Accelerating String Matching on Graphic Processing Units

Cheng-Hung Lin
National Taiwan Normal University
Taipei, Taiwan

Outline

- Introduction
- Parallel Failureless Aho-Corasick Algorithm
- Memory-Efficient Memory Architecture
- M-DFA (Multithreaded DFA) for Regex Matching

String Matching

- String matching engine plays an important role in many applications, such as network intrusion detection systems, spam filters, and bioinformatics.

- String matching is used to find a place where one or several strings (also called patterns) are found within a larger string or text.

Patterns
"TACT"
"CTO"
"TOE"

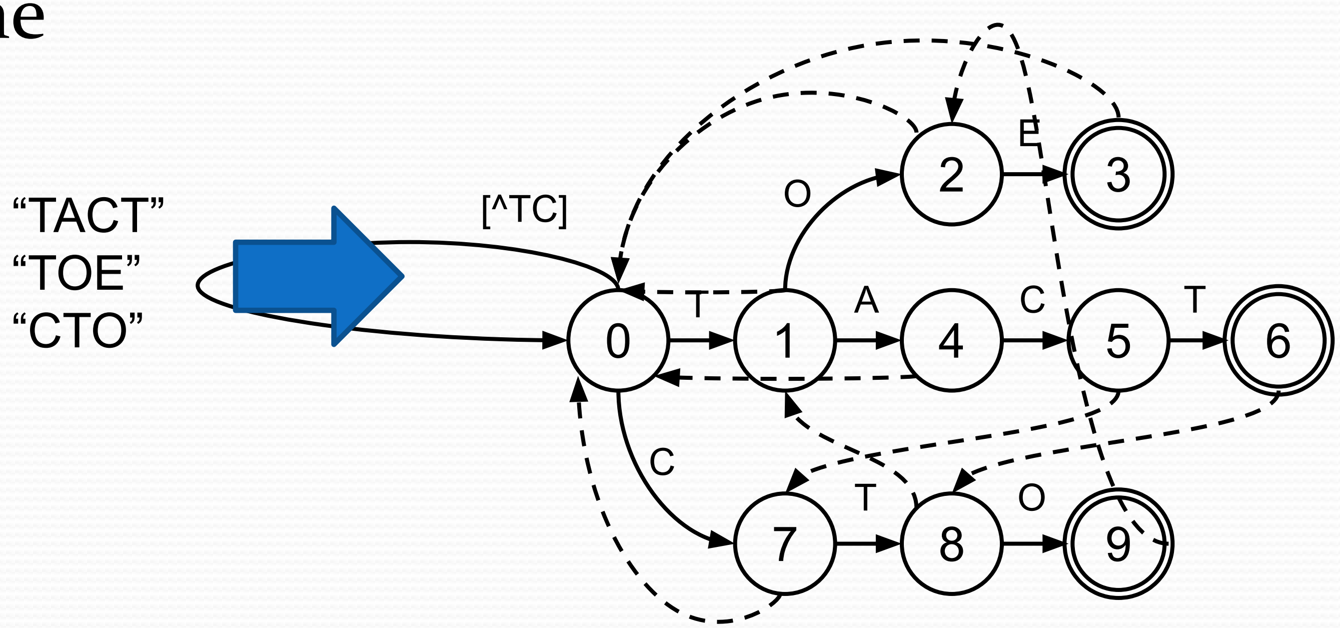
Input string
T A C T O E

T A C T
C T O
T O E

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

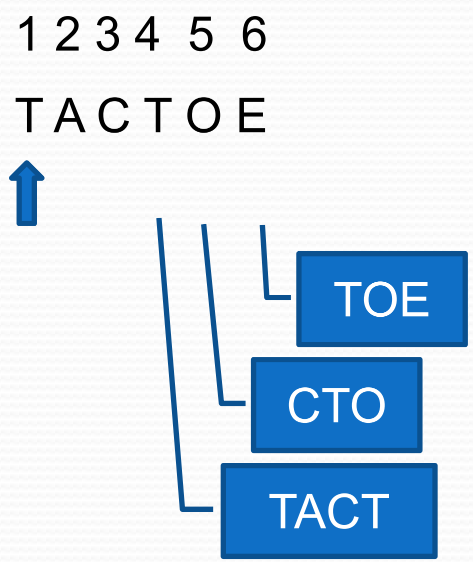
- Aho-Corasick algorithm compiles multiple string patterns into a state machine

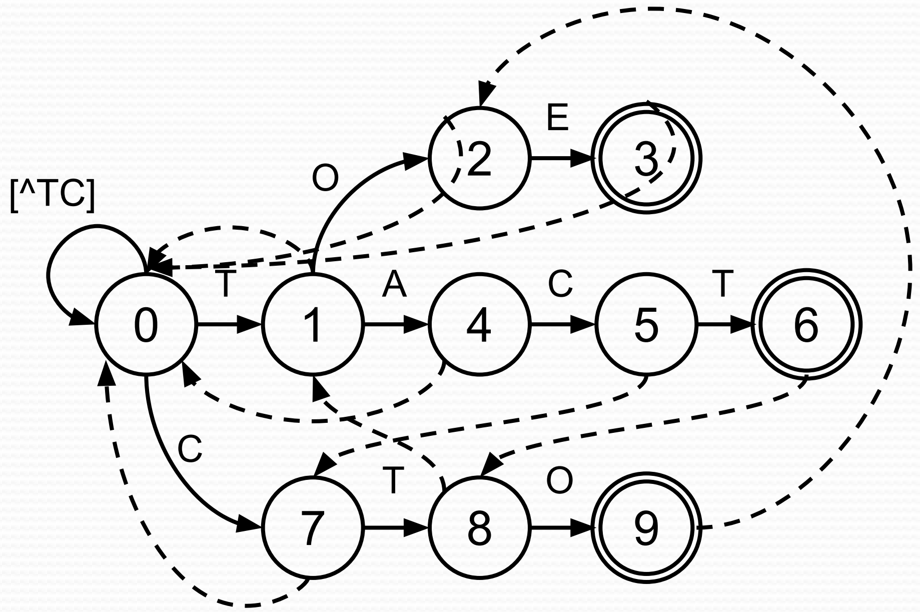


Aho-Corasick Algorithm (cont.)

