Asynchronous Automata Processing on GPUs

HONGYUAN LIU*, William & Mary, USA / The Hong Kong University of Science and Technology (Guangzhou), China
SREEPATHI PAI, University of Rochester, USA
ADWAIT JOG*, William & Mary, USA / University of Virginia, USA

Finite-state automata serve as compute 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 while the other two levels of parallelism depend on the number of automata and input streams to be processed. As GPU resources increase, a parallelism-limited automata processing task can underutilize GPU compute resources.

To this end, we propose AsyncAP, a low-overhead approach that optimizes for both scalability and throughput. Our insight is that most automata processing tasks have an additional source of parallelism originating from the input symbols which has not been leveraged before. Making the matching process associated with the automata tasks asynchronous, i.e., parallel GPU threads start processing an input stream from different input locations instead of processing it serially, improves throughput significantly and scales with input length.

When the task does not have enough parallelism to utilize all the GPU cores, detailed evaluation across 12 evaluated applications shows that AsyncAP achieves up to 58× speedup on average over the state-of-the-art GPU automata processing engine. When the tasks have enough parallelism to utilize GPU cores, AsyncAP still achieves 2.4× speedup.

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

Additional Key Words and Phrases: pattern matching, automata processing, GPGPUs

ACM Reference Format:
Hongyuan Liu, Sreepathi Pai, and Adwait Jog. 2023. Asynchronous Automata Processing on GPUs. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 1, Article 27 (March 2023), 27 pages. https://doi.org/10.1145/3579453

1 INTRODUCTION

Finite automata (or Finite State Machines, FSMs) are used as compute kernels for many applications in various domains such as bioinfomatics [16], machine learning [44, 50, 55, 63], intrusion detection [3, 43], and textual data analytics [24, 25, 37]. Irregular memory accesses and intrinsic data dependencies make efficient automata processing extremely challenging for traditional architectures. The users, therefore, resort to domain-specific accelerators based on ASICs or FPGAs [19, 21, 28, 30, 41, 47, 49, 52, 53, 73]. For example, Micron's Automata Processor [19, 62]

*A majority of this work was done while the authors were with William & Mary, USA.

Authors' addresses and current affiliations: Hongyuan Liu, hongyuanliu@ust.hk, The Hong Kong University of Science and Technology (Guangzhou), Guangdong, China; Sreepathi Pai, sree@cs.rochester.edu, University of Rochester, Rochester, NY, USA; Adwait Jog, ajog@virginia.edu, University of Virginia, Charlottesville, VA, USA.

2476-1249/2023/3-ART27
https://doi.org/10.1145/3579453

Table 1. Parallelism in NFA processing

Parallelism Source	Example
Input Streams	Many network packets
NFAs	Many intrusion signatures
Active NFA States	Non-determinism intrinsic to NFAs
Input Symbols	Intrusion pattern can start at any input locations

repurposes DRAM to process automata and outperforms traditional architectures by orders of magnitude. However, these accelerators bring additional heterogeneity to the computing systems [57] and are slow to configure [62, 68] and inflexible to algorithm changes [22].

In contrast, GPUs are found in many computing systems from mobile phones to data center servers, working as general-purpose accelerators for performance-critical compute kernels as they provide massive data-level parallelism and high memory bandwidth. The computing power of GPU has scaled faster than CPUs in recent years [54]. Therefore, running automata processing tasks on GPUs has attracted significant attention [18, 31, 58, 70, 71, 76]. Existing automata processing works on GPUs have demonstrated that GPU achieves better performance than CPUs [34]. To process automata, two representations of automata are often used – Deterministic Finite Automata (DFAs) and Non-deterministic Finite Automata (NFAs). We focus on NFAs in this work because they are compact in size [31] and have more parallelism which makes them a good fit for the GPU.

An automata processing task is to find patterns defined by automata in the input streams. To parallelize the automata processing tasks on GPUs, as the first three rows of Table 1 show, existing works leverage three levels of parallelism: (1) input stream level, (2) automaton-level, and (3) state-level. First, the user may need to find a pattern across multiple input streams. 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. This level of parallelism only pertains to NFAs.

Only the state-level parallelism is intrinsic to NFAs, while the other two depend on the scale of the task that the user runs [42]. In order to meet different latency requirements, a task expects the execution engine to provide *strong scaling* such that adding more compute resources can significantly reduce the latency of the task [73]. On the other hand, the execution engine needs to utilize the compute resources well to retain high throughput when the task saturates compute resources. In short, an automata processing task requires both *scalability* and *throughput*.

To scale out tasks to more compute resources, other lines of work increase the parallelism by splitting input streams into input segments and breaking the dependencies between adjacent input symbols, enabling parallel execution of input segments. Ladner and Fischer [29] use parallel prefix sum to parallelize automata, though it leads to significantly more work. Speculation [38, 39, 74, 75] and enumeration [33] are two categories of approaches that increase the parallelism of automata on multi-core processors. These works focus on scalability rather than throughput, and are often evaluated using DFAs as each DFA only has one active state at 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 input segment; 2) enumerating every *set of states* at every input segment leads to prohibitive execution paths.

To address these problems and provide high throughput and scalability for different task sizes, we propose Asynchronous Parallel Automata Processing, AsyncAP, a simple and low-overhead way to exploit an additional source of parallelism of automata processing on GPU. Unlike prior works, AsyncAP does not speculate or enumerate on execution paths. As a result, AsyncAP requires no

validation process on the correctness of the extra execution paths, resulting in better utilization of GPU resources. Instead, 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, bringing an additional level of parallelism that was not exploited by the prior works as shown in the fourth row of Table 1. 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. Thus, the matching processes mapped to GPU threads can start from different locations in parallel.

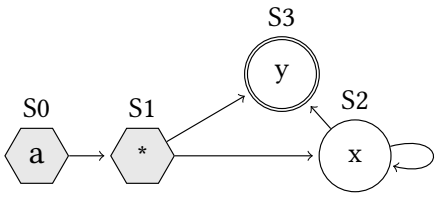
However, AsyncAP has higher worst-case time complexity because an input stream with n symbols needs to be read $O(n^2)$ times at most. Nevertheless, we observe that this upper bound is rarely reached in practice for two reasons: first, due to mismatches (when no state is active in a thread), the amount of work depends on both application and input. An automaton that matches every symbol of the input stream regardless of the starting location is very likely to be practically meaningless. Second, the time complexity only shows the upper bound of the useful work, but the total work considering implementation and synchronization may be different in AsyncAP and traditional approaches. To the best of our knowledge, no prior work has analyzed the source of work in practice.

To understand the source of work, we perform a systematic characterization on previous synchronous automata processing (e.g., GPU-NFA [31]) and AsyncAP. 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 blocks or SIMD-style execution. We find that while AsyncAP has higher time complexity than GPU-NFA, AsyncAP only requires 6% of extra useful work than GPU-NFA on average across the evaluated applications. Further, if the latency per unit work is considered as weight, AsyncAP requires 15% less total weighted work on average compared to GPU-NFA. We conclude that despite its apparent higher time complexity, AsyncAP does not lead to considerably more work in practice.

To show whether AsyncAP provides both scalability and throughput, we evaluate AsyncAP in scenarios with a different amount of parallelism. Compared with GPU-NFA, a state-of-the-art automata execution engine on GPU, when the parallelism is not enough to utilize GPU cores, an optimized version of AsyncAP achieves 9.4× to 57.9× speedup on average across 12 evaluated applications, respectively, demonstrating it provides strong scaling for parallelism-limited tasks. When the original parallelism of the automata task is enough to utilize GPU cores, AsyncAP achieves 2.4× speedup over the state-of-the-art prior work demonstrating that it also achieves high throughput. Executed by AsyncAP, one of the evaluated applications has a significant slowdown due to the imbalance of threads caused by long patterns. However, static analysis and runtime detection can help us alleviate the impact of the slowdown. Statically, our study on the topology of automata shows that long patterns are uncommon as most automata usually do not have infinitely long patterns. Hence, one can choose only offloading the automata of short patterns to AsyncAP. Moreover, at runtime, we can detect the slowdown easily, so only marginal overhead is incurred before rolling back to the traditional approach.

In summary, this paper makes the following contributions:

- We analyze and find that the prior work cannot adapt to automata processing tasks with a different amount of parallelism.
- We propose AsyncAP, a simple and low-overhead way to execute automata on GPUs that exploits an additional source of parallelism neglected by prior works.
- Regardless of AsyncAP's higher time complexity, based on our characterization, on average, AsyncAP only incurs marginal extra useful work and needs less total work considering both per symbol latency and useless work caused by the implementations.



(a) NFA accepting a*..*x*y. *"All-input"* starting states S_0, S_1 are always active.

Step	Symbol	Active States	Matched States
0	x	S0 S1	S1
1	y	S0 S1 S2 S3	S1 **S3**
2	z	S0 S1 S2 S3	S1

(b) The complete matching process of input stream xyz: bold font indicates a report is generated.

Fig. 1. Illustrating the Matching Process of an NFA

- Evaluation results demonstrate that our approach provides both high scalability and throughput across the evaluated applications over the prior state-of-the-art automata execution engine on GPU.

