GPU-based NFA Implementation for Memory Efficient High Speed Regular Expression Matching [*]

Yuan Zu   Ming Yang   Zhonghu Xu   Lin Wang   Xin Tian   Kunyang Peng   Qunfeng Dong [†]

Institute of Networked Systems (IONS) & School of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, P. R. China

wyn@mail.ustc.edu.cn   yangm@ustc.edu.cn   {xzhh,xiaquhet,tianxin,pengkuny}@mail.ustc.edu.cn   qunfeng@ustc.edu.cn

Abstract

Regular expression pattern matching is the foundation and core engine of many network functions, such as network intrusion detection, worm detection, traffic analysis, web applications and so on. DFA-based solutions suffer exponentially exploding state space and cannot be remedied without sacrificing matching speed. Given this scalability problem of DFA-based methods, there has been increasing interest in NFA-based methods for memory efficient regular expression matching. To achieve high matching speed using NFA, it requires potentially massive parallel processing, and hence represents an ideal programming task on Graphic Processor Unit (GPU). Based on in-depth understanding of NFA properties as well as GPU architecture, we propose effective methods for fitting NFAs into GPU architecture through proper data structure and parallel programming design, so that GPU's parallel processing power can be better utilized to achieve high speed regular expression matching. Experiment results demonstrate that, compared with the existing GPU-based NFA implementation method [9], our proposed methods can boost matching speed by 29∼46 times, consistently yielding above 10Gbps matching speed on NVIDIA GTX-460 GPU. Meanwhile, our design only needs a small amount of memory space, growing exponentially more slowly than DFA size. These results make our design an effective solution for memory efficient high speed regular expression matching, and clearly demonstrate the power and potential of GPU as a platform for memory efficient high speed regular expression matching.

[*] This work was supported in part by the Ministry of Education (MOE) Program for New Century Excellent Talents (NCET) in University, by the Science and Technological Fund of Anhui Province for Outstanding Youth under Grant No. 10040606Y05, by the Fundamental Research Funds for the Central Universities under Grant No. WK0110000007 and Grant No. WK0110000019, by the National Natural Science Foundation of China under Grant No. 61073184, and by Jiangsu Provincial Science Foundation under Grant No. BK2011360.

[†] All correspondence should be directed to Prof. Qunfeng Dong (Email: `qunfeng@ustc.edu.cn`).

Categories and Subject Descriptors C.2.0 [*Computer-Communication Networks*]: General—Security and protection (e.g., firewalls); D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; F.1.1 [*Computation by Abstract Devices*]: Models of Computation—Automata (e.g., finite, push-down, resource-bounded)

General Terms Algorithm, Design, Experimentation, Performance, Security

Keywords CUDA, Deep Packet Inspection, GPU, NFA, Pattern Matching, Regular Expression Matching

1. Introduction

Regular expression pattern matching is the foundation and core engine of many network functions, such as intrusion detection, worm detection, traffic analysis, web applications and so on. For instance, known worm signatures can be each formulated as a regular expression pattern; if such a regular expression pattern finds a match in network traffic, an alert of detecting the corresponding worm can be generated for taking actions accordingly. Besides, regular expression matching has also found a broad range of applications in other technological fields such as web, database, text processing, programming languages and so on.

Given the fundamental importance of regular expression matching, intensive research has been conducted in the past years, in order to obtain high speed and, no less importantly, memory efficient solutions. In spite of this long line of research, it is inherently hard to accommodate the two Genies — matching speed and storage space — into one jar. Specifically, regular expressions are matched using either deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA), each having its own merits and problems.

DFA is guaranteed to process each input character with one state lookup and transition, as it has precisely one single active state at any time; this enables DFA to provide fast and stable matching speed. However, this processing efficiency is achieved at the cost of exponentially exploding storage space; just a few regular expression patterns have been sufficient to generate a gigantic DFA containing hundreds of thousand states. With this exponential explosion, practical systems can hardly scale with even moderate size pattern sets; while in practice, real life systems (such as the Snort intrusion detection system [2]) have already deployed thousands of patterns, to be matched against network traffic flowing at link speeds (e.g. 10Gbps OC-192 link speed). To remedy this problem, numerous methods have been proposed for compressing DFA storage space. Although these research efforts have achieved impressive results [4–7, 10, 11, 16, 17, 19, 23–28], none has been able to deflate the exponential explosion of storage space while preserving the matching speed of original uncompressed DFA.[1]

Given this scalability problem of DFA-based methods, there has been increasing potential interest in NFA-based solutions for memory efficient regular expression matching. Unlike DFA, each NFA state can have multiple possible transitions on an input character, and NFA can have multiple states active simultaneously. This non-deterministic nature enables NFA to represent a pattern set with a much smaller state space, growing linearly instead of exponentially with pattern set size. Hence, NFA-based solutions are inherently memory efficient. However, as each NFA state can go on activating multiple other states, the NFA can have unpredictably many active states, on all of which state transitions have to be performed for an input character. Given that, to achieve high matching speed, it requires potentially massive parallel processing, and hence represents an ideal programming task on Graphic Processor Unit (GPU) [3, 12–15, 18, 22, 29, 30].

Recently, there has been a GPU-based NFA implementation method called *iNFAnt* [9] proposed by Cascarano et al. for memory efficient regular expression matching. While it is memory efficient like all NFA-based solutions, it has not been able to understand and exploit some important properties of NFA for potentially drastic performance boost, as we shall achieve in this work. Consequently, their method has only been able to match at a few hundred megabits per second, lagging far behind high speed link rates.

In this work, we shall analyze and demonstrate some important properties of NFA, using real life pattern sets as examples. Based on this understanding of NFA properties as well as GPU architecture, we shall conduct in-depth study, both experimental and analytical, of how NFAs can be best fitted into GPU architecture through proper data structure and parallel programming design, so that GPU's parallel processing power can be fully mobilized to achieve high speed regular expression matching. In particular, our study will proceed in three stages, each stage building upon the insights and design obtained in the preceding stage. In each stage, we shall figure out through experiments and analysis some key limitations of the preceding design, and then demonstrate how should we reform our design so that matching speed can be boosted significantly.

We evaluated the performance of our GPU-based NFA implementation design using real life pattern sets collected from the Snort intrusion detection system [2], on NVIDIA GTX-460 GPU. Experiment results demonstrate that, compared with iNFAnt [9], our GPU-based solution can boost matching speed by 29∼46 times, consistently yielding matching speed above 10Gbps. Meanwhile, compared with exponentially growing DFA state space, our NFA-based design only needs a very small amount of memory space, growing exponentially more slowly than DFA size. These results make our proposed design an effective solution for memory efficient high speed (e.g. 10Gbps OC-192 link speed) regular expression matching, and clearly demonstrate the power and potential of GPU as a platform for memory efficient high speed regular expression matching.

The rest of this paper is organized as follows. We start in Section 2 with an introduction of the existing GPU-based NFA implementation method [9] proposed by Cascarano et al., as well as relevant GPU architecture knowledge. Then in Section 3, the first stage of our study, we shall reveal through analysis the key drawback of that design, which motivates our basic design. We present this basic design and verify its effectiveness through experiment results. The limitation of this basic design will subsequently be analyzed and enhanced in the second stage of our study in Section 4, whose effectiveness will also be verified through experiment results. In the third stage of our study, we shall figure out the key issues of this design and culminate our study with the *Virtual NFA* design in Section 5. After evaluating our Virtual NFA design in Section 6, we conclude the paper in Section 7.

[1] In the literature, there are TCAM-based DFA deflation methods [20, 21] proposing to effectively deflate DFA state space while preserving one memory lookup per input character. However, TCAM is well known to take much more hardware expense, power consumption and chip area than RAM-based computing architectures, which is the focus of this work.

