PFAC Library: GPU-Based String Matching Algorithm

Cheng-Hung Lin	National Taiwan Normal University, Taipei, Taiwan
Lung-Sheng Chien	National Tsing-Hua University, Hsinchu, Taiwan
Chen-Hsiung Liu	National Tsing-Hua University, Hsinchu, Taiwan
Shih-Chieh Chang	National Tsing-Hua University, Hsinchu, Taiwan
Wing-Kai Hon	National Tsing-Hua University, Hsinchu, Taiwan

GPU Technology Conference 2012

Outline

- Introduction
- Review of Aho-Corasick Algorithm
- Parallel Failureless Aho-Corasick Algorithm
- Experimental Results and Conclusion

What is PFAC?

- **PFAC** is an open source library for multiple string matching performed on **Nvidia GPUs**.
  - PFAC runs on Nvidia GPUs that support **CUDA**, including NVIDIA 1.1, 1.2, 1.3, 2.0 and 2.1 architectures.
  - Supporting OS includes ubuntu, Fedora and MAC OS.

- Released at Google code project
  - http://code.google.com/p/pfac/
  - Provides C-style API
  - Users don't need to have background on GPU computing or parallel computing.

What is PFAC?

PFAC is an open library for exact string matching performed on **GPUs**. PFAC runs on processors that support **CUDA**, including NVIDIA 1.1, 1.2, 1.3, 2.0 and 2.1 architectures. Supporting OS includes ubuntu, Fedora and MAC OS.

PFAC library provides C-style API and users need not have background on GPU computing or parallel computing. PFAC has APIs hiding CUDA stuff.

News

- PFAC r1.0 updated 2011/02/23
- PFAC r1.o released 2011/02/21
- PFAC r1.1 released 2011/04/27
- PFAC r1.2 released 2011/04/29

Simple Example

Example 1: Using PFAC_matchFromHost function

The file "example_pattern" in the directory "../test/pattern/" contains 4 patterns.

```
AB
ABG
BEDE
ED
```

The file "example_input" in the directory "../test/data/"contains a string.

```
ABEDEDABG
```

Five Steps to Use PFAC for String Matching

The following example shows the basic steps to use PFAC library for string matching.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <PFAC.h>

int main(int argc, char **argv)
{
    char dumpTableFile[] = "table.txt" ;
    char inputFile[] = "../test/data/example_input" ;
    char patternFile[] = "../test/pattern/example_pattern" ;
    PFAC_handle_t handle ;
    PFAC_status_t PFAC_status ;
    int input_size ;
    char *h_inputString = NULL ;
    int  *h_matched_result = NULL ;

    // step 1: create PFAC handle
    PFAC_status = PFAC_create( &handle ) ;
    assert( PFAC_STATUS_SUCCESS == PFAC_status );

    // step 2: read patterns and dump transition table
    PFAC_status = PFAC_readPatternFromFile( handle, patternFile) ;
    if ( PFAC_STATUS_SUCCESS != PFAC_status ){
        printf("Error: fails to read pattern from file, %s\n", PFAC_getErrorString(PFAC_status) );
        exit(1) ;
    }

    // dump transition table
    FILE *table_fp = fopen( dumpTableFile, "w") ;
    assert( NULL != table_fp ) ;
    PFAC_status = PFAC_dumpTransitionTable( handle, table_fp );
    fclose( table_fp ) ;
    if ( PFAC_STATUS_SUCCESS != PFAC_status ){
        printf("Error: fails to dump transition table, %s\n", PFAC_getErrorString(PFAC_status) );
        exit(1) ;
    }
```

```
    //step 3: prepare input stream
    FILE* fpin = fopen( inputFile, "rb");
    assert ( NULL != fpin ) ;

    // obtain file size
    fseek (fpin , 0 , SEEK_END);
    input_size = ftell (fpin);
    rewind (fpin);

    // allocate memory to contain the whole file
    h_inputString = (char *) malloc (sizeof(char)*input_size);
    assert( NULL != h_inputString );

    h_matched_result = (int *) malloc (sizeof(int)*input_size);
    assert( NULL != h_matched_result );
    memset( h_matched_result, 0, sizeof(int)*input_size ) ;

    // copy the file into the buffer
    input_size = fread (h_inputString, 1, input_size, fpin);
    fclose(fpin);

    // step 4: run PFAC on GPU
    PFAC_status = PFAC_matchFromHost( handle, h_inputString, input_size, h_matched_result ) ;
    if ( PFAC_STATUS_SUCCESS != PFAC_status ){
        printf("Error: fails to PFAC_matchFromHost, %s\n", PFAC_getErrorString(PFAC_status) );
        exit(1) ;
    }

    // step 5: output matched result
    for (int i = 0; i < input_size; i++) {
        if (h_matched_result[i] != 0) {
            printf("At position %4d, match pattern %d\n", i, h_matched_result[i]);
        }
    }
```

The screen shows the following matched results.

