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

Hongyuan Liu
hliu08@email.wm.edu
William & Mary
Williamsburg, VA

Sreepathi Pai
sree@cs.rochester.edu
University of Rochester
Rochester, NY

Adwait Jog
ajog@wm.edu
William & Mary
Williamsburg, VA

Abstract

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

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

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378471

Keywords Finite State Machine; GPU; Parallel Computing

ACM Reference Format:
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 '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378471

1 Introduction

Finite automata are the workhorses of pattern matching, data analytics, malware detection, bio-informatics, and XML parsing among many other applications [1, 2, 4–6, 12, 13, 33–35, 39, 40, 45, 64, 65]. Two representations of finite automata—non-deterministic finite automata (NFAs) and deterministic finite automata (DFAs)—are commonly used in the implementation of finite automata based applications [16]. Although DFAs are simpler in terms of transitions, DFA execution is embarrassingly serial, and DFAs can be exponentially larger than equivalent NFAs [40, 59, 63]. Prior work [21, 25, 30, 31, 62, 63] significantly reduces the latency of DFA execution by parallelizing chunks of the input stream and resolving the dependencies across states. However, current enumeration or speculation mechanisms increase parallelism which is not always needed, especially for large-scale automata applications.

The non-deterministic nature of NFAs lends itself naturally to parallel execution leading to a number of NFA accelerators [18, 22, 36–38, 42, 43, 66]. In particular, the Automata Processor (AP) proposed by Micron [17] is an in-memory accelerator for NFAs. The AP achieves significant throughput and energy benefits because of its ability to perform in-memory computations that exploit the parallelism of NFAs [17, 54]. However, APs have to deal with several challenges. First, APs can hold a limited number of NFA states at a time and when executing large-scale workloads need repeated re-executions and re-configurations that hamper throughput significantly [24]. Second, their multiple-instruction single data (MISD) model means their ability to execute multiple input streams in parallel is limited.

On the other hand, GPUs are massively parallel accelerators that are widely used. Execution of NFAs on

highly parallel architectures like GPUs, therefore, appears very attractive. However, NFA-based applications are very hard to accelerate on traditional von Neumann architectures (e.g., CPUs and GPUs) [17, 27, 32, 44]. In this paper, we address this hard problem with the help of a careful analysis of the bottlenecks of NFA execution on GPUs. Specifically, we find that there are two main problems. First, a typical NFA execution incurs significant data movement because for processing each input byte, a large transition table stored in the global memory needs to be looked up. Solutions to reduce this data movement invariably use a fixed mapping of NFA states to GPU threads, which leads to the second problem—hardware under-utilization. Many NFA states are not active (cold) in a given NFA, so a fixed mapping leads to idle threads. These idle threads unnecessarily consume GPU resources and *do not* perform any *useful* work leading to low throughput and poor hardware utilization. Overall, high data movement and poor hardware utilization are the major sources of inefficiencies.

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

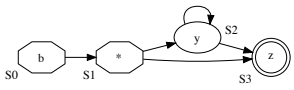
- This paper analyzes the bottlenecks of NFA execution on GPUs and finds that high data movement and low utilization are the two major inefficiencies leading to low throughput.
- We find that the data movement problem is due to the irregular accesses to the transition table. This table is too large to fit in the on-chip resources due to significant redundancy and sparsity in the transition table. Our solution stores the topology and matchset information in a novel compact format such that it can be stored and accessed from the on-chip resources for most NFA states.
- We find that utilization is low because not all states are active. Hence, we take advantage of state activity information to assign one thread per hot state and other cold states are executed on-demand. This improves utilization as a result of increased activity of threads.
- Overall, our mechanisms outperform the state-of-the-art work in this area. Specifically, across 16 NFA applications, the best of our schemes improves the throughput on average by 26.5× over iNFAnt [14] and 5.3× over NFA-CG [67]. Further, we only require 0.7% of the global memory transactions used by iNFAnt.

2 Background

This section describes NFAs and their processing on GPUs.

2.1 Pattern Matching via NFAs

A non-deterministic finite automaton[1] (NFA) is a directed graph where each node represents a state and each edge represents a state transition. Every state in the NFA has a *matchset* that contains the alphabets (symbols) it *matches*. Every NFA has at least one *start state* and at least one *reporting state*. The matching process begins by activating the *start states*. An NFA consumes one symbol at a time from the input stream. For each symbol, all currently *active* states attempt to match the incoming symbol with their matchset. If any of them match, they activate their successors. Unlike deterministic finite automata, where only one state is active, NFAs can have multiple states active simultaneously—making them ideal for parallel architectures. If a *reporting state* matches an input symbol, it generates a report showing that a relevant pattern has been observed in the input stream. Usually, all starting states are *always-active*, unless a user wants to search patterns that only start at a certain position of the input stream.



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

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

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

Figure 1. Working example of an NFA.

For example, Figure 1 (a) shows an NFA accepting pattern `b*.y*z`. It has two start states S0 and S1, and a reporting state S3. Suppose S0 and S1 are active and the incoming symbol is `x`. Since S1 matches `x`, its successors S2 and S3 become active. S0 and S1 are always-active, so in the next step, the active states are S0, S1, S2, S3.

[1] In this paper, we focus on Glushkov NFAs [19]. They are ϵ-free and the matchset is on the node instead of on the edge. Any NFA that accepts a non-empty string can be transformed into an equivalent Glushkov NFA [19].

2.2 NFA Processing on GPUs

GPUs support concurrent execution of a large number of threads and also have very high memory bandwidth – orders of magnitude more than CPUs. NFAs are a good fit for GPUs because they exhibit parallelism at multiple levels [27]. Consider the NFA processing mechanism as shown in Algorithm 1. First, multiple input streams (e.g., different network packets) can be processed in parallel (Line 2). Second, many NFAs (e.g., different intrusion signatures) can run in parallel on the same input stream (e.g., a single network packet, Line 3). Finally, within the same NFA and the same input symbol, multiple states can be active at the same time (Line 8). Therefore, there are multiple sources of parallelism: (1) input stream level parallelism, (2) NFA-level parallelism, and (3) state-level parallelism.

Algorithm 1 Parallelism in NFA Processing

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

A General Approach for NFA Processing on GPUs. Given that NFAs must process each input symbol serially (Line 7 in Algorithm 1), and that most GPUs do not support a cheap global barrier, it is natural to map an entire NFA to a single *thread block*, which is a group of threads that can execute a hardware barrier. This hardware barrier can be used to step through the input stream. An application usually contains many NFAs (Table 2) with different sizes. However, all thread blocks are of the same size, so actual implementations will pack multiple NFAs into the same thread block forgoing some NFA-level parallelism. A naïve implementation would then map an individual state to a thread in the thread block. Each thread block then maintains two bitsets, one showing which states are active in the current step and another identifying those that will be active in the next step. When an active state matches the current input symbol, its successors are set in the next active bitset (Line 13-14).

