This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

Perfect Hashing Based Parallel Algorithms for Multiple String Matching on Graphic Processing Units

Cheng-Hung Lin
Dept. of Electrical Engineering
National Taiwan Normal University
Taipei, Taiwan
brucelin@ntnu.edu.tw

Jin-Cheng Li
Dept. of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan
lgc800430@yahoo.com.tw

Chen-Hsiung Liu
Dept. of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan
lgen7604@gmail.com

Shih-Chieh Chang
Dept. of Computer Science
National Tsing-Hua University
Hsinchu, Taiwan
scchang@cs.nthu.edu.tw

Abstract- **Multiple string matching has a wide range of applications such as network intrusion detection systems, spam filters, information retrieval systems, and bioinformatics. To accelerate multiple string matching, many hardware approaches are proposed to accelerate string matching. Among the hardware approaches, memory architectures have been widely adopted because of their flexibility and scalability. A conventional memory architecture compiles multiple string patterns into a state machine and performs string matching by traversing the corresponding state transition table. Due to the ever-increasing number of attack patterns, the memory used for storing the state transition table increased tremendously. Therefore, memory reduction has become a crucial issue in optimizing memory architectures. In this paper, we propose two parallel string matching algorithms which adopt perfect hashing to compact a state transition table. Different from most state-of-the-art approaches implemented on specific hardware such as TCAM, FPGA, or ASIC, our proposed approaches are easily implemented on commodity DRAM and extremely suitable to be implemented on GPUs. The proposed algorithms reduce up to 99.5% memory requirements for storing the state transition table compared to the traditional two-dimensional memory architecture. By studying existing approaches, our results obtain significant improvements in memory efficiency.**

Keywords-perfect hashing; string matching; deterministic finite automaton

I. INTRODUCTION

Multiple string matching is widely used in many applications such as network intrusion detection systems (NIDS), spam filters, information retrieval systems, and bioinformatics to find all locations of multiple patterns simultaneously. To accelerate multiple string matching, many hardware architectures [1][8][9][14][19][21] are proposed. Among the proposed architectures, the Aho-Corasick (AC) algorithm is widely adopted in memory architectures because it uses linear time to perform multiple string matching. A traditional memory architecture for multiple string matching works as follows. First, multiple string patterns are compiled into a single deterministic finite automaton (DFA) using the AC algorithm. Then, the corresponding state transition table of the DFA is stored in a memory. Finally, multiple string matching is performed by traversing the DFA.

Traditional memory architecture uses a two-dimensional memory to store the corresponding state transition table of the DFA shown in Figure 1. The figure depicts each row as representing a state which contains 256 columns to store the next state information for each ASCII alphabet and a column to store match vectors. In addition, the *state* and *char* registers storing information of the current state and the input character are used to look up the next state information from the state transition table.

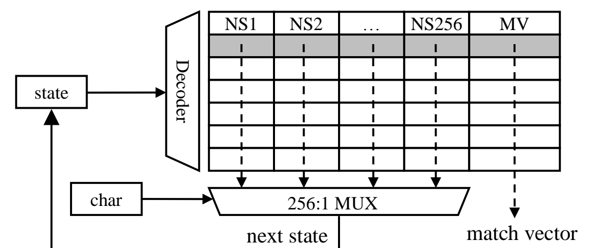


Figure1. Traditional memory architecture

With growing number of attack patterns, the memory required for use of storing the corresponding state transition table increased tremendously. Considering the memory architecture in Figure 1, each row needs 256 x 4 (1K) bytes to store the next state information and one byte to store a match vector that indicates which pattern is matched. In other words, a state machine with one million states needs approximate 1G bytes of memory for storing the state transition table. Therefore, reducing the required memory for storing the state transition table has become a crucial issue.

There are two ways to reduce the memory requirement. One is to reduce the size of an automaton while the other is to compact the storage of an automaton. Delayed DFA (D^2FA) [20] is proposed to remove duplicated transitions by introducing "default" transitions. A compact data structure [10] is proposed to reduce a DFA by merging non-equivalent states. A memory-efficient algorithm [4] is proposed to reduce an AC state machine by merging non-equivalent states. The B-FSM based pattern

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

matching (BFPM) scheme [8] is based on the concept of so-called state transition rules which specify the relationship of each state transition and their corresponding next state information. To compress state transition rules, BFPM introduces wildcard symbol "*" to represent "don't care" condition in their rule sets. BFPM prioritizes the transition rules and adopts the Balanced Routing Table (BART) search algorithm for exact-, prefix- and range-match searches. To improve storage efficiency, the BFPM adopts a hash function and limits the maximum number of hash collision of any state transition by a bound of 4. However, the processing of hash collisions may increases memory access and degrades system performance. Thus, by extending the work of BFPM, a cached deterministic finite automaton (CDFA) [21] is proposed to reduce transitions in an AC_DFA, while a next state addressing (NSA) scheme is proposed to efficiently store transitions using less memory. Nevertheless, the NSA scheme still uses two-dimensional memory to store the multiple transitions from a certain state which has multiple next states. When such states are increased, the memory efficiency of NSA is decreased. A bit-split memory architecture [9] is proposed to split an AC_DFA finite state machine into a set of small AC_DFA and reduces the total memory requirement. Still, the bit-split memory architecture uses two-dimensional memories to store state transition tables. Bitmap compression and path compression [13] are proposed to achieve compact storage of an AC state machine and has better performance on worst cases. Even though the computational cost of the proposed compression scheme remains high. CompactDFA [1] also introduces wildcard symbol "*" to minimize the memory of an AC_DFA by encoding all transitions to a specific state as a single prefix. The scheme resolves the string matching problem by applying the longest prefix matching problem with a little overhead in the number of bits of state encoding. Since CompactDFA addresses the pattern matching problem by applying the longest prefix matching problem such as the IP-lookup problem. In addition, CompactDFA stores the compressed rule sets on Ternary Content Addressable Memory (TCAM) which permits rapid table lookups through longest prefix matches. Furthermore, a HEXA [19] approach is proposed to significantly reduce memory by a compact, and to be viewed as a historical representation of state transition. However, TCAMs are expensive in chip area and power consumption.

Nevertheless, another way to reduce memory is to compact the two-dimensional memory using hashing techniques. However, common hashing algorithms are noted for their hashing collision problem which requires timing and hardware overhead to resolve. A progressive perfect hashing (P^2-hashing) [22] algorithm is proposed to store an AC automaton in a compact hash table without collisions. The main idea of the P^2-hashing algorithm is to divide the AC transitions into multiple sets according to state numbers and incrementally put these sets into a hash table. If a hash collision happens during the placement of a transition in one set, the already-placed transitions in this set are removed and the state number of the set is renamed. Then, a new trial starts until all transition sets are put into a hash table. However, the construction time of the perfect hash table is not deterministic.

In our previous work, the Parallel Failureless Aho-Corasick (PFAC) [3] algorithm is proposed to parallelize string matching processes on GPUs. Compared to state-of-the-art approaches, the PFAC state machine has the minimum number of transitions because it removes all failure transitions as well as the self-loop transitions to the initial state. In other words, the state transition table is sparse and most entries of the two-dimensional table are empty. Continually, we propose to use a static perfect hashing to compress the two-dimensional sparse table and achieve significant memory reduction [5].

