Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPU

Cheng-Hung Lin*, Chen-Hsiung Liu**, Lung-Sheng Chien***, Shih-Chieh Chang**
*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—**Graphics processing units (GPUs) have attracted a lot of attention due to their cost-effective and enormous power for massive data parallel computing. In this paper, we propose a novel parallel algorithm for pattern matching on GPUs. Traditional pattern matching algorithms traverse a special state machine called Aho-Corasick machine. Considering the particular parallel architecture of GPUs, in this paper, we first propose an efficient state machine on which we can perform very efficient parallel algorithms. Also several techniques are introduced to do optimization on GPUs, including reducing the global memory transaction of the input buffer, reducing latency of transition table lookup, avoiding bank-conflict of the shared memory, and coalescing write to the global memory. We evaluate the performance of the proposed algorithm using attack patterns from Snort V2.8 and input streams from DEFCON. The experimental results show that the proposed GPU algorithm achieves significant performance enhancements compared to the traditional Aho-Corasick algorithm implemented on CPUs and outperforms all other state-of-the-art GPU approaches.**

I. INTRODUCTION

Network Intrusion Detection Systems (NIDS) have been widely used to protect computer systems from network attacks such as denial of service attacks, port scans, or malware. The pattern matching engine used to identify network attacks by inspecting packet content against thousands of predefined patterns is the most critical operation affecting the performance of NIDS. Due to the ever-increasing number of attacks and network complexity, traditional sequential pattern matching algorithms have become inadequate for high-speed networks.

In the past few years, many approaches have been proposed to accelerate pattern matching. Hardware approaches can be classified into logic [1][2][3][4] and memory [5][6][7][8][9][10] architectures. Logic architectures convert attack patterns into logic circuits and implement on field-programmable gate array (FPGA) to match multiple patterns in parallel while memory architectures compile attack patterns into state machines and use commodity memories to store state transition tables for pattern matching.

To accelerate pattern matching, recently GPUs have been adopted because of their enormous power for massive data parallel computing. Huang *et al.* [12] propose a Wu-Manber-like, GPU-based multiple-pattern matching algorithm and achieved speeds twice as fast as the modified Wu-Manber algorithm used in Snort. Smith *et al.* [13] propose a programmable signature matching system prototyped on an NVIDIA G80 GPU. This system out-performs a Pentium4 by up to 9x and a Niagara-based 32-threaded system by up to 2.3x. Tumeo *et al.* [14] present an efficient implementation of the Aho-Corasick (AC) algorithm [15] for accelerating DNA analysis on GPU clusters. Peng *et al.* [16] propose a GPU-based advanced AC algorithm for webpage matching system and achieved speeds that were 28 times faster than the AC algorithm used in Snort [17]. Vasiliadis *et al.* [18] propose a GPU-based regular expression matching engine for Snort patterns. Kouzinopoulos *et al.* [19] present experimental results on the parallel processing for some well known on-line string matching algorithms using GPUs. Mu *et al.* [20] develop efficient GPU implementations for a series of key router applications including IP routing table lookup and pattern match for network intrusion detection.

In this paper, we exploit the parallelism of pattern matching algorithms and propose a novel parallel algorithm that is performed on multi-core GPUs. To efficiently utilize the power of GPUs, we also introduce several throughput-oriented techniques. These techniques can effectively reduce the global memory transaction, reduce the latency of transition table lookup, avoid bank-conflict of the shared memory, coalesce writing to the global memory, and enhance GPU/CPU communication. Finally, we measure the performance of the proposed algorithm by attack patterns from Snort V2.8 and input stream from DEFCON [21]. The experimental results show that the proposed algorithm performed on the Nvidia® GeForce® GTX-480 achieves significant speedup compared to the traditional AC algorithm performed on the Intel® Core^{TM} i7-950 processor.

II. AHO-CORASICK ALGORITHM

Among string matching algorithms, the Aho-Corasick algorithm has been widely adopted for string matching due to its advantage of locating all occurrences of patterns within an input stream simultaneously. Different from the deterministic finite automaton (DFA), the AC algorithm introduces a new transition called the failure transition to reduce the number of outgoing transitions of each state.

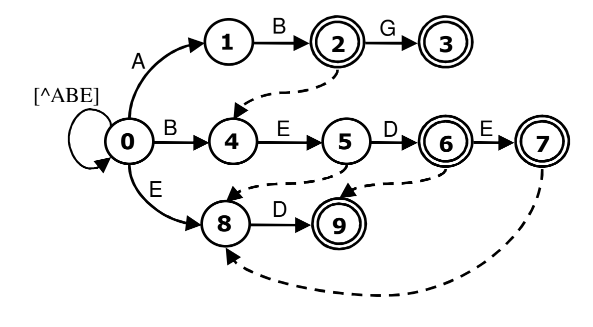


Figure 1. AC state machine of the patterns “AB”, “ABG”, “BEDE”, and “ED”

The failure transitions are used to back-track the state machine to recognize patterns in any location of an input stream. Given a current state and an input character, the AC machine first looks up the valid transition table to check whether there is a valid transition for the input character; otherwise, the machine looks up the failure transition table and jumps to the failure state where the failure transition points. Then, the machine regards the same input character until the character causes a valid transition. Figure 1 shows an AC state machine matching patterns, “AB”, “ABG”, “BEDE”, and “ED” where the double-circled nodes represent the final states of matched patterns, the solid lines represent the valid transitions and the dotted lines represent the failure transitions. As shown in Figure 1, states 2, 3, and 7 are the final states of the patterns “AB”, “ABG”, and “BEDE” while states 6 and 9 represent the final state of the pattern “ED”. The internal state 6 becomes a final state because the pattern “ED” is a suffix of the pattern “BED”. Therefore, when reaching state 6, the state machine matches the pattern “ED”.