A *transition table* lookup combines state match and fetching successors. Prior work [7, 14, 15, 46, 48, 49, 56, 67], for example, use variants of a transition table where each row is indexed by a symbol α and each column is indexed by a state S—an *alphabet-oriented transition table*. Each entry (or cell) of the table contains the successors of S when S matches with α. If S is a reporting state and matches with α, a report is generated.

In our example, Figure 1 (b) shows an example of the transition table lookup for the NFA shown in Figure 1 (a). Assume the incoming symbol is `x` (Figure 1 (b)), and the current active states are S0 and S1. The threads assigned to the two states, therefore, fetch the two shaded cells from the transition table. States S2 and S3 are set to the next active bitset along with the always-active states (S0, S1 in this NFA). When all states have processed the current input symbol, we synchronize the threads using a `__syncthreads()` barrier, swap the current and next active bitsets, and reset the next active bitset (Line 15-17).

3 Problem and Previous Efforts

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

3.1 Data Movement

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

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

3.2 Compute Utilization

As nodes in an NFA are only activated based on the sequence of input symbols, many states in an NFA are never or very rarely activated [24]. Therefore, having a one-to-one mapping between states to threads means several

inactive states would waste thread resources leading to poor utilization and throughput. Across all the evaluated applications (Section 6), we observe that only a small percentage of states are active during the execution. The average and maximum percentages of active states are 0.39% and 3.05%, respectively. Although this percentage still implies hundreds to thousands of active states—more parallelism than a CPU could handle—a large fraction of GPU threads are still idle.

It is instructive to examine this problem from the perspective of graph processing algorithms. This style of NFA processing would be classified as topology-driven [26], which is known to be work-inefficient. In those algorithms, therefore, a worklist containing the frontier of active vertices is maintained. The threads are mapped only to the active vertices, thus utilization is 100% since threads do only useful work. To verify the feasibility of a worklist approach, we use the IrGL compiler [29] to generate worklist versions for NFAs. Using a 1MB input stream, we found that in Brill, the best version achieves 114KB/s. By contrast, iNFAnt [14] achieves 143KB/s. Hence, applying the worklist directly for NFA processing on GPU is not efficient for two reasons. First, unlike graph processing, some states (i.e., nodes) are always active in NFA processing. Second, NFAs require synchronization after each input symbol and perform very little work per input symbol. Therefore, maintaining a worklist incurs high overhead and hence it is critical to have a lightweight and efficient way to increase utilization.

3.3 Limitations of Prior Efforts

iNFAnt [14] uses a variant of the alphabet-oriented transition table described earlier. The columns represent edges in the NFA, not states, and the rows continue to represent alphabet symbols. In this table, a cell can have at most one state. For example, an edge $u \rightarrow v$ that matches on symbol a creates a column for u, whose row for a contains v. During execution, each column (and therefore an edge (u, v)) is mapped to a single GPU thread. Each thread checks if its assigned state u is active in the active bitset, and activates v if the current input symbol is a. iNFAnt does not perform any special optimizations for data movement and utilization.

Zu et al. [67] introduce the notion of *compatible groups*, where the states that belong to the same compatible group cannot be active simultaneously. This allows a compatible group to be assigned to a single thread, improving utilization. However, this approach, which we name NFA-CG, uses a very expensive method to compute compatible groups – its time complexity is at least quadratic in the number of NFA states and other non-heuristic methods are exponential. Our proposal for improving utilization is linear in the number of states, and also achieves better utilization.

Ideally, the only memory loads should be for the input symbols. Consider a kernel launched with T thread blocks (each containing W warps) and the length of the input stream

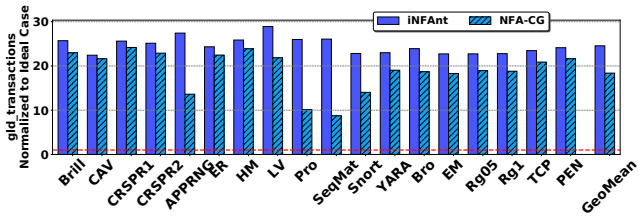


Figure 2. The data movement normalized to the ideal cases: two prior schemes use 25× and 18× compared to the ideal case where only the input stream is loaded. The evaluation methodology is discussed in Section 6.

in symbols is L. In such a case, the number of global memory load transactions for input only is TWL. Figure 2 shows the number of global load transactions for existing works iNFAnt [14] and NFA-CG [67] normalized to the ideal case for input-only memory transactions. On average, iNFAnt has 25× more transactions, and NFA-CG has 18× in the evaluated applications.

4 Addressing the Data Movement Problem via Matchset Analysis

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

4.1 Inefficiencies in the Transition Table

As discussed in Section 2, the existing transition table stores both the matchset and NFA topology information. Instead of checking whether the current input symbol is present in the matchset of the current state, it converts this computation to transition table lookups. However, we find that the resultant transition table can no longer fit in the GPU on-chip memory as storing the combination of matchset and NFA topology introduces redundancy and increases sparsity.

To understand and quantify the volume of redundancy and sparsity in the transition table, consider Figure 3 which shows two metrics. Redundancy is defined as the ratio of the total number of edges across all NFAs in the application to the total number of non-empty (or occupied) entries in the transition table. As the number of edges can only be less than (or equal to) the number of occupied entries, a low ratio indicates higher redundancy since an edge is stored in multiple locations of the transition table. Sparsity is defined as the ratio of occupied entries to the total number of entries in the transition table. A low ratio for sparsity suggests that not all transition table entries are occupied. We observe from Figure 3 that on average both metrics are very low across all the evaluated NFA applications showing that alphabet-oriented transition table wastes a lot of memory.

From our discussion in Section 2, we can identify two reasons that lead to these inefficiencies. First, an edge in

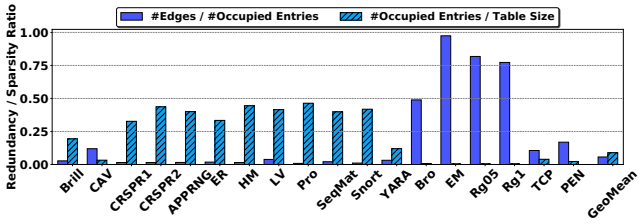


Figure 3. Two metrics showing the redundancy (#edges/#occupied-entries) and sparsity (#occupied-entries/table-size) in the transition table. Lower is worse.

the NFA can occur multiple times in the transition table. For example, all entries in column S1 in Figure 1 (b) store the same value (S2, S3) because S1 accepts a wildcard. In general, if a state accepts k symbols, all its outgoing edges have to be stored k times. Second, in most NFA applications, a large percentage of states only accept a few symbols. For example, the column for S0 only contains one entry as S0 only accepts b, but the entire column has to be kept in the transition table which makes it sparse. In conclusion, there is excessive redundancy and sparsity in the transition table. This makes it an inefficient way to store the matchset and topology information.