2. State of the art

The entire design of iNFAnt [9] is built upon three data structures (as shown in Figure 1): NFA transition table, current active state vector (CASV) and future active state vector (FASV). CASV is a bit vector where each bit corresponds to a distinct NFA state; if an NFA state is currently active, its corresponding bit in CASV is set, and is cleared otherwise. Similarly, FASV is such a bit vector indicating whether each NFA state will be active after performing relevant transitions on the current input character. A bit more complex is the NFA transition table, which is stored as 256 arrays for compressed storage space, each array consisting of the NFA's transitions on one of the 256 possible input characters. For example in Figure 1, the array corresponding to character `a` stores the NFA's transitions on character `a`.

As we have discussed in Section 1, NFA-based regular expression matching can be slow for each individual packet. To achieve high matching throughput, a bunch of packets are to be matched simultaneously, exploiting the massive parallel processing power of GPU. Each packet is handled by a separate matching process (consisting of a certain number of threads); the NFA transition table is to be shared by the matching processes of all packets, while the matching process of each packet has its own CASV and FASV.

Upon these data structures, the matching process of a packet is carried out by an exclusive set of threads; the number of threads is equal to the maximum number of NFA transitions in any of the 256 arrays, which is 34 in Figure 1. The entire matching process of a packet can be viewed as an iterative process; during each iteration, one input character is matched using the NFA. The process of matching an input character can be summarized as follows.

Step 1. Each thread uses the input character as an index to locate which of the 256 arrays (of transitions) to look up. The base address of the array is obtained.

Step 2. Within the set of threads for matching the same packet, each thread is assigned a unique thread ID (starting from zero), which is used by the thread as the offset plus the above obtained base address to get the corresponding NFA transition stored in that position in the array. For example in Figure 1, suppose there are 34 threads working to match a packet; the thread with thread ID 5 will obtain the sixth NFA transition stored in the transition array corresponding to input character `a`.

Step 3. The obtained transition is composed of a source state ID and a destination state ID, meaning if the source state is active, the destination state will be active after matching the input character. Hence, each thread will use the source state ID as an index into the CASV to find out if the source state is currently active. If yes, it uses the destination state ID as an index into the FASV to set the bit belonging to that destination state.

Step 4. After all transitions have been processed, FASV is copied into CASV, and then cleared for next input character.

In practice, the number of transitions stored in the 256 arrays can exceed the number of threads we use for each packet; multiple rounds of the above operations described in step 2 and step 3 can be conducted to process all transitions in an array. For example in Figure 1, there can be 34 transitions in an array. If we use one warp of 32 threads to process a packet, it simply takes two rounds of step

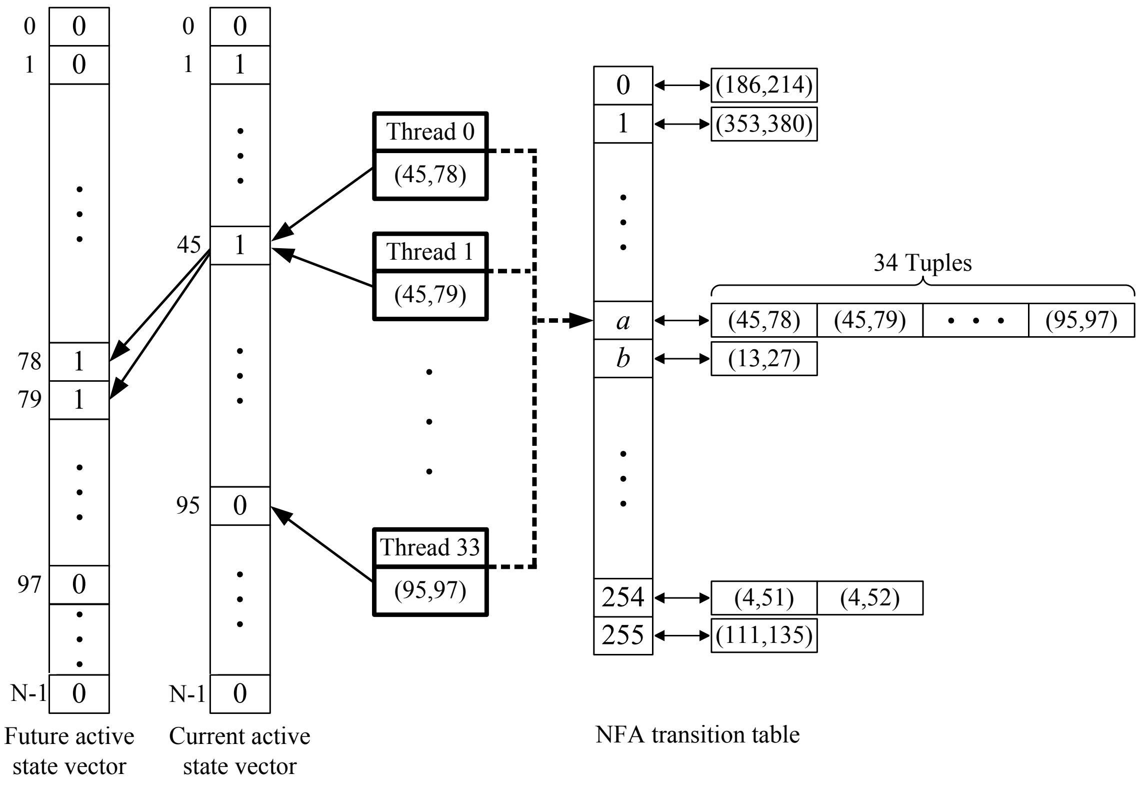

Figure 1. Data structures of iNFAnt.

2-3 operations to process these 34 transitions. In the sequel, we shall use a Snort pattern set consisting of 36 regular expression patterns (denoted by *Snort36*) as an example for illustration; its NFA can take up to five rounds of these operations to finish matching an input character.

We start our work with evaluating the performance of this GPU-based NFA implementation on NVIDIA GTX-460 GPU, using the *Snort36* pattern set. (Details about experiment setup are presented in Section 6.) The obtained matching speed is 0.26 Gbps, which is far below needed to keep up with today's high speed link rates (e.g. 10Gbps OC-192 link speed). In Section 3, we shall analyze the design drawbacks of iNFAnt, which motivate our basic design.

3. Basic design

The iNFAnt design has two major drawbacks. Firstly, the number of NFA transitions on an input character can be large, consuming a significant amount of computing resources. Especially, as GPU threads are allocated and launched in warps, each warp consisting of 32 threads, even more computing resources can be consumed. For example in Figure 1, there are 34 transitions on input character a; it will take 64 threads to process. (Actually, this is equivalent to letting one warp of 32 threads work on it for two rounds.) Secondly, notice that in step 3 of iNFAnt design, depending on whether the obtained source state is currently active or not, different threads may next execute different instructions. In current SIMD GPU architecture, such execution divergence will make the threads in one warp proceed in a sequentialized instead of parallel manner, resulting in severe performance degradation. (In Section 4.3, we shall present a systematic demonstration and analysis of how our proposed design eliminates divergence as well as potential conflict among concurrent threads.)

In light of the above insights, we now propose a different design that is immune to these problems. Our key motivating observation is that, while the number of NFA transitions on an input character can be large, the number of NFA states that can be active simultaneously is much smaller. For the NFA of pattern set *Snort36*, the number of transitions to be processed for an input character can be over five times larger than the maximum number of simultaneously active states. Therefore, while a large number of threads may be needed to process the transitions in iNFAnt design, it turns out that most of these threads will find its obtained source state inactive; their transition processing work is hence wasted. That said, if we could (somehow) accurately identify and locate those active NFA states, and make each thread responsible for performing transitions for one of the active states, the number of threads needed for matching an input character (and hence packet) can be greatly reduced, leading to significant boost in matching speed.

