Perfect Hashing of String Matching on GPU

Lung-Sheng Chien***, Wing-Kai Hon**, Cheng-Hung Lin*, Chen-Hsiung Liu**
*National Taiwan Normal University, Taipei, Taiwan
**Dept. of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
*** Dept. of Mathematics, National Tsing Hua University, Hsinchu, Taiwan

***Abstract*—traditional state-machine based string matching algorithm allocates a 2-D array to store state transition table which represents matched patterns. This pays O(1) for table lookup but consumes a lot of memory, more than 99% of 2-D array are empty. Such 2-D array is not feasible if patterns are huge. Perfect hashing technique is a good tool to compress a sparse dataset into a compact array with O(1) random access time. However it needs two hashing functions which heavily rely on modulo operations. In this paper we combine perfect hashing technique and previous work of exact string matching on GPU. Since modulo operation is not efficient on GPU computing, we propose a modulo-free perfect hashing algorithm which replaces modulo operations by easy shift/and operations. Besides we have a priori bound on size of hash table. In our experiments, modulo-free perfect hashing is at least 4x faster than traditional perfect hashing on GPU. Moreover the former has better compression rate than the latter.**

I. INTRODUCTION

String matching has long history and is popular adopted in Network Intrusion Detection Systems (NIDS) and DNA analysis. The authors have developed a PFAC library [1] which is a variant of well-known Aho-Corasick (AC) algorithm but removing all failure transitions and self-loop transitions of initial state. For example, given a set of 10 patterns, "s", "h", "he", "she", "hers", "her", "his", "iis", "is" and "ii", PFAC library constructs a finite state machine, called PFAC state machine, depicted in Figure 1. The state is represented by a circle within a number, and an edge is a valid transition, for example, state 11 with input character 'h' would go to state 2, so there is an edge from state 11 to state 2, abbreviated as (current state = 11, input character = 'h') -> (next state = 2) . We only show valid transitions and other combinations are invalid. The number of states is 14, labeled from 1, 1,2, ..., 14, and initial state is 11 (in our implementation, state 0 is useless). There are 10 final states (orange circle, labeled from 1,2,...,10) corresponding to 10 patterns respectively. Common prefix of two patterns would share same transition path, for example, "hers" and "his" have common prefix "h", so transition (11, 'h')->2 is shared. Given an input string of length m, PFAC creates m threads, thread j checks if any pattern matches a substring starting at position j. For example, suppose input string is "hershey", then 6 threads handle 6 substrings in Figure 2. Thread 0 matches four patterns, "h", "he", "her" and "hers" and reports longest pattern "hers". Thread 1 does not match any pattern starting by character 'e'.

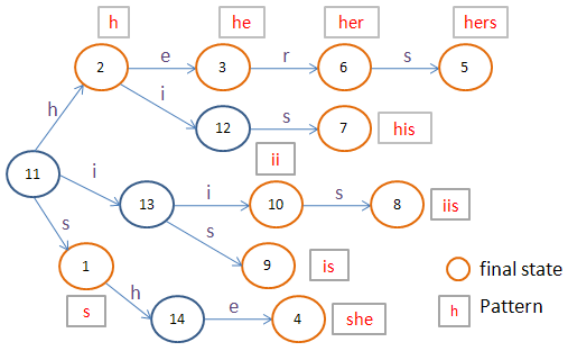

Figure 1: PFAC state transition diagram of patterns "s", "h", "he", "she", "hers", "her", "his", "iis", "is", "ii".

The first version of PFAC library adopts explicit 2D array to store transition table as Figure 3, however this is not space-efficient because of tree-structure of PFAC state machine. Thanks to removal of failure transitions and self-loop, PFAC state machine has no circuits, and each path starting from initial state (root) would end at ONLY one final state (leaf node). In order to simplify discussion, we define several parameters:

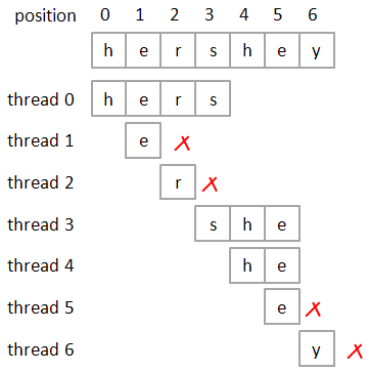

Figure 2: PFAC uses 7 threads to process an input string of size 7, thread j processes substring starting at position j.

F denotes number of final states (number of patterns),
S denotes number of states,
R denotes number of valid transitions,
L denotes number of leaf nodes, and
C denotes number of characters and C is 256 in our work.
For example, $F = 10$, $S = 14$, $R = 13$, $L = 5$ in Figure 1.

Remark 1: five leaf nodes are 4, 5, 7, 8 and 9, contains part of final states but not all because patterns share common prefix substring, and then final state of a shorter

one becomes internal node. For example, 'h" is prefix of "he", so final state of "h" is an internal state.

As we know, tree structure has an important property: number of edges is equal to number of internal nodes plus 1. The consequence is $R=S-1$. However 2-D array needs $S*C$ integers, PFAC library wastes 99% ($1 = \frac{R}{S*C}$) storage.

Space complexity is very important on huge problems, for example, typical size of EST database in DNA analysis is GB ([2] and [3]). If we use 2-D array to store a transition table, then we need 1TB memory. Unfortunately current high-end GPUs only support up to 5.25GB RAM (C2070 in Table 1). In other words, if we still want to process such huge database, then the only way is to split dataset into 1000 small patterns, and process one input string against these 1000 patterns, then merge 1000 partial results. This would incur large overhead on memory transfer, one should remember that time complexity of string matching is O(M) where M is size of input string, and overhead of merging two partial results is also O(M). Second, GPU has small L2 cache (768KB) shared by thousands of threads, compression of transition table can also improve cache hit rate. Third, the smaller transition table is, the larger input string can be processed without splitting.

