*ng*AP: Non-blocking Large-scale Automata Processing on GPUs

Tianao Ge
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
tge601@connect.hkust-gz.edu.cn

Tong Zhang
Samsung Electronics
San Jose, CA, USA
t.zhang2@samsung.com

Hongyuan Liu
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
hongyuanliu@hkust-gz.edu.cn

Abstract

Finite automata serve as compute kernels for various applications that require high throughput. However, despite the increasing compute power of GPUs, their potential in processing automata remains underutilized. In this work, we identify three major challenges that limit GPU throughput. 1) The available parallelism is insufficient, resulting in underutilized GPU threads. 2) Automata workloads involve significant redundant computations since a portion of states matches with repeated symbols. 3) The mapping between threads and states is switched dynamically, leading to poor data locality. Our key insight is that processing automata "one-symbol-at-a-time" serializes the execution, and thus needs to be revamped. To address these challenges, we propose Non-blocking Automata Processing, which allows parallel processing of different symbols in the input stream and also enables further optimizations: 1) We prefetch a portion of computations to increase the chances of processing multiple symbols simultaneously, thereby utilizing GPU threads better. 2) To reduce redundant computations, we store repeated computations in a memoization table, enabling us to substitute them with table lookups. 3) We privatize some computations to preserve the mapping between threads and states, thus improving data locality. Experimental results demonstrate that our approach outperforms the state-of-the-art GPU automata processing engine by an average of 7.9× and up to 901× across 20 applications.

CCS Concepts: • **Computing methodologies** → **Massively parallel algorithms**; • **Computer systems organization** → **Single instruction, multiple data**; • **Theory of computation** → **Formal languages and automata theory**.

Keywords: Finite State Machine; GPU; Parallel Computing

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0372-0/24/04.
https://doi.org/10.1145/3617232.3624848

ACM Reference Format:
Tianao Ge, Tong Zhang, and Hongyuan Liu. 2024. *ng*AP: Non-blocking Large-scale Automata Processing on GPUs. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages.
https://doi.org/10.1145/3617232.3624848

1 Introduction

Finite automata are workhorses for many applications across various domains such as data analytics [10, 31, 49, 54, 76], machine learning [62], network intrusion detection [42, 46, 73], graph processing [48], and bioinformatics [18, 47]. Widely used regular expression engines also use finite automata as the compute kernels to match patterns [2, 7, 68]. However, processing automata on compute-centric architectures is extremely challenging due to irregular accesses and data dependency. The former causes poor cache performance, while the latter serializes the execution, known as "embarrassingly sequential" [75]. Worse, in recent years, applications with automata as compute kernels have become increasingly larger [41]. For example, Snort [46] is a network intrusion detection and prevention system that matches a series of rules to identify malicious network activity from many packets (input streams), where each rule can be represented as an automaton. From 2014 to 2021, the number of rules has increased by 71% in a ruleset (ET Pro) [6] while the third-party users continue contributing new rulesets.

To provide high throughput for such large-scale automata processing applications, many domain-specific accelerators are proposed. Many of them address the irregular data movement with processing-in-memory architectures [22, 29, 50–53, 59, 60, 64]. While they could achieve orders of magnitude higher performance than von Neumann architectures [39], they lack programmability [13, 21, 23] and are not easily accessible to all users [40]. Integrating many domain-specific accelerators into a computer system for diverse workloads also leads to higher heterogeneity and complexity [63].

Due to their massive data parallelism, GPUs have become the most widely accepted general-purpose accelerator [37], and continue to scale faster than CPUs [61]. Processing large-scale automata applications on GPUs is therefore an attractive option if high throughput can be achieved. Most previous

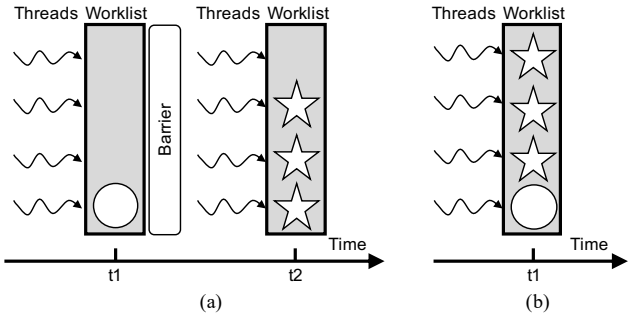


Figure 1. Comparison of processing automata using the prior "blocking" approach (a) vs. the "non-blocking" approach (b) proposed in this work: (a) The states matched with two adjacent symbols are denoted by ○ and ☆. A barrier is required between them; (b) Our proposal allows matching states with different symbols in parallel, making the process "non-blocking".

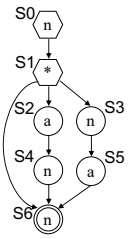
works have used two representations of automata: Deterministic Finite Automata (DFAs) and Non-deterministic Finite Automata (NFAs). Since NFAs allow multiple *active* states and are compact in size, they can better utilize the parallelism of GPUs [34]. While DFAs are inherently NFAs by definition [55], they only permit one state to be active at any time, thereby losing a potential source of parallelism. Although processing automata on GPUs has been studied for many years, early works that use small automata up to a few thousand states [20, 63, 77] perform suboptimally on large-scale automata applications. In contrast, recent work [34] has improved the throughput for large-scale automata processing on GPUs. However, our detailed characterization reveals that three challenges have not been systematically addressed:

- **Challenge #1: GPU Threads Underutilization**. Prior works program GPU to process one single symbol from the input stream at a time. Consequently, numerous threads that do not have matched states finish earlier and wait on the cross-symbol barrier until all GPU threads finish processing the current symbol (Figure 1 (a)). Improving GPU thread utilization is challenging when the GPU is only allowed to process one symbol at a time due to the insufficient parallelism offered by that symbol.
- **Challenge #2: Redundant Computations**. Due to the nature of the NFA workloads, the always-active states need to be processed for every symbol from the input stream. This leads to repeated computations for matching symbols with the next states, resulting in the inefficient utilization of GPU cycles.
- **Challenge #3: Poor Data Locality**. In each iteration, GPU threads access items from a shared worklist and push the newly generated computations back. However, the dynamically changing mapping between threads and data impairs data locality.

To this end, we propose Non-blocking Automata Processing (*ng*AP in short) on GPUs. In contrast to the prior works that process one symbol at a time, *ng*AP allows states matched with different symbols to be processed in parallel and hence improves thread utilization (Figure 1 (b)). More importantly, *ng*AP provides support for more optimizations to address these challenges systematically: To address **Challenge #1**, we prefetch the matches between symbols and always-active states to the worklist, enabling concurrent processing of more symbols. This results in improved thread utilization due to the availability of more parallelism. Second, we address **Challenge #2** by designing a memoization mechanism for a portion of the states in which a sequence of matching operations can be converted by a table look-up. We extend the memoization table to support prefixes of patterns, further reducing redundant computations. Third, to address **Challenge #3**, a thread can selectively privatize the newly generated computations by preserving the data in registers instead of writing it back to the shared worklist, thus improving data locality. Overall, a detailed evaluation demonstrates the impact of each optimization and shows that combined optimizations significantly improve the throughput of automata processing on GPUs.

To the best of our knowledge, this is the first work that reduces the scope of barrier synchronization, allowing many symbols to be processed in parallel. The non-blocking processing enables new opportunities for unexplored optimizations to address challenges in automata processing on GPUs. Our contributions can be summarized as follows:

- We characterize and identify three major challenges – thread underutilization, redundant computations, and poor data locality – that limit the throughput of automata processing on GPUs, but remain unsolved in prior works.
- We propose a new approach, *ng*AP, that allows matches between different states and symbols to be processed in parallel, enabling us to propose three new optimizations that address the identified challenges synergistically.
- Our detailed evaluation demonstrates that the proposed approach and optimizations achieve a 7.9× geometric mean speedup across a wide range of 20 applications compared to the state-of-the-art GPU automata processing engine. Moreover, our approach outperforms HyperScan, the advanced CPU automata processing engine, by 11.9×.



Iter.	Sym.	Active States	Matched States	Reporting States
0	b	S0, S1	S1	
1	a	S0, S1, S2, S3, S6	S1, S2	
2	n	S0, S1, S2, S3, S4, S6	S0, S1, S3, S4, S6	S6
3	a	S0, S1, S2, S3, S5, S6	S1, S2, S5	
4	n	S0, S1, S2, S3, S4, S6	S0, S1, S3, S4, S6	S6
5	a	S0, S1, S2, S3, S5, S6	S1, S2, S5	

(a) NFA **(b)** The matching process of the NFA against the input stream banana.

Figure 2. An NFA example of pattern $/n*.+(na|an)?n/$. "$n*$" represents zero or more repetitions of "n"; ".+" matches any character one or more times; "$(na|an)?$" matches "na", "an" or nothing; "n" matches a character "n".

2 Background

2.1 Non-deterministic Finite Automata

A Non-deterministic Finite Automaton (NFA) is defined as a quintuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is the alphabet defined by the NFA, q_0 is a set of *starting states*, and F is a set of *reporting states*. Transition function $\delta(S, \alpha)$ defines the set of states to be activated when a set of state S matches with input symbol α, where $S \subseteq Q$ and $\alpha \in \Sigma$. As used in ANML [1] or MNRL [12] format, this work focuses on homogeneous NFAs where each state has valid incoming transitions for only one input symbol, which could be converted from classical NFA representations by Glushkov construction [25]. An NFA processing application consists of multiple homogeneous NFAs, where each NFA represents a pattern that the application searches for in the input streams.

Starting States. The matching process begins by activating the starting states. An NFA processing application can have two types of starting states: "all-input" states and "start-of-data" states, defined in ANML or MNRL format. The "all-input" starting states are used in applications that search for a pattern throughout the input stream regardless of the starting location of the pattern. For example, the pattern "`/apple/`" searches for all occurrences of the word "`apple`" in the input stream. The "all-input" starting states are always active, so we refer to them as always-active states in this work. In contrast, if the application is only interested in the pattern that appears at the beginning of the input stream (e.g., "`/^apple/`"), the "start-of-data" starting states are only active at the beginning of the input stream.