The main contributions of this paper are summarized as follows.

1. We propose two parallel string matching algorithms which adopt a hardware-friendly perfect hashing algorithm [17] to compact a state transition table. The perfect hashing algorithm stores the valid transitions in a compact hash table and takes constant time to generate the hash index and access the hash table without collisions.
2. We have implemented the proposed memory architecture on graphic processing units (GPU) and evaluated the performance of the proposed architecture using attack patterns from Snort [21] V2.8 and input packets from DEFCON [6]. Our architecture achieves up to 99.5% of memory reduction compared to the traditional two-dimensional memory architectures. The experimental results show that the proposed algorithm outperforms state-of-the-art memory reduction architectures on memory efficiency.

II. BACKGROUND

A. Aho-Corasick algorithm

Among the proposed pattern matching algorithms, the Aho-Corasick (AC) algorithm has been widely adopted because of its ability of matching multiple patterns simultaneously. The AC algorithm introduces a new transition called failure transition to replace all backtracking transitions of a DFA. For example, Figure 2 shows an AC state machine of the three patterns, "SFTP", "PPS", and "FTPS", where the solid lines represent valid transitions and the dotted lines represent failure transitions. However, the AC algorithm is not suitable for hardware implementation because taking failure transitions may incur cycle penalties for state traversing.

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

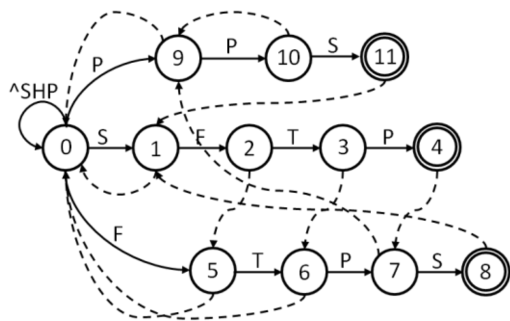


Figure 2. AC state machine of "SFTP", "PPS", and "FTPS"

B. Parallel Failureless Aho-Corasick algorithm

In our previous work, we propose a Parallel Failureless Aho-Corasick (PFAC) [3] algorithm which achieves significant throughput on GPUs. The library and source code of the PFAC algorithm are accessible in Google Code (http://code.google.com/p/pfac/)[16].

Using PFAC has two steps. First, attack patterns are compiled into a PFAC state machine. And then, each byte of an input stream is assigned an individual thread to traverse the PFAC state machine. Figure 3 shows the PFAC architecture where each thread traverses the same state machine. The most important property of the PFAC architecture is that each thread of PFAC is only responsible for identifying the pattern starting at the thread's starting position. Whenever a thread cannot find the beginning of a pattern at its starting position, it terminates immediately without taking failure transitions. In other words, each thread of PFAC runs in the best time $O(1)$ and the worst time $O(m)$ where m is the longest pattern length. Therefore, we can eliminate all failure transitions as well as the self-loop transition to the initial state in the traditional AC state machine. Precisely speaking, a PFAC state machine of n states contains only n-1 valid transitions. In other words, the transition-to-state ratio of the PFAC automaton is less than 1 which indicates that the PFAC automaton is an intrinsic sparse automaton. For example, Figure 4 shows the PFAC state machine of the three patterns, "SFTP", "FTPS", and "PPS", which has 12 states and only 11 valid transitions left. Compared to Figure 2, all backtracking transitions and the self-loop transition to the initial state are all removed. Compared to state-of-the-art architectures, a PFAC state machine has minimum number of valid transitions. Another important property of PFAC is that each final state of the PFAC machine represents a unique pattern without handling multiple outputs. Therefore, we eliminate output table accesses by reordering the number of final states to represent matching vectors. The encoding technique is described as follows. For a PFAC state machine which has n final states (patterns), the number of final states is encoded from 1 to n, and all internal states including the initial state are numbered from n+1. As shown in Figure 4, the final states of the patterns "SFTP", "FTPS", and "PPS" are numbered from 1 to 3 while the other states including the initial state are numbered from 4. Whenever the PFAC machine reaches a state whose number is smaller than the initial state, we know that the machine reaches a final state and directly outputs the number of the final state without looking up the output table.

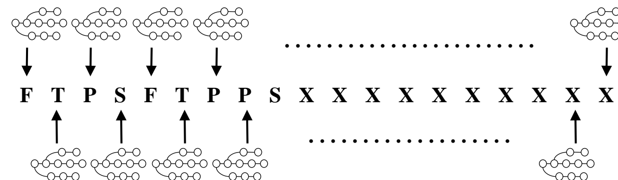


Figure 3. PFAC architecture which allocates each byte of the input stream via an individual thread

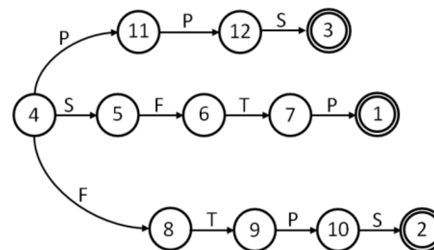


Figure 4. PFAC state machine of "SFTP", "FTPS", and "PPS"

In order to achieve the best result, the PFAC algorithm stores the state transition table in a two-dimensional memory because only one memory access is required to look up the next state information that includes a given state and an input character. However, it's extremely space-inefficient to use a two-dimensional memory to store the sparse PFAC automaton.

To compress the PFAC state transition table, our basic idea is to use a perfect hash algorithm to store only valid transitions in a hash table as shown in Figure 5. The hash table has two fields to store valid keys and the corresponding next state information. To query the next state information of an input key (transition), the hash index is first generated by the perfect hash function and then the key stored in the hash table is retrieved to compare with the input key. If they are equal, the next state is delivered to update the state value. Otherwise, the state value is set as a trap state and then we stop traversing the state machine.

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

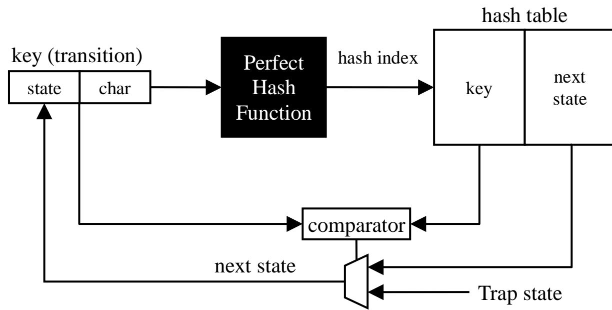


Figure 5: Perfect-hashing based memory architecture

Unlike traditional hash functions which may map two or more keys to the same hash value, a perfect hash function maps each key (valid transition) to a distinct hash value without collisions. However, the cost of computing the hash index and accessing the hash table would degrade the performance. As shown in Figure 5, the time required to query a key is equal to the amount of the time computing the hash index and the time looking up the hash table. Although many perfect hashing algorithms are available in literature, most of them require multiple memory accesses to obtain the accurate hash index from a hash table lookup. Therefore, choosing an efficient perfect hashing algorithm is crucial to optimize the performance of the architecture.

Note that many applications need to build a hash table on the fly, as in our case, the content in the hash table is known in advance so that a static perfect hash function can be applied.