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

To reduce the excessive data movement problem incurred by the transition table, we propose a new way to store matchset and topology information. The key idea is to create a per-node data structure that contains the node's: a) matchset, b) outgoing edges, and c) other miscellaneous attributes. This per-node data structure is stored only *once* eliminating redundancy. Furthermore, we avoid storing the complete Cartesian product of alphabets and states in an off-chip transition table addressing the sparsity problem. Our proposal converts the per symbol look-ups of the transition table to a one-time memory access per state, which makes the data movement of our scheme close to the ideal case as shown in Section 3.3.

Per-node data structure. Figure 4 ❶ shows the per-node data structure `NodeInfo`. We use an array of eight 32-bit integers for the 256-bit matchset (`matchset`, 32 bytes). When symbol `a` is examined, each active thread checks for a match by checking if the bit corresponding to `a` is set: `matchset[a / 32] & (1 << (a % 32))`. Since NVIDIA GPUs do not support indexing into a register, the `matchset` must be stored in the local memory which is private to each thread. We will show in Section 4.3 how we also put the matchset in the registers when possible. We maintain 4 out edges per node in a 64-bit integer (`outedges`, 8 bytes), which consumes two 32-bit registers per thread. We also need to encode

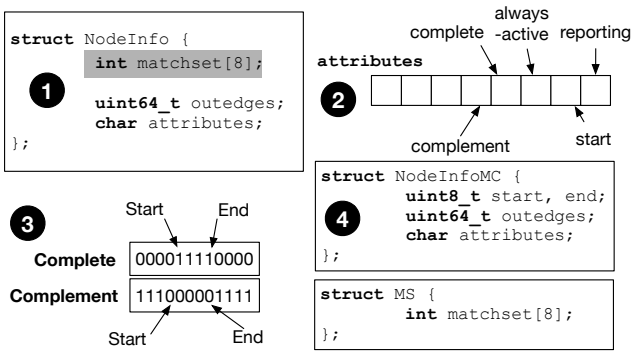


Figure 4. Illustrating the per-node data structure of NewTran (NT). Shaded variables are in the local memory and others are in the registers.

the attributes of a state (as shown in Figure 4 ❷), so we maintain these attributes in the 8-bit variable `attributes`, which consumes an additional register. Currently, we use 5 bits of the attribute variable. Three bits record if a state is a *reporting*, *start*, or *always-active* state. Two additional bits record if the *matchset* is *complete* or *complement* to enable the compression optimization described in Section 4.3.

Matching process. The per-node data structure is fixed mapped to each thread. Before processing the input symbols, each thread loads the per-node data structure from the global memory. After the data structure is loaded, the thread starts to iterate over the input stream. Instead of looking up the transition table for determining a match, each active state compares the incoming symbol against its matchset in the privatized `NodeInfo` data structure. We still keep the double-buffered *active bitset* to record whether a state is active as described in Section 2.2. We use an array in global memory to hold the reports generated during the NFA processing. GPU atomic instructions are used to perform concurrent writes to this array.

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

4.3 Optimization II: Matchset Compression (MaC)

Figure 4 shows that matchset information is stored in local memory. We find additional opportunity to compress this information to reduce the global memory transactions. To compress the matchset information, we focus on two categories of states:

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

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

When a state is either complete or complement, we can represent its entire matchset as a range using only two 8-bit variables, start and end (Figure 4 ❸) to denote the input symbols it matches. Then, for complete states, we can check if the incoming symbol s is matched simply by evaluating s ≥ start && s ≤ end. For complement states, we can simply invert the sense of the result. Thus, all accesses to the matchset can be eliminated by converting the indirect memory read on the transition table to a pure range check computation.

Figure 5 shows that a large portion of states are either complete or complement. On average, matchset lookups for 70% of states can be replaced by range checks.

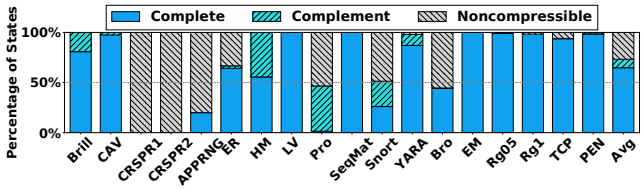


Figure 5. Percentage of states whose matchsets are complete, complement, or not compressible.

To implement the complete/complement compression scheme, we split the NodeInfo data structure into two structures as shown in Figure 4 ❹, namely NodeInfoMC and MS. If a state is compressible, we only load NodeInfoMC, which uses 16 bits (start and end) to store the matchset. If the state is not compressible, we load MS as before. Depending on whether a state is compressible or not, we load 16 bits or 272 (16 + 256) bits for the matchset. In general, if a fraction p of states are compressible, the average global load per node data structure is $16p + 272(1 - p)$. As the majority of the states are compressible (Figure 5), our matchset compression scheme uses fewer loads to the local memory while also reducing the global memory transactions.

5 Addressing the Utilization Problem via Activity Analysis

In this section, we focus on addressing the problem of under-utilization as discussed earlier in Section 3.2. We first analyze the activity of states and use this information for an intelligent mapping of states to threads. The key idea is to map only the highly active states to dedicated threads and assign resources to remaining low activity states on-demand.

5.1 Analysis of Activation Frequency

It follows from Section 2 that *always-active* start states do useful work during the entire execution. However, the activity of the other states is not clear. Figure 6 shows the CDF of the activation *frequency* of all the non-starting states across seven representative applications. All other evaluated applications are similar to the representative applications. We observe that for the majority of applications, 80% of non-starting states are activated for less than 1% of the processed symbols. Similar behavior was also observed by Liu et al. [24]. However, their study did not consider the frequency of the activity and only evaluated whether the state is active or never-active. Our finding is that although many states can be active at least once, the frequency of activation is usually very low.

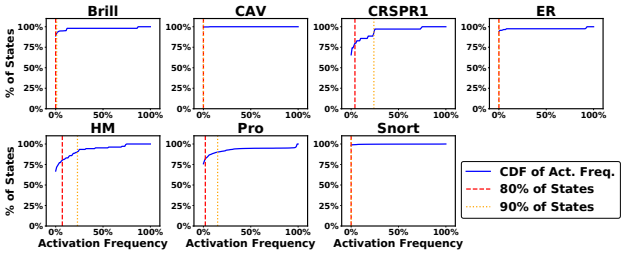


Figure 6. The activity profile of the states. For the majority of applications, 80% of non-starting states are activated for only less than 1% of the processed symbols.