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

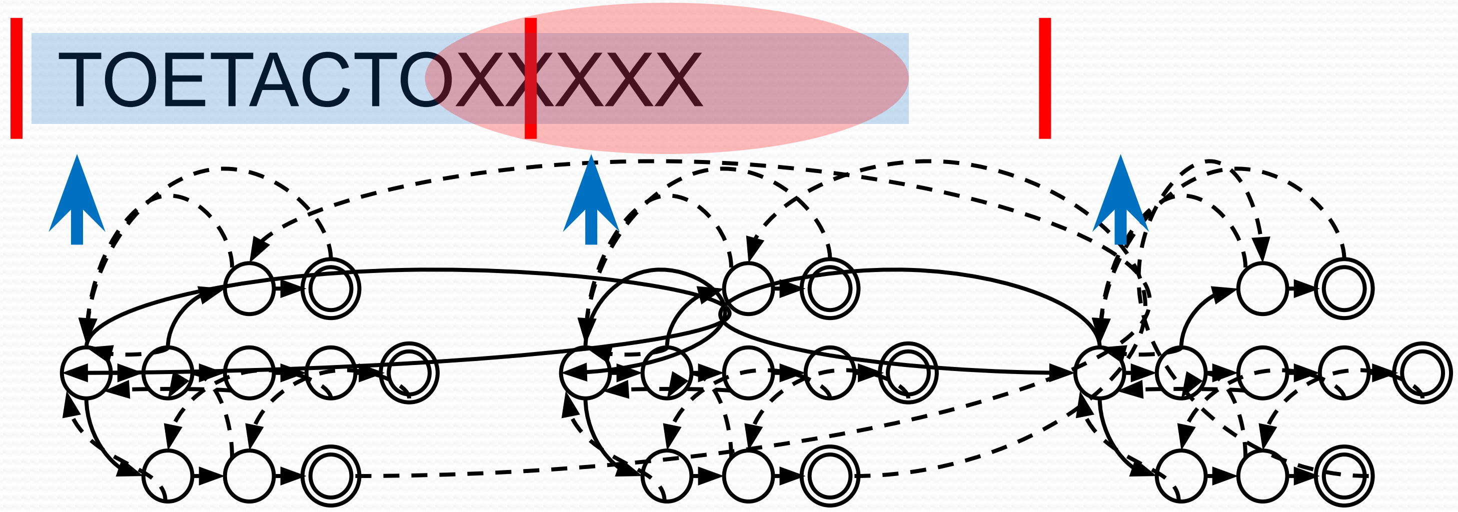
- Failure transitions are used to backtrack the state machine to recognize patterns in different locations.





