2022

Techniques For Accelerating Large-Scale Automata Processing

Hongyuan Liu
William & Mary - Arts & Sciences, lliuhy@gmail.com

Techniques for Accelerating Large-scale Automata Processing

Hongyuan Liu

Jinan, Shandong, China

Bachelor of Engineering, Shandong University, 2013
Master of Science, University of Hong Kong, 2016

A Dissertation presented to the Graduate Faculty of
The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
January 2022

APPROVAL PAGE

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

Doctor of Philosophy

Hongyuan Liu

Approved by the Committee, January 2022

Committee Chair
Adwait Jog, Associate Professor, Computer Science
College of William & Mary

Pradeep Kumar, Assistant Professor, Computer Science
College of William & Mary

Zhenming Liu, Assistant Professor, Computer Science
College of William & Mary

Weizhen Mao, Professor, Computer Science
College of William & Mary

Zhijia Zhao, Associate Professor
University of California, Riverside

ABSTRACT

The big-data era has brought new challenges to computer architectures due to the large-scale computation and data. Moreover, this problem becomes critical in several domains where the computation is also irregular, among which we focus on automata processing in this dissertation. Automata are widely used in applications from different domains such as network intrusion detection, machine learning, and parsing. Large-scale automata processing is challenging for traditional von Neumann architectures. To this end, many accelerator prototypes have been proposed. Micron's Automata Processor (AP) is an example. However, as a spatial architecture, it is unable to handle large automata programs without repeated reconfiguration and re-execution. We found a large number of automata states are never enabled in the execution but still configured on the AP chips, leading to its underutilization. To address this issue, we proposed a lightweight offline profiling technique to predict the never-enabled states and keep them out of the AP. Furthermore, we develop SparseAP, a new execution mode for AP to handle the misprediction efficiently. Our software and hardware co-optimization obtains 2.1× speedup over the baseline AP execution across 26 applications.

Since the AP is not publicly available, we aim to reduce the performance gap between a general-purpose accelerator—Graphics Processing Unit (GPU) and AP. We identify excessive data movement in the GPU memory hierarchy and propose optimization techniques to reduce the data movement. Although our optimization techniques significantly alleviate these memory-related bottlenecks, a side effect of them is the static assignment of work to cores. This leads to poor compute utilization as GPU cores are wasted on idle automata states. Therefore, we propose a new dynamic scheme that effectively balances compute utilization with reduced memory usage. Our combined optimizations provide a significant improvement over the previous state-of-the-art GPU implementations of automata. Moreover, they enable current GPUs to outperform the AP across several applications while performing within an order of magnitude for the rest of them.

To make automata processing on GPU more generic to tasks with different amounts of parallelism, we propose AsyncAP, a lightweight approach that scales with the input length. Threads run asynchronously in AsyncAP, alleviating the bottleneck of thread block synchronization. The evaluation and detailed analysis demonstrate that AsyncAP achieves significant speedup or at least comparable performance under various scenarios for most of the applications.

The future work aims to design automatic ways to generate optimizations and mappings between automata and computation resources for different GPUs. We will broaden the scope of this dissertation to domains such as graph computing.

TABLE OF CONTENTS

Acknowledgments v

Dedication vi

List of Tables vii

List of Figures ix

1 Introduction 2

1.1 Towards Efficient Large-scale Automata Accelerator 3

1.2 Reducing the Gap between GPGPU and Automata Accelerator 4

1.3 Designing and Analyzing a Generic Automata Processing Scheme on GPU 5

1.4 Dissertation Organization 5

2 Background 7

2.1 Automata 7

2.2 Automata Processor 8

2.3 Graphics Processing Units (GPUs) 9

3 Architectural Support for Efficient Large-Scale Automata Processing 11

3.1 Introduction 12

3.2 Background and Terminology 15

3.2.1 NFA-based Pattern Matching 15

3.2.2 Baseline Automata Processor (AP) 16

3.3 Motivation and Analysis . . . 18
3.3.1 Topological Order and Normalized Depth . . . 18
3.3.2 Analysis of Normalized Depth and Enabled NFA States . . . 20
3.3.3 Analysis of Performance Benefits . . . 21
3.4 Design and Implementation of NFA Partitioning . . . 23
3.4.1 Profiling-based Hot/Cold State Prediction . . . 24
3.4.2 Where to Partition? . . . 25
3.4.3 How to Partition? . . . 26
3.4.4 Discussion . . . 27
3.5 Hardware Support for Intermediate Report Handling and Partitioned NFA Processing . . . 28
3.5.1 Analysis of New Execution Modes for AP . . . 28
3.5.2 Implementation Details . . . 31
3.6 Evaluation Methodology . . . 33
3.6.1 Applications . . . 33
3.6.2 Experimental Setup . . . 34
3.7 Experimental Results . . . 36
3.8 Related work . . . 41
3.9 Conclusions . . . 42

4 Why GPUs are Slow at Executing NFAs and How to Make them Faster 43
4.1 Introduction . . . 44
4.2 Background . . . 46
4.2.1 Pattern Matching via NFAs . . . 46
4.2.2 NFA Processing on GPUs . . . 47
4.3 Problem and Previous Efforts . . . 49
4.3.1 Data Movement . . . 49

4.3.2 Compute Utilization 50
4.3.3 Limitations of Prior Efforts 51
4.4 Addressing the Data Movement Problem via Matchset Analysis 52
4.4.1 Inefficiencies in the Transition Table 52
4.4.2 Optimization I: A New Way to Store and Access Matchset and Topology Information (NewTran/NT) 54
4.4.3 Optimization II: Matchset Compression (MaC) 56
4.5 Addressing the Utilization Problem via Activity Analysis 57
4.5.1 Analysis of Activation Frequency 57
4.5.2 Optimization III: Activity-based Processing 58
4.5.3 How do we choose the hot states? 61
4.6 Evaluation Methodology 63
4.7 Experimental Results 66
4.8 Related Work 71
4.9 Conclusions 72

5 Generalizing Automata Processing on GPUs by Leveraging Symbol-level Parallelism 74
5.1 Introduction 75
5.2 Background 79
5.3 Asynchronous Parallel Automata Processing on GPUs 80
5.3.1 Why do we need a new way to process Automata on GPUs? . . 80
5.3.2 Overview of Asynchronous Parallel Automata Processing 82
5.3.3 Design and Implementation 84
5.3.4 Analysis 87
5.4 Characterization of Synchronous Automata Processing and AsyncAP 89
5.4.1 Applications Configurations 89

5.4.2 Comparison of Identified Patterns 89
5.4.3 Characterization of Work by Emulation 92
5.4.4 Comparison of Useful Work 95
5.4.5 Comparison of Total Work 95
5.4.6 How balance the work is in AsyncAP? 96
5.5 Evaluation 97
5.5.1 Evaluation Configurations 97
5.5.2 Experimental Results 98
5.5.3 Analysis of Pattern Lengths 102
5.6 Related Work 105
5.6.1 Mapping Automata to Computation Resources 106
5.6.2 Increasing Parallelism of Automata 106
5.7 Conclusions 107

6 Conclusions and Future Work 109
6.1 Summary of Dissertation Contributions 109
6.2 Future Research Directions 110

Bibliography 112

Vita 133

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help and support of many people.

I would like to express my deepest appreciation to my advisor, Adwait Jog for his helpful and valuable guidance throughout my Ph.D. journey. I have been extremely lucky to work with Adwait and I enjoyed my Ph.D. life. Adwait has been a role model for me. I also thank his family for their support.

I would like to thank our collaborators. I am so grateful to Sreepathi Pai for his valuable ideas and insights, which have greatly helped my automata projects in this dissertation. I also wish to thank Onur Kayıran for his knowledge and insights in the automata accelerator project. I very much appreciate Bogdan Nicolae, Sheng Di, and Franck Cappello for the valuable mentorship on the DNN project when I was a visiting student at ANL.

I would like to thank my dissertation committee members, Pradeep Kumar, Zhenming Liu, Weizhen Mao, and Zhijia Zhao, for their valuable feedback, suggestions, and time.

I would like to thank Meena Arunachalam for the mentorship when I interned at Intel.

I thank the entire technical staff for managing computing facilities at William & Mary. Special thanks to Eric Walter, who helped me with my requests efficiently. I also thank the Administrative Director of the computer science department, Vanessa Godwin, for being efficient, professional, and caring.

I am especially thankful to our lab members Mohamed Assem Ibrahim, Gurunath Kadam, Ying Li, and Haonan Wang for the support, companionship, and discussions. It's a bit sad to say goodbye to our previous office McG-101B.

I would like to extend my gratitude to all my friends. Especially, I am also very fortunate to have a group of dear friends in William & Mary!

I am deeply indebted to my family. My wife and our cat Lume always support me.

I also thank my parents and other family members for their care and encouragement without which I would not be what I am today.

To my family

LIST OF TABLES

3.1 The effectiveness of profile-based prediction. © 2018 IEEE. 25

3.2 List of evaluated applications: "RStates" stands for reporting states and "MaxTopo" stands for maximum topological order across NFAs. "Grp" stands for resource requirement groups: High (H), Medium (M), Low (L). © 2018 IEEE. 34

3.3 Summary of Execution Scenarios. © 2018 IEEE. 35

3.4 Runtime statistics for AP and BaseAP/SpAP (under 1% profiling input): The first three columns show the number of executions on the AP, BaseAP mode and SpAP mode, respectively. "EStalls" stands for the stalls caused by enable operations for handling simultaneous intermediate reports. "JumpRatio" is defined as the proportion of cycles skipped in the SpAP mode. © 2018 IEEE. 39

4.1 Overview of the evaluated schemes on GPU 63

4.2 Characteristics of evaluated NFA applications. 65

4.3 Absolute throughput with our schemes (MB/s). The best performance among GPU schemes is highlighted. 66

5.1 Three levels of parallelism in NFA processing 76

5.2 Categorization of Prior Works 81

5.3 Comparison of Time Complexity. n: number of symbols; m number of states. 88

5.4 Overview of Evaluated Applications 90

5.5 Characteristics of applications based on our execution models: Our key observations: (1) The GPU utilization depends on applications. (2) Although AsyncAP has higher time complexity, in reality only 5% more useful work is needed on average. (3) Most applications balance the work across threads well, but rarely the work is severely imbalanced. 92
5.6 Absolute throughput (in MB/s) of evaluated applications under the scenarios with different amounts of parallelism. 100
5.7 Pattern Lengths of Applications 103

LIST OF FIGURES

2.1 Illustrating an Automata Processor and the NFA configured to it . . . 8
2.2 Overview of GPU Memory Hierarchy 9

3.1 A large portion of NFA states are cold (never-enabled) but are still configured on the AP leading to its underutilization. © 2018 IEEE. . 14
3.2 A homogeneous NFA that accepts regular expression `a((bc)|(cd)+)f`: the doubled circle represents starting state and the hexagon represents reporting state. © 2018 IEEE. 16
3.3 The figure illustrates the first execution cycle of an AP configured with the NFA shown in Figure 3.2. S1 is enabled when input symbol `a` arrives, which activates S1, and enables S2 and S4 in the next cycle. Downward arrows represent the enable signal being fed to routing matrix in the current cycle. Upward arrows enable successor states for the next cycle. The physical connections between STEs and routing matrix are bi-directional, which are represented by the dashed arrows. © 2018 IEEE. 17
3.4 Illustration of topological ordering and normalized depth. © 2018 IEEE. 20
3.5 Distribution of normalized depth for NFA states. For presentation purposes only, normalized depth is classified as: i) shallow ([0–0.3)), ii) medium ([0.3–0.6)), and iii) deep ([0.6–1]). © 2018 IEEE. 21

3.6 An illustrative figure showing that by not configuring cold states on AP, all the hot states can fit onto an AP at the same time, reducing the number of re-executions over the input and hence saving time. © 2018 IEEE. 22
3.7 Partitioning an NFA by the partition layer. © 2018 IEEE. 27
3.8 Constrained states are cold states but configured on the AP due to the constraints in our topological-order-based partitioning scheme. Consequently, some AP resources are underutilized with a few applications. © 2018 IEEE. 28
3.9 Illustration of performance benefits under realistic partitioning: because of the *jump* operation, only a portion of input symbols are executed in the SpAP mode execution. © 2018 IEEE. 29
3.10 Speedup and Resource Savings on AP. © 2018 IEEE. 36
3.11 Performance per STE of various AP sizes with BaseAP/SpAP execution considering 1% profiling input. © 2018 IEEE. 37
3.12 Comparison of number of reporting states: "IM" stands for intermediate reporting states. "True" stands for original reporting states on BaseAP mode. "P" stands for profiling. © 2018 IEEE. 38
3.13 Sensitivity on the different capacities of AP chip. © 2018 IEEE. . . . 40

4.1 Working example of an NFA. 47
4.2 The data movement normalized to the ideal cases: two prior schemes use 25× and 18× compared to the ideal case where only the input stream is loaded. The evaluation methodology is discussed in Section 4.6. 52
4.3 Two metrics showing the redundancy (#edges/#occupied-entries) and sparsity (#occupied-entries/table-size) in the transition table. Lower is worse. 53

4.4 Illustrating the per-node data structure of NewTran (NT). Shaded variables are in the local memory and others are in the registers. . . . 55
4.5 Percentage of states whose matchsets are complete, complement, or not compressible. 56
4.6 The activity profile of the states. For the majority of applications, 80% of non-starting states are activated for only less than 1% of the processed symbols. 58
4.7 Illustrating the activity-based processing 60
4.8 Throughput sensitivity to the selection of hot states. Detailed evaluation methodology is in Section 4.6. HotStart (or HotStart_Opt, an optimized version) has the best performance among these selection schemes. Hence, we choose the always-active start states as hot states. 61
4.9 Throughput enhancement results normalized to iNFAnt. On average HotStart-MaC achieves 26.5× speedup across 16 applications. The best GPU results outperform an AP chip in 5 applications (CAV, YARA, Snort, LV, and Bro). 66
4.10 Throughput enhancement for the applications without *always-active* start states in the single input stream scenario. Our schemes outperform NFA-CG and iNFAnt by at least 9% and 2.6×, respectively. . 68
4.11 Effect on data movement reduction: our schemes use significantly fewer gld_transactions than prior work. For example, HotStart-MaC reduces gld_transactions by 99.3% over iNFAnt. 69
4.12 Effect on the number of NFA states per thread block (a proxy for compute utilization). More states are handled per thread block in HotStart. 69

4.13 Performance sensitivity to Volta GPU Architecture. Both HotStart-MaC and HotStart show more than 15× speedup over iNFAnt, indicating their effectiveness on newer GPU architectures. . 70

5.1 Illustrating an NFA that accepts `a*.x*y`. S_0 and S_1 are *all-input* starting states, which are always active in the execution. 80

5.2 Revisiting traditional synchronous automata processing on GPU (a) and the basic idea of AsyncAP (b). The executions try to find pattern `def` in an input stream `abcdefg...` 83

5.3 Worklist holds active states or matched states. WL stands for worklist. 85

5.4 Performance of Selected Implementations of AsyncAP. Evaluation methodology is described in Section 5.5. 86

5.5 Illustrating our implementation of AsyncAP 87

5.6 Synchronous execution identifies disjoint patterns; Patterns identified by different threads in asynchronous execution may overlap. 91

5.7 Ratio of Overlapped Patterns (R) in Evaluated Applications 91

5.8 Illustrating the Execution Models of Synchronous (GPU-NFA) and Asynchronous (AsyncAP) Automata Processing on GPU 93

5.9 Performance of synchronous and asynchronous automata executions on GPU under different amounts of parallelism 99

5.10 Performance sensitivity to Ampere GPU Architecture 101

5.11 Slowdown of PEN 102

5.12 Large portion of NFAs can never have infinite long patterns. 104

5.13 When limiting the pattern length to 1K, PEN does not exhibit slowdown (i.e., slowdown is less than 1). 105

Techniques for Accelerating Large-scale Automata Processing

Chapter 1

Introduction

For decades, computing has relied on Moore's law to provide abundant computing resources. However, achieving continued growth in terms of performance and energy efficiency has become extremely challenging due to near-end of Moore's law and Dennard scaling. Being compute-centric [25], the current general-purpose von Neumann architectures (CPUs and GPUs) need to fetch instructions/data from memory involving costly operations in terms of energy and speed. Further, achieving fine-grain parallelism and synchronization on these general-purpose processors is difficult from both software and hardware perspectives [7]. Therefore, it is very challenging for these conventional processors to efficiently execute all types of computations.

To address such inefficiencies, domain-specific accelerators (DSAs) are being developed to achieve ASIC-like performance and at very high energy efficiency. A plethora of DSAs have emerged in recent years from both industry and academia—Google's Tensor Processing Unit (TPU) [58] and Pixel Visual Core [4] for machine learning and image processing workloads respectively; NVIDIA's Deep Learning Accelerator (DLA) [3]; ARM's Trillium Project [2]; Microsoft's Brainwave [1]; the DianNao ML accelerator family [39, 40, 45]; GenAx and Darwin for bioinformatics [49], and many more.

This dissertation considers an important domain—automata processing. Similar to the aforementioned examples, many accelerators are proposed to accelerate automata

processing [44, 98, 47, 21]. Automata work as computation kernels of a large set of applications used in different areas such as machine learning [93, 112], bioinformatics [34, 90], and network intrusion detection [10, 131]. However, in the big-data era, automata applications are also growing fast. The applications require many automata and input streams running at the same time. For example, ClamAV [6] is an application containing a lot of virus patterns, which are accumulated very fast with the increased number of internet viruses.

This dissertation focuses on tackling the large-scale automata processing problem on both domain-specific accelerators and general-purpose accelerators.

We study three aspects for efficient large-scale automata processing: 1) How to make current automata DSAs (i.e., Automata Processor) efficient for large-scale automata processing [66]? 2) How can we reduce the gap between the general-purpose accelerator (i.e., General-purpose Graphics Processing Units, GPGPUs) and the DSA (i.e., Automata Processor) for large-scale automata processing [69]? 3) How do we generalize the automata processing on GPU to adapt to various automata task sizes?

1.1 Towards Efficient Large-scale Automata Accelerator

We focus on Automata Processor [44] (AP), which is one of the recently proposed Automata DSAs. We found a fundamental problem of AP—it is unable to handle large-scale automata programs without repeated reconfiguration and re-execution.

To achieve higher throughput in those large-scale automata applications, we propose efficient architectural support for large-scale automata processing. Our mechanisms are based on our key observation that not all states of the automata are enabled during execution, and hence need not be configured to the AP. Specifically, a large fraction of states unnecessarily take space in the AP chip but are not part of any state transitions, leading to its underutilization. To address this problem, we propose a profiling scheme to predict which states are not needed to be configured to AP. We propose a lightweight

misprediction handling approach working as an execution mode of AP with marginal hardware overhead. The detailed experiments demonstrate that our software/hardware co-optimization significantly improves the performance of AP as well as performance per area.

1.2 Reducing the Gap between GPGPU and Automata Accelerator

GPUs are massively parallel accelerators that are widely used and available on the market. Besides its ability in graphics, GPUs also accelerate many general-purpose workloads such as machine learning and scientific computing. Execution of automata on highly parallel architectures like GPUs, therefore, appears very attractive. However, automata applications are very hard to accelerate on traditional von Neumann architectures. In this direction, we analyze the bottlenecks of the large-scale automata processing on GPU. Specifically, we observe two major bottlenecks: 1) excessive data movement through the GPU memory hierarchy is needed for *every* input byte; 2) many threads are idle in the execution due to the characteristics of automata execution.

To tackle the two bottlenecks, we propose new data structures and parallel execution mechanisms tailored for GPU. The new data structures fit the topology of NFA into GPU registers, which avoids the data movement for transition table look-ups. Our parallel execution mechanisms map the active states to threads while mapping the infrequently active states to worklists. These optimizations not only outperform the state-of-the-art GPU automata processing approaches but also reduce the gap between GPU and AP into an order of magnitude. Moreover, for several applications we evaluated, our approach outperforms AP.

1.3 Designing and Analyzing a Generic Automata Processing Scheme on GPU

To make automata processing on GPU more generic for tasks with different parallelism, we propose AsyncAP, an approach that processes automata asynchronously on GPU. Our approach exploits an additional source of parallelism that scales with the length of an input stream. As the input stream is often long enough, AsyncAP enables all tasks to have enough parallelism that can utilize all GPU cores. Each thread of AsyncAP works asynchronously, reducing the overhead incurred by thread block synchronization.

Theoretically, AsyncAP has more time complexity than the traditional synchronous automata processing approaches on GPUs. However, no prior work has studied the amount of work in practice. To understand the amount of work needed in reality, our detailed characterization of synchronous automata execution and AsyncAP demonstrates that AsyncAP only incurs marginal more work on average across the evaluated applications.

For most of the applications, the evaluation demonstrates that a significant speedup is obtained when the original parallelism of a task is not enough. Further, even when the task is equipped with enough parallelism originally, the new approach achieves comparable performance to the state-of-the-art GPU automata processing engine.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces the background for automata, Automata Processor (AP), and General-purpose Graphics Processing Units (GPGPUs). Chapter 3 introduces our observation that the current automata processor cannot efficiently execute large-scale automata processing. Based on our key observations that a large fraction of states are never enabled, we propose software-hardware co-design optimizations for AP to accelerate large-scale automata processing. Chapter 4 answers why GPUs are not efficient at executing automata and propose software approaches to

accelerate it. Our optimizations reduce the gap between a general-purpose accelerator (GPU) and a domain-specific accelerator (AP) and even outperform AP for a few applications. Chapter 5 focuses on the problem that automata processing tasks may not always have enough parallelism to utilize GPU cores. To address this problem, we propose a more generic scheme that exploits symbol-level parallelism, achieving at least comparable performance as the state-of-the-art scheme on GPU for most of the evaluated applications. Chapter 6 summarizes the dissertation and discusses the future work.

Chapter 2

Background

2.1 Automata

Automata are also known as finite state machines (FSMs). Deterministic Finite Automaton (DFA) and Non-deterministic Finite Automaton (NFA) are two commonly used representations. DFA allows only one *active* state for each symbol. On the contrary, NFA allows multiple active states, which is ideal for parallel processing. The NFA used in Automata Processor is Glushkov NFA (aka homogeneous NFA) [51]. In this dissertation, we focus on Glushkov NFA and use *automata* or NFA to term it.[1] We refer the readers to the classical textbooks for the basic knowledge of automata theory [102].

NFA and matching process. An NFA is a directed graph where each node represents a state and each edge represents a state transition. Every state in the NFA has a *matchset* that contains the alphabets (symbols) it matches. Every NFA has at least one *start* state and at least one *reporting* state. The matching process begins by activating the start states. An NFA consumes one symbol at a time from the input stream. For each symbol, all currently active states attempt to match the incoming symbol with their matchset. If any of them match, they activate their successors. Unlike DFA, where only one state is active, NFAs can have multiple states active simultaneously—making them ideal for

[1] Those representations can be transformed with each other, with the same expressiveness.

parallel architectures. If a reporting state matches an input symbol, it generates a report showing that a relevant pattern has been observed in the input stream. Usually, all starting states are always-active, unless a user wants to search patterns that only start at a certain position of the input stream.

2.2 Automata Processor

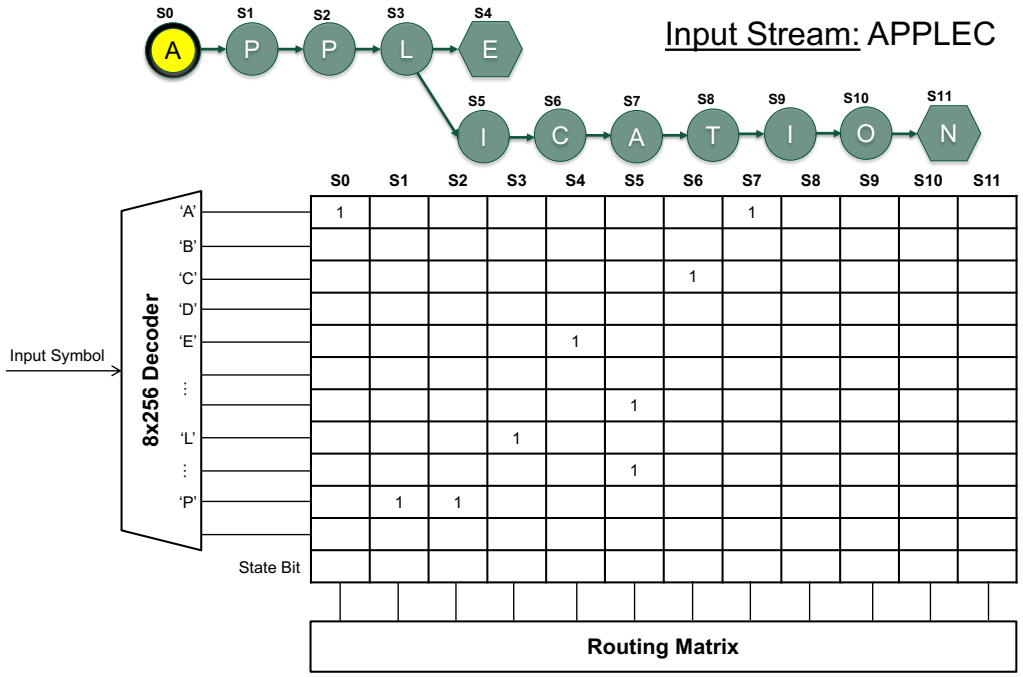


Figure 2.1: Illustrating an Automata Processor and the NFA configured to it

Automata Processor (AP) is a memory-centric accelerator for finite state automata processing by implementing NFA states and state transitions in memory. AP exploits the high parallelism in DRAM. Figure 2.1 illustrates an AP and the NFA configured to it. The matchset of each state is stored in the DRAM columns. For example, S0 accepts symbol `A`, then the bitset of `A` is mapped to the first column. The transitions are configured to the *routing matrix*. Each cycle, the 8×256 decoder selects the row of the incoming symbol. If a state accepts the incoming symbol (i.e., its state bit is 1), then the routing matrix sets the state bit of its successors. The entire input stream is processed sequentially at the

rate of one symbol per cycle.

We refer the readers to prior work [44, 126] and Chapter 3.2 for more details of AP.

2.3 Graphics Processing Units (GPUs)

GPU was an accelerator designed for computer graphics originally. Now, GPUs are found in a wide range of computer systems such as data centers, embedded systems, mobile phones, personal computers, workstations, and game consoles [67, 68, 136, 54]. Compared to the CPU, GPU provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU [129, 137, 134, 133, 132, 135], making GPU a general-purpose accelerator. We refer the readers to this document [7] for more background about GPU.

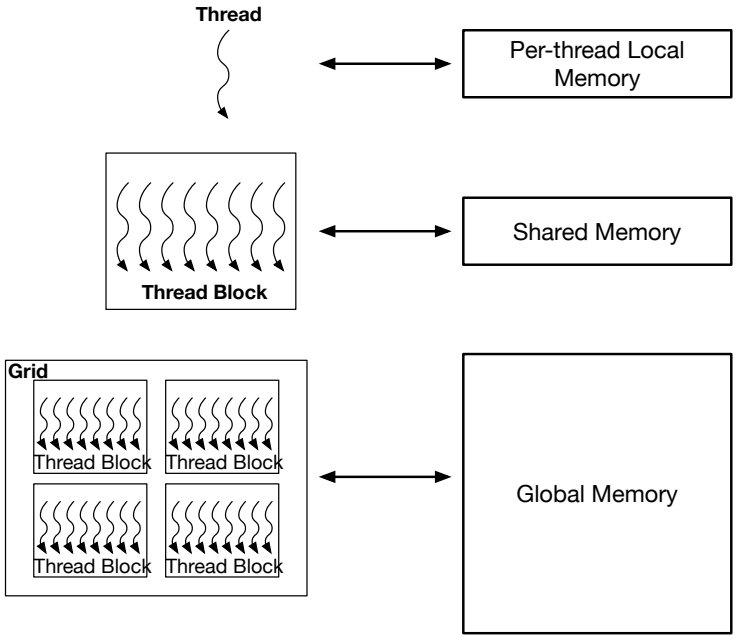


Figure 2.2: Overview of GPU Memory Hierarchy

Programming model and memory hierarchy. GPU uses SIMT programming model [7, 64]. The code running on the device (i.e., *kernel*) is launched in a *grid*. The grid

consists of multiple *thread block*s (aka Cooperative Thread Arrays, CTA, or workgroups). A thread block contains one or more *warps* where 32 *thread*s are grouped. The thread block scheduler assigns thread blocks to stream multiprocessors (SMs) in a many-to-one manner depending on the SM resources and the requirements of thread blocks. In each cycle, the warp schedulers issue an instruction from an active warp to the SIMD executions units. The independent scheduled multiple warps lead to high throughput because of better latency hiding [57, 56].

Figure 2.2 shows the memory hierarchy corresponding to the programming model. *Local memory* is private to each thread. While the local memory can use cache hierarchy in recent GPU architectures, it is located in the off-chip DRAM. *Shared memory* is allocated to each thread block. The threads within a thread block can communicate through the low-latency on-chip shared memory. Global memory is located in the off-chip DRAM which is accessible to all threads within the same CUDA context.

Chapter 3

Architectural Support for Efficient Large-Scale Automata Processing

The Automata Processor (AP) accelerates applications from domains ranging from machine learning to genomics. However, as a spatial architecture, it is unable to handle larger automata programs without repeated reconfiguration and re-execution. To achieve high throughput, this paper proposes for the first time architectural support for AP to efficiently execute large-scale applications. We find that a large number of existing and new Non-deterministic Finite Automata (NFA) based applications have states that are never enabled but are still configured on the AP chips leading to their underutilization. With the help of careful characterization and profiling-based mechanisms, we predict which states are never enabled and hence need not be configured on AP. Furthermore, we develop SparseAP, a new execution mode for AP to efficiently handle the mis-predicted NFA states. Our detailed simulations across 26 applications from various domains show that our newly proposed execution model for AP can obtain 2.1× geometric mean speedup (up to 47×) over the baseline AP execution.

3.1 Introduction

Many applications from domains such as genomics, malware detection, machine learning, and data analytics exhibit high levels of parallelism and are being accelerated through the use of *spatial architectures* that can exploit higher levels of parallelism than CPUs and also can significantly reduce data movement [39, 71, 45, 49, 110, 47, 100, 73, 19]. Spatial architectures usually consist of many interconnected processing elements that expose a very high degree of parallelism. Field-programmable gate arrays (FPGAs) are a classic example; the systolic-array-based Matrix Multiply Unit in Google's Tensor Processing Unit [58] is also a spatial architecture. One of the fundamental challenges with spatial architectures is that program size is a first order concern – there are a fixed number of states available and a spatial program must fit *completely* to begin execution. Otherwise, execution may be impossible, or in the best case multiple rounds of reconfiguration and re-execution may be required that can incur significant performance penalties [144]. On traditional von Neumann architectures, these issues can typically be handled by traditional mechanisms such as context switching and virtualization. However, the large size of the spatial program state means that these techniques do not transfer directly. Some of these issues affect also traditional architectures like the Graphics Processing Units (GPUs), whose massive parallelism also means that the amount of state is often prohibitively large to support efficient multitasking [109, 80, 65, 38].

