2011 IEEE International Conferences on Internet of Things, and Cyber, Physical and Social Computing

GPEP: Graphics Processing Enhanced Pattern-Matching for High-Performance Deep Packet Inspection

Lucas John Vespa
Department of Computer Science
University of Illinois at Springfield
Springfield, IL, U.S.A.
Email: lvesp2@uis.edu

Ning Weng
Department of Electrical and Computer Engineering
Southern Illinois University Carbondale
Carbondale, IL, U.S.A.
Email: nweng@siu.edu

***Abstract*—Graphics processing units (GPU) can be used to accelerate deep packet inspection. However, the state transition tables used to implement deterministic finite automata are very large and must be stored in DRAM, which inhibits performance and may cause non-deterministic scanning rates. In this work we present GPEP, a GPU-based deep packet inspection engine. GPEP uses an optimized version of our pattern matching algorithm called P^{3}FSM, which has low operational complexity, but reduces the memory requirement such that the state tables can fit into the small on chip memories of a GPU. This allows GPEP to scan quickly and deterministically with no global memory accesses to state tables. We optimize our P^{3}FSM (Portable Predictive Pattern Matching Finite State Machine) algorithm for execution on SIMD devices and to exploit the parallelism of the VLIW arrangement of the GPU processing cores. We show that GPEP consistently achieves over 30 Gb/s deep packet inspection.**

I. Introduction

Deep packet inspection (DPI) is a key component in detecting network attacks in network intrusion detection systems [1], [2], [3], [4]. DPI scans every byte of incoming packet payloads using pattern matching algorithms to identify the presence of known attack patterns [5]. Due to the rapid increase in network attacks, current DPI cannot keep up with the multiple gigabit rates of modern networks. This lack of speed causes DPI to be a potential bottleneck of traffic gateways and security systems, which affects service availability and system vulnerability.

Methods to increase DPI speed by increasing packet stride [6], [7], [8] have been developed, but these methods are limited by pattern characteristics such as pattern length. The potential performance improvement using these methods is therefore limited.

Graphics processing units (GPU) [9] have tens to hundreds of processing cores and can be used to accelerate DPI to tens of gigabits per second. However, the SIMD configuration of GPU processing cores requires a simple pattern matching algorithm. Accordingly, deterministic finite automata (DFA) have been implemented in a GPU [10], [11], [12] due to the simplicity and determinism of DFA. Unfortunately, state transition tables require excessive memory and can require truncation of attack patterns, therefore acting as a prefilter and not a full verifier. Also, the state transition tables must be stored in large global memory on the GPU and therefore require one global memory access for every packet byte processed in the worst case.

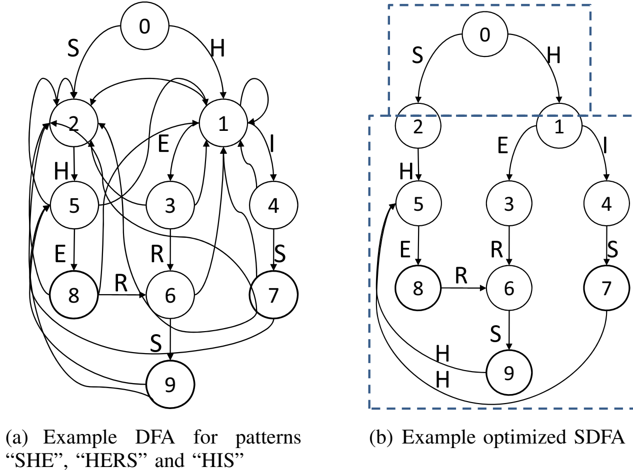
We have previously presented a pattern matching algorithm called P^{3}FSM [13], [14] which has lookup complexity similar to a state transition table, but reduces the memory by over 90%. P^{3}FSM is ideal for implementation in a GPU because the required memory tables can fit in the local memory of each processing core of the GPU. This allows for fast and deterministic operation regardless of packet content because there are no global memory accesses for DFA operation. In addition, the small memory requirement means that large pattern sets can be implemented in full, allowing the GPU to act as a full verifier rather than a prefilter.

In this work we present GPEP, a GPU-based DPI engine that utilizes our P^{3}FSM pattern matching algorithm. We optimize P^{3}FSM for SIMD operation by removing branches. We also optimize P^{3}FSM to increase the utilization of the 5-way VLIW processors on the GPU. We show that GPEP achieves a deep packet inspection throughput of over 30 Gb/s on a single ATI Radeon GPU [15]. We also present a performance model to aid in verifying our experimental results.