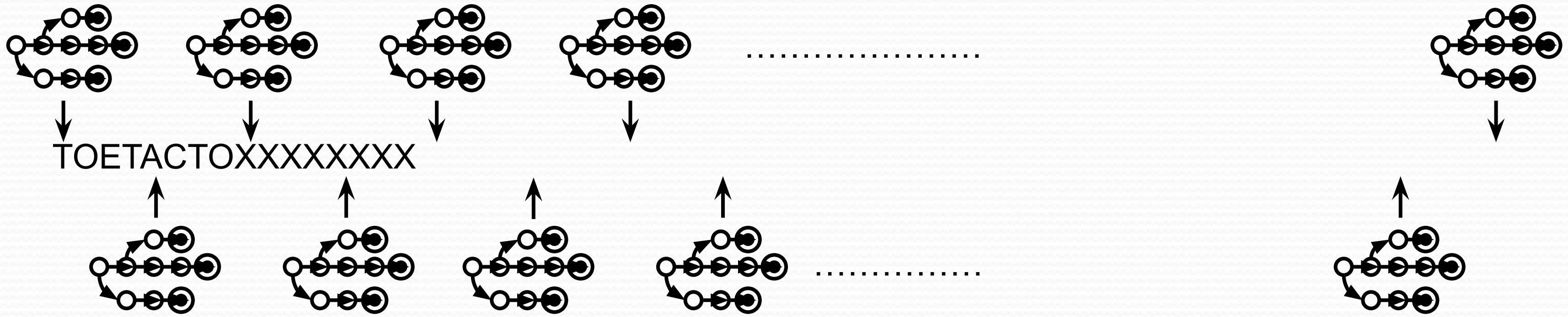
Naïve Data Parallel Approach

- 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



Parallel Failureless Aho-Corasick Algorithm

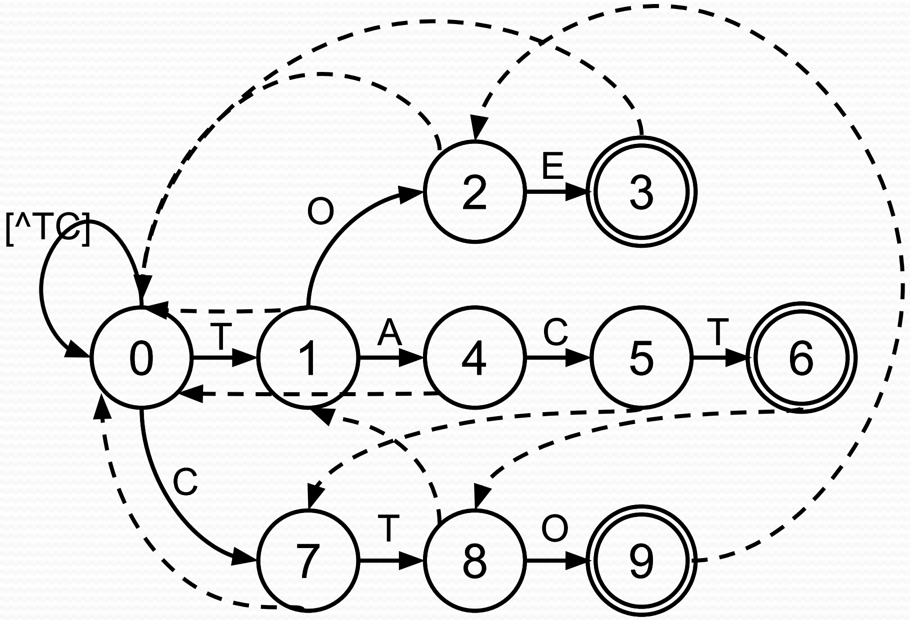
- Parallel Failureless Aho-Corasick (PFAC) algorithm on graphic processing units
  - Allocate each byte of input an individual thread to traverse a state machine



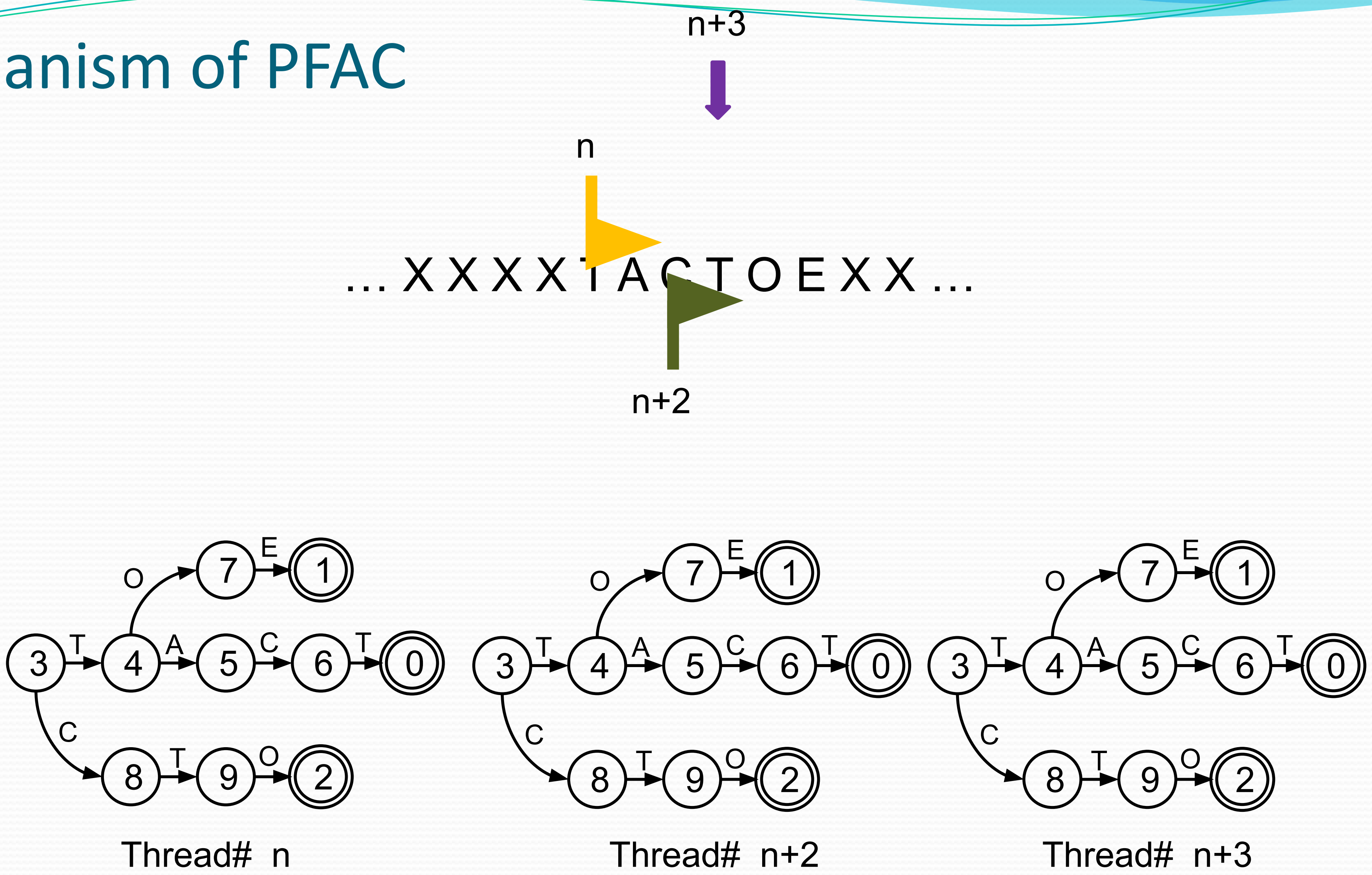
- Reference:
  - C.-H. Lin *et al.*, "Accelerating String Matching Using Multi-Threaded Algorithm on GPU," in *GLOBECOM 2010*.
  - C.-H. Lin *et al.*, "Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs", IEEE Transactions on Computers

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



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^{TM} i7 using a single thread and optimized by GCC 4.4.3 using the compiler flags "-O2 –msse4".