In this paper, we focus on providing architectural support for executing large-scale tasks on a special class of spatial architectures, known as automata processors (APs) [44]. These architectures accelerate the processing of Non-deterministic Finite Automata (NFA), a widely used representation of Finite State Machines (FSMs). FSMs are foundational in a wide range of application domains such as DNA sequence matching, network intrusion detection and machine learning [152, 113, 91, 33, 81, 121]. Although many existing approaches [36, 146, 154, 88] accelerate NFA processing on CPUs or GPUs, none of them completely solve the problem of data movement caused by irregular accesses

due to NFA transition table lookups. In comparison, the AP executes NFAs natively and achieves significant performance speedup [123, 76] primarily because of: a) AP's massive parallelism where NFA states are mapped to columns in DRAM and can be activated independently and simultaneously in a given cycle; and b) AP's in-memory processing capability that handles NFA transitions without data movement between processor and memory.

An AP half-core (the basic processing unit of AP) can hold up to 24K states. However, in future, we expect that the NFA-based applications are going to scale both in terms of the number of NFAs per application and the number of states in an NFA. We expect this scaling from at least two aspects. First, in the era of big-data, the new applications will likely be mining even larger databases. For example, ClamAV [6], an anti-virus application, uses a variant of regular expression to specify each virus signature in an ever-enlarging database. The number of NFA states constructed from these signature regular expressions is consequently larger and state-of-the-art AP chips can no longer hold all the states at once. Second, a number of existing and newly proposed techniques enhance the throughput of FSM processing, but only by increasing the number of states. For example, existing AP supports duplicating NFAs to run multiple input symbol streams in parallel [5]; newly proposed Parallel Automata Processor [106] duplicates NFAs for parallel enumeration; and the Multi-stride NFAs [35, 29] transformation increases the number of transitions for processing multiple symbols at one step. Current AP chips execute these applications with a large number of NFAs/states by making independent batches of NFAs and executing each batch on the entire input while reconfiguring the AP between each batch.

To address the performance inefficiencies from repeated re-executions, we propose hardware and software support for large-scale NFA-based applications that currently do not fit in the AP chips. Our mechanisms are based on our key observation that not all states of an NFA are enabled during execution, and hence need not be configured to the AP. Specifically, a large fraction of states unnecessarily take space in the AP chip but are

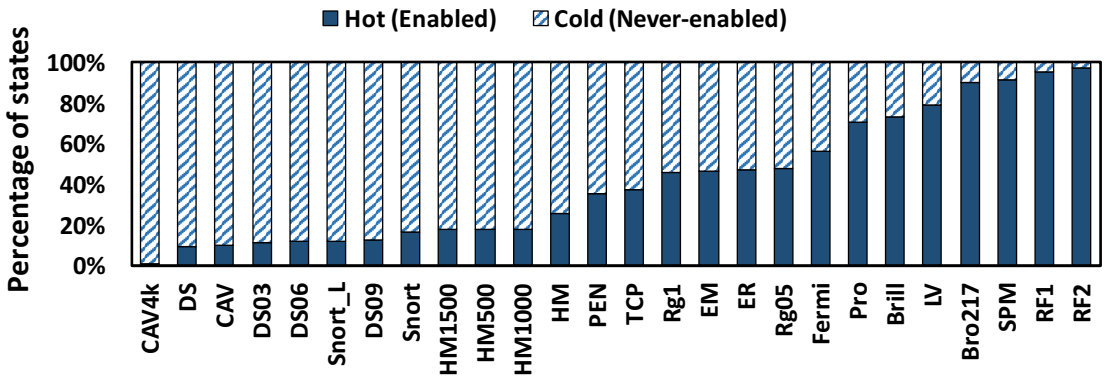


Figure 3.1: A large portion of NFA states are cold (never-enabled) but are still configured on the AP leading to its underutilization. © 2018 IEEE.

not part of any state transitions. We refer to such never enabled states as *cold* states and the remaining (enabled) states as *hot* states. Figure 3.1 quantitatively shows our observation across 26 diverse applications [123, 30] sorted in the increasing order of their percentage of hot states (across all NFAs in an application). We find that on average 59% of states are cold and it can be up to 99% in applications such as `CAV4k`.

These observations can be explained by revisiting the way NFAs process inputs. NFA behavior is highly input dependent. A state can attempt to match a symbol of input only if it is enabled. In the most general case, a state is enabled only if at least one of its predecessor states matched a symbol of input (the exceptions being starting states, which are always enabled). A match indicates that the current input string is plausibly still a valid prefix of the regular language recognized by the NFA. States stop matching as soon as the input string is definitely not in the language. However, the AP must still process all input symbols as long as there is one state enabled (which is always true for an NFA with at least one starting state that is always enabled), thus leaving many states never enabled. Section 3.3 shows that this is indeed the case for the NFAs running on the AP.

Based on the above key insight, we first develop software-based mechanism to predict which states are cold and hence need not be configured on the AP. Next, we propose changes in the AP hardware to efficiently execute the mis-predicted cold states. To the best

of our knowledge, this is the first work that proposes architectural support for efficiently executing large-scale NFA-based applications on the AP. In summary, this paper makes the following contributions:

• We demonstrate that a large number of NFA states are cold during execution but are still configured on the AP. This leads to its severe underutilization.

• We develop a prediction mechanism to classify the NFA states into *predicted hot* and *predicted cold* sets. We use properties of NFA execution to develop a simple and effective partitioning scheme based on a state's topological order and profiling information.

• We develop efficient hardware mechanisms to execute *predicted cold* states using a new *sparse execution mode* for the AP (called as SparseAP). Our detailed evaluation shows that we can achieve 2.1× geometric mean speedup (up to 47×) over the baseline AP execution across a wide range of 26 applications.

3.2 Background and Terminology

In this section, we provide a brief background on NFAs and their processing on the AP.

3.2.1 NFA-based Pattern Matching

An NFA is represented by a 5-tuple, $(Q, \Sigma, \Delta, q_0, F)$, where Q is a set of states, Σ is the alphabet (set of input symbols), Δ is a transition function which maps $\Sigma \times \Delta$ pairs to a new set of states, q_0 is the set of *starting* states, and F is a set of *accepting* or *reporting* states. Because there can be more than one possible state on a transition, such FSM is called *non-deterministic*. The NFAs used by APs are *homogeneous*[1].

These NFAs can be visualized as a directed graph where each node represents a state and each edge represents a state transition. Each state in the NFA has a *symbol-set* that represents what symbols can be accepted by this state. Each state has one or multiple

[1]In homogeneous NFAs [44, 20], all incoming transitions to any given state must accept the same set of input symbols (symbol-set). In the rest of this paper, we treat *homogeneous* NFA synonymous with NFA, because they have the same computational ability and time complexity.

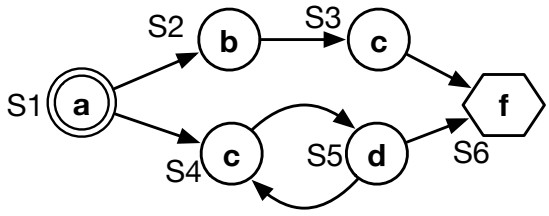


Figure 3.2: A homogeneous NFA that accepts regular expression `a((bc)|(cd)+)f`: the doubled circle represents starting state and the hexagon represents reporting state. © 2018 IEEE.

successors connected by directed edges. In each step, the NFA has a number of *enabled states*. The *starting states* are *enabled* prior to the execution. The matching process is driven by a stream of input symbols. Each cycle, an enabled state compares the input symbol with its symbol-set for matching; when the symbol matches, the state is *activated*, and all its successor states are enabled in the next cycle. When a *reporting state* is activated, it generates a report showing that a relevant pattern has been observed in the input symbol stream.

Figure 3.2 shows the NFA of the regular expression `a((bc)|(cd)+)f`. At first, the starting state S_1 is enabled. `abcf` is the input symbol stream. `a` activates state S_1, resulting in the successors of S_1 (i.e., S_2 and S_4) to be enabled in the next cycle. `b` activates state S_2 (S_4 is not activated since it does not accept symbol `b`), then the successor of S_2 (i.e., S_3) is enabled. The process repeats until all input symbols are consumed. In this case, since reporting state S_6 is activated by input symbol `f`, a report is generated indicating a successful match.

3.2.2 Baseline Automata Processor (AP)

Figure 3.3 shows a schematic of the considered baseline AP chip. The AP is a DRAM-based spatial architecture in which each state of NFA is stored in a memory column of the DRAM, namely a state transition element (STE). A bit in the column represents whether the STE can accept the corresponding input symbol represented by each row.

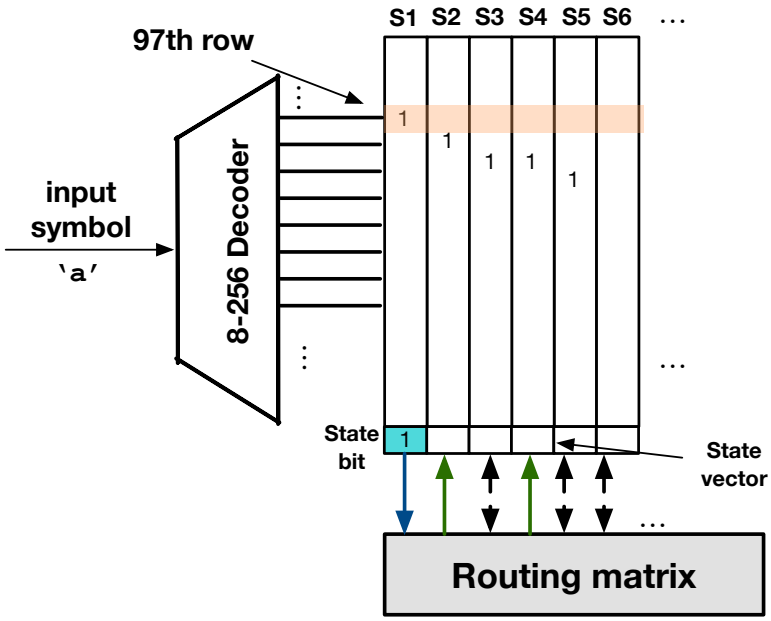


Figure 3.3: The figure illustrates the first execution cycle of an AP configured with the NFA shown in Figure 3.2. S1 is enabled when input symbol `a` arrives, which activates S1, and enables S2 and S4 in the next cycle. Downward arrows represent the enable signal being fed to routing matrix in the current cycle. Upward arrows enable successor states for the next cycle. The physical connections between STEs and routing matrix are bi-directional, which are represented by the dashed arrows. © 2018 IEEE.

The maximum size of the alphabet is 256 as this is the width of the address decoder in the current AP architecture. Therefore, there are 256 rows in total. An AP chip consists of two *half-cores*. The state transition cannot go across half-cores due to the limitation of the interconnect. The state transitions are compiled to the reconfigurable interconnecting network namely *routing matrix*.

The entire input stream is processed sequentially with the rate of one symbol per cycle. Each cycle, one input symbol is fed into the address decoder, which selects a whole row (out of 256) of the DRAM (orange shaded part in Figure 3.3). Each STE column has a bit that represents whether the STE is enabled or not, namely *state bit*. The state bits for all STEs are combined as a *state vector*. This information is available from the previous cycle. An *AND* operation is performed between the selected row (e.g., shaded part) and

the state vector resulting in a vector that determines the activated states. This activation information is sent to the routing matrix, which updates the state vector with the enabled states for processing next symbol. Such a process is repeated until the entire input symbol stream is processed.

To understand the working of AP, we illustrate the execution of previously considered NFA (Figure 3.2) via Figure 3.3. We previously observed in Figure 3.2 that S_1 accepts symbol `a`. Accordingly, the bit stored in the 97th row (corresponding to the ASCII of `a`) and the column of STE that stores S_1 is set to 1 and the others remain 0. The state bit of S_1 is 1 *and* $\{a\}$ is in the symbol-set of S_1, therefore, S_1 is activated and it broadcasts the enable signals to the successor states (S_2, S_4) via the routing matrix (upward arrows in Figure 3.3).

3.3 Motivation and Analysis

In this section, we analyze why a high percentage of states are cold, which states are more likely to be cold, and how avoiding these states from being configured to AP can improve the performance.

3.3.1 Topological Order and Normalized Depth

In general, it is hard to predict which states will be enabled in NFAs [149]. Clearly, all starting states will be enabled at least once and this does not depend on the input. The states that are further away from the starting state, however, depend on the input. Each subsequent state transition in a homogeneous NFA must match a symbol of input (homogeneous NFAs do not have ϵ-transitions [101]). Intuitively, a state that is further away from the starting state is less likely to be enabled since each additional state on the path to it increases the chances of a mismatch.

To verify if this intuition holds on NFAs from real-world applications executing on the AP, we study whether states are hot or cold with respect to their depths in the NFAs.

For simplicity of exposition, we first consider only NFAs that are also directed-acyclic graphs (DAGs). In this case, the depth of a state is simply its topological order (i.e., the maximum steps from the starting state to itself in the matching process). Thus, the matching process goes from states with a lower topological order to states with a higher topological order but cannot go back as DAGs do not have cycles. Such an NFA can be viewed as a graph with layers, where all starting states are in the first layer (i.e., their topological order is one), states in the second layer (i.e., states with topological order of two) are reachable from the first layer, states in the third layer are reachable from the first and second layers, and so on.

However, NFAs are not always DAGs, because they can contain back edges (i.e., from a later layer to an earlier layer) and cycles. For example, the NFA in Figure 3.4 (❶) contains a cycle between states S_4 and S_5. Topological sort cannot be performed on such graphs. Therefore, we pre-process an NFA by identifying all its strongly connected components (SCC) [42]. Each state s is marked with a connected component number $\text{SCC}(s)$, such that the states belonging to the same SCC are marked with the same number. We construct graph G' from directed graph G (i.e., the NFA) by treating each SCC in G as a single node in G' (e.g., in Figure 3.4, the SCC that includes states S_4 and S_5 is considered as a single node in G'). For each edge (u, v) in G, an edge $(\text{SCC}(u), \text{SCC}(v))$ is added in G' if nodes u and v are in different SCCs. The resulting G' is a DAG on which we can run a topological sort. Figure 3.4 (❷) shows the results of identifying SCCs and topological sort. The topological order of each state is indicated as a number right to the state. Since S_4 and S_5 belong to the same SCC, they are assigned with the same topological order.

The absolute topological order or depth of a state is uninformative as different NFAs can have a different number of layers, even within the same application. Therefore, we normalize the depth of a state to the maximum depth in the NFA it belongs to, resulting in *normalized depth*. For example, in Figure 3.4 (❷), because the maximum topological order is 4 (S_6), the normalized depth of each state s is $\mathit{topoorder}(s)/4$ (e.g., for S_4 and S_5, it is 2/4 or 0.5) where *topoorder* is a function that returns the topological order of

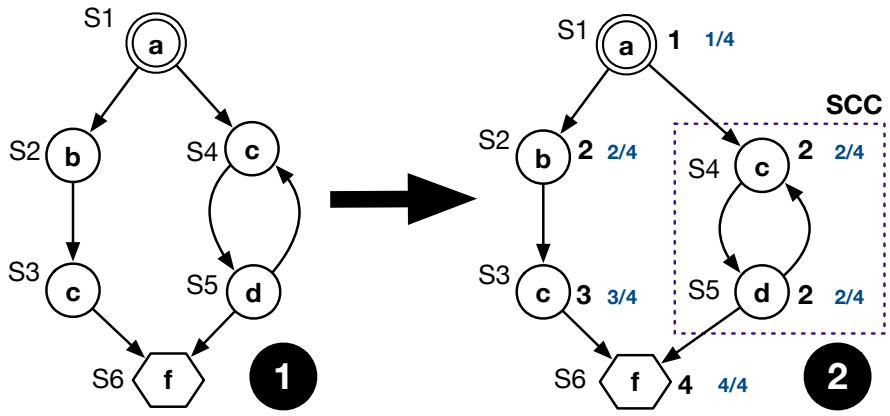


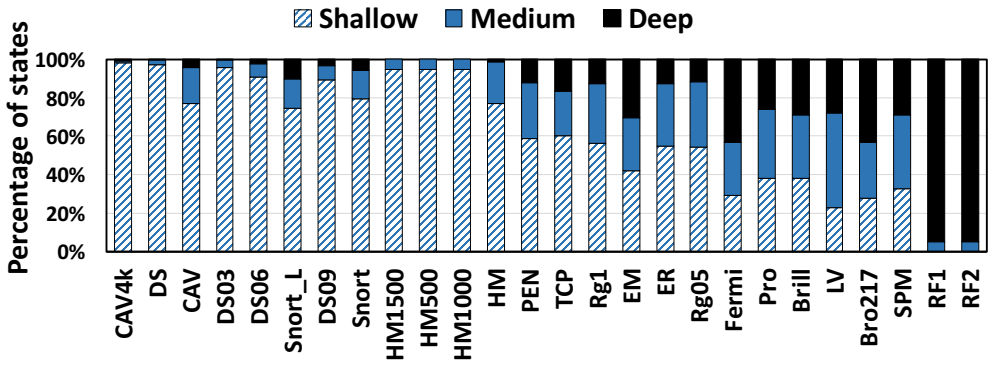
Figure 3.4: Illustration of topological ordering and normalized depth. © 2018 IEEE.

a state. A normalized depth closer to 1 indicates the state is at the bottom of the NFA (or relatively deep), while a value closer to 0 indicates the state is closer to the top (or relatively shallow).

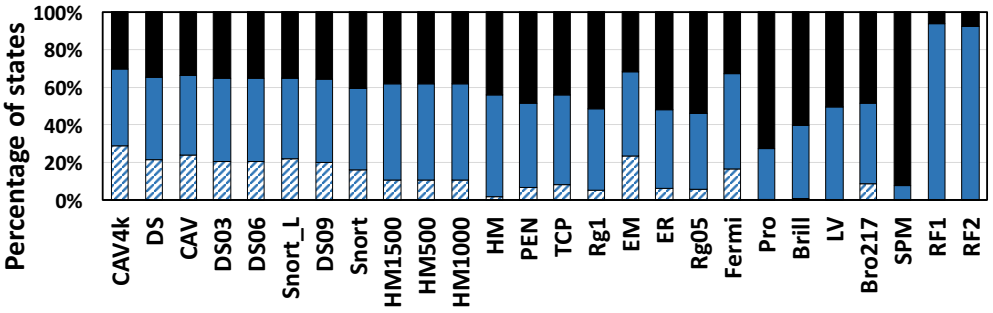
3.3.2 Analysis of Normalized Depth and Enabled NFA States

Figure 3.5(a) shows the normalized depth distribution of enabled (hot) states for our evaluated applications. Each application is comprised of many NFAs, each representing a different pattern. We find that for the majority of applications, the hot states have low normalized depth (i.e., they are closer to the starting state of the NFAs). Furthermore, for the same set of applications, Figure 3.5(b) shows the normalized depth distribution of cold (never enabled) states. We observe that the cold states in the majority of the applications have high normalized depth (i.e., they are in deeper regions of the NFAs). To confirm this conclusion further, we also find that there is a significant negative correlation (average correlation coefficient is -0.82) between normalized depth and percentage of hot states for all applications, except `ER`.

We conclude that whether a state is hot or cold is highly correlated with its normalized depth. Overall, "shallow" states are more likely to be hot while "deep" states are more likely to be cold.



(a) Hot (Enabled) states.



(b) Cold (Never-enabled) states.

Figure 3.5: Distribution of normalized depth for NFA states. For presentation purposes only, normalized depth is classified as: i) shallow ([0–0.3)), ii) medium ([0.3–0.6)), and iii) deep ([0.6–1]). © 2018 IEEE.

3.3.3 Analysis of Performance Benefits

We analyze the ideal performance benefits when we completely eliminate the cold states from being configured on the AP. We show the potential benefits using a performance model assuming oracular knowledge of which states are cold and not configured on the AP.

Performance Model. Consider the case of the baseline AP execution, where the application has S states (across all NFAs) and the number of states the AP half-core can hold (capacity) is C_{AP}. Without loss of generality, we only discuss the case of one AP half-core. If the number of states (S) is larger than the size of AP (C_{AP}), it is not possible to configure the entire application at once to the AP and will require configuring the AP

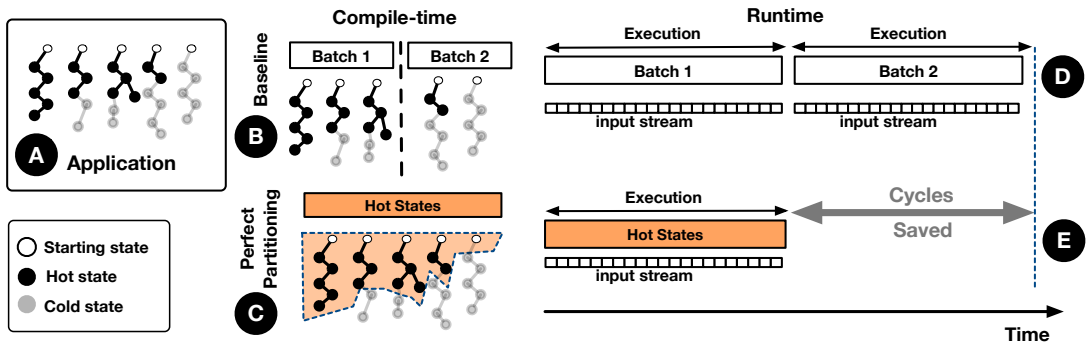


Figure 3.6: An illustrative figure showing that by not configuring cold states on AP, all the hot states can fit onto an AP at the same time, reducing the number of re-executions over the input and hence saving time. © 2018 IEEE.

multiple times. Each configuration places a set of NFAs that can collectively fit in the AP. Suppose the size of each NFA in the application is less than the size of AP, therefore, the number of configurations to the AP would be $N_{config} = \lceil \frac{S}{C_{AP}} \rceil$, under the assumption that individual NFAs can be split at state granularity. In the current AP architecture, batches (partitions) usually contain whole NFAs, so the number of configurations may be even higher.

To maintain semantics, each configuration batch must see the same input stream. The matching process finishes after all batches of NFAs are executed on the same input stream. Thus, the total number of cycles spent on the same input stream is $N_{config} \times n$, where n is the length of the input stream and N_{config} is the number of batches. Under a perfect scenario where we can identify cold states (S_{cold}) with 100% accuracy, we can reduce N_{config} by not configuring the cold states to the AP. We define the resource saving $p = \frac{S_{cold}}{S}$. Therefore, the speedup over the baseline case is $\lceil \frac{S}{C_{AP}} \rceil / \lceil \frac{(1-p) \cdot S}{C_{AP}} \rceil$. If the number of states is sufficiently large, the speedup we can get is proportional to $\frac{1}{1-p}$, $p \neq 1$. Thus, the larger the proportion of cold states that can be correctly identified and eliminated, the more speedup we can have over the baseline execution scenario.

Illustrative Example. To illustrate the benefits of configuring the AP with only hot states, Figure 3.6 shows two scenarios: a) the baseline AP execution, and b) the AP that

only executes hot states. The execution in both cases considers the same application (Ⓐ). In the baseline scenario, if the number of total states is more than the AP capacity, the execution will need to be done in batches as discussed before. In this example, the compiler partitions the application into two batches, where each batch can individually fit in the AP (Ⓑ). Hence, the same input stream is executed twice in a sequential manner (Ⓓ). However, with the oracular knowledge of cold states, the compiler can generate a *perfect partition* of the application with only the hot states (Ⓒ). If this perfect partition fits in the AP, it can execute on it by consuming the same input stream only once (Ⓔ), resulting in significant savings in the execution cycles.

In summary, significant speedup can be achieved if cold states are not configured to AP. In the next section, we propose a simple and effective profiling-based mechanism to identify such states in realistic scenarios and then leverage the profiling information to efficiently partition them from the NFAs.

3.4 Design and Implementation of NFA Partitioning

Any realistic implementation that eliminates cold states from NFAs (i.e., partitions NFAs into cold and hot states, and only configures AP with hot states) has to deal with at least three challenges. First, although it is not possible to predict cold states with 100% accuracy in general, we need to develop low-overhead techniques to improve the accuracy of prediction as much as possible. Second, in the case of a mis-prediction, some transitions may require states that are not configured on the AP. To this end, we need a mechanism working as a safety net to handle a transition from a state on the AP to a state that is not on the AP. Third, to minimize the cost of such mis-predictions, transitions should be unidirectional to avoid re-executions of inputs on the AP.

Our proposed partitioning scheme systematically addresses these challenges. First, we use a profiling-based scheme to identify the topological layer that acts as a *partition layer* for each NFA in the application. Second, our proposed scheme handles transitions out of

the AP by adding *intermediate reporting states* that piggyback on existing AP reporting hardware. Finally, to ensure unidirectional transitions, we partition the NFA at a specific topological order. Since the matching always proceeds from a lower to a higher topological order, edges that cross partitions go only in one direction.

3.4.1 Profiling-based Hot/Cold State Prediction

We use a small portion of input for each application as profiling input. Basically, at compile time, we run the profiling input on the NFAs of the application and determine whether a state is hot or cold. We assume that this profiling information holds true during the actual execution and hence are able to predict which states will be hot or cold. In the following parts of this sub-section, we evaluate the effectiveness of our profiling-based prediction.

Profiling and Testing Inputs. Each application that we evaluate has a 1MB input. We divide this 1MB input into two equal parts of 512KB. The first 512KB of input is used for creating different sizes of profiling inputs and the last 512KB is used for testing input. We create different sizes of profiling inputs by using the first 0.2%, 2%, 20%, 100% symbols of the 512KB portion, which is essentially 0.1%, 1%, 10%, 50% of the entire input.

Methodology for Evaluating the Effectiveness of Profiling. In our evaluation, we treat hot as *positive* (P) and cold as *negative* (N). Therefore, *true positives* (TP) are states that are hot both under profiling input and testing input. Similarly, *false positives* (FP) are states that are hot under profiling input but actually cold under testing input. True negatives (TN) and false negatives (FN) are defined similarly. We define: a) $accuracy = \frac{TP+TN}{P+N}$, which measures overall how well is the profiling-based prediction; b) $recall = \frac{TP}{TP+FN}$, which measures how complete our prediction is terms of predicting hot states; and c) $precision = \frac{TP}{TP+FP}$, which measures how well the prediction could realize the resource saving scope (p).

Effectiveness of Profiling. Table 3.1 shows the average numbers for accuracy, recall, and precision when we use different sizes of profiling inputs. We evaluate all applications

Table 3.1: The effectiveness of profile-based prediction. © 2018 IEEE.

Percentage of the entire input ⇒	0.1%	1%	10%	50%
Accuracy	87%	90%	93%	97%
Recall	64%	76%	87%	97%
Precision	94%	92%	90%	92%

except `Fermi` and `SPM`. Specifically, using only 2% prefix of the first 512KB (i.e., 1% of the entire input) can achieve 76% recall, which means 76% of hot states under testing input are also hot with the small profiling input. The results are consistent across 24 applications (recall varies from 49% to 100%). In addition, the prediction also has good results in terms of accuracy and precision. To conclude, only a small profiling input can identify most of the hot states during the actual execution. Therefore, we use 0.1% and 1% of the entire input for profiling and the remaining for the actual evaluation[2] (Section 3.7).

3.4.2 Where to Partition?

In current AP architecture, the application is split at NFA granularity into batches. In contrast, we partition the NFAs at topological-order granularity. There are two reasons that we use topological-order as our partition granularity. First, our previous analysis (Section 3.3.2) shows there is a correlation between normalized depth and percentage of hot states. Second, partition at topological-order granularity can guarantee the unidirectional transition between predicted cold and hot states. In this sub-section, we show how do we obtain partition layer k_U for each NFA U of the application. We will show how to partition each NFA at the topological-order granularity in Section 3.4.3.

Choosing Partition Layer. At compile time, we functionally simulate all NFAs of the application using the profiling input and predict whether a state is hot or cold. After simulation, for each NFA U, we set $k_U = \max\{topoorder(s)\}$, $\forall s$ is a hot state in NFA U under the profiling input. We define the *predicted hot set* $= \{s \mid s \in U \wedge topoorder(s) \leq k_U, \forall U\}$. Accordingly, the *predicted cold set* $= \{s \mid s \in U \wedge topoorder(s) > k_U, \forall U\}$. We

[2]For `Fermi` and `SPM`, we use the entire input for the actual execution because their starting states are only enabled at position 0 (*start-of-data* in ANML configuration).

divide the predicted hot set at NFA level into batches that can fit in AP and configure each batch sequentially.

Optimization. As an optimization, to make each batch fill the AP completely, we assign additional states to the predicted hot set from predicted cold set. This is achieved by incrementing k_U, which adds the states of the subsequent partition layers for each NFA U. This process terminates when the capacity of AP is met for each batch.

3.4.3 How to Partition?

In this sub-section, we demonstrate how to partition an NFA into two parts at a given partition layer k calculated based on the description presented in Section 3.4.2 and how to handle state transitions when the partitioning is imperfect. For brevity, we describe our partitioning scheme for a single NFA, which then can be separately applied to each NFA in the application. Figure 3.7 illustrates NFA partitioning using the partition layer $k = 3$ and cut the edges that connect states with $k \leq 3$ to states with $k > 3$ (indicated as dashed lines in Figure 3.7 (❶)). However, the prediction may not be perfect – a state in the predicted cold set could end up being enabled during matching. Since only states in the predicted hot set are present on the AP, the matching process must transition out of the AP.

To handle such cases, for each edge (u, v) we cut in the original NFA, we introduce an *intermediate reporting state* v' and an edge (u, v'). The state v' matches exactly the same input symbols (symbol-set) as v but is also a *reporting state*. During execution, the AP contains these *intermediate reporting states* along with the predicted hot set. Therefore, when the matching process tries to enable a state that is not on the AP (i.e., in the predicted cold set), it activates the corresponding intermediate reporting state instead. Consequently, an *intermediate report* is generated that notifies a handler (Section 3.5). The handler will enable corresponding states in predicted cold set to continue the matching process. Since we use topological order to partition, after the matching process continues, it will never go back to the predicted hot set. In Figure 3.7 (❷), the intermediate reporting

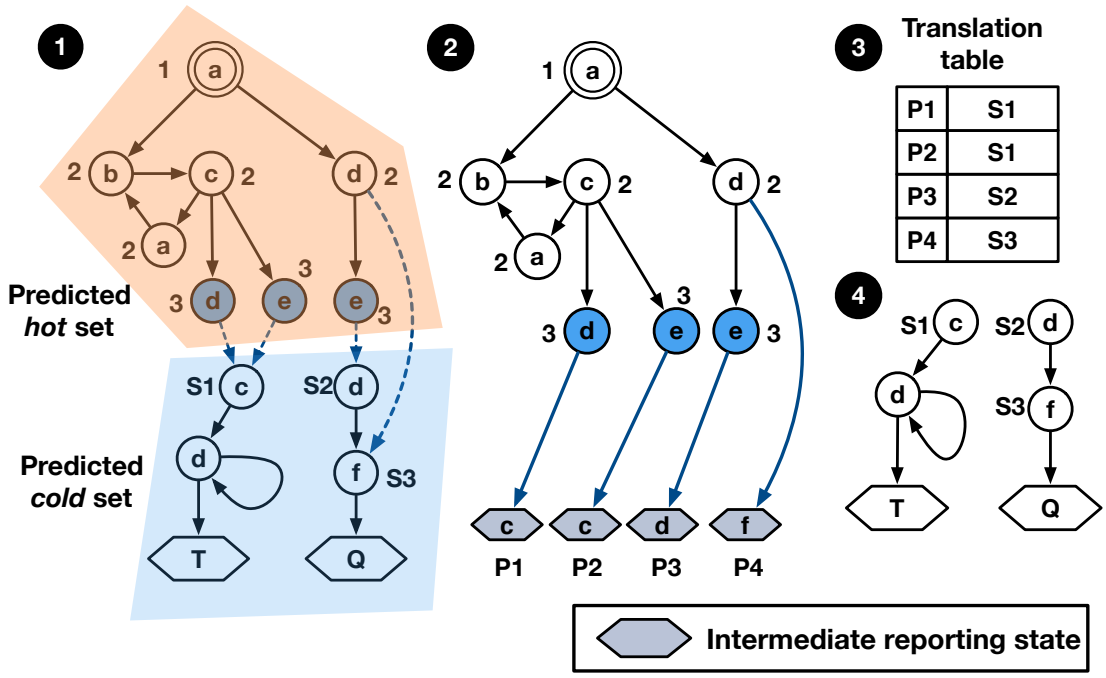


Figure 3.7: Partitioning an NFA by the partition layer. © 2018 IEEE.

states are P_1 through P_4. When activated, these states enable their corresponding states S1, S2 and S3 as indicated in the translation table (Figure 3.7 (❸)), which lie in the predicted cold set shown in Figure 3.7(❹).