2 BACKGROUND

2.1 Pattern Matching via NFAs

A finite automaton is a mathematical model of computation in which the computations are abstracted as a 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. As also used in many prior works [65], especially accelerators [47, 52], we focus on Glushkov NFAs [23], which are ϵ-free and the set of symbols that are matched 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.

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 1, shown in hexagons) and *reporting states* (Figure 1, shown in double circles).

Types of Starting States. A NFA processing application often uses ANML [1] or MNRL [10] format to define the NFAs. Two types of starting states are used in the NFAs of the applications. If an NFA searches for patterns that appear regardless of the starting position in the input stream, it has only *all-input* starting states 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. In contrast, an NFA equivalent to pattern `/^apple/` contains *start-of-data* starting states, as it requires "apple" to appear only in the first position.

Matching Process. 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 *activate* their successors. This process finishes when all the symbols of the input stream are processed.

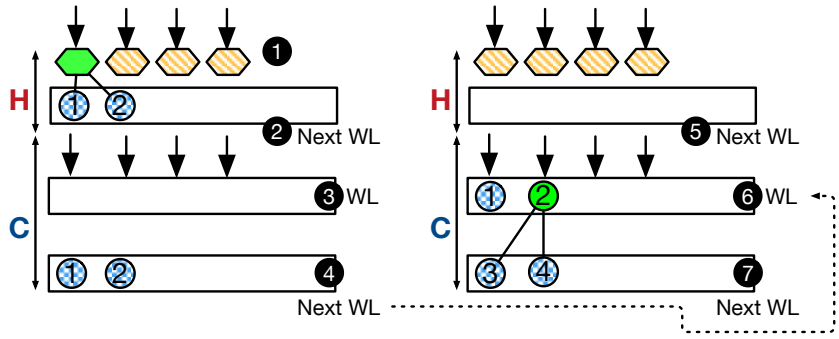


Fig. 2. Key Idea of the State-of-the-art NFA Processing Engine on GPU

2.2 Automata 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. Currently, applications with a large number of automata are a good fit for GPUs because they exhibit parallelism at multiple levels [34, 61] as shown in Table 1. First, input streams (e.g., different network packets) can be processed in parallel. Second, many NFAs (e.g., different intrusion signatures) can run in parallel on the same input stream (e.g., a single network packet). Third, multiple states can be active at the same time within the same NFA and the same input symbol. The state-of-the-art NFA processing engine on GPU, GPU-NFA [31], leverages these three levels of parallelism.

Figure 2 illustrates the basic idea of GPU-NFA. The key observation of GPU-NFA is that the activation frequency varies for automata states. A portion of states are frequently activated (hot) while other states are infrequently activated (cold). To increase the utilization of GPU, GPU-NFA only maps hot states to GPU threads (❶), and stores the neighbors and matchsets of the hot states in the registers. The cold states are processed by a worklist shared by the thread block. To process an input symbol, the execution comprises a hot stage (H) and a cold stage (C). At the hot stage, when a hot state is matched, it activates its neighbors and pushes them (①②) into the *next* worklist (❷). Then, the same threads are reused to process the cold states that are in the worklist (which was the `next` worklist of the prior symbol) (❸). If no state is in the worklist, the *next* worklist is assigned to the worklist, and the *next* worklist is emptied (❹). A barrier is needed before processing the next symbol. Next, the hot stage finishes when no hot state matches the input symbol (❺). The cold stage, hence, starts to process cold states ① and ② (❻). The matched state ② activates its neighbors and pushes them into the *next* worklist (❼). This process continues until all the input streams and all the symbols in the inputs are consumed.

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.

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

To understand the need for AsyncAP, we revisit the existing works of automata processing on CPUs, GPUs, and accelerators. Table 2 categorizes the existing works for automata (both DFA and NFA) processing into three categories based on input stream accessibility, the number of automata, and input streams. We make the following observations.

Table 2. Categories of Prior Works

Type	#Input	#FAs	HW Util.	Focus	Examples
Buffered	Many	Many	Over	Throughput	GPU-NFA [31] NFA-CG [76], iNFAnt [18]
Buffered	Single	Few	Under	Scalability	Speculation [26, 39, 75] Enumeration [33]
Streaming	Single	Any	Under	Latency	Multi-stride [14] Graph Transformation [72] HW Accelerators [30, 47]

Throughput-focused works are not scalable. The first category (the first row in Table 2) focuses on throughput, which considers the *high parallelism case* when the task requires more resources than the hardware can provide. Particularly, this type of work addresses the bottleneck of automata processing in oversubscribed hardware. Typical optimizations include data movement and hardware utilization optimizations. For example, NFA-CG [76] calculates compatible groups of NFA states to enhance the thread utilization of GPU. GPU-NFA [31] 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 do not provide strong scaling for parallelism-limited tasks.

Scalability-focused works have higher overhead. The second category (the second row in Table 2) focuses on the scalability issue in a *low parallelism* case of automata processing. Existing works [33, 39, 74, 75] extract additional parallelism by splitting the input stream into segments and making each segment of the input stream run in parallel. To address the dependencies across input symbols and ensure the results are correct, these approaches 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 leads to more overhead and limits the throughput.

Streaming-focused works only optimize for latency. The third category (the third row in Table 2) optimizes for per-symbol latency when the input stream does not support random access. The domain-specific accelerators for automata processing often fall into this category [19, 21, 30, 45–49, 52, 53, 69]. Many works leverage in-memory processing to reduce the latency caused by data movement through the memory hierarchy. These accelerators cannot provide scalability for parallelism-limited tasks. Moreover, when the number of automata states exceeds the capacity of the hardware, the accelerator requires many batches to finish the task, resulting in poor throughput [30]. Other works in software construct multi-stride automata [11, 14, 17] or compress automata by graph transformation [13] to reduce the per symbol latency at the cost of an increased number of states and state transitions. When the tasks have abundant parallelism, the increased number of states and transitions hurts the throughput on limited hardware resources.

In summary, no single prior work can handle all scenarios of automata processing tasks with varying parallelism. A *scalable* and *high-throughput* scheme on GPU for all automata tasks is required.

3.2 Overview of Asynchronous Parallel Automata Processing

Figure 3 compares the traditional synchronous execution of automata processing with our proposed asynchronous automata processing on GPU. First, Figure 3 (1) shows the basic idea of synchronous execution. Generally, the NFAs states are mapped to the GPU threads (❶). In particular, as discussed in Section 2.2, GPU-NFA maps only hot states to threads. The thread block reads the symbols from the input stream (❷). A thread block barrier (`__syncthreads()`) ensures that all states have

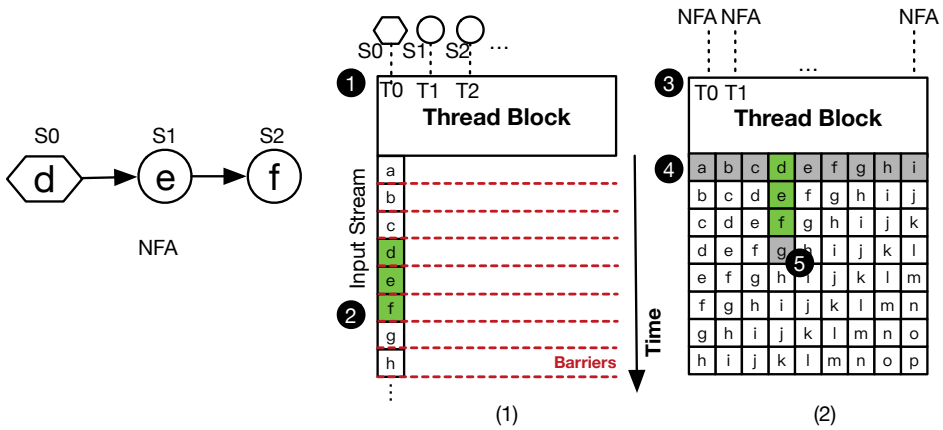


Fig. 3. Revisiting a generic synchronous automata processing on GPU (a) and the basic idea of AsyncAP (b). The executions try to find pattern `def` in an input stream `abcdefg...`.

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 input streams—the two numbers come from the automata processing task. When the task is limited in parallelism, it cannot utilize all cores of the GPU.

To address this issue, AsyncAP aims to increase the parallelism of automata on GPU. AsyncAP separates the execution paths *for each symbol of the input stream* to gain parallelism (Figure 3 (2)), namely *symbol-level parallelism*. To ensure the correctness, the automata task should be able to start matching anywhere in the input stream, i.e. the starting states of the NFAs must be "*all-input*" starting states.

We map each symbol of the input stream to each thread (❸). Each thread keeps its index at which the symbol being processed is located, and also accesses the topology and matchsets of NFAs from GPU global memory. 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-data* starting states. We match them for all starting positions, 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). For example, thread 3 starts by loading an input symbol from `d`, which is the 3rd location of the input stream (`abcdefghi`).