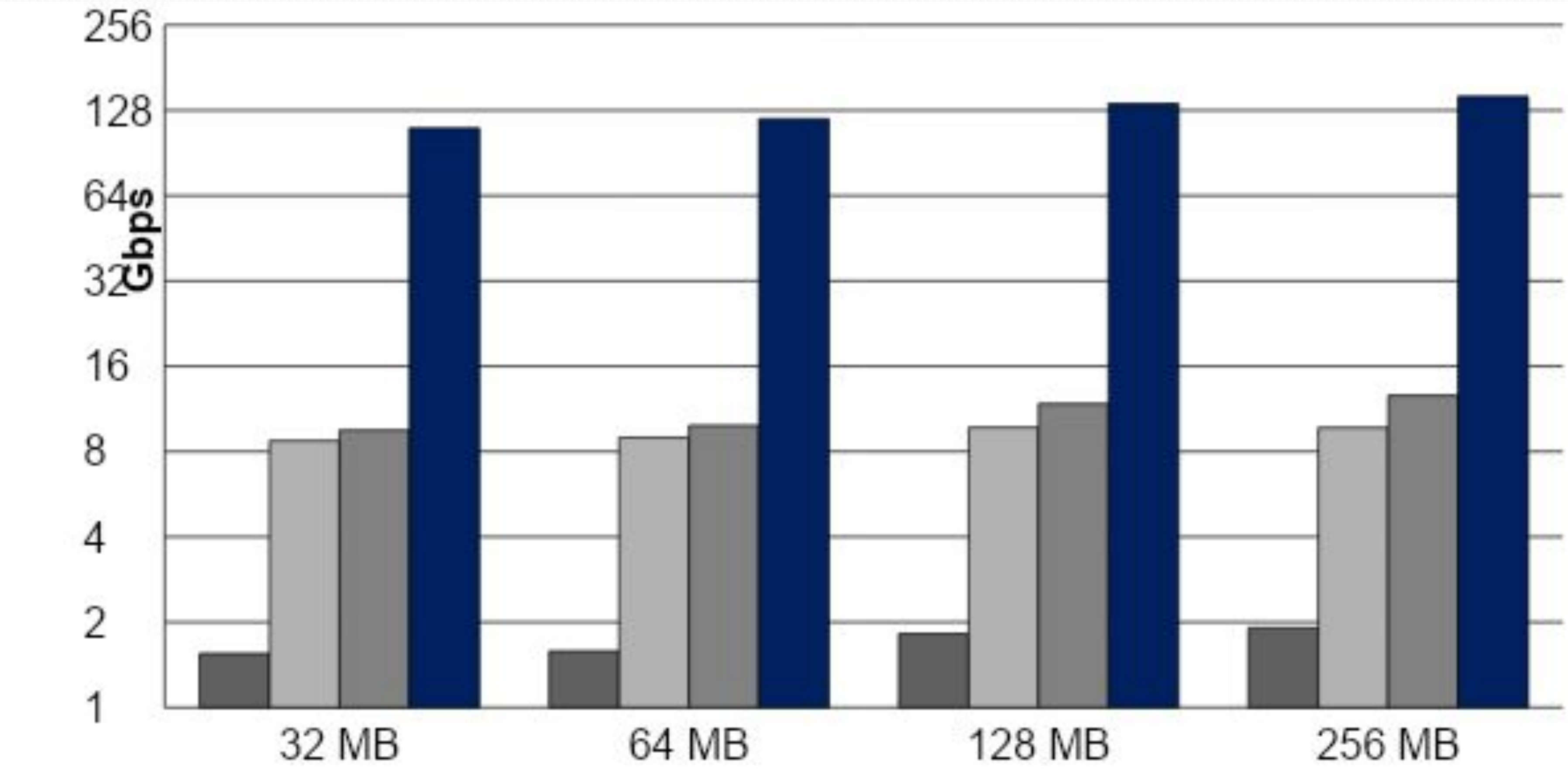
- $DPAC_{OMP}$: implementation of the DPAC algorithm on Intel Core^{TM} i7 CPU with OpenMP and optimized by GCC 4.4.3 using the compiler flags "-O2 –msse4".

- $PFAC_{OMP}$: implementation of the PFAC algorithm on Intel Core^{TM} 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