3.4.4 Discussion

The use of SCC and topological-order-based partitioning imposes constraints that lead to more states than necessary being added to the predicted hot set. Specifically, (1) even if only one state in an SCC is hot, the whole SCC must be included in predicted hot set, and (2) a cold state with topological order less than the partition layer k is still included in the predicted hot set. This might reduce the AP resource savings.

To study the extent of this underutilization, Figure 3.8 shows that for all the 26 evaluated applications, our topological-order based perfect partitioning constrains only 4% on average more states to the predicted hot set (which in reality are not going to be enabled), compared with perfect partitioning that can cut NFAs at arbitrary edges. Two

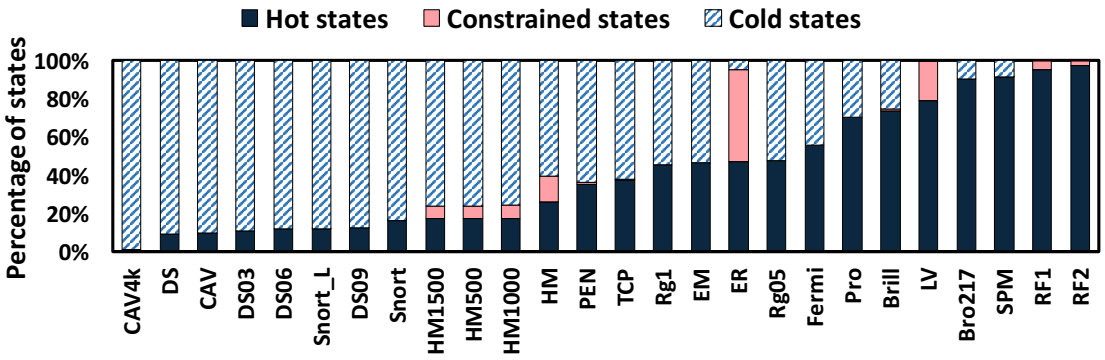


Figure 3.8: Constrained states are cold states but configured on the AP due to the constraints in our topological-order-based partitioning scheme. Consequently, some AP resources are underutilized with a few applications. © 2018 IEEE.

exceptions are `LV` and `ER` whose large SCCs prevent effective partitions. In summary, we still have a significant opportunity for resource savings if we can accurately identify the partition layer for each NFA.

3.5 Hardware Support for Intermediate Report Handling and Partitioned NFA Processing

In this section, we discuss how to efficiently handle the intermediate reports generated from the execution of the predicted hot set. To this end, we propose to: a) enable the states that intermediate reporting state directs to, and b) continue the matching process from the cycle (i.e., the input position) where the intermediate report was generated at. Although both steps can be performed on CPU, it incurs significant performance slowdown (Section 3.7), therefore we propose a new execution mode for the AP.

3.5.1 Analysis of New Execution Modes for AP

In order to support the aforementioned steps, we propose an augmented AP which supports two modes: BaseAP mode, and SparseAP (SpAP) mode. The BaseAP mode execution is similar to the baseline AP execution, however, AP in this mode is configured with only the

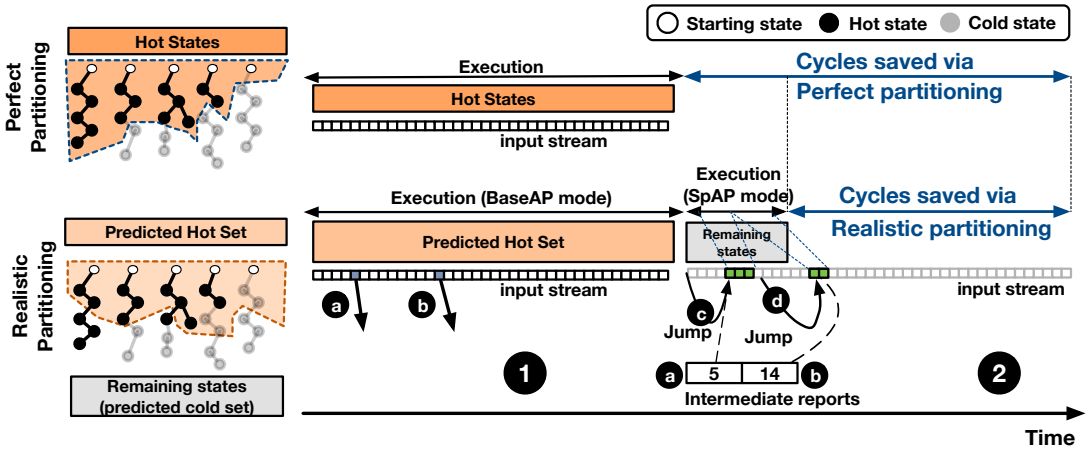


Figure 3.9: Illustration of performance benefits under realistic partitioning: because of the *jump* operation, only a portion of input symbols are executed in the SpAP mode execution. © 2018 IEEE.

predicted hot set. Once the execution of BaseAP mode finishes, the generated intermediate reports are handled in the SpAP mode. In the SpAP mode, the AP is configured with the predicted cold set. The AP in this mode not only consumes input symbols but is also driven by the intermediate reports.

In this context, we develop two major operations for the SpAP mode: *enable* and *jump*. The enable operation allows each intermediate report to enable the appropriate state in the predicted cold set. The jump operation skips over the input symbols that are not necessary for handling the intermediate reports. Since no back-edge exists from predicted cold states to predicted hot states (discussed in Section 3.4), no back and forth switching between BaseAP and SpAP modes is required.

Each intermediate report in the list of intermediate reports (L) is represented by a tuple: input position and state ID (c, sid) denoting that the intermediate report is generated at input position c (i.e., cycle c in the BaseAP mode execution) and the state to be enabled is sid. Algorithm 1 shows the pseudo code for the SpAP mode execution. In each cycle, if no state is enabled (Line 4), it performs a *jump* operation setting the current input position i to the input position where next intermediate report was generated. The

Algorithm 1 Functionality of SpAP mode

Input: L, the list of intermediate reports. Each element in L contains (c, sid) showing the input position where the report was generated, and the state id to be enabled.
Input: *input*, the input symbol stream.
Output: *out_list*, the list of reports.

```
1:  i ⇐ 0
2:  j ⇐ 0                          ▷ i is the index (input position) of input, j is the index of L.
3:  while i < input.length do
4:      if E is ∅ then                                   ▷ E is the set of enabled states.
5:          if j < L.length then
6:              i ⇐ L[j].c                                          ▷ Jump operation.
7:          else
8:              break
9:      while L[j].c = i and j < L.length do
10:         enable L[j].sid                                         ▷ Enable operation.
11:         j ⇐ j + 1
12:     A ⇐ {states in E that accept input[i]}
13:                                              ▷ A is the set of activated states.
14:     E ⇐ ∅
15:     for all s in A do
16:         if s is a reporting state then
17:             append (i, s.id) to out_list
18:         E ⇐ E ∪ {successors of s}.
19:     i ⇐ i + 1
```

enable operation (Line 9 to Line 11) is performed due to either scenario: current input position i reaches the input position in next intermediate report or the current input position i was just set to $L[j].c$ by the jump operation. The remaining functionality of the SpAP mode is the same as the BaseAP mode. We describe next how these operations are used to handle realistic partitioning scenarios with the help of an illustrative example.

Illustrative Example. Figure 3.6 earlier discussed the performance benefits of perfect partitioning. Under realistic partitioning, inaccurate predictions of cold states require intermediate report handling. Figure 3.9 shows an illustrative example demonstrating the benefits of executing AP in BaseAP and SpAP modes. The execution starts in the BaseAP mode (❶) that is configured with the predicted hot set. During its execution, two intermediate reports are generated at input position 5 and input position 14, respectively and are stored (ⓐ, ⓑ). Once all the input symbols are consumed, the SpAP mode begins

(❷), which is driven by both the input stream and the intermediate reports. If no state is enabled, SpAP mode jumps to the input position where the next intermediate report was generated. In this example, initially, it jumps to the input position 5 of the first intermediate report directly (ⓒ). During the execution, when there is no enabled state (at input position 8), the SpAP jumps to input position (14) of the next intermediate report (ⓓ). Therefore, under SpAP, only a portion of the input symbols are executed (green shaded part in ❷).

3.5.2 Implementation Details

We describe the required hardware implementation supporting SpAP mode by implementing the *jump* and *enable* operations on top of the current AP architecture. We start by the implementation of the SpAP operations. Then we estimate the execution time overhead of these operations. Finally, we demonstrate the storage requirements for the intermediate reports.

Jump Operation. The *jump* operation modifies a register that tracks the current input position. Specifically, if no STE is enabled, the *jump* operation updates the register value with the input position from the next intermediate report. Since no state configured to SpAP is always enabled, the enabled states in next cycle are only determined by the activated states in the current cycle. Therefore, given that the routing matrix routes the enable signal from the activated states, we assume that the routing matrix provides a flag that is set if no STE is enabled.

Enable Operation. Given an intermediate report, we use the state ID information to enable the corresponding STE. Since STEs are connected to the routing matrix, and the routing matrix follows a hierarchical design (block, rows, and STEs) [44], we utilize such hierarchy to perform the *enable* operation. The routing matrix consists of 96 blocks per half core. Each block is a group of 16 rows, and each row is a group of 16 STEs. Since state ID is represented by 16 bits, we divide these bits to enable the required STE in a hierarchical manner. We use the first 8 bits to select the block, the middle 4 bits to select

the row, and the last 4 bits to select the required STE within the row. We use a total of three decoders to select the required block, row, and STE, respectively. Specifically, a 7×128 decoder is used to select the block. Then, a 4×16 decoder selects the row. Finally, a 4×16 decoder enables the required STE. The *enable* operation works in parallel with the processing of input symbols during SpAP mode.

Enable Operation Execution Overhead. We can overlap the *enable* operation of only one intermediate report with the processing of the input symbols in SpAP mode. Thus, if multiple intermediate reports were generated in the same input position during BaseAP mode, the input processing is stalled until all the states in the simultaneous intermediate reports are enabled. In SpAP mode, to do that, we compare the input position of the head intermediate report with the next input position (current input position + 1). Similarly, we compare the input position of the second intermediate report with the next input position. If both of these comparisons are set, we pause the processing of the input symbols. After enabling the states in all simultaneous intermediate reports, the input processing resumes. The cycles spent to enable the simultaneous intermediate reports are considered overhead to the overall SpAP mode execution and are accounted for in our evaluation methodology.

Intermediate Reports Storage Overhead. The list of intermediate reports is stored in the off-chip device memory. Only a portion of the reports is loaded to the on-chip memory to be consumed during the SpAP mode. We use a queue of 128 entries to store the loaded intermediate reports. Because each intermediate report is a (input position, state ID) tuple, we need 6 bytes per intermediate report (4 bytes for the input position, and 2 bytes for the state ID). Thus, the overall storage required for the intermediate reports queue is 128×6 bytes.

3.6 Evaluation Methodology

3.6.1 Applications

We evaluate our mechanisms with all applications in the ANMLZoo benchmark suite [123] and the Regex benchmark suite [30]. Table 3.2 shows that these applications have states ranging from approximately 2K to 100K, and several of them have states more than 24K, which is the size of our baseline AP half-core. In order to evaluate applications with an even larger number of states, we generate multiple applications based on three sources: ClamAV [6], Hamming [90], and Snort [89].

ClamAV4k (CAV4k). We convert the regular expressions in `main.cvd` of the Q1 2018 ClamAV Virus Database to ANML format. We select the first 4,000 patterns from the virus database. We use the same input of ClamAV in ANMLZoo [123].

Hamming. We generate Hamming automata using the same approach as the ANMLZoo benchmark suite[90]. To keep it consistent with Hamming in ANMLZoo, we also create the automata in the BMIA (Bounded Mismatch Identification Automaton) format. We created three different workloads from Hamming that contain different number of NFAs, namely `HM500`, `HM1000` and `HM1500`. For each workload we generate, we create a mix of different expected pattern lengths (8, 12, 20, 30), each with a distance of 2 to 20% of the pattern length (e.g., $0.2 \times 30 = 6$). Similar to Hamming in ANMLZoo [123], we generate the inputs randomly.

Snort_L. Our `Snort_L` application includes 3,126 rules from both community rules and registered rules of the Snort network intrusion detector [89]. We convert the regular expressions to ANML format. We use the same network traffic input as the Snort application in ANMLZoo.

We consider a total of 26 applications and divide them into three groups based on the number of states they contain. The high resource requirement (high) group contains applications with states more than the capacity of an AP chip (49K). The medium resource requirement (medium) group contains applications with states more than the capacity

Table 3.2: List of evaluated applications: "RStates" stands for reporting states and "MaxTopo" stands for maximum topological order across NFAs. "Grp" stands for resource requirement groups: High (H), Medium (M), Low (L). © 2018 IEEE.

Application	Abbr.	Grp.	#States	#NFAs	MaxTopo	#RStates
ClamAV4000 [6]	CAV4k	H	1124947	4000	2080	4015
Hamming1500 [90]	HM1500	H	366000	3000	32	6000
Hamming1000 [90]	HM1000	H	244000	2000	32	4000
Snort_big [89]	Snort_L	H	132171	3126	4509	4043
Hamming500 [90]	HM500	H	122000	1000	32	2000
SPM [123]	SPM	H	100500	5025	16	5025
Dotstar [123]	DS	H	96438	2837	95	2838
EntityResolution[123]	ER	H	95136	1000	64	1000
RandomForest1 [123]	RF1	H	75340	3767	3	3767
Snort [123]	Snort	H	69029	2687	133	4166
ClamAV[123]	CAV	H	49538	515	542	515
Brill [123]	Brill	M	42658	1962	38	1962
Protomata [123]	Pro	M	42009	2340	123	2365
Fermi [123]	Fermi	M	40783	2399	13	2399
PowerEN [123]	PEN	M	40513	2857	44	3456
RandomForest2 [123]	RF2	M	33220	1661	3	1661
TCP [30]	TCP	L	19704	738	100	767
Dotstar06 [30]	DS06	L	12640	298	104	300
Ranges05 [30]	Rg05	L	12621	299	94	299
Ranges1 [30]	Rg1	L	12464	297	96	297
ExactMath [30]	EM	L	12439	297	87	297
Dotstar09 [30]	DS09	L	12431	297	104	300
Dotstar03 [30]	DS03	L	12144	299	92	300
Hamming [123]	HM	L	11346	93	20	186
Levenshtein [123]	LV	L	2784	24	23	96
Bro217 [30]	Bro217	L	2312	187	84	187

of an AP half-core (24K). The rest of the applications are grouped into low resource requirement (low) group.

3.6.2 Experimental Setup

We build our mechanisms on top of the open-source virtual automata simulator – VASim [124]. As we mentioned in Section 3.5, we evaluate both AP–CPU and BaseAP/SpAP execution. In the AP–CPU execution, the states that are executed in the SpAP mode are instead executed on the CPU. Table 3.3 shows a summary of the evaluated scenarios. We model different timing mechanisms for AP–CPU and BaseAP/SpAP in the simulator as detailed below.

Timing AP–CPU Execution. We record the total amount of time that the CPU spends

Table 3.3: Summary of Execution Scenarios. © 2018 IEEE.

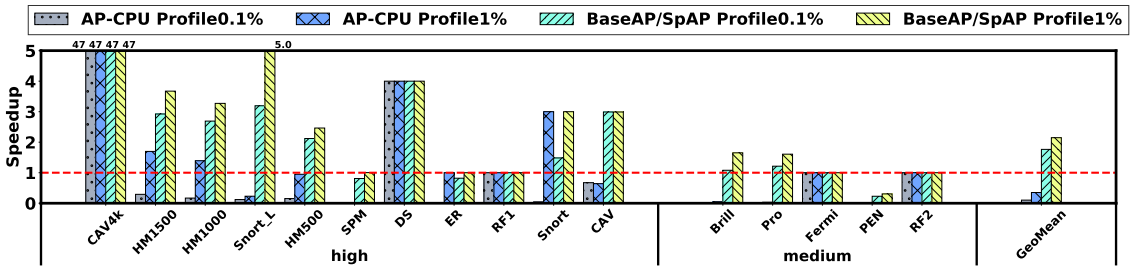
System	Software	Hardware		
		Execution of entire NFAs	Execution of predicted hot set	Execution of predicted cold set
AP	Partition (at NFA granularity)	BaseAP Mode	N/A	N/A
AP–CPU	Partition (hot/cold set)	N/A	BaseAP Mode	CPU
BaseAP/SpAP	Partition (hot/cold set)	N/A	BaseAP Mode	SpAP mode

to handle the intermediate reports by using `std::chrono` in C++ library. Therefore, we use the real time when we calculate the speedup in the AP–CPU execution. We run our experiments on a machine with Intel(R) Xeon(R) CPU E5-2683 v3. We use 7.5 ns as the cycle time per symbol [106] for the BaseAP execution.

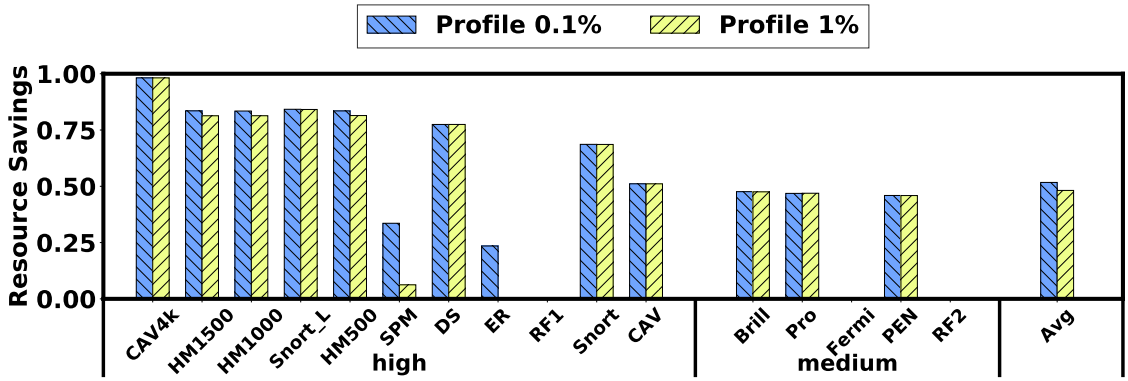
Recording the Cycles in BaseAP/SpAP Execution. In the BaseAP/SpAP execution, we record the execution cycles via the simulator. The number of cycles in BaseAP/SpAP execution is the sum of cycles spent on BaseAP mode and SpAP mode. Therefore, $Speedup_{BaseAP/SpAP} = \frac{Number\ of\ cycles\ on\ AP\ baseline\ execution}{Number\ of\ cycles\ on\ BaseAP\ Mode + Number\ of\ cycles\ on\ SpAP\ Mode}$.

Performance per STE. We define a metric called *performance per STE* to show how much throughput each STE can provide on average. Specifically, $performance\ per\ STE = \frac{throughput}{C_{AP}}$, where $throughput = \frac{number\ of\ input\ symbols}{number\ of\ cycles}$. This allows us to compare APs with different capacities while also considering techniques that improve performance solely by increasing the AP size. Because each STE in the AP occupies die area, we can also consider this metric as a proxy for performance/area.

Overheads. In this paper, we focus on reducing the re-execution overhead as we found it is the major performance bottleneck in AP. The new SpAP mode incurs the stall cycles due to simultaneous intermediate reports (Section 3.5.2). Our final results include these stall cycles. There are two more generic overheads related to output and reconfiguration. In our evaluations, we do not include the output overhead [5] and rely on existing work [122] that proposes both hardware and software techniques to address it. We also do not include the reconfiguration overhead (50 ms [92, 126] for reconfiguring a full AP board) in our



(a) Speedup with AP–CPU and BaseAP/SpAP execution using 0.1% and 1% profiling input (capacity = 24K).



(b) Resource savings (i.e., the portion of states that are not configured in the BaseAP mode)

Figure 3.10: Speedup and Resource Savings on AP. © 2018 IEEE.

results as we believe it can be amortized over AP execution, especially when it executes very large inputs.

3.7 Experimental Results

Effect on Performance. To show the benefits of our schemes, we evaluate the speedup for applications in the high and medium groups. Our mechanisms do not change the throughput of AP for applications in the low category since the sizes of applications are smaller than our baseline AP with 24K STEs. Figure 3.10(a) shows the performance results of our proposal, from which we draw four major observations. First, The AP–CPU execution shows a significant geometric mean slowdown of 9.8× and 2.9× under 0.1% and 1% profiling input, respectively. However, five applications out of 16 applications (`CAV4k`, `HM1500`, `HM1000`, `DS`, `Snort`) achieve a 4.2× geometric mean speedup at no cost of hard-

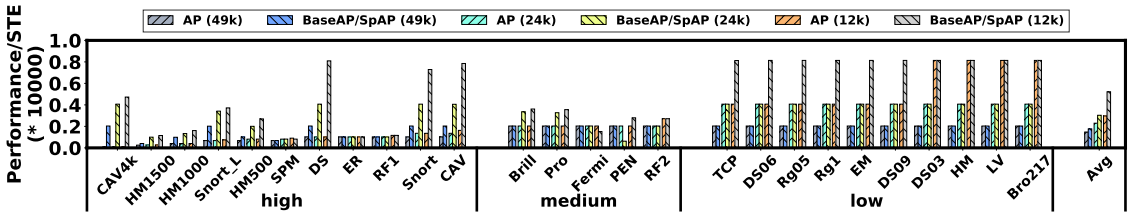


Figure 3.11: Performance per STE of various AP sizes with BaseAP/SpAP execution considering 1% profiling input. © 2018 IEEE.

ware modification. Second, we find that BaseAP/SpAP execution shows a speedup in the majority of evaluated applications. It can achieve 1.8× and 2.1× geometric mean speedup using 0.1% and 1% of input as profiling input, respectively. Third, BaseAP/SpAP execution can be slower than the AP in a few applications (e.g., `PEN`), since these applications generate many simultaneous intermediate reports, leading to lengthy enable stalls on the SpAP mode (shown in Table 3.4). Fourth, in applications with large SCCs that prevent efficient partitioning (e.g., `ER`, see Figure 3.8), our scheme configures all the states to the BaseAP mode execution with no change in execution time.

Effect on Performance per STE. In order to evaluate the efficiency of our schemes across a wider set of system sizes and configurations, we show *performance per STE* in Figure 3.11, from which we draw two major observations. First, although different sizes of AP chips can execute the same application with the same performance (e.g. an application in *low* group fits and runs on both an AP chip or an AP half-core), larger AP chips have less performance/STE, because fewer STEs in the larger AP are utilized for the same application size. Such underutilization leads to less performance/STE. Second, on average, our scheme not only increases performance/STE by 32.1% under the scenario of AP half-core and using 1% profiling input, but consistently achieves better performance/STE under different sizes of AP as well. There are two major reasons: (1) we predict cold states and eliminate them from being configured, which increases AP utilization; (2) we use fewer cycles in the SpAP mode for mis-prediction handling than re-execution by batches hence increasing the throughput.

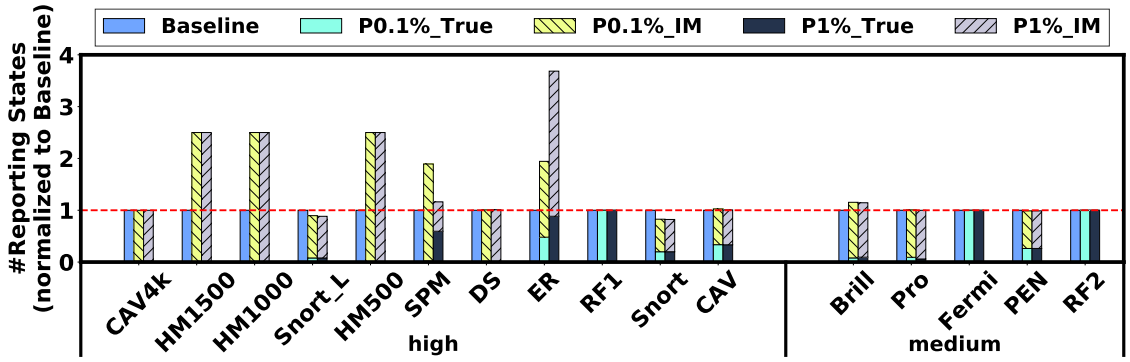


Figure 3.12: Comparison of number of reporting states: "IM" stands for intermediate reporting states. "True" stands for original reporting states on BaseAP mode. "P" stands for profiling. © 2018 IEEE.

Resource Savings and Speedup. We show the results of resource savings in Figure 3.10(b). By comparing it with Figure 3.10(a), we make three observations. First, generally, the applications with high resource savings also have good speedups. Second, `PEN` shows slowdown although it has good resource savings. This is because its SpAP mode execution has lots of enable stalls due to a large amount of simultaneous intermediate reports (Table 3.4). Third, although the resource savings may be the same under different profiling inputs, the speedup may be different (e.g., `Snort`). It is because the original size of the predicted hot set was different, but due to the optimization in Section 3.4.2, each batch was extended with a part of the predicted cold states to match the capacity of AP. Consequently, this leads to the same resource savings. However, since larger profiling input has higher *recall* for hot states (Section 3.4.1), the speedup is also higher. To conclude, the speedup is generally related to resource savings as we explained in Section 3.3.3, but the speedup also depends on other factors such as the quality of prediction and the number of enable stalls.

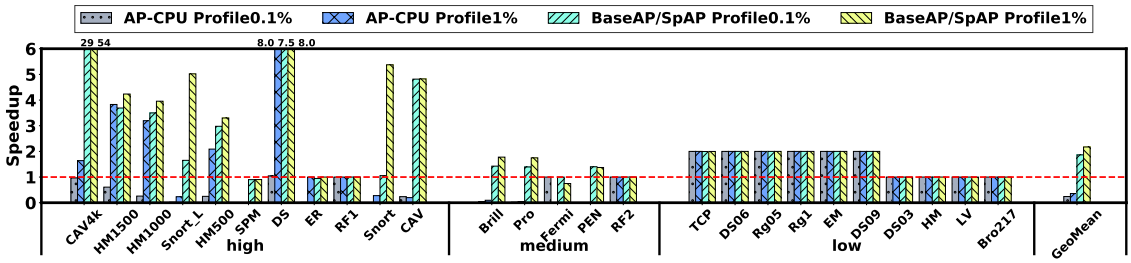
Intermediate Reporting States. The addition of intermediate reporting states increases the total number of states which could increase the total number of configurations and executions (e.g., `HM500` in Table 3.4). Figure 3.12 shows the effect on the number of reporting states in BaseAP mode normalized to that of the baseline. In the BaseAP/SpAP

Table 3.4: Runtime statistics for AP and BaseAP/SpAP (under 1% profiling input): The first three columns show the number of executions on the AP, BaseAP mode and SpAP mode, respectively. "EStalls" stands for the stalls caused by enable operations for handling simultaneous intermediate reports. "JumpRatio" is defined as the proportion of cycles skipped in the SpAP mode.

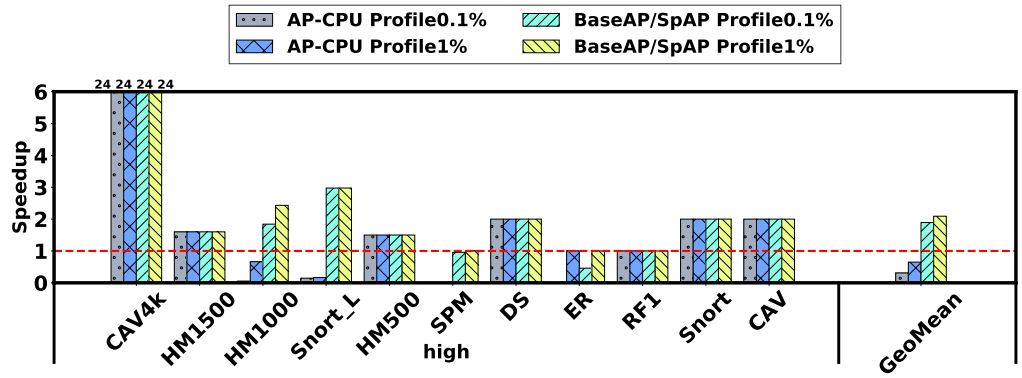
	#Baseline Execution	#BaseAP/SpAP Execution		BaseAP/SpAP Runtime Statistics		
App	AP	BaseAP Mode	SpAP Mode	#Intermediate Reports	#EStalls	JumpRatio
CAV4k	47	1	0	0	0	-
HM1500	15	4	13	80680	248	99.37%
HM1000	10	3	9	54075	180	99.39%
Snort_L	6	1	5	172665	87882	97.99%
HM500	5	2	5	27815	79	99.43%
SPM	5	4	1	63490	119	2.11%
DS	4	1	0	0	0	-
ER	4	4	0	0	0	-
RF1	4	4	0	0	0	-
Snort	3	1	2	70	4	99.99%
CAV	3	1	1	3215	0	99.67%
Brill	2	1	1	68125	23997	81.51%
Pro	2	1	1	89733	15862	77.43%
Fermi	2	2	0	0	0	-
PEN	2	1	1	5450318	4509743	1.96%
RF2	2	2	0	0	0	-

mode, the total number of reporting states includes both original reporting states and intermediate reporting states (stacked bars in the figure). We make two observations. First, the total number of reporting states in BaseAP mode could be more than the baseline AP execution that only contains original reporting states. For example, the number of reporting states in `ER` increases by 3.6×, because it has a large number of crossing edges between predicted hot set and predicted cold set. Second, the number of reporting states could decrease (e.g., `Snort` and `Snort_L`) in the BaseAP mode execution because the number of crossing edges is smaller than the number of original reporting states. Although our scheme may increase the number of reporting states, we are aware that an effective software-based reporting state compression technique [122] could be applied on top of our scheme.

Effect of Jump Operations. In Table 3.4, although for some applications (e.g., `HM500`,



(a) Speedup on a small AP (capacity = 12K)



(b) Speedup on an AP chip (capacity = 49K)

Figure 3.13: Sensitivity on the different capacities of AP chip. © 2018 IEEE.