The remainder of this work is organized as follows. Section II gives background information on deep packet inspection and our P^{3}FSM pattern matching algorithm as well as how to optimize it for GPU processing. Section III presents the architecture and memory management of our GPU-based engine, GPEP. Section IV presents performance analysis. Other works related to accelerating DPI are discussed in Section V and the paper is concluded in Section VI.

†This work was performed while at Souther Illinois University.

978-0-7695-4580-6/11 $26.00 © 2011 IEEE
DOI 10.1109/iThings/CPSCom.2011.36



(a) Example DFA for patterns "SHE", "HERS" and "HIS"

(b) Example optimized SDFA

Figure 1. DFA example

II. Deep Packet Inspection

In this section we introduce deep packet inspection using deterministic finite automata (DFA). We begin by summarizing DFA and what is required to optimize DFA for use in multicore devices. We then introduce our P^{3}FSM algorithm and show how to efficiently optimize and encode a DFA for use in graphics processing units. We show the memory requirement of our P^{3}FSM algorithm, demonstrate how P^{3}FSM operates and optimize it for SIMD operation.

A. Deterministic Finite Automata

Deterministic finite automata (DFA) are used for multiple pattern matching in deep packet inspection. The performance of DFA is deterministic, regardless of the number and length of the patterns used to construct the DFA. An automata is constructed from a set of many known attack patterns. Figure 1(a) shows a sample DFA for the patterns "SHE", "HERS" and "HIS". Each state in the DFA has 256 labeled transitions, one for each character in the ASCII alphabet. A DFA is traditionally encoded in memory as a state transition table (STT). The current state and input character form the address for indexing the next state pointer in the STT.

B. DFA Optimized for Parallel Processing

The simple operation of a STT is ideal for the SIMD operation. Unfortunately, a STT requires tens of kilobytes per pattern. A state transition table, even for only a few patterns, cannot fit into the on-chip memory of most devices, especially that of multicore and multiprocessor devices. Therefore a STT must be stored in the global memory of these devices. Global memory accesses are slow and therefore slow down pattern matching operations. P^{3}FSM uses a novel memory encoding technique that reduces the memory requirement substantially while maintaining similar operational complexity to a STT. This allows the memory tables to fit into the local memory of any device.

Table I
Clustering

Cluster	Character	Group
C1	H S E R	G1 G5 G2 G7 G9 G3 G8 G6
C2	I	G4

Table II
Group Coding

Group	Char Sig	State Sig
G1	0 0	0 1
G5	0 0	1 0
G2	0 1	0 1
G7	0 1	1 0
G9	0 1	1 1
G3	1 0	0 1
G8	1 0	1 0
G6	1 1	0 1
G4	0	1

Table III
State Codes

State	Group	Code
S1	G3 G4	1 0 0 1 0 1
S2	G5	0 0 1 0 0 0
S3	G6	1 1 0 1 0 0
S4	G7	0 1 1 0 0 0
S5	G8 G4	1 0 1 0 0 1
S6	G9	0 1 1 1 0 0
S7	G5	0 0 1 0 0 0
S8		0 0 0 0 0 0
S9	G5	0 0 1 0 0 0

P^3FSM achieves these properties by only storing one entry in memory for each state in a DFA, rather than storing numerous DFA transitions as in typical memory based DFA implementations. The code for each state is derived such that it is predictive, in that, each state code contains information that denotes all of its possible next states. The FSM formed from these codes is able to isolate the appropriate next state very quickly due to the unique properties of these codes.

C. P^{3}FSM DFA Encoding

This section discusses how to optimize a DFA and then encode a DFA for our P^{3}FSM algorithm.

1) DFA Optimization: Before encoding a DFA into memory state tables, we optimize the DFA with an optimization we call split-DFA. This optimization splits the DFA transitions into primary and secondary blocks at the first level of the DFA. All incoming transitions to the primary block are removed from the DFA. An example split-DFA is shown in Figure 1(b). The two blocks are encoded into two separate memory tables. If a transition is not present in the secondary block table, then the primary block table acts as a default transition lookup for the current input character.