NFA Example. Figure 2 (a) illustrates an NFA that accepts regular expression $/n*.+(na|an)?n/$, in which each node is a state and each edge is a state transition. Each state has a *matchset*[1] of symbols in the alphabet that the state accepts. The states in hexagon (S_0, S_1) denote *always-active starting states* while the state (S_6) in double-circle denotes a *reporting state*. Initially, only the starting states are *active*. In each iteration, a symbol of the input stream matches with the active states. When the matchset of an active state contains the incoming symbol, the state becomes *matched* and then activates its neighbors. Figure 2 (b) illustrates the matching process of this NFA under the input stream "banana". For example, when the incoming symbol is "b", S_0 and S_1 are active in iteration 0. Since S_1 accepts "b", S_1 becomes *matched*, and then activates its neighbors S_2, S_3 and S_6 in the next iteration. As the starting states (S_0, S_1) are always active, they are also active in iteration 1. We will use this NFA as examples for illustration purposes in the following sections.

[1]The symbol "$*$" in NFA's matchset means the state can accept any symbol.

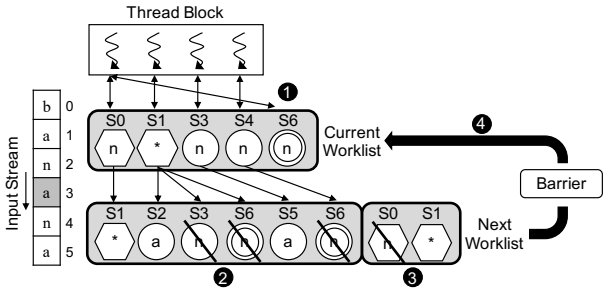


Figure 3. Illustrating the blocking automata processing (BAP) on GPUs corresponding to iteration 3 of Figure 2.

2.2 Processing Automata on GPUs

An NFA application has multiple levels of parallelism [34]. For instance, many input streams (e.g., network packets) and NFAs (e.g., network intrusion signatures) can be processed concurrently. Further, multiple states could be active at the same time. Plenty of parallelism makes NFAs a good fit for GPUs.

Prior works that process NFAs on GPUs can be categorized into *topology-driven* and *data-driven* approaches. Topology-driven approaches [20, 71, 72] statically map GPU threads to NFA states or transitions. However, these approaches tend to underutilize GPU threads when states or transitions are idle, leading to lower throughput compared to data-driven approaches. In contrast, to alleviate the underutilization of GPUs, state-of-the-art approaches often adopt variants of data-driven approaches [34, 35, 77], which maintain double-buffered worklists only for the active or matched states. Next, we will introduce the basic idea of data-driven approaches.

Illustrative Example. Figure 3 depicts the matching process in the data-driven approach, where two worklists containing matched states – the *current worklist* and the *next worklist* – are double-buffered. During each iteration, each thread is assigned to one or more states in the worklist (❶). These matched states in the current worklist (S_0, S_1, S_3, S_4, S_6) activate their neighbors ($S_1, S_2, S_3, S_6, S_5, S_6$). Then, the activated neighbors match with the incoming symbol ("a"). Only

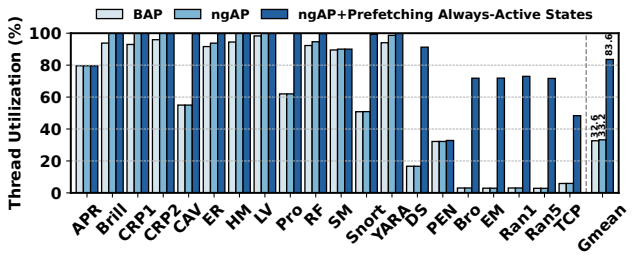


Figure 4. Thread utilization across the evaluated applications. Evaluation methodology is shown in § 5. "BAP" stands for "blocking automata processing", the double-buffered worklist approach. The other two bars show proposed schemes discussed in § 4.1 and § 4.2.

the matched states (S_1, S_2, S_5) are pushed into the next worklist (❷). A report is generated if a reporting state is matched (none in this case). Since S_0 and S_1 are always-active states, we match them with the incoming symbol and only push the matched states (S_1) into the *next worklist* (❸). At the end of the step, the next worklist is assigned to the current worklist while the current worklist is emptied (❹). In other words, every state that shares the same worklist must wait on a barrier until all the states in the current worklist are processed, making the process "*blocking*". This matching process consumes a symbol at each iteration (i.e. "one-symbol-at-a-time") until all symbols in the input stream are processed. We refer to this approach as "BAP" (blocking automata processing).

3 Motivation

3.1 Challenge #1: GPU Threads Underutilization

This section shows factors that can affect GPU thread utilization. We first discuss the definition of thread utilization, followed by an example showing why input can affect it.

Thread Utilization. We define *thread utilization* as the ratio of busy threads count to the total number of threads. If there is no work available to be picked up by GPU threads, they are considered idle. To illustrate this concept, we use an example of matching the input stream `banana` with the NFA in Figure 2. Figure 8 compares the thread utilization between BAP and our optimizations, which will be discussed later. To process the fourth symbol "`a`", Figure 8 (a) indicates that four threads are mapped to the first four states in the worklist (t_1), while the first thread works on the fifth state in the second round (t_2). Consequently, in the second round, all threads except the first thread remain idle, resulting in poor thread utilization (i.e. *thread utilization* = 5/8). When processing the next symbol "`n`", at time t_3, three states in the worklist are assigned to four threads, leaving one thread idle, thus the *thread utilization* is 3/4. Figure 4 depicts the measured thread utilization across the evaluated applications. We observed an average thread utilization of only 32.6% in blocking automata processing (shown in the "BAP" bar). Furthermore, a few applications have a thread utilization of less than 3%.

Insufficient Parallelism. The states in the worklist run in parallel. However, the length of the worklist at any given iteration depends on the number of matched states, which may not be enough to utilize all the threads. This issue is also highlighted by previous research showing only a small fraction of states are active for most of the time [33, 34, 58]. Although decreasing the number of threads allocated to the worklist to improve utilization is possible, it would result in serialized execution and poor performance. Thus, only leveraging the parallelism provided by one symbol is often insufficient, leading to thread underutilization.

Summary. We conclude that when the worklist does not have enough states, the GPU threads are underutilized. One key reason is that the threads can only work on states matched at the *same iteration* of the input stream as a barrier must be performed by the end of each iteration. Otherwise, if threads could proceed without per-symbol barriers, as Figure 1 (b), thread utilization could be improved significantly.

3.2 Challenge #2: Redundant Computations

This section first analyzes the redundant matches associated with always-active states, and then investigates the potential to reduce the work by not computing these matches.

Due to the nature of the matching process of automata, when matching a given string, a constant set of active states will transition to another predetermined set of active states. For example, a set of active states `{S2, S3}` in the NFA of Figure 2 (a) will transition to `{S6}` when the input string is "`an`". When the combination of active states and string recurs during the matching process, computations become redundant. In practice, this happens often: Consider the matching process depicted in Figure 3 (❸), where the always-active states match against the incoming symbols, and the matched elements are pushed into the subsequent worklist. During each iteration, the identical set of always-active states matches against the incoming symbol. Since each application has an alphabet Σ, there are only $|\Sigma|$ combinations of input symbols and the always-active states. As a result, it becomes unnecessary to repeatedly match the always-active states and the incoming symbol in order to obtain the matching results. Instead, these matching results can be stored for future reference. It is important to note that only 2 out of 20 evaluated applications (`APR` and `SM`; see Table 2) do not have always-active states. Consequently, most applications face the challenge of redundant computations.

Potential to Convert the Matches to Table Lookups. We investigate how much work can be eliminated by converting redundant matches to table lookups. Given that always-active states match with every symbol in the input stream, we associate these matches to individual symbols. To achieve

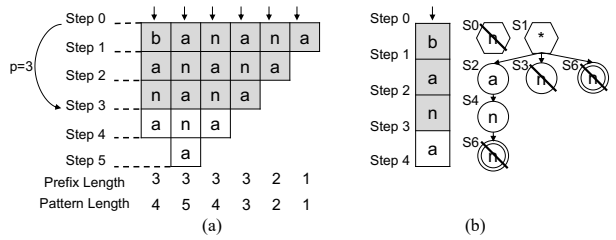


Figure 5. Measuring the potential of omitting the computations of pattern prefixes. (a) We measure the ratio of the number of symbols in *pattern prefixes* of length $p = 3$ to the total number of matched symbols in all patterns ($\frac{3+3+3+3+2+1}{4+5+4+3+2+1} = 0.79$). A larger ratio indicates more work is associated with always-active states and subsequent p steps. (b) Illustrating how the first pattern "**ban**a" is identified.

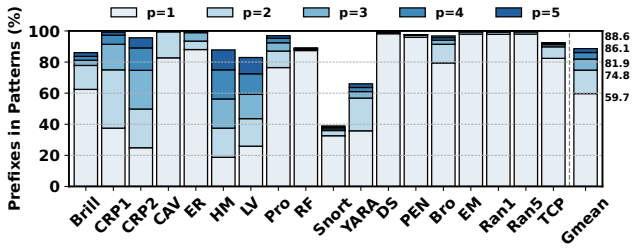


Figure 6. The percentage of work could be eliminated by not computing prefixes varying the length p from one to five associated with the always-active states. The applications without always-active states are excluded.

this, we define a *pattern* to be a sequence of symbols beginning with a match against always-active states and extending until no state stays active. Hence, in Figure 5 (a), we demonstrate the *patterns* vertically and place them along with the symbol at which they start. For example, searching for the NFA depicted in Figure 2 (a) from the first symbol in the input stream banana results in the matching process demonstrated in Figure 5 (b), where **ban**a was identified. When the matching results of a pattern *prefix* and the always-active states are stored, we can convert the matches to table lookups. Figure 5 (a) depicts the pattern prefixes of length $p = 3$ in shaded boxes. By using *the total number of matched symbols* as *an indicator to measure the total amount of work*, storing matching outcomes for all prefixes of length 3 removes a considerable portion of computations (79%), as evidenced by this illustration.