For that purpose, we maintain an *active state array* to record active states, as shown in Figure 2. Just for illustration purpose, let us say the array consists of 32 elements, each containing the state ID (Sid) of an active NFA state. A warp of 32 threads are dedicated for each packet, with the kth thread responsible for the kth element of the array. Upon receiving an input character, the kth thread checks the kth element for an active state ID. The state ID and the input character are combined together to form a two-dimensional index into the NFA transition table, which is essentially a two-dimensional array as shown in Figure 2, to obtain the NFA transitions to be performed.

Then, here comes the problem — into which element of the active state array should a thread write its obtained destination state as the new active state? This write operation has to be collision-free among the 32 threads. Otherwise, if two or more threads write their destination states into the same element and the destination states

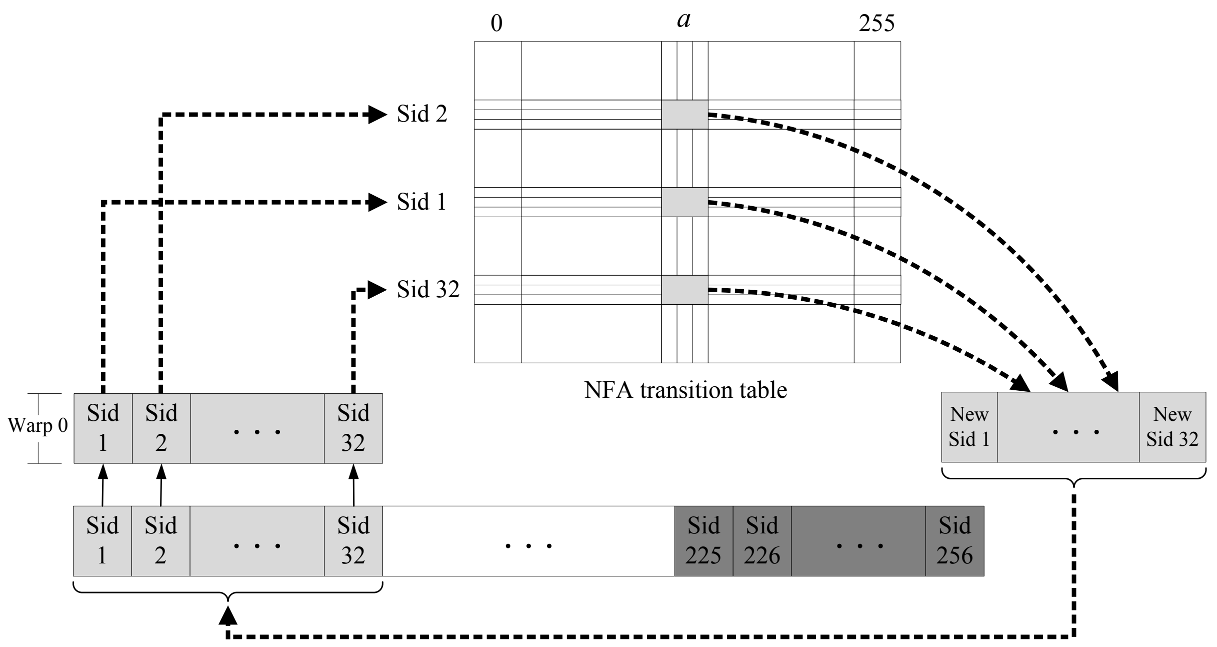

Figure 2. Data structures of our basic design.

are not the same, some of these new active states will be mistakenly overwritten by each other.

To solve this problem, we propose to partition NFA states into a number of *compatible groups*, such that states in the same compatible group can never be active simultaneously. Then, we can assign each element in the active state array to a distinct compatible group; different states in the same compatible group can safely share the same active state array element, since they cannot be active simultaneously and hence cannot be written to that element simultaneously. Every NFA state (as the destination state) can be safely written into the active state array element assigned to its compatible group, without worrying about collision. Because even if multiple threads are writing into the same active state array element, the destination state they are writing must be the same state; otherwise, such different destination states can be active simultaneously and should not have been in the same compatible group.

We mark every NFA state with its compatible group ID, which is equal to the index of the active state array element assigned to all the states in that compatible group. Each thread can simply write its obtained destination state into the active state array element indexed by the destination state's compatible group ID.

3.1 Compatibility between NFA states

For partitioning NFA states into compatible groups, we first need to figure out if two NFA states can be active simultaneously. One simple solution is to use a 2-dimensional incompatibility table. If state i and state j can be active simultaneously (which we call *incompatible*, meaning they cannot share the same active state array element), table entry (i, j) is set `true`; otherwise, we call them *compatible* and table entry (i, j) is set `false`. Using this incompatibility table, compatibility relationship between NFA states can be discovered with the following iterative breadth-first search algorithm.

Suppose the NFA states are numbered 0, 1, 2, . . ., n-1. At the beginning, we have n incompatible state pairs in the form of (i, i), meaning state i is incompatible with itself; these n state pairs are stored in a queue. Then, during each iteration of the algorithm, we take out the state pair (i, j) at the head of the queue. For every possible input character c, we find out the destination state set D_i for state i and the destination state set D_j for state j, respectively. All the states in $D_i \cup D_j$ can be active simultaneously, meaning they are incompatible. For every such state pair (i', j') in $D_i \cup D_j$, if table entry (i', j') is not `true`, we set it `true` and append (i', j') to the queue. The algorithm terminates when the queue becomes empty. During each iteration, one state pair is removed from the queue, while there are n^2 distinct state pairs in total and each state pair enters the queue no more than one time. Therefore, the algorithm runs for no more than n^2 iterations.

If two NFA states i and j are marked incompatible, they are incompatible, the algorithm has actually followed a string of input character(s) which can cause the NFA to transition from one certain state to both i and j. That means i and j can be active simultaneously and hence are truly incompatible.

If two NFA states i and j are not marked incompatible, they are not incompatible. To prove by contradiction, suppose states i and j can actually be active simultaneously, which means there exists at least one shortest string w of l input character(s) that can cause the NFA to transition from the start state q_0 to both i and j. Now consider a sequence $S_i = \{q_0, i_1, i_2, \ldots, i_l{=}i\}$ of states traversed by the NFA in processing this shortest string, starting from the start state q_0 and ending in state i. In parallel, there is also such a sequence $S_i = \{q_0, j_1, j_2, \ldots, j_l{=}j\}$ for state j. Pairing the counterpart states in these two sequences gives us a sequence of state pairs, $S_i = \{(q_0, q_0), (i_1, j_1), (i_2, j_2), \ldots, (i_l$=i, j_l=j$)\}$, traversed by the NFA. Obviously, no state pair can appear twice in this sequence. Because that means the input string can be further reduced into a shorter string and can still cause the NFA to transition from the start state to both i and j. Since there are n^2 distinct state pairs in total, the sequence contains at most n^2 state pairs, meaning the shortest input string contains at most n^2-1 characters. As the breadth-first search algorithm can run as many as n^2 iterations, processing one character during each iteration, the algorithm must be able to find out such a shortest string and hence mark state i and state j as incompatible. It is thus proven by contradiction that state i and state j are truly compatible.

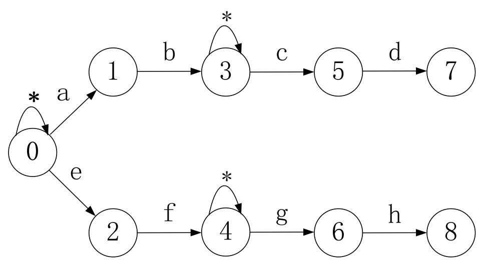

Figure 3. NFA for matching `ab.*cd` and `ef.*gh`.