III. HARDWARE-FRIENDLY PERFECT HASHING ALGORITHMS

Among the proposed perfect hashing algorithms, a hardware-friendly perfect hashing algorithm [17] is proposed to store a static sparse table without collisions. The main idea of the perfect hashing algorithm is to shift the rows (or columns) of a sparse two-dimensional table until no two keys appear in the same position. And then, the sparse two-dimensional table is compacted to a dense one-dimensional table. The main advantage of the perfect hashing algorithm is its linear construction time of hash table and constant time to acquire the hash index. Although the perfect hash function does not guarantee to achieve minimal space theoretically, it's well-suited to be implemented on hardware with a little modification.

In this section, we first discuss two modifications of the perfect hash algorithm in [17], Row-Shifting Perfect Hashing (RSPH) and Column-Shifting Perfect Hashing (CSPH). Then follow up with discussions on how to integrate RSPH and CSPH into string matching algorithms.

A. *Row-Shifting Perfect Hashing (RSPH)*

The procedure of constructing the hash table using RSPH is as follows.

i. Start with width w of a two-dimensional key table and place each valid key k at location (row, col), where row and col are equal to quotient and remainder of the key divided by the width w, respectively. We use the following expressions to denote the two operations.

$$row = \lfloor k / w \rfloor$$
$$col = k \bmod w$$

ii. Rows are prioritized by the number of keys in it and slide rows by order of priority.
   1. First, slide the row to left first in order for the first key in the row be aligned at the first column.
   2. Then, slide the row to right until each column has only one key and record the offset in an array which is defined as *RO* (abbreviated from Row Offset) table.

iii. Compact the two-dimensional array into a row.

We illustrate the procedure of RSPH using the example in Figure 6. In Figure 6, we have the key set, S = {2, 4, 10, 11, 13, 14, 20, 21, 22, 25, 27} which represent valid keys in the corresponding positions. First, we set a two-dimensional key table of width 8 (w=8) and put each key k in S at the location (row, col), where row = $\lfloor k / 8 \rfloor$, col = k mod 8. For example in Figure 6(a), the key 20 is put into the location (row, col) = ($\lfloor 20 / 8 \rfloor$, 20 mod 8) = (2, 4). In the second step, we first shift each row to left and then shift right until each column has only one key and record the offset from the first column in the RO array. The order of a row to be moved is prioritized by the number of keys in it. For example, row 1 has the highest priority because row 1 has 4 keys in it. As shown in Figure 6(b), row 1 is first shifted left so that the key 10 is aligned at column 0 and record the offset, -2 in RO[1]. Then, rows 2, 0 and 3 are shifted to proper positions so that no collisions occur in the same column. The offset of each row is recorded in the RO array. Finally, the two-dimensional table is collapsed to a one-dimensional hash key table. In Figure 6(b), the memory space for storing the 11 keys is reduced from 32 to 16 elements, where 12 elements are for storing keys and 4 elements for recording the offsets of rows.

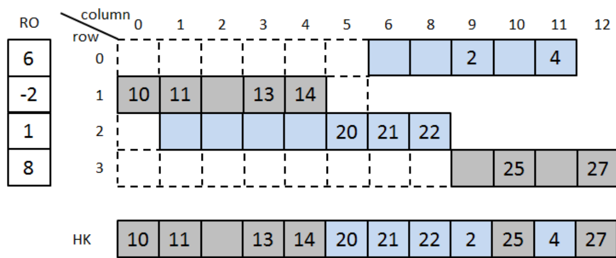
Different than the original algorithm in [17], we propose to slide each row to left first and then slide right to the proper position. The reason is due to the fact that the first 32 symbols of ASCII are non-printable characters and most valid transitions do not contain non-printable characters. Therefore, using our RSPH method can achieve better load factor than the original algorithm in [17].

row \ column	0	1	2	3	4	5	6	7
0			2		4			
1			10	11		13	14	
2					20	21	22	
3		25		27				

(a) Put keys into a two-dimensional key table of width 8

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems



(b) Results of RSPH

Figure 6. Example of creating a perfect hash table by RSPH

On the other hand, given an input key, k, the procedure of validating the input key is as follows.

i. $row = \lfloor k / w \rfloor$;
ii. $col = k \bmod w$;
iii. $index = \mathrm{RO}[row] + col$;
iv. If HK[$index$] == k
   k is a valid key;
  else
   k is an invalid key;

Given an input key, k, the first and second steps calculate the position (row, col) of k in the original table. Because RSPH uses the RO array to store the offset of each row, the third step calculates the index of the query key in the hash table by summing RO[row] and col. The last step validates the query key by comparing the query key with the value stored in HK[$index$]. For example, the index of the key 14 is equal to RO[$\lfloor 14/8 \rfloor$] + 14 mod 8 = RO[1] + 6 = -2 + 6 = 4. Because HK[4] is equal to 14, we know that the key 14 is a valid key and belongs to the key set S. Consider the other key 19. The index of the key 19 is equal to RO[$\lfloor 19/8 \rfloor$] + 19 mod 8 = RO[2] + 3 = 1 + 3 = 4. Because HK[4] is not 19, we know that 19 is not a valid key.

B. Column-Shifting Perfect Hashing (CSPH)

The procedure of the CSPH is as follows.

i. Start with a width w *of* two-dimensional key table and place each valid key k at location (row, col), where row and col are equal to quotient and remainder of the key divided by the width w, respectively. We use the following expressions to denote the two operations.

$$row = \lfloor k / w \rfloor$$
$$col = k \bmod w$$

ii. Columns are prioritized by the number of keys in it and slide columns by order of priority.
 1. First, slide the column up first to let the first key in the column be aligned at the first row.
 2. Then, slide the column down until each row has only one key and record the offset in an array which is defined as CO (abbreviated from Column Offset) table.

iii. Compact the two-dimensional array into a column.

Figure 7 demonstrates the results of CSPH. Each column slides up and down to find a proper position that no collision occurs in the same row. The offsets of columns are recorded in the CO array. Finally, the two-dimensional table collapses into a one-dimensional hash key table. The memory space for storing the 11 keys is reduced from 32 to 19 elements, where 11 elements are for storing keys and 8 elements for recording the offsets of columns.

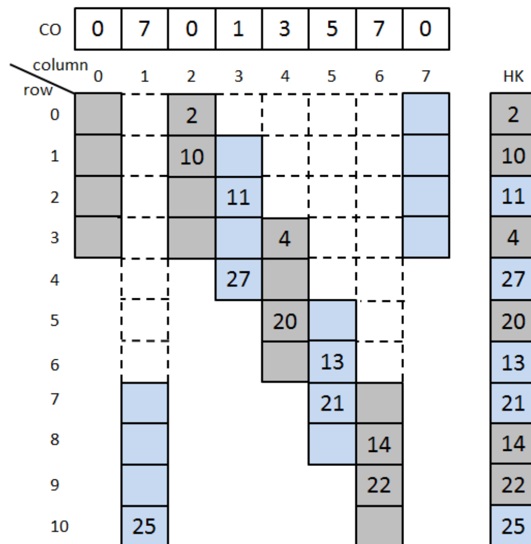


Figure 7. Results of CSPH

On the other hand, given an input key, k, the procedure of validating the input key is as follows.

i. $row = \lfloor k / w \rfloor$;
ii. $col = k \bmod w$;
iii. $index = \mathrm{CO}[col] + row$;
iv. If HK[$index$] == k
   k is a valid key;
  else
   k is an invalid key;

Because CSPH uses the CO array to store the offset of each column, the index is obtained by summing the CO[col] and row. For example, the index of key 14 is equal to CO[14 mod 8] + $\lfloor 14/8 \rfloor$ = CO[6]+1 = 7+1 = 8. Then, we find that HK[8] is equal to 14. Therefore, 14 is a valid key.