Results. We further measure the percentage of work that could be eliminated by storing the matching results of pattern prefixes and always-active states when varying the prefix length p from 1 to 5. Figure 6 displays that eliminating matches associated with all prefixes of length 1 to 5 reduces the total work by 59.7% to 88.6%, on average across the evaluated applications. Notably, this reduction is more pronounced for p values ranging from 1 to 3 (from 59.7% to 81.9%) than for those from 3 to 5 (81.9% to 88.6%).

Summary. Our analysis highlights that transforming pattern prefixes and always-active states matches into table lookups can substantially diminish the workload.

3.3 Challenge #3: Poor Data Locality

In this section, we show why prior data-driven designs lead to poor data locality.

Threads and States Mapping Switches Frequently. As discussed in Section 2.2, in the blocking automata processing (BAP), threads store the matched neighbors of states in the *current worklist* to the *next worklist*. Then, the states in the *next worklist* are evenly assigned to the threads. This approach redistributes the newly generated work for each iteration and hence ensures load balance.

Analysis of Poor Locality. However, the remapping between threads and states happens frequently, leading to poor locality. In each iteration, a thread loads a different state and then fetches its neighbors. For example, in an iteration, thread t_i is mapped to S_i. To match a symbol, thread t_i loads the data structures of S_i and S_i's neighbors from memory. Suppose S_j is a state of S_i's neighbors. At this time, S_j must be in the registers of thread t_i's context and cache. However, thread t_i pushes S_j into the *next worklist*, and then mapped to another state in the next iteration, resulting in a loss of data locality. Similarly, as S_j may be also mapped to a thread other than t_i in the next iteration, we are unsure whether S_j is still in the cache or has been evicted due to limited cache size (especially, L1 cache is small in GPUs).

Summary. Overall, prior approaches do not exploit the temporal locality in the matching process, potentially resulting in suboptimal performance. Therefore, if we could preserve the mapping between threads and data for a longer duration, the data locality would be improved.

4 ngAP: Non-blocking Automata Processing

We propose *ng*AP, Non-blocking Automata Processing, that allows threads to work on *different* symbols in parallel. Our key insight is that the scope of synchronization can be reduced to a single state, eliminating the need to process the input stream one symbol at a time. This *non-blocking* approach leverages parallelism across different symbols of the input stream, making the matching process more efficient. This section first discusses the design of *ng*AP (§ 4.1), followed by new optimizations enabled by *ng*AP to systematically address the three challenges in § 4.2, § 4.3, and § 4.4.

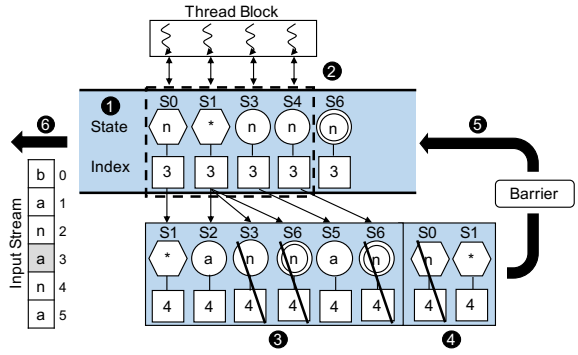


Figure 7. The basic execution flow of *ng*AP.

4.1 Design of Non-blocking Automata Processing

Worklist of State-Index Pairs. The basic execution flow of *ng*AP is presented in Figure 7. In contrast to prior works, we use a single worklist instance for all iterations instead of double-buffered worklists. In BAP, all states in the worklist process the same symbol index, thus synchronization across all states for the next iteration is needed. However, if each state in the worklist is aware of the index of the input symbol it needs to match with, the matching process can continue by matching the state's neighbors and the symbol at the corresponding index. Therefore, we represent each element of the worklist as a pair that contains the symbol index and the matched state (i.e., state-index pair). Figure 7 (❶) illustrates the worklist of state-index pairs. As a result, this worklist allows any indices of the input stream to be included, and thus can be processed in parallel.

State Transition. A sliding window (dashed rectangle) of the pairs is mapped to the threads in each iteration (❷), managed by two pointers `head` and `tail` that point to the range of elements in the worklist. In each iteration, the threads fetch a sliding window of state-index pairs from the worklist, which is done by updating the two pointers atomically. In Figure 7, the second thread is mapped to a pair $(S_1, 3)$. It loads the index 3 from the input stream ("a") and then matches "a" with S_1's neighbors (❸). As S_2 accepts "a", the pair $(S_2, 4)$, where 4 is the next symbol index, will be pushed into the worklist (❺).

Handling Always-Active States. Besides the states in the sliding window, the threads must process the always-active states. When the threads start to process a new index in the input stream that was not processed (in this example, index 3), they match the always-active states with the symbol at the new index. The matched always-active states and the next index are paired $(S_1, 4)$ (❹), and will be pushed into the worklist (❺). Finally, in the next iteration, the threads are mapped to new state-index pairs by updating the sliding window (❻).

Summary. *ng*AP releases the restriction on the synchronization scope from all states in the application to the states within a sliding window. Further, we discuss how *ng*AP provides support for further solutions to the three challenges.

4.2 Optimization #1: Enhancing Thread Utilization via Prefetching Always-Active States

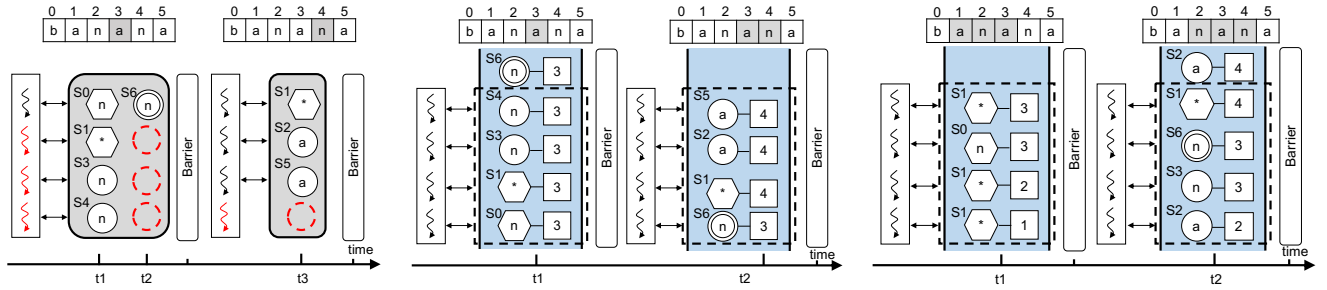
In this section, we first characterize how *ng*AP improves thread utilization, then propose a new optimization that works with *ng*AP to address **Challenge #1**.

Thread Utilization Revisited. In comparison to Figure 8 (a) where the thread utilization is only 8/12, *ng*AP exhibits an improved thread utilization, as depicted in Figure 8 (b), with a shift from 8/12 to 1. Although *ng*AP enables processing of different symbols simultaneously, we have noticed that execution is serial at the state level, and the worklist is scheduled in a first-come-first-serve order, leading to limited parallelism improvement since the indices co-existing are not many in the worklist. Figure 4 shows that *ng*AP only slightly improves thread utilization, increasing it from 32.6% to 33.2%.

Prefetching Always-Active States. To further address **Challenge #1**, we leverage the parallelism that arises from processing different symbols, and maximize the opportunity to process different symbols. As discussed in Section 2, always-active states must match with every symbol in the input stream. Therefore, we propose to prefetch matches between each symbol and always-active states to the worklist in batches. At the start of execution, the threads load a batch of symbols and match them with always-active states. For instance, with a batch size of 3 ($b = 3$), symbols with indices 0 to 2 match with the always-active states, and the resulting state-index pairs are added to the worklist. The iteration t_1 of Figure 8 (c) depicts the content of the worklist after loading the first batch. As a result, symbols at indices 1,2,3,3 are matched with the neighbors of S_1, S_1, S_0, S_1, respectively. In the following iteration t_2, since index 3 hasn't matched with always-active states, a batch of matching results between symbols (indices 3 to 5) and always-active states are loaded to the worklist (omitted from this figure).

Thread Utilization Improvement. Compared with Figure 8 (b), more symbols are processed in each iteration with *Prefetching Always-Active States*, thereby improving thread utilization considerably. According to Figure 4, the addition of Prefetching Always-Active States (batch size $b = 256$) to *ng*AP results in a substantial improvement in thread utilization across the evaluated applications, increasing it from an average of 33.2% to 83.6%. We will discuss how we determine the batch size in Section 5.

Summary. We propose *ng*AP to solve **Challenge #1**, enabling parallel processing of symbols. By prefetching the matches between always-active states and symbols to the



(a) Thread underutilization in BAP: thread utilization = 8/12.

(b) Improved thread utilization by *ng*AP.

(c) Prefetching Always-Active States boosts parallelism from multiple symbols.

Figure 8. Techniques to improve thread utilization compared to BAP. (a) BAP underutilizes the threads as it processes one symbol at a time. (b) *ng*AP improves thread utilization by exploiting parallelism arising from processing multiple symbols simultaneously. (c) *Prefetching Always-Active States* further improves thread utilization by enabling more opportunities for the worklist to have different symbol indices.

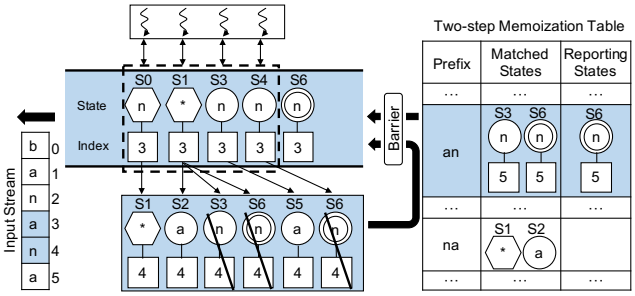


Figure 9. Memoization table for pattern prefixes of length 2.

