automata processing benchmark suite. In *IISWC*. IEEE, 2018.

[96] Junqiao Qiu, Zhijia Zhao, and Bin Ren. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2016.

[97] Zhijia Zhao and Xipeng Shen. On-the-fly principled speculation for fsm parallelization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[98] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the embarrassingly sequential: parallelizing finite state machine-based computations through principled speculation. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[99] Richard L Rudell and Alberto Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, 1987.

[100] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, M Arif Rahman, and Mircea R. Stan. An overflow-free quantized memory hierarchy in general-purpose processors. *IEEE International Symposium on Workload Characterization*, 2019.

[101] Arun Subramaniyan and Reetuparna Das. Parallel automata processor. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 600–612. IEEE, 2017.

[102] Fatih Hamzaoglu, Yibin Ye, Ali Keshavarzi, Kevin Zhang, Siva Narendra, Shekhar Borkar, Mircea Stan, and Vivek De. Dual-v/sub t/sram cells with full-swing single-ended bit line sensing for high-performance on-chip cache in 0.13/spl mu/m technology generation. In *ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No. 00TH8514)*, pages 15–19. IEEE, 2000.

[103] Kevin Zhang, Uddalak Bhattacharya, Zhanping Chen, Fatih Hamzaoglu, Daniel Murray, Narendra Vallepalli, Yih Wang, B Zheng, and Mark Bohr. Sram design on 65-nm cmos technology with dynamic sleep transistor for leakage reduction. *IEEE Journal of Solid-State Circuits*, 40(4):895–901, 2005.

[104] Achiranshu Garg and Tony Tae-Hyoung Kim. Sram array structures for energy efficiency enhancement. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(6):351–355, 2013.

[105] Rui Liu, Xiaochen Peng, Xiaoyu Sun, Win-San Khwa, Xin Si, Jia-Jing Chen, Jia-Fang Li, Meng-Fan Chang, and Shimeng Yu. Parallelizing sram arrays with customized bit-cell for binary neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, page 21. ACM, 2018.

[106] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[107] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators. In *The International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020.

[108] Kevin Angstadt, Westley Weimer, and Kevin Skadron. Rapid programming of pattern-recognition processors. *ACM SIGOPS Operating Systems Review*, 50(2):593–605, 2016.

[109] Wikipedia contributors. Set cover problem — Wikipedia, the free encyclopedia, 2019. [Online; accessed 28-June-2019].

[110] Christopher Umans, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(7):1230–1246, 2006.

[111] Wikipedia contributors. Programmable logic array — Wikipedia, the free encyclopedia, 2019. [Online; accessed 28-June-2019].

[112] Linux programmer's manual for mmap. http://man7.org/linux/man-pages/man2/mmap.2.html. Accessed: 2019-07-15.

[113] Compute express link. https://www.computeexpresslink.org/.

[114] https://github.com/gr-rahimi/APSim.

[115] Veit B Kleeberger, Helmut Graeb, and Ulf Schlichtmann. Predicting future product performance: Modeling and evaluation of standard cells in finfet technologies. In *Proceedings of the 50th Annual Design Automation Conference*, page 33. ACM, 2013.

[116] Chunkun Bo, Ke Wang, Jeffrey J Fox, and Kevin Skadron. Entity resolution acceleration using micron's automata processor.

[117] II Tommy Tracy, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabriel Robins. Nondeterministic finite automata in hardware-the case of the levenshtein automaton.

[118] Jack Wadden. Virtual automata simulator - vasim. https://github.com/jackwadden/vasim.

[119] Wei Chen, Szu-Liang Chen, Siufu Chiu, Raghuraman Ganesan, Venkata Lukka, Wei Wing Mar, and Stefan Rusu. A 22nm 2.5 mb slice on-die l3 cache for the next generation xeon® processor. In *2013 Symposium on VLSI Circuits*, pages C132–C133. IEEE, 2013.

[120] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.

[121] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *International Symposium on Recent Advances in Intrusion Detection*, pages 48–65. Springer, 2015.

[122] Intel cache allocation technology. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html. [online; accessed November 23, 2020].

[123] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the most out of last level cache in intel processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[124] Ke Wang, Yanjun Qi, J.J. Fox, M.R. Stan, and K. Skadron. Association rule mining with the micron automata processor. In *Proc. of IPDPS'15*, May 2015.

[125] 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.

[126] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators. In *24th ACMSIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18)*. ACM, 2018.