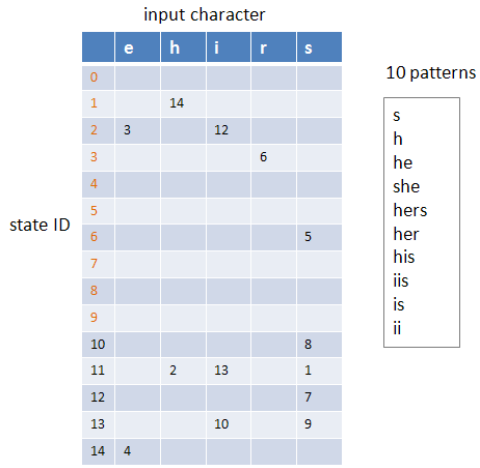

Figure 3: PFAC state transition table, which is expressed as 2-D array and only valid transitions are shown. The number of columns is 256, only 5 among them have valid transitions and others are not shown.

Table 1: Specification of host and device [8]

Specification	host	device	Device
	Core i7 930	GTX480	Tesla C2070
Core frequency	2.8 GHz	1.4 GHz	1.15 GHz
# of Streaming Multiprocessor (SM)	4	15	14
# of Cores	4x4	480	448
Single precision peak perf.	44.8 Gflops	1.344 Tflops	1.03 Tflops
Double precision peak perf	22.4 Gflops	168 Gflops	515 Gflops
Bandwidth GB/s	25.6	177.4	120(ECC on)
Memory	12GB	1.5GB	5.25GB
L1 cache per SM	32KB	16KB	16KB
On-chip shared memory	N/A	48KB per SM	48KB per SM
L2 cache	256KB per SM	768KB for all SMs	768KB for all SMs
L3 cache	8 MB	N/A	N/A

The first question is: what is optimal storage of such transition table? Actually transition table in Figure 3 is a sparse matrix, if we use CSR (Compressed Sparse Row) format to compress this sparse matrix, then it is easy to estimate size of optimal storage. CSR format only records nonzero entries of a sparse matrix A in a triplet $(i,j,A(i,j))$, and removes repeated row indices. Three arrays csrRowPtr, csrColPtr and csrValPtr are used. csrRowPtr[i] is an offset of first nonzero element of row i in csrColPtr and csrValPtr. Also csrRowPtr[$i+1$] - csrRowPtr[i] is number of nonzero elements in row i. csrRowPtr[0] is always 0 and csrRowPtr[m+1] is number of nonzero entries of A. csrColPtr contains column indices of nonzero elements. csrColPtr[csrRowPtr[i]: csrRowPtr[$i+1$]-1] = $\{j : A(i,j)$ is nonzero$\}$. csrValPtr contains nonzero matrix elements. csrValPtr[csrRowPtr[i]: csrRowPtr[$i+1$]-1] =$\{A(i,j)$: $A(i,j)$ is nonzero$\}$. Figure 4 shows CRS format of Figure 3, number of rows is 15 (= number of states) and number of nonzero entries is 13 (= number of valid transitions). csrRowPtr has 15+1 values whereas csrColPtr and csrValPtr have 13 values. State 0 has no valid transitions so csrRowPtr[1] - csrRowPtr[0] = 0. State 2 has two valid transitions, (2, 'e')->3 and (2, 'i')->12, and starting position of first transition is csrRowPtr[2] = 1, so csrColPtr[1:2] = {'e', 'i'} and csrValPtr[1:2] = {3, 12}. One can refer to manual of CUSPARSE library [4] for description of CSR format.

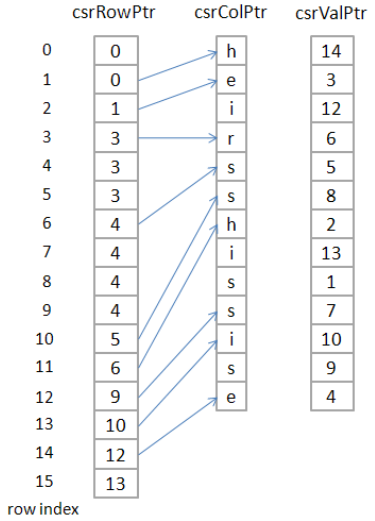

Figure 4: CSR format of sparse matrix in Figure 3.

Total memory of CSR format amounts to $S+1+2R$ integers, only 3/256 of 2-D array which needs $S*C$ integers. So CSR format is perfect on space complexity but it has a vital drawback on GPU computing, load imbalance. GPU is a vector machine and length of vector is 32. So 32 threads form a basic vector, called a warp. Threads within a warp share one program counter, so they execute same instruction but operate on different

data. If two threads of a warp have different paths, for example, thread 0 has true predicate in if-then-else statement and other threads have false predicate, then all threads of a warp must execute both paths. We call this divergent branch, and it is the root of load unbalance. Figure 5 is pseudo-code of PFAC algorithm, it has divergent branch in predicate of while loop (line 4) because each thread may process different length of substring. However it does not have divergent branch on table lookup (line 6) because every thread executes same instruction
"LOAD r0, PFAC_table + map(state,ch)".
However this property does not hold on CSR format.

```
1   // tid is thread ID
2   int pos = tid;
3   int state = 0; // initial sate
4   while ( (state != TRAP) && (pos < size) ) {
5       char ch = input_buffer[pos];
6       int nextState = PFAC_table[state][ch];
7       pos = pos + 1;
8       if (nextState is a final state ) {
9           match_result[tid] = pattern ID;
10      }
11      state = nextState;
12  }
```