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

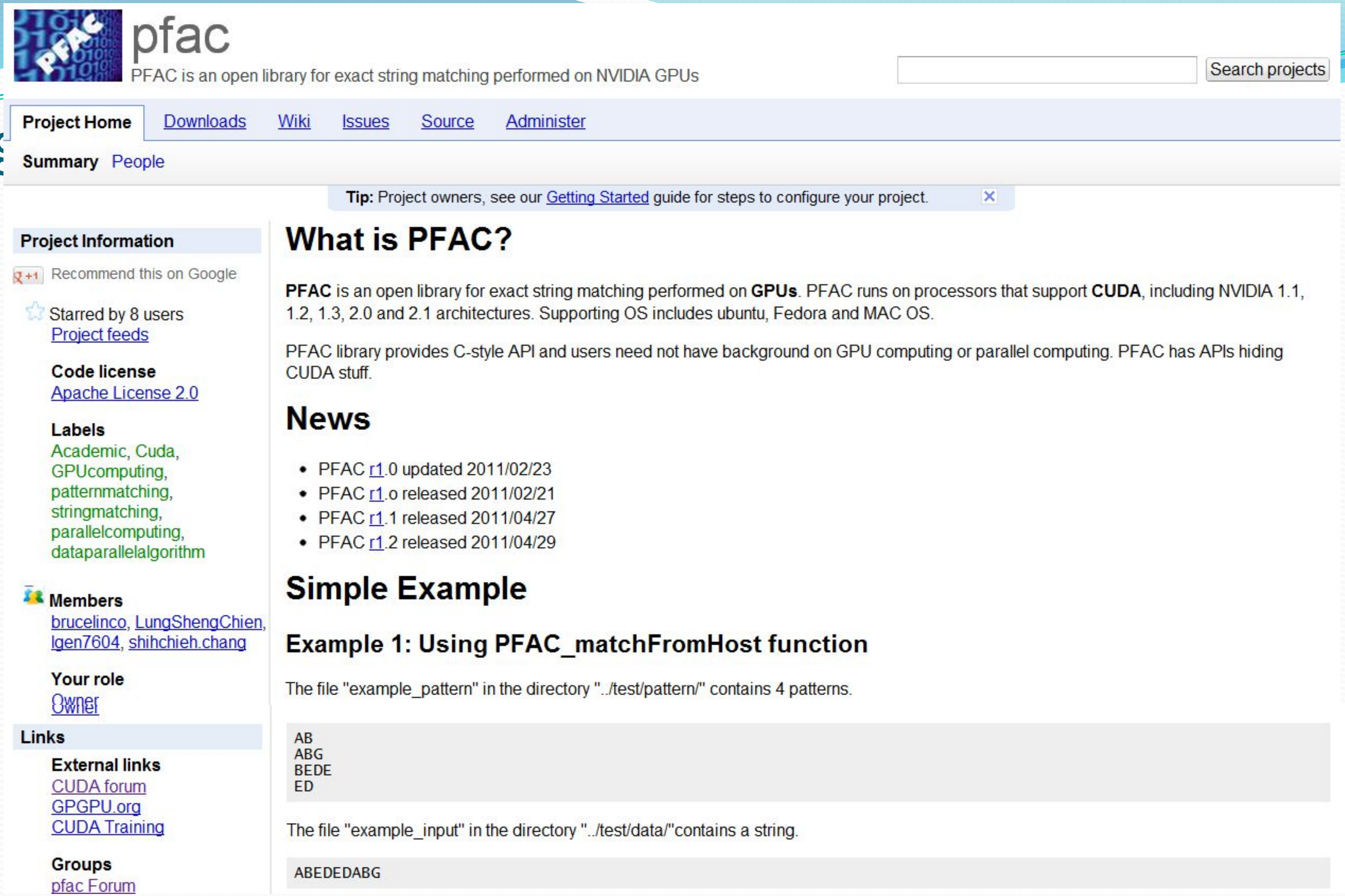


PFAC Library

- **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 of GPU programming

Using



pfac

PFAC is an open library for exact string matching performed on NVIDIA GPUs

Search projects

Project Home | Downloads | Wiki | Issues | Source | Administer

Summary People

Tip: Project owners, see our Getting Started guide for steps to configure your project.

Project Information

Recommend this on Google

Starred by 8 users
Project feeds

Code license
Apache License 2.0

Labels
Academic, Cuda, GPUcomputing, patternmatching, stringmatching, parallelcomputing, dataparallelalgorithm