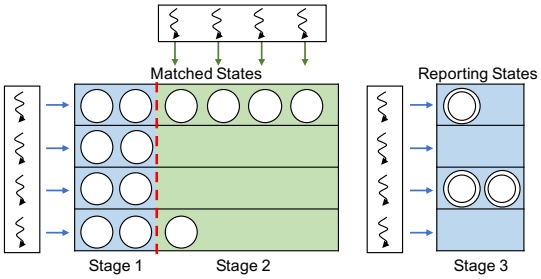


Figure 10. Balancing workload when loading results for multiple prefixes from the memoization table.

worklist, we increase the number of indices coexisting in the worklist. As a result, our approach improves thread utilization significantly.

4.3 Optimization #2: Reducing Redundant Work via Prefix Memoization

To address **Challenge #2**, we introduce Prefix Memoization in this section. Prefix Memoization substitutes matches between always-active states and pattern prefixes with table look-ups, thereby reducing redundant computations.

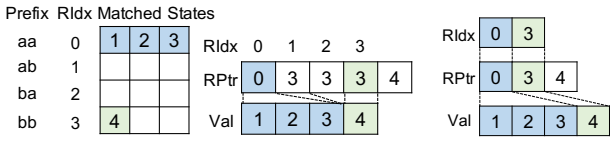
Memoization Table. The memoization table is computed offline since the number of possible combinations between always-active states and short prefixes with a length of p is finite. To construct the memoization table, we first enumerate all prefixes of length p according to the alphabet. Thus, if the alphabet size is $|\Sigma|$, the table needs $|\Sigma|^p$ entries to store all p-length prefixes. We match each prefix with the always-active states and their subsequent states, and store what states are matched when the prefix ends to each entry of the table. Since reports could be generated within p-length patterns, we also store the generated reports in table entries.

Illustrative Example. In Figure 9, an example of using a memoization table for pattern prefixes is illustrated. In the current iteration, we need to match index 3 with always-active states, and the memoization table records prefixes of length 2. Thus, we look up the table for the prefix composed by symbols at indices 3 and 4 ("an", shaded in the figure), and then add the entry to the worklist, along with other matched states at the end of this iteration. If the entry includes reporting states, reports are generated at the symbol indices accordingly. By substituting computations with table lookups, we can eliminate a significant amount of redundant computations since the same prefix can occur many times during execution.

Integrating with Prefetching Always-Active States. We can build Prefix Memoization on top of Prefetching Always-Active States (Section 4.2): We prefetch the matching results from the *memoization table* in batches. Threads load every p adjacent symbols from the input stream and look up the corresponding entries in the table. However, it's worth noting that the sizes of the matching results for prefixes and always-active states can vary significantly. As shown in Figure 10, the entries differ in size. Loading an entry using each thread results in a significant load imbalance, which impinges on thread utilization. To tackle this issue, we distribute the entries evenly across threads by loading

Table 1. The memory consumption and construction time of memoization table with the prefix of length 3 with/without compression. Applications without always-active states are excluded.

App.	Brill	CRP1	CRP2	CAV	ER
W/O Comp.	640.8 MB, 2.4 s	158.7 MB, 3.7 s	3.3 GB, 221.5 s	140.5 MB, 1.2 s	154.6 MB, 9.2 s
W Comp.	507.2 MB, 2.1 s	24.6 MB, 3.8 s	3.1 GB, 223.1 s	8.4 MB, 0.7 s	20.4 MB, 9.7 s
App.	HM	LV	Pro	RF	Snort
W/O Comp.	67.2 GB, 150.4 s	1.2 GB, 6.6 s	410.7 MB, 1.7 s	3.6 GB, 20.6 s	354.5 MB, 2.9 s
W Comp.	67.2 GB, 171.6 s	1.1 GB, 6.3 s	282.6 MB, 1.5 s	3.4 GB, 19.7 s	286.9 MB, 2.8 s
App.	YARA	DS	PEN	Bro	EM
W/O Comp.	5.4 GB, 23.7 s	134.3 MB, 0.4 s	134.2 MB, 0.3 s	134.2 MB, 0.3 s	134.2 MB, 0.3 s
W Comp.	5.3 GB, 26.6 s	128.3 KB, 0.1 s	29.5 KB, 0.1 s	6.0 KB, 0.1 s	1.9 KB, 0.1 s
App.	Ran1	Ran5	TCP		
W/O Comp.	134.2 MB, 0.3 s	134.2 MB, 0.3 s	134.8 MB, 0.3 s		
W Comp.	2.4 KB, 0.1 s	3.8 KB, 0.1 s	1.1 MB, 0.1 s		



(a) Uncompressed **(b)** Compressed

Figure 11. The comparison between uncompressed and compressed memoization table formats.

them in three stages. In the first stage, each thread loads the initial few states in each entry. The dashed line in Figure 10 represents a threshold, and all entries that exceed this threshold are fetched in the second stage. We set up the threshold to the average length of entries as it achieves the best performance empirically. During the second stage, all threads in the thread block collectively load the remaining states in the longer entries, distributing them equally among them. In the final stage, since reports are infrequent in the prefixes, each thread loads only the reports from the entry it needs.

Memory and Time Consumption. Table 1 shows the memory usage and construction time for a memoization table containing 3-length prefixes (p = 3), as indicated in the "W/O Comp" row. The memory space required for the table ranges from hundreds MBs to several GBs in various applications. We observe that most offline computations can finish in several seconds, however, a few applications (e.g. `CRP2` and `HM`) require minutes because they have more active states on average for the prefixes. Nevertheless, since these generated tables can serve for all input streams, the computation time could be amortized.

Prefix Length Selection. With p values ranging from 1 to 3, we observe increased throughput for all applications. Therefore, we use p = 3 for all applications except `YARA` and `HM`, which uses p = 2 due to exceeding the memory capacity of the GPUs for evaluation. In this work, we limit p to 3

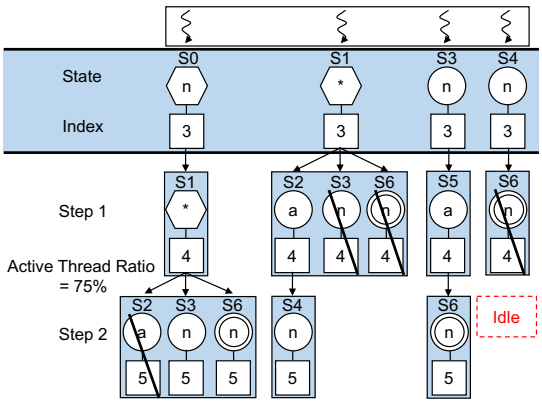


Figure 12. Divergence-aware Work Privatization.

for two reasons: 1) In Section 3.2, we demonstrate that the benefits of increasing the prefix length to 4 or more are less significant. 2) Computing a memoization table with $p \geq 4$ takes at least hours, and storing it is very expensive.

Memoization Table Compression. Since the table has entries with different sizes, we store it in a format similar to Compressed Sparse Rows (CSR, as shown in Figure 11 (a)). However, we found that many rows of the table, such as the row for `ab` and `ba` in this figure, are empty and take up space in the row pointer array (`RPtr`) of the CSR format, even though no values are stored in them. To mitigate this issue, we compress the memoization table by indexing it with the prefix *values*. Figure 11 (b) illustrates our design. The initial two rows contain the values for non-empty prefixes and their corresponding starting locations. Retrieving a prefix's entry requires a thread to perform a binary search on the first row (`RIdx`) and then load the row pointer to access the entry. Table 1 illustrates the impact of compression on the table. In general, compression substantially reduces the storage space required for memoization tables, and takes similar amount of time in preparing them. For instance, certain applications (e.g., `Bro, Ran5, PEN, CAV`) may only need up to a few MBs, as opposed to several hundred MBs without compression. We observe the throughput is also improved by 5.3% on average due to reduced memory footprint.

4.4 Optimization #3: Improving Data Locality via Work Privatization

In this section, we propose *Work Privatization* to improve the data locality. The key reason behind **Challenge #3** is that each thread pushes the state-index pairs produced by it to the shared worklist. However, the consumer thread of the state-index pairs is dynamically mapped. This loses data locality and requires more writes and loads on the GPU memory.

Work Privatization. Figure 12 shows how Work Privatization works. The threads extend to the neighbors of states

Table 2. Characteristics of evaluated applications. CC stands for "connected component".

Suite	Idx	Application	Abbr.	#states	#always-active	#start	#report	#CC	max CC size	avg CC size
AutomataZoo [67]	1	APPRNG4	APR	20000	0	1000	4000	1000	20	20.0
	2	Brill	Brill	115549	5946	0	5946	5946	40	19.4
	3	CRISPR_CasOFFinder	CRP1	74000	4000	0	2000	2000	37	37.0
	4	CRISPR_CasOT	CRP2	202000	4000	0	2000	2000	101	101.0
	5	ClamAV	CAV	2374717	33171	0	33667	33171	22075	71.6
	6	EntityResolution	ER	413352	10000	0	10000	10000	75	41.3
	7	Hamming_N1000_l18_d3	HM	108000	2000	0	2000	1000	108	108.0
	8	Levenshtein_l19d3	LV	109000	4000	0	4000	1000	109	109.0
	9	Protomata	Pro	24103	1302	8	1321	1309	123	18.4
	10	RandomForest_20_400_200	RF	992000	16000	0	16000	16000	62	62.0
	11	SeqMatch_BIBLE_w6_p6	SM	51570	0	10314	1719	1719	30	30.0
	12	Snort	Snort	202043	2507	644	3246	2486	4509	81.3
	13	YARA	YARA	1047528	23537	2	23583	23530	1017	44.5
ANMLZoo [65]	14	Dotstar	DS	96438	2837	0	2838	2837	95	34.0
	15	PowerEN	PEN	40513	2857	0	3456	2857	52	14.2
Regex [17]	16	Bro217	Bro	2312	187	0	187	187	84	12.4
	17	ExactMatch	EM	12439	297	0	297	297	87	41.9
	18	Ranges1	Ran1	12464	297	0	297	297	96	42.0
	19	Ranges05	Ran5	12621	299	0	299	299	94	42.2
	20	TCP	TCP	19704	756	0	767	738	391	26.7

that are mapped to them, and then check whether the neighbors match with the symbols in the corresponding indices (**Step 1**). With *Work Privatization*, each thread can decide whether it privatizes the extended neighbors without writing them back to the shared worklist. As shown in **Step 2**, the thread could further compute ahead without interacting with the shared worklist. This improves temporal locality at the expense of parallelism, as the extended states are processed sequentially.