Figure 5. PFAC algorithm

We replace "PFAC_table[state][ch]" in PFAC algorithm by lookup table of CSR format in Figure 6. Given a pair of (current state *state*, input character *ch*) csrRowPtr[*state*] is starting position of first valid transition of *state*, and csrRowPtr[*state+1*]-1 is position of last valid transition of *state*. We need a linear search for each valid transition of *state* (line 6), and this is the cause of load imbalance. The worst case happens if (*state*, *ch*) is not a valid transition, then all valid transitions of *state* must be queried. Suppose a warp checks for-loop of line 6 at T1 in Figure 7 and thread 1 has to check largest number of transitions, then all 31 threads cannot leave the loop until thread 1 completes its searching at T2.

```
1  int lookup(int state, int ch)
2  {
3      int start = csrRowPtr[state];
4      int end = csrRowPtr[state+1];
5      int nextState = TRAP;
6      for(int j = start ; j < end ; j++){
7          int col = csrColPtr[j];
8          if ( col == ch ){
9              nextState = csrValPtr[j];
10             break;
11         }
12     }
13     return nextState;
14 }
```

Figure 6: linear search of CSR format.

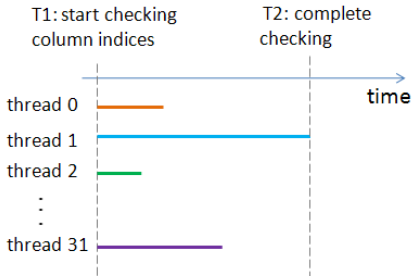

Figure 7: load imbalance of table lookup within a warp.

Although CSR format achieves optimal space complexity, we must give up and find another solution because load imbalance hurts GPU. The key is: membership problem on CSR format is not a constant time, it depends on how many nonzero elements per row. This motivates a perfect hashing solution on string matching. Basically we need following requirement:
Given a sparse 2-D transition table U, we need a compressed data structure W satisfying two properties:
(1) $|W| \leq \alpha|U|$ where α is independent of sparse pattern of U. In practice, α should be much smaller than C.
(2) constant access time for any element (valid or invalid) of U.
We have shown that 2-D array satisfies property 2 but fails on property; CSR format satisfies property 1 but sacrifices property 2.

Remark 2: in this work we don't solve load imbalance of line 4 in Figure 5.

We review traditional perfect hashing in section 2 and show its drawbacks on GPU computing. A new perfect hashing scheme is proposed in section 3, it avoids modulo operations but use shift/and operations instead, we call this new version as modulo-free perfect hashing. Moreover we prove main theorem which gives a priori bound of size of hash table. Two kinds of benchmarks shows performance of different perfect hashing algorithms, and prove effectiveness of modulo-free version. Finally we have a short conclusion in section 5.

II. REVIEW PERFECT HASHING

Michael L. Fredman et al [5] proposed O(1) access time on a two-level hash table in 1984. Moreover size of this hash table is bounded by original sparse table. In other words, Fredman's work satisfies property 1 and 2 simultaneously. We reformulate key lemma 1 of [5] in the following:

Lemma 1: Let $U = \{1,2, ..., m\}$ and $p = m + 1$ is a prime. Given $W \subseteq U$ with $|W| = r$, and $k \in U$, we intend to redistribute W into s bins ($s \geqq r$) by a mapping function $f: W \to [0,s)$ defined by $f(x) = (kx \bmod p) \bmod s$. Let $B(s,W,k,j)$ is the number of elements in bin j, formally, $B(s,W,k,j) = |\{x: x \in W, f(x) = j\}|$ for

$1 \leqq j \leqq s$. Then there exists a $k \in U$ such that

$$\sum_{j=0}^{s-1} \binom{B(s,W,k,j)}{2} < \frac{r^2}{s}$$

According to Lemma 1, the sufficient condition to avoid collisions is to choose $s = r^2$, however this is unacceptable because 1000 transitions need 1000,000 locations. In practice, we choose $s = r$ and have an assertion from Lemma 1 : there exists a $k \in U$ such that $\sum_{j=0}^{r-1} B(r,W,k,j)^2 < 3r$. We take Figure 3 as an example, first we map each valid transition (*state*, *ch*) to an integer (=*state**256+*ch*) exclusively, for example, (1, 'h') is 360, (2, 'e') is 613 in left panel of Figure 8. Obviously, W is the collection of *id* field in Figure 8. Prime p is arbitrary but bigger than 13, here we choose p as 179424691 from [6]. The number of bins is s (=13) and if k is 3, then $\sum_{j=0}^{r-1} B(r,W,k,j)^2 = 25$. However collisions occur at bin 1, 5, 6, and 7. This is reasonable because (k,p) is used to shuffle W but cannot guarantee that there exists a $k \in U$ such that no collision occurs except $s = r^2$.

transition	id
(1, h)	360
(2, e)	613
(2, i)	617
(3, r)	882
(6, s)	1651
(10, s)	2675
(11, h)	2920
(11, i)	2921
(11, s)	2931
(12, s)	3187
(13, i)	3433
(13, s)	3443
(14, e)	3685

bin ID			
0	(6,s)		
1	(1,h)	(11,i)	
2			
3	(13,i)		
4	(10,s)		
5	(2,i)	(11,s)	(14,e)
6	(2,e)	(12,s)	
7	(3,r)	(13,s)	
8			
9			
10			
11	(11,h)		
12			

Figure 8: distribution of transitions into 13 bins under prime p = 179424691, k = 3, and s = 13. Collisions occur at bin 1, 5, 6 and 7.