```
At position    0, match pattern 1
At position    1, match pattern 3
At position    2, match pattern 4
At position    4, match pattern 4
At position    6, match pattern 2
```

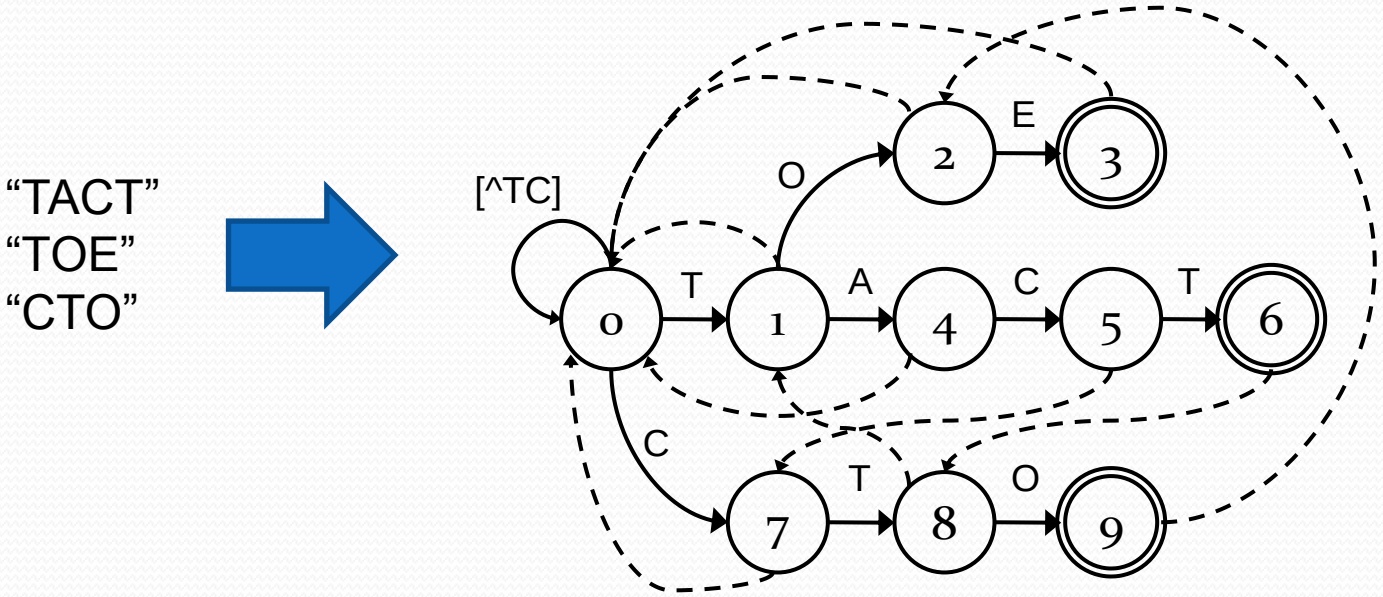
Outline

- Introduction
- Review of Aho-Corasick Algorithm
- Parallel Failureless Aho-Corasick Algorithm
- Experimental Results and Conclusion

Aho-Corasick Algorithm

- Aho-Corasick algorithm has been widely used for string matching due to its advantage of matching multiple string patterns in a single pass

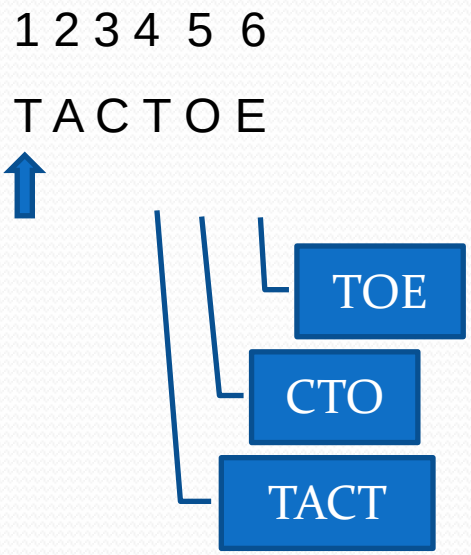
- Aho-Corasick algorithm compiles multiple string patterns into a state machine

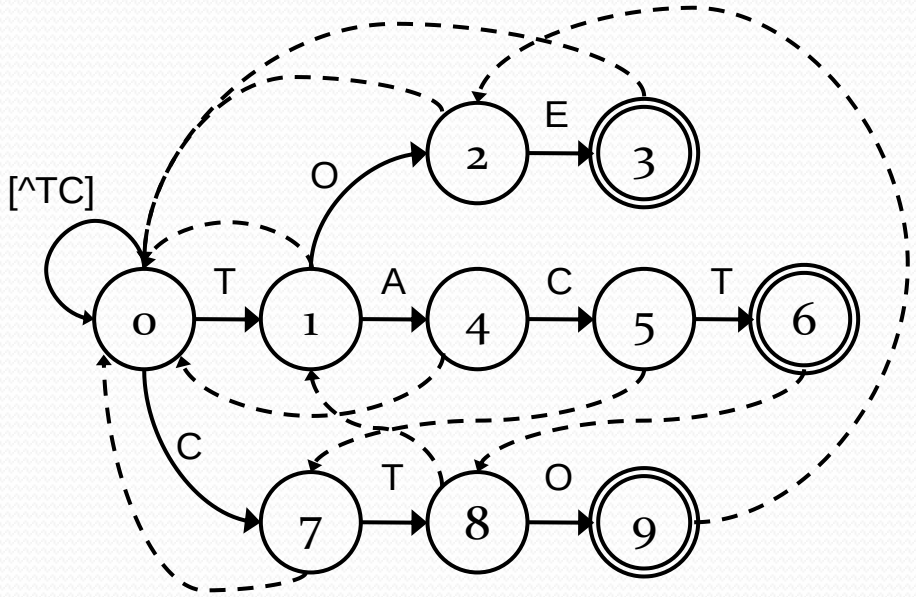


Aho-Corasick Algorithm (cont.)

- String matching is performed by traversing the Aho-Corasick (AC) state machine

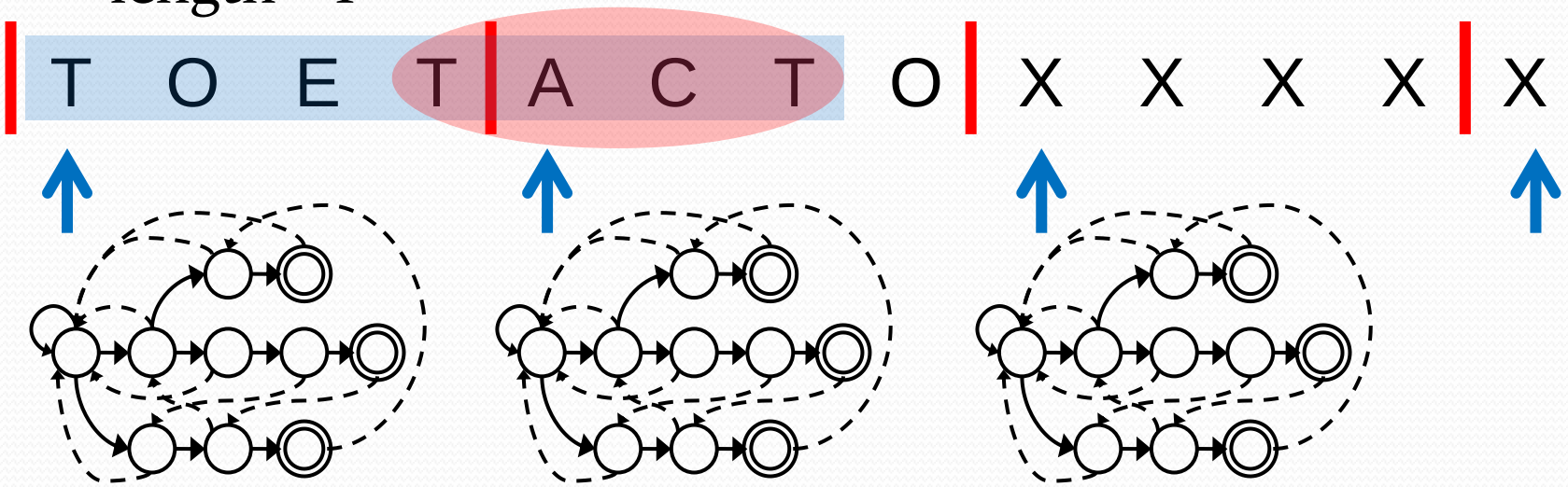
- Failure transitions are used to backtrack the state machine to recognize patterns in different start locations.





Data Parallel AC Approach

- Partition an input stream into multiple segments and assign each segment a thread to traverse AC state machine
- Boundary detection problem
  - Pattern occurs in the boundary of adjacent segments.
  - Duration time of threads = segment size + longest pattern length – 1

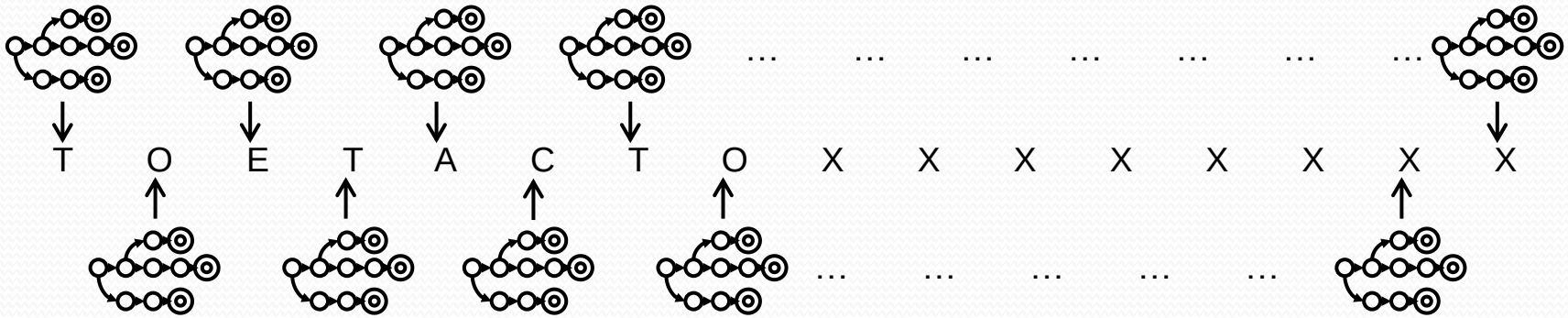


Outline

- Introduction
- Review of Aho-Corasick Algorithm
- Parallel Failureless Aho-Corasick Algorithm
- Experimental Results and Conclusion

Parallel Failureless Aho-Corasick Algorithm

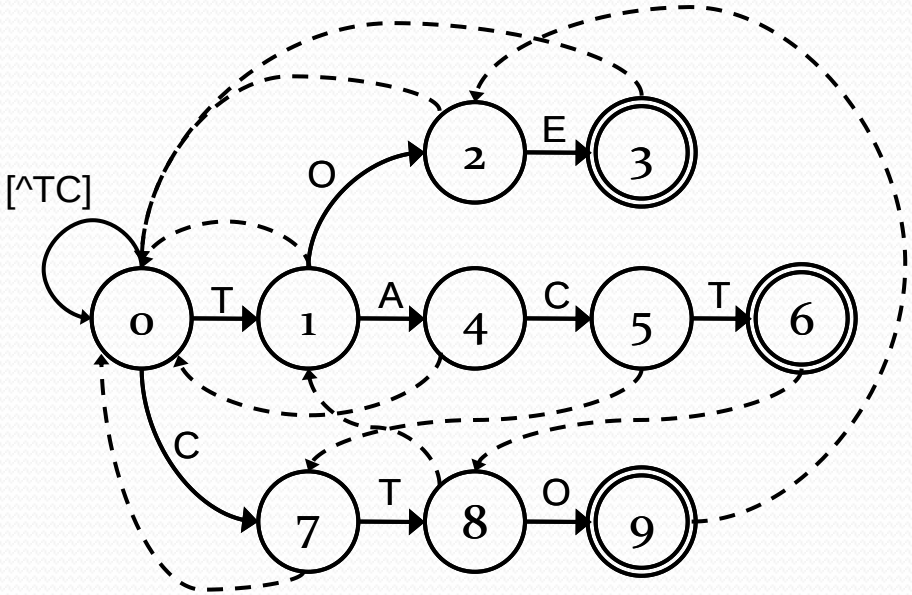