For example, consider two regular expressions, `ab.*cd` and `ef.*gh`. The first expression defines a pattern where `ab` is followed by `cd`, with zero or more arbitrary characters between them. The second expression defines a similar pattern. The NFA for matching these two regular expressions is shown in Figure 3. Its incompatibility table is given in Table 1.

	0	1	2	3	4	5	6	7	8
0	√	√	√	√	√	√	√	√	√
1	√	√		√	√				
2	√		√	√	√				
3	√	√	√	√	√	√	√	√	√
4	√	√	√	√	√	√	√	√	√
5	√			√	√	√			
6	√			√	√		√		
7	√			√	√			√	
8	√			√	√				√

Table 1. Incompatibility table of the NFA in Figure 3.

3.2 Compatible group

With this compatibility relationship between NFA states available, we use a graph theoretic algorithm to partition NFA states into compatible groups. Every NFA state is represented by a distinct vertex; if two NFA states are incompatible, they are connected by an edge. Therefore, a compatible group of states form an independent set in this graph. Our algorithm proceeds in an iterative manner; in each iteration, one independent set of vertices (i.e., states) are obtained and removed from the graph, leaving a residual graph for subsequent iterations. Once all vertices have been removed from the graph, all NFA states have been partitioned into the obtained independent sets (i.e., compatible groups).

Next, we describe the algorithm for obtaining an independent set during each iteration. This algorithm is also an iterative algorithm. During each iteration of this algorithm, we pick an edge connecting two vertices u and v such that the sum of u's degree and v's degree (in the residual graph) is the largest (among all edges in the residual graph); u and v are temporarily removed from the residual graph. If the remaining graph is an independent set, it is taken as the new independent set. One possible case in this algorithm is, after temporarily removing an edge (u, v), all vertices in the residual graph have been removed and hence the obtained independent set is actually an empty set. In this case, we shall take $\{u\}$ and $\{v\}$ as two new independent sets. Finally, we check if some previously picked vertices can also be added into the new independent set(s), and do so if possible. The remaining vertices of those picked edges will form the new residual graph, from which the next independent set will be obtained.

For example, consider the NFA in Figure 3. In the constructed graph for partitioning into compatible groups, states 0, 3 and 4 are connected with each other and hence form a 3-clique; they are also connected with the other six states, while the other six states are not connected with each other. The NFA states are partitioned into compatible groups (i.e., independent sets) as follows.

- *Independent Set 1:* In this initial graph, any one of the following three edges can be picked first: (0, 3), (0, 4) and (3, 4). Without loss of generality, suppose we pick edge (0, 3) and temporarily remove it from the graph. In the remaining graph, every edge is incident to state 4. Again, we assume without loss of generality that edge (4, 1) is picked and temporarily removed. The remaining vertices — $\{2, 5, 6, 7, 8\}$ — now form an independent set. Finally, we find that among the four vertices temporarily removed, state 1 can actually be added into this obtained independent set, which we do. Thus, the residual graph will be composed of states 0, 3 and 4, forming a 3-clique.
- *Independent Set 2:* In the residual 3-clique, assume again without loss of generality that edge (0, 3) is picked; state 4 as the only remaining vertex forms an independent set. The residual graph is now edge (0, 3) alone.
- *Independent Set 3 & 4:* After we pick the only edge (0, 3), there is no other vertex left. Hence, we shall take $\{0\}$ and $\{3\}$ as two new independent sets.

The entire NFA is thus partitioned into four compatible groups: $\{1, 2, 5, 6, 7, 8\}$, $\{0\}$, $\{3\}$ and $\{4\}$.

3.3 Matching operations

Compared with iNFAnt, the entire process of matching an input character is quite simple for each thread in our design, and can be summarized as follows. (Detailed issues including data structures, memory layout, conflict and divergence will be discussed in Section 4.)

Step 1. Each thread reads in the next input character from the packet it is processing.

Step 2. Each thread obtains the current active state ID stored in its assigned active state array element, and clear that element for properly recording the new active state.

Step 3. Each thread combines the obtained active state ID with the input character to form a two-dimensional index into the NFA transition table (stored in GPU's texture memory), in order to obtain the destination state(s) it should write into the active state array as new active state(s).

Step 4. For each destination state obtained, the thread writes the destination state into the active state array element corresponding to the destination state's compatible group ID.

Step 5. Finally, each thread checks if the obtained destination state is an accepting state. If it is, a local flag is set to indicate that the packet being processed has matched the regular expression pattern represented by that accepting state. For example in Figure 3, accepting states 7 and 8 represent matching of patterns `ab.*cd` and `ef.*gh`, respectively.

If the number of compatible groups and hence the number of active state array elements exceed the number of threads in a warp dedicated to processing each packet, which is 32, the 32 threads can simply repeat step 2-5 for more rounds. The only adjustment needed is that, during the kth round, the index of the active state array element read/written by the ith thread is $(k - 1) \times 32 + i$. For ease of discussion, we shall refer to the operations in step 2-5 as state transition operations.

This design is not only simpler than iNFAnt, in terms of both data structures and matching operations, but also much more efficient. On *Snort36*, the matching speed obtained by our design is

1.38 Gbps, about five times the matching speed of iNFAnt. In our experiments, we observed that for an input character, the number of NFA transitions to be processed in iNFAnt can be five times the number of compatible groups in our design.[2] This well explains the $5\times$ speedup achieved by our design. It also verifies the validity of our analysis of iNFAnt's design drawbacks, as well as the motivating ideas underlying our design.

3.4 Discussion

In fact, the idea of splitting NFA states into compatible groups can be generalized to help boost the performance of other GPU-based parallel applications as well. In particular, every parallel application is composed of a set of concurrent tasks. In NFA-based regular expression matching, each task consists of the state transition operations to be performed for individual NFA states. At a certain moment, each task may and may not need to be performed, depending on some condition. In NFA-based regular expression matching, this task-specific condition is whether each individual NFA state is active or not. According to this condition, we can group these tasks into compatible groups, each consisting of tasks that need not be performed simultaneously. It suffices to allocate an exclusive thread for each compatible group of tasks (like in our design), instead of allocating an exclusive thread for each task (like in iNFAnt's design). As a result, the parallel application's performance can be effectively boosted.

4. Memory layout and optimizations

In this section, we shall describe the detailed data structures of our basic design, as well as their memory layout and the operations on them. Through in-depth analysis and experiments, we shall present effective optimizations that will significantly boost the performance of our design. Moreover, we shall also analyze potential conflict and divergence among threads, and demonstrate how they are eliminated from our design through optimized implementation.

4.1 NFA transition table

In our basic design, the NFA transition table is simply implemented as a two-dimensional array, stored in GPU's texture memory. Each table entry is defined as an `int4` type, composed of four internal elements named `w`, `x`, `y` and `z`, respectively. Each internal element is 32-bit `int` type, encoding a distinct destination state. In CUDA, such a 128-bit `int4` type NFA transition table entry can be fetched with one single memory access.

Within each 32-bit internal element encoding a destination state, the least significant two bytes are used to record the destination state's state ID, supporting 65,536 NFA states, which have been more than enough for practical pattern sets. Within the most significant two bytes of the internal element, the least significant five bits are used to encode the destination state's compatible group ID, giving us 32 compatible groups (corresponding to a warp of 32 threads); the most significant bit is used to indicate whether the destination state is an accepting state; the middle 10 bits are unused for now.

With this simple data structure design, we can allow every NFA state to have up to four destination states on a given input character. In our experiments, we have never found any NFA state that has more than four destination states on any given input character. Nevertheless, in our final design presented in Section 5, we shall address the theoretically possible case where an NFA state can have more than four destination states on a given input character.

