SWM: Simplified Wu-Manber for GPU-based Deep Packet Inspection

Lucas Vespa
Department of Computer Science
University of Illinois at Springfield
lvesp2@uis.edu

Ning Weng
Department of Electrical and Computer Engineering
Southern Illinois University Carbondale
nweng@siu.edu

***Abstract*—Graphics processing units (GPU) have potential to speed up deep packet inspection (DPI) by processing many packets in parallel. However, popular methods of DPI such as deterministic finite automata are limited because they are single stride. Alternatively, the complexity of multiple stride methods is not appropriate for the SIMD operation of a GPU. In this work we present SWM, a simplified, multiple stride, Wu-Manber like algorithm for GPU-based deep packet inspection. SWM uses a novel method to group patterns such that the shift tables are simplified and therefore appropriate for SIMD operation. This novel grouping of patterns has many benefits including eliminating the need for hashing, allowing processing on non-fixed pattern lengths, eliminating sequential pattern comparison and allowing shift tables to fit into the small on-chip memories of GPU stream cores. We show that SWM achieves 2 Gb/s deep packet inspection even on a single GPU with only 32 stream cores. We expect that this will increase proportionally with additional stream cores which number in the hundreds to thousands on higher end GPUs.**

I. Introduction

Throughput requirements continue to increase for network applications and services. Deep packet inspection (DPI) has the most strenuous throughput requirement and is a key component of network intrusion detection systems [1], [2], [3], [4], [5]. DPI scans payloads for the presence of known attack patterns [6]. Increasing DPI speed can alleviate the bottleneck behavior that current payload search systems impose.

Graphics processing units (GPU) [7] have potential to speed up DPI by processing many packets in parallel. However, the SIMD configuration of GPU processing cores require simplicity for efficient utilization and fast kernel execution. Although deterministic finite automata (DFA) have been implemented in a GPU [8], [9], [10] due to their simplicity of operation, state transition tables require excessive memory and one global memory access for every packet byte processed in the worst case. Therefore, reduced memory algorithms [11], [12] have been used to allow DFA to be efficiently implemented in a GPU [13], however, DFA algorithms are single stride, which limits their speedup capability.

Methods to increase stride [14], [15], [16], [17] have been developed, however, the complexity of these algorithms is not appropriate for the SIMD operation of a GPU. For example, in the Wu-Manber [14] algorithm, when a substring that is a suffix of multiple patterns is encountered in a packet, multiple patterns must be compared to the packet text through hashing. This causes great divergence among processors in an SIMD arrangement, and thus a significant performance degradation.

In this work we present a simplified Wu-Manber like algorithm called SWM, which uses a novel method to group patterns such that the shift tables are simplified and the algorithm becomes appropriate for SIMD operation. Specifically, our grouping method simplifies the algorithm resulting in the following properties. Multiple pattern comparisons never occur, allowing for SIMD operation without path divergence between stream cores. This also allows for the use of shift tables which include the entire length of all patterns, rather than truncating the patterns when creating shift tables. Our pattern groupings also allow the use of 2-byte substrings for the shift tables which creates two other benefits. First, the shift tables are very small and can be stored directly in the local memories of the GPU stream cores which improves performance and determinism. Second, 2-byte substrings can be direct indexed which removes the need to hash the packet text. This also improves performance by removing the hash calculation entirely.

We further optimize SWM for the VLIW arrangement of stream processors and efficient access of the global memory packet buffer. We implement SWM in an ATI Radeon GPU [18], and show that, even on a low end GPU, SWM can achieve 2 Gb/s deep packet inspection.

The remainder of this work is organized as follows. Section II discusses the SWM algorithm. Section III discusses the architecture of the GPU system. Performance analysis and experimental results are presented in Section IV and related work is covered in Section V. The paper is concluded in Section VI.

II. SWM Algorithm

This section begins by discussing the construction of shift tables and operation of the Wu-Manber algorithm. It continues by discussing our novel method for grouping patterns, shift table construction and SWM algorithm operation. It concludes by presenting some optimizations required for efficient GPU execution.