Each thread keeps its own set of active states as different threads act asynchronously. 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 3 (2), thread 3 has active states from positions 3 and 5 of the input stream (`def`), but mismatches at position 6 (g), so the thread terminates at position 6. Other threads in Figure 3 (2) do not match in the beginning, so they finish when matching the first symbols mapped to them.

Since the input stream is very long (otherwise it is a trivial problem), it provides enough parallelism to utilize GPU cores. Although GPUs may not have enough physical thread contexts to accommodate all threads at once, the hardware thread block scheduler schedules thread blocks to

GPUs as resources become available and older thread blocks finish [9]. We also applied granularity coarsening [32] to ensure the maximum number of threads needed does not exceed the limitation of the thread block scheduler (currently $2^{31} - 1$).

Applicability of AsyncAP. AsyncAP applies to the NFAs that only have *all-input* starting states. We investigate all applications in two benchmark suites, ANMLZoo [59] and AutomataZoo [61]. 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. Based on this observation, we conclude that AsyncAP is applicable for most of the applications in the existing benchmarks suites.

Compatibility with Modern GPUs. The implementation of AsyncAP is written in CUDA. We have run it on NVIDIA GPUs of Pascal and Ampere Architecture. We expect the proposed AsyncAP could be compatible with most NVIDIA GPUs.

3.3 Design Space Exploration and Implementation

We examine the major decisions across several points in the design space of implementing AsyncAP.

NFA Data Structures. Each thread accesses the NFA data structures to check the neighbors of the active states and whether the active states match with the symbol. Two ways are commonly seen in the literature to store the topology of the NFAs. First, most prior works [18, 76] 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 current state. Second, other works [31, 34] use per-node data structures similar to Compressed Sparse Rows (CSR) that decouple the alphabet and the states. With matchset compression [31], such data structure reduces the data movement by placing it 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 AsyncAP does not reuse the topology or matchsets of *all-input* states. The per-node data structure is larger because it stores matchsets separately. When they are not put into registers and are reused frequently, more data movement is needed. Therefore, AsyncAP uses transition tables as data structures of NFAs.

Per-thread Worklists. Each thread maintains double-buffered private worklists. Initially, the worklist only contains the starting states, and then these states match with the symbol mapped to this thread. If matched, the states extend their neighbors into the other worklist. The other worklist, in the next iteration, is assigned to the current worklist by reference, and the next worklist is emptied by updating its tail pointer. The matching process terminates when the current worklist is empty. A simple way to implement the double-buffered worklists is to store them in arrays private to each thread in the local memory. Although local memory can also utilize the caches in recent generations of GPUs [2], since the GPU kernel of NFA processing is latency-bound, the worklist arrays compete for the cache space with other data structures (e.g., transition table). This may affect latency and degrade performance. To address this issue, we propose hybrid worklists in which the first K elements are stored in the GPU's shared memory while the rest is stored in the local memory. The key observation behind it is that for most of the symbols, worklists are short. We observe that the length of worklists is not larger than 1 for over 99% of the time by profiling the first 1,000 symbols for each of the evaluated applications. However, the optimal K depends on input (i.e., the length of the worklists at runtime). For the baseline version of AsyncAP, we set $K = 0$

(local memory only). We treat K as a tuning knob and determine it empirically in Section 5 for an optimized version of AsyncAP.

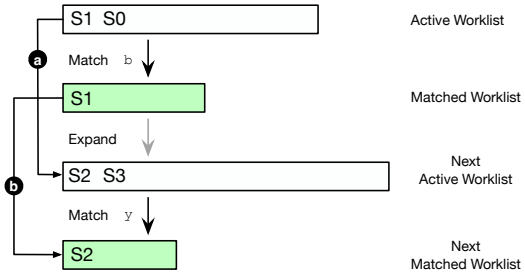


Fig. 4. Worklist holds active states or matched states

Storing Active or Matched States in Worklists? Whether worklists hold *active* states or *matched* states affects the matching process. Figure 4 illustrates this. If a worklist holds active states, when one of the active states matches the incoming symbol, it activates its successors before the next symbol comes. In this figure, S1 matches with b and expands to S2 and S3, which are then pushed to the *next active worklist* (a). In contrast, if a worklist holds *matched* states, each state tries to match the next symbol with all its successors one by one and pushes the matched successors to the *next matched worklist*. In the figure, matched state S1 first expands to S2 and S3, but only S2 matches with symbol y, and hence only S2 is stored in the matched worklist (b). Essentially, the steps are the same (*match* and *expand*) whereas the key differences are what is stored in the memory and what is computed on-the-fly. GPU-NFA [31] holds *active* states in its worklists because it keeps hot states in registers. When a match happens, it does not need to access *matchset* of each state. On the other hand, AsyncAP disables the *all-input* states, so these hot states are not reused across threads and therefore we found that holding matched states performs *significantly better* in AsyncAP.

How many independent NFAs are grouped for a kernel launch? AsyncAP exploits symbol-level parallelism, so even a 1MB input stream can utilize all GPU threads in high-end GPUs. It is always sufficient to utilize all GPU cores regardless of other levels of parallelism. As a result, we have the option of running only one automaton with a kernel launch. This trades NFA-level parallelism for the locality as each kernel instance only accesses smaller transition tables ($\leq$ a few MBs). In contrast, launching a kernel processing a group of NFAs (e.g., every M connected components are grouped) at a time reduces the total accesses to the memory. For instance, having fewer kernel launches reduces the total loads from the input stream as each kernel launch has to load the input stream. For the baseline version of AsyncAP, we set M to 1 (i.e., "one-automaton-one-kernel"). We empirically determine M to *tune* an optimized version of AsyncAP in Section 5.

Putting it All Together. Figure 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). Here, we examine the Tth thread which starts execution from position T of the input stream. Suppose the current location is $T + 2$ (❸), and the current worklist of Tth thread contains matched states S_u and S_v (❹). Then, two cells of the transition table are accessed (❺), and the matched states are pushed to the next worklist (❻). This finishes processing the current symbol at $T + 2$. As soon as the current worklist is empty, the execution of thread T terminates.

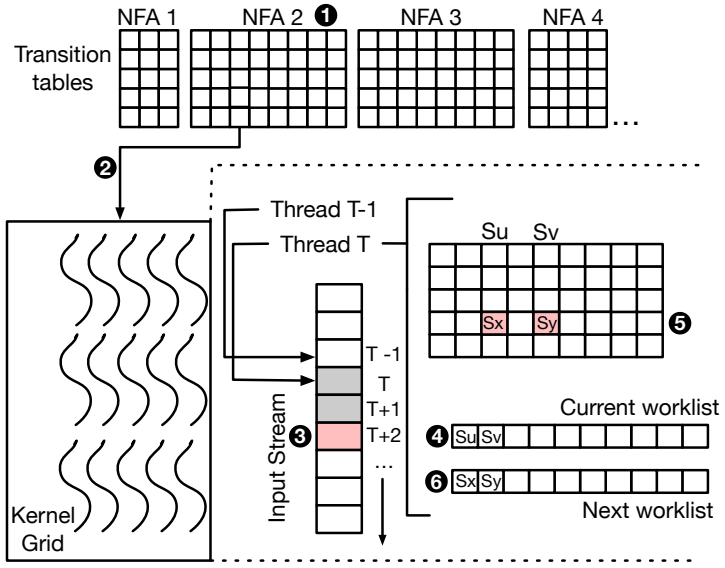


Fig. 5. Illustrating the Implementation of AsyncAP

3.4 Analysis

Correctness. We analyze why AsyncAP generates 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 s_0. It is always active. Let the set of active states for the NFA at position p as S_p (S_p must contain s_0 for any p). The function $expand(S_p, input[p])$ takes S_p and $input[p]$ (input symbol at position p), and calculates what the active states are 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$[p]$.

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

$$= \underbrace{expand \ldots (expand(expand}_{p \text{ times}}(\{s_0\}) \cup \{s_0\})) \cup \{s_0\}) \cup \{s_0\} \tag{2}$$

$$= \underbrace{expand \ldots}_{p \text{ times}}(\{s_0\}) \cup \underbrace{expand \ldots}_{p-1 \text{ times}}(\{s_0\}) \cup \ldots expand(\{s_0\}) \cup (\{s_0\}) \tag{3}$$

Here, the first line shows the synchronous execution. Based on the definition, we have $expand(A \cup expand(B)) = expand(A) \cup expand(expand(B))$, where A and B are sets of active states. Consequently, it reduces to the last equation which 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 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. In the worst case, thread at position i ($0 \leq i < n$) needs to match with $n - i$ symbols. Summing 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 3), but in Section 4, by detailed characterization, we will show that the worst case is unlikely in real applications.

4 UNDERSTANDING WORK CHARACTERISTICS OF ASYNCAP

Time complexity analysis (Section 3.4) only shows the upper bound of the required amount of work, however, it does not reflect the runtime characteristics that are input-dependent and implementation-dependent. To understand the amount work to be done by AsyncAP, we characterize them focusing on four aspects by emulation. First, to study the real cases compared to theoretical time complexity, we compare GPU-NFA and AsyncAP in terms of useful work (Section 4.3). Second, we show how GPU is utilized for useful work and useless work (Section 4.4). Third, we compare them in terms of total work (Section 4.5). Fourth, we study how work is distributed across GPU threads in AsyncAP (Section 4.6).