To handle the realistic case where there are less than four destination states in an NFA transition table entry, we introduce a *dumb state* into the NFA. If an NFA transition table entry contains less than four destination states, we shall add some dumb states to fill up the `int4` type table entry. For example, for a table entry with two real destination states, we can record them as `w` and `x`, respectively; then, we record a dumb state into `y` and `z`, respectively. The dumb state has transitions on all possible input characters, leading back to the dumb state itself. We implement the dumb state like a real NFA state and assign it to a compatible group according to the algorithm described in Section 3.

4.2 Active state array

Each warp (consisting of 32 threads) maintains its own active state array (consisting of 32 elements), stored in GPU's shared memory. Each active state array element is also 32-bit `int` type, the same as the four destination states stored in each NFA transition table entry, since destination states are stored in active state array elements.

4.3 Conflict and divergence

Next, we analyze the operations involved in our design, demonstrating how potential conflict and divergence are solved through optimized implementation. As described in Section 3, the operations involved in processing an input character are as follows.

Step 1. *Obtain the next input character.* Input characters are obtained in two stages: (1) from global memory to shared memory; (2) from shared memory to each thread. During the first stage, all the 32 threads in a warp read the same 32-byte packet slice, each thread reading a different byte in the 32-byte slice; there is no conflict. All the threads execute the same read operation, except the target addresses are different; there is no divergence, either. After the first stage, a slice of 32 characters are transferred from global memory to a buffer in the shared memory, to be read and processed in the second stage. The second stage consists of 32 rounds of state transition operations, processing one character from the buffer in each round. During each round, all the 32 threads in a warp read the same next character from the buffer in shared memory. There is no divergence, and the same byte in shared memory can be read by all the threads simultaneously without conflict.

Step 2. *Obtain the current active state.* Each thread reads the current active state ID stored in its assigned active state array element, and clears that element for properly recording the new active state. There is neither divergence nor conflict.

Step 3. *Obtain the destination states.* Each thread combines the current active state ID with the input character to form a two-dimensional index into the NFA transition table, in order to obtain the destination state(s) it will write into the active state array as new active state(s). Here, notice that not all active state array elements will contain a valid active NFA state; for example, the entire NFA may have only one state active at that moment and many active state array elements simply do not contain any valid active NFA state. To avoid divergence, even if a thread does not obtain a valid NFA state ID from its active state array element, it still must index into the NFA transition table and read out some table entry, just like a thread that has obtained a valid active state ID. For that purpose, as the "clear" operation after reading from the active state array element, we can let each thread put the dumb state into its active state array element. If no new active state is subsequently written into that element, during the next round the thread will read out this dumb state as its active state. Since the dumb state is physically stored in the NFA transition table like a real state, it allows the thread to perform subsequent state transition operations as if the obtained dumb state is a real NFA state. Even if multiple threads all obtain the dumb state as their current active state and hence read the same NFA transition table entry simultaneously, there will not be conflict. Because GPU will coalesce these read requests into one

[2] Note that the number of compatible groups can be larger than the maximum number of simultaneously active states.

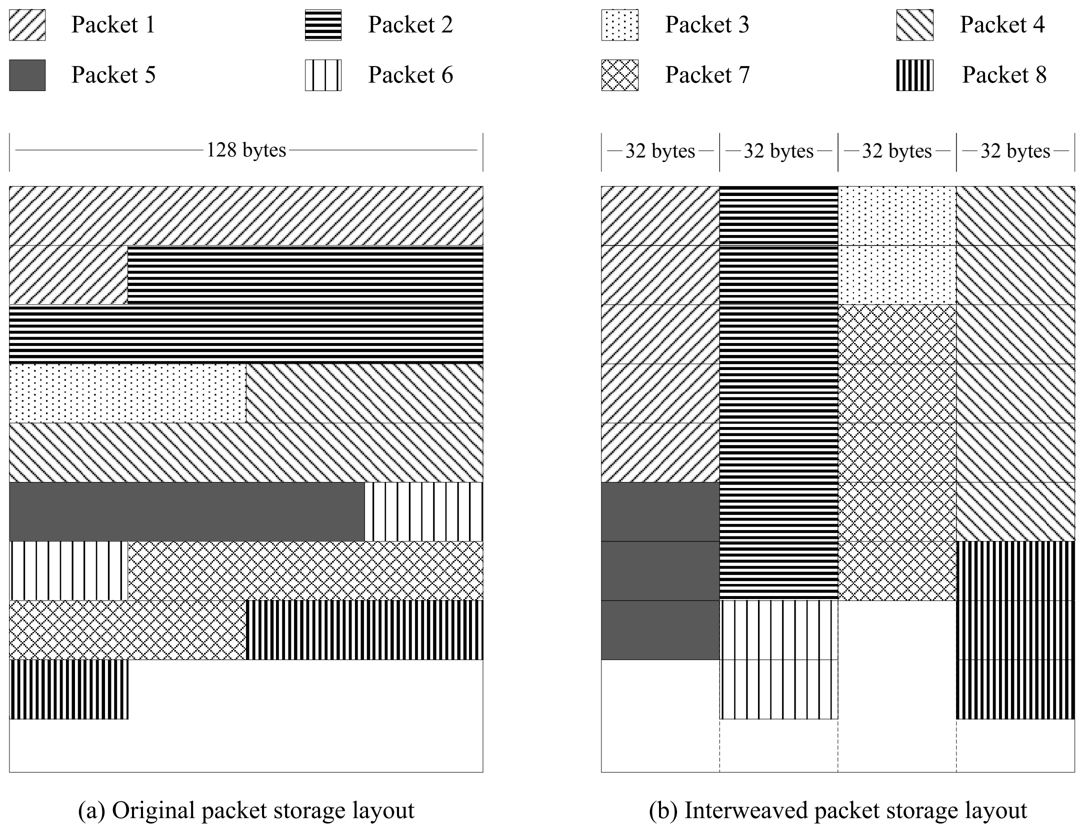

Figure 4. Packet storage layout in global memory.

single read request, before issuing the read operation to the texture memory.

Step 4. *Update the active state array.* Each thread writes every obtained destination state into the active state array element assigned to that destination state's compatible group. There is no divergence, since each thread obtains precisely four destination states, possibly including dumb states, and the dumb state is physically stored in the NFA transition table as well. As to conflict, it is possible that two threads may write their destination states into the same active state array element simultaneously. On one hand, note that the two destination states being written will be active simultaneously; on the other hand, the two destination states being written into the same active state array element means they are in the same compatible group. Therefore, the only possibility is that the two destination states are the same state. The active state array is stored in GPU's shared memory. Such concurrent writes can proceed simultaneously, while only one of the concurrent writes will go through. Since the concurrent writes are writing the same destination state into the same active state array element, it does not matter which one goes through. Thus, no cost is paid for this collision.

Step 5. *Record pattern matching information.* As the last step, each thread checks if the obtained destination state is an accepting state. If it is, a local flag is set to indicate that the packet being processed has matched the pattern represented by that accepting state. As the 32-bit encoding of the destination state contains a bit indicating whether the destination state is an accepting state, we can simply let each thread write that indicator bit into a local flag. To avoid conflict, each thread is allocated a local flag of its own, stored in shared memory for fast access. Since all the threads execute the same write operation into their own flag, there is no conflict or divergence. Finally, after the entire packet has been processed, the local flags will be transferred to global memory for subsequent processing. For each packet, if any one of the threads has its flag set, it means the packet has matched some pattern and hence requires subsequent further processing.

4.4 Input packets

In our basic design, incoming packets are simply stored sequentially (in GPU's global memory), one after one in order of arrival, as shown in Figure 4(a). All the 32 threads in a warp read the same next byte simultaneously, since they are supposed to work on this same byte. GPU will coalesce their read requests for this same byte in global memory into one single request for that byte, before issuing to the global memory.