We exploit this activity profile in GPUs and propose to map only the states that are activated frequently to the dedicated threads. Other infrequently activated states and are assigned resources on-demand. To accomplish this, we need to answer two research questions: (1) Given a certain mapping, how do we coordinate between fixed mapped *hot* (active) states and the on-demand loaded *cold* (rarely active) states? (2) How do we classify *hot* states and *cold* states? We will answer these two questions in Section 5.2 and Section 5.3, respectively.

5.2 Optimization III: Activity-based Processing

In this section, we discuss how we handle hot and cold states to improve the overall utilization. We propose to develop a hybrid approach that uses one-one mapping (topology-driven) for hot states and a worklist (data-driven) for the cold states. Each hot state is given to a dedicated thread, while cold states are not assigned to any dedicated threads. As in NT, each thread loads the NodeInfo[2] data structure for its hot state before processing the input stream. As described in Section 2.2, we store whether a hot state is active in the per-block *active bitset*. Each thread also reserves space for an

[2]This can be NodeInfoMC depending on whether we turn on the matchset compression (Section 4.3). We will use NodeInfo for the two cases.

additional NodeInfo data structure to be used for any cold states dynamically assigned to it during execution.

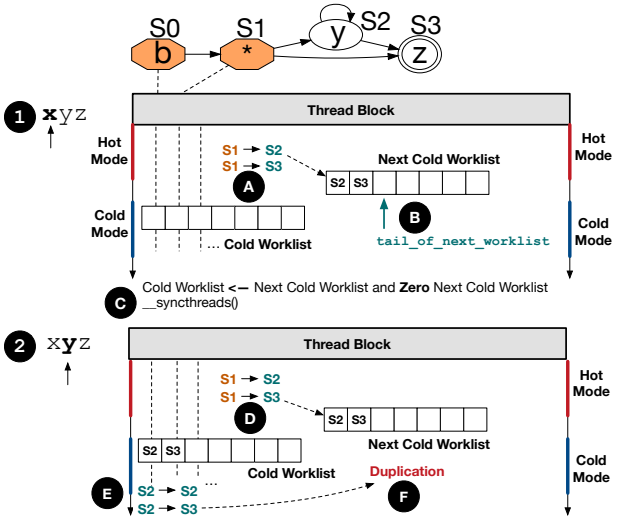
Execution for all input symbols takes place in two modes: first, a hot mode (which executes all hot states) followed by a cold mode (which executes any cold states that have been activated). If a hot state activates a cold state, it places the cold state ID in the *next cold worklist*. This worklist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

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

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

Illustrative Example. Figure 7a illustrates our activity-based optimizations using an example with a thread block. Assume that the hot states S0, S1 are mapped to the threads and the cold states S2, S3 are processed through the worklists. Figure 7a ❶ shows that currently, we are processing the symbol x of the input stream xyz, and S0 and S1 (both hot states) are active. Symbol x (Figure 7a ❶) triggers two hot-to-cold transitions (S1 to S2 and S1 to S3, Figure 7a Ⓐ). S2 and S3 are pushed to the next cold worklist, by atomically incrementing tail_of_next_worklist (Figure 7a Ⓑ). After the hot mode, the threads move to the cold mode to process the current cold worklist, which is empty. Since there is no work in this step, the cold mode does not start. In the end of the current step, the next cold worklist and the cold worklist are swapped and the tail pointer of the next cold worklist is reset to 0 (Figure 7a Ⓒ).

The processing of input symbol y then begins (Figure 7a ❷), with hot-to-cold transitions S1 to S2 and S1 to S3 in the hot mode (Figure 7a Ⓓ). They are pushed to the cold worklist. The *deduplication bitset* is also set. In the cold mode, two threads process the S2 and S3 that was pushed to the CW in the previous step when we processed x. Since S2 matches with y, it generates two cold-to-cold transitions (Figure 7a Ⓔ). However, by checking the worklist deduplication bitset (Figure 7a Ⓕ), S2 and S3 are not pushed to the worklist. After the cold mode, the threads in the thread block are synchronized and the next step of the



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

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

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

Figure 7. Illustrating the activity-based processing

execution begins with states S0, S1, S3 and S4 as active. Figure 7b summarizes the complete steps to process the xyz input stream.

5.3 How do we choose the hot states?

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

Profiling. The first scheme is based on prior work [24], which shows that the activation frequency of NFA states in a small representative input is similar to the entire input. In this evaluation, we use the 1KB prefix of the 1MB input as the profiling input. If a state has an activation frequency more than a threshold in the profiling input, we consider it as a hot state during the entire execution. In the experimental results, we use 10% as the threshold. We have also tested other thresholds but they do not affect our conclusion.

Offloading by BFS layers. Second, we consider a percentage of states (ordered by their BFS layers) as hot states. Users can control the percentage of states to be deemed as hot. The assignment of hot states is an iterative process. We

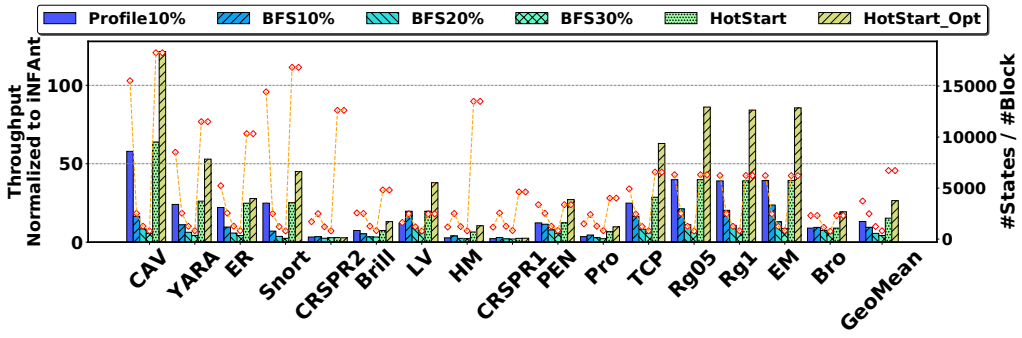


Figure 8. Throughput sensitivity to the selection of hot states. Detailed evaluation methodology is in Section 6. HotStart (or HotStart_Opt, an optimized version) has the best performance among these selection schemes. Hence, we choose the always-active start states as hot states.

sort the states by their BFS-layers and then mark them as hot states in the ascending order until the number of hot states reaches the percentage specified by the user. This scheme relies on the topology of the NFA and does not need any profiling information. In this experiment, we use 10%, 20%, and 30% as the percentage of the hot states.

Make start states hot—HotStart. Our third scheme only considers the *always-active* start states as hot states. It does not require profiling or tuning of parameters/thresholds. The scheme is based on the observation from Figure 6 that other than the always-active starting states, the activation frequency for most other states is very low.