Members
brucelinco, LungShengChien, lgen7604, shihchieh.chang

Your role
Owner

Links

External links
CUDA forum
GPGPU.org
CUDA Training

Groups
pfac Forum

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.

```c
#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) ;
    }
```

```c
//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.

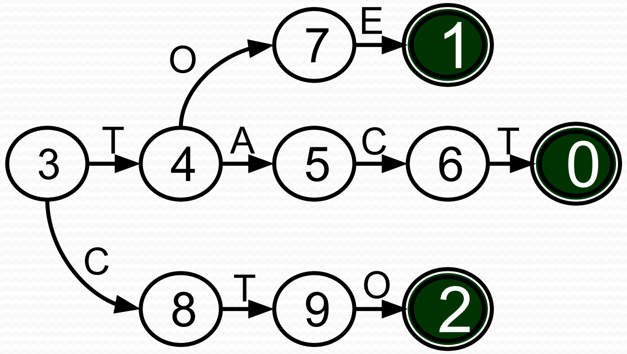
```
At position    0, match pattern 1
At position    1, match pattern 3
At position    2, match pattern 4
At position    4, match pattern 4
At position    6, match pattern 2
```

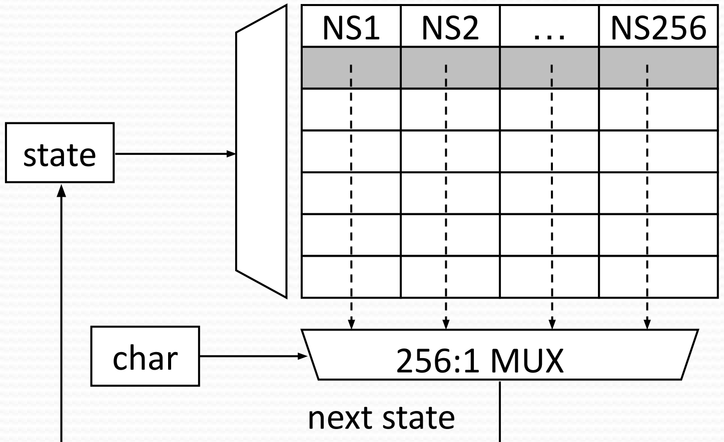
Outline

- Introduction
- Parallel Failureless Aho-Corasick Algorithm
- Memory-Efficient Memory Architecture
- M-DFA (Multithreaded DFA) for Regex Matching

Memory Issue of PFAC

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

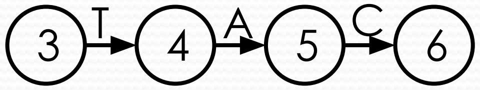


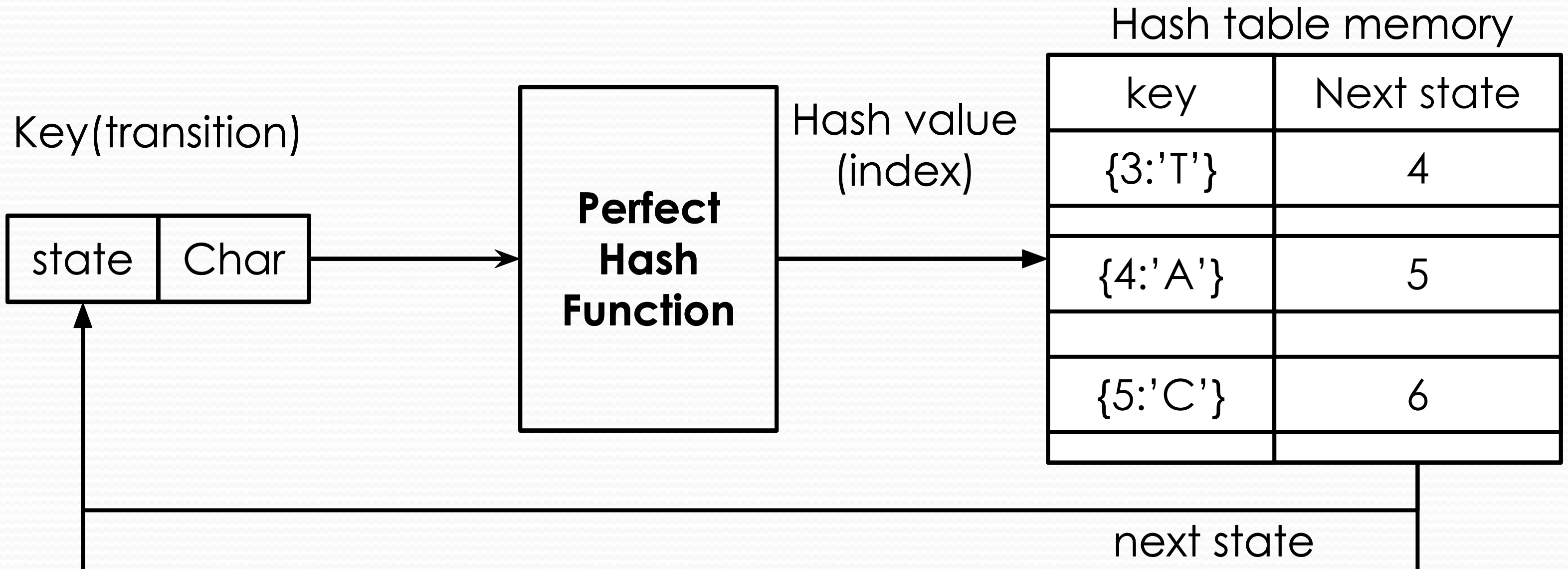


- Design a compact storage mechanism for storing PFAC state transition table is essential for GPU implementation.

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, *et al*. "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.

- Reference
    - R. E. Tarjan and A. C.-C. Yao, "Storing a sparse table," Commun. ACM, vol. 22, pp. 606-611, 1979.

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 $row = k / w$, $column = k \bmod w$.

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] = 9			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.

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

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

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* = 19

row = 19 / 8 = 2

col = 19 mod 8 = 3

index = RT[2] + 3 = 1 + 3 = 4

HK[4] =14

19 is an invalid 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 ;

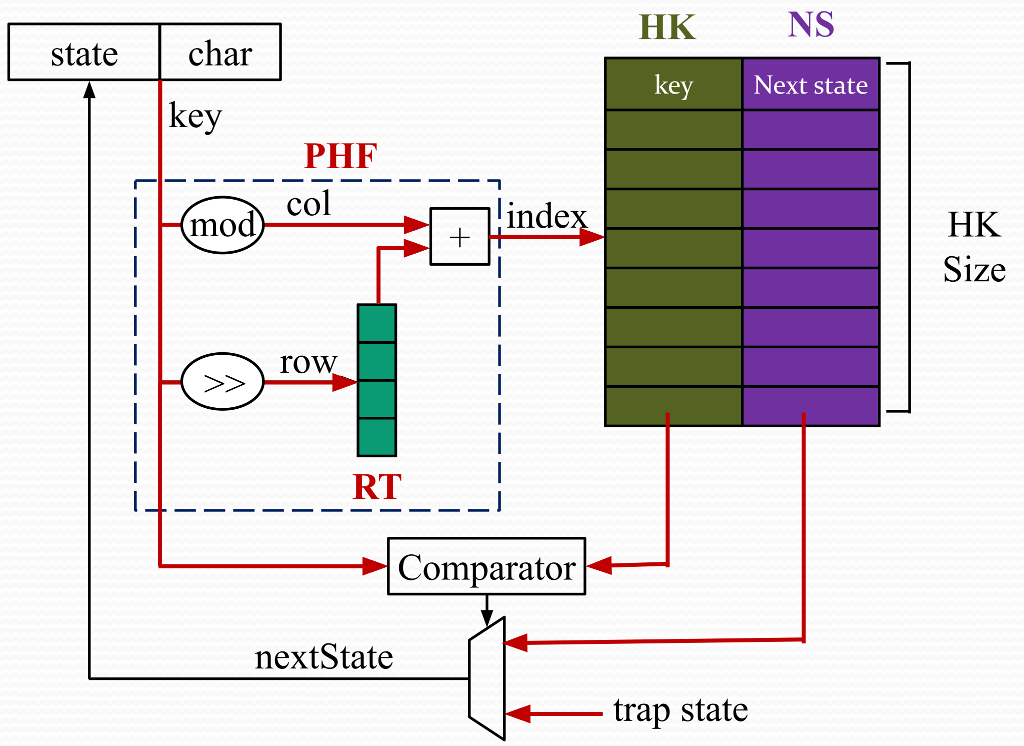
    else

    k is an invalid key ;
  - If *k* is a valid key