However, GPU is actually able to perform even more powerful coalescence. Specifically, GPU manages its global memory as aligned 128-byte blocks, each block having the same size as a cache line. With every single global memory read operation, GPU can fetch such a 128-byte block out of the global memory and store it into a cache line for subsequent quick access. Therefore, GPU actually combines all read requests targeting within the same 128-byte block into one request.

Given that, we modify packet storage layout in global memory to better exploit this feature of GPU architecture. In particular, we partition every packet into 32-byte slices (with padding if needed). Slices from different packets are interweaved together, so that a 128-byte block in global memory is likely (although not necessarily) composed of four slices from four different packets, as shown in Figure 4(b). Then in our program, the 32 threads within a warp each reads a different byte from the same 32-byte slice, using their thread ID as offset inside the slice. These 32 concurrent read requests are coalesced into one request for the block containing that slice, and the 32 bytes read out of global memory are then stored in the shared memory, to be processed in the next 32 rounds of state transition operations.

GPU achieves high computing performance through inherent massive parallelism, by processing a large number of packets concurrently. A large number of warps will be running concurrently inside GPU. By mixing slices from different packets into 128-byte blocks, after one of the four packet slices is fetched by the warp responsible for processing that packet, the entire 128-byte block is stored into a cache line; it increases the chance that the warps fetching the other three packet slices in the same block will find their slices already in the cache line. Consequently, computing performance can be effectively boosted. For example, the matching speed achieved by our design on *Snort36* is raised from 1.38 Gbps to 4.01 Gbps, a $3\times$ speedup.

Packet slicing and interweaving as described above are done by CPU, which transmits 32-byte slices from different packets to GPU in interweaving order. Before slicing and transmitting packets to GPU, CPU can sort packets (in order of arrival) into four queues. For example, suppose there are eight packets, which are originally stored sequentially as shown in Figure 4(a). After sorting them into four queues, CPU transmits the first 32-byte slice from each queue to GPU, then the second slice from each queue to GPU, and so on, leading to the storage layout as shown in Figure 4(b), where each row is a 128-byte block.

4.5 Discussion

The idea of interweaving slices of different packets into the same 128-byte block can also be generalized to boost performance of other GPU-based parallel applications. On one hand, parallel applications achieve high computing performance through massive parallelism; many warps run at the same time. On the other hand, GPU has a fixed number of cache lines. Thus, if we can preload data of more warps into the fixed number of cache lines, more (concurrently running) warps will benefit from higher cache hit ratio. In the example of our regular expression matching application, the data of each warp is the packet to be processed by that warp; the way to load data of more warps into cache is to make data of different warps share the same 128-byte block, by slicing and interweaving data of different warps. In other GPU-based parallel applications, data of different warps can be similarly sliced and interweaved for improved performance.

5. Virtual NFA

Thus far, we have stuck to the simplistic design principle of dedicating a whole warp of 32 threads to processing every single packet, without worrying about the following realistic issues.

- Firstly, if an NFA can never have 32 compatible groups, some of the 32 threads in a dedicated warp will be running in vain, which means wasted computing power.
- Secondly, if an NFA can have more than 32 compatible groups, the 32 threads in a dedicated warp will need two or more rounds of state transition operations to finish processing an input character. Again, this means degraded matching performance.
- Thirdly, each NFA transition table entry is assumed to contain at most four destination states, nicely fitting into the `int4` data type. However, this assumption may and may not be the case in practice. How can we handle more than four destination states residing in one table entry, or prevent such cases from happening?

In this section, we present a more elaborated design that is able to address all these issues, by transforming an original NFA into a *virtual NFA* whose states can be partitioned into a small fixed number of, say K, compatible groups ($K = 4$ in our design); each compatible group is still handled by one separate thread. After transformation into such a virtual NFA, no matter how many compatible groups are needed for the original NFA, we can always process a packet using K threads and one round of state transition operations for each input character, leading to fast and stable matching speed. This will address the first two issues discussed above. Furthermore, by using K threads for each packet, instead of using all the 32 threads in a warp for one single packet in our basic design, $\lfloor \frac{32}{K} \rfloor$ packets instead of one packet can be processed by a warp of 32 threads simultaneously. In our design where $K = 4$, eight packets can be processed by a warp of 32 threads simultaneously. As a result, further multiplied matching speed can be achieved.

Finally, if we transform into such a virtual NFA that only $K = 4$ compatible groups are needed, we will be able to ensure that at most $K = 4$ states can be active simultaneously in the virtual NFA. Since destination states residing in the same NFA transition table entry can obviously become active simultaneously, having at most four simultaneously active states directly means none of the virtual NFA's transition table entries can contain more than four destination states. Thus, the third issue discussed above will be solved as well.

The entire process of transforming an original NFA into a virtual NFA is composed of two stages: (1) grouping original NFA states into compatible groups; (2) combining compatible groups into K compatible super groups. The first stage has been described in Section 3. In the rest of this section, we shall present our detailed solution for the second stage. First of all, combining multiple compatible groups into one super group immediately raises a question: now multiple states in a super group can be active simultaneously, how can we transform the super group into a compatible one where at most one of its states can be active at any time, so that it can still be properly handled using one active state array element and one single thread? We answer this question in Section 5.1, with the notion of *virtual state*. Based on that, we shall then present in Section 5.2 an algorithm for combining compatible groups into such compatible super groups. Finally in Section 5.3, we describe relevant adjustment to the basic design for efficient implementation of the virtual NFA design.

5.1 Virtual state

Suppose multiple compatible groups are combined into one super group. Now, two different states (say X and Y) within this super group, originally from two different compatible groups, can be active simultaneously. In this case, our basic design may not be able to preserve proper operation of the NFA. Because on an input character, there can be two active states A and B that lead to destination states X and Y, respectively, while X and Y are combined into the same super group. The threads responsible for processing A and B will then try to write X and Y into the same active state array element assigned to their super group. Consequently, either X or Y will be lost due to this collision.

Therefore, we need a solution for uniquely representing the status of such a super group, in terms of which states in this super group are currently active. For that, we think of each distinct status of the super group as a distinct *virtual state* of the super group. By replacing the original NFA states in this super group with such virtual states representing all possible combinations of active states in the super group, the super group will be transformed into a compatible group where at most one of its virtual states can be active at any time.

Just for illustration purpose, let us suppose all the nine NFA states in Figure 3 are somehow allocated to the same super group. Each virtual state of this super group represents a set of active states in this super group. For instance, if states 0 and 2 are active in the NFA in Figure 3, the virtual state at that point is the active state set $\{0, 2\}$. We encode every virtual state of the super group as a bit vector that is unique within the range of this super group. The super group's bit vector is composed of smaller bit vectors, each representing the sole active state within a distinct compatible group that has been combined into that super group.

To encode a compatible group of m states, we only need a bit vector of $\lceil \log(m+1) \rceil$ bits; because at most one of the states in a compatible group can be active at any time. We assign to each state a unique internal ID within the compatible group (starting from 1). The bit vector being all zero means none of the states in this compatible group is currently active; otherwise, the bit vector records the internal ID of the sole active state in that compatible group.

Suppose the nine NFA states are grouped into four compatible groups — $\{0\}$, $\{3\}$, $\{4\}$ and $\{1,2,5,6,7,8\}$ — that are combined into one single super group. Within the compatible group $\{1,2,5,6,7,8\}$, states 1, 2, 5, 6, 7 and 8 are assigned 1, 2, 3, 4, 5 and 6 as their internal ID, respectively. Internal ID 0 is reserved for the situation that none of the states is active. (This is why the number of bits in the bit vector is $\lceil \log(6+1) \rceil$ instead of $\lceil \log 6 \rceil$, although they do not make any difference in this particular example.) If states 0 and 2 are currently active in this super group, the bit vector encoding the virtual state representing active state set $\{0, 2\}$ will be $\underline{\texttt{1}}\ \underline{\texttt{0}}\ \underline{\texttt{0}}\ \underline{\texttt{010}}$, composed of the four smaller bit vectors encoding its four composing compatible groups. The first single-bit vector indicates state 0, the only state in compatible group $\{0\}$, is currently active; the next two single-bit vectors similarly indicate that states 3 and 4 are not active; the last 3-bit vector `010` indicates that state 2 in compatible group $\{1,2,5,6,7,8\}$ is currently active. It is clear that any virtual state of this nine-state super group can be uniquely represented by such a 6-bit vector.

