Parallelization and Characterization of Pattern Matching using GPUs

Giorgos Vasiliadis
FORTH-ICS, Greece
gvasil@ics.forth.gr

Michalis Polychronakis
Columbia University, USA
mikepo@cs.columbia.edu

Sotiris Ioannidis
FORTH-ICS, Greece
sotiris@ics.forth.gr

Abstract

Pattern matching is a highly computationally intensive operation used in a plethora of applications. Unfortunately, due to the ever increasing storage capacity and link speeds, the amount of data that needs to be matched against a given set of patterns is growing rapidly. In this paper, we explore how the highly parallel computational capabilities of commodity graphics processing units (GPUs) can be exploited for high-speed pattern matching. We present the design, implementation, and evaluation of a pattern matching library running on the GPU, which can be used transparently by a wide range of applications to increase their overall performance. The library supports both string searching and regular expression matching on the NVIDIA CUDA architecture. We have also explored the performance impact of different types of memory hierarchies, and present solutions to alleviate memory congestion problems. The results of our performance evaluation using off-the-self graphics processors demonstrate that GPU-based pattern matching can reach tens of gigabits per second on different workloads.

1 Introduction

With ever increasing storage capacity and link speeds, the amount of data that needs to be searched, analyzed, categorized, and filtered is growing rapidly. For instance, network monitoring applications, such as network intrusion detection systems and spam filters, need to scan the contents of a vast amount of network traffic against a large number of threat signatures. Moreover, the scanning of unstructured data, like full-text searching and virus scanning, usually relies heavily on some form of regular expression matching. As the amount of data to be analyzed and the number and complexity of the patterns to be searched increase, content searching is becoming more difficult to perform in real time.

An important class of algorithms used for searching and filtering information relies on *pattern matching*. Pattern matching is one of the core operations used by applications such as traffic classification [1], intrusion detection systems [5], virus scanners [3], spam filters [6], and content monitoring filters [2, 4]. Unfortunately, this core and powerful operation has significant overheads in terms of both memory space and CPU cycles, as every byte of the input has to be processed and compared against a large set of patterns. A possible solution to the increased overhead introduced by pattern matching is the use of hardware platforms, although with a high and often prohibitive cost for many organizations. Specialized devices, such as ASICs and FPGAs, can be used to inspect an input data stream and offload the CPU [10–12]. Both are very efficient and perform well, however they are complex to program and modify, and they are usually tied to a specific implementation.

The advent of commodity massively parallel architectures, such as modern graphics processors, is a compelling alternative option for inexpensively removing the burden of computationally-intensive operations from the CPU. The data-parallel execution model of modern graphics processing units (GPUs) is a perfect fit for the implementation of high-performance pattern matching algorithms [7, 9, 14, 17–21]. Moreover, the fast-growing video game industry exerts strong economic pressure that forces constant innovation, while keeping the cost at a low rate.

In this paper, we focus on the implementation of string searching and regular expression matching on the GPU. With sufficient performance, two orders of magnitude faster than traditional CPU algorithms, a GPU-based pattern matching engine enables content scanning at multi-gigabit rates, and allows for real-time inspection of the large volume of data transferred in modern network links. Efficient GPU algorithms are capable of scanning up to 30 times faster than a single CPU core, including the cost of *all* data transfers to and from the device, reaching a maximum throughput of about 30 Gbit/s.

The contributions of this work include:

- The implementation of a GPU-based pattern matching library for inspecting network packets in real-time, with support for both string searching and regular expression matching operations.

- The performance evaluation of the pattern matching engine using different memory hierarchies that modern graphics processors provide. We characterize the performance of our implementation on different types of memory, and identify the setup that performs best.

- An efficient packet buffering scheme for transferring network packets to the memory space of the GPU for inspection. Our scheme is quite efficient on different packet lengths, alleviating the performance degradation that small packets incur in previous implementations. The overall packet processing throughput ranges between 6.49 Gbit/s for very small packets, up to 29.7 Gbit/s for packets with full payload.

2 Background

2.1 Graphics Processors

Modern graphics processing units (GPUs) have evolved to massively parallel computational devices, containing hundreds of processing cores that can be used for general-purpose computing beyond graphics rendering. The fundamental difference between CPUs and GPUs comes from how transistors are assigned to different tasks in the processor. A GPU devotes most of its die area to a large array of Arithmetic Logic Units (ALUs). In contrast, most CPU resources serve a large cache hierarchy and a control plane for sophisticated acceleration of a single thread.