How could we remove collisions? The answer is very simple: replay Lemma 1 again on each bin. For example, bin 1 has two entries (1,'h') and (11,'i'), and if we use 4 bins ($s_1 = 4$) to separate these two entries, then Lemma 1 guarantees existence of $k_1 \in U$ of collision free. In other words, for each bin j, $B(r,W,k,j)$ is nonzero, we can set $s_j = B(r,W,k,j)^2$ and find a $k_j \in U$ such that triplet (p, k_j, s_j) can separate entries in bin j via new mapping function $f_j(x) = (k_j x \bmod p) \bmod s_j$. The bound of storage is $\sum_{j=0}^{r-1} s_j < 3r$ which is consequence of $s_j = B(r,W,k,j)^2$ and $\sum_{j=0}^{r-1} B(r,W,k,j)^2 < 3r$

This is a two-level hash table, use first triplet (p,s=*13*,k=*3*) to determine *row* index and then use second triplet (p, k_j, s_j) to determine *column* index. For example, (1,'h') and (11,'i') belongs to bin 1 (row=1) under first triplet. Also bin 1 has second triplet $(p, k_1 = 1, s_1 = 4)$ from right table of Figure 10. Then (1,'h') is column 0 and (11,'i') is column 1 under second triplet. Though only two entries in bin 1, we still allocate four locations to ensure collision free. The new locations of valid transitions are show in left table of Figure 10, and hash table is a 2-D array with variable columns shown in Figure 9. There are some comments of implementation on GPU

bin ID									
0	(6,s)								
1	(1,h)	(11,i)							
2									
3	(13,i)								
4	(10,s)								
5					(14,e)	(2,i)	(11,s)		
6		(2,e)		(12,s)					
7			(3,r)	(13,s)					
8									
9									
10									
11	(11,h)								
12									

Figure 9: two-level hash table, each bin contains variable entries.

2.1 modulo operation

From CUDA programming guide [8], NVIDIA's GPU has no 64-bit integer operations and no hardware instructions to do modulo operation, so compiler needs to use several 32-bit instructions to simulate 64-bit integer operations as well as modulo operations. Perfect hashing heavily uses modulo operation on hash functions $f(x) = (kx \bmod p) \bmod s$ and $f_j(x) = (k_j x \bmod p) \bmod s_j$. There are two approaches to do modulo, one 64-bit integer operation and the other is 64-bit float point operation.

We use 64-bit float point (double precision) arithmetic to compute modulo operation. More precisely, $(x \bmod p)$ is replaced by $x - \left\lfloor \frac{x}{p}(1+\epsilon) \right\rfloor p$ where $0 < \varepsilon < \frac{1}{x+p+1}$. If p=*179424691* and $\varepsilon = 10^{-14}$ (near machine zero of double precision), then we have a upper bound on x, say $x < 10^{13}$. Besides division on GPU is much more expensive than multiplication because GPU has no dedicated hardware to do IEEE-compliant division. So we prefer to compute $\frac{1+\epsilon}{p}$ and $\frac{1+\epsilon}{s_j}$ by CPU. So $(kx \bmod p) \bmod s$ is realized by

```
1  double xk_db = k_db * (double)x ;
2  double quotient = floor( xk_db * one_over_p );
3  double r = xk_db - p_db * quotient ;
4  quotient = floor( r * one_over_s );
5  r = r - s_db * quotient ;
```

transition	id	new loc.
(1, h)	360	(1, 0)
(2, e)	613	(6,1)
(2, i)	617	(5,5)
(3, r)	882	(7,2)
(6, s)	1651	(0 ,0)
(10, s)	2675	(4,0)
(11, h)	2920	(11,0)
(11, i)	2921	(1,1)
(11, s)	2931	(5,6)
(12, s)	3187	(6,3)
(13, i)	3433	(3,0)
(13, s)	3443	(7,3)
(14, e)	3685	(5,4)

bin	offset	(k, s)
0	0	(1, 1)
1	1	(1, 4)
2	-1	(0 , 0)
3	5	(1, 1)
4	6	(1, 1)
5	7	(1, 9)
6	16	(1, 4)
7	20	(1, 4)
8	-1	(0, 0)
9	-1	(0, 0)
10	-1	(0, 0)
11	24	(1, 1)
12	-1	(0, 0)

Figure 10: two-level hash table, first triplet is (*p=179424691, k=3, s=13*), and second triplet varies in different bins.

2.2 data structure of hash table

We adopt idea of CSR format to represent hash table in Figure 9, only csrRowPtr and csrValPtr are used, but csrColPtr is ignored because column index can be computed by $f_j(x) = (k_j x \bmod p) \bmod s_j$, so we don't need to search column index. In order to distinguish hash table from general sparse matrix, we replace csrRowPtr and csrValPtr by hashRowPtr and hashValPtr. However additional information should be put into hashRowPtr and hashValPtr. In Figure 11 hashRowPtr contains not only an offset but also $\left(k_j, s_j, \frac{1+\epsilon}{s_j}\right)$ because first triplet (*p,s,k*) is universal but second triplet (p, k_j, s_j) varies per bin.

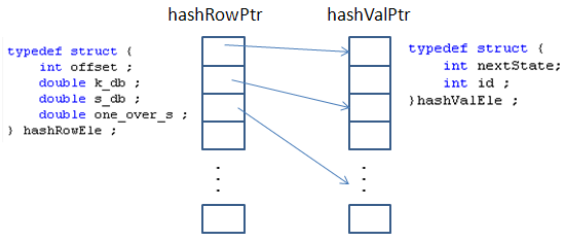

Figure 11: CSR format of hash table. Need extra data to do modulo operation.