2) Deriving State Codes: The following example illustrates the derivation of state codes for the DFA in Figure 1(b). The first step in deriving the state codes is to group all the states in the DFA that have the same next state into a

group, along with the character required for said transitions. The result is one group for each state (i.e., G1[S0][H], G2[S0][S], G3[S1][E], G4[S1, S5][I], G5[S2, S7, S9][H], G6[S3][R], G7[S4][S], G8[S5][E], G9[S6][S]), where Gi is the group representing states that transition to state i.

The second step is to combine these groups together to form clusters. Groups with the same character are combined into a cluster. Next, the number of clusters are reduced by merging all the clusters that do not have common states to form one cluster. The clustering result obtained is shown in Table I. As seen in this table, characters 'H', 'S', 'E', 'R' do not have common states and hence, can be placed in one cluster.

The third step in generating state codes is encoding the groups. The encoding has two parts, the character signature and state signature. The character signature identifies the character required for transitions to a state. Characters are assigned a unique signature beginning with 0. This signature remains unchanged for the groups with same character. The state signature identifies the next state and is assigned beginning with 1. The resulting group codes are shown in Table II. For example, from this table we see that the groups G1 and G5 have the same character signature 00, this is because both the groups have character 'H'. However, their state signatures are 01 and 10 respectively because G1 and G5 represent two different states.

Finally, a state code for each state is obtained by concatenating the group codes for the groups that a state is a member of. The group codes are formed into a state code by being placed in the position of the cluster that the group is a member of. Table III shows the state codes for all states. As an example, state S5 is a member of groups G8 and G4 and the codes for G8 and G4 are 1010 and 01 respectively. The state code for S5 is formed by placing the code for G8 in the position of cluster C1 and the code for G4 in the position of cluster C2. The state code for S5 is therefore the concatenation of 1010 and 01, to yield 101001.

D. P^3FSM Operational Tables

This section demonstrates how the P^3FSM operational tables are formed and then gives an example of the algorithm's operation.

1) Character/Cluster Table: The character/cluster table (CC) is created to contain several pieces of information. Firstly, the character signature for each character is stored. Secondly, the cluster that contains all the states associated with that character. Thirdly, each character is given an offset value. The offset starts at 0 for the first character and is incremented for each character by the number of states with the previous character. Finally, each character is assigned an index. If a valid transition is not produced during DFA operation, the index automatically becomes the next state index. Table IV shows the final character/cluster table (CC) for this example.

Table IV
CHARACTER/CLUSTER TABLE (CC)

Character	Signature	Cluster	Offset	Index
H	0 0	1	0	1
S	0 1	1	2	3
E	1 0	1	5	0
R	1 1	1	7	0
I	0	2	8	0

Table V
CODE TABLE (CODE)

Index	State_Code	State
1	1 0 0 1 0 1	S1
2	1 0 1 0 0 1	S5
3	0 0 1 0 0 0	S2
4	0 0 1 0 0 0	S7
5	0 0 1 0 0 0	S9
6	1 1 0 1 0 0	S3
7	0 0 0 0 0 0	S8
8	0 1 1 1 0 0	S6
9	0 1 1 0 0 0	S4

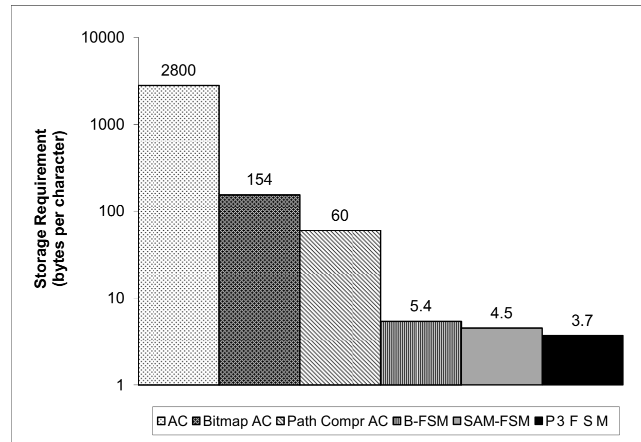


Figure 3. P^3FSM memory requirement comparison with related work

2) Code Table: The code table (code) consists solely of the state codes placed in the correct order. The index position for each state code in the code table can be calculated by Equation 1, by adding the state signature (S_{sig}) to the character offset (Ch_{offset}) for any given state. Table V shows the final code table (code) for this example.

$$S_{index} = Ch_{offset} + S_{sig} \quad (1)$$

3) Example Operation: Figure 2 demonstrates how P^3FSM uses the current state and current input character to find the next state. In this example the current state is 7 and the current input character is 'H'. P^3FSM requires two steps to find that the next state is state 5.