A. Multiple Stride Basics (Wu-Manber)

Given a list of patterns P, Wu-Manber constructs a shift table which stores a shift value (in bytes) for all B-byte substrings in the first m bytes of each pattern $p \in P$. Each pattern therefore

has one shift value for $m - B + 1$ substrings. The shift value for substring s, is the position of s from the end of the pattern, subtracted from m. Here is an example where m = 5 and b = 2. In the pattern 'HELLO', the substring 'HE' occurs 2 bytes into the pattern so the shift value for 'HE' is $5 - 2 = 3$, meaning that if 'HE' is encountered in a packet, we can shift forward 3 bytes. In the pattern 'HELLO', the substring 'LO' occurs 5 bytes into the pattern so the shift value for 'LO' is $5 - 5 = 0$, meaning that, if 'LO' is encountered in a packet we cannot shift forward because we could potentially pass over the string 'HELLO'. Shift table entries are created for all B-byte substrings, including those that do not exist in any pattern. The stride value for substrings that do not exist in a pattern is $m - B + 1$.

Shift table operation begins by examining B-bytes of a packet starting at offset $m-B+1$. This B-byte value is looked up in the shift table. If the shift value found is non-zero, then we simply stride in the packet by this shift value and examine the B-bytes at this next location. If the shift value is zero then the packet text must be compared to any patterns that share this B-byte substring as a suffix. This could potentially be many patterns that need sequential comparison. This is the most time consuming part of the algorithm. Wu-Manber uses several methods to help with this problem but none are appropriate for a GPU-based multiple stride algorithm. The following are the methods used by Wu-Manber:

- Using a larger value for B helps reduce the number of patterns that share suffixes, but this also requires more memory for the shift tables
- The shift tables for larger values of B utilize a hash table to reduce memory, but this requires calculating a hash value for each B-bytes examined in the packet. Also, this reduces the average stride because substrings that hash to the same value must store the minimum stride value
- Hashing is used to reduce the time to compare many patterns that share suffixes

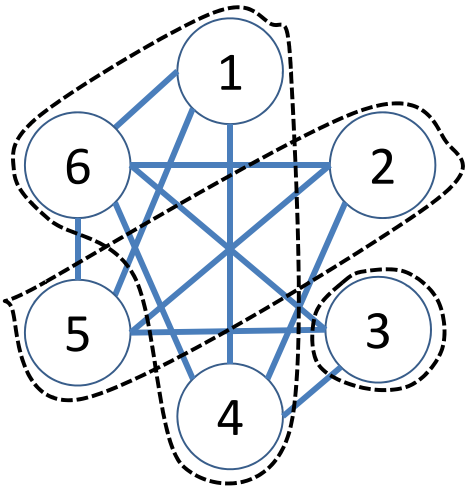
The goal of SWM is therefore to avoid these methods and produce a simpler algorithm with the following goals:

- Avoid using larger values for B so that the shift table size is small enough for GPU stream core caches
- Use a direct indexed lookup for each B-bytes to avoid the overhead of hashing and divergence between SIMD processors
- Avoid hashing when comparing patterns to the packet text in order to remove divergence between SIMD processors
- Avoid sequential pattern comparison to the packet text in order to remove divergence between SIMD processors
- Create shift tables with full patterns rather than the first m bytes of each pattern

B. Overview of SWM

All of the aforementioned properties can be achieved by circumventing the occurrence of patterns that share suffixes. If all patterns have a unique suffix, then any time that a stride of zero occurs, the current B-bytes are known to belong to only one specific pattern. If no two patterns share the same 2-byte suffix, then B can be chosen to be two bytes and the following properties apply:

(a) Extracting pattern suffixes



(b) Compatibility graph

Fig. 1. Example of creating a compatibility graph for patterns 'PROGRAMMER', 'ACCELEROMETER', 'FOLKSINGER', 'GINGERBREAD', 'WIDESPREAD', 'INSPECTION'. Each pattern has a vertex and vertexes are joined if the pattern prefixes are different. Maximal cliques are chosen from the graph to form pattern groups.