The architecture of modern GPUs is based on a set of *multiprocessors*, each of which contains a set of *stream processors* operating on SIMD (Single Instruction Multiple Data) programs. For this reason, a GPU is ideal for parallel applications requiring high memory bandwidth to access different sets of data. Both NVIDIA and AMD provide convenient programming libraries to use their GPUs as a general purpose processor (GPGPU), capable of executing a very high number of threads in parallel.

A unit of work issued by the host computer to the GPU is called a *kernel*. A typical GPU kernel execution takes the following four steps: (i) the DMA controller transfers input data from host memory to GPU memory; (ii) a host program instructs the GPU to launch the kernel; (iii) the GPU executes threads in parallel; and (iv) the DMA controller transfers the results data back to host memory from device memory. A kernel is executed on the device as many different *threads* organized in *thread blocks*, and each multiprocessor executes one or more thread blocks.

A fast shared memory is managed explicitly by the programmer among thread blocks. The *global*, *constant*, and *texture* memory spaces can be read from or written to by the host, are persistent across kernel launches by the same application, and are optimized for different memory usages [13].

In the new Fermi architecture, each multiprocessor has 32 stream processors, instead of eight in the previous generation. The GTX480 graphics card, that we used in this work, contains 15 multiprocessors. In addition, each multiprocessor has 16–48 KB of L1 cache, while all multiprocessors share 768 KB of coherent L2 cache (both handled by the GPU automatically). The shared memory is increased from 16 to 48 KB, however shared memory and L1 cache must add up to 64 KB in total, as they are implemented using the same physical memory.

2.2 Pattern Matching

String searching and regular expression matching are two of the most common pattern matching operations. In string searching, a set of fixed strings is searched in a body of text. Regular expressions, on the other hand, offer significant advantages, providing flexibility and expressiveness in specifying the context of each match. In addition to matching strings of text, they offer wild-card characters, logical operators, repeating patterns, range constraints, and recursive forms. Thus, a single regular expression can cover a large number of individual string representations.

Both string patterns and regular expressions can be matched efficiently by compiling the patterns into a *Deterministic Finite Automaton (DFA)*. A sequence of n bytes can be processed using $O(n)$ operations irrespectively of the number of patterns, which is very efficient in terms of speed. This is achieved because at any state, every possible input byte leads to *at most* one new state.

Aiming to take advantage of the extreme thread-level parallelism of modern GPUs, we have parallelized the DFA-based matching process by splitting the input data stream into different chunks. Each chunk is scanned independently by a different thread using the same automaton that is stored in device memory. Although threads use the same automaton, each thread maintains its own state, eliminating any need for communication between them.

2.2.1 Regular Expressions and Finite Automata

A regular expression is a very convenient form of representing a set of strings. They are usually used to give a concise description of a set of patterns, without having to list all of them. For example, the expression *(a* $|$ *b)* $*$ *aa* represents the infinite set $\{$*"aa", "aaa", "baa", "abaa", ...*$\}$, which is the set of all strings with characters *a* and *b* that end in *aa*. Formally, a regular expression contains at least one of the operations described in Table 1.

A deterministic finite automaton (DFA) represents a finite state machine that recognizes a regular expression. A finite automaton is represented by the 5-tuple (Σ, Q, T, q_0, F), where: Σ is the alphabet, Q is the set of states, T is

Table 1. Regular expression operations.

$Name:$	$Reg.$ $Expr.$	$Designation$
Epsilon	ϵ	$\{$””$\}$
Character	α	For some character α.
Concatenation	*RS*	Denoting the set $\{\alpha\beta \mid \alpha$ in *R* and β in *S*$\}$. e.g., $\{$”ab”$\}\{$”d”, ”ef”$\}$ = $\{$”abd”, ”abef”$\}$
Alternation	$R \mid S$	Denoting the set union of *R* and *S*. e.g., $\{$”ab”$\} \mid \{$”ab”,”d”,”ef”$\}$ = $\{$”ab”,”d”,”ef”$\}$.
Kleene star	$A*$	Denoting the smallest super-set of *R* that contains ϵ and is closed under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in *R*. e.g., $\{$”ab”, ”c”$\}$* = $\{\epsilon$,”ab”,”c”,”abab”,”abc”,”cab”,”ababab”,...$\}$