Controlling Warp Divergence and Work Serialization. First, A thread must decide whether it privatizes its computations, or terminates the privatization by pushing the results back to the worklist. However, on a GPU, threads within the same warp must run in lockstep [24]. If a few threads within a warp choose to privatize their computations while others do not or cannot due to mismatches, warp divergence impairs thread utilization. To address it, each warp calculates an *active thread ratio* for the next step, depending on whether its threads have more work to do. For example, assuming the warp size is 4, in **step 1** of Figure 12, only 3/4 threads have more computations, thus the active thread ratio is 3/4. To control the warp divergence, we set up a threshold d: When *active thread ratio* $\leq d$, the threads terminate privatization as thread utilization will be low if continues. If $3/4 < d$, all threads within the same warp terminate privatization; otherwise, the threads could step forward. Essentially, $d = 100\%$ indicates we disable Work Privatization while $d = 0\%$ indicates a thread always privatizes its work. Second, a thread needs to limit the number of steps to privatize the computations because privatization for many steps accumulates the works of a thread, serializing the computations. To control the serialization, we limit the maximum number of steps (K). We empirically observe that when $K > 1$, the throughput degrades drastically due to work serialization, thus we set $K = 1$ in this scheme while leaving the divergence threshold d as a tunable parameter.

Table 3. Evaluated schemes. *ng*AP's parameters: sliding window (s), batch size (b), prefix length (p), and divergence threshold (d).

Scheme	Description
HyperScan [68]	State-of-the-art automata processing engine on CPUs
NFA-CG [77]	Calculating compatible groups and mapping to threads.
AsyncAP [35]	Matching patterns asynchronously in the input stream.
GPU-NFA [34]	Flexibly scheduling hot and cold states to threads.
*ng*AP-DEFAULT	Our proposed design with default parameters. ($s = 25600, b = adaptive, p = 3, d = 0\%$)
*ng*AP-BEST	Our proposed design with tuned parameters. ($s \in \{256, 5120, 10240, 25600\}$, $b \in \{32, 64, 128, 256, 512, 1024, 2048, 4096, 1000000, adaptive\}$, $p = 3$, $d \in \{0\%, 25\%, 50\%, 100\%\}$)

Discussion. Although our proposed optimizations tackle different challenges, they share a common requirement: the worklist must be able to handle various indices of symbols. Consequently, these optimizations cannot be applied to prior work, and they must be built on top of *ng*AP. We will show that these optimizations can work synergistically to address three challenges effectively in § 6.

5 Evaluation Methodology

System Configuration. We perform most experiments on a computer with an NVIDIA RTX 3090 (Ampere architecture, 24 GB memory, 6 MB L2 cache, and 82 SMs). The system runs Linux on a 12-core Intel Xeon 4214R CPU and 128 GB memory. We use NVIDIA NSight Compute to profile the kernels. All CUDA/C++ programs were compiled with `-O3` flag with GCC 9.5 and CUDA 12.0. We use an NVIDIA Tesla V100 (Volta architecture, 32 GB memory, 6 MB L2 cache, and 80

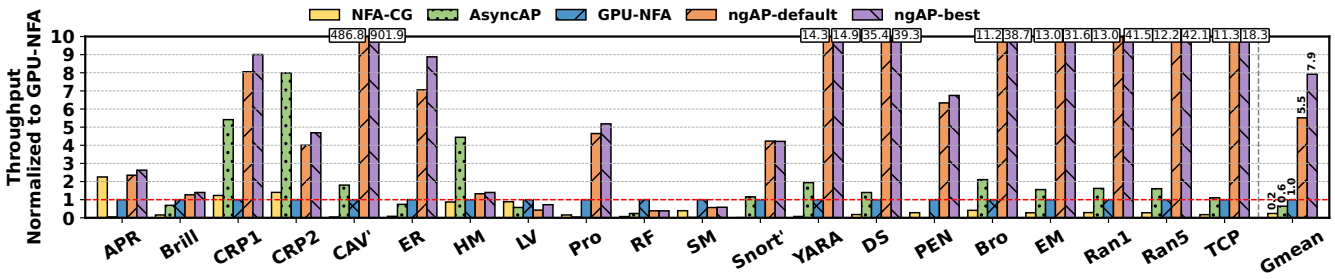


Figure 13. Throughput results normalized to GPU-NFA.

SMs) to evaluate the performance sensitivity. The throughput is defined as the number of input symbols processed per second. The execution time is averaged across three repeated runs excluding automata loading and data preparation, as they can be amortized with long input streams.

Benchmarks. We evaluate a wide range of applications from AutomataZoo [67], ANMLZoo [65], and Regex [17]. Only two were selected from ANMLZoo as most of its applications were updated in AutomataZoo. Table 2 shows the characteristics of the evaluated applications. To simulate a real-world scenario, we use a 1 MB input stream equipped with each application in the benchmark suites and then generate 600 copies as the input. In order to reduce the duration of experiments, for the applications that cannot finish within an hour, we use a throughput of 0.16 MB/s (i.e., 600MB/3,600s) to estimate its upper bound. To validate our implementations, we use a serial version of automata processing on the CPU as a reference and verify that the generated reports, which include the reporting states and corresponding symbol indices, are identical.

Evaluated Schemes. Table 3 summarizes the evaluated schemes. For GPU work, we evaluate three prior data-driven designs: NFA-CG [77], GPU-NFA [34], and AsyncAP [35]. NFA-CG and GPU-NFA are variants of BAP. AsyncAP increases the parallelism by starting matching from different input locations in the input stream. HyperScan [68] is the state-of-the-art CPU automata processing engine, combining many optimizations. We use MNCaRT [11] and VASim [66] to convert the automata for HyperScan. *ng*AP-DEFAULT and *ng*AP-BEST are our schemes with different parameter setups.

Parameter Setup. Table 3 also shows the tunable parameters: sliding window size s (§ 4.1), batch size b (§ 4.2), and divergence threshold d (§ 4.4). *ng*AP-DEFAULT uses a set of parameters that work well for all applications, while *ng*AP-BEST tunes the best parameter combination for each application. We propose an *adaptive* scheme for batch size b, as we observe a larger value of it has better performance but may overflow the worklist. When *Prefix Memoization* and *Prefetching Always-Active States* are enabled, we estimate the number of state-index pairs to be included in the worklist as the entry with a maximum number of states for prefixes to be loaded (e). Thus, when a batch is loaded, the batch size is set to the remaining spaces of the worklist divided by e.

Table 4. Absolute throughput for evaluated applications in MB/s. T: Timeout (time limit: 1 hour). U: Unsupported applications.

App	HyperScan	NFA-CG	AsyncAP	GPU-NFA	ngAP-default	ngAP-best
APR	T	8.0	U	3.6	8.4	**9.4**
Brill	1.2	1.1	4.7	6.8	8.6	**9.5**
CRP1	0.3	5.1	22.3	4.1	33.1	**37.0**
CRP2	T	2.5	**14.5**	1.8	7.3	8.5
CAV	327.0	U	6.6	U	2334.8	**5596.9**
ER	1.6	0.7	6.8	9.1	64.3	**80.9**
HM	T	4.9	**25.1**	5.6	7.5	7.9
LV	T	1.1	0.7	**1.2**	0.5	0.9
Pro	1.9	4.8	1.3	30.5	141.7	**158.0**
RF	0.4	0.3	1.1	**4.8**	1.8	1.8
SM	2.9	3.1	U	**8.0**	4.5	4.6
Snort	36.6	U	20.2	U	63.8	**114.9**
YARA	**65.13**	0.3	6.9	3.5	50.4	52.8
DS	196.0	7.3	57.3	41.1	1452.6	**1612.1**
PEN	214.2	9.3	T	33.0	208.9	**222.5**
Bro	895.8	143.2	732.3	347.7	3897.3	**13452.8**
EM	3693.0	78.5	440.6	283.2	3690.2	**8947.4**
Ran1	485.1	79.6	448.1	276.4	3598.9	**11461.3**
Ran5	641.5	77.6	447.9	278.9	3405.5	**11743.4**
TCP	148.9	31.3	198.5	179.8	2037.3	**3298.0**

6 Experimental Results

6.1 Throughput

The normalized throughput of the evaluated applications is shown in Figure 13. It should be noted that GPU-NFA and NFA-CG transform NFAs to restrict each node from having an out-degree fewer than 4. However, a few NFAs cannot be transformed, we excluded 2 NFAs from `CAV` and 5 NFAs from `Snort` and refer to them as `CAV'` and `Snort'`, respectively. In contrast, our design utilizes a compressed sparse rows (CSR) format for NFA topology, eliminating the need for NFA transformation and avoiding this limitation. The absolute throughput of full applications is in Table 4.