- Shift tables for B = 2 are small enough for GPU stream core local memories
- Shift tables for B = 2 can be direct indexed
- Sequential pattern comparison and hashing are not needed to compare patterns to the packet for shift values of zero because there is a one-to-one relationship between patterns and suffixes
- Shift tables can be created for full pattern lengths because the length of the pattern associated with each shift value of zero is known

In order to remove the occurrence of patterns that share suffixes, we must group patterns such that no two patterns in a group share the same two byte suffix. Specifically, we must find the minimal number of groups such that no two patterns in a group share a suffix. This grouping takes advantage of the parallel processing capability of graphics processing units. Each stream core in a GPU can processing one group of

patterns. Also, in a modern GPU, even though multiple stream cores must be used to process the full set of patterns, there are still enough stream cores to replicate these groups and process the full pattern set for many packets in parallel.

C. SWM Pattern Grouping

We create a compatibility graph to group patterns such that no two patterns in the same group share a suffix. In the compatibility graph, each pattern is represented by a vertex. If two patterns have the same 2-byte suffix, their vertices receive no adjoining edge. On the contrary, if two patterns have different suffixes, their vertices are joined by an edge.

We group vertices together by finding maximal cliques (k_n subgraphs with maximal n). Each clique becomes a pattern group. This is a minimal graph coloring problem so we use a graph coloring heuristic. Finding a low number of groups to cover all patterns allows greater replication of the entire pattern set in the GPU.

Figure 1 shows an example compatibility graph and corresponding groupings for patterns 'PROGRAMMER', 'ACCELEROMETER', 'FOLKSINGER', 'GINGERBREAD', 'WIDESPREAD' and 'INSPECTION'. There are three resulting pattern groups as shown in Figure 1(b). The groups are ('PROGRAMMER','GINGERBREAD','INSPECTION), ('ACCELEROMETER','WIDESPREAD') and ('FOLKSINGER').

D. SWM Shift Table Construction

We begin shift table construction by finding the number of characters m, in the shortest pattern. We use the value of m to derive a shift value for all 2-byte substrings within the patterns in a set. To find the shift value for any 2-byte substring we use the distance in bytes v from the end of the pattern that the substring occurs. The shift value for any substring is calculated to be MIN(v, $m - B + 1$). Figure 2(a) shows the shift values for the 2-byte substrings in the pattern 'ACCELEROMETER'.

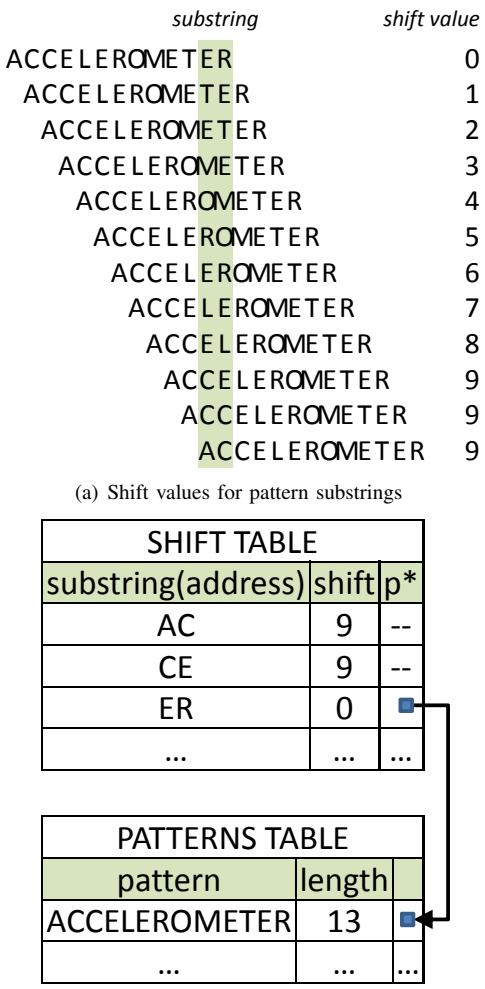
The shift values for each substring are stored in a direct indexed shift table as shown in Figure 2(b). If a substring has a shift value of zero, a pointer is added to the patterns table which contains the length and full pattern from which the substring was derived. This is used for comparing the pattern to the packet when a shift value of zero is found. Because of the pattern grouping, there will be only one pointer in the shift table for each pattern in the patterns table.