IV. Perfect-Hashing Based Memory Architecture

In this section, we propose two basic perfect-hashing based memory architectures and their modifications for space and time optimization.

A. Row-Shifting Perfect-Hashing Memory Architecture

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

Figure 8 shows the Row-Shifting Perfect-Hashing Memory Architecture (RSPHMA). In Figure 8, the *NS* table (abbreviated from Next State) is used to store the next state information which corresponds to each key (valid transition) stored in the HK table. The input key (transition) is composed of a current state (stored in *state*) and an input character (stored in *char*). The comparator is used to compare the input key with the key stored in the HK table. If the input key matches the key stored in the HK table, it means that the input key (transition) is a valid transition and the current state is updated by the next state information stored in NS. Otherwise, the input key is not a valid transition and the current state is updated as a trap state which indicates there is no valid next state for the current state and input character. The PHF block generates the hash index.

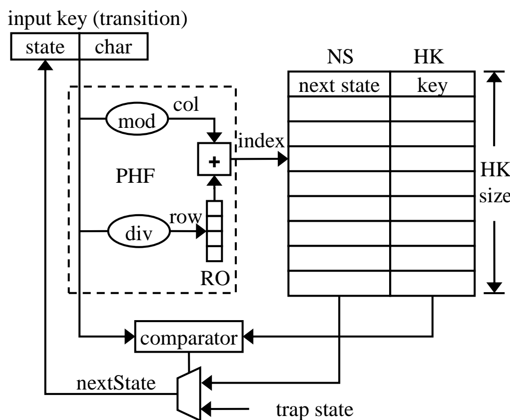


Figure 8. Row-Shifting Perfect-Hashing Memory Architecture (RSPHMA)

The procedure to query the next state in RSPHMA is as follows.

i. *row* = $\lfloor k / w \rfloor$;
ii. *col* = *k* mod *w*;
iii. *index* = RO[*row*] + *col*;
iv. If HK[*index*] == *k*
    k is a valid key;
  else
    k is an invalid key;
v. If *k* is a valid key
    nextState = NS[*index*];
  else
    nextState = trap state;

In Figure 8, we find that the HK table used for storing keys consumes a lot of memory. In addition, the cost of modulation and division computation in PHF is expensive. Therefore, we propose two modifications to optimize the RSPHMA: one is to reduce the memory of HK to improve memory efficiency while the other is to reduce the complexity of PHF to generate hash index. The space-efficient and time-efficient RSPHMA are described as follows.

B. *Space-efficient RSPHMA*

In Figure 8, the HK table is used for storing keys which is composed of current states and input characters. The keys stored in HK are used to verify whether an input key is a valid key (transition) or not. In Figure 6(b), if there are two distinct keys having the same index, the two keys must locate in different rows in the key table. The proof of the proposition is as follows.

PROPOSITION 1. In Row-Shifting Perfect-Hashing algorithm, if there are two distinct keys having the same index, the two keys must locate in different rows in the key table.

Proof. For the sake of contradiction, suppose that if there are two distinct keys having the same index, the two keys must locate in the same row in the key table. In Row-Shifting Perfect-Hashing algorithm, the two keys are as follows.

$$key1 = row * width + column1$$

$$key2 = row * width + column2$$

Since key1 is distinct from key2, column1 is distinct from column2.

Furthermore, in Row-Shifting Perfect-Hashing algorithm, the indices of the two keys are calculated as follows where RO is the table storing offsets of each row.

$$index1 = RO[row] + column1$$

$$index2 = RO[row] + column2$$

Since column1 is distinct from column2, index1 is distinct from index 2. The result contradicts our assumption that if there are two distinct keys having the same index, the two keys must locate in the same row. ■

For example, the two keys 14 and 19 mapping to the same position (the fifth element) of the hash table locate in the second and third rows, respectively. In other words, the correctness of a key can be verified by checking the row of a key instead of checking the whole key. Therefore, the hash table can be further reduced by storing the row of keys instead of the whole keys. Furthermore, in order to reduce the complexity of PHF, the width of key table is set to power of two, 512, 1024, or 2048 typically. Therefore, the operations of modulation and division can be replaced by mask and shifter, respectively. Figure 9 shows the space-efficient RSPHMA where the HK table only stores the row number of keys. In Figure 9, the right-shifter is used to shift the input key to get the row number of the input key. For example, if the width of the key table is 1,024, the right-shifter shifts the input key 10 bits. In other words, the hash key is reduced by 10 bits. It's noted that by increasing the width of the key table, it reduces the width of hash key but increases the HK size and decreases the load factor, the ratio of the number of valid keys to the HK size.

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

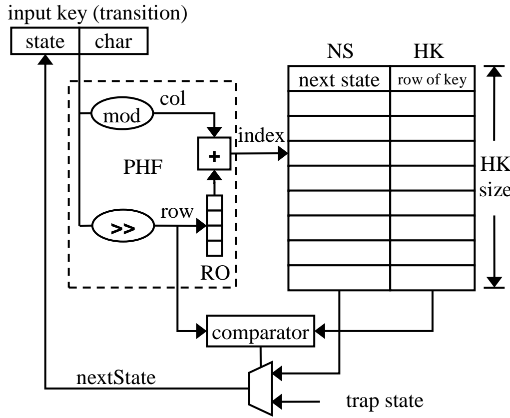


Figure 9. Space-efficient RSPHMA

The procedure to query the next state in the space-efficient RSPHMA is as follows.

i. $row = k \gg log_2 w$;
ii. $col = k \bmod w$;
iii. $index$ = RO[row] + col;
iv. If HK[$index$] == ***row***
  k is a valid key;
 else
  k is an invalid key;
v. If k is a valid key
  $nextState$ = NS[$index$];
 else
  $nextState$ = trap state;

C. Time-efficient RSPHMA

From the above discussion, a less complicated perfect hash function can improve the performance of the proposed memory architecture. If the width of the key table is set to 256, the row of a key is equal to the state number. Therefore, the HK table stores state numbers as hash keys. Because the HK table stores state numbers as hash keys, the calculations of *row* and *col* in PHF can be eliminated and replaced by *state* and *char*. However, setting the width of the key table to 256 increases the width of the HK table as well as the size of the RO table.

Figure 10 shows the time-efficient RSPHMA where the HK table stores state numbers as hash keys. The *state* and *char* are directly used to generate the index without modulation and division operations.