Overall Improvement. As shown in Figure 13, on average, *ng*AP-DEFAULT and *ng*AP-BEST significantly outperform other designs. Specifically, *ng*AP-BEST is 39.5×, 13.2×, and

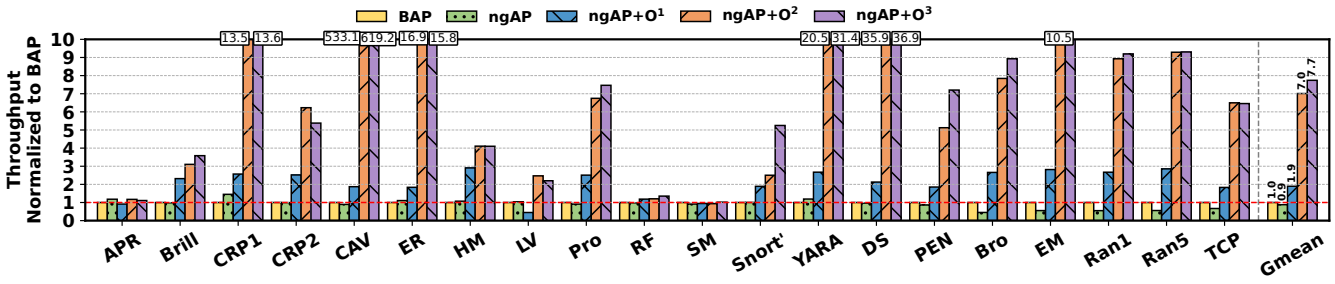
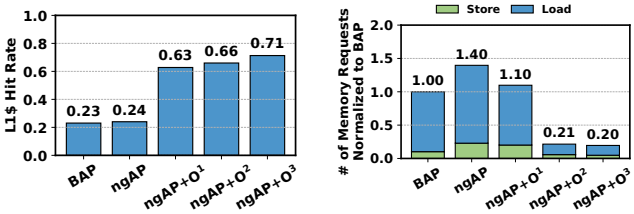


Figure 14. Throughput results normalized to BAP for breakdown analysis. To demonstrate the impact of proposed optimizations, we evaluate them individually by adding them one at a time.

Table 5. Breaking down *ng*AP to evaluate effect of each optimization.

Abbr.	Solutions			
	Non-blocking (s = 25600)	Prefetching Always-Active States (b = 256)	Prefix Memoization (p = 3)	Work Privatization (d = 0% or 50%)
BAP				
*ng*AP	✓			
*ng*AP+O^1	✓	✓		
*ng*AP+O^2	✓	✓	✓	
*ng*AP+O^3	✓	✓	✓	✓



(a) L1 Cache **(b)** Memory Traffic

Figure 15. Average profiling results for evaluated applications.

7.9× faster than NFA-CG, AsyncAP, and GPU-NFA, respectively. *ng*AP-DEFAULT also improves throughput over prior schemes by 27.5×, 9.2×, and 5.5× without parameter tuning.

Application Analysis. Compared to GPU-NFA, *ng*AP-DEFAULT and *ng*AP-BEST achieve maximal speedups of 486.6× and 901.9× for CAV'. 8 out of 15 applications experience more than 10× speedup. Our schemes improve performance in these applications for two reasons: 1) Some applications have imbalanced worklists, resulting in low thread utilization. Applications with few states, such as DS, BRO, and EM, lack parallelism to fully utilize threads when processing one symbol at a time. 2) These applications have short patterns, allowing the memoization table to eliminate most computations during prefix lookup. Our schemes synergistically address thread utilization and redundant work problems, resulting in significant speedup.

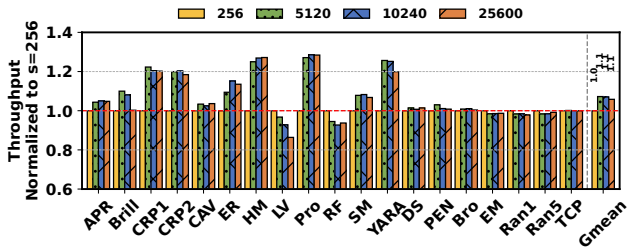
Certain applications do not have speedup due to the absence of always-active states (SM) or the memoization table covering only a small percentage of work (LV, RF, HM). Our schemes are slower than AsyncAP for a few applications (CRP2 and HM) because when an application's patterns are balanced, AsyncAP has less overhead, but it performs poorly in imbalanced scenarios (e.g., PEN). Overall, our scheme achieves significant speedup for most evaluated applications.

Comparison with HyperScan. Table 4 shows HyperScan throughput, which is faster than prior GPU work on some applications. However, overall, *ng*AP-BEST outperforms HyperScan, except for YARA. Notably, in intrusion detection application Snort that HyperScan focused on, *ng*AP-BEST achieves a 3.1× speedup. Finally, *ng*AP-DEFAULT and *ng*AP-BEST are 7.9× and 11.5× faster than HyperScan, respectively.

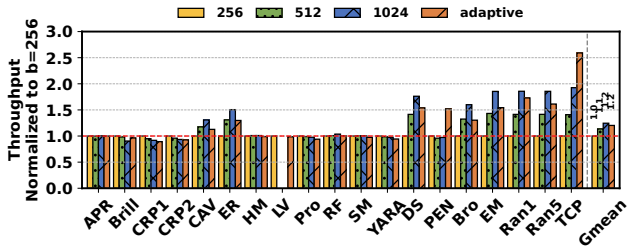
6.2 Breakdown Analysis

To understand the impact of proposed optimizations for *ng*AP, we evaluate them incrementally on top of a baseline blocking automata processing on GPUs ("BAP", discussed in § 2). Figure 14 shows the throughput normalized to BAP for each optimization by adding it on top of the former one (as listed in Table 5). Our observations are as follows: 1) On average, *ng*AP slightly decreases throughput by 10% compared to BAP due to increased overhead from maintaining the worklist of state-index pairs. 2) *ng*AP+O^1 achieves a 1.9× increase in throughput compared to BAP due to significant improvement in thread utilization from Prefetching Always-Active States. 3) Throughput is significantly improved by 7× with *ng*AP+O^2, indicating that Prefix Memoization effectively eliminates redundant computations. 4) *ng*AP+O^3 improves throughput to 7.7× compared to BAP, demonstrating the effectiveness of Work Privatization to improve the data locality.

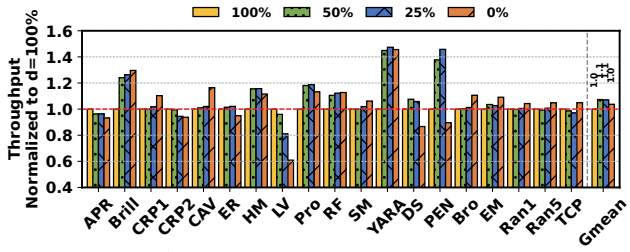
Profiling Results. We profiled cache and memory statistics to understand the reasons behind the improved throughput. Figure 15 (a) shows that all optimizations increase the L1 cache hit rate, with *ng*AP+O^1 showing the most significant improvement due to Prefetching Always-Active States. This



(a) Sliding Window Size (§ 4.1)



(b) Batch Size (§ 4.2)



(c) Divergence Threshold (§ 4.4)

Figure 16. Performance sensitivity to parameter tuning.

optimization gathers the same states with different symbol indices, allowing better cache utilization when accessing the NFA topology. *ng*AP+O^3 further increases the hit rate by 5% compared to *ng*AP+O^2 by reducing thread switching and allowing cache reuse. Although *ng*AP generates more memory requests due to additional symbol index information, other optimizations alleviate this problem. *ng*AP+O^2 significantly reduces store and load requests by removing redundant computations, as seen in Figure 15 (b).

6.3 Sensitivity Studies

To investigate the impact of each parameter on sensitivity, we have fixed the parameters as listed in Table 5 and then varied one parameter at a time. Then, we study whether our approaches work in other GPU architectures.

Sensitivity to Sliding Window Size. As discussed in Section 4.1, the sliding window size (s) can affect thread utilization. Figure 16 (a) demonstrates the performance sensitivity to sliding window size. We observe that the throughput is generally better but not very sensitive when $s \geq 5120$ compared with $s = 256$, because a larger sliding window size amortizes the overhead of updating the sliding window pointers.

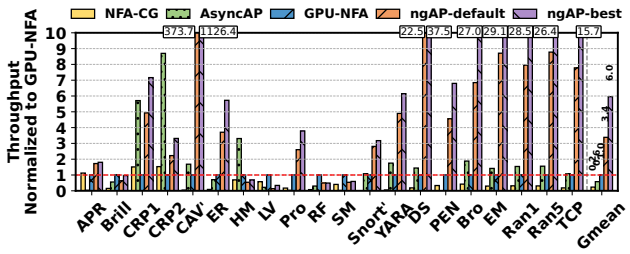


Figure 17. Throughput sensitivity to Volta architecture.

Sensitivity to Batch Size. Figure 16 (b) shows the sensitivity of throughput to the batch size. Larger batch sizes generally result in better performance due to improved parallelism, but if the batch size is too large, it may exceed the GPU memory capacity (e.g., $b = 512, 1024$ in LV). We observe that an adaptive batch size approach (discussed in §5) that considers the remaining memory and entries to be loaded, which often leads to acceptable performance.

Sensitivity to Divergence Threshold. Figure 16 (c) shows the sensitivity results of different divergence thresholds (d). We make the following observations: 1) Work Privatization ($d < 100\%$) generally benefits most applications, but can have a negative impact on a few. 2) The benefits of Work Privatization depend on the specific application, as the trade-off between work serialization and data locality improvement can vary. Enabling work privatization without considering divergence ($d = 0\%$) can cause a performance loss, as seen in the case of LV, which achieves only 60% throughput compared to disabling it ($d = 100\%$). Therefore, the divergence threshold is a tunable parameter for each application.

Sensitivity to Volta architecture. We conducted the experiment on an NVIDIA V100 GPU, where *ng*AP-DEFAULT uses the same parameters as those used for the 3090 GPU, and *ng*AP-BEST is tuned for the V100. The results are presented in Figure 17. Our observations indicate that *ng*AP-DEFAULT and *ng*AP-BEST achieve a speedup of 3.4× and 6.0× over GPU-NFA, respectively. The lower speedup of *ng*AP-DEFAULT suggests that the default parameters may not be portable enough for all GPU architectures. Nevertheless, *ng*AP still achieves a significant improvement on the V100 compared to prior works.

Discussion on GPU Architectures. To understand why GPU architectures result in different performance profiles for *ng*AP, which only requires integer operations, we first analyze their performance under the integer roofline model of 3090 and V100. As shown in Figure 18, NFA-CG, AsyncAP, and GPU-NFA are compute-bound, suggesting that their performance may benefit from increased integer capabilities.