- Parallel Failureless Aho-Corasick (PFAC) algorithm on graphic processing units
  - Allocate each byte of input an individual thread to traverse a state machine



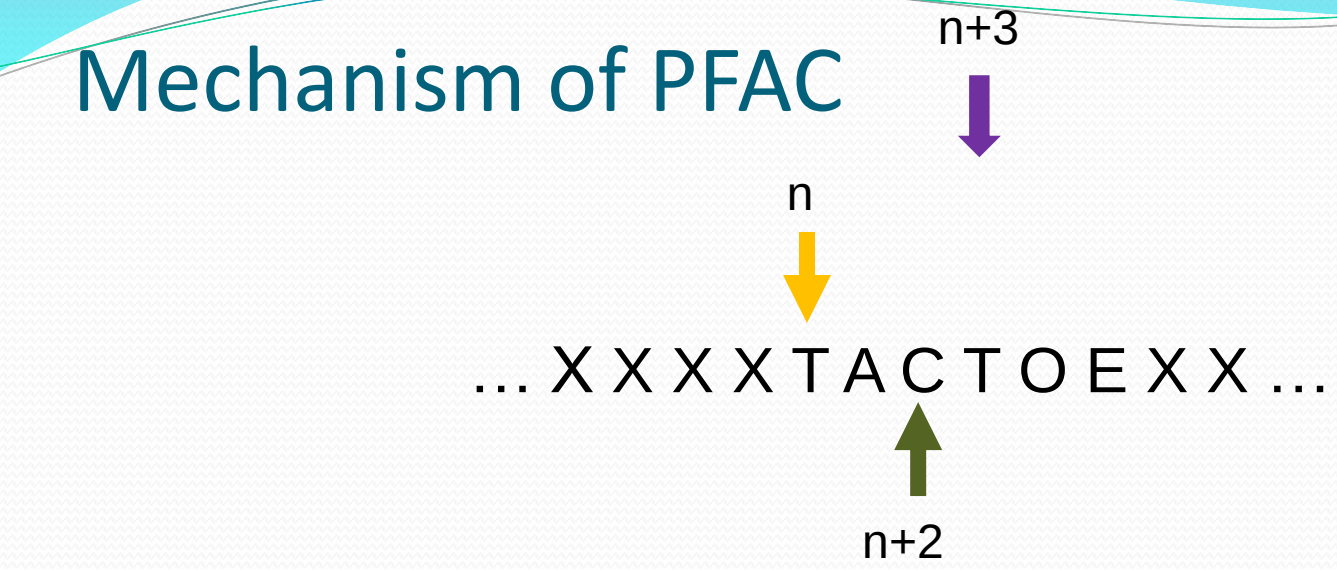
- Reference:
  - C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu, "Accelerating String Matching Using Multi-Threaded Algorithm on GPU," in *GLOBECOM 2010, 2010 IEEE Global Telecommunications Conference*, 2010, pp. 1-5.

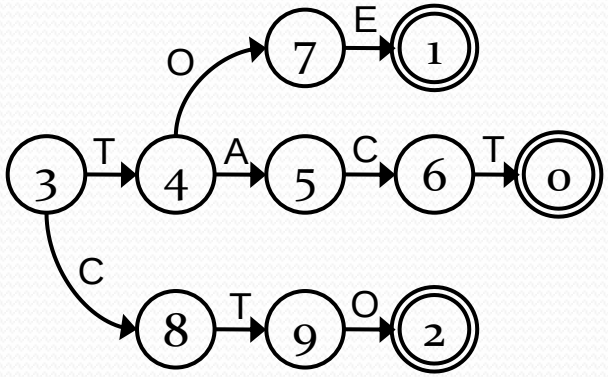
Failureless-AC State Machine

- Remove all failure transitions as well as the self-loop transitions backing to the initial state
  - Minimum number of valid transitions
  - Thread is terminated when no valid transitions

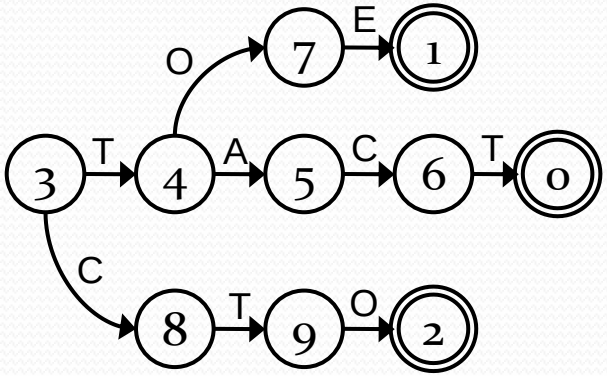


Mechanism of PFAC

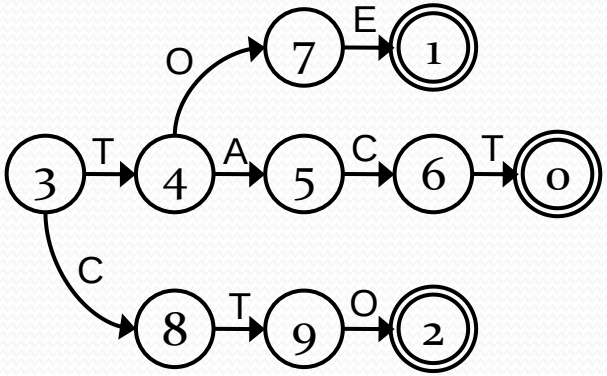




Thread# n



Thread# n+2



Thread# n+3

Pros and Cons

	Data Parallel AC	PFAC
Time complexity	O(N + ms)	O(mN)
Space complexity	O(256 * S)	O(256 * S)
Load imbalance	low	high
Performance variation	low	high

- *N* is the input length
- *m* is the longest pattern length.
- *s* is the number of segments
- *S* is number of states.

Optimization techniques

- Reduce memory transactions of global memory
  - load input from global memory to shared memory by **coalescing read** of integer
  - fetch input from shared memory.
- Reduce latency of transition table lookup
  - bind the state transition table to **texture memory**
  - load the first row of the *PFAC_table* into shared memory