`Brill`) the number of executions of BaseAP/SpAP may be greater than or equal to the baseline, we still obtain speedups on them because SpAP mode execution can reduce total number of cycles due to the *jump* operations. To show the effect of jump operations, we define *JumpRatio* as the proportion of cycles skipped in the SpAP mode. Formally, $JumpRatio = 1 - \frac{Total\ cycles\ on\ SpAp\ mode}{Number\ of\ batches\ on\ SpAP\ mode \times Length\ of\ input\ stream}$. Higher *JumpRatio* indicates better effect of *jump* operations. We show *JumpRatio* in Table 3.4 for the applications that use SpAP mode. To conclude, the majority of the applications only execute a few percent of input symbols with the help of jump operations.

Sensitivity of speedup on capacity of AP. The applications in the low resource requirement group require fewer states than the capacity of AP half-core. Figure 3.13(a) shows the speedup achieved by our schemes when the capacity of AP is 12K. Similar observations still hold as discussed in Figure 3.10(a). Specifically, BaseAP/SpAP achieves 1.9× and 2.2× speedup using 0.1% and 1% profiling input, respectively. In addition,

we demonstrate another sensitivity study on AP with 49K STEs for the applications in the high group. Figure 3.13(b) shows BaseAP/SpAP execution achieves 1.9× and 2.1× speedup using 0.1% and 1% profiling input in the 11 applications of this group.

3.8 Related work

To the best of our knowledge, this is the first work that designs an efficient architectural support for large-scale NFA applications on AP.

Spatial Architectures. Multitasking on spatial architectures is usually carried out through the use of multiple contexts [50], which can consume extra memory. In contrast, our BaseAP/SpAP proposal relies on the ability to eliminate dynamically unused states from NFAs to improve AP utilization. We rely on a mechanism to transfer control to a spatially distinct partition to accommodate larger than device NFAs, though these could be implemented as multiple contexts. Recently, gate removal has been proposed to eliminate unused logic gates from general purpose processor IPs to customize processors to specific applications [41]. In our approach, we only eliminate states from the NFA (i.e., the program), and not the hardware. There are also alternative implementations of AP [59, 107, 139]. For example, cache automaton [107] re-purposes the last-level cache for automata processing. We believe our techniques are complementary as we propose hardware/software mechanisms to make the automata processing itself more efficient.

DFA and NFA Acceleration. Deterministic finite automata (DFA) have been characterized previously – with respect to implementing special machines [119] and for parallelization [74, 149, 84, 148, 85]. Parallel execution of NFAs on the AP processor has been proposed by trading AP resources for higher throughput [106]. However, our characterization of the dynamic execution properties of NFAs specific to the AP execution model is, to our knowledge, the first of its kind. Our elimination of dynamically unused states can free up AP resources to complement parallel execution.

FSM Decomposition. FSM decompositions [72, 70, 22, 23] could reduce the complex-

ity of placement and routing in the routing matrix by simplifying the layout. While cascade decompositions are the closest to our studies, they are often static, for deterministic machines only, and are mostly not based on dynamic state behavior (i.e., predicted hot vs. predicted cold states). In contrast, our proposed approach (which uses graph-theoretic techniques, rather than sequential machine theory) is focused on increasing the AP throughput by allowing only predicted hot states to be configured to the AP. We believe both approaches are complementary and can be applied to different bottlenecks in the AP execution pipeline. For example, FSM decomposition can make the *reconfiguration* process efficient while our technique can accelerate the NFA execution on AP by reducing the number of *re-executions* of the input symbol stream.

3.9 Conclusions

Automata processors (AP) are very efficient in executing Non-deterministic Finite Automata (NFAs). However, like other types of spatial architectures, AP faces major challenges in its execution model to efficiently execute very large tasks. In this paper, we make use of the inherent properties of NFAs to avoid using compute resources for states that are never used during execution by a low-cost software/hardware-coordinated approach. Consequently, this results in a new execution model for APs that enables efficient and high-performance processing for large-scale tasks. We believe this work will be helpful towards wider adoption of APs and will open up new research directions for enabling efficient NFA processing.

Chapter 4

Why GPUs are Slow at Executing NFAs and How to Make them Faster

Non-deterministic Finite Automata (NFA) are space-efficient finite state machines that have significant applications in domains such as pattern matching and data analytics. In this paper, we investigate why the Graphics Processing Unit (GPU)—a massively parallel computational device with the highest memory bandwidth available on general-purpose processors—cannot efficiently execute NFAs. First, we identify excessive data movement in the GPU memory hierarchy and describe how to privatize reads effectively using GPU's on-chip memory hierarchy to reduce this excessive data movement. We also show that in several cases, indirect table lookups in NFAs can be eliminated by converting memory reads into computation, to further reduce the number of memory reads. Although our optimization techniques significantly alleviate these memory-related bottlenecks, a side effect of these techniques is the static assignment of work to cores. This leads to poor compute utilization, where GPU cores are wasted on idle NFA states. Therefore, we propose a new dynamic scheme that effectively balances compute utilization with reduced memory usage. Our combined optimizations provide a significant improvement over the

previous state-of-the-art GPU implementations of NFAs. Moreover, they enable current GPUs to outperform the domain-specific accelerator for NFAs (i.e., Automata Processor) across several applications while performing within an order of magnitude for the rest of the applications.

4.1 Introduction

Finite automata are the workhorses of pattern matching, data analytics, malware detection, bio-informatics, and XML parsing among many other applications [11, 8, 12, 90, 152, 151, 89, 112, 6, 34, 99, 93, 13, 33, 103]. Two representations of finite automata—non-deterministic finite automata (NFAs) and deterministic finite automata (DFAs)—are commonly used in the implementation of finite automata based applications [43]. Although DFAs are simpler in terms of transitions, DFA execution is embarrassingly serial, and DFAs can be exponentially larger than equivalent NFAs [149, 103, 141]. Prior work [74, 149, 148, 84, 86, 55] significantly reduces the latency of DFA execution by parallelizing chunks of the input stream and resolving the dependencies across states. However, current enumeration or speculation mechanisms increase parallelism which is not always needed, especially for large-scale automata applications.

The non-deterministic nature of NFAs lends itself naturally to parallel execution leading to a number of NFA accelerators [107, 97, 98, 153, 106, 95, 47, 61]. In particular, the Automata Processor (AP) proposed by Micron [44] is an in-memory accelerator for NFAs. The AP achieves significant throughput and energy benefits because of its ability to perform in-memory computations that exploit the parallelism of NFAs [44, 126]. However, APs have to deal with several challenges. First, APs can hold a limited number of NFA states at a time and when executing large-scale workloads need repeated re-executions and re-configurations that hamper throughput significantly [66]. Second, their multiple-instruction single data (MISD) model means their ability to execute multiple input streams in parallel is limited.

On the other hand, GPUs are massively parallel accelerators that are widely used. Execution of NFAs on highly parallel architectures like GPUs, therefore, appears very attractive. However, NFA-based applications are very hard to accelerate on traditional von Neumann architectures (e.g., CPUs and GPUs) [44, 76, 88, 111]. In this paper, we address this hard problem with the help of a careful analysis of the bottlenecks of NFA execution on GPUs. Specifically, we find that there are two main problems. First, a typical NFA execution incurs significant data movement because for processing each input byte, a large transition table stored in the global memory needs to be looked up. Solutions to reduce this data movement invariably use a fixed mapping of NFA states to GPU threads, which leads to the second problem—hardware under-utilization. Many NFA states are not active (cold) in a given NFA, so a fixed mapping leads to idle threads. These idle threads unnecessarily consume GPU resources and *do not* perform any *useful* work leading to low throughput and poor hardware utilization. Overall, high data movement and poor hardware utilization are the major sources of inefficiencies.

We solve the first problem by identifying that the transition table used in most implementations contains redundant entries and is also highly sparse. Therefore, we propose a new compact data structure to access transition information that can significantly reduce off-chip memory accesses. We also show that some indirect table lookups can be eliminated by converting them into local computations. However, we find that such data movement optimizations require an undesirable fixed mapping between threads and states. Therefore, we solve the second problem by developing a hybrid mapping where the most active (hot) NFA states receive a static mapping and all other states are assigned resources dynamically. Our mapping scheme not only significantly boosts *useful* work as most threads are assigned to active states, but also allows more NFAs to execute concurrently. To the best of our knowledge, in the context of NFA processing, no prior work has considered both data movement and utilization problems in conjunction. In summary, this paper makes the following contributions:

- This paper analyzes the bottlenecks of NFA execution on GPUs and finds that

high data movement and low utilization are the two major inefficiencies leading to low throughput.

• We find that the data movement problem is due to the irregular accesses to the transition table. This table is too large to fit in the on-chip resources due to significant redundancy and sparsity in the transition table. Our solution stores the topology and matchset information in a novel compact format such that it can be stored and accessed from the on-chip resources for most NFA states.

• We find that utilization is low because not all states are active. Hence, we take advantage of state activity information to assign one thread per hot state and other cold states are executed on-demand. This improves utilization as a result of increased activity of threads.

• Overall, our mechanisms outperform the state-of-the-art work in this area. Specifically, across 16 NFA applications, the best of our schemes improves the throughput on average by 26.5× over iNFAnt [36] and 5.3× over NFA-CG [154]. Further, we only require 0.7% of the global memory transactions used by iNFAnt.

4.2 Background

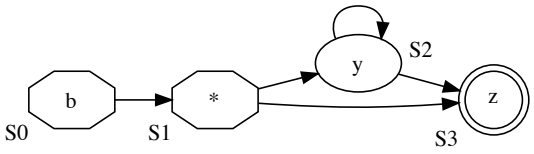
This section describes NFAs and their processing on GPUs.

4.2.1 Pattern Matching via NFAs

A non-deterministic finite automaton[1] (NFA) is a directed graph where each node represents a state and each edge represents a state transition. Every state in the NFA has a *matchset* that contains the alphabets (symbols) it *matches*. Every NFA has at least one *start state* and at least one *reporting state*. The matching process begins by activating the *start states*. An NFA consumes one symbol at a time from the input stream.

[1]In this paper, we focus on Glushkov NFAs [51]. They are ϵ-free and the matchset is on the node instead of on the edge. Any NFA that accepts a non-empty string can be transformed into an equivalent Glushkov NFA [51].

For each symbol, all currently *active* states attempt to match the incoming symbol with their matchset. If any of them match, they activate their successors. Unlike deterministic finite automata, where only one state is active, NFAs can have multiple states active simultaneously—making them ideal for parallel architectures. If a *reporting state* matches an input symbol, it generates a report showing that a relevant pattern has been observed in the input stream. Usually, all starting states are *always-active*, unless a user wants to search patterns that only start at a certain position of the input stream.



(a) NFA accepting pattern `b*.y*z`: the start states are shown in octagons and the reporting states are shown in double-circles.

	S0	**S1**	S2	S3
...		S2, S3		
b	S1	S2, S3		
...		S2, S3		
x		S2, S3		
y		S2, S3	S2, S3	
z		S2, S3		report

(b) Illustrating a transition table lookup. If the input symbol is `x` and the current active states are S0 and S1, then the shaded cells are fetched and S2, S3 become active.

Figure 4.1: Working example of an NFA.

For example, Figure 4.1 (a) shows an NFA accepting pattern `b*.y*z`. It has two start states S0 and S1, and a reporting state S3. Suppose S0 and S1 are active and the incoming symbol is `x`. Since S1 matches `x`, its successors S2 and S3 become active. S0 and S1 are always-active, so in the next step, the active states are S0, S1, S2, S3.

4.2.2 NFA Processing on GPUs

GPUs support concurrent execution of a large number of threads and also have very high memory bandwidth – orders of magnitude more than CPUs. NFAs are a good fit for GPUs because they exhibit parallelism at multiple levels [76]. Consider the NFA processing mechanism as shown in Algorithm 2. First, multiple input streams (e.g., different network packets) can be processed in parallel (Line 2). Second, many NFAs (e.g., different intrusion signatures) can run in parallel on the same input stream (e.g., a single network packet, Line 3). Finally, within the same NFA and the same input symbol, multiple states can be

active at the same time (Line 8). Therefore, there are multiple sources of parallelism: (1) input stream level parallelism, (2) NFA-level parallelism, and (3) state-level parallelism.

Algorithm 2 Parallelism in NFA Processing

```
1:  procedure NFA_PROCESSING
2:      for all input stream ss do                          ▷ Input stream-level parallelism
3:          for all NFAs n do                               ▷ NFA-level parallelism
4:              Process_Input_Stream(n, ss)
5:  procedure PROCESS_INPUT_STREAM(n, ss)
6:      Initialize starting nodes in active_bitset
7:      while i < ss.length do                              ▷ Process each symbol serially
8:          for all s in n do          ▷ State-level parallelism; s: NFA State, n: NFA
9:              if active_bitset[s] then
10:                 tablecell ⇐ T[ss[i]][s]                 ▷ Transition table lookup
11:                 if 'report' in tablecell then
12:                     report(s, i)
13:                 for all c in tablecell do           ▷ Matched state activates successors
14:                     next_active_bitset[c] ⇐ 1
15:         active_bitset ⇐ next_active_bitset
16:         Zero next_active_bitset
17:         i ⇐ i + 1
```

A General Approach for NFA Processing on GPUs. Given that NFAs must process each input symbol serially (Line 7 in Algorithm 2), and that most GPUs do not support a cheap global barrier, it is natural to map an entire NFA to a single *thread block*, which is a group of threads that can execute a hardware barrier. This hardware barrier can be used to step through the input stream. An application usually contains many NFAs (Table 4.2) with different sizes. However, all thread blocks are of the same size, so actual implementations will pack multiple NFAs into the same thread block forgoing some NFA-level parallelism. A naïve implementation would then map an individual state to a thread in the thread block. Each thread block then maintains two bitsets, one showing which states are active in the current step and another identifying those that will be active in the next step. When an active state matches the current input symbol, its successors are set in the next active bitset (Line 13-14).

A *transition table* lookup combines state match and fetching successors. Prior work [36,

154, 18, 118, 37, 115, 117, 130], for example, use variants of a transition table where each row is indexed by a symbol α and each column is indexed by a state S—an *alphabet-oriented transition table*. Each entry (or cell) of the table contains the successors of S when S matches with α. If S is a reporting state and matches with α, a report is generated.

In our example, Figure 4.1 (b) shows an example of the transition table lookup for the NFA shown in Figure 4.1 (a). Assume the incoming symbol is `x` (Figure 4.1 (b)), and the current active states are S0 and S1. The threads assigned to the two states, therefore, fetch the two shaded cells from the transition table. States S2 and S3 are set to the next active bitset along with the always-active states (S0, S1 in this NFA). When all states have processed the current input symbol, we synchronize the threads using a `__syncthreads()` barrier, swap the current and next active bitsets, and reset the next active bitset (Line 15-17).

4.3 Problem and Previous Efforts

In this section, we first characterize the problem of high data movement and low compute utilization when processing NFAs on GPUs. We discuss the high-level reasons for these inefficiencies followed by a discussion on how previous works attempt to address them.

4.3.1 Data Movement

NFAs on general-purpose processors read from memory for three reasons: checking the active bitset, loading the input symbols, and accessing the transition table. Of these, the active bitset can be stored in the GPU on-chip shared memory. The input streams and the transition table, though, reside in global memory in the previous works. Indeed, the alphabet-oriented transition table size is $O(N \cdot A)$, where N is the number of states, and A is the size of the alphabet (256 in our work). This is too large to fit in on-chip memories or registers.

Accessing global memory for NFAs incurs performance overheads. Consider that *each* thread must read the *entire* input stream. In the ideal case, all these requests would be satisfied from the cache, but this is not guaranteed. It is also not possible to omit the memory accesses caused by loading the bytes of input streams. Another source of global memory accesses are lookups of the transition table. Each active state must, based on the current symbol, look up a cell to identify successors to activate. Assuming 32-bit state identifiers and an average of 4 successors to activate, each active state must read 16 *additional* bytes per input symbol, which is a significant overhead. With the help of our optimizations, we shall show later that these additional reads can be reduced.

4.3.2 Compute Utilization

As nodes in an NFA are only activated based on the sequence of input symbols, many states in an NFA are never or very rarely activated [66]. Therefore, having a one-to-one mapping between states to threads means several inactive states would waste thread resources leading to poor utilization and throughput. Across all the evaluated applications (Section 4.6), we observe that only a small percentage of states are active during the execution. The average and maximum percentages of active states are 0.39% and 3.05%, respectively. Although this percentage still implies hundreds to thousands of active states—more parallelism than a CPU could handle—a large fraction of GPU threads are still idle.

It is instructive to examine this problem from the perspective of graph processing algorithms. This style of NFA processing would be classified as topology-driven [75], which is known to be work-inefficient. In those algorithms, therefore, a worklist containing the frontier of active vertices is maintained. The threads are mapped only to the active vertices, thus utilization is 100% since threads do only useful work. To verify the feasibility of a worklist approach, we use the IrGL compiler [79] to generate worklist versions for NFAs. Using a 1MB input stream, we found that in Brill, the best version achieves 114KB/s. By contrast, iNFAnt [36] achieves 143KB/s. Hence, applying the worklist

directly for NFA processing on GPU is not efficient for two reasons. First, unlike graph processing, some states (i.e., nodes) are always active in NFA processing. Second, NFAs require synchronization after each input symbol and perform very little work per input symbol. Therefore, maintaining a worklist incurs high overhead and hence it is critical to have a lightweight and efficient way to increase utilization.

4.3.3 Limitations of Prior Efforts

iNFAnt [36] uses a variant of the alphabet-oriented transition table described earlier. The columns represent edges in the NFA, not states, and the rows continue to represent alphabet symbols. In this table, a cell can have at most one state. For example, an edge $u \rightarrow v$ that matches on symbol `a` creates a column for u, whose row for `a` contains v. During execution, each column (and therefore an edge (u, v)) is mapped to a single GPU thread. Each thread checks if its assigned state u is active in the active bitset, and activates v if the current input symbol is `a`. iNFAnt does not perform any special optimizations for data movement and utilization.

Zu et al. [154] introduce the notion of *compatible groups*, where the states that belong to the same compatible group cannot be active simultaneously. This allows a compatible group to be assigned to a single thread, improving utilization. However, this approach, which we name NFA-CG, uses a very expensive method to compute compatible groups – its time complexity is at least quadratic in the number of NFA states and other non-heuristic methods are exponential. Our proposal for improving utilization is linear in the number of states, and also achieves better utilization.

Ideally, the only memory loads should be for the input symbols. Consider a kernel launched with T thread blocks (each containing W warps) and the length of the input stream in symbols is L. In such a case, the number of global memory load transactions for input only is TWL. Figure 4.2 shows the number of global load transactions for existing works iNFAnt [36] and NFA-CG [154] normalized to the ideal case for input-only memory transactions. On average, iNFAnt has 25× more transactions, and NFA-CG has 18× in

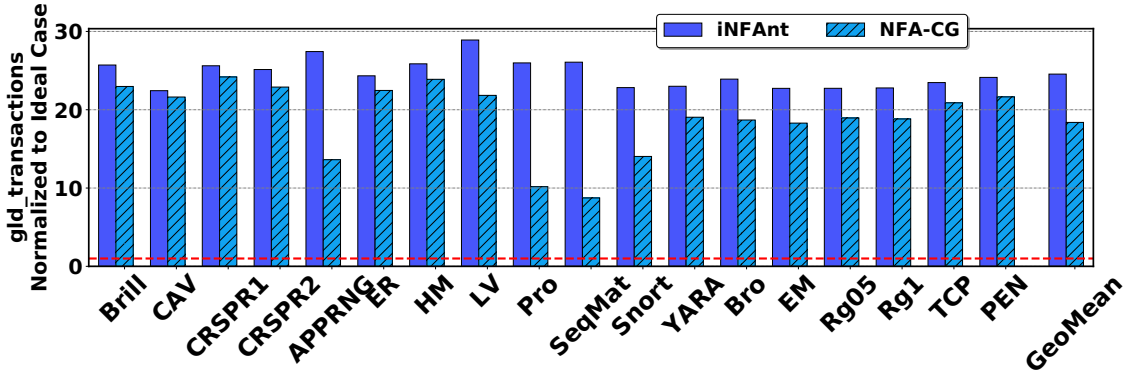


Figure 4.2: The data movement normalized to the ideal cases: two prior schemes use 25× and 18× compared to the ideal case where only the input stream is loaded. The evaluation methodology is discussed in Section 4.6.

the evaluated applications.

4.4 Addressing the Data Movement Problem via Matchset Analysis

In this section, we first analyze the inefficiencies associated with the alphabet-oriented transition table. We discuss how addressing these inefficiencies can reduce off-chip accesses and alleviate the problem of data movement.

4.4.1 Inefficiencies in the Transition Table

As discussed in Section 4.2, the existing transition table stores both the matchset and NFA topology information. Instead of checking whether the current input symbol is present in the matchset of the current state, it converts this computation to transition table lookups. However, we find that the resultant transition table can no longer fit in the GPU on-chip memory as storing the combination of matchset and NFA topology introduces redundancy and increases sparsity.

To understand and quantify the volume of redundancy and sparsity in the transition table, consider Figure 4.3 which shows two metrics. Redundancy is defined as the ratio

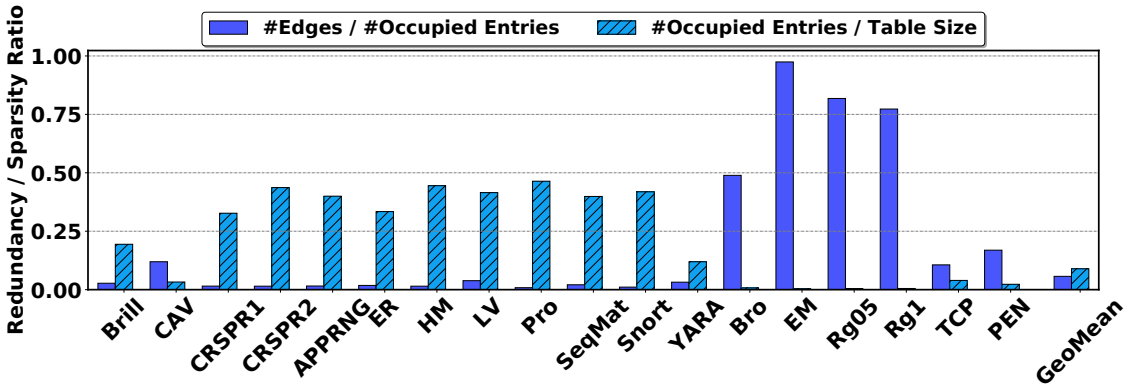


Figure 4.3: Two metrics showing the redundancy (#edges/#occupied-entries) and sparsity (#occupied-entries/table-size) in the transition table. Lower is worse.

of the total number of edges across all NFAs in the application to the total number of non-empty (or occupied) entries in the transition table. As the number of edges can only be less than (or equal to) the number of occupied entries, a low ratio indicates higher redundancy since an edge is stored in multiple locations of the transition table. Sparsity is defined as the ratio of occupied entries to the total number of entries in the transition table. A low ratio for sparsity suggests that not all transition table entries are occupied. We observe from Figure 4.3 that on average both metrics are very low across all the evaluated NFA applications showing that alphabet-oriented transition table wastes a lot of memory.

From our discussion in Section 4.2, we can identify two reasons that lead to these inefficiencies. First, an edge in the NFA can occur multiple times in the transition table. For example, all entries in column S1 in Figure 4.1 (b) store the same value (S2, S3) because S1 accepts a wildcard. In general, if a state accepts k symbols, all its outgoing edges have to be stored k times. Second, in most NFA applications, a large percentage of states only accept a few symbols. For example, the column for S0 only contains one entry as S0 only accepts `b`, but the entire column has to be kept in the transition table which makes it sparse. In conclusion, there is excessive redundancy and sparsity in the transition table. This makes it an inefficient way to store the matchset and topology information.

4.4.2 Optimization I: A New Way to Store and Access Matchset and Topology Information (NewTran/NT)

To reduce the excessive data movement problem incurred by the transition table, we propose a new way to store matchset and topology information. The key idea is to create a per-node data structure that contains the node's: a) matchset, b) outgoing edges, and c) other miscellaneous attributes. This per-node data structure is stored only *once* eliminating redundancy. Furthermore, we avoid storing the complete Cartesian product of alphabets and states in an off-chip transition table addressing the sparsity problem. Our proposal converts the per symbol look-ups of the transition table to a one-time memory access per state, which makes the data movement of our scheme close to the ideal case as shown in Section 4.3.3.

Per-node data structure. Figure 4.4 ❶ shows the per-node data structure `NodeInfo`. We use an array of eight 32-bit integers for the 256-bit matchset (`matchset`, 32 bytes). When symbol `a` is examined, each active thread checks for a match by checking if the bit corresponding to `a` is set: `matchset[a / 32] & (1 << (a % 32))`. Since NVIDIA GPUs do not support indexing into a register, the `matchset` must be stored in the local memory which is private to each thread. We will show in Section 4.4.3 how we also put the matchset in the registers when possible. We maintain 4 out edges per node in a 64-bit integer (`outedges`, 8 bytes), which consumes two 32-bit registers per thread. We also need to encode the attributes of a state (as shown in Figure 4.4 ❷), so we maintain these attributes in the 8-bit variable `attributes`, which consumes an additional register. Currently, we use 5 bits of the attribute variable. Three bits record if a state is a *reporting*, *start*, or *always-active* state. Two additional bits record if the *matchset* is *complete* or *complement* to enable the compression optimization described in Section 4.4.3.

Matching process. The per-node data structure is fixed mapped to each thread. Before processing the input symbols, each thread loads the per-node data structure from the global memory. After the data structure is loaded, the thread starts to iterate over the

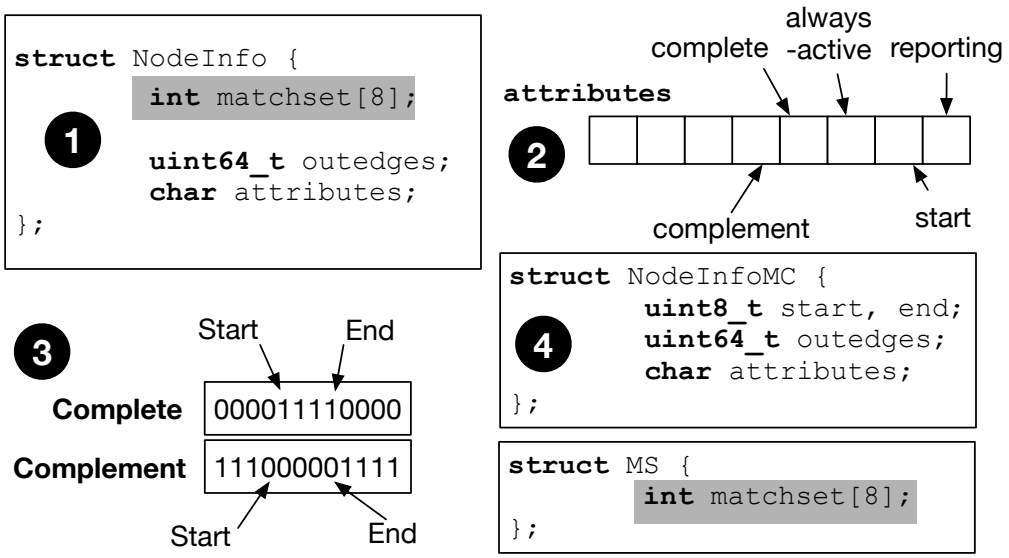


Figure 4.4: Illustrating the per-node data structure of NewTran (NT). Shaded variables are in the local memory and others are in the registers.

input stream. Instead of looking up the transition table for determining a match, each active state compares the incoming symbol against its matchset in the privatized `NodeInfo` data structure. We still keep the double-buffered *active bitset* to record whether a state is active as described in Section 4.2.2. We use an array in global memory to hold the reports generated during the NFA processing. GPU atomic instructions are used to perform concurrent writes to this array.

Space consumption. Each `NodeInfo` data structure consumes $32 + 8 + 1 = 41$ bytes. For N states, this is $41 \times N$ bytes. The alphabet-oriented transition table, on the other hand, requires $256 \times 16 \times N = 4096 \times N$ bytes (256 is the size of the alphabet, while 16 bytes store up to 4 successor states per cell). Hence, our scheme only uses 1% space compared to the alphabet-oriented transition table. The reduced memory consumption enables the execution to better exploit the on-chip resources of GPU for the topology and the matchsets of NFAs.

4.4.3 Optimization II: Matchset Compression (MaC)

Figure 4.4 shows that matchset information is stored in local memory. We find additional opportunity to compress this information to reduce the global memory transactions. To compress the matchset information, we focus on two categories of states:

Complete state. If the matchset for a state, when viewed as a 256-bit string, contains one continuous set of "1"s, we term that state as *complete*.

Complement state. If the matchset for a state is not complete, but its (bitwise) complement is complete, (i.e., ∼`matchset` is complete), we term that state as *complement*.

When a state is either complete or complement, we can represent its entire matchset as a range using only two 8-bit variables, `start` and `end` (Figure 4.4 ❸) to denote the input symbols it matches. Then, for complete states, we can check if the incoming symbol `s` is matched simply by evaluating `s` $\geq$ `start && s` $\leq$ `end`. For complement states, we can simply invert the sense of the result. Thus, all accesses to the matchset can be eliminated by converting the indirect memory read on the transition table to a pure range check computation.

Figure 4.5 shows that a large portion of states are either complete or complement. On average, matchset lookups for 70% of states can be replaced by range checks.

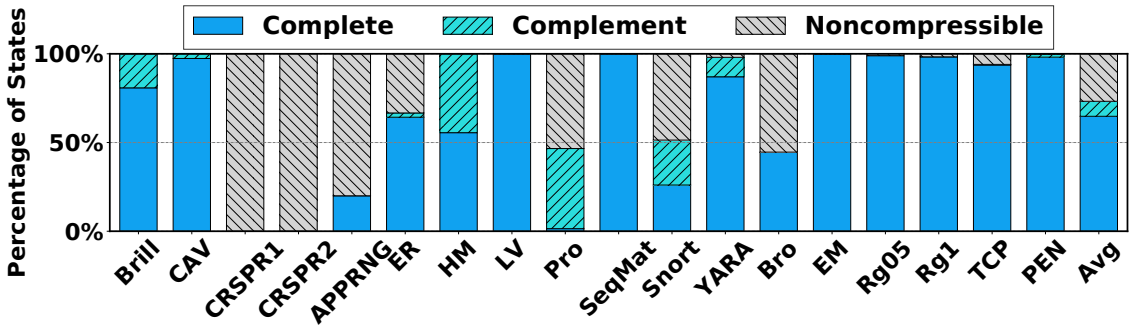


Figure 4.5: Percentage of states whose matchsets are complete, complement, or not compressible.

To implement the complete/complement compression scheme, we split the `NodeInfo` data structure into two structures as shown in Figure 4.4 ❹, namely `NodeInfoMC` and `MS`.

If a state is compressible, we only load `NodeInfoMC`, which uses 16 bits (`start` and `end`) to store the matchset. If the state is not compressible, we load `MS` as before. Depending on whether a state is compressible or not, we load 16 bits or 272 (16 + 256) bits for the matchset. In general, if a fraction p of states are compressible, the average global load per node data structure is $16p + 272(1 - p)$. As the majority of the states are compressible (Figure 4.5), our matchset compression scheme uses fewer loads to the local memory while also reducing the global memory transactions.