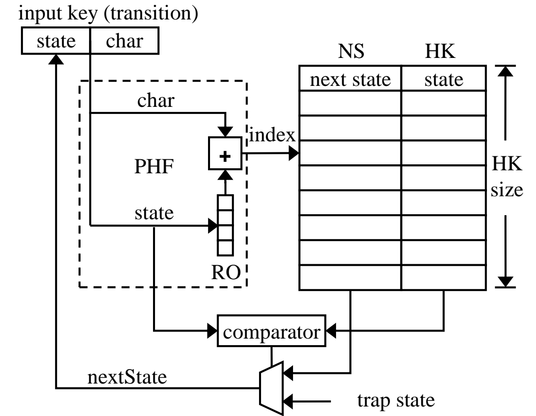


Figure 10. Time-efficient RSPHMA

The procedure to query the next state in the time-efficient RSPHMA is as follows.

i. $index$ = RO[$state$] + $char$;
ii. If HK[$index$] == ***state***
  k is a valid key;
 else
  k is an invalid key;
iii. If k is a valid key
  $nextState$ = NS[$index$];
 else
  $nextState$ = trap state;

D. Column-shifting perfect-hashing memory architecture

Instead of shifting row, shifting column is another way to construct hashing table in [17]. Consider the original two-dimensional state table which has 256 columns, using column-shifting to construct hash table intuitively has two benefits. The first benefit is the size of memory named CO to record column offset is equal to 256 x 4 bytes which is much smaller than the memory used to store row offset in RSPHMA. In RSPHMA, the size of memory named RO used to store row offset is proportional to the number of states.

The second benefit is that the size of hash key is only one-byte long. Since the hash keys stored in HK are used to verify whether an input key is a valid key (transition) or not. In Figure 7, if there are two distinct keys having the same index, the two keys must locate in different columns in the key table. The proof of the proposition is as follows.

PROPOSITION 2. In Column-Shifting Perfect-Hashing algorithm, if there are two distinct keys having the same index, the two keys must locate in different columns in the key table.

Proof. For the sake of contradiction, suppose that if there are two distinct keys having the same index, the two keys must locate in the same column in the key table. In Column-Shifting Perfect-Hashing algorithm, the two keys are as follows.

$$key1 = row1 * width + column$$

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

key2 = row2 *width + column

Since key1 is distinct from key2, row1 is distinct from row2.

Furthermore, in Column-Shifting Perfect-Hashing algorithm, the indices of the two keys are calculated as follows where CO is the table storing offsets of each column.

index1 = CO[column] + row1

index2 = CO[column] + row2

Since row1 is distinct from row2, index1 is distinct from index 2. The result contradicts our assumption that if there are two distinct keys having the same index, the two keys must locate in the same column. ■

For example, the two keys 14 and 19 mapping to the same position (the fourth element) of the hash table locate in the third and fourth columns, respectively. In other words, the correctness of a key can be verified by checking the column number of the key rather than checking the whole key value. Particularly, if the number of column is 256, we can minimize the size of the HK table to exact one-byte long. Compared to the time-efficient RSPHMA, the width of HK table is reduced from four-bytes long to one-byte long.

Although the column-shifting algorithm has two benefits compared to the row-shifting algorithm. The column-shifting algorithm has a major drawback in which the load factor is not satisfied. The load factor represents as the ratio of the number of keys over the number of hash entries. In perfect hashing algorithms, the load factor of a perfect hash table is always less than or equal to one. The load factor close to one means a dense hash table is constructed. On the contrary, a hash table with a small load factor means the hash table is sparse.

In our application, the load factor of column-shifting is not satisfied due to the characteristic that the depth of the two-dimensional table is much larger than its width. Therefore, it is more difficult for column-shifting algorithm to find positions without collisions.

Figure 11 shows the Column-Shifting Perfect-Hashing Memory Architecture (CSPHMA).

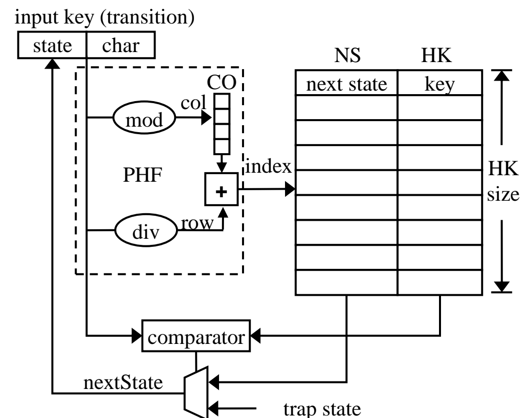


Figure 11. Column-Shifting Perfect-Hashing Memory Architecture (CSPHMA)

The procedure to query the next state in the CSPHMA is as follows.

i. *row* = $\lfloor k / w \rfloor$;
ii. *col* = *k* mod *w*;
iii. *index* = CO[*col*] + *row*;
iv. If HK[*index*] == *k*
    k is a valid key;
  else
    k is an invalid key;
v. If *k* is a valid key
    nextState = NS[*index*];
  else
    nextState = trap state;

V. GPU Implementations and Optimization

In recent years, GPUs have attracted a lot of attention due to its tremendous parallel computing ability and very high memory bandwidth. Several works [16][13][11][7][18] are proposed to accelerate exact and regular expression pattern matching using GPUs.

As discussed in Section II, our previous work, PFAC [3] stores the state transition table in a two-dimensional memory whose size equals the number of states multiplied by 1K bytes (256 column x 4 bytes/column) to achieve best performance. For a state machine of one million states, the memory architecture needs a size 1GBytes two-dimensional memory to store the corresponding state transition table. With the increasing number of patterns, the state transition table will grow too large to fit into the GPU memory. Therefore, to increase the scalability of PFAC to accommodate more patterns, we integrate RSPHMA and CSPHMA to PFAC using CUDA[15] on NVIDIA GPUs.

Compared to the two-dimensional memory architecture, the cost of using perfect hashing includes (1) hash index generation, (2) HK table access, and (3) NS table access. We observe that the perfect hashing architecture is a memory-bound task that has three memory accesses including RO (or CO), HK, and NS tables. To alleviate the cost of memory accesses on GPUs, we discuss

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

several optimization techniques as follows.

A. *Reducing the complexity of hash index computation*

As shown in Figure 6, the calculation of *row* and *col* needs division and modulus operations. However, to alleviate the cost for managing division and modulus operations, we can set the width of the key table, *w,* as power of 2. Then, the division and modulus operations can be replaced by shift-right and bitwise-AND operation, respectively.

In addition, the time-efficient RSPHMA sets the width of the key table as 256, the division and modulus operations can be removed and the *row* and *col* can be replaced by *state* and *char*, respectively. Because the computation of hash indices is reduced, the performance can be improved.

B. *Merging HK and NS tables*

In order to reduce the memory accesses which are expensive in GPU computation, we can merge HK and NS tables to increase performance. Then, the hash key and next state information can be read as one memory access and further performance can be improved. Consider an AC state machine of n states, the size of the traditional two-dimensional table is $n\times256$. If the width of the key table is set to 1,024, the depth of the two-dimensional key table is reduced to $\frac{n}{4}$. Because the number of rows is reduced to $\frac{n}{4}$, the row indices stored in the hash table as keys can be further reduced by 2 bits. Therefore, if we set the width of the key table to 1,024, we can divide an integer of 32 bits as two columns of 17 bits and 15 bits. The column of 17 bits is used to store the next state information while the column of 15 bits is used to store the row of key. In this configuration, the size of the state machine is limited to 131,072 (2^{17}) states. On the other hand, if we adopt the time-efficient RSPHMA to achieve maximum performance, the best way is to use 16 bits to store keys and 16 bits to store next states. In such configuration, the size of the state machine is limited to 65,536 (2^{16}) states. In our experiments, the proposed RSPH and CSPH have two kinds of GPU implementations, one is for smaller state machine which has less than 2^{16} (65,536) states, and the other is for larger state machine which has more than 2^{16} (65,536) states. The former one merges the current state and next state information as a 32-bits key to achieve better performance and smaller hash table size while the latter one separates the current state and next state information to accommodate the state machines having more than 2^{16} (65,536) states.