4.1 Application Configuration

We evaluate 13 applications from three benchmark suites, ANMLZoo [59], AutomataZoo [61], and RegEx [15]. To keep the NFA-level parallelism the same for all applications, we randomly sample 256 NFAs from each of them with a fixed random seed (1234) to ensure reproducibility. Table 4 shows the basic characteristics of the evaluated applications and confirms that sampling does not change the basic characteristics of NFAs in the applications. We also change the random seed to other numbers and observe that the results are similar.

It is important to note that when there is only one input stream, no evaluated application can fully utilize a commodity GPU without exploiting symbol-level parallelism by AsyncAP. The maximum application (CAV) contains 33,171 NFAs, which is also the number of threads by default in the best scheme of GPU-NFA, and which is far lower than the hundreds of thousands of threads a modern GPU can accommodate [8]. Although dynamic activities (e.g., activations of states) may also affect the parallelism of the task, traditional ways to process NFAs on GPUs cannot scale out automatically according to the runtime characteristics.

Each application has a representative input stream collected by the benchmark suites. We use the first 1MB input stream to characterize the applications as suggested by prior work [31].

4.2 Emulation and Work Metrics

This section describes the setup of the emulation of synchronous automata processing (GPU-NFA) and AsyncAP, and shows the metrics we focus on.

To understand the work of the two schemes, We bisect the work at runtime into useful work and useless work according to whether the work is required to generate correct results.

Table 4. Overview of Evaluated Applications

		Sampled			Unsampled			
App.	Abbr.	#States	Avg. NFA Size	SD.	#States	Avg. NFA Size	SD.	#NFAs
Brill [61]	Brill	5005	19.6	6.1	115549	19.4	5.6	5946
ClamAV [61]	CAV	17932	70.0	41.2	2374717	71.6	144.1	33171
CRISPR_CasOT [61]	CRISPR1	25856	101.0	0.0	202000	101.0	0.0	2000
CRISPR_CasOFFinder [61]	CRISPR2	9472	37.0	0.0	74000	37.0	0.0	2000
ExactMatch [15]	EMatch	10594	41.4	15.4	12439	41.9	15.6	297
EntityResolution [61]	ER	10656	41.6	6.7	413352	41.3	7.0	10000
Hamming_l18d3 [61]	HM	27648	108.0	0.0	108000	108	0.0	1000
PowerEN [59]	PEN	3783	14.8	7.8	40513	14.2	7.9	2857
Ranges05 [61]	Rg05	10843	42.4	17.2	12621	42.2	16.9	299
Ranges1 [61]	Rg1	10688	41.8	15.7	12464	42.0	15.4	297
Snort [61]	Snort	19906	77.8	178.1	202043	81.3	242.0	2486
TCP [15]	TCP	6331	24.7	11.3	19704	26.7	22.0	738
YARA [61]	YARA	9331	36.5	45.1	1047528	44.5	59.5	23530

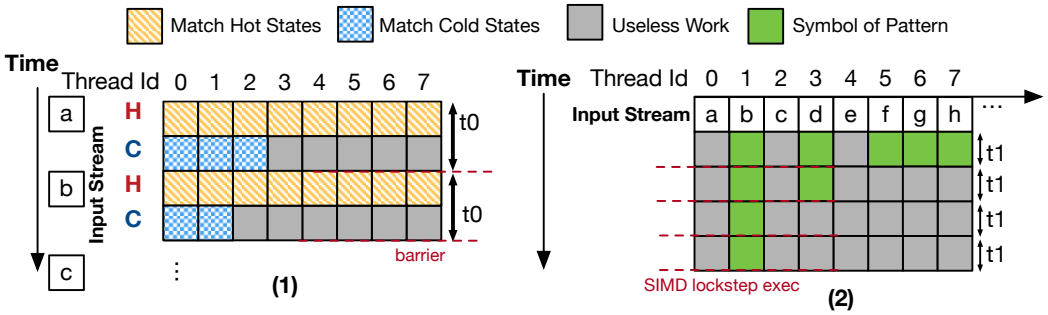


Fig. 6. Illustrating the Execution of GPU-NFA and AsyncAP on GPU

Useful Work. When an active state tries to match with the incoming symbol, we count it as a unit of useful work (i.e., required work) in the emulation. Useful work is necessary to generate correct results. The time complexity analysis (Section 3.4) estimates the lower bound and upper bound of useful work, but is difficult to estimate the actual useful work as it is input-dependent.

Useless Work. In contrast, *useless* work is due to either thread block synchronization or warp synchronization. It is also input-dependent. If a thread is waiting for other threads to match with an incoming symbol for synchronization compulsorily, 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 [5–7], simulators [12, 27], or emulators [20], these tools find it difficult to differentiate the *useful* work and *useless* work specific to the workload on GPU. Therefore, we emulate these workloads on the CPU with the following models based on their simplified implementation ideas.

Model of GPU-NFA. Based on the key idea illustrated in Figure 2, Figure 6 (1) shows how we emulate GPU-NFA [31]. Processing a symbol requires two stages. The hot states mapped statically to threads match with the incoming symbol at the hot stage. Next, in the cold stage, 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, leaving other threads in the same

Table 5. Characteristics of applications based on our emulation results. W. stands for weighted. The weights are the latency of unit work.

	Compare AsyncAP w. GPU-NFA by Ratio				Per Symbol Latency		GPU Utilization	
App.	Useful	Total	W. Useful	W. Total	GPU-NFA	AsyncAP	GPU-NFA	AsyncAP
Brill	1.00	4.87	0.92	4.48	1490.3	1370.6	0.89	0.18
CAV	1.00	0.74	0.84	0.62	789.3	663.4	0.67	0.90
CRISPR1	1.00	1.91	0.50	0.95	2497.2	1256.3	0.96	0.50
CRISPR2	1.00	1.89	0.51	0.98	2672.7	1374.9	0.89	0.47
EMatch	1.00	0.86	0.57	0.48	1545.3	874.1	0.57	0.67
ER	1.00	3.18	1.00	3.17	1133.9	1130.1	0.85	0.27
HM	1.00	1.59	0.82	1.31	1734.7	1430.2	0.93	0.59
PEN	1.04	1.91	0.59	1.08	1483.6	840.5	0.52	0.29
Rg05	1.00	0.84	0.51	0.43	1639.0	838.0	0.57	0.68
Rg1	1.00	0.85	0.54	0.46	1565.5	846.0	0.57	0.67
Snort	1.95	1.70	0.92	0.80	3003.8	1413.4	0.53	0.61
TCP	1.01	0.89	0.56	0.49	1457.8	800.3	0.52	0.58
YARA	1.00	0.66	0.43	0.28	1962.9	845.3	0.56	0.86
Mean	1.06	1.40	0.65	0.85	1767.4	1052.6	0.67	0.51

thread block *idle* for synchronization (grey cells) as the implementation uses static scheduling. The work wasted by the idle threads is considered as *useless* work. We use block size 256 in our emulation, which is also used in GPU execution because it can reach the best performance and highest GPU occupancy.

Model of AsyncAP. Figure 6 (2) shows the execution model of AsyncAP. Each thread is mapped to an input position as depicted in Section 3.2. The thread runs until no state is active. Thus, the threads run for a different number of symbols depending on the length of patterns in the threads. A cell counts as a unit of work. Due to the SIMD execution of GPU, the threads within a warp are synchronized implicitly. For example, when a thread is processing a long pattern, other threads in the same warp are idle (shown in grey cells). These grey shaded cells are *useless* work in AsyncAP and the green cells illustrate useful work. We use 32 as the warp size in the emulation, which is the same as NVIDIA GPUs [2].

Latency of Unit Work. Since time complexity analysis also ignores constants, we measure the average processing cycles of a unit of work on GPU. The t_0 and t_1 shown in Figure 6 illustrate the latency of unit work in GPU-NFA and AsyncAP. The measured latency, therefore, could be treated like the constants in the time complexity analysis. We use t_0 and t_1 as the *weight* of a unit work in GPU-NFA and AsyncAP, respectively. We launch a small grid in which only the first 1,024 symbols are processed to exclude the effects of the latency due to thread block scheduling. We record all latency values in all threads and then average them across the total number of *active* states of all the threads. Table 5 demonstrates that AsyncAP has significant lower (41%) latency for a state to process a symbol than GPU-NFA.

4.3 Analysis of Useful Work

Table 5 shows the results of useful work based on our execution models. The first four columns of the Table 5 compare the metrics of AsyncAP and GPU-NFA by ratios between the two schemes where a value less than 1.0 indicates that AsyncAP requires less work. As we analyzed in Section 3.4, AsyncAP can have more useful work than GPU-NFA, however, we observe that for all evaluated applications, the useful work is *far from the upper bound of AsyncAP's time complexity*. For most applications (11 out of 13), AsyncAP requires less than 1% of useful work compared with GPU-NFA.