4.5 Addressing the Utilization Problem via Activity Analysis

In this section, we focus on addressing the problem of under-utilization as discussed earlier in Section 4.3.2. We first analyze the activity of states and use this information for an intelligent mapping of states to threads. The key idea is to map only the highly active states to dedicated threads and assign resources to remaining low activity states on-demand.

4.5.1 Analysis of Activation Frequency

It follows from Section 4.2 that *always-active* start states do useful work during the entire execution. However, the activity of the other states is not clear. Figure 4.6 shows the CDF of the activation *frequency* of all the non-starting states across seven representative applications. All other evaluated applications are similar to the representative applications. We observe that for the majority of applications, 80% of non-starting states are activated for less than 1% of the processed symbols. Similar behavior was also observed by Liu et al. [66]. However, their study did not consider the frequency of the activity and only evaluated whether the state is active or never-active. Our finding is that although many states can be active at least once, the frequency of activation is usually very low.

We exploit this activity profile in GPUs and propose to map only the states that are

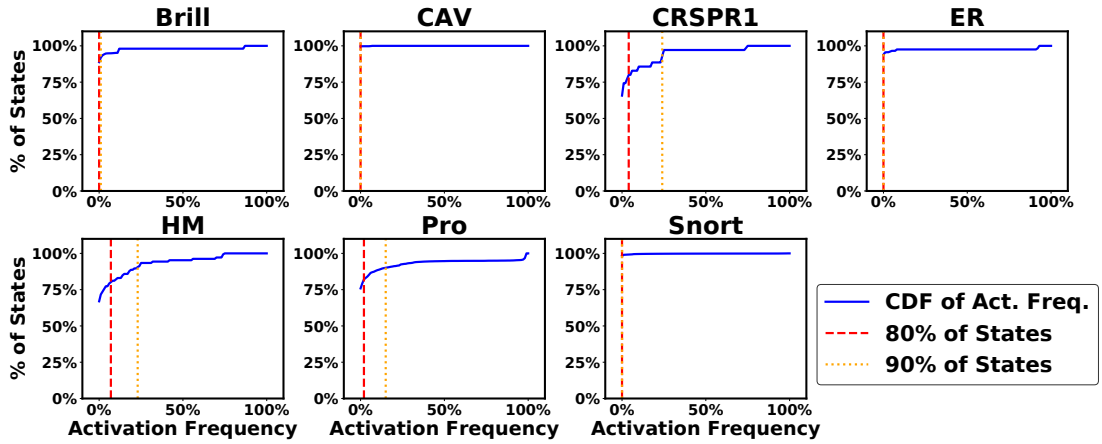


Figure 4.6: The activity profile of the states. For the majority of applications, 80% of non-starting states are activated for only less than 1% of the processed symbols.

activated frequently to the dedicated threads. Other infrequently activated states and are assigned resources on-demand. To accomplish this, we need to answer two research questions: (1) Given a certain mapping, how do we coordinate between fixed mapped *hot* (active) states and the on-demand loaded *cold* (rarely active) states? (2) How do we classify *hot* states and *cold* states? We will answer these two questions in Section 4.5.2 and Section 4.5.3, respectively.

4.5.2 Optimization III: Activity-based Processing

In this section, we discuss how we handle hot and cold states to improve the overall utilization. We propose to develop a hybrid approach that uses one-one mapping (topology-driven) for hot states and a worklist (data-driven) for the cold states. Each hot state is given to a dedicated thread, while cold states are not assigned to any dedicated threads. As in NT, each thread loads the `NodeInfo`[2] data structure for its hot state before processing the input stream. As described in Section 4.2.2, we store whether a hot state is active in the per-block *active bitset*. Each thread also reserves space for an additional `NodeInfo` data structure to be used for any cold states dynamically assigned to it during execution.

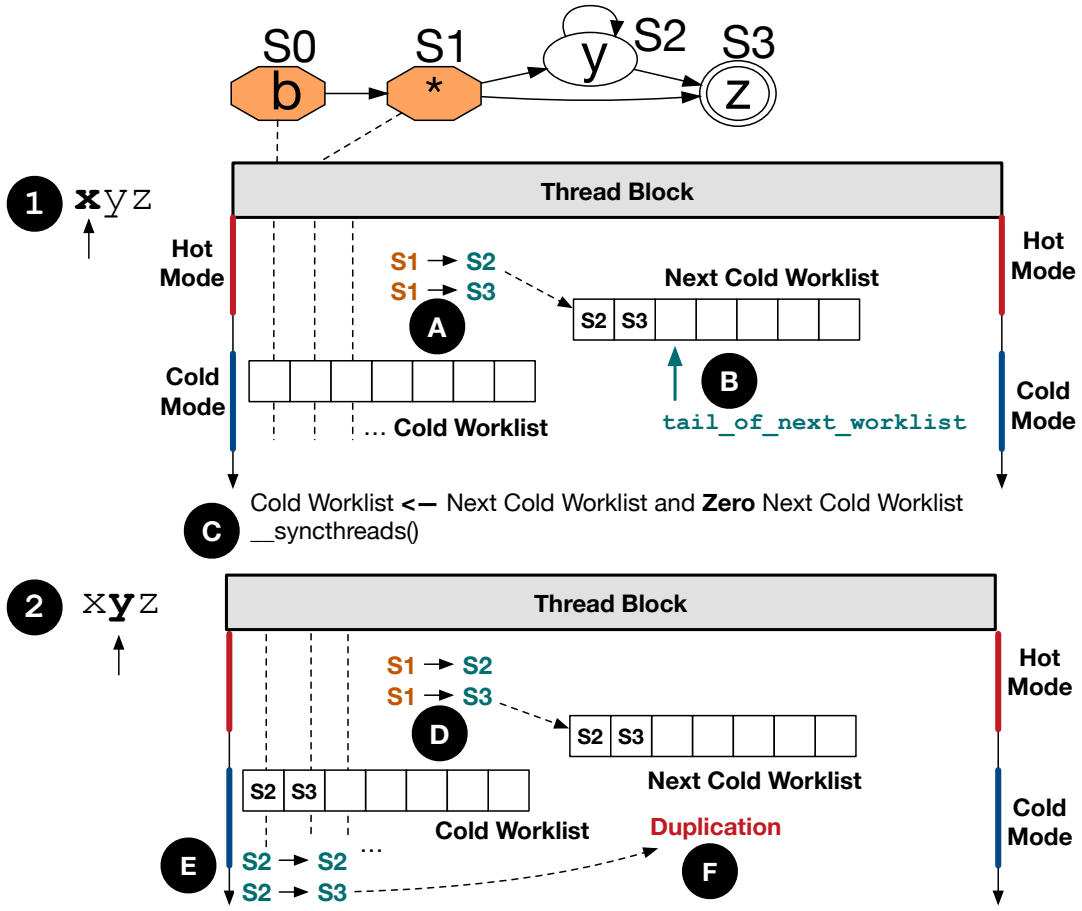
[2]This can be `NodeInfoMC` depending on whether we turn on the matchset compression (Section 4.4.3). We will use `NodeInfo` for the two cases.

Execution for all input symbols takes place in two modes: first, a hot mode (which executes all hot states) followed by a cold mode (which executes any cold states that have been activated). If a hot state activates a cold state, it places the cold state ID in the *next cold worklist*. This worklist is stored in shared on-chip memory and we use a shared *deduplication bitset* to avoid duplication of state IDs in the worklist. After all hot states have completed the processing of the input symbol, the hot mode is complete. Next, execution switches to the cold states in the current cold worklist (CW) populated during the processing of the previous input symbol. If CW is empty, execution skips the cold mode.

If CW is not empty, each thread in the thread block is assigned with one or more states of the worklist. We distribute the elements in the worklist equally across all threads in the thread block. A thread then processes the cold state assigned to it by loading the `NodeInfo` of the cold state from global memory into the reserved cold `NodeInfo`. A cold-to-hot transition is handled by set the bit of the activated hot state in the *active bitset*, while a cold-to-cold transition is handled by placing activated states in the *next CW* if it is not set in *deduplication bitset*.

Before continuing the hot mode of the next symbol, the *next CW* is assigned to the CW, and we reset the tail pointer of the *next CW* emptying it.

Illustrative Example. Figure 4.7a illustrates our activity-based optimizations using an example with a thread block. Assume that the hot states S0, S1 are mapped to the threads and the cold states S2, S3 are processed through the worklists. Figure 4.7a ❶ shows that currently, we are processing the symbol `x` of the input stream `xyz`, and S0 and S1 (both hot states) are active. Symbol `x` (Figure 4.7a ❶) triggers two hot-to-cold transitions (S1 to S2 and S1 to S3, Figure 4.7a Ⓐ). S2 and S3 are pushed to the next cold worklist, by atomically incrementing `tail_of_next_worklist` (Figure 4.7a Ⓑ). After the hot mode, the threads move to the cold mode to process the current cold worklist, which is empty. Since there is no work in this step, the cold mode does not start. In the end of the current step, the next cold worklist and the cold worklist are swapped and the tail pointer of the



(a) An illustrative example for our activity-based processing scheme: orange shaded states are hot (active) states mapped to threads and cold (inactive) states are handled via worklist. The *active bitset* for hot states is not shown.

Input		Hot States		Cold States		Matched States	Actions
Index	Char	Active BS	Next Active BS	CW	Next CW		
1	x	S0 S1	S0 S1	-	S2 S3	S1	S1 activates S2 S3
2	y	S0 S1	S0 S1	S2 S3	S2 S3	S1 S2	S1 activates S2, S3 S2 activates S2, S3
3	z	S0 S1	S0 S1	S2 S3	S2 S3	S1 S3	report S3, 4 S1 enables S2 S3

(b) The complete matching process of input stream `xyz` using the activity-based processing scheme. CW stands for cold worklist; BS stands for bitset.

Figure 4.7: Illustrating the activity-based processing

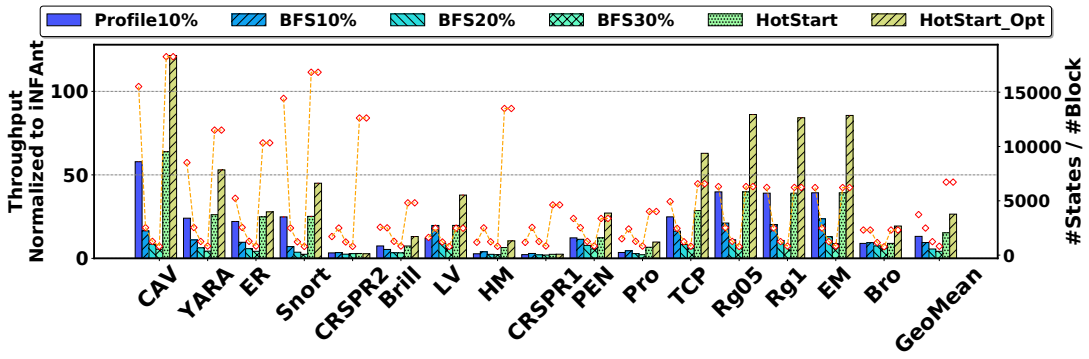


Figure 4.8: Throughput sensitivity to the selection of hot states. Detailed evaluation methodology is in Section 4.6. HotStart (or HotStart_Opt, an optimized version) has the best performance among these selection schemes. Hence, we choose the always-active start states as hot states.

next cold worklist is reset to 0 (Figure 4.7a ⓒ).

The processing of input symbol `y` then begins (Figure 4.7a ❷), with hot-to-cold transitions S1 to S2 and S1 to S3 in the hot mode (Figure 4.7a ⓓ). They are pushed to the cold worklist. The *deduplication bitset* is also set. In the cold mode, two threads process the S2 and S3 that was pushed to the CW in the previous step when we processed `x`. Since S2 matches with `y`, it generates two cold-to-cold transitions (Figure 4.7a ⓔ). However, by checking the worklist deduplication bitset (Figure 4.7a ⓕ), S2 and S3 are not pushed to the worklist. After the cold mode, the threads in the thread block are synchronized and the next step of the execution begins with states S0, S1, S3 and S4 as active. Figure 4.7b summarizes the complete steps to process the `xyz` input stream.

4.5.3 How do we choose the hot states?

In this section, we describe three different ways to classify states as hot or cold and pick the method that performs best empirically.

Profiling. The first scheme is based on prior work [66], which shows that the activation frequency of NFA states in a small representative input is similar to the entire input. In this evaluation, we use the 1KB prefix of the 1MB input as the profiling input. If a state

has an activation frequency more than a threshold in the profiling input, we consider it as a hot state during the entire execution. In the experimental results, we use 10% as the threshold. We have also tested other thresholds but they do not affect our conclusion.

Offloading by BFS layers. Second, we consider a percentage of states (ordered by their BFS layers) as hot states. Users can control the percentage of states to be deemed as hot. The assignment of hot states is an iterative process. We sort the states by their BFS-layers and then mark them as hot states in the ascending order until the number of hot states reaches the percentage specified by the user. This scheme relies on the topology of the NFA and does not need any profiling information. In this experiment, we use 10%, 20%, and 30% as the percentage of the hot states.

Make start states hot—HotStart. Our third scheme only considers the *always-active* start states as hot states. It does not require profiling or tuning of parameters/thresholds. The scheme is based on the observation from Figure 4.6 that other than the always-active starting states, the activation frequency for most other states is very low.

Experimental decision. Figure 4.8 shows the normalized throughput and utilization numbers of the aforementioned three schemes. We make the following observations. First, we find HotStart gives the best performance among the evaluated configurations on average. Second, we find that the performance is correlated to the states per block, which works as a proxy to show the utilization of GPU. As HotStart has the most states per block, its utilization is the best among these cases. Third, for a few applications (e.g. CRSPR1, CRSPR2), HotStart does not give the best performance, but still gives comparable performance. This is because the activation frequency of non-starting states is high (Figure 4.6). Therefore, the loading of the node data structure to the worklist leads to more data movement. To summarize, HotStart gives us a simple and synergistic solution for both data movement and utilization, while achieving the best performance across the evaluated hot states selection schemes.

Elimination of Active Bitset. In HotStart, every activated state is in the worklist except the *always-active* starting states. Hence, we remove the *active bitset* from the

thread block. By this simple optimization specific to HotStart, the register usage is reduced from 70 registers to 40 registers per thread, leading to an increase of occupancy. Figure 4.8 shows that this optimized version of HotStart (HotStart_Opt) gives 77% improvement and we use it in the rest of the experiments.

4.6 Evaluation Methodology

Evaluated Schemes. Table 4.1 summarizes the schemes that we evaluate in this paper. iNFAnt [36] and NFA-CG [154] are prior works in the area of NFA Processing in GPUs as discussed in Section 4.3.3. iNFAnt [36] maps each edge to a thread and NFA-CG [154] maps a compatible group (a group of states) to a thread. Next, we evaluate our schemes NT (Section 4.4.2), NT-MaC (Section 4.4.3), which dedicate each thread to each state. They only focus on reducing data movement. Then we evaluate HotStart and HotStart-MaC (Section 4.5), which are built using HotStart_Opt and work on top of NT and NT-MaC, respectively. They enhance utilization by mapping only *always-active start states* to the threads (Section 4.5.3) without the need of profiling. To demonstrate the effectiveness of data movement optimization, we also evaluate HotStartTT, which uses an alphabet-oriented transition table but also applies our utilization-related optimizations.

Table 4.1: Overview of the evaluated schemes on GPU

Scheme	Thread Mapping	Data Movement	Utilization
iNFAnt [36]	Edge	-	-
NFA-CG [154]	Compatible Group	-	Compatible Group
NT	State	Opt. I	-
NT-MaC	State	Opt. I + Opt. II	-
HotStart	Hot State	Opt. I	Opt. III
HotStart-MaC	Hot State	Opt. I + Opt. II	Opt. III
HotStartTT	Hot State	-	Opt. III

AP Performance Modeling. We also compare to the automata processor (AP) [44], a domain-specific architecture for NFA processing. Since AP is not publicly available, we use a simple, optimistic performance model influenced by VASim [124] for AP performance estimation. If we have n input streams each containing m symbols, each NFA must

process $n \times m$ symbols. If AP can hold C NFA states, and if the application has A NFA states, then $\lceil A/C \rceil$ batches are required. In one AP chip, C equals to 49152. As current AP chip is documented to run at 133MHz, a symbol needs 7.5 ns for processing. Thus, $Time_{AP\_ideal} = 7.5 \cdot mn \cdot \lceil A/C \rceil$. Since APs must be reconfigured between batches, we add per batch reconfiguration overhead of 50 ms [126] to $Time_{AP\_ideal}$ to obtain $Time_{AP}$. Both these models are *optimistic* because we ignore other overheads of AP, such as the time taken to record reports (i.e., matches). Earlier work has noted that this can be a significant overhead in several applications [122]. Note that all the performance numbers for GPUs *include* the report generation overhead.

Experimental Setup. We mainly use an NVIDIA Quadro P6000 GPU for evaluation. We also report the sensitivity of our results on NVIDIA Tesla V100. We report the GPU kernel time gathered using CUDA events. The throughput (our metric for performance) is measured in terms of number of input symbols processed per second. Each set of experiments is performed 7 times and we report 95% confidence intervals for our results (shown as error bars). Our results/conclusions are consistent across different runs. Usually, these error bars are too small to visualize since the widest CI bar is 0.36% of the normalized throughput bar. For a fair comparison with prior works [36, 154], our results do not include the I/O time and data structure preparation time as prior works do not focus on optimizing them. We expect that these overheads will be amortized over long GPU computation time and hence we focus on optimizing the latter.

Application Configurations. We evaluate 18 applications from three different benchmark suites: AutomataZoo [125], Regex [31] and ANMLZoo [123]. Table 4.2 shows the characteristics of the evaluated applications. All these applications have sufficient parallelism in terms of number of states per NFA, number of NFAs (also called as connected components [CC]). The total number of states can be in the order of millions (Table 4.2). We use 1MB input (split into 1000 1KB input streams) for each application (except APPRNG and SeqMat) to provide parallelism for input streams. Two applications (APPRNG and SeqMat) do not have *always-active* start states, so we cannot feed the 1KB chunks of

Table 4.2: Characteristics of evaluated NFA applications.

Application		State Info					Connected Component (CC) Info		
Name	Abbr.	#states	#start	#always-active	#reporting	#compressible	#CC	max_CC_size	avg_CC_size
Brill [125]	Brill	115549	5.1%	5.1%	5.1%	100.0%	5946	40	19.4
ClamAV [125]	CAV	2374717	1.4%	1.4%	1.4%	100.0%	33171	22075	71.6
CRISPR_CasOFFinder [125]	CRSPR1	74000	5.4%	5.4%	2.7%	0.0%	2000	37	37.0
CRISPR_CasOT [125]	CRSPR2	202000	2.0%	2.0%	1.0%	0.0%	2000	101	101.0
APPRNG_4sided [125]	APPRNG1	20000	5.0%	0.0%	20.0%	20.0%	1000	20	20.0
EntityResolution [125]	ER	413352	2.4%	2.4%	2.4%	66.7%	10000	75	41.3
Hamming_l18d3 [125]	HM	108000	1.9%	1.9%	1.9%	100.0%	1000	108	108.0
Levenshtein_l19d3 [125]	LV	109000	3.7%	3.7%	3.7%	100.0%	1000	109	109.0
Protomata [125]	Pro	24103	5.4%	5.4%	5.5%	46.6%	1309	123	18.4
SeqMatch_w6p6 [125]	SeqMat	51570	20.0%	0.0%	3.3%	100.0%	1719	30	30.0
Snort [125]	Snort	202043	1.6%	1.2%	1.6%	51.4%	2486	4509	81.3
YARA [125]	YARA	1047528	2.2%	2.2%	2.3%	98.0%	23530	1017	44.5
Bro217 [31]	Bro	2312	8.1%	8.1%	8.1%	44.6%	187	84	12.4
ExactMath [31]	EM	12439	2.4%	2.4%	2.4%	100.0%	297	87	41.9
Ranges05 [31]	Rg05	12621	2.4%	2.4%	2.4%	99.0%	299	94	42.2
Ranges1 [31]	Rg1	12464	2.4%	2.4%	2.4%	98.3%	297	96	42.0
TCP [31]	TCP	19704	3.8%	3.8%	3.9%	94.0%	738	391	26.7
PowerEN [123]	PEN	40513	7.1%	7.1%	8.5%	99.7%	2857	52	14.2

input to them. Hence, we evaluate them separately in Section 4.7.

Large NFAs are filtered out for NT. Most applications only have small NFAs (connected components). However, ClamAV, Snort, YARA, and TCP have a few NFAs (up to 1.5% of their total number of NFAs) that have more than 256 states. Since our NT proposal does not support NFA size greater than thread block size, and NFA-CG could not finish calculating compatible groups for these NFAs, we filter out the NFAs that have more than 256 states to ensure a fair comparison. *This does not apply to our* HotStart *and* HotStart-MaC *which, like* iNFAnt*, support any size NFAs.*

Out-degree is limited to 4. Different states can have different out degrees leading to load imbalance in amount of work per thread. Like NFA-CG [154], we modify the NFAs so that the outgoing edges of each state is 4 or less using an iterative algorithm. We split each state that has an out-degree greater than 4 into two with each getting half of the original edges, and connect the duplicates to the predecessors and successors to maintain the semantics of the NFA. This process repeats until all states have 4 or fewer out edges. For some graphs, this process does not terminate (e.g., a complete graph where each node has out-degree of 5). If this process does not terminate in N steps (where N is the number of states), the NFA is filtered out. There are 2 NFAs in ClamAV and 5 NFAs in Snort that cannot be limited to states with out-degree ≤ 4 and hence are discarded.

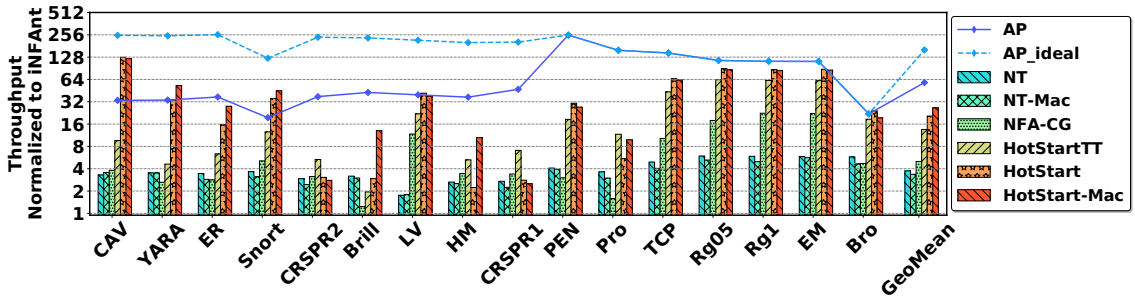


Figure 4.9: Throughput enhancement results normalized to iNFAnt. On average HotStart-MaC achieves 26.5× speedup across 16 applications. The best GPU results outperform an AP chip in 5 applications (CAV, YARA, Snort, LV, and Bro).

Table 4.3: Absolute throughput with our schemes (MB/s). The best performance among GPU schemes is highlighted.

Arch	config	CAV	YARA	ER	Snort	CRSPR2	Brill	LV	HM	CRSPR1	PEN	Pro	TCP	Rg05	Rg1	EM	Bro
AP	AP_ideal	2.72	6.06	14.81	26.67	26.67	44.44	44.44	44.44	66.67	133.33	133.33	133.33	133.33	133.33	133.33	133.33
	AP	0.36	0.82	2.14	4.21	4.21	8.16	8.16	8.16	15.38	133.33	133.33	133.33	133.33	133.33	133.33	133.33
GPU	iNFAnt [36]	0.01	0.02	0.06	0.22	0.11	0.19	0.21	0.22	0.33	0.53	0.84	0.92	1.16	1.19	1.19	6.07
	NT	0.04	0.09	0.20	0.79	0.33	0.61	0.36	0.58	0.88	2.15	3.06	4.51	6.86	6.97	6.96	35.01
	NT-Mac	0.04	0.09	0.17	0.67	0.27	0.57	0.37	0.56	0.73	2.07	2.53	3.71	6.05	5.90	6.72	28.16
	NFA-CG [154]	0.04	0.06	0.16	1.10	0.35	0.24	2.42	0.76	1.11	1.58	1.34	9.36	20.70	26.57	26.53	28.57
	HotStartTT	0.10	0.11	0.36	2.71	**0.60**	0.37	4.53	1.16	**2.31**	9.73	**9.84**	39.83	73.41	73.55	73.32	112.94
	HotStart	**1.38**	0.82	0.90	7.63	0.34	0.56	**8.59**	0.49	0.92	**16.06**	4.61	**59.54**	**103.58**	**102.84**	**104.47**	**145.88**
	HotStart-Mac	1.31	**1.29**	**1.60**	**9.70**	0.31	**2.50**	7.80	**2.32**	0.82	14.26	8.26	57.66	99.54	99.85	101.95	117.58

4.7 Experimental Results

Figure 4.9 shows the performance results of our schemes and the performance achieved by AP normalized to iNFAnt. The y-axis is in log scale. Table 4.3 shows the raw throughput achieved by our evaluated schemes. On average, HotStart-MaC gives 26.5× speedup and HotStart gives 20.5× speedup, which achieves the best and the second-best performance among all schemes. They also achieve the best performance on all applications except CRSPR1, CRSPR2 and Pro, where HotStartTT is the fastest. Furthermore, HotStart-MaC and HotStart are 5.3x and 4.7x faster than NFA-CG, respectively.

Analysis of cases where HotStart-MaC is the best. HotStart-MaC performs the best in YARA, ER, Snort, Brill, and HM among all GPU schemes (Table 4.3). Specifically, in these applications, HotStart-MaC achieves significant speedup — at least 27% improvement in Snort and up to 373% improvement in HM compared to HotStart. There are two reasons. First, in these five applications, at least 50% of their states are

compressible. Second, all of them have a large number of states. Their per-node data structures are too large to fit into the L1 cache of the GPU, especially when many states are handled through the worklist, reducing the size of per-node data structures yields significant improvement.

Analysis of cases where HotStart is the best. On the other hand, HotStart-MaC does not outperform HotStart for the rest of the applications, although the gap is consistently within 10%. This is expected since if an application (e.g., CRSPR2) has no compressible states HotStart-MaC will have more overhead than HotStart. However, even for an application that has many compressible states, HotStart-MaC can perform worse than HotStart, because the overhead of converting memory accesses to computation may outweigh the benefit. For example, in CAV, although all its states are compressible, HotStart-MaC still has a 5% slowdown than HotStart. One reason is the matching process in CAV only goes to very shallow parts of the NFAs, where only very a small set of states are used frequently. In this case, very few states are swapped in and out from the worklist, and hence the benefit of matchset compression is small.

The performance impact of increasing utilization alone. Even though HotStartTT does not use NT for data movement optimization, it is 14x faster than iNFAnt. Compared to NFA-CG that also only has utilization optimization (based on statically computed compatible groups), our HotStartTT is 2.7x faster since compatible groups do not capture the notion of activity. HotStartTT is also the fastest scheme in CRSPR1, CRSPR2, and Pro, since at most 46% of states in these applications are compressible. In addition, as their states are activated frequently (Figure 4.6), the swap-in and swap-out in the worklist of HotStart and HotStart-MaC incur more data movement than HotStartTT.

The performance impact of data movement optimization. NT and NT-MaC are 3.7x and 3.4x faster than iNFAnt respectively, however they are at least 26% worse than NFA-CG because optimizing data movement only without considering core utilization is not sufficient. Although our matchset compression optimization works well in

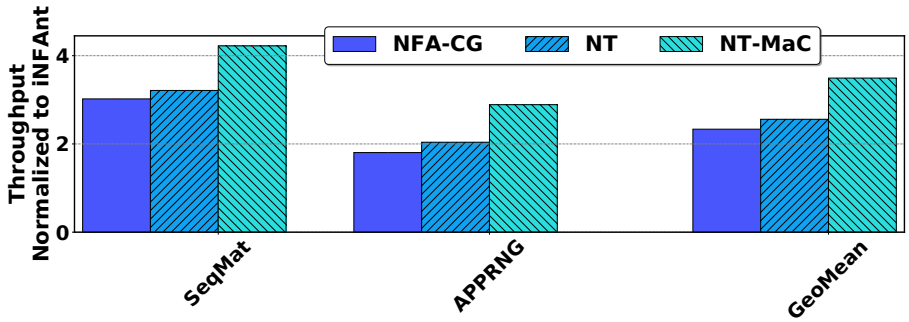


Figure 4.10: Throughput enhancement for the applications without *always-active* start states in the single input stream scenario. Our schemes outperform NFA-CG and iNFAnt by at least 9% and 2.6×, respectively.

HotStart-MaC, it demonstrates performance degradation when applied with pure NT. This is because matchset compression converts memory accesses to the range checking computation. As a result, the NT-MaC GPU kernel uses more registers than NT. If the per-node data structures fit into L1, the loss of register resources potentially affects performance.

Evaluation of SeqMat and APPRNG. Two applications (SeqMat, and APPRNG) do not have *always-active* start states. Therefore, we do not evaluate them using HotStart in the multiple input streams scenario. Instead, we use a 1MB input stream to evaluate these applications. Figure 4.10 shows the performance of these applications. We compare our data movement optimization schemes NT and NT-MaC with iNFAnt and NFA-CG. In these two applications, we found that on average, NT and NT-MaC have 2.6x and 3.5x speedup over iNFAnt respectively. Our NT and NT-MaC schemes also show 9% and 50% improvement over NFA-CG.

How far are we from AP? In CAV, YARA, Snort, LV, our best GPU scheme outperforms the domain-specific AP chip by 3.8×, 1.6×, 2.3×, 1.1× (Table 4.3), because these applications have a large number of states requiring repeated re-configurations and re-executions on AP. Bro also has 1.1× speedup than AP. All other applications except Pro also perform within 10× of the AP performance as they make good use of the GPU resources, where Pro performs 12.54× worse than AP.

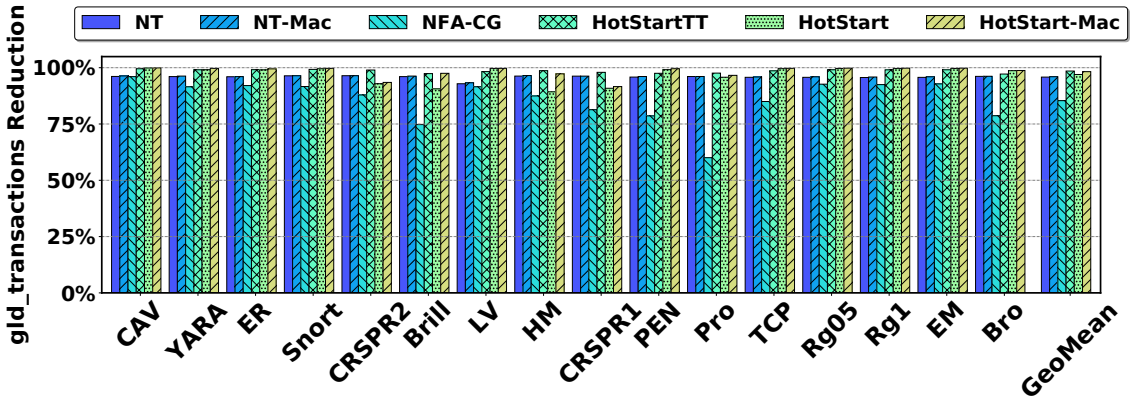


Figure 4.11: Effect on data movement reduction: our schemes use significantly fewer gld_transactions than prior work. For example, HotStart-MaC reduces gld_transactions by 99.3% over iNFAnt.