C. *Binding hash table and offset table to texture memory*

The memory hierarchy of GPU provides on-chip cached texture memory to take the benefits of spatial and temporal locality of date reference. To achieve higher bandwidth, we bind the hash table as well as the RO and CO tables to texture memory instead of off-chip global memory. Although the application of pattern matching exhibits weak locality of data reference, still 10% improvement in throughput is achieved.

D. *Pushing frequent date into shared memories*

Among the memory hierarchy of CUDA, shared memory is the fastest memory, but is very limited. Recall that the PFAC issues continuous threads to each input character. Therefore, a thread has the chance to read the symbols which the consequent threads have read from global memory. Thus, before starting to traverse a state machine, every thread in a block puts a portion of input characters from global memory to shared memory. And then, neighboring threads can read input characters from shared memory instead from global memory. In addition, we observe that most threads terminate after reading the first character. To improve performance, we also put the first row of the state transition table into shared memory.

In the following, we explain the RSPHMA using a piece of code as shown in Figure 12. In the fourth and fifth line, each thread moves two characters from global memory to shared memory. Assume BLOCK_SIZE is 512, the shared memory size for storing input texts is only 1024 bytes. If the maximum pattern length is less than or equal to BLOCK_SIZE, we can ensure that shared memory is sufficient no matter how long the input texts. If the maximum pattern length is greater than BLOCK_SIZE, we can solve the problem by increasing the number of characters moved from global memory to shared memory. This section is part of implementation details and omitted in the pseudo code. According to our experimental environment, as long as the maximum pattern length is less than or equal to 4096 bytes, shared memory is sufficient to store its input texts in a block. From lines 6 to 8, the threads whose tid are 0 to 255 move the first row of the state transition table into shared memory. And then, all threads in the same block are synchronized at line 9 to ensure that the data transfer from global memory to shared memory is completed. After synchronization, line 10 generates the starting position, pos of each thread and line 11 retrieves the first input character from shared memory. Line 12 delivers the next state information of the initial state from shared memory.

In our implementation, a thread terminates anytime when it encounters a trap state which is specified as -1. In Figure 12, line 13 checks the next state of the initial state. If the variable state is greater than or equal to zero, the thread proceeds with the process from line 14. On the contrary, if the state variable is -1 which represents a trap state, the thread terminates immediately. Lines 14-16 check whether the state is a final state. In our implementation, if a state machine has n final states, the n final states are encoded from zero to n-1. Therefore, if the state variable is less than n, the state is a final state and then is assigned to the match array which is also stored in shared memory for speedup. The while loop in lines 18-38 continues traversing the state machine until a trap state is met. The key is generated by combining the state and input character at line 21. Lines 22 and 23 calculate row and col variables, respectively. Line 24 calculates the hash index by adding col and the row offset which is retrieved from the texture memory. If the hash index is greater than or equal to the hash table size, a trap state is set at line 26;

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

otherwise, line 28 retrieves hash value from the hash table stored in texture memory. The row information in a hash key located at the lower 15 bits of a hashValue is compared with the row of an input key at line 29. If the comparison returns true, line 30 updates the state variable as the next state information located at the higher 17 bits of the hashValue; otherwise, line 32 sets the state variable as a trap state and the thread terminates immediately at line 34. From lines 35 to 37, if the state is a final state, the state information is assigned to the match array. Finally, match results are moved from shared memory to global memory in line 42.

```
1   gbid = blockIdx.y * gridDim.x + blockIdx.x;
2   tid = threadIdx.x;
3   start = gbid * BLOCK_SIZE + tid ;
4   sharedInput[tid] = globalInput[start];
5   sharedInput[tid+BLOCK_SIZE] = globalInput[start+BLOCK_SIZE];
6   if (tid < 256) {
7       s_s0Table[tid] = d_s0Table[tid];
8   }
9   __syncthreads();
10  pos = tid;
11  inputChar = sharedInput[pos];
12  state = s_s0Table [inputChar];
13  if (state >= 0) {
14      if (state < num_final_state) {
15          s_match[tid] = state;
16      }
17      pos += 1;
18      while (1) {
19          if (pos >= boundary ) break;
20          inputChar = sharedInput [pos];
21          key = (state << 8) + inputChar;
22          row = key >> width_bit;
23          col = key & ((1<<width_bit)-1);
24          index = tex1Dfetch(tex_RO, row) + col;
25          if (index >= HTSize)
26              state = -1; //trap state
27          else {
28              hashValue= tex1Dfetch(tex_HT, index);
29              if ((hashValue & 0x7FFF) == row)
30                  state=(hashValue >> 15) & 0x1FFFF ;
31              else
32                  state = -1; //trap state
33          }
34          if (state == -1) break;
35          if (state < num_final_state) {
36              s_match[tid] = state;
37          }
38          pos += 1;
39      }
40  }
41  //move match result from shared memory to global memory
42  d_match[start] = s_match[tid];
```

Figure 12. A piece of code in RSPHMA

VI. EXPERIMENTAL RESULTS

The experimental environment is composed of a host machine and a device machine. The host machine is equipped with an Intel® Core™ i7-3770 running the Linux X86_64 operating system with 16GB DDR3 memory while the device machine is equipped with an Nvidia® GeForce® GTX680 GPU with 2,048 MB GDDR5 memory and an Nvidia® GeForce® GTX TITAN X GPU with 12GB GDDR5 memory. The version of CUDA toolkit is 7.0. The test patterns are extracted from Snort V2.8 where the length of exact patterns varies between one to 243 characters long. We divide the Snort patterns into two sets; the large one contains 10,076 patterns of total 187,329 characters, while the small one contains 1,998 patterns of total 41,997 characters. The former state machine has 126,776 states while the latter machine has 27,754 states. The proposed architectures are tested using DEFCON [6] packets which contain large amounts of real attack patterns. The size of the extracted DEFCON packets is 256 MB. In Table I, we compare several recent published memory architectures [1][3][8][13][21] with our proposed RSPH and CSPH architectures. In addition, we also implement single-threaded and multi-threaded AC and PFAC algorithms on CPU for comparisons.

The GPU and CPU implementations are described as follows.