Table 6. Spearman's Correlation between Execution Model Metrics and Execution Time: Using weighted work correlates better to the throughput. PEN was excluded when calculating this table due to its skewed load balance.

Scheme	AsyncAP		GPU-NFA	
Metrics \ Results	Corr.	P-Value	Corr.	P-Value
Useful Work	0.83	0.0008	0.75	0.001
Total Work	0.95	2×10^{-6}	0.88	0.00001
Weighted Total Work	0.96	9×10^{-7}	0.90	0.000005

The only outlier, Snort, needs 95% more useful work in AsyncAP. Overall, AsyncAP only requires 6% more useful work than GPU-NFA across the evaluated applications on average (geometric mean).

We also compare the useful work considering the latency of a unit of work. We calculate *weighted useful work* by multiplying the latency (introduced in Section 4.2) by the amount of useful work. Overall, according to the third column of Table 5, AsyncAP spends 35% and 15% *less* useful work compared to GPU-NFA.

We conclude that *on average AsyncAP requires less weighted useful work for all applications than GPU-NFA, indicating that shorter latency of AsyncAP could compensate for the extra useful work.*

4.4 How GPU is Utilized?

Only useful work is necessary. Therefore, we study how much percentage of the work is spent on useful work for GPU-NFA and AsyncAP.

The last two columns of Table 5 show *useful work utilization ratio* of the two schemes, calculated by $\frac{\text{Useful work}}{\text{Total work}}$. Larger values indicate higher GPU utilization. We observe that the useful work utilization ratio is application-dependent. Overall, the ratio of useful work is 0.51 and 0.67 in AsyncAP and GPU-NFA on average (geometric mean) across the evaluated applications, respectively. For example, AsyncAP has a larger utilization ratio for several applications (e.g., YARA, Rg1, CAV). We observe that the variance of pattern length affects the utilization of AsyncAP (Table 7), and affects the work distribution in AsyncAP (Section 4.6). Less variance is favorable to the useful work utilization ratio of AsyncAP. We conclude that *GPU-NFA and AsyncAP both have a large portion of useless work* that could be further improved.

4.5 Analysis of Total Work

Total work consists of useful work and useless work. Table 6 corroborated that total work is more accurate to estimate the execution time as it considers the synchronization overhead (modeled as *useless work*).

The second column of Table 5 shows the ratio of total work between AsyncAP and GPU-NFA. A value greater than 1.0 indicates that AsyncAP requires more total work and vice versa. We observe the total work is highly application-dependent. Compared to GPU-NFA, Brill and ER executed with AsyncAP use multiple times of total work (4.87×/3.18× work of GPU-NFA, respectively), while for applications such as CAV and YARA, AsyncAP has around 30% less total work than GPU-NFA. On average across the evaluated applications, AsyncAP requires 40% more total work than GPU-NFA, because the implicit warp synchronization caused by processing patterns with various lengths leads to more useless work.

Considering the latency of unit work, we multiply the measured latency of GPU-NFA and AsyncAP by the total work, which is shown in the fourth column of Table 5. In contrast, AsyncAP

Table 7. Pattern Length of the Evaluated Applications and Imbalance Ratio of AsyncAP

	Pattern Length			Imb. Ratio
App.	Max	Avg.	SD.	
Brill	75	1.59	2.0	14.6
CAV	16	2.67	0.9	3.2
CRISPR1	24	4.02	1.3	11.6
CRISPR2	13	1.00	0.1	3.2
EMatch	22	1.14	0.5	27.00
ER	42	1.02	0.2	10.0
HM	19	5.33	1.3	2.3
PEN	923049	1.05	142.4	486913.1
Rg05	46	1.02	0.2	53.8
Rg1	46	1.02	0.2	47.7
Snort	9558	1.31	6.8	5677.1
TCP	1108	1.02	1.1	2795.7
YARA	27	1.12	0.4	114.5

requires ~15% less weighted total work than GPU-NFA due to its lower average latency to match a symbol.

4.6 How the work is distributed in AsyncAP?

GPU-NFA balances the works in the worklist within a thread block. However, how the work is balanced in AsyncAP depends on the pattern characteristics of applications, relying on the thread block scheduler to balance them. Therefore, we measure how the work is distributed across threads in AsyncAP.

$$Imbalance\ Ratio = \frac{Max\ Work\ of\ Warps}{Average\ Work\ Per\ Warp} \tag{4}$$

Although the standard deviation of pattern length (shown in Table 7) indicates how the useful work varies in threads, it does not reflect how many thread blocks (or warps) we need in the kernel grid to balance the work across GPU compute units. Therefore, we define *imbalance ratio* shown in Equation 4 that indicates how many average warps are needed to offset a very slow warp.

Table 7 demonstrates that the thread imbalance situation varies across the applications. A few applications, such as `CRISPR2` and `HM`, have very balanced work across threads (i.e., the warp that has the most work only costs a few times more works than the average), potentially because their automata are similar in shape and size. While `CAV`, `ER`, and `Rg05` are more diverse, their work is also well balanced. On average (geometric mean), the applications need 75.1 warps to offset a warp with a lot of work. On the other hand, a few applications (e.g., `Snort`, `TCP`) have a relatively large imbalance ratio, but launching many thread blocks (i.e., the number of thread blocks for a 1MB input stream) can compensate for the imbalance. However, `PEN` has a dramatically large value (orders of magnitude larger than the values of other applications), showing its threads are severely imbalanced.

We do not apply software-based load balance to AsyncAP as it requires global synchronization across all threads, whose performance is not optimal in automata processing [31]. Instead, the thread block scheduler can balance the work in a coarse-grained way by assigning a new block. As AsyncAP has a bigger pool of thread blocks due to the increase in parallelism, we rely on the hardware scheduler to balance the work. *We conclude that most applications distribute the work across threads well, but rarely, extremely imbalanced work can happen.*

5 EVALUATION METHODOLOGY

This section describes our evaluation methodology.

Application Configuration. We use the same application configuration as Section 4.1.

Hardware Platforms. We primarily use an NVIDIA Quadro P6000 GPU for evaluation. We also perform a sensitivity study on an NVIDIA RTX 3090 GPU to show the effectiveness for other GPU architectures. We use a 12-core Intel Xeon Silver 4214R for CPU evaluation.

Performance Measurement. We use throughput = $\frac{\#\text{symbols}}{\text{Time}}$ to measure the performance. 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. We measure the end-to-end time from launching the first kernel to the end of execution when the reports 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 a different amount 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 compute 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 SMs, so we consider the scenario with 240 input streams as *high* parallelism as it requires at least 240 thread blocks [4].

Table 8. Overview of Evaluated Schemes

Scheme	Description
VASim	CPU Implementation VASim [60]
Sync-HS	HotStart in GPU-NFA [31]
Sync-HC	NT-MaC-ACP in GPU-NFA [31]
ScaleOut-HC	Sync-HC such that each thread block starts at a different input location
AsyncAP	Proposed scheme detailed in Section 3.3
AsyncAPOpt	AsyncAP with tuned parameters for better performance

Evaluated Schemes. Table 8 summarizes the evaluated schemes. We compare our schemes with two synchronous execution schemes proposed by prior work – GPU-NFA [31]: Sync-HS (HotStart [31]) and Sync-HC (NT-MaC-ACP [31]). The latter supports the flexible placement of hot and cold states to threads. We enable Sync-HC to leverage symbol-level parallelism by starting each thread at a different input location (namely ScaleOut-HC). We use VASim [60] running in multi-thread mode as the CPU baseline. Programs running on P6000 GPU are compiled with `nvcc` 11.0 with `-O3 -arch=sm_61` and the programs running on RTX 3090 GPU are compiled with `nvcc` 11.7 with `-O3 -arch=sm_86`.

Performance tuning for AsyncAPOpt. To optimize AsyncAP, we sweep the parameters of the two tuning knobs introduced in Section 3.3. First, as discussed in Section 3.3, part (K elements) of thread-local worklist can be stored in shared memory to make room in caches for other data structures. However, a large K may reduce the occupancy as shared memory is limited. Figure 7 depicts the

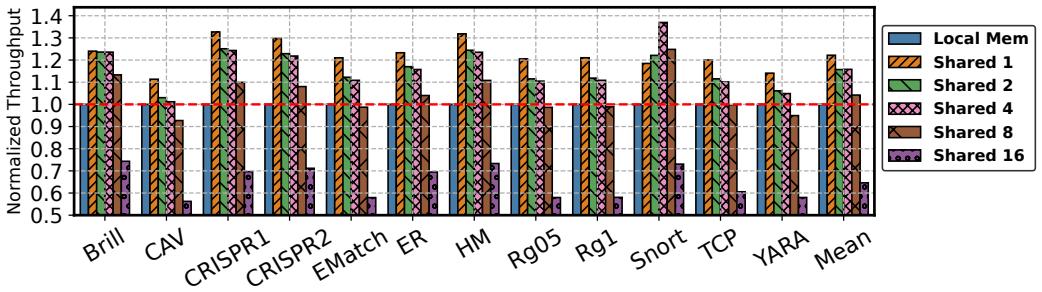


Fig. 7. Performance sensitivity to the implementation of per-thread-worklist under high parallelism scenario. The throughput is normalized to the local memory-only implementation. "Shared K" in the legend indicates that K elements of the worklist are in shared memory while the rest is stored in local memory.