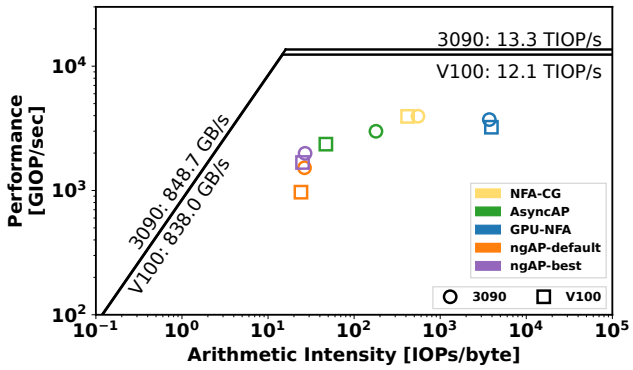


Figure 18. Roofline model focusing on Integer Operations Per Second (IOPS) for the RTX 3090 and V100 GPUs. We quantify the performance and arithmetic intensity of various schemes by calculating their geometric means across 14 different automata applications. Applications that do not run with any scheme or ncu profiler are excluded.

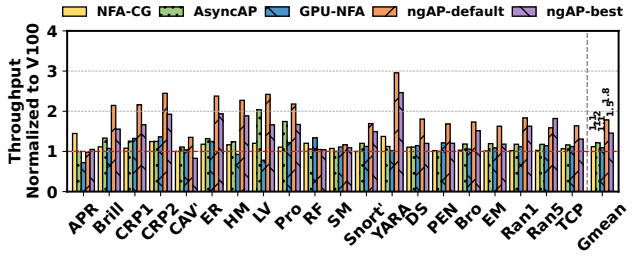


Figure 19. Performance improvement of evaluated schemes on 3090 compared to executing them on V100.

Figure 19 illustrates the speedup that each evaluated scheme could achieve on 3090 over V100. We observe that the performance of NFA-CG, AsyncAP, and GPU-NFA is 1.1×, 1.2×, and 1.1× over V100, respectively. This speedup can be attributed to the compute-bound nature of these schemes, as depicted in Figure 18, which aligns with the 3090's 1.1× peak integer performance compared to V100 (13.1 TIOPS vs. 12.0 TIOPS [3, 4]).

In contrast, *ng*AP-DEFAULT and *ng*AP-BEST are likely to be latency-bound (Figure 18), as they exhibit lower compute and memory efficiency. They achieve even higher performance improvements of 1.8× and 1.5× on 3090, respectively, compared to the peak integer performance differences between the 3090 and V100 GPUs (Figure 19), and this can be attributed to two key factors: First, 3090 has more registers per thread [3, 4], enabling more complex GPU kernels to have higher occupancy and hence hiding latency by fine-grained multithreading [5]. This is supported by the fact that both *ng*AP-BEST and *ng*AP-DEFAULT achieve an average *achieved occupancy* of 0.62 on 3090, compared to 0.48 and 0.36, respectively, on V100. Second, the 3090 operates at a higher SM frequency (13.9 GHz vs. 12.5 GHz), further reducing the latency, given that cache accesses require a similar number of cycles [8, 26].

Table 6. Absolute throughput (in MB/s) with one input stream for evaluated applications. U: Unsupported applications.

App	HyperScan	NFA-CG	AsyncAP	GPU-NFA	ngAP-default	ngAP-best
APR	0.03	1.28	U	**2.03**	0.56	0.56
Brill	1.19	0.98	2.78	0.49	2.44	**4.85**
CRP1	0.32	1.51	10.56	0.44	7.84	**22.34**
CRP2	0.16	1.16	**8.14**	0.23	4.22	6.28
CAV	**262.67**	U	1.05	U	13.71	38.92
ER	1.6	0.78	2.58	1.63	13.17	**36.63**
HM	0.04	1.37	**13.21**	0.27	4.27	6.46
LV	0.02	**0.96**	0.69	0.21	0.62	0.72
Pro	1.85	1.78	1.29	0.53	8.82	**13.65**
RF	0.4	0.39	0.67	1.06	5.72	**7.57**
SM	**2.89**	1.75	U	1.11	0.88	0.88
Snort	**31.54**	U	4.29	U	1.36	1.83
YARA	**61.31**	0.25	1.52	0.79	3.11	3.35
DS	**71.91**	1.37	10.46	1.11	15.08	45.04
PEN	**84.03**	1.58	0.08	1.08	1.11	1.16
Bro	**285.88**	1.69	39.06	1.49	18.78	111.66
EM	**1177.86**	1.66	35.71	1.74	18.84	133.28
Ran1	**138.93**	1.64	35.72	1.66	18.87	126.42
Ran5	**195.16**	1.6	36.79	1.66	18.84	117.47
TCP	**57.65**	1.66	11.86	1.39	15.65	51.95

In summary, to improve the performance of *ng*AP, GPU architectures could employ techniques that reduce latency or improve the concurrency to hide latency, while other schemes are more sensitive to compute resources for integer operations.

6.4 Latency

We measure the latency of *ng*AP by evaluating its throughput in processing a 1 MB input stream. To coordinate all thread blocks to process the NFAs, *ng*AP groups all connected components (CC; the total number of CC for the evaluated application shows in Table 2) into M groups, and utilizes each thread block with its own worklist to process each group. We use $M = 200$ as the default group number in *ng*AP-DEFAULT while *ng*AP-BEST tunes to the best M for lower latency. Other parameters remain unchanged from those specified in Section 6. Figure 20 shows the speedup over GPU-NFA. We observe that *ng*AP-BEST results in better performance compared to prior GPU works by 10.9×, 3.2×, and 12.0×, respectively.

Table 6 shows the absolute throughput with one input stream. We observe that HyperScan has an advantage, especially for the applications with fewer NFAs, resulting in significantly shorter latency. In this scenario, the level of input stream parallelism is insufficient to utilize all the compute resources of GPUs. Therefore, we suggest enhancing *ng*AP by incorporating speculation or enumeration schemes (will discuss them in Section 7), allowing *ng*AP to function as if operating in a multi-input scenario. This augmentation would leverage all available GPU resources, resulting in latency improvements. *We conclude that while* ng*AP primarily focuses on enhancing the throughput of large-scale automata*

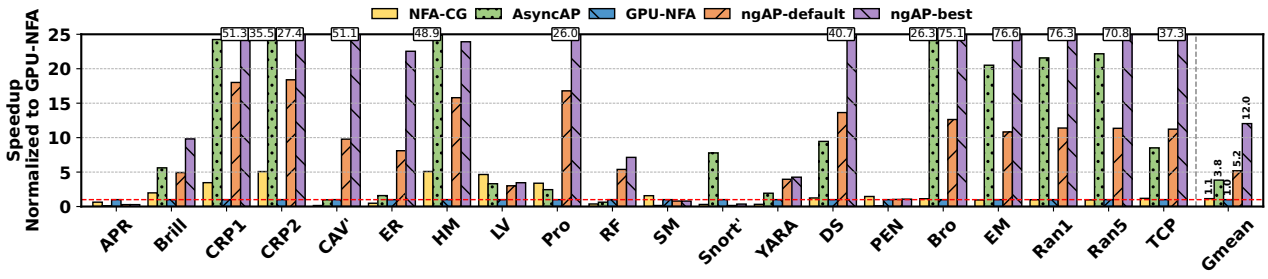


Figure 20. Latency results normalized to GPU-NFA. We define throughput (in bytes per second) for a single input stream as the measure of latency.

applications, it effectively reduces latency compared to prior GPU schemes.

7 Related Work

This section provides an overview of related work on automata processing, specifically focusing on the discussed techniques.

Improving Parallelism. Previous works have attempted to improve parallelism by enumerating, speculating, or using hybrid approaches to partition the input stream into segments and allow them to run in parallel. Enumeration approaches [30, 38] enumerate all possible active states at starting locations of input segments, but require significantly more work. Speculation approaches [36, 43–45, 69, 74, 75] speculate one active state at an input segment to reduce the work, but must recompute misspeculated input segments. Speculative enumeration [27, 70] is a hybrid of these two approaches. All these approaches focus on DFAs as only one state is active at any iteration, making it easier to speculate or enumerate. AsyncAP [35] focuses on NFAs by starting to process patterns from always-active states in parallel but performs poorly under imbalanced workloads. In contrast, *Prefetching Always-Active States* works on top of *ng*AP to improve parallelism and thread utilization. Unlike previous approaches, our optimization does not require enumeration or misspeculation handling, resulting in lower overhead. Additionally, *Prefetching Always-Active States* improves data locality due to optimized worklist scheduling (§ 6).

Reducing Computations. String algorithms [9, 28, 32] reduce redundant computations by keeping a memoization table. However, these approaches cannot be applied to automata processing. HyperScan [68] transforms a subset of NFAs into string matching, resulting in reduced time complexity. Other works use multi-striding processing [14, 16, 19] to process a stride at a time, reducing the total computations on symbols. However, multi-striding approaches do not scale well for large-scale automata problems, as the number of state transitions grows exponentially after transformation. In contrast, *Prefix Memoization* memoizes only a small portion of transitions into a look-up table, effectively reducing computations.

Data Movement and Locality. Many accelerators for automata processing reduce data movement by in-memory processing [22, 29, 50–53, 59, 60, 64]. Other than accelerators, transformation approaches [15, 56, 57] construct new representations for automata that have better locality and memory efficiency, which are complementary to our work. GPU-NFA [34] loads the topology information of always-active states into GPU registers, thereby reducing data movement. In contrast, our work optimizes data locality by new schedules of the worklist with *Prefetching Always-Active States* and *Work Privatization*, which are general to any automata.