In general cases, compatible groups of an NFA can be combined into up to four such super groups. To assign to each virtual state a state ID that is unique within the entire NFA, we can sort the super groups in non-increasing order of their size; within each super group, virtual states can be sorted in lexicographic order of the bit vectors encoding the virtual state. Then in this sorted list of all virtual states, the kth virtual state can be assigned state ID k-1, which is clearly unique within the entire NFA. At this point, the NFA is composed of these virtual states instead of the original NFA states, and hence is referred to as a *virtual NFA*. Each virtual state's state ID can be used as the row number of this virtual state in the virtual NFA's transition table, just like the original NFA.

In this virtual NFA, every super group is now a compatible group — at most one virtual state of each super group can be active at any time. Consequently, we can safely assign to each compatible super group one active state array element and one thread, just like we allocate to each compatible group one active state array element and one thread in our basic design. Proper operation of the virtual NFA is thus achieved with the same mechanism as our basic design.

5.2 Combining into super groups

Suppose the states of an original NFA are partitioned into n compatible groups, following the method described in Section 3. We now proceed to combine these n compatible groups into K super groups, which will then be transformed into compatible super groups of virtual states as described in Section 5.1.

In principle, the way we group these n groups into K super groups will not directly affect matching speed. However, it can have considerable impact on the total number of virtual states, and hence storage space of the virtual NFA. To minimize the virtual NFA's state space, we want to minimize the length of the bit vector encoding individual super groups. To achieve this objective, we sort the n compatible groups in decreasing order of their size. We allocate the largest compatible group to the first super group. Then, note that every time we add a compatible group into a super group, the bit vector encoding that super group is appended with the bit vector encoding the joining compatible group. Therefore, we add each remaining compatible group to the super group whose encoding bit vector is currently the shortest.

Just for illustration, suppose we are to use $K = 2$ super groups for the example NFA in Figure 3. The NFA, as described in Section 3.2, is partitioned into four compatible groups. We first allocate the largest compatible group, $\{1, 2, 5, 6, 7, 8\}$, to the first super group. Then, we keep allocating the remaining three compatible groups to the second super group, since its bit vector is always shorter than the bit vector of the first super group. Consequently, the bit vectors of both super groups are 3-bit long. In total, the virtual NFA is thus composed of $2 \times 2^3 = 16$ virtual states.

5.3 Memory layout and state transition

Unlike in the basic design, where every warp of 32 threads are dedicated to a single packet, we hereby allocate 4 threads for each packet only, having 8 packets share the 32 threads in each warp. Accordingly, we partition every packet into 4-byte slices (with padding if needed). To interweave slices from different packets together, CPU sorts packets (in order of arrival) into 32 queues; then, CPU keeps transmitting the next 4-byte slice from each queue to GPU, producing an interweaved storage layout where each 128-block of GPU's global memory is composed of 4-byte slices from the 32 queues.

Then in our program, the 32 threads in a warp each reads a different byte from the 4-byte slice of its packet; the kth thread reads the $(((k-1) \bmod 4)+1)$th byte in the 4-byte slice of the $\lceil k/4 \rceil$th packet. The 32 read requests are coalesced into one read request for the contiguous 32-byte region within a 128-byte block of GPU's global memory. The eight slices are read out with one single memory access and then stored in the shared memory. During each of the following four rounds of state transition operations, one byte from each of the eight slices will be processed by the four threads responsible for the packet where that slice is from.

To perform state transition for an input byte, the kth thread reads the virtual state stored in the kth active state array element, and clear that element for properly recording the new active virtual state. Then, each thread uses the obtained active virtual state ID and the input character as a two-dimensional index to obtain the destination virtual states from the virtual NFA's transition table. For each destination virtual state obtained by the kth thread, suppose it belongs to the lth compatible super group where $1 \leq l \leq 4$; corresponding to that super group is the $(4 \times \lfloor (k-1)/4 \rfloor + l)$th active state array element, into which the destination virtual state is written.

With this virtual NFA design using four compatible super groups and hence four threads for each packet, we achieved 12.50 Gbps matching speed on *Snort36*, representing a 3× speedup compared with the design in Section 4.

6. Experiments

We evaluated the performance of our GPU-based NFA implementation method using real life pattern sets collected from the Snort intrusion detection system [2], packet traces generated using the workload generator introduced in [8] and NVIDIA GTX-460 GPU. We evaluated our proposed method through experiments based on six Snort pattern sets and compared with iNFAnt [9]. Characteristics of the pattern sets and packet traces are reported in Section 6.1.

	Snort16				Snort23			
	p_M = 0.05	p_M = 0.35	p_M = 0.65	p_M = 0.95	p_M = 0.05	p_M = 0.35	p_M = 0.65	p_M = 0.95
Virtual NFA (Gbps)	13.08	13.93	13.17	12.41	13.36	13.23	13.42	13.01
iNFAnt (Gbps)	0.44	0.38	0.36	0.33	0.43	0.37	0.34	0.31
Virtual NFA/iNFAnt Speedup	29.39	35.98	36.01	36.52	29.95	34.01	39.18	40.91

	Snort24				Snort27			
	p_M = 0.05	p_M = 0.35	p_M = 0.65	p_M = 0.95	p_M = 0.05	p_M = 0.35	p_M = 0.65	p_M = 0.95
Virtual NFA (Gbps)	13.20	12.52	12.43	12.32	13.15	12.34	10.28	10.08
iNFAnt (Gbps)	0.43	0.39	0.34	0.31	0.43	0.39	0.31	0.28
Virtual NFA/iNFAnt Speedup	30.37	31.76	36.15	38.59	30.30	31.62	32.55	35.33

	Snort34				Snort36			
	p_M = 0.05	p_M = 0.35	p_M = 0.65	p_M = 0.95	p_M = 0.05	p_M = 0.35	p_M = 0.65	p_M = 0.95
Virtual NFA (Gbps)	13.12	13.79	12.04	12.52	13.18	13.99	12.99	12.50
iNFAnt (Gbps)	0.42	0.36	0.32	0.29	0.41	0.33	0.30	0.26
Virtual NFA/iNFAnt Speedup	31.08	37.43	37.24	42.38	31.73	42.22	42.01	46.31

Table 2. Matching speed.

Matching speed results are reported in Section 6.2. Storage space results are reported in Section 6.3.

6.1 Experiment setup

As shown in Table 3, the six Snort pattern sets used in our experiments are diverse in nature. Some pattern sets are relatively simple, while some others are much more complex. Consequently, in terms of inflation ratio (i.e., DFA size divided by NFA size), their DFAs are larger than their NFAs by 15.51 times to 281.22 times; their DFAs consist of 13,825 states to 190,951 states.

	DFA size	NFA size	DFA/NFA inflation
Snort-16	67,682	447	151.41
Snort-23	32,518	518	62.77
Snort-24	13,886	575	24.14
Snort-27	106,452	499	213.33
Snort-34	13,825	891	15.51
Snort-36	190,951	679	281.22

Table 3. Characteristics of pattern sets.

Workloads to be used as input character stream are generated for each individual pattern set, respectively, using the workload generator proposed in [8]. Every workload is generated as a byte stream, according to a specified parameter p_M. When generating the next byte, it is chosen with probability p_M such that the byte will lead away from the start state (of the pattern set's finite automaton); with probability $1-p_M$, the next byte is chosen randomly. After a workload is generated by the workload generator, we partition the workload into 1KB segments, each segment representing the payload of a packet.