E. SWM Operation

SWM begins by looking up the shift value for 2-bytes (p) of a packet starting at offset $m - B + 1$. If the shift value (v) is non-zero, this is used as the stride and SWM repeats the process at packet byte (p + v). If the shift value is zero then the pointer in the shift table is used to compare a pattern from the patterns table to the packet text.

F. Group Balancing

Because substrings that do not exist in a pattern set receive a default stride value of m, which is the largest possible stride,



(a) Shift values for pattern substrings

(b) Shift tables

Fig. 2. Extracting shift values and shift table examples for the pattern 'ACCELEROMETER'.

pattern sets with fewer patterns may have a larger average stride. Since a packet must be processed by all pattern sets before deep packet inspection is complete for a packet, it makes sense that we should try to make the average stride for each pattern set similar, in order to equalize the processing time for each pattern set and minimize the overall latency for processing a packet. We therefore perform further refinement on our pattern groupings. After the minimum number of pattern groups is found, we attempt to equalize the number of patterns in each group without increasing the number of groups.

III. GPU Architecture

The functionality of SWM is split between the CPU and the GPU, as illustrated by Figure 3. The following sections describe the functionality of SWM in the CPU and the GPU.

A. CPU

As shown in Figure 3, the CPU host has several responsibilities. These responsibilities include creating the SWM shift

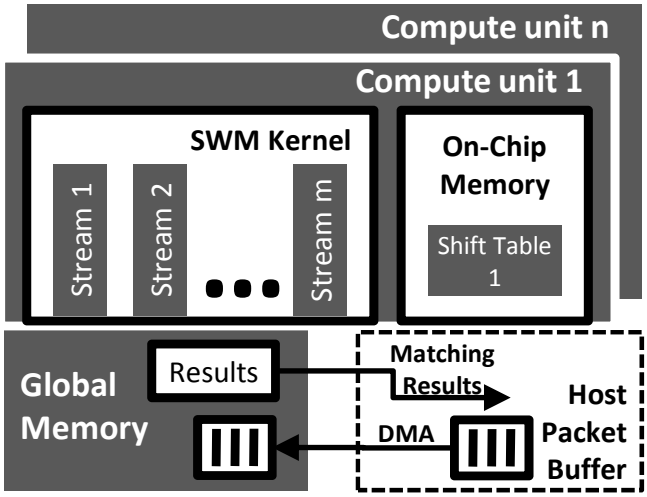


Fig. 3. SWM system architecture. Deep packet inspection is performed by the GPU where each stream core processes a separate packet and any matches are reported back to the CPU.

tables and transferring the tables to the local memory of the GPU compute units. The host maintains a current packet buffer which is mapped to the global memory of the GPU. The host also reads results from the matching buffer on the GPU and reports any potential attack patterns.

B. GPU

The SWM kernel runs on each stream core in the GPU, as shown in Figure 3. The following are specifics about the functionality of the kernel as well as GPU memory management.

1) Kernel: Each stream core on the GPU has a VLIW processor. In order to more efficiently use the VLIW processors we thread multiple, non-adjacent packet sections simultaneously per work item. This increases the utilization of the individual processing elements in each stream core. 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 global memory accesses. The ATI Radeon HD 6450 has 32 stream cores so this is the minimum number of work items that we will run on the GPU.

2) Memory: The local data store (LDS) of each compute unit, and the private memory of each stream core, contain the shift tables necessary for SWM kernel operation. Local access to the shift 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 to 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.