- Eliminate output table by reordering state number

Outline

- Introduction
- Review of Aho-Corasick Algorithm
- Parallel Failureless Aho-Corasick Algorithm
- Experimental Results and Conclusion

Experimental Environment

- Intel Core™ i7-950
  - Quad cores
  - 12GB DDR3 memory
- Nvidia® GeForce® GTX580
  - 512 cores
  - 1536MB GDDR5 memory
- Patterns: String pattern extracted from Snort V2.8, containing 27754 states, 1998 final states (patterns)
- Input: extracted from DEFCON

Implementations

- AC_{CPU} : implementation of the AC algorithm on the Core™ i7 using a single thread and optimized by GCC 4.4.3 using the compiler flags "-O2 –msse4".

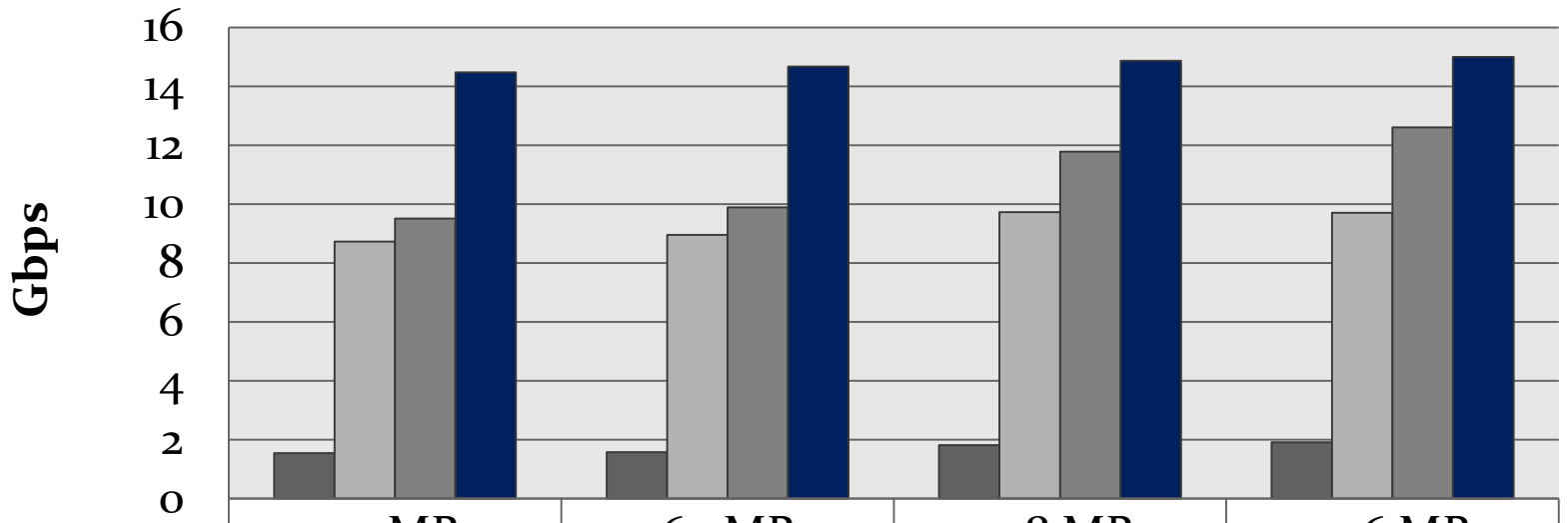
- $DPAC_{OMP}$: implementation of the DPAC algorithm on Intel Core™ i7 CPU with OpenMP and optimized by GCC 4.4.3 using the compiler flags "-O2 –msse4".

- $PFAC_{OMP}$: implementation of the PFAC algorithm on Intel Core™ i7 CPU with the OpenMP and optimized by GCC 4.4.3 using the compiler flags "-O2 –msse4".

- $PFAC_{GPU}$: implementation of the PFAC algorithm on NVIDIA GPUs.

Performance Evaluation

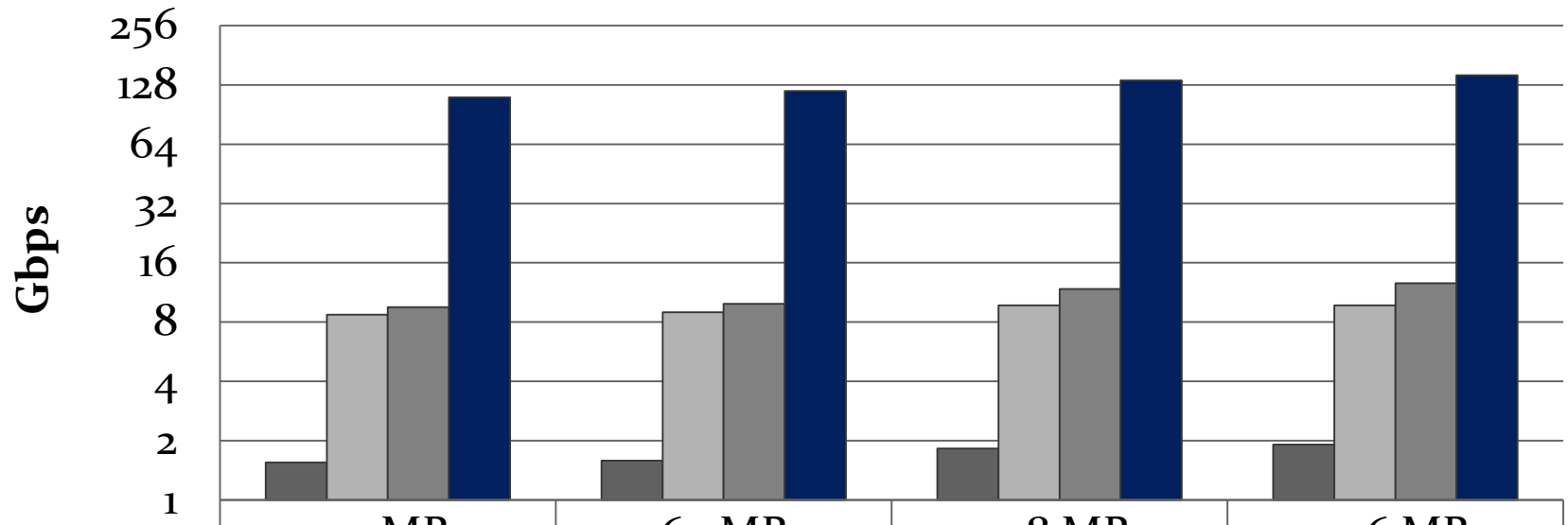
$$System\ throughput = \frac{input\_size}{t_{H2D} + t_{GPU} + t_{D2H}}$$



	32 MB	64 MB	128 MB	256 MB
AC_CPU	1.55	1.58	1.82	1.91
DPAC_OMP	8.73	8.96	9.73	9.71
PFAC_OMP	9.51	9.89	11.79	12.61
PFAC_GPU	14.48	14.68	14.87	15

Performance Evaluation

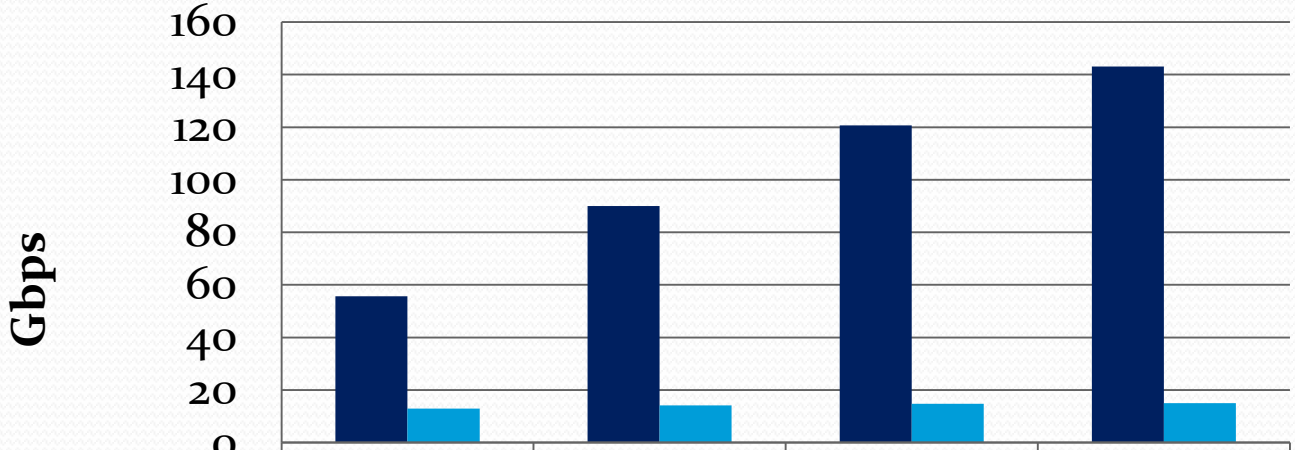
$$Raw\ data\ throughput = \frac{input\_size}{t_{GPU}}$$



	32 MB	64 MB	128 MB	256 MB
AC_CPU	1.55	1.58	1.82	1.91