Effect on Data Movement Figure 4.11 shows the percentage of reduced global load transactions compared to iNFAnt. We observe that HotStart and HotStart-MaC use 98.9% and 99.3% fewer `gld_transactions` respectively than iNFAnt, because they optimize for both data movement and utilization. Although HotStartTT does not optimize for data movement, it uses 98.7% fewer `gld_transactions` than iNFAnt. This is because the utilization optimization reduces the number of thread blocks that access the transition table and the input streams. Similarly, NFA-CG uses 88.2% fewer `gld_transactions` than iNFAnt. With only data movement optimizations, NT and NT-MaC use 95.9% and 96.1% fewer `gld_transactions` than iNFAnt respectively.

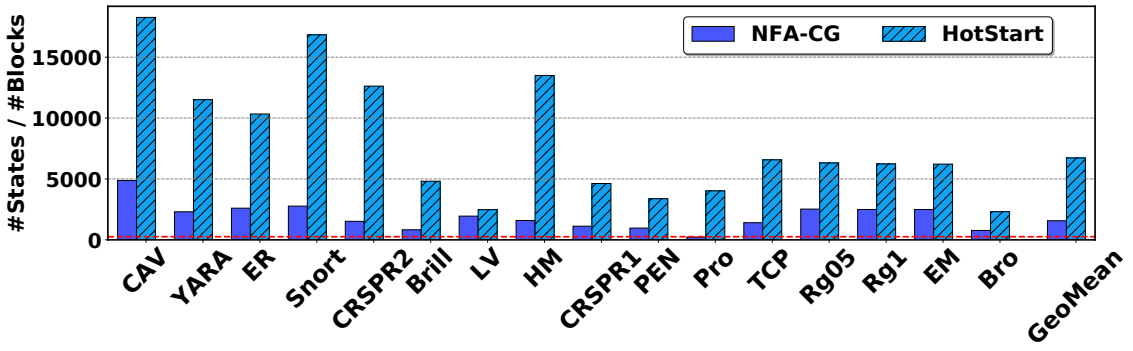


Figure 4.12: Effect on the number of NFA states per thread block (a proxy for compute utilization). More states are handled per thread block in HotStart.

Effect on Utilization. We use the number of NFA states per thread block as a metric for evaluating utilization. Figure 4.12 shows this metric for the evaluated schemes. As we do not change the amount of work per thread, mapping more states per block implies better utilization.

We limit our comparison of utilization to NFA-CG and our HotStart/HotStart-MAC/HotStartTT, because NT and NT-MAC do not focus on utilization and always map one state to a thread. Additionally, in iNFAnt, increasing the states mapped to a thread block does not increase the *useful* work per thread, so we do not include it in our comparison.

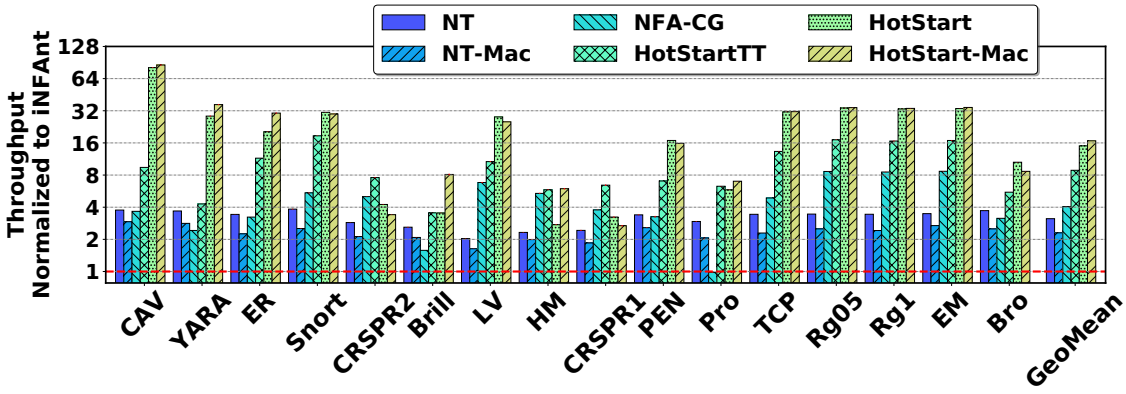


Figure 4.13: Performance sensitivity to Volta GPU Architecture. Both HotStart-MAC and HotStart show more than 15× speedup over iNFAnt, indicating their effectiveness on newer GPU architectures.

We found for most of the applications, HotStart achieves better utilization than NFA-CG, because only the always-active start states are mapped to threads, which means they are always doing useful work. In particular, NFA-CG fails to improve utilization in Pro, because each statically constructed compatible group only has one state in it, meaning any pair of states can be activated at the same time due to the NFA topology and matchsets of Pro. In contrast, by leveraging our insight into activation frequency (Figure 4.6) our HotStart can improve utilization even for Pro.

Sensitivity to Volta Architecture. We also evaluated our mechanisms on NVIDIA V100 GPU [9] and the results are shown in Figure 4.13. We observe a similar trend as what

has been shown in Figure 4.9—our schemes still give significant speedup. Specifically, on average HotStart-MaC and HotStart give 16.7× and 15.0× speedup over iNFAnt, respectively. HotStartTT gives 8.9× speedup over iNFAnt. Given that the Volta has larger L1 caches compared to Pascal GPU, the magnitude of speedup we achieve is lower but still very significant indicating that our data movement and utilization optimizations are effective on newer architectures as well.

4.8 Related Work

There is a large body of work on pattern matching on CPUs [131], network processors [32], and custom accelerators coupled to CPUs [53]. We recommend interested readers to an excellent survey paper [140] for a broad overview of the field. Here, we restrict ourselves to GPU/SIMD implementations of pattern matching using finite automata.

Reducing Data Movement. DFA-using engines [118, 116, 18, 128, 146] try first to reduce the size of the state transition tables using compression [146, 27, 143], which is often necessary to fit the DFAs in GPU global memory. However, as DFAs are serial, they are mapped to a single thread whose on-chip resources cannot accommodate the footprint of individual DFAs. Alphabet reduction [62, 130] reduces the size of the symbol set by merging behaviorally-equivalent symbols and introducing an indirection table, however, it is ineffective on large DFAs [146]. Our matchset compression technique is orthogonal to alphabet reduction and exploits the sparsity patterns in the matchsets.

To reduce cost of memory accesses, input symbols were loaded to shared memory [128, 18], input data layout was changed [128] to avoid uncoalesced accesses, or vector loads were used [118]. Others used k-stride NFAs [24], which consume k bytes at a time, but can blow up the alphabet to $|\Sigma|^k$. Some implementations place state transition tables in texture memory [118] or on-chip constant memory [18]. Prior work [104] explored packet signature matching on GPU using DFA and XFA [103] and though that work suggested profiling hot XFA states and storing them in on-chip shared memory, it did not implement

it as the size of on-chip shared memory was too small. Both textures and on-chip constant memory are limited in size, so large parts of the state transition tables remain in and are accessed using global memory.

If NFA topology is fixed, memory use can be reduced by embedding the topology of the NFA in the code [77] or inferring it from pattern-specific data [111]. The matchsets continue to be stored in global memory allowing different matchsets to be loaded at runtime. However, this technique is most suited for fixed-topology NFAs.

Improving Utilization. NFAs are compact in size, but their non-deterministic parallel execution requires that each thread handle a single transition [36] or a single state (this work). To improve utilization, more states must be mapped to a single thread. In previous works [154, 146], clusters of states called *compatible groups* are created that contain states that cannot be active at the same time [154] or that are likely to be active at the same time [146]. Compatible groups are mapped to the same thread [154] (for improving utilization) or are used to limit the lookups needed [146] (for decreasing work). The compatible groups of Zu et al. [154] improve utilization but are computationally expensive to compute. In contrast, our schemes construct a subset of NFA nodes that are mapped to threads at linear cost while achieving greater utilization than their compatible groups. Compared to Yu et al. [146], we separate topology and symbol/matchset information to limit the lookups without having to compute their notion of compatible groups.

4.9 Conclusions

In this paper, we proposed and evaluated three optimizations to significantly improve the throughput of NFA Processing on GPUs. These optimizations address sub-optimal data movement and low utilization, which stem from primarily two aspects: a) the matchset and topology information is stored off-chip and accessed in an irregular fashion, b) not all the NFA states are active all the time but still consume resources leading to GPU under-utilization. Our first two optimizations focus on the data movement problem and allow

the needed matchset and topology information to remain on-chip as much as possible. Our third optimization identifies and maps only active states to dedicated threads while less active states are processed on-demand using a worklist-based approach. Overall, we achieve significant improvement in NFA processing throughput over the state-of-the-art mechanisms across a wide range of emerging applications. Moreover, our optimizations enable GPUs to outperform the domain-specific accelerator (AP) for several applications while being within an order of magnitude of AP performance for the remainder. As a part of our future work, we plan to close this remaining gap with the help of hardware/software co-design optimizations.

Chapter 5

Generalizing Automata Processing on GPUs by Leveraging Symbol-level Parallelism

Finite-state automata serve as computation kernels for many application domains such as pattern matching and data analytics. Existing approaches on GPUs exploit three levels of parallelism in automata processing tasks: 1) input stream-level, 2) automaton-level, and 3) state-level. Among these, only state-level parallelism is intrinsic to automata. When an automata processing task does not have enough parallelism, the computation resources of GPU can be underutilized.

We propose AsyncAP, a low-overhead approach that exploits an additional source of parallelism from input symbols. The matching processes mapped to GPU threads start at different locations of the input stream in parallel. This new mapping helps in searching for patterns in parallel *asynchronously* thereby improving the automata performance significantly. However, theoretically, AsyncAP has higher time complexity. To understand the amount of work needed in reality, our detailed characterization of synchronous automata execution and AsyncAP demonstrates that AsyncAP only incurs 5% more work on average across the evaluated applications.

Overall, the evaluation shows that our new approach achieves up to 40× speedup on average across 14 evaluated applications when the task does not have enough parallelism to utilize all the GPU cores. When the task has enough parallelism to utilize GPU cores, AsyncAP also achieves comparable performance (∼99.3%) to the state-of-the-art that processes automata on GPUs.

5.1 Introduction

Finite automata (or Finite State Machines, FSMs) have been working as computation kernels for many applications in different domains such as bioinfomatics [34], machine learning [112, 93, 99, 127], intrusion detection [89, 12], and textual data analytics [52, 82].

Processing automata efficiently is extremely challenging for traditional architectures due to irregular memory accesses and intrinsic data dependencies. The users, therefore, resort to domain-specific accelerators based on ASICs or FPGAs [44, 47, 66, 96, 98, 107, 87, 106, 150]. For example, Micron's Automata Processor [44, 126] repurposes DRAM to process automata and outperforms traditional architectures by orders of magnitude. However, these accelerators bring additional complexity to the computing systems [118], and are slow to configure [140, 126].

By contrast, GPUs are found in many computing systems from mobile phones to data center servers. They serve as general-purpose accelerators for performance-critical computation kernels as they provide massive data-level parallelism and high memory bandwidth. The computation power of GPU has scaled faster than CPUs in recent years [108]. Therefore, running automata processing tasks on GPU has attracted significant attention [36, 154, 69, 120, 146, 145].

Existing automata processing works on GPUs [36, 154, 146, 76, 69] have demonstrated that GPU achieves better performance than CPUs. Non-deterministic Finite Automata (NFAs) are favorable on GPU because they are compact in size [69]. Notably, GPU-NFA [69] that optimizes data movement and compute utilization for automata processing

Table 5.1: Three levels of parallelism in NFA processing

Parallelism Source	Example
Input Streams	Many network packets
NFAs	Many intrusion signatures
Active NFA States	Non-determinism intrinsic to NFAs

achieves comparable performance to Automata Processor on a single GPU for several applications.

An automata processing task is to find patterns defined by automata from the input streams. To parallelize the automata processing tasks on GPUs, as Table 5.1 shows, existing works leverage three levels of parallelism: (1) input stream level, (2) automaton-level, and (3) state-level. First, the user can have multiple input streams to find patterns. For example, in a network intrusion detection application, multiple network packets can be processed in parallel. Second, multiple NFAs can be processed in parallel. For example, since each NFA represents an interesting pattern, an application can have many patterns to find (e.g., signatures of network intrusions). Third, non-determinism means multiple states can be active at the same time in NFAs, so they can be processed in parallel.

Only the state-level parallelism is intrinsic to NFAs, while the other two depend on how large the task that the user runs [88]. Tasks may have different latency requirements. To meet the latency requirement, the user may distribute a large task into small tasks, and use more computation resources to achieve shorter latency [150]. However, if a small task does not have enough parallelism to utilize all GPU cores, the performance may be sub-optimal.

To scale a small task on more computation resources, other lines of works increase the parallelism by breaking the dependencies between adjacent input symbols, which enables executing input chunks in parallel. Ladner and Fischer [63] use parallel prefix sum to parallelize automata, but it leads to significantly more work. Speculation [149, 84, 148, 83] and enumeration [74] are two categories of approaches that increase the parallelism of automata on multi-core processors. These works are often evaluated using Deterministic

Finite Automata (DFAs) as each DFA only has one active state every step. However, using speculation or enumeration for NFAs is difficult: 1) It is challenging to speculate which *set of states* are active at every beginning of input chunks; 2) Applying enumeration on NFAs leads to more execution paths. Further, these works are not using GPU currently because merging results from many different execution paths requires communication across GPU threads, which is complex to implement.

To address these problems, we propose a generic approach that increases the parallelism of an automata processing task to make it fully utilize the GPU. The increased parallelism needs to incur low overhead and exhibit high performance for different task sizes. To this end, we analyze and enable Asynchronous Parallel Automata Processing, AsyncAP, a simple and low-overhead way to exploit an additional source of parallelism of automata processing on GPU. Like previous work [69], we focus on NFAs,[1] because they are compact in size, which is easier to fit into GPU memory. We leverage the fact that most evaluated applications search for patterns from *any* position in the input streams rather than only from the starting position. AsyncAP converts the input stream indexed by 0 to n to many input streams where the kth input stream starts from position k of the original input. Therefore, the matching processes mapped to different GPU threads can start from different locations in parallel.

Overall, our new mapping helps in searching for patterns in parallel thereby improving the automata performance significantly. However, theoretically, AsyncAP has a higher time complexity, because, in the worst case, an input stream with n symbols needs to be read $O(n^2)$ times. Nevertheless, this upper bound may not reflect the real cases for two reasons. First, due to mismatches (when no state is active in a thread), the amount of work in practice is application- and input-dependent. Second, the implementation on GPU may bring useless work, but the time complexity only shows the useful work upper bound. To the best of our knowledge, no prior work has analyzed the different sources of works in practice.

[1] We use "automata" in the paper to refer to Non-determistic Finite Automata (NFAs), if not specified.

To this end, we perform a systematic characterization on previous synchronous automata processing (e.g., GPU-NFA [69]) and AsyncAP to understand the amount of work in practice. Our characterization by emulation analyzes both *useful* work (i.e., number of matches between states and input symbols) and *useless* work caused by idle threads due to the synchronizations by thread block or SIMD execution. Our key finding is that while AsyncAP has worse time complexity than the synchronous execution of automata, the amount of useful work in AsyncAP is only 5% more than the traditional synchronous execution on average across the evaluated applications. We conclude that AsyncAP has the potential to work as an approach to increase the parallelism of automata tasks with low-overhead in real cases.

To show the effectiveness of AsyncAP, we evaluate AsyncAP in scenarios with different amounts of parallelism. Under the scenarios where enough parallelism is not present to utilize GPU cores, AsyncAP achieves 5× to 40× speedup on average across 14 evaluated applications, respectively. When the original parallelism of the automata task is enough to utilize GPU cores, AsyncAP achieves comparable performance as the state-of-the-art prior work showing that its implementation incurs low overhead. In one of the evaluated applications, AsyncAP can have extreme slowdown due to extreme long patterns. However, our study shows the extreme long patterns are not frequent. If they happen, we discuss simple ways to avoid or detect them to prevent slowdown.

In summary, this paper makes the following contributions:

- We analyze and find that the prior work cannot adapt to automata processing tasks with different amounts of parallelism.
- We propose AsyncAP, a simple and low-overhead way to execute automata on GPUs that exploits an additional source of parallelism.
- We perform detailed characterization of synchronous execution and AsyncAP to understand the amount of work in real cases. Different from its theoretical upper bound, AsyncAP only incurs marginal extra work on average across the evaluated

applications.

- Evaluation results demonstrate that AsyncAP achieves 5× to 40× speedup when the existing parallelism is not enough to utilize all GPU cores. In other cases, AsyncAP is comparable to the state-of-the-art automata processing works on GPU regardless of the parallelism levels.

5.2 Background

A finite automaton is a mathematical model of computation in which the computations are abstracted as finite number of *states* and *transitions*. Two representations of finite automaton are widely used: deterministic finite automaton (DFA) and non-deterministic finite automaton (NFA). Although DFAs are simpler in transitions as only one active state is allowed, DFA execution is embarrassingly serial, and DFAs can be exponentially larger than equivalent NFAs. In this work, we focus on NFA[2] because of its compact nature and support of more parallelism.

NFA An NFA can be represented as a directed graph, where nodes represent *states*, edges represent state *transitions*. Each state has a *matchset* that contains the symbols it can accept. An automaton has one or more *starting states* (Figure 5.1, shown in hexagons) and *reporting states* (Figure 5.1, shown in double circles).

Types of Starting States If an NFA searches for patterns that appear regardless of the starting position in the input stream, it has only *all-input* starting states [5] that are *active* at every symbol in the input stream. For example, a pattern `/apple/` searches "apple" in the text no matter from which position of the text it starts. An NFA equivalent to it contains only *all-input* starting states. On the contrary, an NFA equivalent to pattern

[2]We focus on Glushkov NFAs [51], which are $\epsilon-$free and the matchset is on the node instead of on the edge. Any NFA that accepts a non-empty string can be transformed into an equivalent Glushkov NFA.

`/^apple/` contains *start-of-input* starting states, as it requires "apple" appears only in the first position.

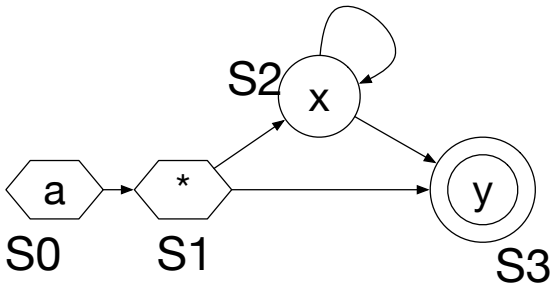


Figure 5.1: Illustrating an NFA that accepts `a*.x*y`. S_0 and S_1 are *all-input* starting states, which are always active in the execution.

How States are Matched Initially, only the starting states are *active*. The symbols of the input stream are fed into the NFA one by one. The *active* states match with the incoming symbol. If the incoming symbol falls into the matchset of an active state, the active state becomes a *matched* state. If a reporting state is matched, a report is generated showing an interesting pattern is identified. The *matched* states then *activates* their successors. This process continues until all the symbols of the input stream are consumed.

5.3 Asynchronous Parallel Automata Processing on GPUs

This section first introduces the motivation of proposing a new approach to process automata on GPU and then describes the implementation details.

5.3.1 Why do we need a new way to process Automata on GPUs?

In order to understand the need of AsyncAP, we revisit the existing works of automata processing on CPUs, GPUs and accelerators. Table 5.2 categorizes the existing works for automata (both DFA and NFA) processing into three categories based on input stream accessibility, number of NFAs, and input streams. We make the following observations.

Table 5.2: Categorization of Prior Works

Type	#Streams	#Automata	HW Utilization	Challenges	Examples
Buffered	Many	Many	Oversubscribed	Throughput	GPU-NFA [69] NFA-CG [154]
Buffered	Single	Few	Underutilized	Scalability	Speculation [55, 84, 149] /Enumeration [74]
Streaming	Single/Few	Few/Many	Underutilized	Latency	Multi-stride [29] Graph Transformation [147] HW Accelerators [66, 96]

Throughput-focused works are not scalable The first category (the first row in the table) considers the *high parallelism case* when the task requires more resources than the hardware can provide. Therefore, the main focus of this type of work is to find the bottleneck of automata processing in oversubscribed hardware. Typical optimizations include data movement and hardware utilization optimizations. For example, NFA-CG [154] calculates compatible groups of NFA states to enhance the thread utilization of GPU. GPU-NFA [69] proposed new data structures to reduce the data movement, and map hot and cold states differently to increase the GPU thread utilization. However, these approaches only use the levels of parallelism from the automata processing tasks. When the task does not have enough parallelism, these approaches cannot utilize all GPU cores.

Scalability-focused works have higher overhead The second category (the second row in the table) focuses on the scalability issue in a *low parallelism* case of automata processing. Since dependencies exist across input symbols, when the task does not have enough parallelism, the compute units of the hardware are underutilized. Thus, the existing works [84, 148, 149, 74] add additional parallelism by chunking the input stream and making each chunk of the input stream run in parallel. To keep the correctness, they must handle the dependencies correctly. To this end, they either speculate which states are active or enumerate all execution paths starting from every state. However, such approaches are difficult to adapt for NFAs, because speculating or enumerating on *state combinations* of NFAs are needed leading to more overhead.

Streaming-focused works only optimize for latency The third category (streaming, the third row of the table) considers the case that the input stream does not support random access. They optimize automata processing only for the per-symbol latency. The domain-specific accelerators for automata processing often fall into this category [66, 96, 97, 98, 44, 48, 107, 106, 94, 95, 142]. Most of them use in-memory processing to reduce the latency caused by data movement through the memory hierarchy. Other works such as constructing multi-stride automata [35, 24, 29] or automata compression [28] also reduce the per-symbol latency by graph transformation. These software optimizations are complementary to most of the other works.

Therefore, no single prior work is generic for all scenarios as of automata processing tasks with various parallelism. A scalable and high-performance scheme on GPU for all automata tasks is required.

5.3.2 Overview of Asynchronous Parallel Automata Processing

Figure 5.2 compares the traditional synchronous execution of automata processing with our proposed asynchronous automata processing on GPU. First, Figure 5.2 (ⓐ) shows the basic idea of synchronous execution. The NFAs states are mapped to the GPU threads (❶). The thread block reads the symbols from the input stream (❷). A thread block barrier (`__syncthreads()`) ensures that all states have finished the current symbol before moving to the next symbol. When the thread block reads a symbol, it is broadcast to the entire thread block and matches with the active NFA states. The successors of the matched states will be added to the *next active worklist* before processing the next symbol. Overall, the total number of threads (or thread blocks) depends on the number of NFAs and the number of input streams—the two numbers come from the automata processing task. When the total number of thread blocks cannot achieve its full theoretical occupancy [14], the task is not able to utilize all cores of GPU.

To address this issue, the basic idea AsyncAP is to increase the parallelism of automata on GPU. AsyncAP separates the execution paths *for each symbol of the input*

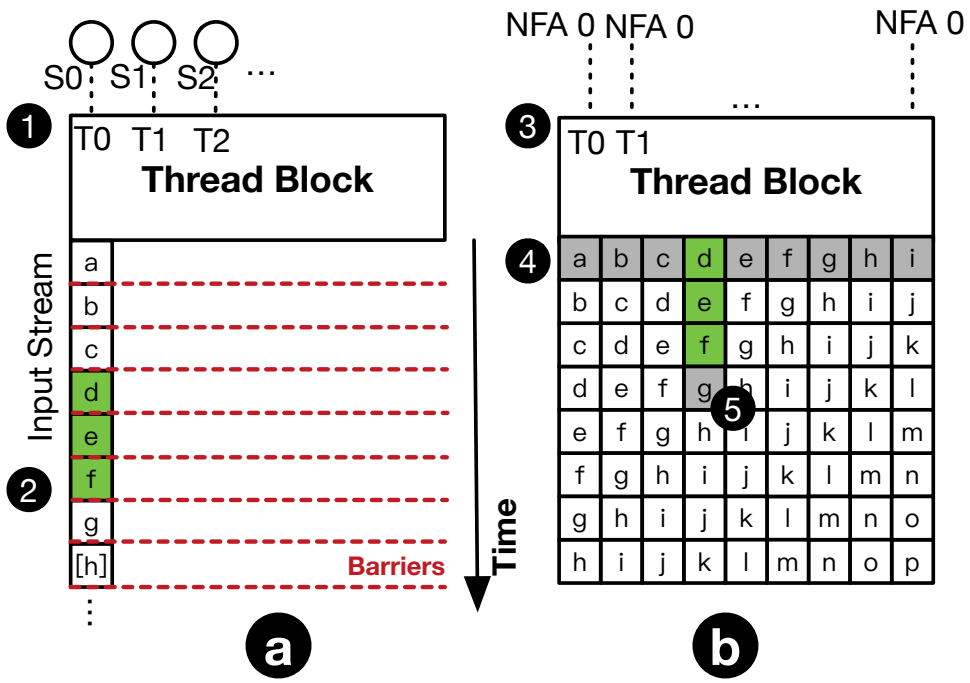


Figure 5.2: Revisiting traditional synchronous automata processing on GPU (a) and the basic idea of AsyncAP (b). The executions try to find pattern `def` in an input stream `abcdefg...`

stream to gain parallelism (Figure 5.2 (❻)), namely *symbol-level parallelism*. Therefore, all the starting states of the NFAs in this approach must be *all-input* starting states.

We map each symbol of the input stream to each thread. All threads process the same automaton (❸). Since we start from every position of the input stream, we disable the *all-input* starting states such that they are only active at the first symbol of their threads. This is the same as converting them to *start-of-input* starting states. We match them from every symbol of the input stream, making the results equivalent to their original semantics.

When the matching process begins, thread i reads the input stream independently from input position i to the end of the input stream (❹). We disable the *all-input* (always active) starting states. As a result, when there is no active state in a thread, the thread finishes (❺). For example, in Figure 5.2 (❻), thread 3 has active states from positions

3 and 5 of the input stream (`def`), but mismatches at position 6 (`g`), so the thread is terminated at position 6. Other threads do not match in the beginning, so they finish when matching the first symbols mapped to them. The work of each thread depends on when the pattern that it is matching with ends.

Applicability of AsyncAP We investigate all applications in two benchmark suites, ANMLZoo [123] and AutomataZoo [125]. Among 12 applications in ANMLZoo, all applications except `SPM` and `Fermi` contain NFAs with only *all-input* starting states. Among the 13 applications of AutomataZoo, all applications except `SeqMat` and `APPRNG` contain only NFAs with *all-input* starting states. AsyncAP applies to the NFAs that only have *all-input* starting states. Based on this observation, we conclude that AsyncAP is applicable for most of the applications in the existing benchmarks suites.

5.3.3 Design and Implementation

There are many points in the design space of implementing an automata processing scheme on GPU. We examine the major decisions below.

NFA Data Structures Two ways are commonly seen in the literature to store the topology of the NFAs. First, most prior works [154, 36] use an alphabet-oriented transition table to store the topology of the NFAs. It is a two-dimension table `T` where the rows are indexed by the alphabet, and the columns are indexed by the states. For example, `T['a'][S]` stores which states are matched when the incoming symbol is 'a' and `S` is the matched state currently. Second, other works [76, 69] use per-node data structures similar to Compressed Sparse Rows (CSR) that decouples the alphabet and the states. With matchset compression [69], such data structure reduces the data movement as it resides in GPU registers when possible. In the best case, the matching process does not require reading global memory to proceed.

We observe that using a transition table is *significantly more efficient* in AsyncAP

because there is not much reuse on the *all-input* states. The per-node data structure is larger because it stores matchsets separately. When they are not put into registers and have been reused frequently, more data movement is needed. We also choose to run each NFA in a kernel launch based on experiments (described later), so the transition table of each NFA is small to better utilize GPU caches. We conclude that the transition table works better in AsyncAP, and hence we use it in our scheme.

Per-thread Worklist Each thread in AsyncAP works asynchronously maintaining a private worklist. A simple way to do so is to use double-buffered arrays that are private to each thread. However, such private arrays are in the local memory. Although local memory can also utilize the cache hierarchy in recent generations of GPUs, the worklist arrays compete for the cache space with other data structures (e.g., transition table). This affects latency and degrades the performance significantly. We observe that for most of the symbols, the worklist size is small. Therefore, we implement hybrid worklists whose first 16 elements are stored on shared memory. When any of them overflows, it starts to write to local memory. Figure 5.4 shows that this optimization (`AsyncShrWL`) has a better median, 25th and 75th percentile performance for the evaluated applications.

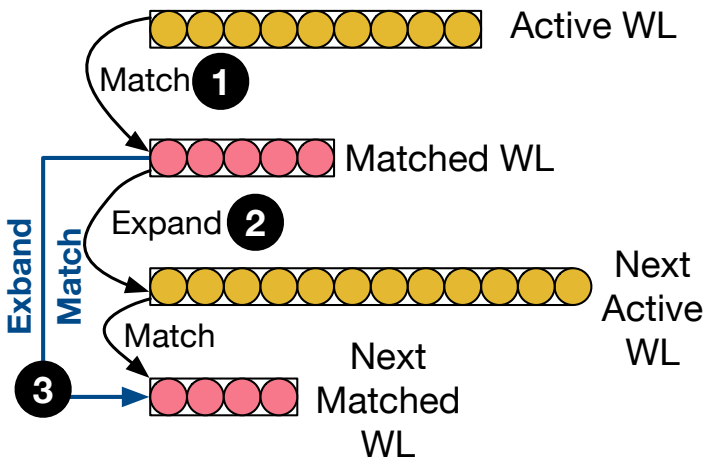


Figure 5.3: Worklist holds active states or matched states. WL stands for worklist.

What is in the Worklists? Whether worklists hold *active* states or *matched* states affects the matching process. Figure 5.3 illustrates them. If a worklist holds active states, when one of the active states matches the incoming symbol (❶), it activates its successors before the next symbol comes (❷). If a worklist holds *matched* states, it matches the next symbol with all its successors one by one and pushes the matched successors to the worklist in the next symbol (❸). GPU-NFA [69] holds *active* states in its worklists because it retains hot states to registers. When a match happens, this way does not need to access *matchset* of each state. However, we found that holding matched states performs *significantly better* in AsyncAP, because AsyncAP disables the *all-input* states, so these hot states are not reused across threads.

One-Kernel-One-Automaton or One-Kernel-Many-Automata AsyncAP exploits symbol-level parallelism, so even 1MB input stream can utilize all GPU threads in high-end GPUs. It is always sufficient for GPU regardless of the other levels of parallelism. As a result, we have the option of running only one automaton with a kernel launch. Essentially, this trades NFA-level parallelism for smaller transition tables (a few MBs), as the largest NFA we evaluated has ∼200 states (in CAV). Figure 5.4 shows that one automaton per kernel launch (`AsyncBase`) has a higher median throughput than a kernel that executes all NFAs in one launch (`AsyncMergeCC`).