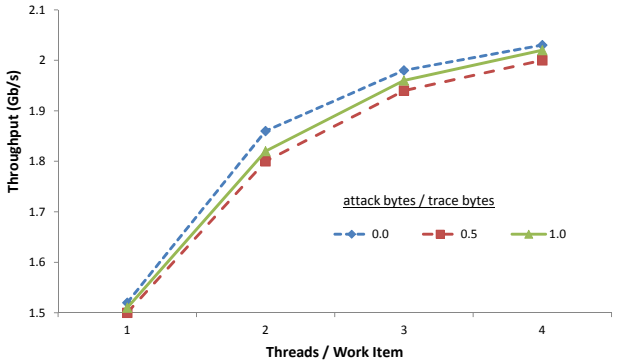


Fig. 4. Kernel performance using 1 to 4 copies of SWM sequentially per work item (kernel instance). Increasing the number of copies increases available instruction parallelism to efficiently utilize the VLIW processors. Increasing the number of payload bytes derived from attack signatures being searched also affects the performance of the SWM kernel.

IV. Performance Analysis

In this section we evaluate the performance of SWM on our test GPU. We begin by describing the experiment hardware, followed by evaluation of the SWM kernel performance.

A. Experiment Setup

We implement SWM on an ATI Radeon HD 6450 which has 32 stream cores and 512MB of DDR memory. Our host system contains an Intel I5 processor running at 3.3 GHZ and 8GB memory. The GPU and host interconnect via a PCIe 2.1 x 16 bus. SWM is written using Open Computing Language (OpenCL) [19] which abstracts the programming of various parallel computing devices. Using OpenCL allows SWM to be portable amongst most newer graphics processing units.

B. Kernel Performance

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

1) Kernel Thread Optimization: Figure 4 demonstrates the performance of SWM using an increasing number of copies of the SWM code in the kernel. We do this by copying the SWM code within the kernel code. This increases the parallel instructions available to the VLIW processors. As shown in Figure 4, the throughput increases when increasing the number of copies of SWM due to the increase in processor utilization.

Figure 4 also shows that SWM achieved a throughput of over 2 Gb/s. This throughput is achieves on a GPU with only 32 stream cores. Other GPUs have many more cores, such as the AMD 6970 which has 640 stream cores. Also observed in Figure 4, changing the payload content to contain a varying percentage of attack strings affects the throughput a minor amount.

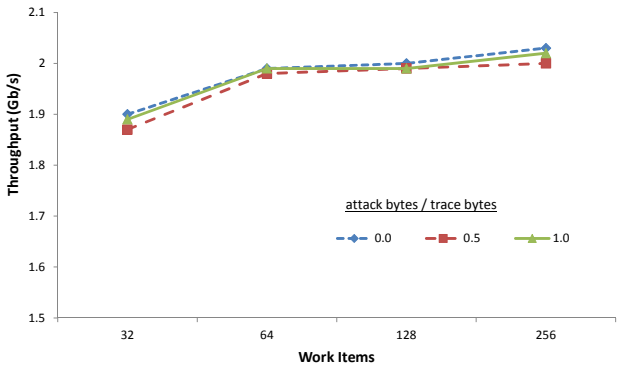


Fig. 5. Kernel performance when the number of work items is increased. Increasing the number of work items allows a stream core to execute one work item while another awaits a memory access.

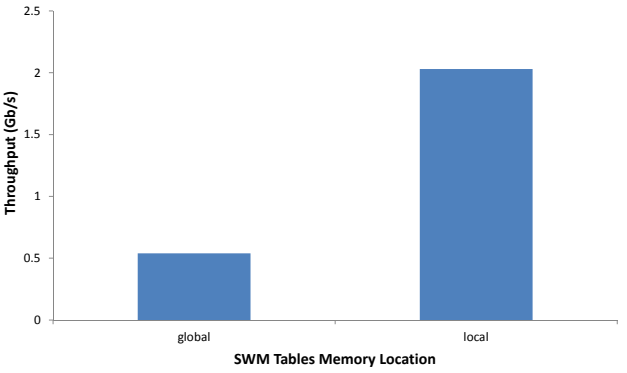


Fig. 6. Performance using global vs local memory to store the SWM shift tables on the GPU.