Experimental decision. Figure 8 shows the normalized throughput and utilization numbers of the aforementioned three schemes. We make the following observations. First, we find HotStart gives the best performance among the evaluated configurations on average. Second, we find that the performance is correlated to the states per block, which works as a proxy to show the utilization of GPU. As HotStart has the most states per block, its utilization is the best among these cases. Third, for a few applications (e.g. CRSPR1, CRSPR2), HotStart does not give the best performance, but still gives comparable performance. This is because the activation frequency of non-starting states is high (Figure 6). Therefore, the loading of the node data structure to the worklist leads to more data movement. To summarize, HotStart gives us a simple and synergistic solution for both data movement and utilization, while achieving the best performance across the evaluated hot states selection schemes.

Elimination of Active Bitset. In HotStart, every activated state is in the worklist except the *always-active* starting states. Hence, we remove the *active bitset* from the thread block. By this simple optimization specific to HotStart, the register usage is reduced from 70 registers to 40 registers per thread, leading to an increase of occupancy. Figure 8 shows that this optimized version of HotStart (HotStart_Opt) gives 77% improvement and we use it in the rest of the experiments.

6 Evaluation Methodology

Evaluated Schemes. Table 1 summarizes the schemes that we evaluate in this paper. iNFAnt [14] and NFA-CG [67] are prior works in the area of NFA Processing in GPUs as discussed in Section 3.3. iNFAnt [14] maps each edge to a thread and NFA-CG [67] maps a compatible group (a group of states) to a thread. Next, we evaluate our schemes NT (Section 4.2), NT-MaC (Section 4.3), which dedicate each thread to each state. They only focus on reducing data movement. Then we evaluate HotStart and HotStart-MaC (Section 5), which are built using HotStart_Opt and work on top of NT and NT-MaC, respectively. They enhance utilization by mapping only *always-active start states* to the threads (Section 5.3) without the need of profiling. To demonstrate the effectiveness of data movement optimization, we also evaluate HotStartTT, which uses an alphabet-oriented transition table but also applies our utilization-related optimizations.

Table 1. Overview of the evaluated schemes on GPU

Scheme	Thread Mapping	Data Movement	Utilization
iNFAnt [14]	Edge	-	-
NFA-CG [67]	Compatible Group	-	Compatible Group
NT	State	Opt. I	-
NT-MaC	State	Opt. I + Opt. II	-
HotStart	Hot State	Opt. I	Opt. III
HotStart-MaC	Hot State	Opt. I + Opt. II	Opt. III
HotStartTT	Hot State	-	Opt. III

AP Performance Modeling. We also compare to the automata processor (AP) [17], a domain-specific architecture for NFA processing. Since AP is not publicly available, we use a simple, optimistic performance model influenced by VASim [52] for AP performance estimation. If we have n input streams each containing m symbols, each NFA must process $n \times m$ symbols. If AP can hold C NFA states, and if

the application has A NFA states, then $\lceil A/C \rceil$ batches are required. In one AP chip, C equals to 49152. As current AP chip is documented to run at 133MHz, a symbol needs 7.5 ns for processing. Thus, $Time_{AP\_ideal} = 7.5 \cdot mn \cdot \lceil A/C \rceil$. Since APs must be reconfigured between batches, we add per batch reconfiguration overhead of 50 ms [54] to $Time_{AP\_ideal}$ to obtain $Time_{AP}$. Both these models are *optimistic* because we ignore other overheads of AP, such as the time taken to record reports (i.e., matches). Earlier work has noted that this can be a significant overhead in several applications [50]. Note that all the performance numbers for GPUs *include* the report generation overhead.

Experimental Setup. We mainly use an NVIDIA Quadro P6000 GPU for evaluation. We also report the sensitivity of our results on NVIDIA Tesla V100. We report the GPU kernel time gathered using CUDA events. The throughput (our metric for performance) is measured in terms of number of input symbols processed per second. Each set of experiments is performed 7 times and we report 95% confidence intervals for our results (shown as error bars). Our results/conclusions are consistent across different runs. Usually, these error bars are too small to visualize since the widest CI bar is 0.36% of the normalized throughput bar. For a fair comparison with prior works [14, 67], our results do not include the I/O time and data structure preparation time as prior works do not focus on optimizing them. We expect that these overheads will be amortized over long GPU computation time and hence we focus on optimizing the latter.

Application Configurations. We evaluate 18 applications from three different benchmark suites: AutomataZoo [53], Regex [10] and ANMLZoo [51]. Table 2 shows the characteristics of the evaluated applications. All these applications have sufficient parallelism in terms of number of states per NFA, number of NFAs (also called as connected components [CC]). The total number of states can be in the order of millions (Table 2). We use 1MB input (split into 1000 1KB input streams) for each application (except APPRNG and SeqMat) to provide parallelism for input streams. Two applications (APPRNG and SeqMat) do not have *always-active* start states, so we cannot feed the 1KB chunks of input to them. Hence, we evaluate them separately in Section 7.

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

Out-degree is limited to 4. Different states can have different out degrees leading to load imbalance in amount of work per thread. Like NFA-CG [67], we modify the NFAs so that the outgoing edges of each state is 4 or less using an iterative algorithm. We split each state that has an out-degree greater than 4 into two with each getting half of the original edges, and connect the duplicates to the predecessors and successors to maintain the semantics of the NFA. This process repeats until all states have 4 or fewer out edges. For some graphs, this process does not terminate (e.g., a complete graph where each node has out-degree of 5). If this process does not terminate in N steps (where N is the number of states), the NFA is filtered out. There are 2 NFAs in ClamAV and 5 NFAs in Snort that cannot be limited to states with out-degree ≤ 4 and hence are discarded.

7 Experimental Results

Figure 9 shows the performance results of our schemes and the performance achieved by AP normalized to iNFAnt. The y-axis is in log scale. Table 3 shows the raw throughput achieved by our evaluated schemes. On average, HotStart-MaC gives 26.5× speedup and HotStart gives 20.5× speedup, which achieves the best and the second-best performance among all schemes. They also achieve the best performance on all applications except CRSPR1, CRSPR2 and Pro, where HotStartTT is the fastest. Furthermore, HotStart-MaC and HotStart are 5.3x and 4.7x faster than NFA-CG, respectively.

Analysis of cases where HotStart-MaC is the best. HotStart-MaC performs the best in YARA, ER, Snort, Brill, and HM among all GPU schemes (Table 3). Specifically, in these applications, HotStart-MaC achieves significant speedup — at least 27% improvement in Snort and up to 373% improvement in HM compared to HotStart. There are two reasons. First, in these five applications, at least 50% of their states are compressible. Second, all of them have a large number of states. Their per-node data structures are too large to fit into the L1 cache of the GPU, especially when many states are handled through the worklist, reducing the size of per-node data structures yields significant improvement.