DPAC_OMP	8.73	8.96	9.73	9.71
PFAC_OMP	9.51	9.89	11.79	12.61
PFAC_GPU	110.64	119.37	135.02	143.16

Performance Evaluation

NVIDIA GPU	# of cores	Memory bandwidth(GB/s)	Compute Capacity
GTX 580	512	192.4	2.0
GTX 480	480	177.4	2.0
Tesla C2050	448	148.4	2.0
GTX 295	2x240	111.9	1.3



	GTX 295	Tesla C2050	GTX 480	GTX 580
raw data throughput	55.7	90.07	120.71	143.16
system throughput	12.88	14.12	14.7	15

Conclusions

- We have proposed an open source library to accelerate multiple sting matching on NVIDIA GPUs.

- We have applied perfect hashing to reduce memory requirements of PFAC.
  - PFAC Library V1.1 and V1.2

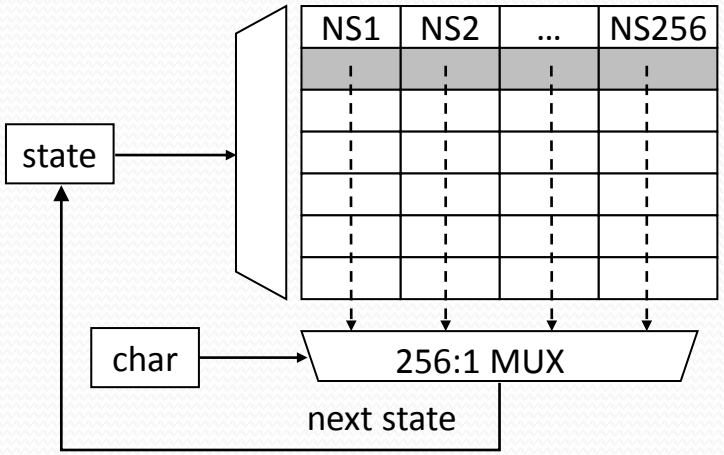
Thanks for your attention!

Q&A

Backup Slides

Memory Issue of PFAC

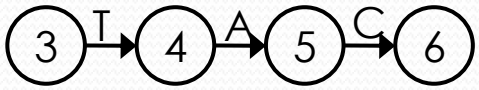
- The two-dimensional memory is sparse.
  - Each state (row) needs 1K (256 x 4)bytes
  - A state machine with 1M states needs 1G bytes
  - 99% of memory is wasted

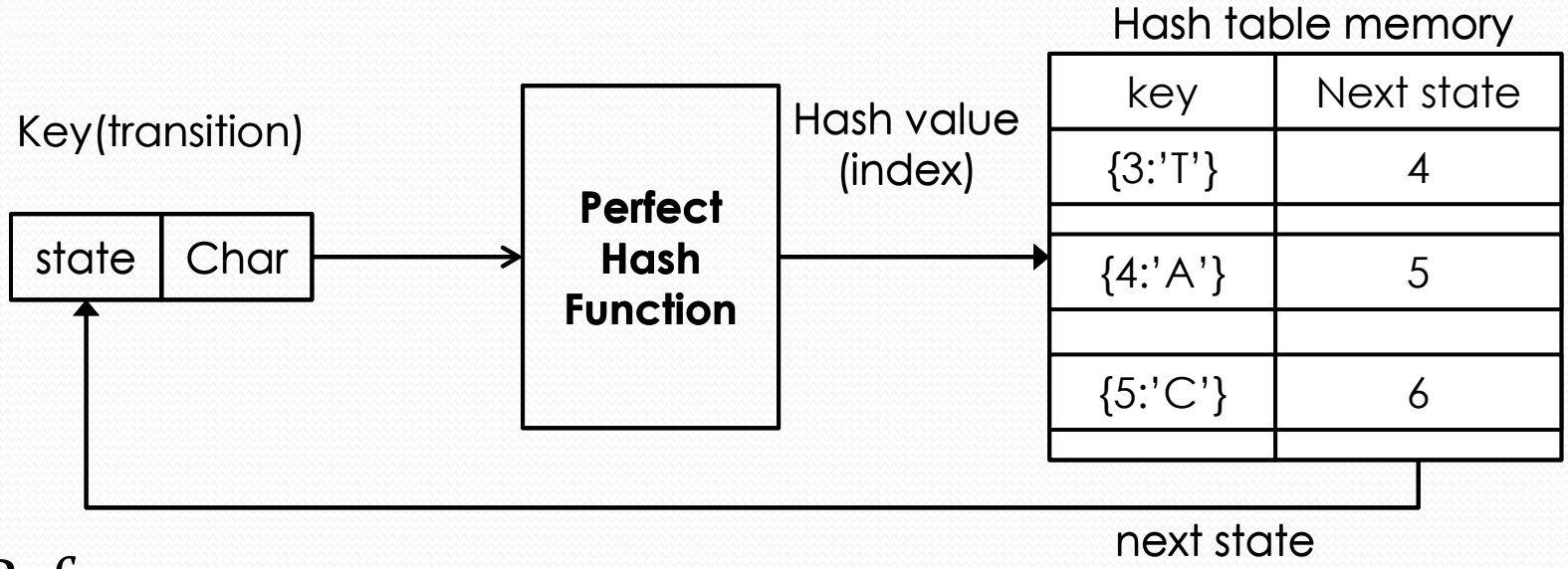


- Design a compact storage mechanism for storing PFAC state transition table is essential for GPU implementation.

Perfect Hashing Memory Architecture

- Use a perfect hash function to store only valid transitions of a PFAC state machine in a hash table





- Reference
  - 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," in IEEE INFOCOM, 2012

Hardware-friendly Perfect Hash Function

- Slide-Left-then-Right First-Fit (SLRFF) algorithm
- Steps to create PHF table