For example, consider the use of AC state machine in Figure 1 to match an input stream containing “ABEDE”. The AC state machine starts from state 0, state 1, to state 2 which is the final state of the pattern “AB”. Because state 2 has no valid transition for the input “E,” the AC state machine first takes a failure transition to state 4 and then regards the same input “E” leading to state 5. After taking the next character “D”, the state machine reaches state 6, the final state of the pattern “ED”. Finally, the AC state machine reaches state 7 which is the final state of pattern “BEDE”. Therefore, we find that the AC algorithm matches the three patterns “AB”, “BEDE”, and “ED” at the position of “A”, “B”, and “E”, using a single thread to traverse the AC state machine. In summary, the AC algorithm can match all patterns simultaneously in time O(n) for processing an input stream of length n.

III. PROBLEMS OF SIMPLE PARALLEL APPROACH OF Aho-Corasick ALGORITHM

In this section, we discuss a data-parallel approach [22][23][24][25][26] to parallelize the AC algorithm, which is referred to as the data-parallel AC (DPAC) approach. The DPAC approach first divides an input stream into multiple segments and allocates each segment an individual thread to perform string matching. However, the DPAC approach has a problem, in that it cannot detect a pattern occurring in the boundary of adjacent segments. We call the new problem the “*boundary detection*” problem. For example, in Figure 2, the pattern “BEDE” occurs in the boundary of segments 3 and 4 and cannot be identified both by threads 3 and 4.

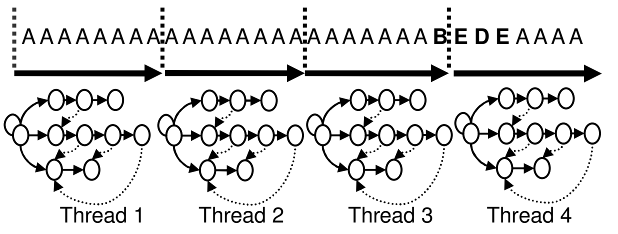


Figure 2. Simple parallel approach which incurs “*boundary detection*” problem.

To resolve the boundary detection problem, each thread must scan across the boundary. The additional length which needs to be scanned across the boundary equals to the length of the longest pattern minus one. The overhead caused by scanning the additional length across the boundary is referred to as overlapped computation. For example, in Figure 2, thread 3 can find the pattern “BEDE” by scanning an additional length of 3 characters. The complexity of overlapped computation is as follows. Each thread of DPAC works in time O($n/s + m$) where m is the longest pattern length, n is the total length and s is the number of segments. Therefore, the times complexity of memory-lookup is O((n/s + m)*s) = O(n + ms). Compared to the AC algorithm whose memory-lookup complexity is O(n), DPAC needs more memory-lookup operations. Note that the memory-lookup is the bottleneck of GPU computation.

IV. PARALLEL FAILURELESS-AC ALGORITHM

In this section, we propose a novel algorithm which efficiently exploits the parallelism of the AC algorithm, the algorithm of which is referred to as *Parallel Failureless-AC* (*PFAC*) *Algorithm.* As shown in Figure 3, the PFAC Algorithm creates an individual thread for each byte of an input stream to identify any pattern starting at the thread’s starting position. The number of threads created by the PFAC algorithm is equal to the length of an input stream.

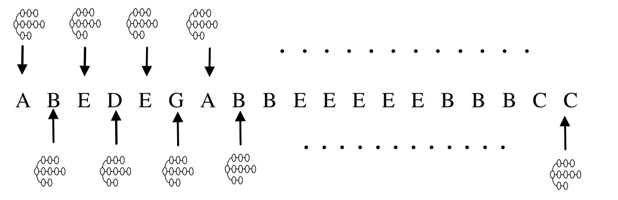


Figure 3. Parallel Failureless-AC algorithm

The idea of allocating an individual thread to each byte of an input stream has important implications on the efficiency of the PFAC state machine. First, each thread of PFAC is only responsible for identifying the pattern starting at the thread starting position. Therefore, whenever a thread cannot find any pattern located at its beginning position, it terminates immediately without taking failure transitions to back-track the state machine. Therefore, in PFAC, the failure transitions of the AC state machine can all be removed as well as the self-loop transition of the initial state. An AC state machine with all its failure transitions and self-loop transition removed is called *Failureless-AC state machine*. Figure 4 shows the Failureless-AC state machine to identify the four patterns, “AB”, “ABG”, “BEDE”, and “ED”.

Second, each final state of the PFAC machine represents a unique pattern. In the tradition AC algorithm, a final state may represent more than one pattern. For example, if we add a new pattern “B” into the state machine in Figure 1, state 4 will become a final state and state 2 will be modified as the final state of patterns “AB” and “B” because “B” is a suffix of “AB”. On the other hand, if we add the same pattern “B” into the PFAC state machine in Figure 4, only state 4 becomes a final state which represents the pattern “B”.

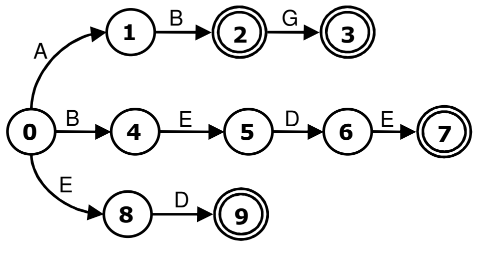


Figure 4. Failureless-AC state machine of the patterns “AB”, “ABG”, “BEDE”, and “ED”

We now use an example to illustrate the PFAC algorithm. Consider an input stream which contains a substring “ABEDE”. In Figure 5, the thread t_n is allocated to the input “A” to traverse the Failureless-AC state machine. After taking the input “AB”, t_n reaches state 2, which indicates the pattern “AB” is matched. Because there is no valid transition for “E” in state 2, t_n terminates at state 2. Similarly, the thread t_{n+1} is allocated the input “B”. After taking the inputs “BEDE”, the thread t_{n+1} reaches state 7, which indicates the pattern “BEDE” is matched. Moreover, the thread t_{n+2} reaches state 9 which indicates the pattern “ED” is matched. Instead of using one thread to find the three patterns in the DPAC approach described in section III, the PFAC algorithm uses three threads to find the three patterns in parallel.