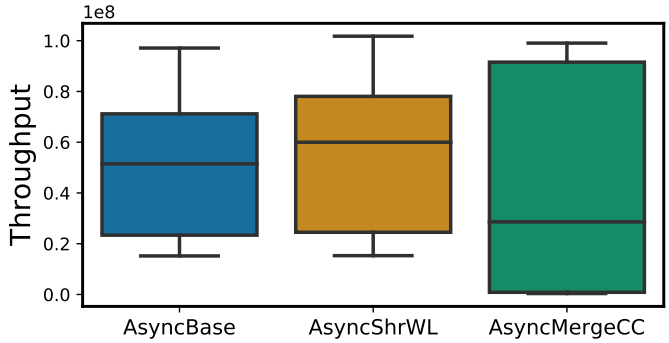


Figure 5.4: Performance of Selected Implementations of AsyncAP. Evaluation methodology is described in Section 5.5.

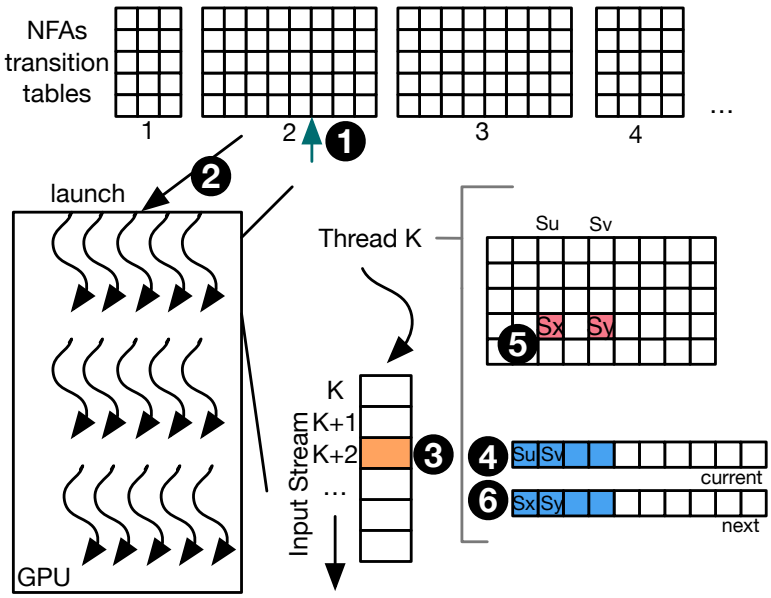


Figure 5.5: Illustrating our implementation of AsyncAP

Execution of AsyncAP Figure 5.5 illustrates the execution of AsyncAP. On the host side, NFAs are executed in a one NFA one kernel fashion (❶). Each NFA is launched on a different CUDA stream to allow concurrent kernel execution (❷). To zoom in a thread k, it starts execution from position k of the input stream. Suppose the current location is $k+2$ (❸), and the current worklist contains matched states S_u and S_v (❹). In the worklists, the shaded portion shows that the first few (4 in our implementation) elements are stored in shared memory, and the later elements are stored in local memory. Then, two cells of the transition table are accessed (❺), and the matched states are then pushed to the next worklist (❻). This finishes processing the current symbol at $k+2$. When the current worklist is empty, the execution finishes.

5.3.4 Analysis

Correctness We analyze why AsyncAP can generate the same reports as synchronous automata processing on GPU. The basic idea is to show synchronous execution and AsyncAP have the same set of active states at any position of the input stream.

Suppose the *all-input* starting state of the NFA is $s0$. It is always active. Let the set of active states for the NFA at position p as S_p (S_p must contain $s0$ for any p). $expand(S_p, input[p])$ is a function that takes S_p and $input[p]$ (input symbol at position p), and calculate what are the active states based on S_p and the incoming symbol. We omit $input[p]$ part because the value of p depends on the order that the function is called. For example, if *expand* is nested called k times, then $p = k$. In the synchronous execution, since all starting states are *all-input*, S_{p+1} is calculated by using S_p and the starting state s_0 to match with the incoming symbol input[p].

$$S_{p+1} = expand(S_p, input[p+1]) \cup \{s_0\} \tag{5.1}$$

$$= \underbrace{expand \ldots (expand(expand}_{p \text{ times}}(\{s_0\}) \cup \{s_0\})) \cup \{s_0\}) \cup \{s_0\} \tag{5.2}$$

$$= \underbrace{expand \ldots}_{p \text{ times}}(\{s_0\}) \cup \underbrace{expand \ldots}_{p-1 \text{ times}}(\{s_0\}) \cup \ldots expand(\{s_0\}) \cup (\{s_0\}) \tag{5.3}$$

Here, the first line shows the synchronous execution, and the last equation describes the process of AsyncAP. This proves that we will have the same active set of states in position $p + 1$ (i.e., S_{p+1}). Since the reports are generated when reporting states are matched, AsyncAP generates the same results as synchronous execution does.

Table 5.3: Comparison of Time Complexity. n: number of symbols; m number of states.

	Synchronous Execution	AsyncAP
Lower Bound	$\Omega(n)$	$\Omega(n)$
Upper Bound	$O(mn)$	$O(mn^2)$

Time Complexity We calculate the total time complexity of AsyncAP. In the best case, all the threads only read one symbol and then mismatch. The time complexity lower bound is $\Omega(n)$, where n is the number of symbols in the input stream. At the worst case, thread at position i ($0 \leq i < n$) needs to match with $n - i$ symbols. To sum up, they have read $O(n^2)$ symbols. At each position, at most, all the m states can be active. Therefore,

the time complexity is $O(mn^2)$, which is higher than synchronous execution (Table 5.3). However, in Section 5.4, we will show by characterizing actual applications that this worst case is unlikely in practice.

5.4 Characterization of Synchronous Automata Processing and AsyncAP

To show the potential of AsyncAP in practice, in this section, we characterize both synchronous execution and AsyncAP on GPU. First, we study how different the patterns are identified from synchronous execution and AsyncAP. Second, we characterize synchronous execution and AsyncAP to compare how they utilize the GPU from three aspects: 1) useful work, 2) useless work, and 3) work imbalance.

5.4.1 Applications Configurations

We show the applications evaluated in this work. Similar to prior work on GPU [69], we evaluate 15 applications from three benchmark suites, ANMLZoo [123], AutomataZoo [125], and RegEx [31]. To keep the NFA-level parallelism the same for all applications, we sample 256 NFAs for each of them uniformly at random. We set the random seed to 1234 to ensure the results are reproducible. We also change the random seed to other numbers and observe that the results are similar. Table 5.4 shows the basic information of the evaluated applications. Each application has one representative input stream collected by the benchmark suites. We use the first 1MB input stream to characterize those applications as suggested by prior work [69].

5.4.2 Comparison of Identified Patterns

First, we generalize the definition of *pattern*.

Table 5.4: Overview of Evaluated Applications

App.	Abbr.	Avg. NFA Size	Total States
Brill	Brill	19.55	5005
Bro217	Bro217	12.36	2312
CRISPR_CasOFFinder	CRISPR2	37.00	9472
CRISPR_CasOT	CRISPR1	101.00	25856
ClamAV	CAV	70.05	17932
EntityResolution	ER	41.62	10656
ExactMatch	EMatch	41.38	10594
Hamming_l18d3	HM	108.00	27648
PowerEN	PEN	14.78	3783
RandomForest	RF	31.00	7936
Ranges05	Rg05	42.36	10843
Ranges1	Rg1	41.75	10688
Snort	Snort	77.76	19906
TCP	TCP	24.73	6331
YARA	YARA	36.45	9331

Definition of a "Pattern" When a reporting state is activated, a pattern is identified. However, this definition of pattern does not characterize the case of mismatch, which is very common [69]. Even when an automaton cannot recognize a pattern from the input stream, the automaton might have matched with several contiguous symbols of the input stream. Therefore, we generalize the definition of a pattern to include the case of mismatches. We define a pattern as the contiguous symbols where the starting position is when the starting state is *matched*, and the ending position is when there is no active non-starting state. For example, if automaton `/apple/` tries to match with input stream `application`, `appl` is the pattern that can be identified starting from position 0.

We compare the patterns identified by synchronous execution and AsyncAP. Figure 5.6 shows an illustrative example of the patterns identified by synchronous execution (e.g., GPU-NFA) and AsyncAP. Suppose the NFA identifies `(a|p)+le` (❶). Figure 5.6 (❷) shows the synchronous execution. The pattern P1 in the input stream are shaded. On the contrary, Figure 5.6 (❸) shows the patterns identified by AsyncAP.

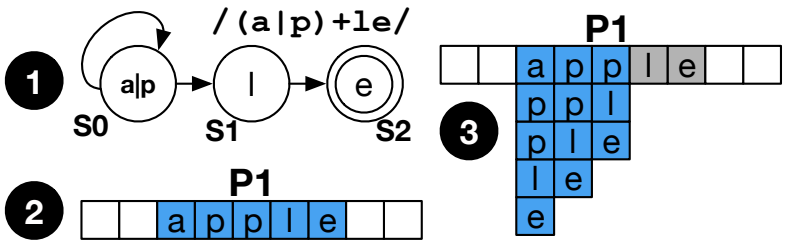


Figure 5.6: Synchronous execution identifies disjoint patterns; Patterns identified by different threads in asynchronous execution may overlap.

Pattern P1 is discovered by three execution paths (❶) and these execution paths stop at the same position (therefore, the reports are the same). In summary, synchronous execution only matches `apple` (5 symbols), but AsyncAP matches `apple`, `pple`, and `ple` (12 symbols in total). AsyncAP captures overlapped patterns ending in the same location, leading to more work.

How many overlapped patterns can be identified depends on both the automata and input streams Consider an extreme case: if we have an NFA that has only one state accepting `*`, and has a self-loop, all the patterns starting from different locations of the input stream are overlapped. In this case, AsyncAP reaches its theoretical upper bound of work. However, since NFAs in real applications identify meaningful patterns in real inputs, we will study what happens in this practical case.

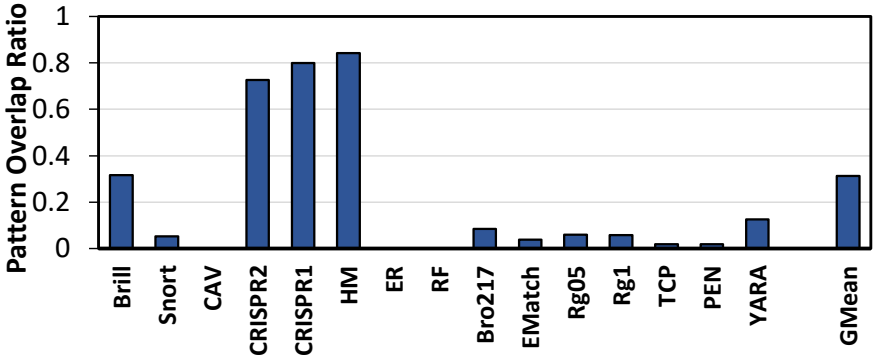


Figure 5.7: Ratio of Overlapped Patterns (R) in Evaluated Applications

Measuring the Overlapped Patterns To study how common the overlapped patterns are in the evaluated applications, we define a metric: $R = 1 - \frac{\text{length of disjoint pattern}}{\text{length of all pattern}}$. R is in the range of 0 to 1. The more the value is, the more portion of overlapped patterns are identified by AsyncAP.

Figure 5.7 shows the results about the ratio of overlapped patterns across the evaluated applications. We observe that on average, $R = 0.31$, showing that the sum of the lengths of overlapped patterns is 31% longer. Many applications have R closing to 0, indicating that no overlapped pattern happens in AsyncAP. However, a few applications (e.g., CRISPR1, CRISPR2, Hamming) have a large R, indicating they have a large portion of overlapped patterns.

5.4.3 Characterization of Work by Emulation

Table 5.5: Characteristics of applications based on our execution models: Our key observations: (1) The GPU utilization depends on applications. (2) Although AsyncAP has higher time complexity, in reality only 5% more useful work is needed on average. (3) Most applications balance the work across threads well, but rarely the work is severely imbalanced.

App.	Util. AsyncAP	Util. GPU-NFA	$\frac{\text{AsyncAP}}{\text{GPU-NFA}}$ Useful Work	$\frac{\text{AsyncAP}}{\text{GPU-NFA}}$ Total Work	Imb. Ratio
Brill	0.18	0.89	1.00	4.87	0.07
Snort	0.61	0.53	1.95	1.70	0.000176
ClamAV	0.90	0.67	1.00	0.74	0.09
CRISPR2	0.47	0.89	1.00	1.89	0.32
CRISPR1	0.50	0.96	1.00	1.91	0.31
HM	0.59	0.93	1.00	1.59	0.44
ER	0.27	0.85	1.00	3.18	0.10
RF	0.22	0.60	1.00	2.69	0.02
Bro217	0.64	0.76	1.00	1.17	0.03
EMatch	0.67	0.57	1.00	0.86	0.04
Rg05	0.68	0.57	1.00	0.84	0.02
Rg1	0.67	0.57	1.00	0.85	0.02
TCP	0.58	0.52	1.01	0.89	0.000358
PEN	0.29	0.52	1.04	1.91	0.000002
YARA	0.86	0.56	1.00	0.66	0.01

In this section, we compare the synchronous and asynchronous automata processing on GPU in terms of *work*—which is comprised of useful work and useless work.

Useful Work In every iteration of automata execution, the states match with the incoming symbol of the input stream, and decide what to do next (e.g., expand to their successors, generate reports). We define the number of matches as *useful* work: When an active state matches with the incoming symbol, we count it as a unit of useful work.

Useless Work By contrast, *useless* work is due to either thread block synchronization or warp synchronization. If a thread is waiting for other threads matching with an incoming symbol for synchronization, we count the activity of the idle thread as a unit of useless work.

Although one can easily collect GPU utilization results from existing profilers [15, 16, 17], simulators [26, 60], or emulators [46], these tools cannot differentiate the *useful* work and *useless* work specific to automata processing on GPU. Therefore, we emulate them by execution models based on their simplified implementation ideas on GPU.

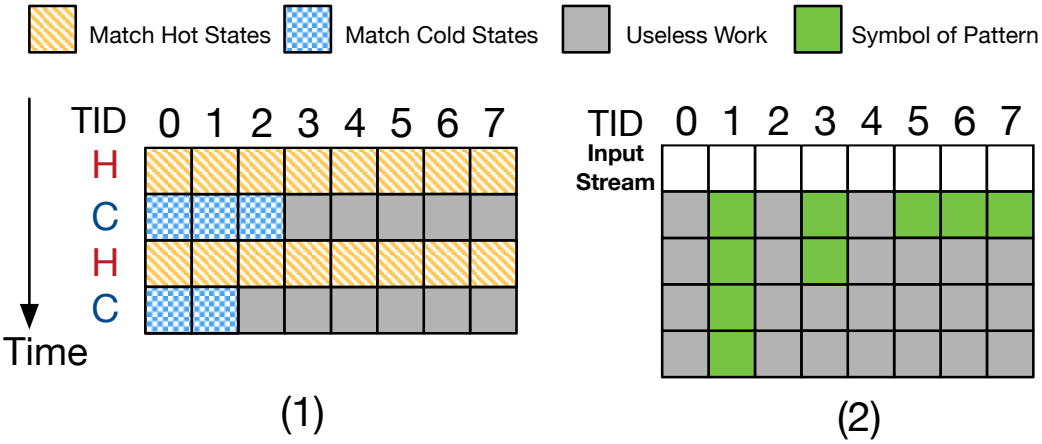


Figure 5.8: Illustrating the Execution Models of Synchronous (GPU-NFA) and Asynchronous (AsyncAP) Automata Processing on GPU

Figure 5.8 illustrates our proposed execution models of synchronous and asynchronous (AsyncAP) Automata Processing on GPU. We use GPU-NFA [69] as a representative

approach of synchronous automata processing on GPU. Each cell in the figure is either a unit of useful work or useless work.

Model of Synchronous Automata Processing Figure 5.8 (1) shows the execution of GPU-NFA. Each thread of GPU (`TID`) is mapped to a hot state. The execution contains two stages: hot stage and cold stage. The hot states mapped statically to threads match with the incoming symbol at the hot stage. Next, the threads process the elements in the worklist before the thread block synchronization. However, the worklist may not contain enough active states for all threads to process. As the implementation uses static scheduling, when several threads are matching with incoming symbols, other threads in the same thread block are idle, waiting for synchronization (grey cells). These idle threads are doing *useless* work in GPU-NFA. We use block size 256 in our emulation, which is also used in GPU execution because it can reach the highest GPU occupancy.

Model of AsyncAP Figure 5.8 (2) shows the execution model of AsyncAP. Each thread (TID) is mapped to an input position. The thread runs until no state is active. Thus, the threads run for a different number of symbols depending on the lengths of patterns in the threads. Due to the SIMD execution of GPU, the threads within a warp are synchronized implicitly. When a thread is processing a long pattern, other threads in the same warp are idle (shown in grey-shaded cells). These grey shaded cells are *useless* work in AsyncAP. We use 32 as the warp size, which is the same as NVIDIA GPUs [9].

We emulate the two execution models to compare three aspects of GPU-NFA and AsyncAP. First, to study the real cases compared to theoretical time complexity, we compare GPU-NFA and AsyncAP in terms of useful work (Section 5.4.4). Second, we study how much useless work is in their implementations (Section 5.4.5). Third, we study how the total work is balanced across GPU threads (Section 5.4.6).

5.4.4 Comparison of Useful Work

Table 5.5 shows the results based on our execution models.

How does each one utilize GPU? The first two columns show the utilization ratio of useful work in AsyncAP and GPU-NFA, respectively. The utilization ratio is calculated by $\frac{\text{Amount of useful work}}{\text{Amount of total work}}$. The larger the value is, the more utilized the GPU is. We observe that the ratio of useful work is 0.49 and 0.67 in AsyncAP and GPU-NFA on average (geometric mean) across the evaluated applications, respectively. The utilization ratio is also application-dependent. For example, for a number of applications (e.g., YARA, Rg1, CAV), AsyncAP has done a larger utilization ratio. We conclude that the various lengths of patterns may affect the utilization of AsyncAP.

Comparison of Useful Work The third column of Table 5.5 shows the useful work ratio comparing AsyncAP with GPU-NFA. It is calculated by $\frac{\text{Amount of useful work in AsyncAP}}{\text{Amount of useful work in GPU-NFA}}$. AsyncAP can have more useful work than GPU-NFA because it can identify overlapped patterns (shown in Section 5.4.2).

For all evaluated applications, the useful work is far from achieving the upper bound of AsyncAP's time complexity (shown in Table 5.3). On average (geometric mean), we observe that AsyncAP only requires 5% more useful work than GPU-NFA across the evaluated applications. For most of the applications (11 out of 15), AsyncAP only requires less than 0.1% of useful work compared with GPU-NFA. We observed that the only outlier is Snort, in which AsyncAP needs 95% more useful work. Different from the theoretical time complexity, the marginal extra useful work required by AsyncAP indicates asynchronous execution of automata is potentially efficient in practice.

5.4.5 Comparison of Total Work

The fourth column shows the total work ratio of AsyncAP and GPU-NFA calculated by $\frac{\text{Amount of total work in AsyncAP}}{\text{Amount of total work in GPU-NFA}}$. We observe the total work is highly application-dependent.

Compared to GPU-NFA, Brill and ER use multiple times of total work in AsyncAP (4.9× and 3.2× of GPU-NFA, respectively). On the other hand, for applications such as CAV and YARA, AsyncAP uses around 30% less total work than GPU-NFA. On average, AsyncAP requires 44% more total work. More total work is due to two reasons. First, AsyncAP requires more useful work because of overlapped patterns. Second, the various lengths of patterns and the implicit warp synchronization lead to useless work. We will further compare the total work with their actual performance in Section 5.5.2.

5.4.6 How balance the work is in AsyncAP?

GPU-NFA balances the works in a thread block level. However, how the work is balanced in AsyncAP depends on the pattern characteristics of applications.

Therefore, we measure how the work is distributed across threads in AsyncAP. We define *Imb. Ratio* $= \frac{\text{Sum of Work of Warps}}{\text{Sum of Max Work of Warps}}$. The smaller the value is, the more imbalance the work is in the warps. We observe that the thread imbalance situation is various across the applications. A few applications such as CRISPR1 and HM have very balanced work across threads. Most of the applications have *Imb. Ratio* greater than 0.01, indicating that ∼100 warps with less work can offset a warp with a lot of work. On the other hand, PEN has a dramatically small value (orders of magnitude smaller than the values of other applications), showing its threads are extremely imbalanced.

We do not apply software-based load balance to AsyncAP because it requires synchronization of threads, which causes more latency for every step. Nevertheless, the thread block scheduler can balance the work in a coarse-grain way by assigning a new block, and AsyncAP has a bigger pool of thread blocks as it increases the parallelism.

We conclude that most of the applications distribute the work across threads well, but in a rare case, the work is extremely imbalanced.

In summary, by the characterization of GPU-NFA and AsyncAP, we observe that AsyncAP needs a moderate amount of more useful work and total work compared with the synchronous execution, and has a fair work balance for most evaluated applications.

The observed characteristics of AsyncAP enable it to work as an alternative way for adapting GPU automata processing to different task sizes.

5.5 Evaluation

This section first describes our evaluation configurations. Next, we discuss the performance results of AsyncAP.

5.5.1 Evaluation Configurations

Evaluated Schemes We compare our schemes with two synchronous execution schemes proposed by prior work – GPU-NFA [69]: Sync_HS (HotStart [69]) and Sync_HC (NT-MaC-ACP [69]). The latter supports the flexible placement of hot and cold states to threads. All the programs are compiled with `nvcc` 11.0 with `O3`.

Hardware Platforms We primarily use an NVIDIA Quadro P6000 GPU for evaluation. We also perform a sensitivity study on an NVIDIA A100 GPU to show the effectiveness for other GPU architectures.

Performance Measurement We use throughput to measure the performance, where throughput $= \frac{\text{Number of symbols}}{\text{Time}}$. We do not consider the time of copying NFAs and input streams to GPU, but we have confirmed this only incurs negligible (less than 10%) overhead. Before launching kernels, we moved them to GPU. We measure the end-to-end time from launching the first kernel to the end of execution where the reported are copied back to the host. Each set of experiments is performed 3 times and we report 95% confidence intervals for our results (shown as error bars).

Evaluated Scenarios We evaluate our approach for automata processing tasks with different amounts of parallelism. Since NFAs have different runtime characteristics, to better control the parallelism, we vary the number of input streams. We study three

scenarios, low (1 input stream), medium (15 input streams), and high parallelism (240 input streams). The *low* parallelism only has one input stream, which is evaluated in most of the prior works that focus on automata parallelization (such as speculation/enumeration techniques). The *medium* parallelism scenario is a case when the parallelism is not enough to utilize all the computation resources. Since our evaluated GPU has 30 stream multiprocessors (SMs), we use half of the number of SMs as the number of input streams. Last, the evaluated GPU supports 240 thread blocks running on the stream multiprocessors (SMs), so we consider the scenario with 240 input streams as *high* parallelism as it requires at least 240 thread blocks.

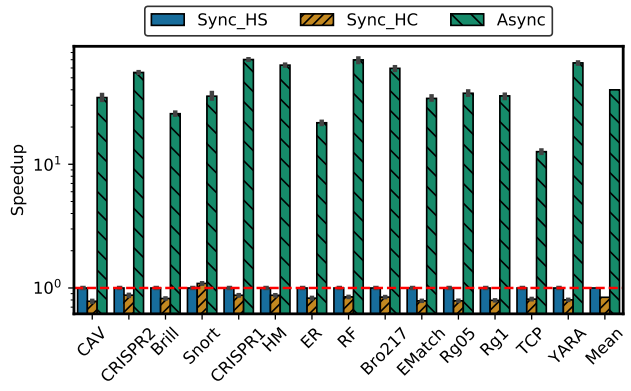
5.5.2 Experimental Results

We describe the performance results under different scenarios. Table 5.6 shows the absolute performance of evaluated applications in different scenarios.

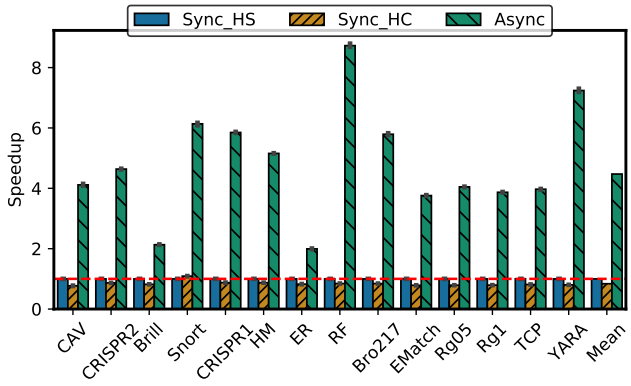
To simplify the discussion, we exclude the application, PEN, from this section and discuss it separately.

Low and Medium Parallelism In this scenario, the approaches of GPU-NFA only need one thread block to execute the NFAs on an input stream. Thus, originally, it severely underutilized GPU cores. Figure 5.9a shows the throughput obtained from this scenario. We found that AsyncAP achieves 40× speedup for the evaluated applications on average. Figure 5.9b shows that 4.5× speedup is achieved on average in the medium parallelism scenario. We conclude that *for all applications* AsyncAP *achieves significant speedup showing its effectiveness in increasing the parallelism of automata processing tasks.*

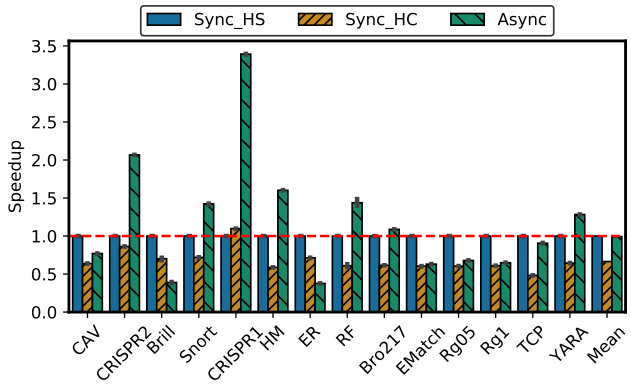
High Parallelism To show that AsyncAP adapts to all scenarios in terms of parallelism, we compare AsyncAP with GPU-NFA when the GPU is fully utilized. Figure 5.9c show our results. Overall, we observe that AsyncAP *achieves 99.3% of the performance compared to Sync_HS, the fastest evaluated implementation of* GPU-NFA*, on average*



(a) Low Parallelism



(b) Medium Parallelism



(c) High Parallelism

Figure 5.9: Performance of synchronous and asynchronous automata executions on GPU under different amounts of parallelism

across the evaluated applications. AsyncAP achieve up to 3.4× speedup over GPU-NFA (CRISPR1). Although CRISPR1, CRISPR2, HM, and Snort have more total work

in AsyncAP (up to 1.95× than GPU-NFA), AsyncAP still significantly outperforms GPU-NFA in these applications. To further study the reason, we observe that AsyncAP has more advantages in these applications that have more activation or have longer *average* pattern lengths (will be demonstrated in Table 5.7), showing the lower overhead of state matching without synchronization.

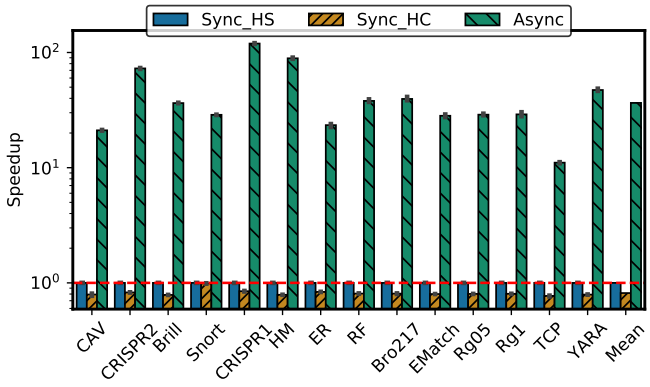
On the other hand, AsyncAP has a slowdown in several applications due to two reasons: First, a few applications such as Brill and ER have more total work (4.9× and 3.2×, respectively) in AsyncAP. As a result, AsyncAP achieves around 37% performance of GPU-NFA in ER and 38% in Brill, but the slowdown is lighter than the incurred more total work in AsyncAP. Second, applications, such as EMatch, Rg05, Rg1, and TCP, although do not need more work in AsyncAP, have a slowdown because their characteristics are favorable for GPU-NFA. We further investigate these applications and observe that the matching processes do not go deeper in their NFAs. Also, the *all-input* starting states of them have compressible matchsets, on which GPU-NFA applied matchset compression, so GPU-NFA achieves better performance for them in the full parallelism scenario.

Table 5.6: Absolute throughput (in MB/s) of evaluated applications under the scenarios with different amounts of parallelism.

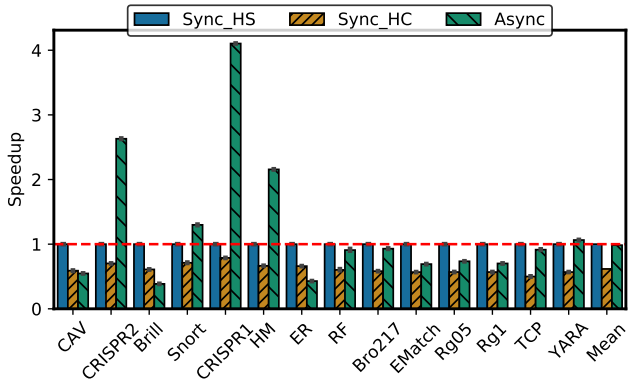
Scenario	Scheme	Application														
		Brill	Bro217	CAV	CRISPR1	CRISPR2	EMatch	ER	HM	PEN	RF	Rg05	Rg1	Snort	TCP	YARA
Low	AsyncAP	14.66	52.93	55.48	13.71	19.48	38.13	28.28	12.99	0.30	64.60	39.68	38.83	19.47	11.60	51.33
	Sync_HS	0.57	0.89	1.60	0.20	0.35	1.12	1.31	0.21	0.77	0.93	1.06	1.09	0.55	0.92	0.78
Medium	AsyncAP	18.32	77.38	99.22	16.99	24.53	63.59	38.82	15.80	0.28	120.53	64.54	63.68	50.45	54.77	84.27
	Sync_HS	8.59	13.36	24.11	2.90	5.29	16.91	19.48	3.06	11.46	13.81	15.94	16.46	8.22	13.80	11.64
High	AsyncAP	19.19	103.65	114.23	17.43	25.64	71.84	41.43	16.25	1.94	139.56	73.35	71.94	59.69	74.70	95.99
	Sync_HS	49.38	95.30	148.31	5.14	12.41	113.94	110.00	10.14	83.79	97.12	108.15	110.82	41.98	82.47	74.79

Comparison with Oracular Speculation A speculation mechanism for automata processing needs two components: 1) Speculation component decides the state to start for each input stream chunk; 2) The matching component to handle all the input stream chunks and the starting states. Currently, no prior speculation scheme is implemented on GPU, so we perform an oracular case study. We suppose the speculation component is oracular, and use GPU-NFA, the state-of-the-art scheme on GPU, as the matching com-

ponent for the input stream chunks. This ideal speculation scenario is equivalent to having more input streams for free. Figure 5.9 (3) is a proxy comparison between AsyncAP and the ideal speculation. In conclusion, AsyncAP achieves comparable performance to *ideal speculation* with GPU-NFA [69] as the matching component.