    nextState = NS[*index*] ;

    else

    nextState = trap state;



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

Experimental Results

	# of Rules	# of char	Memory (Bytes)	$\frac{mem}{char}$	Throughput (Gbps)
PHM	**10K**	**187K**	**620KB**	**3.39B**	**100.76**
	2K	**41K**	**137KB**	**3.34B**	**135.93**
PFAC	2K	41K	27.1MB	677B	146.63
B-FSM	39.5K	25.2K	188KB	7.4B	2
CDFA	1,785	29.0K	129KB~ 256KB	4.45B~ 8.2B	11.7
Bitmap Compression	1.5K	18.2K	2.8MB	154B	7.6
Path Compression	1.5K	18.2K	1.1MB	60B	7.6

*PHM : Perfect Hashing Memory Architecture

Conclusions

- The PFAC algorithm is adaptive to be implemented on GPUs and multicore CPUs.
- The perfect hash algorithm significantly reduces the memory for storing state transition table with little penalty on performance.

M-DFA (Multithreaded DFA): An Algorithm for Reduction of State Transitions and Acceleration of REGEXP Matching

Cheng-Hung Lin National Taiwan Normal University Taipei, Taiwan brucelin@ntnu.edu.tw	Jyh-Charn Liu Dept. of Computer Science & Engineering Texas A&M University liu@cse.tamu.edu

Regular Expression Matching

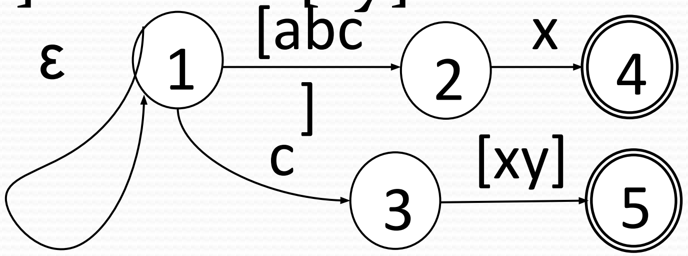
- Regular Expression (REGEXP) matching is typically implemented as a **nondeterministic finite automaton** (**NFA**) or its equivalent **deterministic finite automata** (**DFA**).

- NFA has smaller sizes in terms of memory space, but it may take multiple cycles to match an input symbol when multiple states become *active* concurrently.

- An NFA can be mapped to its DFA equivalence by mapping concurrent active states in NFA to one single active state in the DFA.
  - State explosion arises during integration of multiple regexps into a DFA

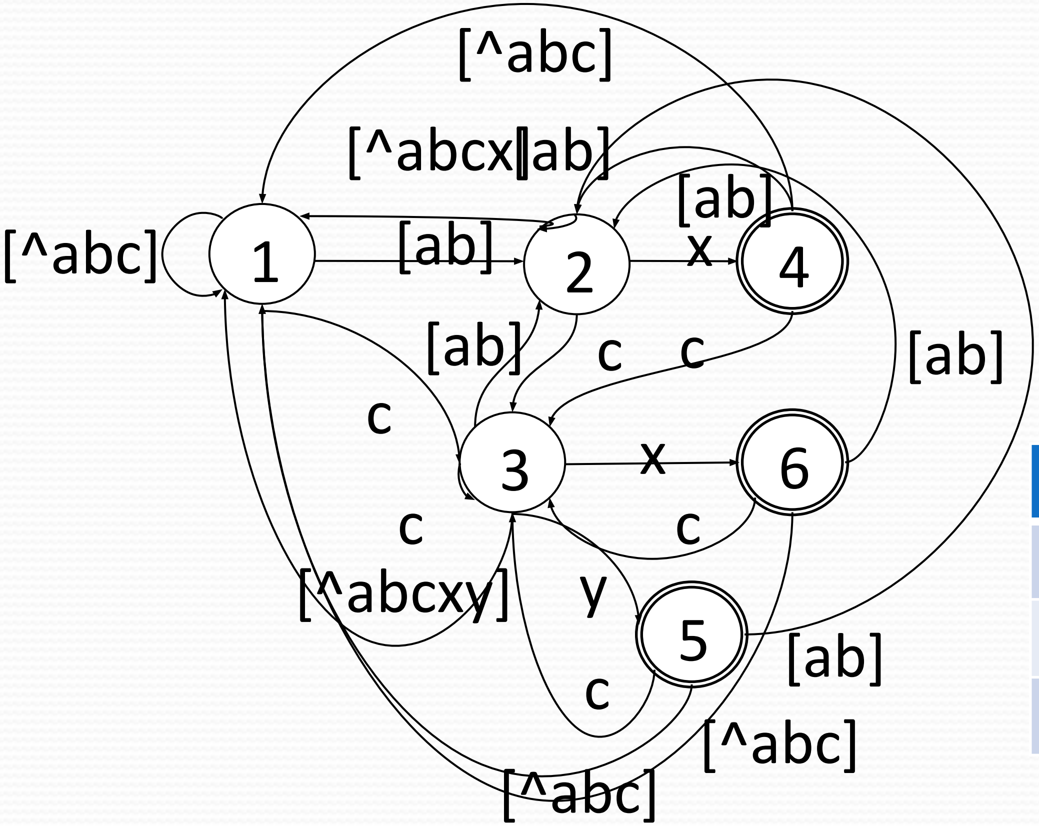
NFA and DFA

- **NFA of “[*abc*]*x*” and “*c*[*xy*]”**



state	Match Patterns
4	“[*abc*]*x*”
5	“*c*[*xy*]”

- **DFA**



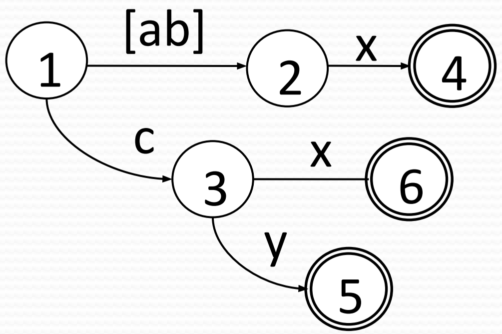
state	Match Patterns
4	“[*abc*]*x*”