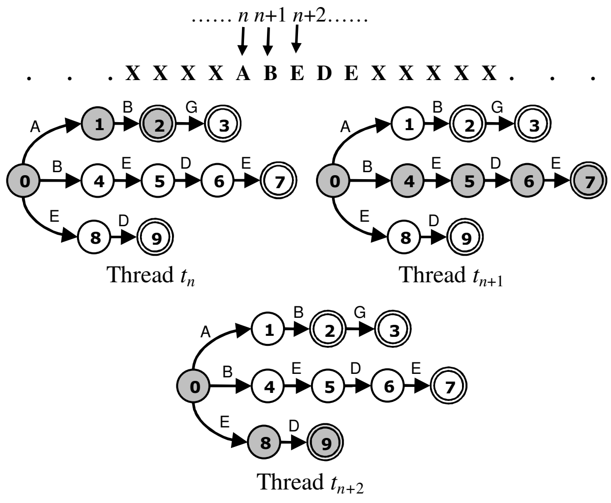
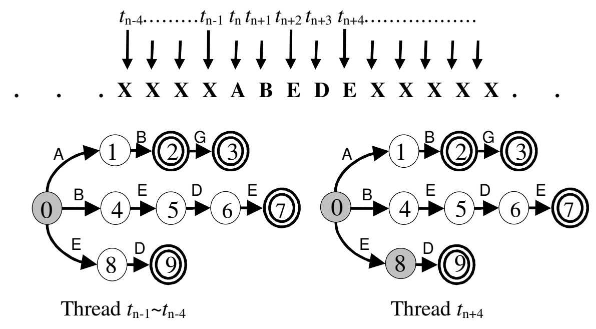


Figure 5: Example of PFAC

There are several characteristics of a PFAC algorithm. First, although PFAC creates huge amounts of threads, most threads have a high probability of terminating very early because a thread of PFAC is only responsible for matching the pattern beginning at its starting position. As shown in Figure 7, threads t_{n-1} to t_{n-4} terminate early at state 0 because there are no valid transitions for “X” in state 0. The thread t_{n+4} terminates early at state 8 because there are no valid transitions for the input “X” in state 8.


t_{n-4}………t_{n-1} t_n t_{n+1} t_{n+2} t_{n+3} t_{n+4}………………
. . . X X X X A B E D E X X X X X . .
Thread t_{n-1}~t_{n-4}
Thread t_{n+4}
Figure 6. Most threads terminate early

Second, because string matching by traversing a state transition table is intrinsically a memory-bounded application, the performance is limited by the latency/throughput of memory accesses. In DPAC, each thread is assigned to the first position of a segment. This approach results in worse spatial locality of memory accesses. In contrast, in PFAC, threads are assigned to subsequent locations, which results in better spatial locality of memory accesses. Because PFAC exhibits better spatial locality, PFAC has more opportunities to use small and high-speed memory such as the shared memory of GPUs to serve most of the accesses.

Third, GPUs feature a 32-wide SIMT (Single Instruction Multiple Threads) architecture which creates, manages, schedules, and executes threads in groups of 32 threads called *warp*. When GPU hardware detects all threads in the same warp access subsequent locations of the global memory, CUDA has a special technique known as “*memory coalescing*” which allows GPU hardware to combine all these accesses into a consolidated access to subsequent DRAM locations. Because each thread of PFAC accesses consecutive global memory to read input characters, all these accesses are coalesced into a single request for all consequent DRAM locations. The technique of memory coalescing allows the DRAMs to deliver data at a rate close to the peak bandwidth of the global memory. In contrast, in DPAC, each thread of DPAC reads input characters at discrete memory locations. None of these accesses can be coalesced.

Finally, the memory usage of the PFAC algorithm is smaller than the DPAC algorithm, due to the removal of all failure transitions.

V. Optimization of GPU Implementation

In this section, we first analyze the PFAC implementation on GPUs and then propose several throughput-oriented optimization techniques considering the particular hardware architecture of GPUs.

Before discussing how PFAC algorithm can be optimized for the GPU architecture, we first analyze the cost of each operation of PFAC. Figure 8 shows a single thread version of the PFAC algorithm which takes four steps. First, the PFAC algorithm reads a character *ch* indexed by *pos* from *input_buffer* at line 5. And then, in the second step, the PFAC algorithm uses the current state *state* and the input character *ch* as an index to retrieve the next state information from the PFAC_table and assign to the variable *nextState*, at line 6. The *PFAC_table* stores the next state information and is arranged as a two-dimensional array where each row represents a state and each column represents an input character. In PFAC, a thread terminates if its next state is a trap state which indicates there is no valid transition for the current state and input character. A positive number is stored for a valid next state while a negative number, -1 is stored for a trap state. The lookup of the PFAC_table is efficient because only one random access of memory is required to query the next state information for a given row index (current state) and a column index (input character).

Then, in the third step, the PFAC retrieves match vectors from the output_table and assigns it to a variable *matchVec*, at line 8. The output_table is organized as a one-dimensional array which stores the match vector of a state. The match vectors of final states are represented as positive numbers while the match vectors of all other internal states are 0.