Given a pair of (state,ch), we check next state by following computation:

```
1  int id = 256*state + ch
2  int row = (k * id mod p) mod s ;
3  (offset, kj, sj) = hashRowPtr[row];
4  int nextState = -1 ; // trap state
5  if ( 0 <= offset ){
6      int col = (kj * id mod p) mod sj ;
7      if (id == hashValPtr[offset+col].id )
8          nextState = hashValPtr[offset+col].nextState ;
9  }
```

hashValPtr must contain *id* of valid transition because hashing function is one-to-one restricted to valid transitions but many-to-one for all combination of (state,ch). Hence a final verification "id == hashValPtr[offset+col]" is necessary.

2.3 compression rate of hash table

Given a PFAC state machine with R valid transitions, Lemma 1 proves there exists a k such that $\sum_{j=0}^{R-1} B(R,W,k,j)^2 < 3R$, however it is time consuming to find out such k. Fortunately, corollary 3 of [5] states:
For at least one-half of the values k in U,

$$\sum_{j=0}^{R-1} B(R,W,k,j)^2 < 5R$$

So we set above inequality as stopping criterion to find k.

We have shown that total memory of CSR format amounts to *S+1+2R* integers. What is total memory of such hashing table? The answer is not 5R because of extra data in hashRowPtr and hashValPtr. hashRowPtr has S elements and size of each element is 7 integers (3 double and 1 int). hashValPtr has at most $5R$ elements and size of each element is 2 integers. Hence total memory of hash table is *7S+10R* at most.

Remark 3: if we compute $\frac{1+\epsilon}{s_j}$ in GPU kernel, then structure of hashRowPtr can be reduced to 3 integers.

Total memory is down to *3S+10R*.

III. MODULO-FREE PERFECT HASHING

Although perfect hashing gives good compression, its performance is still much worse than explicit 2D array. There are several reasons:
1) two hashing evaluations, one is for row = (k*id mod p) mod s and the other is for column = $(k_j * id \bmod p) \bmod s_j$.
2) modulo operation is expensive on GPU. (k*id mod p) needs at least 7 64-bit floating point operations. Also from Table 1 peak performance of double precision is only half of single precision on Fermi card C2070 which is designed for High Performance Computing (HPC). If one plays game card GTX480, then peak performance of double precision is only 1/8th of single precision.
3) String matching is a memory-bound application, the more data is transmitted, the worse performance is. In perfect hashing algorithm, hashRowPtr and hashValPtr contain extra data used in hashing computation. The extra data would increase workload of bus and slow down performance because limited bandwidth.
4) Original perfect hashing does not consider tree-structure of PFAC state machine.

Based on above observation, we design a variant of perfect hashing algorithm. This new version replaces modulo operation by shift/and and is more space-efficient than original prefect hashing algorithm, so we call it modulo-free perfect hashing. PFAC state machine has following properties:
(P1) except for initial state, every state has few valid transitions,
(P2) column indices are limited by [0, 255], and
(P3) PFAC state machine is a tree.

3.1 idea of modulo free

From property 1, we can do perfect hashing on each row of PFAC table respectively and then only one hashing computation $(k_j * ch \bmod p) \bmod s_j$ is needed for state *j*. Moreover column indices are bounded by 255, we can choose prime $p = 257$, then (x mod p) can be done by following lemma.

Lemma 2(shift and correct): Given a prime number $p = 257$, and $k < p$, $ch < p$, (k*ch mod p) can be done via

```
1 int x = k * ch ;
2 int alpha_hat = x >> 8 ;
3 int beta = x - p*alpha_hat ;
4 if (0 > beta){
5     beta += p ;
6 }
```

proof: we prove above assertion on general $p = 2^m + 1$.

Let $x = k \cdot ch = p \cdot \alpha + \beta,\ 0 < \beta < p$.

step 1: use shift operation to find $\hat{\alpha} = x \gg m$, then decompose $x = 2^m \cdot \hat{\alpha} + \hat{\beta}$, which implies $2^m \cdot \hat{\alpha} \le x < 2^m \cdot (\hat{\alpha} + 1)$. On the other hand, $x = p \cdot \alpha + \beta$ implies $(2^m + 1)\alpha \le x < (2^m + 1)(\alpha + 1)$. Above four inequalities show $\alpha \le \hat{\alpha} \le (\alpha + 1)$, i.e. either $\alpha = \hat{\alpha}$ or $\hat{\alpha} = (\alpha + 1)$.

step 2: compute remainder $\beta = x - p \cdot \hat{\alpha}$.

step 3: test remainder, if $\beta < 0$, $\hat{\alpha} = (\alpha + 1)$ and $\beta \mathrel{+}= p$.

We redefine $B_j = B(S, W, k, j)$ which is number of valid transitions of state *j*. Then total number of valid transitions $R = \sum_{j=0}^{S} B_j$ and number of leaf nodes $L = \left|\{j : B_j = 0\}\right| \le F$. The following lemma shows relationship between B_j and *L*.

Lemma 3: $\left[\sum_{j=0}^{S}(B_j - 1)\right]_{B_j>0} = L - 1$

proof: *R*=*S*-*1* due to tree property.

$$\left[\sum_{j=0}^{S}(B_j - 1)\right]_{B_j>0} = \left[\sum_{j=0}^{S} B_j\right]_{B_j>0} - (S - L)$$

$$= R - (S - L)$$

First modulo operation of ((k*id mod p) mod s) is replaced by shift-and-correct. Next we can replace second modulo $(x \bmod s_j)$ by $(x \gg \log s_j)$ if we choose s_j as power of 2. Let C_k be collection of B_j such that $C_k = \{B_j > 0 : \text{corresponding to } s = 2^k\}$ and $c_k = |C_k|$. The relationship between B_j and s_j is shown in Figure 12. All but $12 \le B_j \le 255$ satisfy $s_j \ge B_j^2$, which is sufficient condition of collision free from Lemma 1. However since column indices are smaller than 256, we can choose $k_j = 1$ and $s_j = 256$ to separate all valid transitions.