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

Table 2. Characteristics of evaluated NFA applications.

Application		State Info					Connected Component (CC) Info		
Name	Abbr.	#states	#start	#always-active	#reporting	#compressible	#CC	max_CC_size	avg_CC_size
Brill [53]	Brill	115549	5.1%	5.1%	5.1%	100.0%	5946	40	19.4
ClamAV [53]	CAV	2374717	1.4%	1.4%	1.4%	100.0%	33171	22075	71.6
CRISPR_CasOFFinder [53]	CRSPR1	74000	5.4%	5.4%	2.7%	0.0%	2000	37	37.0
CRISPR_CasOT [53]	CRSPR2	202000	2.0%	2.0%	1.0%	0.0%	2000	101	101.0
APPRNG_4sided [53]	APPRNG1	20000	5.0%	0.0%	20.0%	20.0%	1000	20	20.0
EntityResolution [53]	ER	413352	2.4%	2.4%	2.4%	66.7%	10000	75	41.3
Hamming_l18d3 [53]	HM	108000	1.9%	1.9%	1.9%	100.0%	1000	108	108.0
Levenshtein_l19d3 [53]	LV	109000	3.7%	3.7%	3.7%	100.0%	1000	109	109.0
Protomata [53]	Pro	24103	5.4%	5.4%	5.5%	46.6%	1309	123	18.4
SeqMatch_w6p6 [53]	SeqMat	51570	20.0%	0.0%	3.3%	100.0%	1719	30	30.0
Snort [53]	Snort	202043	1.6%	1.2%	1.6%	51.4%	2486	4509	81.3
YARA [53]	YARA	1047528	2.2%	2.2%	2.3%	98.0%	23530	1017	44.5
Bro217 [10]	Bro	2312	8.1%	8.1%	8.1%	44.6%	187	84	12.4
ExactMath [10]	EM	12439	2.4%	2.4%	2.4%	100.0%	297	87	41.9
Ranges05 [10]	Rg05	12621	2.4%	2.4%	2.4%	99.0%	299	94	42.2
Ranges1 [10]	Rg1	12464	2.4%	2.4%	2.4%	98.3%	297	96	42.0
TCP [10]	TCP	19704	3.8%	3.8%	3.9%	94.0%	738	391	26.7
PowerEN [51]	PEN	40513	7.1%	7.1%	8.5%	99.7%	2857	52	14.2

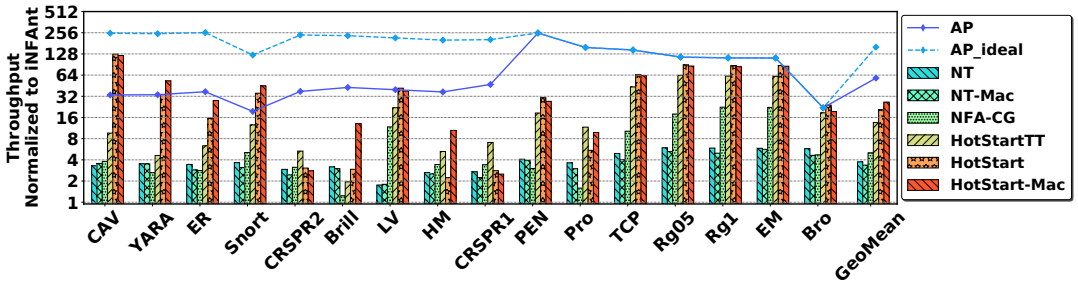


Figure 9. Throughput enhancement results normalized to iNFAnt. On average HotStart-MaC achieves 26.5× speedup across 16 applications. The best GPU results outperform an AP chip in 5 applications (CAV, YARA, Snort, LV, and Bro).

Table 3. Absolute throughput with our schemes (MB/s). The best performance among GPU schemes is highlighted.

Arch	config	CAV	YARA	ER	Snort	CRSPR2	Brill	LV	HM	CRSPR1	PEN	Pro	TCP	Rg05	Rg1	EM	Bro
AP	AP_ideal	2.72	6.06	14.81	26.67	26.67	44.44	44.44	44.44	66.67	133.33	133.33	133.33	133.33	133.33	133.33	133.33
	AP	0.36	0.82	2.14	4.21	4.21	8.16	8.16	8.16	15.38	133.33	133.33	133.33	133.33	133.33	133.33	133.33
GPU	iNFAnt [14]	0.01	0.02	0.06	0.22	0.11	0.19	0.21	0.22	0.33	0.53	0.84	0.92	1.16	1.19	1.19	6.07
	NT	0.04	0.09	0.20	0.79	0.33	0.61	0.36	0.58	0.88	2.15	3.06	4.51	6.86	6.97	6.96	35.01
	NT-Mac	0.04	0.09	0.17	0.67	0.27	0.57	0.37	0.56	0.73	2.07	2.53	3.71	6.05	5.90	6.72	28.16
	NFA-CG [67]	0.04	0.06	0.16	1.10	0.35	0.24	2.42	0.76	1.11	1.58	1.34	9.36	20.70	26.57	26.53	28.57
	HotStartTT	0.10	0.11	0.36	2.71	**0.60**	0.37	4.53	1.16	**2.31**	9.73	**9.84**	39.83	73.41	73.55	73.32	112.94
	HotStart	**1.38**	0.82	0.90	7.63	0.34	0.56	**8.59**	0.49	0.92	**16.06**	4.61	**59.54**	**103.58**	**102.84**	**104.47**	**145.88**
	HotStart-Mac	1.31	**1.29**	**1.60**	**9.70**	0.31	**2.50**	7.80	**2.32**	0.82	14.26	8.26	57.66	99.54	99.85	101.95	117.58

out from the worklist, and hence the benefit of matchset compression is small.

The performance impact of increasing utilization alone. Even though HotStartTT does not use NT for data movement optimization, it is 14x faster than iNFAnt. Compared to NFA-CG that also only has utilization optimization (based on statically computed compatible groups), our HotStartTT is 2.7x faster since compatible groups do not capture the notion of activity. HotStartTT is also the fastest scheme in CRSPR1, CRSPR2, and Pro, since at most 46% of states in these applications are compressible. In addition, as their states are activated frequently (Figure 6), the swap-in and swap-out in the worklist of HotStart and HotStart-MaC incur more data movement than HotStartTT.