1) RSPH: implementation of the Row-Shifting Perfect Hashing algorithm on GPU using single stream. The RSPH can handle both large and small pattern benchmarks.
2) CSPH: implementation of the Column-Shifting Perfect Hashing algorithm on GPU using single stream. The CSPH can handle both large and small pattern benchmarks.
3) PFAC: Our previous work on GPU [3]
4) AC_CPU: Traditional single-threaded Aho-Corasick algorithm on CPUs
5) AC_CPU_OMP: Traditional multi-threaded Aho-Corasick algorithm on CPUs parallelized by OpenMP
6) PFAC_CPU: single-threaded Parallel Failureless Aho-Corasick algorithm on CPUs
7) PFAC_CPU_OMP: multi-threaded Parallel Failureless Aho-Corasick algorithm on CPUs parallelized by OpenMP

In Table I, columns 2, 3, 4, 5, 6, 7, and 8 show number of rules, number of characters, number of states, number of transitions, total memory usage, memory efficiency, and load factor, while column 9 and 10 show the kernel throughput and system throughput, respectively.

The following memory efficiency is defined to represent the memory requirements (Bytes) per character.

$$\text{memory efficiency} = \frac{\text{memory size}}{\text{number of characters}} \quad (1)$$

The kernel throughput and system throughput are defined as follows.

$$\text{kernel throughput} = \frac{\text{input size}}{\text{elapsed time of kernel launch}} \quad (2)$$

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

$$\text{system throughput} = \frac{\text{input size}}{\text{elapsed time of kernel launches and data transfers}} \quad (3)$$

As mentioned above, our previous work, PFAC [3] adopts a two-dimensional memory to store a PFAC state transition table whose size equals the number of states multiplied by 1K bytes (256 column x 4 bytes/column) to achieve the best performance. For example, considering the state machine of 27,754 states, the two-dimensional memory size is 27M bytes (27,754 x 1K bytes).

In our experiments, the proposed RSPH and CSPH have two kinds of GPU implementations. One is for smaller state machine which has less than 2^{16} (65,536) states, and the other is for larger state machine which has more than 2^{16} (65,536) states. Then, we chose two sets of patterns, the larger one has 126,776 states, and the smaller one has 27,754 states. Since the smaller one has only 27,754 states which can be encoded using 15 bits, we can merge the current state and next state information into a 32-bits word as a key. On the other hand, since the larger one has 126,776 states which has to be encoded using more than 16 bits, we cannot merge the current state and next state information as a 32-bits key. Therefore, the hash table for storing 126,776 states is twice the size of the one for storing 27,754 states. In addition, since the small one merges the current state and next state information as a 32-bits key, the performance of the small one is better than the large one which separates the current state and next state information. In addition, the small one allocates a 16-bits linear array to store match results that can save the data transmission time via PCIe while the large one allocates a 32-bits linear array to store match results. As a result, the small one has better system throughput than the large one.

In terms of memory consumption, the proposed RSPH and CSPH architectures consume 217KB and 213KB of memory for processing the small rule set containing 1,998 rules, respectively. Both the RSPH and CSPH architectures achieve more than 99% of memory reduction compared to the two-dimensional memory architecture. For processing the Snort rule set of 10,076 rules, the CSPH consumes 781KB memory while the RSPH consumes 1,485KB memory. The experimental results show that the CSPH achieves better memory reduction than the RSPH.

In terms of memory efficiency, the CSPH outperforms most state-of-the-art memory compression techniques approaches [1][8][13] [21]. In addition, all these state-of-the-art approaches need specific hardware such as TCAM, FPGA or ASIC. For example, both CompactDFA[1] and BFPM[8] approaches introduce wildcard symbol "*" to represent "don't care" condition in their rule sets to compress state transition rules. Both approaches share the same situation that a transition may simultaneously match multiple rules. In order to resolve the problem, the former approach, CompactDFA [1] resolves the problem by applying the longest prefix matching problem while the latter approach, BFPM [8] prioritizes the transition rules and adopts the Balanced Routing Table (BART) search algorithm for exact-, prefix- and range-match searches. In addition, both [1] and [8] approaches need specific hardware for storing and searching their compressed rule sets. The CompactDFA addresses the pattern matching problem by adopting the longest prefix matching problem such as the IP-lookup problem. In addition, CompactDFA stores the compressed rule sets on Ternary Content Addressable Memory (TCAM) which permits rapid table lookups through longest prefix matches. However, high speed TCAMs are extremely expensive with high power usage and take up quite a bit of silicon space. On the other hand, the BFPM stores its rule sets in a specific transition-rule memory implemented on FPGA or ASIC.

Our proposed approaches are also based on the concept of state transition rules, but with two major differences compared to [1] and [8]. First, our approaches reduce the storage space for a PFAC [4] state machine which is much smaller than a traditional DFA, which has 256 transitions for each state and is reduced by the [1] and [8] approaches. Not to mention, the size of a PFAC state machine is generally less than 1% of a traditional DFA state machine. Second, our proposed approaches do not need specific hardware and can be easily implemented on commodity DRAM. This is more cost-effective than the [1] and [8] approaches which require specific hardware such as TCAM, FPGA or ASIC to accelerate longest prefix matching and wildcard "*" matching, respectively.

In table I, AC_CPU denotes single-threaded CPU implementation of traditional Aho-Corasick algorithm while AC_CPU_OMP denotes multi-threaded CPU implementation of traditional Aho-Corasick algorithm optimized by OpenMP. PFAC_CPU denotes single-threaded CPU implementation of Parallel Failureless Aho-Corasick algorithm while PFAC_CPU_OMP denotes multi-threaded CPU implementation of Parallel Failureless Aho-Corasick algorithm optimized by OpenMP. In terms of kernel and system throughputs, multi-threaded PFAC_CPU_OMP achieves an average of 6.9 and 3.4 times faster than single-threaded PFAC_CPU and traditional AC algorithm, respectively. On the other hand, the proposed RSPH performed on TITAN X achieves an average of 25.5 and 10.4 times faster than PFAC_CPU_OMP on kernel throughput for processing the small and large pattern sets, respectively. However, compared with the best multi-threaded PFAC_CPU_OMP, the proposed GPU implementations cannot have significant improvement due to the bottleneck of data transmission via PCIe.

VII. Conclusions

In this paper, we have proposed two means of parallel string matching algorithms which adopts perfect hashing to compact a state transition table. Different from most state-of-the-art approaches which need specific hardware such as TCAM, FPGA, or ASIC, our proposed approaches do not need specific hardware and can be easily implemented on commodity DRAM. Our

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

proposed algorithms are extremely suitable to be implemented on GPUs. The time and space complexity of the proposed algorithms have been evaluated and compared with state-of-the-art approaches as well as the traditional AC algorithm on multicore CPUs. Experimental results show that the proposed perfect-hashing based parallel algorithms achieve significant memory reduction and performance when performed on NVIDIA GPUs.