5	“*c*[*xy*]”
6	“[*abc*]*x*” and “*c*[*xy*]”

M-DFA (multithreaded DFA)

- Multi-thread based regular expression (regexp) matching algorithm for parallel computer architectures such as multi-core processors and graphic processing units (GPU)
  - Each input symbol is treated as the first symbol of a possibly matched substring to the regexp
  - A DFA free of backtracking transitions is assigned to a thread, which terminates either when it reaches the final state or any mismatch occurs,
  - Multiple threads concurrently read in each input symbol for their independent matching.

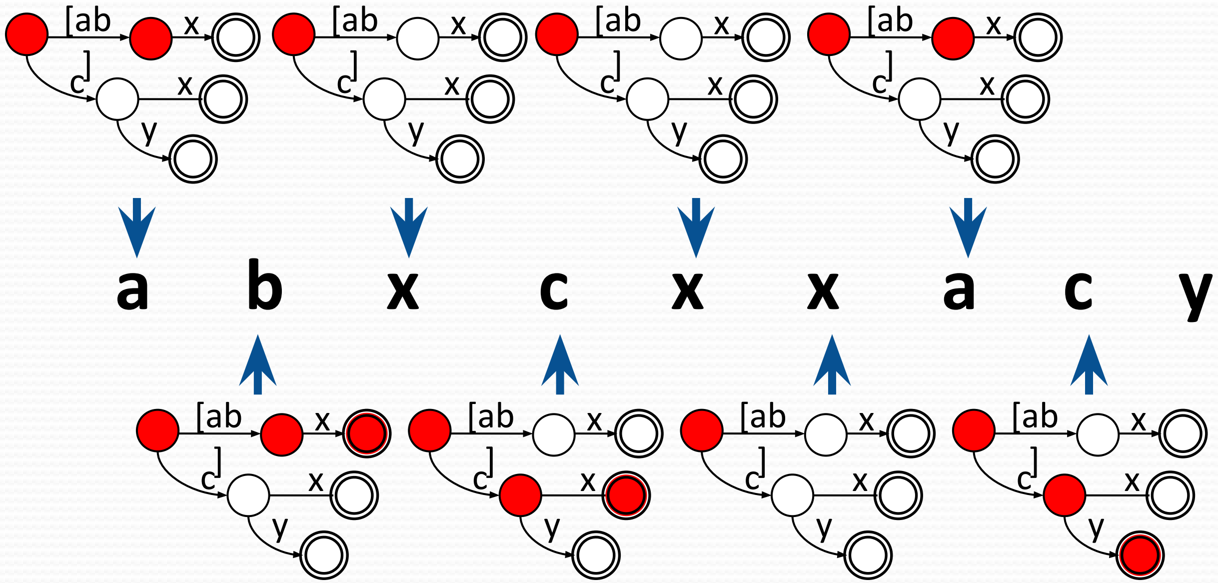
Multi-threaded M-DFA Model

- DFA_{LF}



state	Match Patterns
4	"[*abc*]*x*"
5	"*c*[*xy*]"
6	"[*abc*]*x*" and "*c*[*xy*]"

- Multi-threaded execution of multiple DFA_{LF}



Construction of DFA_{LF}

- Three steps:
  1. Convert regexps into an NFA by Thompson's algorithm
  2. Convert the NFA into an equivalent DFA by Rabin-Scott powerset construction algorithm
  3. Remove all backtracking transitions of the DFA

Advantages of M-DFA

- M-DFA resembles the behavior of an NFA at the string matching level, yet each thread is a DFA.

- M-DFA is guaranteed to eliminate backtracking transitions.
  - Significant memory reduction

Experimental Results

- M-DFA on Nvidia® GeForce® GTX480 GPU
- RE2 [5] on Intel Core™ i7-950 CPU
- Benchmarks: 16 regular expression patterns and 2 exact string patterns from a public benchmark
- 35 times faster than RE2
- 44% of state reduction and 99.8% of transition reduction

	RE2 library			$M\text{-}DFA_{GPU}$						
Input size (bytes)	# of states	# of state transition	elapsed time(ms)	# of states	state reduction	# of state transition	Transition reduction	GPU elapsed time (ms)	Total elapsed time (ms)	speedup
100K	162	41,472	14.08	91	44%	90	99.8%	0.05	0.35	40.23
32M			2556					5.59	69.24	36.92
64M			4980					11.16	139.32	35.74

Thanks for your attention!

Q&A