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

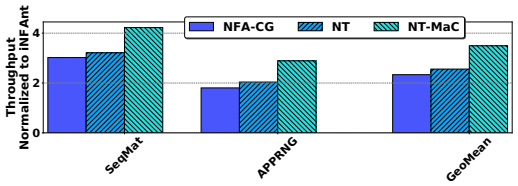


Figure 10. Throughput enhancement for the applications without always-active *start states in the single input stream scenario. Our schemes outperform NFA-CG and iNFAnt by at least 9% and 2.6×, respectively.*

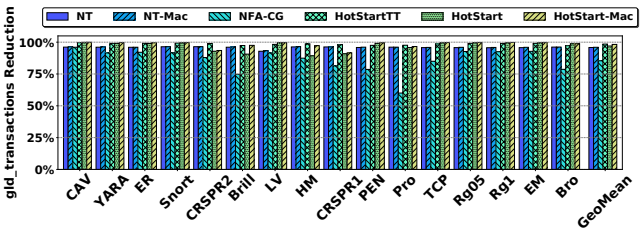


Figure 11. Effect on data movement reduction: our schemes use significantly fewer gld_transactions than prior work. For example, HotStart-MaC reduces gld_transactions by 99.3% over iNFAnt.

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

Evaluation of SeqMat and APPRNG. Two applications (SeqMat, and APPRNG) do not have *always-active* start states. Therefore, we do not evaluate them using HotStart in the multiple input streams scenario. Instead, we use a 1MB input stream to evaluate these applications. Figure 10 shows the performance of these applications. We compare our data movement optimization schemes NT and NT-MaC with iNFAnt and NFA-CG. In these two applications, we found that on average, NT and NT-MaC have 2.6x and 3.5x speedup over iNFAnt respectively. Our NT and NT-MaC schemes also show 9% and 50% improvement over NFA-CG.

How far are we from AP? In CAV, YARA, Snort, LV, our best GPU scheme outperforms the domain-specific AP chip by 3.8×, 1.6×, 2.3×, 1.1× (Table 3), because these applications have a large number of states requiring repeated re-configurations and re-executions on AP. Bro also has 1.1× speedup than AP. All other applications except Pro also perform within 10× of the AP performance as they make good use of the GPU resources, where Pro performs 12.54× worse than AP.

Effect on Data Movement Figure 11 shows the percentage of reduced global load transactions compared to iNFAnt.

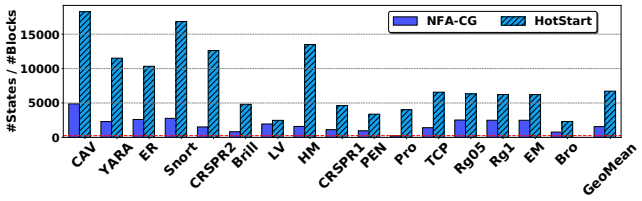


Figure 12. Effect on the number of NFA states per thread block (a proxy for compute utilization). More states are handled per thread block in HotStart.

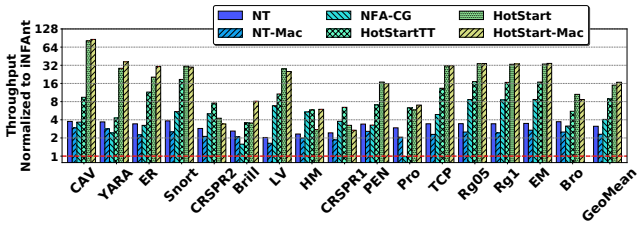


Figure 13. Performance sensitivity to Volta GPU Architecture. Both HotStart-MaC and HotStart show more than 15× speedup over iNFAnt, indicating their effectiveness on newer GPU architectures.

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

Effect on Utilization. We use the number of NFA states per thread block as a metric for evaluating utilization. Figure 12 shows this metric for the evaluated schemes. As we do not change the amount of work per thread, mapping more states per block implies better utilization.

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

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

by leveraging our insight into activation frequency (Figure 6) our HotStart can improve utilization even for Pro.

Sensitivity to Volta Architecture. We also evaluated our mechanisms on NVIDIA V100 GPU [3] and the results are shown in Figure 13. We observe a similar trend as what has been shown in Figure 9—our schemes still give significant speedup. Specifically, on average HotStart-MaC and HotStart give 16.7× and 15.0× speedup over iNFAnt, respectively. HotStartTT gives 8.9× speedup over iNFAnt. Given that the Volta has larger L1 caches compared to Pascal GPU, the magnitude of speedup we achieve is lower but still very significant indicating that our data movement and utilization optimizations are effective on newer architectures as well.

8 Related Work

There is a large body of work on pattern matching on CPUs [57], network processors [11], and custom accelerators coupled to CPUs [20]. We recommend interested readers to an excellent survey paper [58] for a broad overview of the field. Here, we restrict ourselves to GPU/SIMD implementations of pattern matching using finite automata.

Reducing Data Movement. DFA-using engines [7, 47, 49, 55, 61] try first to reduce the size of the state transition tables using compression [9, 60, 61], which is often necessary to fit the DFAs in GPU global memory. However, as DFAs are serial, they are mapped to a single thread whose on-chip resources cannot accommodate the footprint of individual DFAs. Alphabet reduction [23, 56] reduces the size of the symbol set by merging behaviorally-equivalent symbols and introducing an indirection table, however, it is ineffective on large DFAs [61]. Our matchset compression technique is orthogonal to alphabet reduction and exploits the sparsity patterns in the matchsets.

To reduce cost of memory accesses, input symbols were loaded to shared memory [7, 55], input data layout was changed [55] to avoid uncoalesced accesses, or vector loads were used [49]. Others used k-stride NFAs [8], which consume k bytes at a time, but can blow up the alphabet to $|\Sigma|^k$. Some implementations place state transition tables in texture memory [49] or on-chip constant memory [7]. Prior work [41] explored packet signature matching on GPU using DFA and XFA [40] and though that work suggested profiling hot XFA states and storing them in on-chip shared memory, it did not implement it as the size of on-chip shared memory was too small. Both textures and on-chip constant memory are limited in size, so large parts of the state transition tables remain in and are accessed using global memory.

If NFA topology is fixed, memory use can be reduced by embedding the topology of the NFA in the code [28] or inferring it from pattern-specific data [44]. The matchsets continue to be stored in global memory allowing different matchsets to be loaded at runtime. However, this technique is most suited for fixed-topology NFAs.

Improving Utilization. NFAs are compact in size, but their non-deterministic parallel execution requires that each thread handle a single transition [14] or a single state (this work). To improve utilization, more states must be mapped to a single thread. In previous works [61, 67], clusters of states called *compatible groups* are created that contain states that cannot be active at the same time [67] or that are likely to be active at the same time [61]. Compatible groups are mapped to the same thread [67] (for improving utilization) or are used to limit the lookups needed [61] (for decreasing work). The compatible groups of Zu et al. [67] improve utilization but are computationally expensive to compute. In contrast, our schemes construct a subset of NFA nodes that are mapped to threads at linear cost while achieving greater utilization than their compatible groups. Compared to Yu et al. [61], we separate topology and symbol/matchset information to limit the lookups without having to compute their notion of compatible groups.