For each pattern set, we generated four types of workloads using $p_M = 0.05, p_M = 0.35, p_M = 0.65$ and $p_M = 0.95$, respectively. Each generated workload is 280KB in length. For each p_M value, we generated a number of such workloads and combine them into a single large workload, which is 256MB in size and divided into 256K packets. In total, 24 such large workloads are generated.

6.2 Matching speed

We run our virtual NFA design and iNFAnt[3] on each of the 24 workloads, and report the matching speed results in Table 2. 4,096 blocks, each consisting of 256 threads, are employed for each workload. Here, matching speed is calculated by dividing workload size with the time taken to finish matching the workload.[4] As we can see, on all pattern sets and parameter settings, our virtual NFA design consistently achieves matching speed above 10Gbps (OC-192 link rate). Compared with iNFAnt, our virtual NFA design can boost matching speed by 29∼46 times.

6.3 Memory space

To compare the scalability our virtual NFA design with DFA-based methods, we conducted a series of experiments to reveal the growth trend of virtual NFA size. For a pattern set (which in this case is *Snort-36* whose DFA is the largest) consisting of n patterns, we generated a series of $\lceil n/4 \rceil$ subsets. The kth subset consists of the first MIN$(4k, n)$ patterns of the original pattern set. For each subset, we measured the number of states in the virtual NFA and the DFA, respectively. Then, we plot the results in Figure 5, where the X-axis represents the number of virtual NFA states and the Y-axis represents the number of DFA states. As we can see, the virtual NFA size tends to grow exponentially more slowly than the DFA size, demonstrating much better scalability.

We also measured the storage space needed for implementing the pattern sets. The virtual NFA transition tables of *Snort16*, *Snort23*, *Snort24*, *Snort27*, *Snort34* and *Snort36* use 3.02MB, 6.5MB, 3.06MB, 12.5MB, 6.13MB and 14MB, respectively.[5] As virtual NFA transition tables are stored in inexpensive texture memory (which has up to 1GB capacity on NVIDIA GTX-460 GPU), these memory space requirements incur very low cost.

[3] The authors of iNFAnt[9] proposed a multi-striding technique for accelerating iNFAnt, by processing multiple bytes per state transition. However, through private communication we have confirmed with the authors that their experiments reported in [9] are actually erroneous and the proposed multi-striding technique is not as practical as demonstrated by the experiment results in [9]. Therefore, we compared our design with their basic iNFAnt design, where one byte is processed per state transition.

[4] The experiment results did not include the time spent on constructing virtual NFAs. Because the goal is to boost matching speed. Virtual NFA construction is a one time cost; after the virtual NFA is constructed, we will not need to pay this cost again during the matching process, until the regular expressions change (which may not happen over days/months).

[5] For these virtual NFAs, we are able to use 64-bit `short4` type instead of 128-bit `int4` type for their transition table entry.

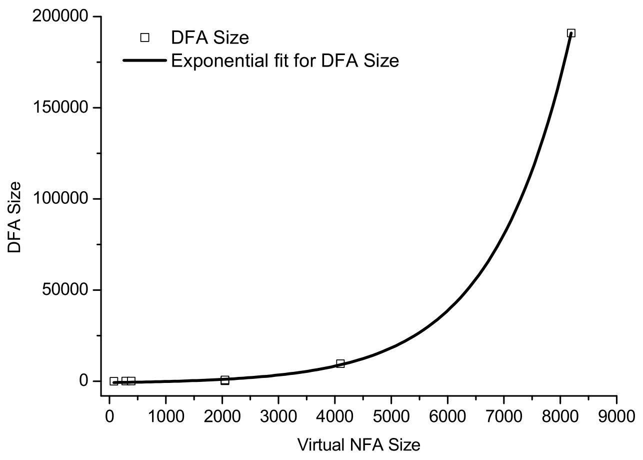

Figure 5. Growth trend of virtual NFA size.

7. Conclusions

In this work, we analyzed and demonstrated some important properties of NFA, using real life pattern sets as examples. Based on this understanding of NFA properties as well as GPU architecture, we conducted in-depth study, both experimental and analytical, of how NFAs can be best fitted into GPU architecture through proper data structure and parallel programming design, so that GPU's parallel processing power can be effectively mobilized to achieve high speed regular expression matching. The three pivot ideas of our design include compatible group, packet interweaving and transforming the original NFA into our proposed virtual NFA.

We evaluated the performance of our virtual NFA design using real life pattern sets collected from the Snort intrusion detection system [2], on NVIDIA GTX-460 GPU. Experiment results demonstrate that, virtual NFA can achieve 29∼46 times speedup, consistently yielding over 10Gbps matching speed. Meanwhile, compared with DFA size, our virtual NFA design only needs a very small amount of memory space, growing exponentially more slowly than DFA size. These results make our virtual NFA design an effective solution for memory efficient high speed (e.g. 10Gbps OC-192 link speed) regular expression matching, and clearly demonstrate the power and potential of GPU as a platform for memory efficient high speed regular expression matching.

References

[1] *PCRE - Perl Compatible Regular Expressions*. http://www.pcre.org/.

[2] *Snort intrusion detection system*. http://www.snort.org/.

[3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. mei W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of ACM PPoPP*, 2010.

[4] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of CoNext*, 2007.

[5] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of ANCS*, 2007.

[6] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of ANCS*, 2008.

[7] M. Becchi and P. Crowley. Extending finite automata to efficient match perl-compatible regular expressions. In *Proceedings of CoNext*, 2008.

[8] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Proceedings of IISWC*, 2008.

[9] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA pattern matching on GPGPU devices. *SIGCOMM CCR*, 40(5):21–26, 2010.

[10] M. Chen. TCAM-based high speed regular expression matching. Bachelor thesis, Institute of Networked Systems (IONS), University of Science and Technology of China, June 2010.

[11] M. Chen, Q. Dong, and K. Peng. TCAM-based DFA implementation: A novel approach to efficient regular expression matching. Technical report, Institute of Networked Systems (IONS), University of Science and Technology of China.

[12] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of ACM PPoPP*, 2010.

[13] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. In *Proceedings of PPOPP*, 2011.

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of ACM PPoPP*, 2011.

[15] J. Kim, H. Kim, J. H. Lee, and J. Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of ACM PPoPP*, 2011.

[16] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of ANCS*, 2007.

[17] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of ACM SIGCOMM*, 2006.

[18] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In *Proceedings of ACM PPoPP*, 2009.

[19] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of USENIX Security*, August 2010.

[20] K. Peng, Q. Dong, and M. Chen. TCAM-based DFA deflation: A novel approach to fast and scalable regular expression matching. In *Proceedings of ACM/IEEE IWQoS*, 2011.

[21] K. Peng, S. Tang, Q. Dong, and M. Chen. Chain-based DFA deflation for fast and scalable regular expression matching using TCAM. In *Proceedings of ANCS*, 2011.

[22] E. F. O. Sandes and A. C. M. de Melo. CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences. In *Proceedings of ACM PPoPP*, 2010.

[23] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proceedings of IEEE Symposium on Security and Privacy*, 2008.

[24] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of ACM SIGCOMM*, 2008.

[25] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *Proceedings of ISPASS*, 2009.

[26] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Proceedings of RAID*, 2008.

[27] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of RAID*, 2009.

[28] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of ANCS*, 2006.

[29] Y. Zhang, J. Cohen, and J. D. Owens. Fast tridiagonal solvers on the GPU. In *Proceedings of ACM PPoPP*, 2010.

[30] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. GRace: a low-overhead mechanism for detecting data races in GPU programs. In *Proceedings of ACM PPoPP*, 2011.