  1. Start with a two-dimensional table of width *w* and place each key *k* at location (*row*, *col*), where *row* = k / w, *col* = *k* mod *w*.
  2. Rows are prioritized by the number of keys in it and move rows in order of priority as following steps.
     a) First, slide the row left to let the first key in the row be aligned at the first column.
     b) Then, slide the row right until each column has only one key and record the offset in an array.
  3. Collapse the two-dimensional key table to a linear array.

Step 1 of Creating PHF

- Key set, S = {2, 4, 10, 11, 13, 14, 17, 20, 21, 25, 27}
- Start with a two-dimensional table of width *w* and place each key *k* at location (*row*, *column*), where *row* = k / w , *column* = *k* mod *w*.

Key table (*w* = 8)

		2		4			
		10	11		13	14	
	17			20	21		
	25		27				

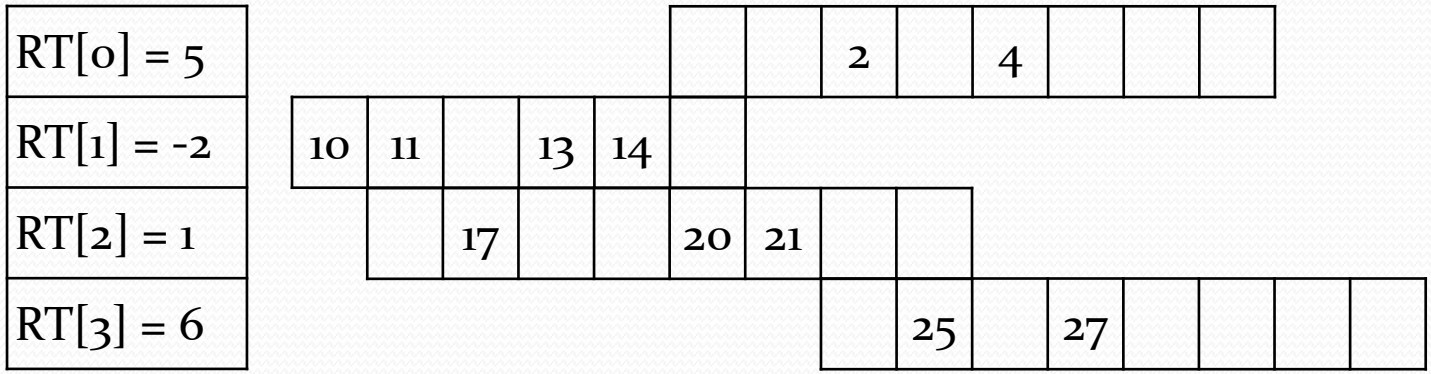
Step 2 of Creating PHF

- Rows are prioritized by the number of keys in it
- According to the order of priority
  a) Slide each row left to let the first key be aligned at the first column.
  b) Slide each row right until each column has only one key and record the offset in the array RT.

3 RT[0] = 5			2		4			
1 RT[1] = 2			10	11		13	14	
2 RT[2] = 0		17			20	21		
4 RT[3] = 6		25		27				

Step 3 of Creating PHF

- Collapse the two-dimensional table to a linear table HK.



HK :

10	11	17	13	14	20	21	2	25	4	27

Computation of Hash Value

- *row* = k / w ;
- *col* = *k* mod *w* ;
- *index* = RT[*row*] + *col* ;
- If HK[*index*] == *k*
  k is a valid key ;
  else
  k is an invalid key ;

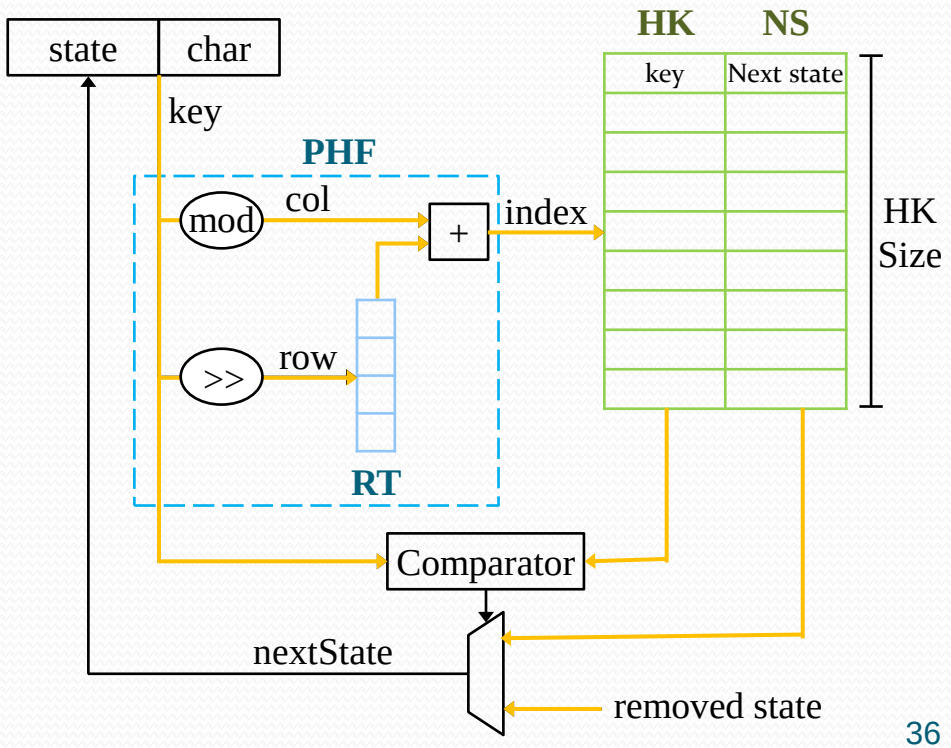