TABLE I. COMPARISONS WITH STATE-OF-THE-ART MEMORY REDUCTION APPROACHES

Name	# of Rules	# of characters	# of states	# of transitions	Memory size (bytes)	Memory efficiency	Kernel Throughput (Gbps)	System Throughput (Gbps)	Platform
RSPH	10,076	187,329	126,776	126,775	1,485KB	8.12B	55.73	14.89	NVIDIA GTX680
CSPH	10,076	187,329	126,776	126,775	781KB	4.27B	50.56	14.44	
PFAC	10,076	187,329	126,776	126,775	126MB	693B	83.32	16.39	
RSPH	1,998	41,997	27,754	27,753	217KB	5.29B	108.83	25.87	
CSPH	1,998	41,997	27,754	27,753	213KB	5.21B	95.52	24.74	
PFAC	1,998	41,997	27,754	27,753	27MB	677B	143.37	27.46	
RSPH	10,076	187,329	126,776	126,775	1,485KB	8.12B	136.50	17.55	NVIDIA TITAN X
CSPH	10,076	187,329	126,776	126,775	781KB	4.27B	120.26	17.03	
PFAC	10,076	187,329	126,776	126,775	126MB	693B	124.58	17.28	
RSPH	1,998	41,997	27,754	27,753	217KB	5.29B	333.92	29.18	
CSPH	1,998	41,997	27,754	27,753	213KB	5.21B	262.94	30.07	
PFAC	1,998	41,997	27,754	27,753	27MB	677B	385.46	31.04	
AC_CPU	10,076	187,329	126,776	126,775	126MB	693B	2.01	2.01	Intel Core i7-3770
AC_CPU_OMP	10,076	187,329	126,776	126,775	126MB	693B	1.72	1.72	
PFAC_CPU	10,076	187,329	126,776	126,775	126MB	693B	4.13	4.13	
PFAC_CPU_OMP	10,076	187,329	126,776	126,775	126MB	693B	13.90	13.90	
AC_CPU	1,998	41,997	27,754	27,753	27MB	677B	2.59	2.59	
AC_CPU_OMP	1,998	41,997	27,754	27,753	27MB	677B	2.08	2.08	
PFAC_CPU	1,998	41,997	27,754	27,753	27MB	677B	5.23	5.23	
PFAC_CPU_OMP	1,998	41,997	27,754	27,753	27MB	677B	18.03	18.03	
B-FSM[8] (4 subsets)	39.5K	25.2K	n/a	n/a	188KB	7.46B	n/a	2	ASIC/FPGA
CDFA[21] (4 subsets)	1,785	29.0K	n/a	n/a	181KB	6.2B	n/a	11.7	ASIC/FPGA
Bitmap Compression[13]	1.5K	18.2K	n/a	n/a	2.8MB	154B	n/a	7.6	ASIC
Path Compression[13]	1.5K	18.2K	n/a	n/a	1.1MB	60B	n/a	7.6	ASIC

REFERENCE

[1] A. Bremler-Barr, D. Hay, Y. Koral, “CompactDFA: Generic State Machine Compression for Scalable Pattern Matching,” in *INFOCOM 2010. The 29th Conference on Computer Communications. IEEE*, pp. 659-667, 2010.

[2] A. V. Aho and M. J. Corasick, "Efficient String Matching: an Aid to Bibliographic Search," *Commun. ACM*, vol. 18, pp. 333-340, 1975.

[3] C.-H. Lin, C.-H. Liu, L.-S. Chien, S.-C. Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 1906-1916, Oct. 2013

[4] C.-H. Lin and S.-C. Chang, "Efficient Pattern Matching Algorithm for Memory Architecture" *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 19, No.1, pp.33 - 41, 2011.

[5] C.-H. Lin, C.-H. Liu, S.-C. Chang, and W.-K. Hon, "Memory-Efficient Pattern Matching Architectures Using Perfect Hashing on Graphic Processing Units," *31st Annual IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[6] DEFCON, Available: http://cctf.shmoo.com

[7] G. Vasiliadis , M. Polychronakis, S. Antonatos , E. P. Markatos, and S. Ioannidis, “Regular Expression Matching on Graphics Hardware for Intrusion Detection,” In *Proc. 12th International Symposium on Recent Advances in Intrusion Detection*, 2009.

[8] J. V. Lunteren, "High-Performance Pattern-Matching for Intrusion Detection", *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pp. 1 - 13, April 2006.

[9] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," in *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, 2005, pp. 112-122.

[10] M. Becchi, and S. Cadambi, “Memory-Efficient Regular Expression Search Using State Merging,” In *Proc. of the 26th IEEE*

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2674664, IEEE Transactions on Parallel and Distributed Systems

International Conference on Computer Communications (INFOCOM). 1064–1072, 2007.

[11] M. C. Schatz and C. Trapnell, "Fast Exact String Matching on the GPU," Technical report.

[12] M. Roesch. "Snort- Lightweight Intrusion Detection for networks," in *Proceedings of LISA99, the 15th Systems Administration Conference*, 1999.

[13] N. F. Huang, H. W. Hung, S. H. Lai, Y. M. Chu, and W. Y. Tsai, "A Gpu-based Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems," in *Proc. 22nd International Conference on Advanced Information Networking and Applications (AINA)*, 2008, pp. 62–67.

[14] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," in *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, 2004, pp. 2628-2639 vol.4.

[15] NVIDIA Corporation. NVIDIA CUDA programming Guide, 2010 Available: http://developer.nvidia.com

[16] PFAC library, Available: http://code.google.com/p/pfac/

[17] R. E. Tarjan and A. C.-C. Yao, "Storing a Sparse Table," Commun. ACM, vol. 22, pp. 606-611, 1979.

[18] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, C. Estan, "Evaluating GPUs for Network Packet Signature Matching," in *Proc. of the International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2009.

[19] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "HEXA: Compact Data Structures for Faster Packet Processing," in *Proc. of IEEE ICNP'07*, Beijing, China, October, 2007.

[20] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in ACM SIGCOMM, 2006.

[21] T. Song, *et al.,* "A Memory Efficient Multiple Pattern Matching Architecture for Network Security," in *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, 2008, pp. 166-170.

[22] Y. Xu, L. Ma, Z. Liu, and H. J. Chao, "A Multi-Dimensional Progressive Perfect Hashing for High-Speed String Matching," in *Proc. 17th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2011.

Cheng-Hung Lin (S'06-M'08) received the Ph.D. degree in computer science from National Tsing Hua University in 2008. He is currently an associate professor with the department of electrical engineering, National Taiwan Normal University. His current research interests include parallel computing, multicore programming, and parallel algorithm design.

Jin-Cheng Li (S'13) received the B.S. degrees in computer science from National Tsing-Hua University in 2013.He is currently working towards the M.S. degree in the department of computer science, National Tsing-Hua University. His research interests include network intrusion detection, GPU programming and related computer-aided design (CAD) techniques.

Chen-Hsiung Liu received the B.S. and M.S. degrees in computer science from National Tsing-Hua University in 2009 and 2011, respectively. In 2011, he joined MStar Semiconductor, Inc. His research interests include network intrusion detection, GPU programming and related computer-aided design (CAD) techniques.

Shih-Chieh Chang (S'92–M'95) received the B.S. degree in electrical engineering from National Taiwan University in 1987 and the Ph.D. degree from the University of California, Santa Barbara in 1994. He worked at Synopsys, Inc. in mountain view, CA, from 1995 to 1996. He is now a professor in the department of computer science in National Tsing-Hua University. Professor Shih-Chieh Chang is currently the executive director of national program for intelligent electronics in Taiwan and also an Associate Editor of ACM Transaction on Design Automation of Electronic System. He has published more than 100 technical papers and has served in the several Program committees such as ICCAD, DAC, ICCD, ISQED, and ASPDAC. His current research interests include low power and low energy optimization, variation aware optimization and tolerance, 3D design methodology.

1045-9219 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.