(a) Low Parallelism



(b) High Parallelism

Figure 5.10: Performance sensitivity to Ampere GPU Architecture

Sensitivity to Ampere Architecture We perform *high parallelism* and *low parallelism* scenarios on NVIDIA A100 GPU. Since A100 has 108 SMs, we use 864 (108×8) input streams in the high parallelism scenario. Figure 5.10 demonstrates the performance results. In the low parallelism scenario, we observe that several applications (e.g., CRISPR2, CRISPR1, and HM) achieve better speedup compared to the P6000 GPU because more

cores benefit from increased parallelism. For example, CRISPR1 achieves 120× speedup in A100 compared to 70× speedup in P6000. However, compared with Figure 5.9a, a few applications achieve lower speedup (e.g., 21.1× vs. 34.6× speedup in CAV, and 47.2× vs. 65.9× speedup in YARA) than the results on P6000 GPU. We found that most matchsets of them are compressible (i.e., matchset compression in GPU-NFA are applied). Hence, GPU-NFA obtains better performance due to lower latency in A100. On average, AsyncAP achieves 36.5× speedup (y-axis is in log-scale). In the high parallelism scenario, AsyncAP achieves 98.4% throughput compared to GPU-NFA on average across the 14 evaluated applications. *We conclude that* AsyncAP *is also effective in the A100 GPU.*

5.5.3 Analysis of Pattern Lengths

Slowdown of PEN Figure 5.11 shows that AsyncAP has an extreme slowdown in PEN in all scenarios. The slowdown is because PEN has very long patterns, so the matching processes get stalled at certain states.

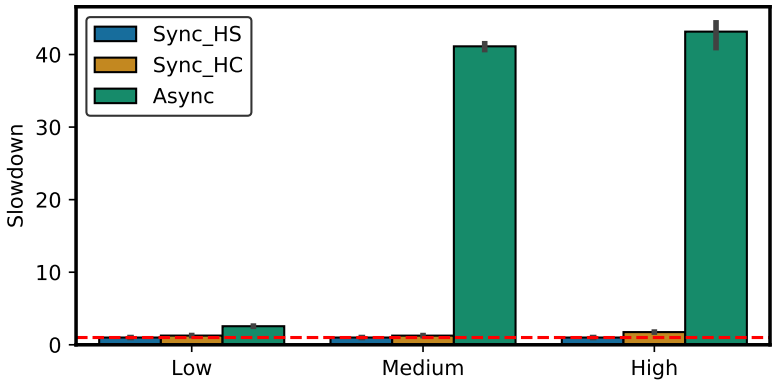


Figure 5.11: Slowdown of PEN

We further examine the patterns of applications to understand the slowdown. Table 5.7 shows the pattern lengths characteristics.

We observe that most applications have very short patterns, and have a small standard deviation. Applications that have larger average pattern lengths (e.g., CRISPR1, CRISPR2, HM) are more likely to achieve better performance in AsyncAP because longer

Table 5.7: Pattern Lengths of Applications

App.	Max. Len.	Avg. Len.	SD.
Brill	75	1.591	2.044
Bro217	46	1.034	0.274
CRISPR2	16	2.667	0.943
CRISPR1	24	4.016	1.257
CAV	13	1.004	0.069
ER	22	1.136	0.542
EMatch	42	1.020	0.232
HM	19	5.333	1.333
PEN	923049	1.053	142.436
RF	1	1.000	0.000
Rg05	46	1.020	0.230
Rg1	46	1.022	0.239
Snort	9558	1.313	6.828
TCP	1108	1.022	1.102
YARA	27	1.125	0.350

patterns indicate more frequent activation. When non-starting states are activated frequently, the worklist used in GPU-NFA needs to read the per-node data structure of NFAs, which causes a huge data movement overhead. However, a few applications can have very long patterns (e.g., PEN, Snort), but these two have significantly different standard deviations (142.4 and 6.8, respectively), leading to their different performance results. When the number of warps that execute on shorter patterns can offset the warps for long patterns, the performance degradation is not much (e.g., Snort). If the number of thread blocks with short patterns cannot offset the thread blocks with extreme long patterns, the performance drops significantly (PEN) due to imbalance.

In summary, AsyncAP performs well in all scenarios when the pattern lengths are moderate but can lead to slowdown when the patterns are extremely long.

How likely is it for the pattern to be long? To study how common the long patterns are, we classify the NFAs into four types by static analysis on NFA graphs, as NFAs are also direct graphs on which the state transitions follow the edges.

1. The first type can generate patterns with infinite lengths when going through a cycle

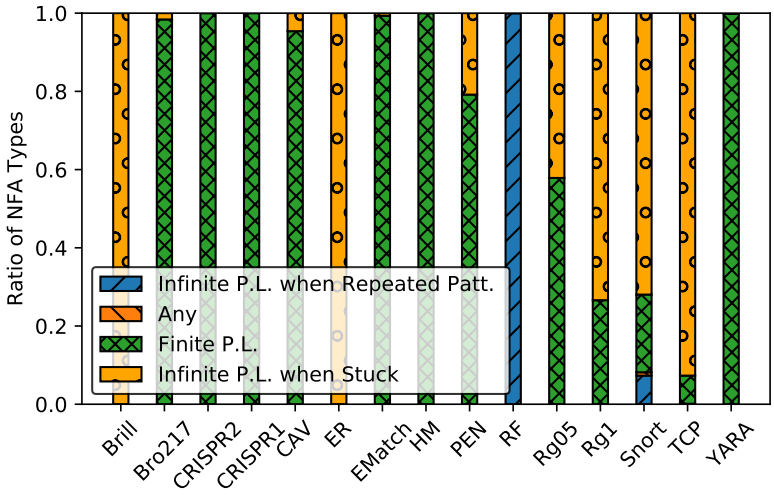


Figure 5.12: Large portion of NFAs can never have infinite long patterns.

containing more than one state (the NFA graph has no state with self-loop but has back edges).

2. The second type can generate any lengths of patterns (the NFA graph has both back edge and self-loop).

3. The third type only generates patterns with finite lengths (the NFA graphs are directed acyclic graphs).

4. The fourth type can generate patterns with infinite lengths only when the matching process is stuck on the same state (the NFA graph has no back edge but has self-loops).

Figure 5.12 shows the results for the applications (Here, NFAs are not sampled. The sampled case also shows a very similar figure). We observe that *a large portion of the NFAs can only generate patterns with finite lengths*. In other applications, although statically it is possible to have very long patterns, they do not have long patterns when executing a real input stream (see Table 5.7). Although most NFAs in PEN belong to the third type, the NFAs generate long patterns are in the fourth type. For example, state `__26986__` that

accepts `[x00-x09x0b-xff]` has a self-loop. The matching process is stalled at this state leading to an extremely long pattern.

Addressing Slowdown caused by Long Patterns We use a simple but effective way to dynamically detect the extreme slowdown. A thread of AsyncAP terminates when it has no matched states. We set another limit to the termination condition that when the current loop exceeds a pre-defined length the thread is terminated.

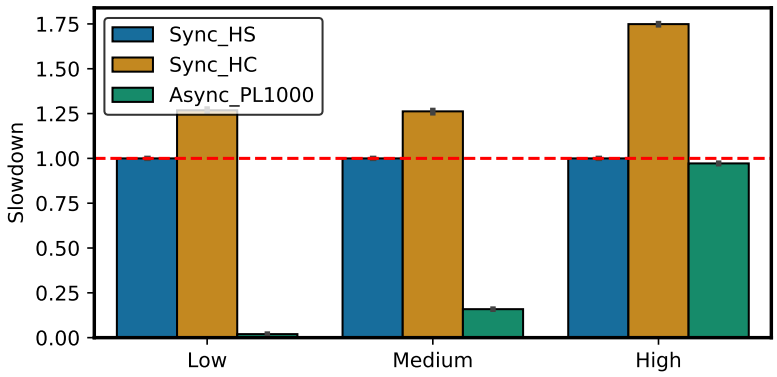


Figure 5.13: When limiting the pattern length to 1K, PEN does not exhibit slowdown (i.e., slowdown is less than 1).

Figure 5.13 demonstrates that when we apply the long pattern detection and termination on AsyncAP, AsyncAP has speedup in all cases for PEN (slowdown is less than 1) compared to schemes in GPU-NFA. While such an early termination mechanism may omit reports, those long patterns did not trigger any reporting state in the case of PEN.

5.6 Related Work

This section summarizes prior works into two categories. First, we discuss how prior works maps automata processing to hardware resources. Second, we discuss existing works that increase the parallelism of automata.

5.6.1 Mapping Automata to Computation Resources

To parallelize the automata processing, the program needs to map automata to computation resources. Vasiliadis et al. [118] and Gregex [128] use GPU threads to handle different network packets (i.e., input stream level parallelism). Smith et al. [105] merges DFAs (and extended DFAs) with a 64MB memory budget and executes them in batches on GPU threads. Tran et al. [114] use each warp to handle a pair of NFA and input stream. HyperScan [131] maps NFA states to SIMD lanes of CPU and uses parallel bit operations of SIMD to calculate the next states of NFA. Prior work by Vu [120] maps threads to state vectors and performs vector and matrix multiplication to leverage bit-parallelism of NFAs. iNFAnt [36] and Nourian et al. [76] map state transitions to threads. Nourian et al. also propose compiler-based schemes to map the traversals of NFAs to the code of GPUs [77] or FPGAs [78], but these approaches are limited to fixed-topology NFAs.

These works treat all states equally, neglecting the matching activity of automata states. Zu et al. [154] statically groups the states that can never be grouped into compatible groups, and map the compatible groups to threads to increase the GPU utilization. GPU-NFA [69] classifies the states to hot and cold and maps the hot states to threads.

In contrast, AsyncAP maps input symbols to GPU threads. Since the input stream is long enough, the mapping of our approach increases the parallelism.

5.6.2 Increasing Parallelism of Automata

A large body of work focuses on how to break dependencies across symbols to gain more parallelism. The approaches can be categorized as prefix-sum parallelization [63], enumeration [74], speculation [149, 148, 83, 85], or hybrid methods of them [55, 138]. All of these works assume the computational resources are more than the requirement of the parallelism coming from the original task (i.e., automata level, input stream level, and state-level parallelism). Speculation is more work-efficient than prefix-sum parallelization and enumeration [149].

All the aforementioned works focus on DFAs because they are simpler in transitions, which makes it easier to speculate or apply path merging [74] in enumeration schemes. Also, since these works are not available on GPU due to the complexity of GPU implementation, we do not compare AsyncAP with them directly. Nevertheless, we compared AsyncAP with an ideal speculation approach that uses GPU-NFA as matching processes in Section 5.5.2 and found AsyncAP has comparable performance as an *ideal* speculation approach.

The separation of execution paths of AsyncAP can also be treated as a variant of enumeration. However, traditional enumeration schemes enumerate the active states at the beginning of each chunk. The matching processes need to be synchronized because they need to discard the incorrect results from the enumeration of active states. In contrast, AsyncAP enumerates the starting positions of the input stream to gain parallelism and does not require synchronization to guarantee correctness. As a result, AsyncAP is more scalable when the processor becomes larger, or the user distributes the workloads to multiple devices. The limitation of AsyncAP is that it only applies to *all-input* starting states.

No prior works adapt to tasks with different levels of parallelism, whereas we have demonstrated in this paper that AsyncAP work well for all levels of parallelism and incurs low overhead.

5.7 Conclusions

With each generation, GPUs are becoming more capable and equipped with more compute/memory resources. However, not all automata tasks bring enough parallelism to utilize these GPUs well.

We propose a lightweight approach to increase the parallelism of automata processing on GPUs. The new approach scales with the input stream and is able to search patterns in parallel *asynchronously*. However, theoretically, such an approach can generate

additional work. We study the amount of work in real applications and observe that our approach only incurs marginal extra work compared to the traditional synchronous automata processing on GPUs.

The increased parallelism enabled by our new mapping approach reduces the under-utilization of GPU cores – leading to significant speedup when parallelism of the task does not saturate the GPU. When the task has enough parallelism, our approach performs comparably to the state-of-the-art NFA processing engine on GPU.

Chapter 6

Conclusions and Future Work

6.1 Summary of Dissertation Contributions

Achieving high throughput in large-scale automata processing is challenging. CPUs are not ideal for this problem because of their limited parallelism and memory bandwidth. Mainstream solutions include the use of domain-specific accelerators or general-purpose accelerators. However, both of them have their inefficiencies, leading to hardware underutilization. To solve such underutilization, in this dissertation, we make the following contributions:

Addressing the Underutilization of the Domain-specific Accelerator. Although Automata Processor is orders of magnitude faster than von Neumann architectures, we show that the AP is underutilized because it fails to leverage the runtime characteristics of the automata. This underutilization leads to suboptimal performance as well as performance per area. We address the problem via software/hardware co-design optimizations which result in a new mode of AP. With marginal hardware overhead, our approaches achieve significant speedup as well as performance per area.

Reducing the Gap between General-purpose Accelerators and Domain-specific Accelerators. Domain-specific accelerators are fast but have limited availability. Further, too many heterogeneous accelerators lead to more complex computing systems. We,

therefore, focus on the general-purpose accelerator, GPU, which provides massive parallelism and very high memory bandwidth in commodity processors. We identified two major throughput bottlenecks. By our proposed optimizations, we achieve significant improvement in automata processing throughput over state-of-the-art mechanisms across a wide range of emerging applications. Moreover, our optimizations enable GPUs to outperform the domain-specific accelerator (AP) for several applications while being within an order of magnitude of AP performance for the remainder.

Adapting Automata Processing to Different Task Sizes on GPU. GPU is becoming more capable in computation and memory resources. However, existing works that leverage three levels of parallelism of automata cannot always utilize GPU cores well. To adapt to different task sizes with a more generic scheme, we propose AsyncAP, a new approach that processes automata asynchronously on GPU. Our approach exploits an additional source of parallelism, symbol-level parallelism, that scales with the length of an input stream. The new approach has more theoretical time complexity, however, our characterization shows that it only incurs 5% more work. The evaluation shows up to 40× speedup is obtained because of the increased parallelism on average across 14 applications. Further, the new approach achieves comparable performance to the state-of-the-art GPU automata processing engine when the GPU is fully used.

6.2 Future Research Directions

Our future work targets to bridge the performance/energy-efficiency gap between DSAs and general-purpose accelerators (e.g., GPUs).

For automata processing, in this dissertation, while we have proposed and implemented many optimizations for automata processing on GPUs, as well as different ways to map automata to GPU resources, our approaches do not exhaust all combinations of them. Due to the diversity of GPUs and automata applications, generating a specific combination of optimizations can enable the various automata applications to utilize different GPUs more

efficiently. Our future work will develop an automatic way to generate the combinations of optimizations and evaluate them on GPUs.

We will also broaden the research domain to other irregular computations such as graphs, bioinformatics algorithms, and neural architecture search.

Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) grants (#1657336 and #1750667). This work was performed in part using computing facilities at William & Mary. The Quadro P6000 GPU was donated by NVIDIA.

Bibliography

[1] Project Brainwave. https://www.microsoft.com/en-us/research/project/project-brainwave/.

[2] PROJECT TRILLIUM. https://www.arm.com/products/silicon-ip-cpu/machine-learning/project-trillium.

[3] The NVIDIA Deep Learning Accelerator (NVDLA). http://nvdla.org/.

[4] Google Pixel Visual Core. https://en.wikichip.org/wiki/google/pixel_visual_core, 2017.

[5] ANML Documentation. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html, 2018.

[6] Clamav net. https://www.clamav.net/, 2018.

[7] CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2019.

[8] GNU Grep. https://www.gnu.org/software/grep/, 2019.

[9] NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2019.

[10] Snort. https://snort.org/, 2019.

[11] The Lex & Yacc Page. `http://dinosaur.compilertools.net`, 2019.

[12] The Zeek Network Security Monitor. `https://www.zeek.org`, 2019.

[13] YARA: The pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`, 2019.

[14] CUDA Occupancy Calculator. `https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls`, 2021.

[15] NVIDIA Nsight Compute. `https://docs.nvidia.com/cuda/profiler-users-guide/`, 2021.

[16] NVIDIA Nsight Compute. `https://developer.nvidia.com/nsight-compute`, 2021.

[17] NVIDIA Visual Profiler. `https://developer.nvidia.com/nvidia-visual-profiler`, 2021.

[18] Manel Abdellatif, Chamseddine Talhi, Abdelawahab Hamou-Lhadj, and Michel Dagenais. On the Use of Mobile GPU for Accelerating Malware Detection Using Trace Analysis. In *Proceedings of the Symposium on Reliable Distributed Systems Workshop (SRDSW)*, 2015.

[19] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. A Many-core Architecture for In-memory Data Processing. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, 2017.

[20] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem. *IEEE Computer Architecture Letters (CAL)*, 2018.

[21] 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 *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018.

[22] P. Ashar, S. Devadas, and A. R. Newton. A Unified Approach to the Decomposition and Re-decomposition of Sequential Machines. In *Proceedings of the Design Automation Conference (DAC)*, 1990.

[23] Pranav Ashar, Srinivas Devadas, and A. Richard Newton. *Finite State Machine Decomposition*, pages 117–168. Springer US, 1992.

[24] 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 (ToN)*, 2016.

[25] John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978.

[26] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[27] Michela Becchi and Srihari Cadambi. Memory-Efficient Regular Expression Search Using State Merging. In *Proceedings of the International Conference on Computer Communications (INFOCOM)*, 2007.

[28] 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 (ANCS)*, 2007.

[29] Michela Becchi and Patrick Crowley. Efficient Regular Expression Evaluation: Theory to Practice. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008.

[30] Michela Becchi, Mark Franklin, and Patrick Crowley. A Workload for Evaluating Deep Packet Inspection Architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2008.

[31] Michela Becchi, Mark Franklin, and Patrick Crowley. A Workload for Evaluating Deep Packet Inspection Architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2008.

[32] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating Regular Expression Matching Engines on Network and General Purpose Processors. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.

[33] C. Bo, K. Wang, J. J. Fox, and K. Skadron. Entity resolution acceleration using the automata processor. In *Proceedings of the International Conference on Big Data (BigData)*, 2016.

[34] 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 *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[35] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006.

[36] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *ACM SIGCOMM Computer Communication Review (CCR)*, 2010.

[37] Yeim-Kuan Chang and Yu-Hao Tseng. Fast and Memory Efficient NFA Pattern Matching using GPU. In *Proceedings of the International Conference on Communications, Computation, Networks and Technologies (INNOV)*, 2016.

[38] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[39] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[40] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[41] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. Bespoke Processors for Applications with Ultra-low Area and Power Constraints. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[42] Thomas T. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[43] Russ Cox. Regular Expression Matching can be Simple and Fast. `https://swtch.com/~rsc/regexp/regexp1.html`, 2007.

[44] Paul Dlugosch, Dave Brown, Paul Glendenning, Leventhal Leventhal, and Harold Noyes. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2014.

[45] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[46] Amr S. Elhelw and Sreepathi Pai. Horus: A Modular GPU Emulator Framework. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[47] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015.

[48] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2015.

[49] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy. GenAx: A Genome Sequencing Accelerator. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[50] Mingyu Gao, Christina Delimitrou, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Christos Kozyrakis. DRAF: A

Low-power DRAM-based Reconfigurable Acceleration Fabric. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2016.

[51] Victor Mikhaylovich Glushkov. The Abstract Theory of Automata. *Russian Mathematical Surveys*, 16(5):1, 1961.

[52] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Transactions on Database Systems*, 2004.

[53] Timothy Heil, Anil Krishna, Nicholas Lindberg, Farnaz Toussi, and Steven Vanderwiel. Architecture and Performance of the Hardware Accelerators in IBM's PowerEN Processor. *ACM Transactions on Parallel Computing (TOPC)*, 2014.

[54] Mohamed Assem Ibrahim, Hongyuan Liu, Onur Kayiran, and Adwait Jog. Analyzing and Leveraging Remote-Core Bandwidth for Enhanced Performance in GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[55] Peng Jiang and Gagan Agrawal. Combining SIMD and Many/Multi-Core Parallelism for Finite State Machines with Enumerative Speculation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[56] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[57] Adwait Jog, Onur Kayiran, Nachiappan C. Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R.

Das. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *ASPLOS*, 2013.

[58] Norman P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[59] R. Karakchi, L. O. Richards, and J. D. Bakos. A Dynamically Reconfigurable Automata Processor Overlay. In *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017.

[60] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[61] HyunJin Kim and Kang-Il Choi. A Pipelined Non-Deterministic Finite Automaton-Based String Matching Scheme Using Merged State Transitions in an FPGA. *PLOS ONE*, 11, 2016.

[62] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2006.

[63] Richard E. Ladner and Michael J. Fischer. Parallel Prefix Computation. *Journal of the ACM (JACM)*, 1980.

[64] Huanxin Lin, Cho-Li Wang, and Hongyuan Liu. On-GPU Thread-data Remapping for Branch Divergence Reduction. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):1–24, 2018.

[65] Z. Lin, L. Nyland, and H. Zhou. Enabling Efficient Preemption for SIMT Architectures with Lightweight Context Switching. In *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2016.

[66] Hongyuan Liu, Mohamed Ibrahim, Onur Kayiran, Sreepathi Pai, and Adwait Jog. Architectural Support for Efficient Large-Scale Automata Processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018. © 2018 IEEE. Reprinted, with permission.

[67] Hongyuan Liu, King Tin Lam, Huanxin Lin, Cho-Li Wang, and Junchao Ma. Lightweight Dependency Checking for Parallelizing Loops with Non-Deterministic Dependency on GPU. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[68] Hongyuan Liu, Bogdan Nicolae, Sheng Di, Franck Cappello, and Adwait Jog. Accelerating DNN Architecture Search at Scale Using Selective Weight Transfer. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2021.

[69] 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*, ASPLOS '20, page 251–265, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378471.

[70] Y. Liu, S. Sezer, and J. McCanny. NFA Decomposition and Multiprocessing Architecture for Parallel Regular Expression Processing. In *Proceedings of the International SOC Conference*, 2009.

[71] Paul Merolla, John Arthur, Filipp Akopyan, Nabil Imam, Rajit Manohar, and Dharmendra S Modha. A Digital Neurosynaptic Core using Embedded Crossbar Memory with 45pJ per Spike in 45nm. In *IEEE Custom Integrated Circuits Conference (CICC)*, 2011.

[72] José C. Monteiro and Arlindo L. Oliveira. Finite state machine decomposition for low power. In *Proceedings of the Design Automation Conference (DAC)*, 1998.

[73] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *Proceedings of the VLDB Endowment*, 2009.

[74] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. Data-parallel Finite-state Machines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[75] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[76] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying Automata Processing: GPUs, FPGAs or Micron's AP? In *Proceedings of the International Conference on Supercomputing (ICS)*, 2017.

[77] Marziyeh Nourian, Hancheng Wu, and Michela Becchi. A Compiler Framework for Fixed-Topology Non-Deterministic Finite Automata on SIMD Platforms. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, 2018.

[78] Marziyeh Nourian, Mostafa Eghbali Zarch, and Michela Becchi. Optimizing Complex OpenCL Code for FPGA: A Case Study on Finite Automata Traversal. In *Proceedings of the IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, 2020.

[79] Sreepathi Pai and Keshav Pingali. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the International Conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2016.

[80] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[81] Mateja Putic, A. J. Varshneya, and Mircea R. Stan. Hierarchical Temporal Memory on the Automata Processor. *IEEE Micro*, 2017.

[82] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. Challenging Sequential Bitstream Processing via Principled Bitwise Speculation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[83] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. Scalable FSM Parallelization via Path Fusion and Higher-Order Speculation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[84] 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 (PACT)*, 2016.

[85] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling Scalability-sensitive Speculative Parallelization for FSM Computations. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2017.

[86] Junqiao Qiu, Zhijia Zhao, Bo Wu, Abhinav Vishnu, and Shuaiwen Leon Song. Enabling Scalability-sensitive Speculative Parallelization for FSM Compu-

tations. In *Proceedings of the International Conference on Supercomputing (ICS)*. ACM, 2017.

[87] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs. In *Proceedings of the IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020.

[88] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2013.

[89] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Conference on System Administration (LISA)*, 1999.

[90] I. Roy and S. Aluru. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[91] I. Roy, N. Jammula, and S. Aluru. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

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

[93] 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 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2018.

[94] 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 (ASPLOS)*, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378459.

[95] Elaheh Sadredini, Reza Rahimi, Lenjani Marzieh, Stan Mircea, and Skadron Kevin. Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2020.

[96] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mohsen Imani, and Kevin Skadron. Sunder: Enabling Low-Overhead and Scalable Near-Data Pattern Matching Acceleration. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2021.

[97] 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 (CAL)*, 2019.

[98] 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 International Symposium on Microarchitecture (MICRO)*, 2019.

[99] Elaheh Sadredini, Reza Rahimi, Ke Wang, and Kevin Skadron. Frequent Subtree Mining on the Automata Processor: Challenges and Opportunities. In *Proceedings of the International Conference on Supercomputing (ICS)*, 2017.

[100] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2017.

[101] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

[102] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[103] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *Proceedings of the ACM SIGCOMM Conference on Data Communication (SIGCOMM)*, 2008.

[104] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for network packet signature matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[105] Randy Smith, Neelam Goyal, Justin Ormont, Karthikeyan Sankaralingam, and Cristian Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[106] Arun Subramaniyan and Reetuparna Das. Parallel Automata Processor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[107] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache Automaton. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[108] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Summarizing CPU and GPU Design Trends with Product Data, 2020.

[109] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling Preemptive Multiprogramming on GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[110] Olivier Temam. A Defect-tolerant Accelerator for Emerging High-performance Applications. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012.

[111] Andrew Todd, Marziyeh Nourian, and Michela Becchi. A Memory-Efficient GPU Method for Hamming and Levenshtein Distance Similarity. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2017.

[112] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2016.

[113] Tommy Tracy, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, 2016.

[114] Tuan Tu Tran, Yongchao Liu, and Bertil Schmidt. Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi. *Parallel Computing*, 54:128–138, 2016.

[115] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[116] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. GASPP: A gpu-accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference (ATC)*, 2014.

[117] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[118] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2011.

[119] Lucas Vespa and Ning Weng. Deterministic Finite Automata Characterization and Optimization for Scalable Pattern Matching. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2011.

[120] Kien Chi Vu. Accelerating bit-based finite automaton on a GPGPU device. 2020.

[121] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron. Generating efficient and high-quality pseudo-random behavior on Automata Processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, 2016.

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

[123] Jack Wadden, Vinh Dang, Nathan Brunelle, Tom Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in

Automata Processing Engines and Architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2016.

[124] Jack Wadden and Kevin Skadron. VASim: An Open Virtual Automata Simulator for Automata Processing Application and Architecture Research. Technical Report CS2016-03, University of Virginia, 2016.

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

[126] 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 *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016.

[127] Ke Wang, Elaheh Sadredini, and Kevin Skadron. Sequential Pattern Mining with the Micron Automata Processor. In *Proceedings of the International Conference on Computing Frontiers (CF)*, 2016.

[128] Lei Wang, Shuhui Chen, Yong Tang, and Jinshu Su. Gregex: GPU Based High Speed Regular Expression Matching Engine. In *Proceedings of the International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011.

[129] Qihan Wang, Wei Niu, Li Chen, Ruoming Jin, and Bin Ren. HEALS: A Parallel eALS Recommendation System on CPU/GPU Heterogeneous Platforms. In *HiPC*, 2021.

[130] Xiang Wang. Techniques for Efficient Regular Expression Matching across Hardware Architectures. Master's thesis, University of Missouri–Columbia, 2014.

[131] 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 *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[132] Qiong Wu, Christopher G. Brinton, Zheng Zhang, Andrea Pizzoferrato, Mihai Cucuringu, and Zhenming Liu. Equity2Vec: End-to-end Deep Learning Framework for Cross-sectional Asset Pricing. In *ICAIF*, 2021.

[133] Qiong Wu, Adam Hare, Sirui Wang, Yuwei Tu, Zhenming Liu, Christopher G Brinton, and Yanhua Li. BATS: A Spectral Biclustering Approach to Single Document Topic Modeling and Segmentation. *TIST*, 2021.

[134] Qiong Wu, Wen-Ling Hsu, Tan Xu, Zhenming Liu, George Ma, Guy Jacobson, and Shuai Zhao. Speaking with Actions—Learning Customer Journey Behavior. In *ICSC*, 2019.

[135] Qiong Wu, Lucas C. K. Hui, C. Y. Yeung, and T. W. Chim. Early Car Collision Prediction in VANET. In *Proceedings of the International Conference on Connected Vehicles and Expo (ICCVE)*, 2015.

[136] Qiong Wu and Zhenming Liu. Rosella: A Self-Driving Distributed Scheduler for Heterogeneous Clusters. In *Proceedings of the International Conference on Mobility, Sensing and Networking (MSN)*, 2021.

[137] Qiong Wu, Felix M.F. Wong, Yanhua Li, Zhenming Liu, and Varun Kanade. Adaptive Reduced Rank Regression. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[138] Yang Xia, Peng Jiang, and Gagan Agrawal. Scaling out Speculative Execution of Finite-State Machines with Parallel Merge. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020.

[139] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. REAPR: Reconfigurable engine for automata processing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[140] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu Ming Yiu, and Lucas Chi Kwong Hui. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys Tutorials*, 2016.

[141] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Improving NFA-based Signature Matching using Ordered Binary Decision Diagrams. In *International Workshop on Recent Advances in Intrusion Detection*, 2010.

[142] Yi-Hua Yang and Viktor Prasanna. High-performance and compact architecture for regular expression matching on fpga. *IEEE Transactions on Computers*, 2012.

[143] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2006.

[144] X. Yu, K. Hou, H. Wang, and W. C. Feng. Robotomata: A Framework for Approximate Pattern Matching of Big Data on an Automata Processor. In *Proceedings of the International Conference on Big Data (BigData)*, 2017.

[145] Xiaodong Yu and Michela Becchi. Exploring Different Automata Representations for Efficient Regular Expression Matching on GPUs. In *Proceedings of the Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[146] Xiaodong Yu and Michela Becchi. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the International Conference on Computing Frontiers (CF)*, 2013.

[147] Xiaodong Yu, Wu-chun Feng, Danfeng Yao, and Michela Becchi. O3FA: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection. In *Proceedings of the 2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2016.

[148] Zhijia Zhao and Xipeng Shen. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[149] Zhijia Zhao, Bo Wu, and Xipeng Shen. Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-based Computations Through Principled Speculation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[150] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[151] K. Zhou, J. Wadden, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron. Regular expression acceleration on the micron automata processor: Brill tagging as a case study. In *Proceedings of the International Conference on Big Data (BigData)*, 2015.

[152] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. Brill tagging on the Micron Automata Processor. In *Proceedings of the International Conference on Semantic Computing (ICSC)*, 2015.

[153] Youwei Zhuo, Jinglei Cheng, Qinyi Luo, Jidong Zhai, Yanzhi Wang, Zhongzhi Luan, and Xuehai Qian. CSE: Convergence Set Based Enumerative FSM. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2018.

[154] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

VITA

Hongyuan Liu

Hongyuan Liu is a Ph.D. candidate in the Department of Computer Science at William & Mary under the supervision of Professor Adwait Jog. Hongyuan's research interests lie in the broad area of computer architecture, emphasizing domain-specific accelerators and GPUs. His Ph.D. research has been published in MICRO 2018, ASPLOS 2020, and CLUSTER 2021. Hongyuan interned with Intel in Fall 2019, and worked as a visiting student with Argonne National Lab in Spring 2021. Before joining William & Mary, he received his bachelor's degree from Shandong University and master's degree from the University of Hong Kong.