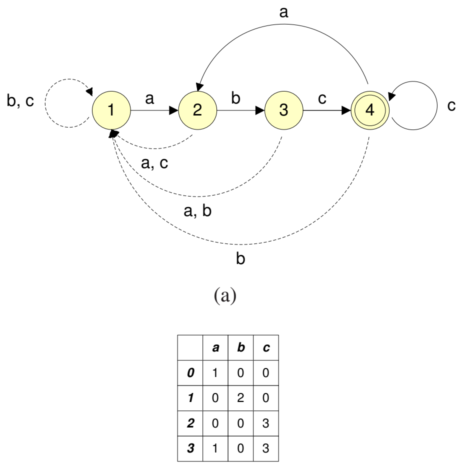
the transition function, q_0 is the initial state, and F is the set of final states. Given an input string $I_0 I_1 ... I_N$, a DFA processes the input as follows: At step 0, the DFA is in state $s_0 = q_0$. At each subsequent step i, the DFA transitions into state $s_i = T(s_{i-1}, I_i)$. To alleviate backtracking at the matching phase, each transition is unique for every state and character combination.

A DFA accepts a string if, starting from the initial state and moving from state to state, it reaches a final state. The transition function can be represented by a two-dimensional table T, which defines the next state $T[s, c]$ for a state s and a character c. For example, the regular expression $(abc+)+$ is recognized by the DFA shown in Figure 1. The automaton has four states, state 0 is the start state, and state 3 is the only final state.

2.2.2 Converting a Regular Expression into a Deterministic Finite Automaton

Many existing tools that use regular expressions, such as `grep(1)`, `flex(1)` and `pcre(3)`, have support for converting regular expressions into DFAs. The most common approach is to first compile them into *non-deterministic finite automata* (NFAs), and then convert them into DFAs. We follow the same approach, and first convert each regular expression into a NFA using the Thompson algorithm [15]. The generated NFA is then converted to a DFA incrementally, using the *Subset Construction* algorithm. The basic idea of subset construction is to define a DFA in which each state is a set of states of the corresponding NFA. The resulting DFA achieves $O(1)$ computational cost for each consumed character of the input during the matching phase.

Each DFA is represented as a two-dimensional state table array that is mapped on the memory space of the GPU. The dimensions of the array are equal to the number of states and the size of the alphabet (256 in our case), respectively. Each cell contains the next state to move to, as well as an indication of whether the state is a final state or not. Since transition numbers may be positive integers only, we represent final states as negative numbers. Whenever the state machine reaches into a state that is represented by a negative number, it considers it as a final state and reports a match at the current input offset.



(a)

	a	*b*	*c*
0	1	0	0
1	0	2	0
2	0	0	3
3	1	0	3

(b)

Figure 1. The DFA state machine (a) and the state transition table (b) for the regular expression $(abc+)+$.

3 Implementation

We have developed a library for executing both string searching and regular expression matching operations on the GPU. The library provides functions for inspecting network packets, and returning any matches found back to the application. The searching functions have support for processing input data either from a saved trace file, or directly from the network interface. This allows the library to be transparently used by a broad range of applications to offload their costly pattern matching operations to the GPU, and thus increase their overall performance.

Initially, all patterns are compiled to DFA state tables. The user is able to compile each pattern to a separate DFA, or combine many different patterns to a single one. The compilation process is performed offline by the CPU, usually during the initialization phase of the user application. The state table is then copied and mapped to the memory space of the GPU. At the searching phase, each thread searches a different portion (i.e., a separate network packet) of the input data stream. In order to fully utilize the data-parallel capabilities of the GPU, the library creates a large number of threads that run simultaneously. The core processing loop splits the input packets, and distributes them for processing to different threads.

Table 2. Data transfer rate between host and device (Gbit/s).