2) Work Item Optimization: Figure 5 demonstrates the effect of using a different number of work items. The lowest number of work items used is 32. With 32 work items, only one kernel instance will run on each stream core. If that stream core is waiting for a global memory access to execute an instruction, then the stream core's ALU will be idle. Increasing the number of work items allows the stream cores to execute a different kernel instance while another instance waits on a memory access. In Figure 5, we increase the number of work items from 32 up to 256. I minor increase in throughput is achieve by increasing the number of work items.

3) Memory Optimization: Figure 6 shows the performance of SWM when storing the shift tables in local vs global memory. As expected, storing the shift tables in local memory achieves much higher performance as this method does not suffer the same wait time as when using global memory. Overall, SWM achieves a throughput of about 2 Gb/s.

V. Related Work

There are two main methods to accelerate deep packet inspection. These are intra-stream parallelism and inter-stream parallelism. In intra-stream parallelism, multiple, contiguous bytes of a packet are scanned simultaneously. In inter-stream parallelism, multiple packets are scanned 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 [14] and derivatives [15] have produced multiple-pattern, multiple-stride average case algorithms. The complexity of these algorithms is not appropriate for GPU implementation. Brodie et al [20] 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 [16] 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 [21], the Cavium Networks CN1700 series [22] and the Netlogic NLS2008 [23] 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 and intra-stream parallelism. Vasiliadis et al [8], [9] have implemented deterministic finite automata (DFA) in a GPU. Unfortunately, the state transition tables have a large memory requirement. The state transition tables must be stored in global memory and require one global memory access for every byte processed in the worst case. GPEP [13] uses an optimized algorithm called P^3FSM which has similar complexity to a state transition table but reduces the memory requirement. However, this algorithm is limited to single stride. A GPU-based Wu-Manber modification [24] has been implemented but does not utilize the Wu-Manber shift table for packet strides and is limited to multiple Mb/s rather than Gb/s. SWM is a simplification rather than a modification of the Wu-Manber, algorithm which is most appropriate for a GPU implementation, allowing for multi-Gb/s speeds.

VI. Conclusion

Accelerating deep packet inspection is important to the effectiveness of network security due to the continuing need to increase the number and complexity of attack signatures. Given this requirement, graphics processing units can be used to process network traffic in parallel and improve signature scanning speeds. However, GPUs have limitations in terms of the type of algorithm that can be implemented, and multiple stride deep packet inspection algorithms are not congruent with these limitations. SWM solves this congruency problem

through a systematic pattern classification system. Our results indicate that SWM can achieve a consistent throughput of 2 Gb/s on a low end GPU.

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] L. Bu and J. A. Chandy, "Fpga based network intrusion detection using content addressable memories," in *FCCM: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Washington, DC, USA, 2004, pp. 316–317.

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

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

[7] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," in *SIGCOMM '10: Proceedings of the ACM SIGCOMM 2010 conference*, 2010, pp. 195–206.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] L. Vespa and N. Weng, "Deterministic finite automata characterization and optimization for scalable pattern matching," *ACM Transactions on Architecture and Code Optimization*, 2011.

[13] ——, "Gpep: Graphics processing enhanced pattern-matching for high-performance deep packet inspection," in *IEEE International Conference on Internet of Things (iThings 2011)*, 2011.

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

[15] 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. IPCCC 2006.*, 2006, pp. 680–686.

[16] 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.

[17] 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.

[18] *ATI Radeon 6450 GPU, note=http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6450/pages/amd-radeon-hd-6450-overview.aspx#1 year= 2011*.

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

[20] 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.

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

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

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

[24] N.-F. Huang, H.-W. Hung, S.-H. Lai, Y.-M. Chu, and W.-Y. Tsai, "A gpu-based multiple-pattern matching algorithm for network intrusion detection systems," in *22nd International Conference on Advanced Information Networking and Applications. AINAW 2008.*, March 2008, pp. 62–67.