Finally, the PFAC algorithm checks whether the next state is a final state by checking the value of matchVec. If the matchVec is a positive number, the next state is a final state, and then the matchVec is written into a one-dimensional array defined as *match_result*, at line 10.

```
1   for (int firstpos = 0; first < size; first++) {
2       int pos = firstpos;
3       int state = 0;
4       while ( (state != -1) && (pos < size) ) {
5           char ch =input_buffer[pos];
6           int nextState = PFAC_table[state][ch];
7           pos = pos + 1;
8           int matchVec = output_table[nextState];
9           if (matchVec > 0) {
10              match_result[start] = matchVec;
11          }
12          state=nextState;
13      }
14  }
```

Figure 7. Single thread version of the PFAC algorithm

We can observe that the PFAC algorithm is a memory-bound application because it accesses the global memory four times. The memory access includes (1) reading input characters from the input buffer at line 5, (2) reading next state information from the PFAC_table at line 6, (3) reading match vectors from the output_table at line 8, and (4) writing match results into the match_result array at line 10. Due to the fact that the global memory implemented with dynamic random access memory which typically has long latencies and finite access bandwidth, it's imperative to reduce the frequency of accessing the global memory to achieve high performance. In the following section, we discuss several techniques for optimizing the performance of the PFAC algorithm implemented on GPUs.

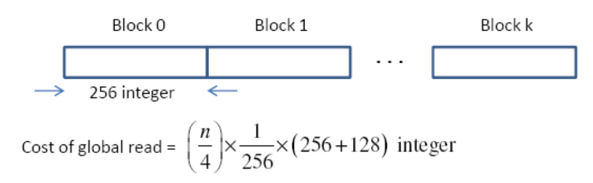
4.1 Reducing memory transaction of input stream

The PFAC deploys *n* threads to an input stream of length *n.* Suppose thread *i* reads *l*(*i*) characters to reach either a final state or a trap state, then the total number of characters read by all threads is $\sum_{i=1}^{n} l(i)$ where $n \le \sum_{i=1}^{n} l(i) \le mn$ and *m* is the longest pattern length.

First, we observe that it is not necessary to directly read input characters from the global memory because the processing window of each thread is overlapped with its adjacent threads. We propose a two-step approach which loads data from the global memory to the shared memory and fetches data from the shared memory.

As shown in Figure 9, the input stream is divided into multiple blocks. Each block consists of 256 32-bits integers (1,024 8-bits characters) and is dealt with a thread block consisting of 256 threads. Figure 10 shows the procedure to move 384 integers from the global memory to the shared memory. In Figure 11, each thread needs to process four substrings starting at position *tid*, *tid*+256, *tid*+512 and *tid*+768, where *tid* is the thread ID. Since date is moved from the global memory to the shared memory block by block, we have to move an additional length of input string to avoid the boundary detection problem. In this paper, we choose 128 integers (512 characters). Therefore, for an input stream of length *n* characters, the total number of characters read from the global memory is 1.5*n*, which is much smaller than the worst case of *mn*.

In addition, the procedure is very efficient because it satisfies the property of memory coalescing because the 32 threads of the same warp access consecutive 32 words (32-bit integer). Although the total number of memory read operation increases from $\sum_{i=1}^{n} l(i)$ to $\sum_{i=1}^{n} l(i)$ + *1.5n* where *1.5n* times of read from the global memory and $\sum_{i=1}^{n} l(i)$ read from the shared memory, the total memory access time is reduced. The reason is that the latency of accessing the shared memory is easily hidden by arithmetic operations especially when all threads in a thread block have 100% occupancy as our GPU kernel implementation does.



$$\text{Cost of global read} = \left(\frac{n}{4}\right) \times \frac{1}{256} \times (256+128) \text{ integer}$$

Figure 8: Job assignment.

```
__shared__ int s_input[256 + 128];
int tid = threadIdx.x ;
int gbid = blockIdx.y * gridDim.x + blockIdx.x ;
int start = gbid * 256 + tid;

s_input[tid] = d_input_string[start];
if ( tid < 128 )
    s_input[tid+256] = d_input_string[start + 256];
__syncthreads();
```

Figure 9: Thread block loads 384 integers from the global memory to shared memory.

4.2 Reducing latency of transition table lookup

As discussed in Section IV, the PFAC algorithm traverses a state machine by looking up two tables, PFAC_table and output_table, which are stored in the global memory. In other words, the PFAC algorithm has to access the global memory twice to traverse a state machine.

If the number of total states and match vectors is smaller than 65,535 which can be encoded as 16-bits format, we can merge the PFAC_table and the output_table into a transition table of 32-bits format. And then, we can retrieve the information of next state and match vector at the same time.

However, we still need $\sum_{i=1}^{n} l(i)$ times of queries to the transition table stored in the global memory. Because the queries of the transition table are intrinsically irregular due to the behavior of string matching, it is difficult for the queries from the concurrent threads to be memory coalesced.

Moreover, since each thread starts from the initial state, the first row of the state transition table is pre-loaded into the shared memory. The pre-loading is powerful and achieves 25% speedup in general. In addition, we bind the transition table to the texture memory of GPUs which has 6~8KB cache memory per SM in order to reduce bus traffic.

```
phi_s02s1[ tid ] = tex2D(PFAC_table, tid, 0) ;
__syncthreads();
unsigned char *s_char = (unsigned char *)s_input;
for(int j = 0 ; j < 4 ; j++ ){
    pos = tid + j * 256 ;
    char ch = s_char[pos++];
    matched_nextState = phi_s02s1[ch];
    state = matched_nextState & 0x0000FFFF ;
    if ( TRAP_STATE != state ){
        matched_nextState >>= 16 ;
        match_pattern = matched_nextState & 0x0000FFFF ;
        if (match_pattern > 0) match[j] = match_pattern;
        while (TRAP_STATE != state) {
            ch = s_char[pos++];
            matched_nextState = tex2D(PFAC_table, ch, state);
            state = matched_nextState & 0x0000FFFF ;
            matched_nextState >>= 16 ;
            match_pattern = matched_nextState & 0x0000FFFF ;
            if (match_pattern > 0) match = match_pattern;
        }
    }
}
// write result to global memory
start = gbid * BLOCKSIZE * 4 + tid ;
for (int j = 0 ; j < 4 ; j++, start += 256 )
   match_result[start] = match[j];
```