E. P^3FSM Memory Efficiency

The memory efficiency of P^3FSM allows GPEP to fit the state tables into the local memory of the stream cores on a

Character/Cluster Table				
Char	Signature	Cluster	Offset	Index
H	0 0	1	0	1
S	0 1	1	2	3
…	…	…	…	…

Code Table		
Index	Code	State
1	1 0 0 1 0 1	1
2	1 0 1 0 0 1	5
3	0 0 1 0 0 0	2
4	0 0 1 0 0 0	7
…	…	…

Current state = 7, Input Character = H

1. Check cluster 1 of state code 7 for character signature of H 00 = 00
2. Compute next state: state signature + character offset
   10 + 0 = 2 -> next state index = 2, next state = state 5

Figure 2. Example operation of P[3]FSM engine. Transition from state 7 to 5 on input “H” from Figure 1(a).

Table VI
MEMORY REQUIREMENTS (KILOBYTES) OF P[3]FSM WITH INCREASING NUMBER OF ATTACK PATTERNS.

# Patterns	# Chars	STT (KB)	P[3]FSM (KB)	Bytes/Char
100	1291	397	3.04	2.41
200	2129	600	5.85	2.82
300	4313	1258	14.26	3.38
400	6722	2116	23.79	3.62
500	7637	2304	27.04	3.63
1000	13525	4072	49.00	3.71

GPU. This allows for faster and more deterministic operation of the GPU-based deep packet inspection. Table VI shows the memory requirements for a state transition table (STT) and for P[3]FSM, with and increasing number of patterns. The equation used to compute the storage requirement in STT implementation is $Q \cdot \lceil log_2 Q \rceil \cdot 2^8$, where Q is the total number of states of the DFA. This formula assumes that the binary encoding scheme is used in the DFA implementation. Thus, each state code needs $\lceil log_2 Q \rceil$ bits. The memory size required by P[3]FSM is calculated by $Q \cdot (L + \lceil log_2 P \rceil)$, where L is the length of state code. P is the number of patterns to be detected. The $\lceil log_2 P \rceil$ term in the above expression represents the additional tag bits used to indicate which patterns are matched.

Table VI demonstrates that P[3]FSM reduces the memory requirement from a STT by many times, but still scales with an increasing number of patterns. Table VI also shows the bytes per pattern character required for P[3]FSM, which is about 3.7 bytes for 1000 patterns. Figure 3 compares storage efficiency in terms of bytes per character with AC [16], Bitmap AC [1], Path compressed AC [1] and B-FSM [17]. This figure shows that P[3]FSM requires only about 3.7 bytes per pattern character.

F. P[3]FSM Flow Control

Each kernel instance on the GPU is called a work item. If two work items within a compute unit diverge execution paths, then the paths must be executed serially. This serialization of instructions severely decreases performance. In order to mitigate this performance issue, we rewrite the P[3]FSM algorithm as simple equations which uses logical operations to replace the branch operations. In other words, we flatten the algorithm so that it will execute the same instructions for all work items, regardless of packet content. The following are the final equations where ns' is an intermediate value calculated before deriving the next state (ns). ss stands for state signature, cs stands for character signature, p is the current packet character.

$$ns' = (cc[p].offset + code[p.cluster][ns].ss) * \\ !(!(code[p.cluster][ns].ss)) * \\ !(code[code[p.cluster]][ns].cs - p.cs) * !(!ns)$$

$$ns = ns' + !ns' * cc[p].index$$

III. GPEP ARCHITECTURE

This section presents the GPEP architecture. The functionality of GPEP is split between the CPU and the GPU.

This is illustrated by Figure 4.

A. CPU

The CPU host has several responsibilities as shown in Figure 4. The host creates and optimizes the DFA according to the encoding techniques previously discussed. The host then transfers the resulting tables to the memory of the GPU. The tables are store in the local memory of the GPU compute units. The host also maintains the current packet buffer which is mapped to the global memory of the GPU. Finally, the host reads the matching results buffer on the GPU and reports any attack pattern matches as a potential attack.