Bj	Sj	
1	1	c_0
2	4	c_2
3, 4	16	c_4
5	32	c_5
6, 7, 8	64	c_6
9, 10, 11	128	c_7
12, 13, ..., 255	256	c_8

Figure 12: *Bj* is number of valid transitions of state *j* in PFAC table, and *Sj* is number of locations preserved to separate valid transitions of state *j*.

3.2 quality of compression

The purpose of $s_j = 2^n$ is for speed but we don't want to sacrifice space. It is necessary to have a priori bound of storage requirement, the following theorem confirms quality of compression.

Theorem 1: Given PFAC state machine with *S* states, and *R* valid transitions, let prime $p = 257$, B_j is number of valid transitions of state *j*, and s_j is number of preserved locations to separate valid transitions of state *j*. Configuration of B_j and s_j is shown in Figure 12. Let s_{MF} denote total space required to hash PFAC table (subscript MF denotes modulo-free), then $s_{MF} = \sum_{j=0}^{S} s_j \le \min\left(21{,}4{,}1 + 71\frac{L-1}{S-1}\right) \mathrm{R}$.

proof: claim total space is bounded by $\left(1 + 71\frac{L-1}{S-1}\right) \mathrm{R}$. First consider positive $B_j = 1 + l_j$, from Lemma 3, we have $\sum_\alpha l_\alpha = L - 1$. On the other hand we can estimate c_k by following inequality

$$L-1 = \sum_{\alpha} l_\alpha = \sum_{B_\alpha \in C_k} l_\alpha + \sum_{others} l_\alpha \ge \min_{B_\alpha \in C_k}\{l_\alpha\} \cdot c_k \quad \text{implies}$$

$$c_k \le \frac{L-1}{\min\limits_{B_\alpha \in C_k}\{l_\alpha\}}, \text{ i.e. } \ c_2 \le \frac{L-1}{2-1},\ c_4 \le \frac{L-1}{3-1},\ c_4 \le \frac{L-1}{5-1},$$

$$c_6 \le \frac{L-1}{6-1},\ c_7 \le \frac{L-1}{9-1} \text{ and } c_8 \le \frac{L-1}{12-1}.$$

$$\text{Total space } = \sum_{k\ge 2} 2^k c_k + 1 \cdot \left|\{B_j = 1\}\right| + 0 \cdot \left|\{B_j = 0\}\right|,$$

the theorem holds because $\sum_{k\ge 2} 2^k c_k \le 72 \cdot (L-1)$ and

$$\left|\{B_j = 1\}\right| = S - L - \sum_{k\ge 2} c_k < S - L \cdot$$

Next claim total space is bounded by $21.4 \cdot R$. From

Figure 12, average space per valid transition is bounded by $\max_k \frac{s_k}{\min_{B_j \in C_k} B_j} = \frac{256}{12} < 21.4$.

Remark 4: **Theorem 1** uses tree property of PFAC state machine to derive space bound. It is easy to estimate size of hash table without construction of state machine. Let F denote number of patterns (final states of state machine) and C_{total} denote number of characters of patterns. Then $L \leq F$ and $S \leq C_{total}$ because common prefix of two patterns shares the same transition path. Average pattern length is around $\frac{C_{total}}{F}$. Hence size of MFPH is bounded by $2\left(2 + \frac{71}{average\ pattern\ length}\right) C_{total}$ integers

3.3 data structure of modulo-free

We adopt same hashRowPtr and hashValPtr but different contents. Except for offset field, hashRowPtr should contain k_j and s_j which are used in hashing function $(k_j \cdot ch \bmod p) \bmod s_j$. However we only use 1 integers to represent k_j and s_j because $k_j < p(=257)$ and $s_j \leq 256$. Moreover s_j is power of 2, so we only store $s_j - 1$ such that $(x \bmod s_j)$ is equal to $\left(x \,\&\, (s_j - 1)\right)$. Hence upper half (16 bits) of an integer is for k_j and lower half is for $s_j - 1$ as you see in Figure 13. hashValPtr is the same as that in original perfect hashing algorithm except that id is replaced by ch.

hashRowPtr has S elements and size of each element is 2 integers. hashValPtr has at most S_{MF} elements and size of each element is 2 integers. So total memory of modulo-free perfect hashing amounts to $2S + 2S_{MF}$ integers.

IV. Experimental Results

We evaluate the time complexity (performance) and space complexity (storage) of the different versions of perfect hashing algorithm on game card GTX480 and Tesla C2070. The SPEC of GPUs are shown in Table 1, the bandwidth of GTX480 is 1.5x of bandwidth of C2070 whereas peak performance of double precision of C2070 is 3x than that of GTX480. We don't compare GPU with CPU because this is done in [1].