Figure 10: each thread processes 4 substrings starting at tid, tid+256, tid+512 and tid+768, respectively

4.3 Reducing output table accesses

Another useful method to reduce output table accesses is to reorder match vectors of final states so that final states are numbered smaller than other states. Suppose a state machine has *n* final states (patterns), then the match vectors of final states are numbered from 0 to *n*-1, and all other internal states including initial state are numbered from *n*. For example, consider the reordering of the match vectors of the four patterns, "AB", "ABG", "BEDE", and "ED". As shown in Figure 13, the output vectors of patterns, "AB", "ABG", "BEDE", and "ED", are numbered as 0, 1, 2, and 3, respectively. And then, the initial state is numbered as 4. Therefore, we read the output table only if the value of current state is less the value of initial state. In addition, because the accesses of output table have spatial locality, binding the output table to the texture memory is preferable.

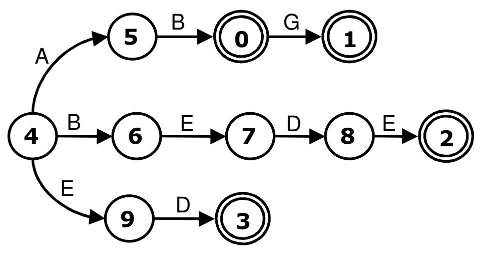


Figure 11. Reorder the output vectors of the PFAC state machine of patterns "AB", "ABG", "BEDE", and "ED"

4.4 Free of bank-conflict of the shared memory

At line 6 of Figure 11, each thread fetches an 8-bit character from the shared memory. The fetch operation may incur bank-conflict because every 4 threads access subsequent byte data in the same bank (32-bits word) of the shared memory. We take the advantage of the new Fermi [27] architecture which broadcasts the 32-bits word data to the requesting threads simultaneously. Therefore, the instruction at line 6 does not incur bank-conflict.

4.5 Coalescing write to the global memory

In the PFAC algorithm, each thread of the same warp writes matches results to the contiguous memory locations of the global memory, at line 26 of Figure 11. This behavior satisfies the property of memory coalescing.

4.6 Enhancing PCIe data transmission

It's well known that the effective bandwidth PCIe is about 2.22~5.74 GB/s which is much smaller than the bandwidth of devices (GPUs). Any CPU/GPU data transfer can dramatically reduce the total effective bandwidth. To enhance PCIe data transmission, CUDA provides a special instruction, cudaMallocHost(), to allocate host (CPU) memory that is page-locked and directly accessible to the device (GPU). Since the memory can be accessed directly by the device, the effective bandwidth of PCIe can be improved with much higher bandwidth than the virtual memory allocated by the instruction, *malloc*(). As shown in Figure 13, using page-locked memory, the bandwidth of PCIe can be accelerated to 6~6.41GB/s.

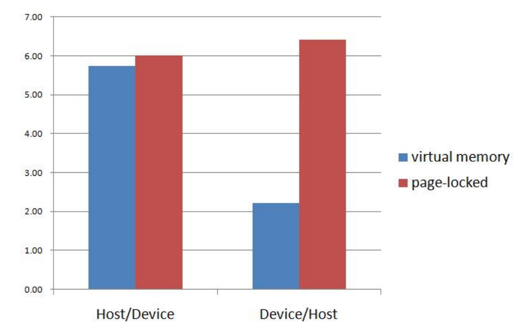


Figure 12: The bandwidth of PCIe using virtual memory and page-lock memory

VI. Experimental Results

In this section, we evaluate the performance of the AC algorithm on the Core^{TM} i7-950 and the PFAC algorithm on GTX480.

6.1 Experimental setup

The experimental setup has two machines: the *host* machine and the *device* machine. The host is equipped with an Intel$^{®}$ Core^{TM} i7-950 running the Linux X86_64 operating system with 12GB DDR3 memory on an ASUS P6T-SE motherboard while the device is equipped with an Nvidia$^{®}$ GeForce$^{®}$ GTX480 GPU in the same Core^{TM} i7 system with Nvidia driver version 260.19.29 and the CUDA 3.2 version.

As show in Table 1, the Intel$^{®}$ Core^{TM} i7-950 is a multi-threaded multicore Intel-Architecture processor featuring an out-of-order super-scalar microarchitecture. It offers quad cores with two-way hyper-threading on a chip operating at 3.07 GHz. The Core^{TM} i7-950 provides a peak external memory bandwidth of 25.6GB/s. In addition, it consists of 4-wide SIMD (Single Instruction Multiple Data) units to support SIMD instructions. Each core has 32KB L1 cache for both instructions and data, and 256KB L2 data cache. All four cores share 8MB L3 data cache.

On the other hand, the Nvidia$^{®}$ GeForce$^{®}$ GTX480 consists of 480 stream processors, organized in 15 multiprocessors, operating at 1.4 GHz with 1,536 MB GDDR5 memory. The GTX480 allows maximum 1,536 threads per multiprocessor. The GTX480 provides a peak memory bandwidth of 177.4 GB/s, 6.9x bigger than the Core^{TM} i7-950. Each multiprocessor has 16KB L1 cache and 48KB on-chip shared memory. All 15 multiprocessors share 768KB L2 cache. In addition, the GTX480 has 6~8KB cached texture memory which is optimized for 2D spatial locality. In other words, the texture memory enables those threads in the same warp to achieve the best performance when reading close memory addresses.

The test patterns are extracted from Snort V2.8 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 PFAC engine is tested using pure packets, and DEFCON packets. The pure packets which do not contain any patterns are used to evaluate the ideal throughput. The DEFCON packets which contain large amounts of real attack patterns are widely used to test commercial NIDS system. The size of test packets varies from 2 MB (Mega Bytes) to 192 MB.