9 Conclusions

In this paper, we proposed and evaluated three optimizations to significantly improve the throughput of NFA Processing on GPUs. These optimizations address sub-optimal data movement and low utilization, which stem from primarily two aspects: a) the matchset and topology information is stored off-chip and accessed in an irregular fashion, b) not all the NFA states are active all the time but still consume resources leading to GPU under-utilization. Our first two optimizations focus on the data movement problem and allow the needed matchset and topology information to remain on-chip as much as possible. Our third optimization identifies and maps only active states to dedicated threads while less active states are processed on-demand using a worklist-based approach. Overall, we achieve significant improvement in NFA processing throughput over the state-of-the-art mechanisms across a wide range of emerging applications. Moreover, our optimizations enable GPUs to outperform the domain-specific accelerator (AP) for several applications while being within an order of magnitude of AP performance for the remainder. As a part of our future work, we plan to close this remaining gap with the help of hardware/software co-design optimizations.

Acknowledgments

The authors thank the anonymous reviewers and members of the Insight Computer Architecture Lab at William & Mary for their feedback. This material is based upon work supported by the National Science Foundation (NSF) grants (#1657336 and #1750667). This work was performed in part using computing facilities at William & Mary.

References

[1] 2018. Clamav net. https://www.clamav.net/.

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

[5] 2019. The Zeek Network Security Monitor. https://www.zeek.org.

[6] 2019. YARA: The pattern matching swiss knife for malware researchers. https://virustotal.github.io/yara/.

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

[8] 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).

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

[10] 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)*.

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

[12] 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)*.

[13] Chunkun Bo, Ke Wang, Jeffrey J. Fox, and Kevin Skadron. 2016. Entity Resolution Acceleration using the Automata Processor. In *Proceedings of the International Conference on Big Data (BigData)*.

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

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

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

[17] 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).

[18] 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)*.

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

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

[21] 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)*.

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

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

[24] 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)*.

[25] 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)*.

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

[27] 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)*.

[28] 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)*.

[29] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

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

[31] 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)*. ACM.

[32] 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)*.

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

[34] 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)*.

[35] 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)*.

[36] 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)*.

[37] 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).

[38] 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)*.

[39] 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)*.

[40] 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)*.

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

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

[43] 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)*.

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

[45] 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)*.

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

[47] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference (ATC)*.

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

[49] 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)*.

[50] 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)*.

[51] 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)*.

[52] 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.

[53] 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)*.

[54] 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)*.

[55] 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*.

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

[57] 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)*.

[58] 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).

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

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

[61] 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)*.

[62] 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)*.

[63] 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)*.

[64] 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)*.

[65] Keira Zhou, Jack Wadden, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Regular Expression Acceleration on the Micron Automata Processor: Brill Tagging as a Case Study. In *Proceedings of the International Conference on Big Data (BigData)*.

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

[67] 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)*.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code and Python/shell scripts, which are expected to reproduce the results of Table 3. We also provide our raw experimental data and the scripts to plot other figures of our paper.

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

- **Algorithm: iNFAnt [14], NFA-CG [67], NT, NT-MaC, HotStart, HotStart-MaC, HotStartTT**
- **Program: CUDA and C/C++ code**
- **Compilation: We use `cmake` to build the project. Specifically, we use GCC 6.5.0 and NVCC 9.2 with the `-O3` flag**
- **Binary: CUDA executables**
- **Datasets: NFAs and input streams from AutomataZoo [53], ANMLZoo [51], and Regex [10] benchmark suites.**
- **Run-time environment: Ubuntu 18.04 with CUDA and GPU Computing SDK installed.**
- **Hardware: CUDA-enabled GPU. We have tested our project on NVIDIA Quadro P6000 and NVIDIA Tesla V100**
- **Output: Achieved throughput (byte per second) in text files that can be used for generating figures.**
- **Experiment workflow: Clone repository, setup environment, run scripts, review generated results.**
- **How much disk space required (approximately)?: 10GB**
- **Publicly available?: Yes**

A.3 Description

A.3.1 Source Code

The source code of our work is hosted in Zenodo and GitHub. The GitHub repository has the most updated version.

`https://github.com/bigwater/gpunfa-artifact`
`https://doi.org/10.5281/zenodo.3560474`

A.3.2 Hardware dependencies

We developed and tested our work on two NVIDIA GPUs (Quadro P6000 and Tesla V100). We expect that our code can run on GPUs with the compute capability no less than 5.0.

A.3.3 Software dependencies

Our work requires CUDA 9.2 SDK. Our work uses cmake 3.13 for compilation. We use Python 3.7 for our Python scripts. Our Python scripts require `matplotlib`, `numpy`, `pandas`, `scipy`, `xlrd` libraries. If you use `conda`, simply run the following bash commands to ensure we have the same Python environment.

```
conda create --name gpunfa_env python=3.7  \
  matplotlib numpy scipy pandas xlrd && \
  conda activate gpunfa_env
```

A.3.4 Datasets

All datasets are from public available benchmark suites. We convert their automata files to ANML format [51]. We provide the dataset that is ready to use in our repository.

A.4 Installation

Setup the project. The `setup.sh` will automatically unzip the datasets, build the executables, and set up environmental variables.

```
git clone https://github.com/bigwater/\
gpunfa-artifact.git &&
cd gpunfa-artifact &&
source setup.sh
```

Our project contains three executables: `infant`, `ppopp12`, and `obat`. They are added to your PATH variable after the setup. The former two executables are our implementations of iNFAnt and NFA-CG, respectively. The `obat` contains our schemes. They share the same options and arguments settings. Take `obat` as an example, you can check how to use it by showing the help using `obat -?` or `obat -h`.

Sanity check. We provide a small NFA and an input stream to verify that the binaries are successfully built. For example, to check NT in the small dataset:

```
cd small_dataset
obat -i inputstream -a apple.anml -g obat2
```

The `apple.anml` automata reports (in `report.txt`) ending positions for pattern `apple` in `inputstream`: 5, 47, 64, and 75, from state `__45__`.

A.5 Experiment workflow

```
cd ${GPUNFA_ROOT} &&
 ./run_experiments_get_table3.sh
```

The entire set requires several hours to finish. It uses the same configuration as the paper and generates Table 3 automatically.

A.6 Evaluation and expected result

Generating Table 3. A CSV file `abs_throughput.csv` (Table 3) will be generated after finishing the previous experiments.

Generating figures from raw data. The following script plots all the figures using our raw data.

```
cd ${GPUNFA_ROOT}
./gen_figures.sh
```

This script needs around 20 minutes to finish. All the figures will be generated in `${GPUNFA_ROOT}/raw_data/figures` folder.