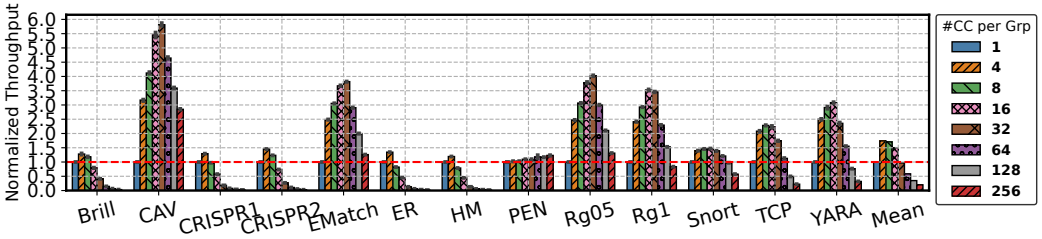


Fig. 8. Performance sensitivity to the number of NFAs (connected components, CCs) per kernel launch under high parallelism scenario

performance results by varying Ks normalized to AsyncAP with all elements of worklists in local memory ($K = 0$). While still application-dependent, we observe that the best average performance is achieved with $K = 1$. Therefore, we use $K = 1$ in AsyncAPOpt. Second, we tune the number of NFAs (M) for each kernel launch. In AsyncAP, we use $M = 1$ (i.e., "one-automaton-one-kernel"). A small M leads to a small transition table at the cost of scanning the input streams multiple times. Figure 8 shows that despite being application-dependent, $M = 4$ achieves the best average performance. Consequently, we set $M = 4$ for AsyncAPOpt.

6 EXPERIMENTAL RESULTS

6.1 Results

Table 9 shows the throughput of evaluated applications in evaluated scenarios. To simplify the discussion, we exclude the application, PEN, from this section and discuss it separately in Section 6.2. Figure 9 shows the performance results. We observe that *the evaluated GPU implementations outperform multi-core CPU significantly, especially when the parallelism is sufficient.*

Low and Medium Parallelism. In this scenario, the approaches of GPU-NFA only use one thread block to execute the NFAs on an input stream, which severely underutilized GPU cores. Figure 9 (a) shows the throughput obtained from this scenario in log-scale. We found that AsyncAP achieves 26.4× speedup for the evaluated applications on average and AsyncAPOpt achieves 57.9× speedup. Figure 9 (b) shows that AsyncAP achieves 4.4× speedup on average in the medium parallelism scenario while the tuned version AsyncAPOpt achieves 9.4× speedup. For all evaluated applications except PEN, AsyncAP achieves significant speedup showing its effectiveness in increasing the

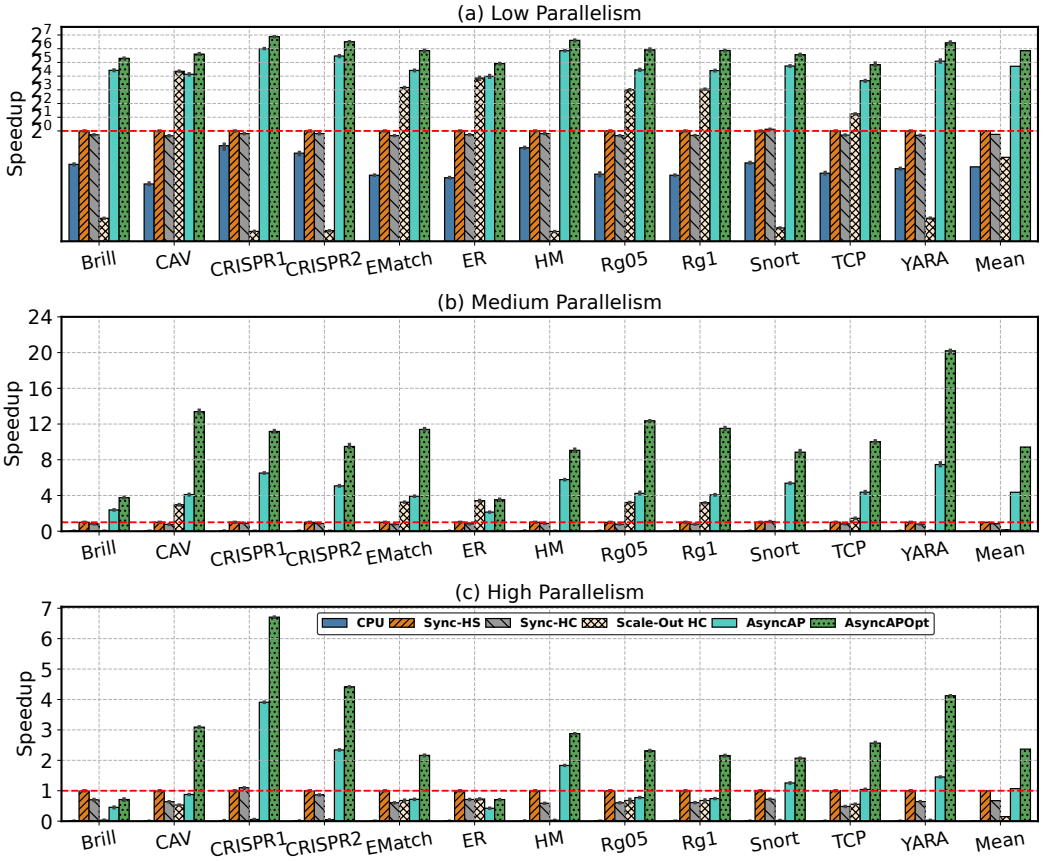


Fig. 9. Performance of synchronous and asynchronous automata executions on GPU under a different amount of parallelism

parallelism of automata processing tasks. *The results confirm that AsyncAP provides high scalability for parallelism-limited tasks.*

High Parallelism. We evaluate AsyncAP to confirm whether AsyncAP provides high throughput compared with GPU-NFA when the GPU is saturated by high-parallelism tasks. Figure 9 (c) shows our results. Overall, we observe that *AsyncAP achieves 1.07× speedup on average compared to Sync-HS, the fastest implementation of GPU-NFA across the 12 evaluated applications.* Further, the throughput of the optimized AsyncAPOpt is 2.4× of GPU-NFA on average. *The results confirm that AsyncAP provides high throughput when GPU is saturated.*

AsyncAPOpt achieves up to 6.7× speedup over GPU-NFA (CRISPR1). Although `CRISPR1`, `CRISPR2`, `HM`, and `Snort` have more total work in AsyncAP (up to 1.91× total work of GPU-NFA), AsyncAP outperforms GPU-NFA in these applications significantly due to lower per symbol overhead. Albeit `Snort` is not a very balanced application (Table 5), coarse-grained thread block load balance can schedule multiple short blocks to compensate for the long blocks. To further study the reason, we observe that AsyncAP works more efficiently for the applications that have more activation or have longer *average* pattern length (demonstrated in Table 4) because of the lower latency

Table 9. Throughput (in MB/s) of evaluated applications under the evaluated scenarios

Parallelism	Low		Medium		High	
Scheme / App.	AsyncAPOpt	Sync-HS	AsyncAPOpt	Sync-HS	AsyncAPOpt	Sync-HS
Brill	**22.5**	0.6	**32.2**	8.6	35.1	**49.4**
CAV	**78.1**	1.6	**322.7**	24.1	**458.5**	148.3
CRISPR1	**22.9**	0.2	**32.5**	2.9	**34.5**	5.1
CRISPR2	**32.1**	0.4	**50.2**	5.3	**54.8**	12.4
EMatch	**64.8**	1.1	**192.7**	16.9	**246.5**	113.9
ER	**39.4**	1.3	**68.5**	19.5	77.8	**110.0**
HM	**20.1**	0.2	**27.7**	3.1	**29.2**	10.1
PEN	0.6	**0.8**	0.5	**11.5**	3.7	**83.8**
Rg05	**63.5**	1.1	**196.9**	15.9	**249.8**	108.1
Rg1	**63.5**	1.1	**189.5**	16.5	**239.0**	110.8
Snort	**26.0**	0.5	**72.8**	8.2	**86.9**	42.0
TCP	**26.2**	0.9	**138.1**	13.8	**211.7**	82.5
YARA	**66.9**	0.8	**234.9**	11.6	**308.3**	74.8

of state matching. In contrast, when the pattern is long (i.e., states are more frequently activated), the overhead to maintain worklists and synchronize of GPU-NFA leads to poor performance.

In comparison, AsyncAP has a slowdown in a few applications for two reasons: 1) For example, even executed with AsyncAPOpt, `Brill` and `ER` incur 1.40× and 1.41× slowdown, because AsyncAP requires more weighted total work (2.1× and 2.4×, respectively; see Table 5). 2) Applications, such as `EMatch`, `Rg05`, `Rg1`, do not need more work in AsyncAP though, have a slowdown because their states are matched rarely, therefore, they are more favorable to be executed by GPU-NFA as it optimizes for skimming over the input streams by the *always-active* starting states. On the other hand, the optimized version AsyncAPOpt does not have a slowdown for these applications.