B. GPU

As shown in Figure 4, the P^{3}FSM kernel runs on each stream core in the GPU. The following are specifics about the functionality of the kernel as well as GPU memory management.

1) Kernel: In order to more efficiently utilize the five element VLIW processors, we thread multiple, non-adjacent bytes simultaneously per work item. This increases the utilization of the individual processing elements in each stream core. For example, with only one copy of the P^{3}FSM code per work item, the processor utilization is around 20%. This means that only about one of the five available processing elements is being used on average. If we increase the number of copies to four, the utilization increases to over 50%. This means that on average about 2.5 processing elements are being used simultaneously. This roughly translates to about 2.5 bytes processed in parallel by a single work item.

Most GPUs have the ability to run more work items than available stream cores. The GPU will trade off active work items in order to help hide the latency caused by memory accesses. The ATI Radeon HD 5970 has 640 stream cores so this is the minimum number of work items that we will run on the GPU.

2) Memory: The memory tables necessary for the P^{3}FSM kernel operation are stored in the local data store (LDS) of each compute unit, and the private memory of each stream core. Storing these tables locally is possible because of the reduced memory footprint of the P^{3}FSM algorithm. Local access to the state tables allows for faster performance. Packet data is stored in a memory buffer on the CPU host. The map_buffer OpenCL command creates a mapping between this host buffer and a buffer in the GPU global memory. This mapping is used for DMA between the GPU memory and host memory. This method is faster that using a write_buffer command to explicitly write packet data from the host the the GPU global memory. The kernel requests 16 byte vectors from the global packet buffer. Fetching 16 byte vectors most efficiently utilizes the memory fetch unit, which can access 128 bits at a time.

Table VII
PARAMETERS USED TO MODEL PERFORMANCE

Parameter	Description
p	processing elements per stream core
f	clock frequency
m	DRAM access time
M_W	DRAM channel width (bytes)
I	number of instructions
c	cycles per instruction
s	number of stream cores on device
w	number of work items
L	number of local memory accesses
I_L	instructions per local memory access
u	processor utilization
n	threads per work-item

C. Performance Model

In this section we present our performance model which can be used to estimate the performance of GPEP with different GPU configurations. The parameters of the model are described in Table VII. The throughput (T) can be estimated as follows, where t_I and t_M are the instruction and memory time to process one byte and s is the number of stream cores:

$$T = \frac{s}{t_I + t_M} \tag{2}$$

Instruction time (t_I) is calculated as follows:

$$t_I = \frac{I \cdot c}{f \cdot u \cdot p} \tag{3}$$

given the number of instructions (I), the clock frequency (f), processor utilization (u) and number of processing elements in a stream core (p). The memory time (t_M) can be calculated as follows:

$$t_M = \frac{m}{M_W} - \frac{m \cdot w \cdot u}{M_W \cdot s \cdot I} \tag{4}$$

using the number of work items (w), memory access time (m) and channel width (M_W). In order to analytically estimate the processor utilization, we use the following equation for utilization which comes from [18], where n is the number of threads per work-item and L is the number of local memory accesses:

$$u = \frac{n}{n + \frac{L}{\lfloor \frac{p}{I_L} \rfloor}} \tag{5}$$

Later, we compare our model to our experimental results.

IV. PERFORMANCE ANALYSIS

In this section we evaluate the performance of GPEP. We begin by describing our experiment setup. We then evaluate GPEP in detail including our kernel optimizations.