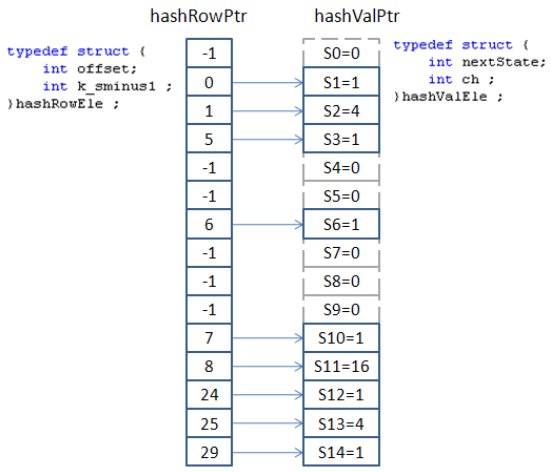

Figure 13: data structure of modulo-free perfect hashing on example of Figure 3.

In this work, we propose three implementation of perfect hashing:

(1) perfect hashing without division (PH without div)

It requires two hashing computations, $(k \cdot id \bmod p) \bmod s$ and $(k_j \cdot ch \bmod p) \bmod s_j$ which are done by division of 64-bit floating point where *1/p* and *1/s* are pre-computed on CPU.

(2) perfect hashing with division (PH with div)

Same as (1) but *1/p* and *1/s* are computed in the kernel of GPU.

(3) modulo-free perfect hashing (MFPH)

It only needs one hashing computation $(k_j \cdot ch \bmod p) \bmod s_j$ which uses shift/and to do modulo operations, $(\cdot \bmod p)$ and $(\cdot \bmod s_j)$.

The data structures of three versions are shown in Table 2. Cost per table lookup is volume of data needed in hashing computation. It is typical summation of Sizeof(*hashRowPtr) and Sizeof(*hashValPtr).

Table 2: data structure of three perfect hashing algorithms

	PH without div	PH with div	MFPH
structure of hashRowPtr	int *offset* double *k_db* double *s_db* double *one_over_s*	int *offset* int *k* int *s*	int *offset* int *k_sminus1*
sizeof(*hashRowPtr) bytes	28	12	8
structure of hashValPtr	int *nextState* int *id*	int *nextState* int *id*	int *nextState* int *ch*
sizeof(*hashValPtr) bytes	8	8	8
Cost per table lookup (bytes)	36	20	16

4.1 space complexity

The test patterns are extracted from Snort V2.8 [7]. We pick 6 patterns, named patterns_state[x].txt where x denotes number of states. We list three parameters of PFAC state machine corresponding to each pattern file, including number of states *S*, number of final states *F* (= number of patterns in a pattern file), and number of leaf nodes *L*. Also we define compression rate of a perfect hashing algorithm as its total space of hashRowPtr and

hashValPtr against space of explicit 2-D array, which is 256*S integers.

$$\text{compression rate} = \frac{\text{total space of the PH algorithm}}{\text{space of 2-D array}}$$

Table 3 lists compression rate of three perfect hashing algorithms, and MFPH is the best because it encodes k_j and s_j into one 32-bit integer and no division is needed to do modulo operation. The second is "PH with div" because it needs not to store pre-computed *1/p* and *1/s*. Besides space requirement of MFPH is only 1/50*th* of size of 2-D array. So MFPH is much attractive in string matching of HUGE patterns.

4.2 time complexity

We show performance of perfect hashing algorithms on two benchmarks:

Benchmark I: input stream is 100M 'a' or 'b' characters and patterns are all 'a' characters. This is a trivial test, and show difference between AC and PFAC. If input stream has N characters, then time complexity of AC is O(N) whereas time complexity of PFAC is $O(\alpha N)$ where α is average length of patterns. For example, if pattern is 100a, then complexity of PFAC is O(100*N*), 100 times bigger than AC. However PFAC has no load unbalance on this trivial test because each thread processes the same length of substring. So this test is sensitive to variation of instruction counts and bandwidth. In other words, if we increase instruction count or data transfer in the algorithm, then performance on this test drops down significantly. It is good to judge complexity of different perfect hashing algorithms.

Benchmark II: we use file patterns_state27754.txt which contains 1,998 exact string patterns of total 41,997 characters. The length of the string pattern varies between one to 243 characters long. The total number of states is 27,754. The input stream comes from DEFCON packets which contain large amounts of real attack patterns are widely used to test commercial NIDS system. We classify input streams into two categories, one is regular with few viruses and the other is worst with many viruses. The size of test packets varies from 2 MB to 64 MB.

We don't use absolute elapsed time as performance metric, but consider raw data throughput instead.

$$\text{Raw data throughput} = \frac{8\text{M}}{\text{t}} \text{ (Gbps)}$$

where *M* denotes the length of an input stream of unit bytes and *t* denotes the GPU elapsed time of performing pattern matching. Gbps stands for Giga bit per second.

Table 4 shows performance on benchmark I. There are two numbers of each field, first one comes from GTX480 and second one is from C2070. Game card GTX480 is better on bandwidth and 32-bit arithmetic operations whereas C2070 is professional on 64-bit arithmetic operations. PFAC is memory-bound discussed in [1], and so is MFPH. However it is not easy to conclude that "PH without div" or "PH with div" are memory-bound because they use many 64-bit floating point arithmetic operations. Besides we take MFPH as baseline and compare it with "PH without div" and "PH with div". For example, if pattern is 1a and input is 100M{a}, then MFPH can reach 103.11Gbps on GTX480 and 78.74 Gbps on C2070. This is reasonable because MFPH is memory-bound and bandwidth of GTX480 is 1.4x bigger than C2070. However "PH without div" only reaches 19.67 Gbps on GTX480, and "PH with div" performs worst, only reaches 12.30Gbps on GTX480. So on GTX480, MFPH is 3.06x faster than "PH without div" and 8.38x faster than "PH with div". Besides "PH with div" is slower than "PH without div", this should be penalty of division which is expensive on GPU.