Table 1: Specification of host and device

Specification	host	device
	Core i7 950	GTX480
Core frequency GHz	3.06	1.4
Number of Streaming Multiprocessor (SM)	4	15
# of Cores	4	480
# of Threads	8	1024
SP SIMD width	4	32
Bandwidth GB/s	25.6	177.4
L1 cache, KB	32KB per SM	16KB per SM
On-chip shared memory KB	N/A	48KB per SM
L2 cache, KB	256KB per SM	768KB for all SMs
L3 cache	8 MB for all SM	N/A

In order to compare the performance of the proposed algorithm, we re-implement three CPU versions and one GPU version as follows:

(1) AC_{CPU} : implementation of the AC algorithm on the Core^{TM} i7 using single thread.

(2) $DPAC_{OMP}$: implementation of the DPAC

algorithm described in Section III on the Core^{TM} i7 with OpenMP [28] library. The OpenMP library is a multi-threaded library used on multicore CPUs to achieve optimum performance. Moreover, because the Core^{TM} i7 processor is a quad-core CPU with 2-way hyper-threading, the best performance comes from running 8 threads on 4 cores. Therefore, the DPAC$_{OMP}$ divides input streams into 8 segments and creates 8 threads to each segment for pattern matching by traversing the AC state machine.

(3) *PFAC$_{OMP}$*: implementation of the PFAC algorithm on the Core^{TM} i7 with OpenMp library. Although PFAC$_{OMP}$ creates multiple threads whose number is equal to the input length, still maximum 8 threads can be activated on the Core^{TM} i7 at one time. To achieve the best performance, dynamic scheduling is applied to schedule the multiple threads.

(4) *PFAC$_{GPU}$*: implementation of the PFAC algorithm on the GTX480. In PFAC$_{GPU}$, the best performance is achieved by creating 256 threads per block which allows 48 warps, 6 thread blocks per multiprocessor to be activated simultaneously.

6.2 The performance metrics

The performance of GPUs is evaluated with respect to the following metrics:

(1) Raw data throughput of matching process

The metric is the data throughput without considering the data transfer time via PCIe and is calculated as follows:

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

where n denotes the length of an input stream of unit bytes and t denotes the GPU elapsed time of performing pattern matching.

(2) Effective bandwidth (BW) of algorithm

The effective bandwidth of the algorithm denotes the bandwidth of memory transactions of a launched kernel function. Our kernel function consists of three parts:

i. Moving input stream to the shared memory: *1.5*n* byte.
ii. Transition table lookup: $L=\sum_{i=1}^{n} l(i)$ integers.
iii. Writing match results to the global memory: n integers

Therefore, the effective bandwidth of the algorithm is defined as follows.

$$\text{Algorithm effective BW} = \frac{1.5n + 4L + 4n}{t}\ \text{(GB/s)}$$

where *1.5n* denotes the data transmission of moving input data from the global memory to the shared memory, *4L* denotes the total bytes of PFAC_table and output_table lookup, and *4n* denotes the total bytes of writing match results to the global memory. We would like to mention that the effective bandwidth of an algorithm is the best metric to measure the performance of a memory-bound algorithm.

(3) Instruction throughput of algorithm

Instruction throughput represents the number of instructions that can be executed in a unit of time without considering the memory latency. If the latency of the shared memory can be hidden by warp scheduling, the read/write operations of the shared memory can be regarded as arithmetic operations. The instruction throughput is defined as follows.

$$\text{Instruction throughput} = \frac{7L+9n}{t}\ \text{(GFLOPS)}$$

Note that we only consider the number of C-level instructions instead of assembly-level instructions because CUDA PTX assembly code is pseudo-assembly, not optimized. This metric can be used to estimate the efficiency of the proposed algorithm even though it's a memory-bound algorithm. The reason is that GPUs are a throughput-oriented architecture which use arithmetic operations to hide memory latency. In other words, to optimize a memory-bound approach, we can insert arithmetic operations to hide memory latency without increasing execution time. However, if the instruction throughput exceeds the maximum instruction throughput of a device, 1.377 TFLOPS of GTX480 for example, the execution time will increase inversely. On the other hand, if the instruction throughput of an algorithm is much smaller than the maximum instruction throughput, there exists an opportunity to add more arithmetic operations to improve the performance.

(4) Effective bandwidth of PCIe transmission

Performing GPU programs has four steps: (i) transferring data from host (CPU) memories to device (GPU) memories via PCIe (PCI Express), (ii) host programs launch GPU kernel functions, (iii) the GPU executes multiple threads in parallel, and (iv) transferring match results from device memory to host memory via PCIe.

The effective bandwidth of host-to-device memory transaction is defined as follows:

$$\text{Effective bandwidth}_{\text{host/device}} = \frac{n}{T_{\text{host/device}}}\ \text{(GB/s)}$$

where n denotes the total number of bytes of input stream and $T_{host/device}$ denotes the elapsed time of moving input data from host to device via PCIe interface.

The effective bandwidth of device-to-host memory transaction is defined as follows:

$$\text{Effective bandwidth}_{\text{device/host}} = \frac{4n}{T_{\text{device/host}}}\ \text{(GB/s)}$$

where 4n denotes the total number of bytes of match results and $T_{device/host}$ denotes the elapsed time of moving match results from device to host via PCIe interface.

(5) System throughput

Considering the bandwidth of PCIe transmission, the system throughput is defined as follows:

$$\text{System throughput}=\frac{8n}{T_{total}}\ (\text{Gbps})$$

where n denotes the length of input stream of unit bytes and T_{total} denotes the total elapsed time including (i) the latency of the host (CPU) to device (GPU) memory transaction via PCIe, (ii) the execution time of GPU function, and (iii) the latency of the device (GPU) to host (CPU) memory transaction via PCIe.