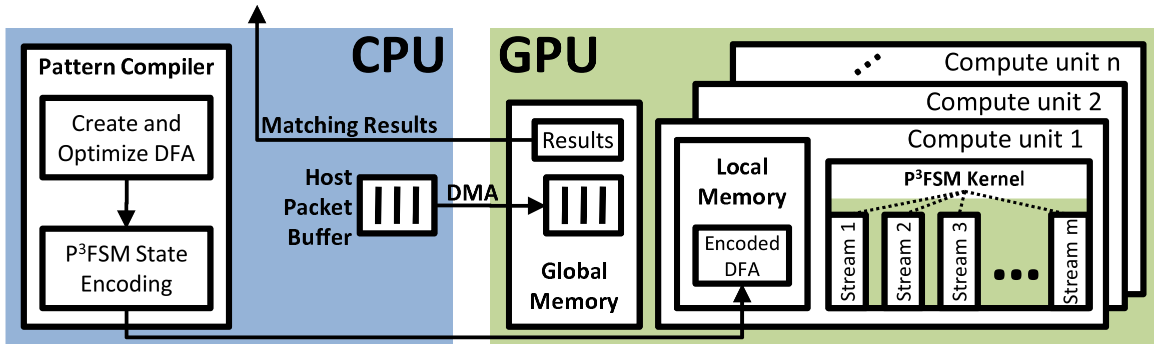


Figure 4. GPEP system architecture. The CPU creates the DFA memory tables from the attack pattern database and transfers incoming packets to the GPU. Deep packet inspection is performed by the GPU where each stream core processes a separate packet and any matches are reported back to the CPU.

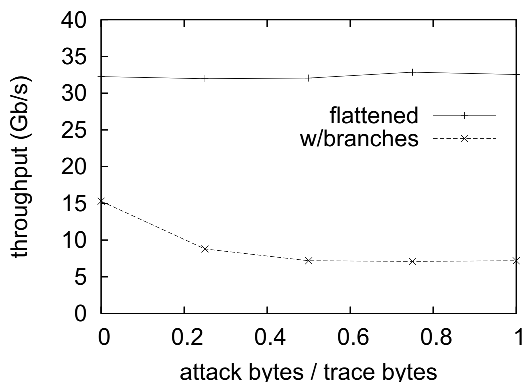


Figure 5. Kernel branch optimization

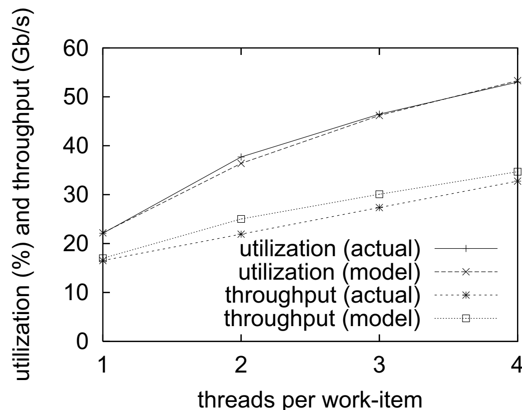


Figure 6. Kernel processor utilization and throughput with increasing number of threads per work-item

A. Experiment Setup

In our experiments we implement GPEP on an ATI Radeon HD 5970, or Hemlock GPU. The Hemlock has 640 stream cores and 2GB of DDR memory. Our host system contains an Intel Pentium G6950 processor running at 2.8 GHz and 4GB of DDR3 memory. The GPU and host interconnect via a PCIe 2.0 x16 bus. GPEP is written using Open Computing Language (OpenCL) [19] which abstracts the programming of various parallel computing devices. Using OpenCL allows GPEP to be portable amongst most newer graphics processing units. The data processed by GPEP comes from network trace files[20].

B. Kernel Optimization

In this section we evaluate the performance of GPEP using different kernel optimization techniques. We examine the performance of the kernel with and without branches. We also evaluate the processor utilization of the kernel.

1) Kernel Path: In this section we evaluate the throughput of GPEP using trace files that have varying numbers of attack patterns. We use both our original P^3FSM kernel with branches, as well as the flattened kernel without branches. Figure 5 shows the throughput of GPEP as more attack patterns are added to the trace files. The throughput of the flattened algorithm is not affected by the presence of attack patterns in the trace files. The original algorithm with branches suffers a performance decrease when more attack patterns are present. This is because the attack patterns cause an increased instruction path divergence between the work items.

Figure 5 also demonstrates that the flattened algorithm is significantly faster than the algorithm with branches. This is because if just one work item executes a different path than the other work items, all work items must wait for said path to execute. This severally decreases kernel performance.