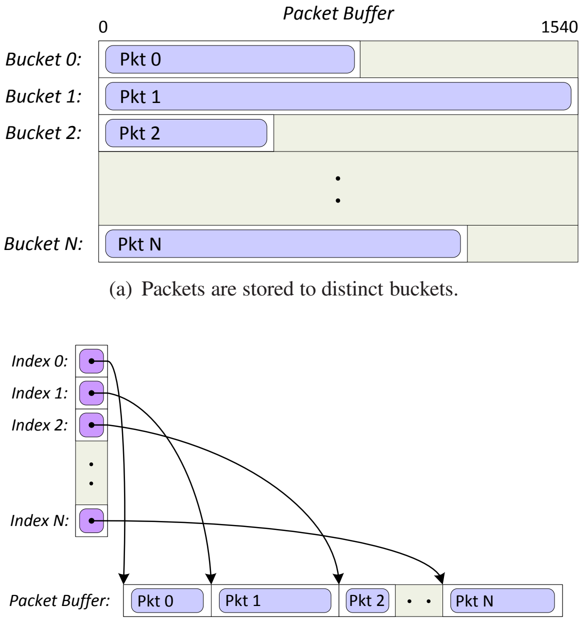
Buffer Size	1KB	4KB	64KB	256KB	1MB	16MB
Host to Device	2.04	7.1	34.4	42.1	44.6	45.7
Device to Host	2.03	6.7	21.1	23.8	24.6	24.9

3.1 Transferring Network Packets to the GPU

The first thing to consider is the transfer of the packets to the memory space of the GPU. A major bottleneck for this operation, is the extra overhead, caused by the PCIe bus that interconnects the graphics card with the base system. The bandwidth of the PCIe bus has evolved over the last versions—v2.0 is able to perform at 512 MB/s per x1 lane. This results to 8 GB/s overall throughput for a PCIe x16 graphics card. Unfortunately, the PCIe bus suffers many overheads, especially for small data transfers. Table 2 shows the transfer rate to move data to a single GPU device, and vice versa. We observe that with a large buffer, the rate for transferring to the device is over 45 Gbit/s,[1] while transferring from device to host decreases to about 25 Gbit/s. The asymmetry in the data transferring throughput from the device, is probably related to our corresponding hardware setup (i.e., the interconnection between the motherboard and the graphics cards), and has been observed by other researchers too [8]. We speculate that future motherboards will alleviate this asymmetry.

As a consequence, network packets are transferred to the memory space of the GPU in batches. A separate packet buffer is allocated to collect the incoming packets. Whenever the buffer gets full, all packets are transferred to the GPU in one operation. As we will see in Section 4.3, the format of the packet buffer plays a significant role in the overall packet processing throughput. First, it affects the transferring overheads, as small data transfer units achieve a reduced bandwidth due to PCIe and DMA overheads. Second, the packet buffer scheme affects the parallelization approach, i.e., the distribution of the network packets to the stream processors. The simplest the buffer format, the better the parallelization scheme.

In this work, we have implemented two different approaches for collecting packets. The first uses fixed buckets for storing the network packets, and has been previously adapted in similar works [17,18]. The second approach uses a more sophisticated, index-based, scheme. Instead of pre-allocating a different, fixed-size, bucket for each packet, all packets are stored back-to-back into a serial packet buffer. A separate index is maintained, that keeps pointers to the corresponding offsets in the buffer, as shown in Figure 2(b). Each thread reads the corresponding packet offset independently, using its own thread number, without any lock or synchronization mechanism needed. In order to avoid an extra transaction over the PCIe bus, the index array is stored in the beginning of the packet buffer. The packet buffer and the indices are transferred to the GPU at once, adding a minor transfer cost, since the size of the index array is quite small in regards to the size of the packet buffer.



(a) Packets are stored to distinct buckets.

(b) Packets are stored sequentially and indexed by a separate directory.

Figure 2. Different packet buffer formats.

3.2 Pattern Matching on the GPU

During scanning, the algorithm moves over the input data stream one byte at a time, as shown in Figure 3. For each consumed byte, the matching algorithm switches the current state according to the state transition table. The pattern matching is performed byte-wise, meaning that we have an input width of eight bits and an alphabet size of $2^8 = 256$. Thus, each state will contain 256 pointers to other states. The size of the DFA state transition table is $|\#States| * 1024$ bytes, where every pointer occupies four bytes of storage. When a final-state is reached, a match has been found, and the corresponding offset is marked. The format of the state table allows its easy mapping to the different memory types that modern GPUs offer. Mapping the state table to each memory yield different performance improvements, as we will see in Section 4.2.