For example:
Given *k* = 14
row = 14 / 8 = 1
col = 14 mod 8 = 6
index = RT[1] + 6 = -2 + 6 = 4
HK[4] =14
14 is a valid key

RT[0] = 5
RT[1] = -2
RT[2] = 1
RT[3] = 6

index :	0	1	2	3	4	5	6	7	8	9	10
HK :	10	11	17	13	14	20	21	2	25	4	27

Perfect Hashing Memory Architecture

- Algorithm
  - *row* = k / w ;
  - *col* = *k* mod *w* ;
  - *index* = RT[*row*] + *col* ;
  - If HK[*index*] == *k*
    k is a valid key ;
    else
    k is an invalid key ;
  - If *k* is a valid key
    nextState = NS[*index*] ;
    else
    nextState = trap state;



Two-level Perfect Hashing

Level-1 hashing

ch

state	e	h	i	r	s
1		14			
2	3		12		
3				6	
6					5
10					8
11		2	13		1
12					7
13			10		9
14	4				

id= (state*256 + ch)

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

$(k \cdot id \bmod p) \bmod s$

p=179424691
k=3
s=13

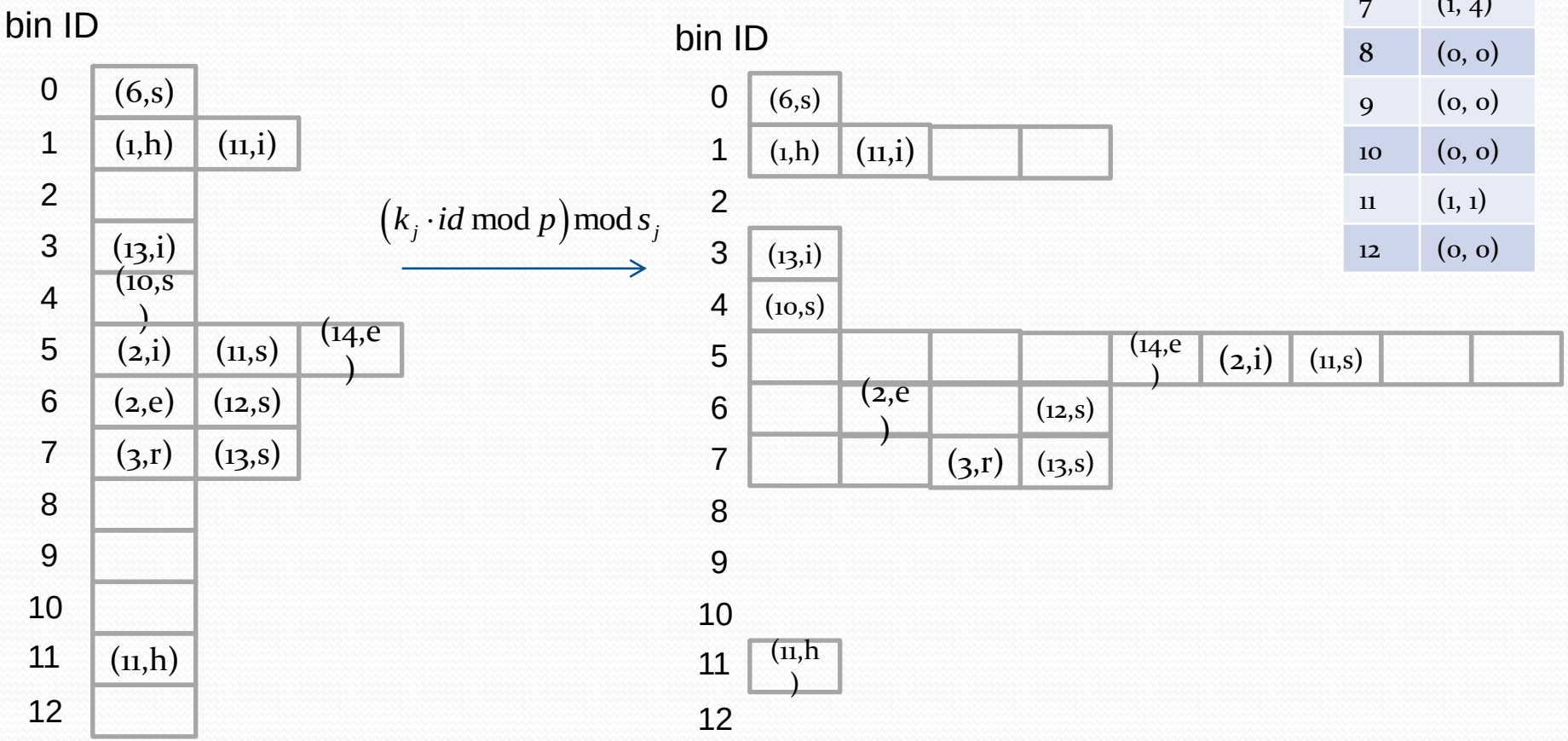
bin ID

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			

Level-2 hashing

- Apply the same hashing function by different parameters

bin	(k, s)
0	(1, 1)
1	(1, 4)
2	(0 , 0)
3	(1, 1)
4	(1, 1)
5	(1, 9)
6	(1, 4)
7	(1, 4)
8	(0, 0)
9	(0, 0)
10	(0, 0)
11	(1, 1)
12	(0, 0)



Drawback of 2-Level Perfect Hashing

- Two hashing evaluations, one is for row and the other is for column.
  - ✓ Modulo operation is expensive on GPU.
  - ✓ (k * id mod p) needs seven 64-bit floating point operations at least.
- String matching is a memory-bound application. Parameters *k*, *p* and *s* would increase bus overhead and slow down the performance.