Scaled Out GPU-NFA (ScaleOut-HC). Figure 9 includes the results of ScaleOut-HC. We observe that ScaleOut-HC is on average slower in all scenarios because ScaleOut-HC is not able to reuse the hot nodes mapped to threads and has a higher overhead in synchronization between every contiguous symbol. Nevertheless, by leveraging symbol-level parallelism, ScaleOut-HC achieved speedup in a few applications under low and medium parallelism scenarios.

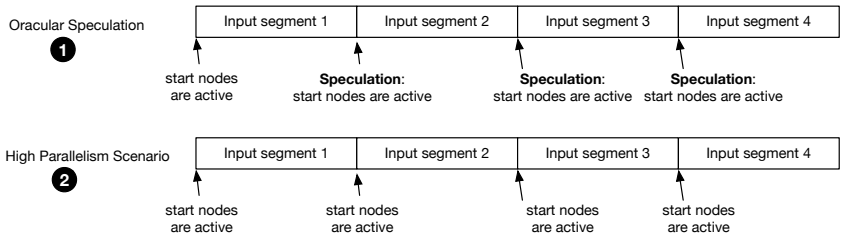


Fig. 10. Oracular Speculation vs High Parallelism Scenario

Comparison with Oracular Speculation. A speculation mechanism for automata processing needs two components: 1) speculation component that decides the active state at the beginning of each input stream segment; 2) matching component that matches all the input segments starting from the speculated states. Since no speculation scheme is implemented on GPU for NFAs, we perform

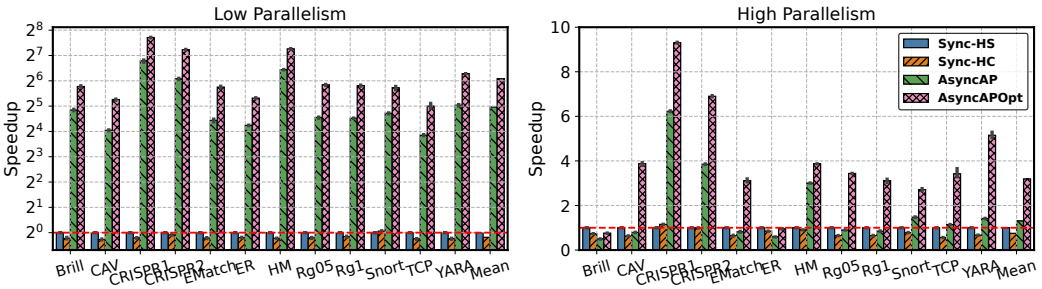


Fig. 11. Performance Sensitivity to Ampere GPU Architecture

an oracular case study where the speculation component is oracular and use GPU-NFA, the state-of-the-art scheme on GPU, as the matching component. Figure 10 (❶) depicts that a speculation approach partitions the input stream into 4 segments. The speculation component decides the set of active states at the beginning of segments 2, 3, and 4. Suppose in this case, it correctly speculates that only the always-active starting states are active, and hence no re-execution is needed. The matching component, therefore, processes the 4 segments in parallel. In comparison, Figure 10 (❷) depicts the high parallelism scenario in which GPU-NFA is processing 4 input streams. Equivalently, GPU-NFA here acts like the matching component that processes input segments. Therefore, the measured throughput could serve as a proxy to study what is the upper bound performance when GPU-NFA works as the matching component. Figure 9 (c) is a proxy comparison between AsyncAP and the oracular speculation. In conclusion, AsyncAPOpt obtains 2.4× speedup over the *oracular speculation approach* with GPU-NFA matching component.

Sensitivity to Ampere Architecture. We evaluate *high parallelism* and *low parallelism* scenarios on NVIDIA RTX 3090 GPU. Since RTX 3090 has 82 SMs, we use 656 (82×8) input streams in the high parallelism scenario. Figure 11 demonstrates the performance results. In the *low parallelism* scenario, on average, AsyncAPOpt achieves 67.7× speedup (y-axis is in log-scale) over Sync-HS. 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 by AsyncAP. For example, with AsyncAPOpt, `CRISPR1` achieves 209.4× speedup on RTX 3090 compared to 117.4× speedup on P6000. We also observe that AsyncAPOpt has better performance improvement compared to AsyncAP in the RTX 3090 GPU, indicating that the two tuned parameters are also portable to Ampere GPUs. In the high parallelism scenario, AsyncAP and AsyncAPOpt achieves 1.3× and 3.2× speedup compared to GPU-NFA on average across the 12 evaluated applications (`PEN` is excluded). The applications with slowdown in P6000 GPU (e.g., `EMatch`, `ER`, `Brill`) also perform better as Ampere architecture has less memory latency and consequently, GPU-NFA slightly loses its edge. *We conclude that AsyncAP also provides high throughput and scalability on the Ampere Architecture.*

6.2 Analysis of Pattern Length and Discussion

Slowdown of PEN. Compared with Sync-HS (the fastest version of GPU-NFA), Table 9 shows that AsyncAPOpt incurs 22.7×, 23.0×, and 1.3× slowdown in the high, medium, and low parallelism scenarios for `PEN`[1]. The slowdown is because the matching process of `PEN` gets stalled at certain states with self-loops.

We further examine the patterns of applications to understand the slowdown. Table 7 shows the length of patterns in the evaluated applications. We observe that most applications' patterns are

[1]As suggested by prior work [61], PowerEN (`PEN`) is an arbitrary benchmark that may not reflect real applications.

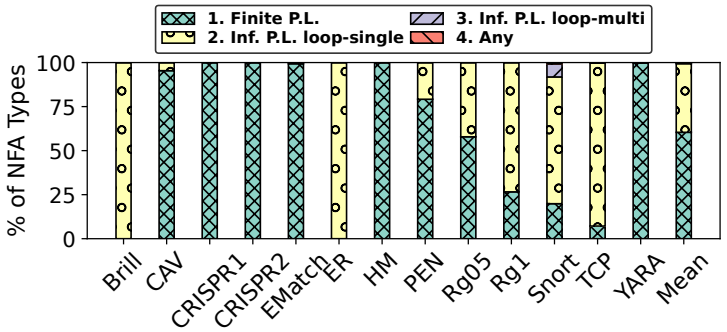


Fig. 12. Large portion of NFAs only generate patterns with limited length.

short with low variance. Applications with longer average pattern length (e.g., CRISPR1, CRISPR2, HM) tend to perform better in AsyncAP because long patterns imply 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 large amount of data movement overhead and higher per symbol latency (Table 5). For a few applications can have very long patterns (e.g., PEN, Snort), since their pattern length has distinct standard deviations (142.4 and 6.8, respectively), they have very 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: For example, the imbalance ratio of Snort is 5.6k, which is far less than a kernel that has 31k warps for a 1MB input. In contrast, the imbalance ratio of PEN is too large (480k) to be offset by the thread block scheduler, and hence the performance drops significantly.

In summary, AsyncAP performs well in all scenarios when the length of patterns is moderate but can lead to slowdown due to imbalance when the patterns are extremely long and the standard deviation of pattern length is also high.

How likely are very long patterns? To study how common the long patterns are, we classify the NFAs into four types by static analysis on NFA graphs: 1) The first type only generates patterns with finite length (the NFA graphs are directed acyclic graphs). 2) The second type can generate patterns with infinite length only when the matching process is stalled in the same state (the NFA graph has no back edge but has self-loops). 3) The third type can generate patterns with infinite length when going through a cycle containing more than one state (the NFA graph has no state with a self-loop but has back edges). 4) The fourth type can generate any length of patterns (the NFA graph has back edges and self-loops).

Figure 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 NFAs can only generate patterns with finite length*. In other applications, although statically it is possible to have very long patterns, they do not have long patterns in real input streams (Table 7).

Performance sensitivity to pattern length cutoff. A simple way to reduce the slowdown caused by extremely long patterns is to cutoff pattern length at a predefined value which each thread terminates when it reaches instead of terminating when the pattern finishes. Figure 13 portrays results of varying pattern length cutoff. We observe that when the pattern length cutoff doubles starting from 1k, the performance degrades slightly due to the coarse-grained load balance performed by GPU's thread block scheduler. However, when the cutoff reaches 32K, the performance degrades more for each step, showing that the imbalance could not be compensated well. We also confirmed that although the pattern length is very high in PEN, these long patterns (matching processes) do not

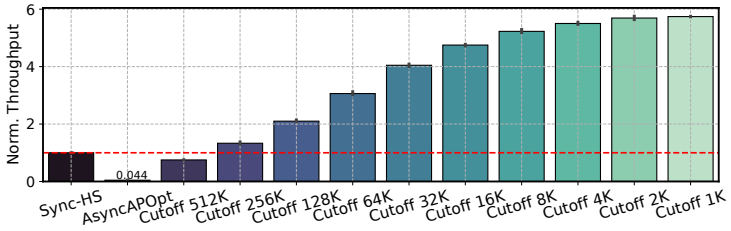


Fig. 13. Performance results of PEN with various pattern length cutoff points under high parallelism scenario

generate reports. However, cutting off the matching process by pattern length may omit reports, leading to correctness issues. Therefore, the users may need to provide insights about how long the patterns are in their applications.