2) Kernel Processor Utilization: In this section we evaluate how threading multiple non-adjacent bytes per work item increases processor utilization and throughput. According to our performance model, processor utilization increases by threading multiple bytes. This is shown in Figure 6 by the utilization(model) line. We tested different numbers of threads to confirm this experimentally as shown by the utilization(actual) line. Figure 6 also shows how multi-threading affects throughput in our model and actual

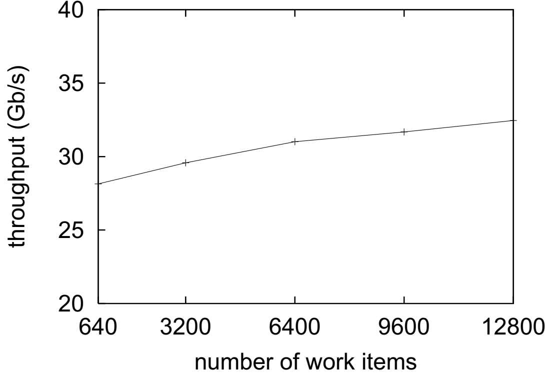


Figure 7. Performance versus number of work items

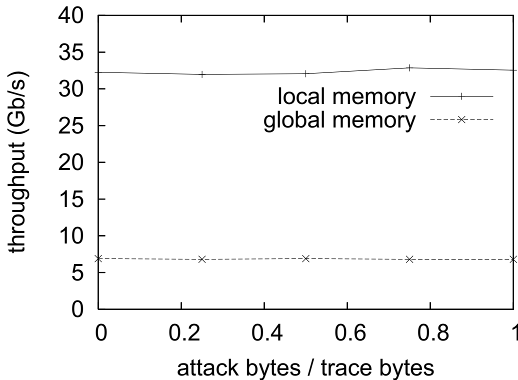


Figure 8. Performance for globally and locally stored state tables

experiments. Increasing the number of threads increases throughput because of the increase in processor utilization.

C. Global Optimization

In this section we evaluate the performance of GPEP using different global design optimizations. First we evaluate the performance of GPEP using a varying number of work items. We also evaluate GPEP by using global memory and local memory to store the state tables.

1) Work Item Optimization: In this section we evaluate the throughput of GPEP using different numbers of work items. Each stream core can execute more than one work item. Only one work item is active at any given point in time, but work items are intermittently executed. Using multiple work items per stream core helps to hide the overhead of memory accesses, by executing one work item while another waits for a memory access. Figure 7 shows the throughput of GPEP verses the number of work items. A small increase in throughput occurs when the number of work items increases.

2) Memory Optimization: This section evaluates the difference in throughput when using local memory verses global memory to store the P^3FSM state tables. Figure 8 shows that the throughput of GPEP is higher when the state tables are stored in local memory. This is because local memory accesses are faster than global accesses. This figure also shows that the throughput of GPEP is deterministic regardless of packet content for both local and global memory. Normally, storing the state tables in global memory would result in unpredictable throughput. However, the global memory accesses on the Hemlock GPU are not cached. Therefore, each state table access requires the same amount of time. Figure 8 also shows that the throughput of GPEP when using local memory is consistently over 30 Gb/s.

V. Related Work

This section discusses work related to accelerating deep packet inspection. The two methods to accelerate deep packet inspection are intra-stream parallelism and inter-stream parallelism. Intra-stream parallelism scans multiple bytes of a packet simultaneously whereas inter-stream parallelism scans multiple packets simultaneously using multiple copies of the pattern matching engine.

Methods have been presented to exploit intra-stream parallelism to increase DPI performance. Wu and Manber [21] and derivatives [6] have produced multiple-pattern, multiple-stride average case algorithms. The stride of these algorithms is heavily based on pattern characteristics and may require sequential pattern comparison. This limits the potential speedup of these algorithms. Brodie et al [22] increases throughput by allowing multiple DFA transitions to be traversed simultaneously. This system uses a specially designed hardware approach and is therefore limited in its implementation possibilities.

Hua et al [7] introduces a variable stride DFA (VS-DFA) which partitions patterns into variable size blocks using a fingerprinting scheme. These blocks are used to construct a multiple byte striding DFA. The same fingerprinting scheme is also used as a preprocessing step on the input source such as incoming packets. This guarantees that the correct size block of characters is fed to the VS-DFA. This preprocessing requires hashing of every byte of the packet before the input is given to the VS-DFA. The VS-DFA operation and the fingerprinting operation must be performed in parallel, again requiring special hardware.

Methods have been presented to exploit inter-stream parallelism to increase DPI performance. Commercial content inspection products use specialized hardware to accelerate pattern matching. Commercial chips such as the LSI Tarari T2000 series [23], the Cavium Networks CN1700 series [24] and the Netlogic NLS2008 [25] are advertised to achieve content inspection speeds of multiple Gb/s. Unfortunately, these are specialized hardware chips and therefore the implementation platforms are very limited.