- Original perfect hashing does not consider tree structure of PFAC state machine.
  - ✓ Except for initial state, every state has few valid transitions
  - ✓ Column indices are limited by [0, 255], and
  - ✓ PFAC state machine is a tree

Modulo-free Perfect Hashing

- Except for initial state, every state has few valid transitions perform perfect hashing on each row of PFAC table, only one hashing computation, (k * ch mod p) mod s.
- Column indices are limited by [0, 255]
  choose prime number p = 257, (x mod p) can be done by (x mod 256)
- PFAC state machine is a tree

Theorem: Given a PFAC state machine with N states and R transitions. Let prime p=257, s_j is number of preserved locations to separate valid transitions of state j. S_{MF} denotes total space required to hash PFAC table, then

$$S_{MF} = \sum_{j=0}^{N} s_j \le \min\left(21.4, 1+71\frac{L-1}{N-1}\right)R$$

Experimental Environment

- Intel Core™ i7-950
  - Quad cores
  - 12GB DDR3 memory
- Nvidia® GeForce® GTX580
  - 512 cores
  - 1536MB GDDR5 memory
- Patterns: String pattern extracted from Snort V2.8, containing 126,776 states, 10,076 final states (patterns)
- Input: 256MB packets extracted from DEFCON

Performance and Memory Evaluation

name	rules	states	Mem of State table	Hash memory / 2D memory	Throughput (Gbps)
PFAC-v1.1 Time-driven	**10,076**	**126,776**	**123.8MB**		**120.59**
PFAC-v1.1 Space-driven (module free)	**10,076**	**126,776**	**2.45MB**	**0.020**	**91.79**
SPHM	**10,076**	**126,776**	**620KB**	**0.005**	**100.76**