Tables 2 and 3 show the relative performance between the proposed GPU approach and the other three CPU approaches tested by pure packets and DEFCON packets, respectively. Columns 1 and 2 show the input size and the number of matched patterns. Columns 3, 4, 5, and 6 show the maximum raw data throughput of the four approaches, AC_{CPU}, $DPAC_{OMP}$, $PFAC_{OMP}$, and $PFAC_{GPU}$. In Table 2, for processing the pure packet of 192MB, $PFAC_{GPU}$ achieves maximum raw data throughput of 208.53 Gbps while AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$ achieves 0.95, 3.92, and 4.20 Gbps. Compared to AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$, $PFAC_{GPU}$ achieves 220x, 53x, and 50x times speedup, respectively. In Table 3, for processing the DEFCON packets of 192MB, $PFAC_{GPU}$ achieves maximum raw data throughput of 122.84 Gbps while AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$ achieves 0.82, 3.28, and 3.67 Gbps. Compared to AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$, $PFAC_{GPU}$ achieves 150x, 37x, and 33x times speedup, respectively. As shown in Figure 13, $PFAC_{GPU}$ significantly outperforms to the AC_{CPU}, $DPAC_{OMP}$, and $PFAC_{OMP}$ approaches. Note that the $PFAC_{OMP}$ has better performance than $DPAC_{OMP}$. We would like to mention that the reason causing the decreasing of raw data throughput comes from the impact of load unbalance. As we mention above, GPU is intrinsically a vector machine which groups 32 threads in the same warp. The load unbalance inside warps affects the performance of the PFAC.

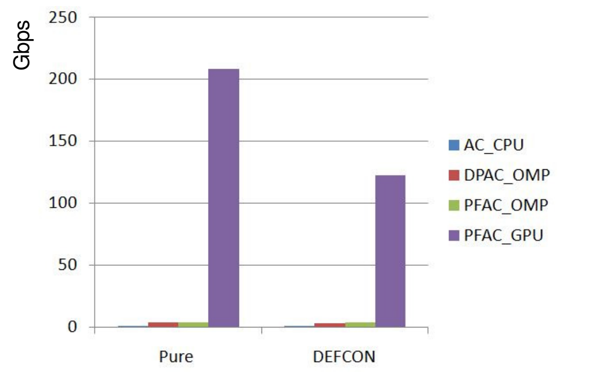


Figure 13: The raw data throughput of AC_{CPU}, $DPAC_{OMP}$, $PFAC_{OMP}$, and $PFAC_{GPU}$ over pure and DEFCON packets of 192MB

Furthermore, columns 7 and 8 show the algorithm throughput and the instruction throughput. We can find that the algorithm throughput of the proposed algorithm achieves up to 247.66 and 147.39 GB/s (Giga Bytes per second) for the pure and DEFCON packets of 192MB. On the other hand, the instruction throughput for processing the DEFCON packet of 192MB is 237.42 GFLOPS, much smaller than the peak instruction throughput of GTX480 (1,377 GFLOPS). The result indicates that we still have chance to add more arithmetic operations to improve the performance.

Moreover, column 9 shows the effective bandwidth of host-to-device (H2D) and device-to-host (D2H) memory transactions and column ten shows the system throughput. As shown in Table 2 and 3, the effective bandwidth is proportional to the input sizes and peaks at 6 GB/s for host-to-device memory transaction and 6.41 GB/s for device-to-host memory transaction. Because the PCIe interface only provides the effective bandwidth of 6~6.41 GB/s, the system throughput would be much smaller due to PCIe overheads. For processing the pure packet and DEFCON packet of 192MB, the $PFAC_{GPU}$ achieves system throughput of 9.65 Gbps and 9.35 Gbps, 10.2x and 11.4x times faster than AC_{CPU}, respectively. As shown in Figure 14, even considering the PCIe overheads, the $PFAC_{GPU}$ still has 2~2.5x times improvements compared to $DPAC_{OMP}$ and $PFAC_{OMP}$.

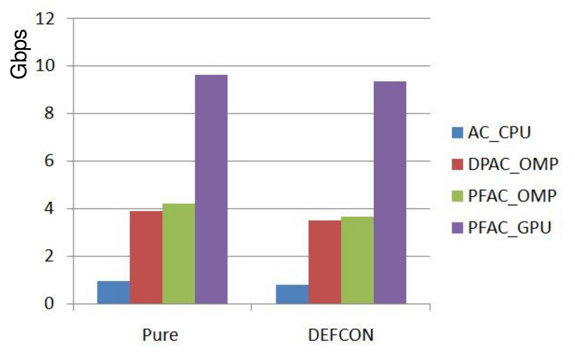


Figure 14: The system throughput of AC_{CPU}, $DPAC_{OMP}$, $PFAC_{OMP}$, and $PFAC_{GPU}$ over pure and DEFCON packets of 192MB

VII. CONCLUSIONS

In this paper, we have proposed a novel parallel algorithm to accelerate pattern matching by GPUs. The experimental results show that the proposed algorithm achieves significant speedup not only on GPUs but also on CPUs compared to the traditional AC algorithm.