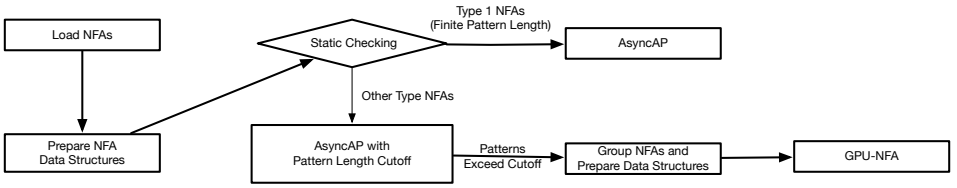


Fig. 14. Workflow to process automata with GPU-NFA to avoid extreme slowdown

Discussion on Workflow with GPU-NFA. We discuss how AsyncAP could work with GPU-NFA to avoid the slowdown due to extremely imbalanced patterns. Figure 14 demonstrates a potential workflow. After loading NFAs and preparing their data structures, we statically check whether these NFAs can lead to long patterns by NFA topology analysis (Figure 12). As a large portion of NFAs will never have long patterns, we offload them to AsyncAP due to better throughput and scalability. Other types of NFAs are opportunistically offloaded to AsyncAP but with a pattern length cutoff. A large cutoff threshold brings more opportunity to run NFAs on AsyncAP while a small cutoff threshold has less chance to suffer from slowdown. For instance, Figure 13 shows that if we limit the cutoff to 1k, AsyncAP has around 5.8× throughput than GPU-NFA. When it fallbacks to GPU-NFA, the overhead would be 17.2% without considering the data structures preparation. If any of the NFAs exceed the cutoff, they will be collected on the CPU and offloaded to GPU-NFA again. We leave this complete workflow for future work.

7 RELATED WORK

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

7.1 Mapping Automata to Compute Resources

To parallelize the automata processing, the program needs to map automata to compute resources. Vasiliadis et al. [57] and Gregex [64] use GPU threads to handle different network packets (i.e., input stream level parallelism). Smith et al. [51] merge DFAs (and extended DFAs) with a 64MB memory budget and execute them in batches on GPU threads. Tran et al. [56] use each warp to handle an NFA and an input stream. HyperScan [65] maps NFA states to SIMD lanes of CPUs and uses parallel bit operations of SIMD to calculate the next states of NFA. Prior work by Vu [58] maps threads to

state vectors and performs vector and matrix multiplication to leverage the bit-parallelism of NFAs. iNFAnt [18] and Nourian et al. [34] map state transitions to threads. Nourian et al. also propose compiler-based schemes to map the traversals of NFAs to the code of GPUs [35] or FPGAs [36], but these approaches are limited to fixed-topology NFAs.

These works treat all states equally, neglecting their matching activity. Zu et al. [76] statically group the states that can never be grouped into compatible groups, and map the compatible groups to threads to increase the GPU utilization. GPU-NFA [31] classifies states as 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, this mapping increases the parallelism.

7.2 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 [29], enumeration [33], speculation [38, 40, 74, 75], or their hybrids [26, 67]. Speculation is more work-efficient than prefix-sum parallelization and enumeration [75]. All of these works assume the compute resources are more than the parallelism provided by the original task (i.e., automata level, input stream level, and state-level parallelism).

All the aforementioned works focus on DFAs because DFA always has one active state at every step, making it easier for speculation to decide the active states and enumeration to merge paths [33]. A recent work [66] implements the idea of speculation for DFAs on GPUs. To reduce the overhead of misspeculation, it proposes techniques to reduce the redundant work caused by speculation. Since it does not support NFAs yet, we cannot compare AsyncAP with it directly. Nevertheless, we compared AsyncAP with an oracular speculation approach that uses GPU-NFA as matching processes in Section 6 and found that AsyncAP achieves significant speedup over the *oracular* speculation approach.

Traditional enumeration schemes enumerate the active states at the beginning of each segment. The matching processes need to be synchronized because they need to discard the incorrect results from the enumeration of active states. Although the separation of execution paths of AsyncAP can also be viewed as an enumeration of input locations, AsyncAP does not enumerate active states and hence needs neither validation nor discarding incorrect results. As a result, AsyncAP is more efficient as it fully utilizes the massively parallel processor GPUs regardless of the amount of parallelism. The limitation of AsyncAP is that it only applies to *all-input* starting states.

No prior work adapts to tasks with a different amount of parallelism, whereas AsyncAP achieves both scalability and high throughput for all evaluated tasks.

8 CONCLUSIONS

With each generation, GPUs are more capable and equipped with more resources. However, not all automata tasks have enough parallelism to fully utilize these GPUs.

We proposed a lightweight approach to increase the parallelism of automata processing on GPUs by searching for patterns in parallel *asynchronously* in the input stream. While it requires additional work theoretically, our study revealed that in real applications this approach performs less work than expected.

Experimental results confirm that our approach provides both high throughput and scalability. The increased parallelism enabled by our approach reduces the under-utilization of GPU cores and leads to significant speedup when original task parallelism does not saturate the GPU. Even when the GPU is saturated, our approach performs considerably faster than the state-of-the-art NFA processing engine on GPU.

ACKNOWLEDGMENTS

We thank our shepherd, Y.C. Tay, and anonymous reviewers for their detailed feedback which significantly improved the quality of this paper. This material is based upon work supported by the National Science Foundation (NSF) grant (#1750667). This work was performed in part using computing facilities (including Quadro P6000 donated by NVIDIA Corp.) at William & Mary.

REFERENCES

[1] 2018. ANML Documentation. http://www.micronautomata.com/documentation/anml_documentation/c_D480_design_notes.html.
[2] 2019. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.
[3] 2019. The Zeek Network Security Monitor. https://www.zeek.org.
[4] 2021. CUDA Occupancy Calculator. https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls.
[5] 2021. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.
[6] 2021. NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler.
[7] 2021. Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/.
[8] 2022. NVIDIA AMPERE GA102 GPU ARCHITECTURE. https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.
[9] M. Karim Abdalla et al. 2013. Scheduling and Execution of Compute Tasks. US 2013/0185728A1.
[10] 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).
[11] 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).
[12] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*.
[13] Michela Becchi and Patrick Crowley. 2007. An Improved Algorithm to Accelerate Regular Expression Evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*.
[14] 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)*.
[15] 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)*.
[16] 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)*.
[17] 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)*.
[18] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. 2010. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Computer Communication Review (CCR)* (2010).
[19] 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).
[20] Amr S. Elhelw and Sreepathi Pai. 2020. Horus: A Modular GPU Emulator Framework. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
[21] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.
[22] Adi Fuchs and David Wentzlaff. 2019. The Accelerator Wall: Limits of Chip Specialization. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*.
[23] Victor Mikhaylovich Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1.
[24] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2004. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Transactions on Database Systems* (2004).
[25] Bingsheng He, Qiong Luo, and Byron Choi. 2006. Cache-conscious Automata for XML filtering. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 18, 12 (2006), 1629–1644.

[26] 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)*.

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

[28] 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)*.

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

[30] 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)*.

[31] 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)*.

[32] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Policy-based tuning for performance portability and library co-optimization. In *Proceedings of the 2012 Innovative Parallel Computing (InPar)*. 1–10.

[33] 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)*.

[34] 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)*.

[35] 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)*.

[36] Marziyeh Nourian, Mostafa Eghbali Zarch, and Michela Becchi. 2020. 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)*.

[37] Junqiao Qiu, Lin Jiang, and Zhijia Zhao. 2020. 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)*.

[38] 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)*.

[39] 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)*.

[40] 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)*.

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

[42] Bin Ren, Tomi Poutanen, Todd Mytkowicz, Wolfram Schulte, Gagan Agrawal, and James R. Larus. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

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

[44] 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)*.

[45] 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)*.

[46] 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)*.

[47] 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).

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

[49] 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)*.

[50] 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)*.

[51] 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)*.

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

[53] 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)*.

[54] 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]

[55] 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)*.

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

[57] 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)*.

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

[59] 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)*.

[60] 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.

[61] 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)*.

[62] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, Jack Wadden, Mircea Stan, and Kevin Skadron. 2016. An Overview of Micron's Automata Processor. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*.

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

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

[65] 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)*.

[66] 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)*.

[67] 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)*.

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

[69] Yi-Hua Yang and Viktor Prasanna. 2012. High-Performance and Compact Architecture for Regular Expression Matching on FPGA. *IEEE Trans. Comput.* (2012).

[70] 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)*.

[71] 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)*.
[72] Xiaodong Yu, Wu-chun Feng, Danfeng Yao, and Michela Becchi. 2016. 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)*.
[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] 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)*.

Received October 2022; revised December 2022; accepted January 2023