Graphics processing units (GPU) have been used to exploit inter-stream parallelism. Vasiliadis et al [10], [11] have implemented deterministic finite automata (DFA) in

a GPU. Unfortunately, the state transition tables have a large memory requirement, which has to be stored in global memory and resulted in slow performance. Our work uses an optimized algorithm called P^3FSM, therefore it can therefore use local memory to store the pattern matching tables which improves performance.

VI. Conclusion

Deep packet inspection (DPI) is important for detecting the presence of an attacker. Accelerating DPI by implementing DFA on graphics processing units is one way to increase the effectiveness of DPI. Unfortunately, memory-based DFA cannot fit into local memory of any devices. Therefore, in this paper we have presented GPEP, a GPU-based pattern matching engine which exploits the low-memory low-complexity of our P^3FSM pattern matching algorithm, to address the problem of implementing pattern sets in GPU on-chip memory. In addition, we demonstrate how to optimize our algorithm for SIMD and VLIW systems. We experimentally evaluate GPEP using an ATI Radeon HD 5970 GPU and show that GPEP achieves a sustained throughput of over 30 Gb/s. The simple design of GPEP will allow for deep packet inspection acceleration on many different GPU platforms.

References

[1] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection." in *Proc. of the IEEE Infocom Conference*, 2004, pp. 333–340.

[2] D. Denning, "An intrusion–detection model," *IEEE Transactions on Software Engineering*, pp. 222–232, Feb. 1987.

[3] M. Roesch, "Snort – lightweight intrusion detection for networks." in *Proc. of the 13th Systems Administration Conference*, 1999.

[4] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, pp. 2435–2463, 1999.

[5] *Snort Rule Database*, http://www.snort.org/snort-rules.

[6] Y. D. Hong, X. Ke, and C. Yong, "An improved wu-manber multiple patterns matching algorithm," in *25th IEEE International Performance, Computing, and Communications Conference.*, 2006, pp. 680–686.

[7] N. Hua, H. Song, and T. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," in *IEEE INFOCOM 2009*, April 2009, pp. 415–423.

[8] L. Vespa, N. Weng, and R. Ramaswamy, "Ms-dfa: Multiple-stride pattern matching for scalable deep packet inspection," *The Computer Journal*, pp. 285–303, December 2010.

[9] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *Proceedings of the ACM SIGCOMM conference*, 2010, pp. 195–206.

[10] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, 2008, pp. 116–134.

[11] G. Vasiliadis and S. Ioannidis, "Gravity: a massively parallel antivirus engine," in *Proceedings of the 13th international conference on Recent advances in intrusion detection*, 2010, pp. 79–96.

[12] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating gpus for network packet signature matching," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2009*, 2009, pp. 175 –184.

[13] L. Vespa, M. Mathew, and N. Weng, "P3fsm: Portable predictive pattern matching finite state machine," in *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Boston, MA, USA, 2009, pp. 219–222.

[14] L. Vespa and N. Weng, "Deterministic finite automata characterization and optimization for scalable pattern matching," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 1, 2011.

[15] *ATI Radeon 5970 GPU, note=http://www.amd.com/us/products /desktop/graphics/ati-radeon-hd-5000/hd-5970/Pages/ati-radeon-hd-5970-overview.aspx year= 2011*.

[16] A. Aho and M. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, 1975.

[17] J. van Lunteren, "High-performance pattern-matching for intrusion detection," *Proceeding of 25th IEEE International Conference on Computer Communications*, pp. 1–13, 2006.

[18] T. Wolf and M. Franklin, "Performance models for network processor design," *IEEE Transactions on Parallel and Distributed Systems*, pp. 548–561, 2006.

[19] *OpenCL (Open Computing Language*, http://www.khronos.org/opencl.

[20] S. Shanbhag and T. Wolf, "Anombench: A benchmark for volume-based internet anomaly detection," in *IEEE Global Telecommunications Conference*, December 2009, pp. 1–6.

[21] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," Tech. Rep., 1994.

[22] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *In SIGARCH Computer Architecture News*, pp. 191–202, 2006.

[23] *LSI Tarari T2000*, LSI Corporation, 2011.

[24] *Cavium NITROX CN17XX*, Cavium Networks, 2011.

[25] *Netlogic NLS2008 NETL7*, Netlogic Microsystems, 2011.