Second if we focus on 100M {a} of input stream, then "PH without div" and "PH with div" perform better on C2070. This evidence shows "PH without div" and "PH with div" are more like computational-bound. But you should bear in mind that this is because benchmark I does not incur load unbalance.

Third if we focus on 100M{b}, then nothing matches, every thread reads one character 'b', then goes to trap state and terminates processing. So time complexity is O(*N*), and independent of length of patterns if patterns are all 'a'. At his time, PH without div" and "PH with div" are neither computational-bound or memory-bound.

Table 5 shows performance on benchmark II. MFPH is still the best and is memory-bound on both regular and worst inputs. "PH without div" inclines to memory-bound on both input categories but "PH with div" is slightly computational-bound on worst inputs. This tells us that division is not good on GPU, we should try to avoid it.

There is a wired thing on regular inputs: "PH without div" is slower than "PH with div". We can only give a heuristic argument but no direct evidence. "PH without div" fetches *1/p* and *1/s* from DRAM but "PH with div" compute them in flight. Average processing length of regular inputs per thread is about 1. Most threads read one character and terminate. So bandwidth pressure is large, and "PH without div" loads more data than "PH with div".

Benchmark II is close to real life, and MFPH is at least 4x faster than both "PH with div" and "PH without div". So idea of modulo-free is excellent in GPU computing.

V. CONCLUSIONS

In this paper, we proposed a modulo-free perfect hashing algorithm (MFPH) on exact string matching and prove its time efficiency and good compression rate against trivial implementation of original perfect hashing algorithm. MFPH has been added into PFAC library and called space-driven method whereas explicit 2-D transition table is called time-driven method.

Space complexity is critical in HUGE database and MFPH can perform well on GPU because of small merging penalty. Also it can play an important role on improving performance of PFAC. In the future, we will like to solve load unbalance of PFAC and propose a new state machine which is bigger than current one and input character is not restricted to [0,255]. 2-D array is impractical anymore and we need a space-efficient method, which is similar to MFPH.

Table 3: quality of compression of different perfect hashing algorithms

Pattern file	Ns = # of final states	F = # of final states	L = # of leaf nodes	PH without div Compression rate	PH with div Compression rate	MFPH Compression rate
patterns_state5176.txt	5,176	340	303	0.062	0.043	0.019
patterns_state10486.txt	10,486	792	727	0.061	0.040	0.020
patterns_state14379.txt	14,379	994	917	0.046	0.027	0.020
patterns_state20281.txt	20,281	1,503	1,342	0.063	0.043	0.020
patterns_state27754.txt	27,754	1,998	1,795	0.062	0.042	0.019
patterns_state30650.txt	30,650	2,002	1,799	0.046	0.026	0.019

Table 4: performance of benchmark I. First number comes from GTX480 and second number comes from Tesla C2070.

pattern	Input	MFPH Throughput (Gbps)		PH without div				PH with div			
				Throughput (Gbps)		MFPH/(PH without div)		Throughput (Gbps)		MFPH/(PH with div)	
1a	100M {a}	103.11	78.74	19.67	25.66	5.24	3.06	12.30	16.23	8.38	4.84
10a	100M {a}	15.01	11.47	3.58	5.06	4.18	2.28	2.23	3.10	6.71	3.69
100a	100M {a}	1.57	1.20	0.39	0.55	4.02	2.16	0.24	0.34	6.45	3.53
1a	100M {b}	196.08	149.05	38.31	47.29	5.11	3.15	24.33	30.67	8.05	4.85
10a	100M {b}	196.09	149.07	66.19	65.58	2.96	2.27	65.38	63.82	2.99	2.33
100a	100M {b}	195.98	149.06	65.62	65.56	2.98	2.27	65.37	63.65	3.00	2.34

Table 5: performance of benchmark II. First number comes from GTX480 and second number comes from Tesla C2070.

Input	MFPH Throughput (Gbps)		PH without div				PH with div			
			Throughput (Gbps)		MFPH/(PH without div)		Throughput (Gbps)		MFPH/(PH with div)	
regular 2MB	60.67	45.97	8.54	6.62	7.09	6.94	9.35	8.88	6.47	5.18
regular 16MB	69.76	52.60	7.39	5.68	9.43	9.25	9.33	8.25	7.47	6.36
regular 64MB	72.41	54.51	7.74	5.94	9.34	9.16	9.66	8.58	7.49	6.34
worst 2MB	8.45	6.39	1.97	1.66	4.29	3.84	1.44	1.70	5.84	3.79
worst 16MB	8.89	6.78	2.09	1.82	4.24	3.72	1.47	1.74	3.98	3.87
worst 64MB	8.97	6.84	2.20	1.92	4.06	3.56	1.48	1.75	6.05	3.90

REFERENCES

[1] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPU," preprint. The paper is available in *doc* subdirectory of PFAC library, http://code.google.com/p/pfac/.

[2] A. Tumeo and O. Villa, "Accelerating DNA analysis applications on GPU clusters", IEEE Symposium on Application Specific Processors (SASP), Anaheim, CA, June 13-14, 2010, pp. 71-76

[3] University of california, santa cruz genome bioinformatics, available at: http://genome.ucsc.edu/

[4] NVIDIA Corporation, CUDA CUSPARSE Library, Available: http://developer.nvidia.com

[5] Michael L. Fredman and Janos Komlos, and Endre Szemeredi, Storing a Sparse Table with O(1) Worst Case Access Time, Journal of the Association for Computing Machinery, Vol 31, No 3, July 1984, pp. 538-544.

[6] http://www.bigprimes.net/archive/prime

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

[8] NVIDIA Corporation. NVIDIA CUDA programming Guide, 2007 Available: http://developer.nvidia.com