Table 2: Throughput comparisons over pure packets of different sizes

Pure Packets		AC_{CPU}	$DPAC_{OMP}$	$PFAC_{OMP}$	$PFAC_{GPU}$				
Input Size (MB)	# of Patterns	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Algorithm Effective Bandwidth (GB/s)	Algorithm Instruction Throughput (GFLOPS)	H2D / D2H Effective Bandwidth (GB/s)	System Throughput (Gbps)
2 MB	0	0.95	3.76	4.08	143.40	169.27	285.09	5.67 / 6.37	9.28
4 MB	0	0.95	3.85	4.14	175.43	206.77	348.25	5.86 / 6.39	9.48
8 MB	0	0.95	3.89	4.17	192.85	229.90	387.21	5.92 / 6.40	9.57
16 MB	0	0.95	3.92	4.19	205.61	244.56	411.91	5.96 / 6.40	9.63
32 MB	0	0.95	3.93	4.20	190.89	221.63	373.27	6.01 / 6.41	9.61
64 MB	0	0.95	3.94	4.20	218.09	258.31	435.05	6.03 / 6.41	9.68
128 MB	0	0.95	3.94	4.21	219.14	260.23	438.29	6.01 / 6.41	9.67
192 MB	**0**	**0.95**	**3.92**	**4.20**	**208.53**	**247.66**	**417.12**	**6.00 / 6.41**	**9.65**
Ratio		**1**	**4.12**	**4.42**	**219.50**				**10**

Table 3: Throughput comparisons over DEFCON packets of different sizes

DEFCON Packets		AC_{CPU}	$DPAC_{OMP}$	$PFAC_{OMP}$	$PFAC_{GPU}$				
Input Size	# of Patterns	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Raw data Throughput (Gbps)	Algorithm Effective Bandwidth (GB/s)	Algorithm Instruction Throughput (GFLOPS)	H2D / D2H Effective Bandwidth (GB/s)	System Throughput (Gbps)
2 MB	28312	0.69	2.88	3.08	72.47	91.35	154.27	5.68 / 6.36	8.74
4 MB	58660	0.70	2.88	3.03	84.58	105.79	178.61	5.83 / 6.38	8.96
8 MB	107433	0.70	3.00	3.18	92.69	115.27	194.58	5.94 / 6.39	9.09
16 MB	215428	0.71	3.09	3.31	95.52	118.87	200.66	5.99 / 6.40	9.14
32 MB	458491	0.70	3.05	3.28	93.31	114.84	193.85	6.01 / 6.41	9.13
64 MB	840956	0.71	3.09	3.34	100.74	124.39	209.95	6.03 / 6.41	9.20
128 MB	933759	0.79	3.25	3.60	116.63	140.77	237.42	6.04 / 6.41	9.32
192 MB	**1025876**	**0.82**	**3.28**	**3.67**	**122.84**	**147.39**	**248.51**	**6.01 / 6.41**	**9.35**
Ratio		**1**	**4**	**4.48**	**149.80**				**11**

REFERENCES

[1] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. 9th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001, pp. 227-238.

[2] B.L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proc.10th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2002, pp. 111-120.

[3] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proc. 11th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2003, pp. 31–38.

[4] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," in *Proc. 12th Ann. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2004, pp. 249-257

[5] M. Aldwairi*, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," in *ACM SIGARCH Computer Architecture News*, 2005, pp. 99–107

[6] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in proc. 32nd Ann. Int. Symp. on Comp. Architecture, (ISCA), 2005, pp. 112-122

[7] H. J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems," in *20th Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2006.

[8] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering," in *Proc. of Symp. Architectures Netw. Commun. Syst. (ANCS)*, 2005, pp. 183-192

[9] Y. H. Cho and W. H. Mangione-Smith, "A Pattern Matching Co-processor for Network Security," in *Proc. 42nd Des. Autom. Conf. (DAC)*, 2005, pp. 234-239

[10] S. Kumar, S. Dharmapurikar, F. Yu, Patrick Crowley, and J. Turner, "Algorithm to accelerate multiple regular expressions matching for deep packet inspection," in ACM SIGCOMM, 2006

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

[12] N. Huang, H. Hung, S. Lai, Y. Chu and W. Tsai, "A GPU-based Multiple-pattern Matching Algorithm for Network Intrusion Detection Systems", International Conference on Advanced Information Networking and Applications – Workshops (AINAW) , Okinawa, March 25-28, 2008, pp. 62-67.

[13] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for Network Packet Signature Matching", IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston, MA, April 26-28, 2009, pp.175-184.

[14] 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.

[15] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *In Communications of the ACM*, 18(6):333–340, 1975.

[16] J. F. Peng, H. Chen, and S. H. Shi "The GPU-based string matching system in advanced AC algorithm", *International Conference on Computer and Information Technology (CIT2010)*, Bradford, UK, June 29 - July 1, 2010, pp. 1158-1163.

[17] M. Roesch. Snort- lightweight Intrusion Detection for networks. In Proceedings of LISA99, the 15th Systems Administration Conference, 1999.

[18] 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.

[19] C. S. Kouzinopoulos, and K. G. Margaritis, “String Matching on a multicore GPU using CUDA”, Panhellenic Conference on Informatics (PCI), Corfu, Sept.10-12, 2009, pp.14-18.

[20] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, “IP Routing Processing with Graphic Processors”, Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, March 8-12, 2010, pp.93-98.

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

[22] J. Yu and J. Li, “A Parallel NIDS Pattern Matching Engine and Its Implementation on Network Processor,” in Proc. the 2005 International Conference on Security and Management (SAM), 2005.

[23] C. V. Kopek, E. W. Fulp, and P. S. Wheeler, “Distributed Data Parallel Techniques for Content-matching Intrusion Detection Systems,” in Proc. IEEE Military Communications Conference (MILCOM), 2007, pp.1-7.

[24] G. Vasiliadis and S. Ioannidis, “GrAVity: A Massively Parallel Antivirus Engine,” in *Proc. 13th international conference on Recent advances in intrusion detection (RAID'10)*, Berlin, Heidelberg, 2010.

[25] O. Villa, D. Chavarria, and K. Maschhoff, “Input-independent, scalable and fast string matching on the Cray XMT,” in *Proc. 23nd IEEE Intl. Parallel & Distributed Processing Symp. (IPDPS’09)*, 2009.

[26] D. P. Scarpazza and G. F. Russell, “High-performance regular expression scanning on the Cell/B.E. processor,” in *Proc. 23rd international conference on Supercomputing (ICS’09)*, 2009.

[27] Fermi, Avaliable: http://www.nvidia.com/object/fermi_architecture.html

[28] OpenMP, Avaliable: http://openmp.org/wp/