8 Conclusions

We present Non-blocking Automata Processing (*ng*AP) which allows multiple symbols to be processed concurrently for automata processing on GPUs, and further broadens the design space in automata processing on GPUs: On top of *ng*AP, this work proposes optimizations focusing on addressing three identified challenges, poor thread utilization, redundant computations, and poor data locality. Evaluation of the synergistic approach demonstrates that our work outperforms state-of-the-art GPU automata processing engines significantly across a wide range of emerging applications.

Acknowledgments

We sincerely thank Tyler Sorensen, our shepherd, and the anonymous reviewers for their detailed feedback, which significantly improved the quality of the paper. We thank Zhenlin Wu and Haosong Zhao's assistance in proofreading. This material is based upon work supported by the National Natural Science Foundation of China (#62302416), the Guangzhou-HKUST(GZ) Joint Funding Program (#2023A03J0138), and a startup grant from The Hong Kong University of Science and Technology (Guangzhou). Corresponding author: Hongyuan Liu.

A Artifact Appendix

A.1 Abstract

This artifact includes the source code of *ng*AP and scripts to reproduce the experimental results for the Figure 13, Figure 14, Figure 20, Table 4 and Table 6. We also provide raw experimental data collected on the NVIDIA RTX 3090 platform and the Python scripts to generate these figures and tables.

A.2 Artifact check-list (meta-information)

- **Algorithm:** HyperScan [68], NFA-CG [77], GPU-NFA [34], AsyncAP [35], BAP, *ng*AP,
- **Program:** CUDA and C/C++ code
- **Compilation:** CMake 3.24.1, GCC 9.4, NVCC 12.0.1
- **Binary:** CUDA executables
- **Data set:** AutomataZoo [67], ANMLZoo [65], and Regex [17] benchmark suites
- **Run-time environment:** Ubuntu 20.04 with CUDA 12.0
- **Hardware:** x86_64 CPU with host memory larger than 32 GB. CUDA-enabled GPU with device memory larger than 24 GB.
- **Metrics:** Achieved throughput (bytes per second)
- **Output:** CSV files and running logs
- **How much disk space required (approximately)?:** 60 GiB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 27 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GNU General Public License v3.0
- **Archived (provide DOI)?:** 10.5281/zenodo.8354806

A.3 Description

A.3.1 How to access. The artifact is archived on Zenodo and made available on GitHub for any future updates or revisions.
https://github.com/getianao/ngAP
https://doi.org/10.5281/zenodo.8354806

A.3.2 Hardware dependencies. We developed and tested our work on two NVIDIA GPUs (RTX 3090 and Tesla V100 SXM2). *ng*AP is expected to run on GPUs with a compute capability of no less than 5.0 (Maxwell). Additionally, we recommend users employ GPUs with device memory exceeding 24 GB to handle large applications.

A.3.3 Software dependencies. All experiments are conducted under Ubuntu 20.04. The artifact requires the NVIDIA CUDA driver of version 525.60.13 or later, and CUDA Toolkit version 12.0.1. The artifact was tested using GCC 9.4.0, Python 3.8, CMake 3.24.1, and TBB 2020.1. HyperScan relies on GCC 5.3, boost, Ragel, and nasm.

A.3.4 Data sets. All datasets are from publicly available benchmark suites: AutomataZoo [67], ANMLZoo [65], and Regex [17]. We convert their automata files to ANML format [65] using MNCaRT [11] and VASim [66]. We provide the dataset that is ready to use in our repository.

A.4 Installation

The environment can be set up via Docker. Follow these steps to install the *ng*AP artifact:

```
$ git clone --recursive \
    https://github.com/getianao/ngAP.git
$ cd ngAP && source env.sh
$ ./1_download_benchmark.sh
$ ./2_build_docker.sh
$ ./3_launch_docker.sh
```

Within the Docker container, do the following steps:

```
$ ./4_build_all.sh
```

This step will generate all executables under `hscompile/build` and `code/build/bin` :

- `hsrun`: HyperScan [68]
- `ppopp12`: NFA-CG [77]
- `asyncap`: AsyncAP [35]
- `obat`: GPU-NFA [34]
- `ngap`: *ng*AP

For each scheme, you can check its usage by using the `-h` option to display help.

A.5 Experiment workflow

To run the experiments provided by the artifact, please follow the command below. Please note that these experiments typically require approximately 27 hours to complete.

```
$ ./5_run_all.sh
```

All resulting CSV files will be stored in the `result/raw` folder, and log files will be located in `raw_results` named according to the timestamp.

A.6 Evaluation and expected results

To generate the figures and tables (Figure 13, Figure 14, Figure 20, Table 4, and Table 6) from the data in the `result/raw` folder, use the following command. The generated figures and tables will be stored in the `result` folder.

```
$ ./6_gen_all.sh
```

For your reference, we have included results collected on the NVIDIA RTX 3090, as well as the figures and tables in the `ref_result` folder.

A.7 Experiment customization

Users are encouraged to conduct experiments with various parameters or additional applications by modifying the configuration file (under `code/scripts/configs`) or specifying the options manually. For more details, please refer to `README.md`.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] 2018. ANML Documentation. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.

[2] 2019. GNU Grep. https://www.gnu.org/software/grep/.

[3] 2019. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[4] 2020. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[5] 2021. CUDA Occupancy Calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls.

[6] 2022. ET Pro Ruleset. https://www.proofpoint.com/us/threat-insight/et-pro-ruleset.

[7] 2023. RE2. https://github.com/google/re2.

[8] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed A. Badawy. 2022. Demystifying the Nvidia Ampere Architecture through Microbenchmarking and Instruction-level Analysis. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*.

[9] Alfred V Aho and Margaret J Corasick. 1975. Efficient string matching: an aid to bibliographic search. *Commun. ACM* 18, 6 (1975), 333–340.

[10] Mehmet Altınel and Michael J Franklin. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*. 53–64.

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

[12] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. 2017. *MNRL and MNCaRT: An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem.* Technical Report CS2017-01. University of Virginia.

[13] Kevin Angstadt, Westley Weimer, and Kevin Skadron. 2016. RAPID Programming of Pattern-Recognition Processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[14] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. 2016. Scalable Algorithms for NFA Multi-Striding and NFA-Based Deep Packet Inspection on GPUs. *IEEE/ACM Transactions on Networking (ToN)* (2016).

[15] Michela Becchi and Patrick Crowley. 2007. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*.

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

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

[18] Chunkun Bo, Vinh Dang, Elaheh Sadredini, and Kevin Skadron. 2018. Searching for Potential gRNA Off-Target Sites for CRISPR/Cas9 using Automata Processing across Different Platforms. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.

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

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

[21] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. 2019. Debugging Support for Pattern-Matching Languages and Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[23] Adi Fuchs and David Wentzlaff. 2019. The Accelerator Wall: Limits of Chip Specialization. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[24] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

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

[26] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[27] Peng Jiang and Gagan Agrawal. 2017. 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)*.

[28] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast pattern matching in strings. *SIAM journal on computing* 6, 2 (1977), 323–350.

[29] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-Hardware Codesign for Efficient in-Memory Regular Pattern Matching. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

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

[31] Vincent T. Lee, Justin Kotalik, Carlo C. Del Mundo, Armin Alaghi, Luis Ceze, and Mark Oskin. 2017. Similarity Search on Automata Processors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[32] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. 2013. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Transactions on Computers (TC)* (2013).

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

[34] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2020. 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)*.

[35] Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2023).

[36] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. 2011. Speculative Parallel Pattern Matching. *IEEE Transactions on Information Forensics and Security* 6, 2 (2011), 438–451.

[37] Satoshi Matsuoka, Jens Domke, Mohamed Wahib, Aleksandr Drozd, and Torsten Hoefler. 2023. Myths and Legends in High-Performance Computing. https://doi.org/10.48550/ARXIV.2301.02432

[38] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. 2014. Data-parallel Finite-state Machines. In *Proceedings of the International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

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

[40] Marziyeh Nourian, Hancheng Wu, and Michela Becchi. 2018. 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).*

[41] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D. Santambrogio. 2021. CICERO: A Domain-Specific Architecture for Efficient Regular Expression Matching. *ACM Transactions on Embedded Computing Systems* (2021).

[42] Vern Paxson. 1999. Bro: a System for Detecting Network Intruders in Real-time. *Computer networks* 31, 23-24 (1999), 2435–2463.

[43] Junqiao Qiu, Xiaofan Sun, Amir Hossein Nodehi Sabet, and Zhijia Zhao. 2021. 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).*

[44] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. MicroSpec: Speculation-Centric Fine-Grained Parallelization for FSM Computations. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT).*

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

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

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

[48] Indranil Roy, Nagakishore Jammula, and Srinivas Aluru. 2016. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[49] Elaheh Sadredini, Deyuan Guo, Chunkun Bo, Reza Rahimi, Kevin Skadron, and Hongning Wang. 2018. 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).*

[50] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. 2020. 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).*

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

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

[53] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. 2019. eAP: A Scalable and Efficient in Memory Accelerator for Automata Processing. In *Proceedings of the International Symposium on Microarchitecture (MICRO).*

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

[55] Michael Sipser. 1996. *Introduction to the Theory of Computation* (1st ed.). International Thomson Publishing.

[56] Randy Smith, Cristian Estan, and Somesh Jha. 2008. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the IEEE Symposium on Security and Privacy.*

[57] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. 2008. 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).*

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

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

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

[61] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. 2020. Summarizing CPU and GPU Design Trends with Product Data. arXiv:1911.11313 [cs.DC]

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

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

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

[65] Jack Wadden, Vinh Dang, Nathan Brunelle, Tom Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. 2016. ANMLZoo: A Benchmark Suite for Exploring Bottlenecks in Automata Processing Engines and Architectures. In *Proceedings of the International Symposium on Workload Characterization (IISWC).*

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

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

[68] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

[69] Yuguang Wang, Robbie Watling, Junqiao Qiu, and Zhenlin Wang. 2022. GSpecPal: Speculation-Centric Finite State Machine Parallelization on GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS).*

[70] Yang Xia, Peng Jiang, and Gagan Agrawal. 2020. 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).*

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

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

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

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

[75] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. 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)*.

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

[77] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. 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)*.