[1] The deviation from the theoretical 64 Gbit/s throughput arises from the 8b/10b encoding scheme at the physical level. The scheme ensures that strings of consecutive binary digits are limited in length.

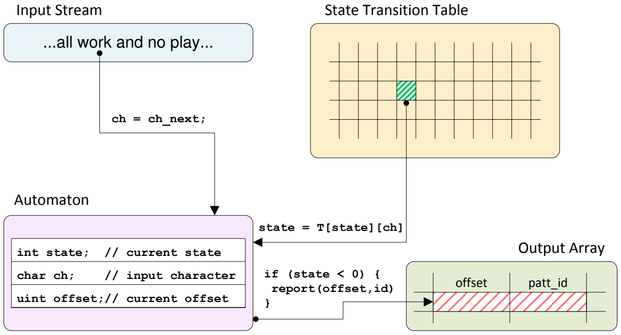


Figure 3. DFA matching on the GPU.

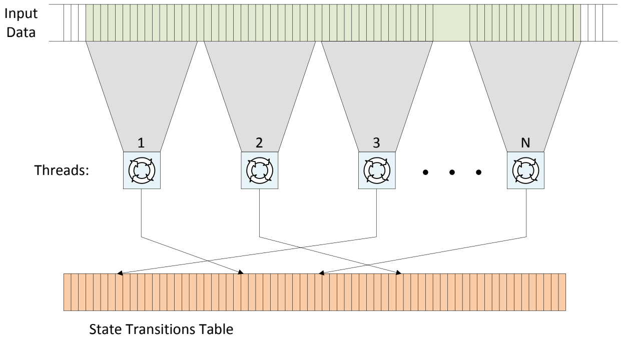


Figure 4. Multi-thread pattern matching on the GPU.

We take advantage of all the available streaming processors of the GPU and utilize them by creating multiple data processing threads. An important design decision is how to assign the input data to each thread. Due to the nature of fine-grained parallelism, we follow the simplest approach and assign a different packet to each thread. This allows the threads to operate independently and avoid any state exchange or synchronization. A potential drawback is the asymmetrical processing effort, due to the packets length variation. However, this is not a problem since the size of a typical network packet lies between 40 and 1500 bytes (as of Ethernet links). In addition, we have try to sort the packets, in order to minimize divergence of control flow, however the cost of sorting is proportionally larger than the gains from the symmetric processing.

Every time a thread matches a pattern, it reports it by appending it in an array that has been previously allocated in the global device memory. Each thread maintains its own memory space inside the array, hence adding new matches can be performed in parallel, without the need of synchronization. For each packet, we maintain a counter for storing the total number of matches found in the packet, and a fixed number of entries (currently 32) for saving the exact matching offsets inside the packet. The match counter allows the quick iteration of the matches at the CPU side, without reading all the entries of the array. Unfortunately, in case the counter exceeds 32, the portion of the packet, starting from the last matching offset up to the end, needs to be rescanned from the CPU, since only the first 32 matches will be reported by the GPU. The intuition behind this scheme is that it is better to keep the transferring costs low, by maintaining a small buffer for the results, rather than sacrificing the overall throughput in order to be more precise for distinct cases. In other words, having 32 matching offsets for 1500-byte packets is fair enough for the majority of network applications that rely on pattern matching, such as NIDS and traffic classification applications.

3.3 Optimized Device Memory Management

The two major tasks of DFA matching, as described in the previous section, is reading the input data and fetching the next state from the device memory. These memory transfers can take up to several hundreds of nanoseconds, depending on the stream conditions and congestion.

In general, memory latencies can be hidden by running many threads in parallel. Multiple threads can improve the utilization of the memory subsystem by overlapping data transfer with computation. To obtain the highest level of performance, we performed several tests to determine how the computational throughput is affected by the number of threads. In Section 4.2 we discuss how the memory subsystem is utilized when there is a large number of threads running in parallel.

Moreover, we have investigated storing the network packets and the DFA state table both in the global memory space, as well as in the texture memory space of the graphics card. The texture memory can be accessed in a random fashion for reading, without the need to follow any coalescence rules. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. In advance, texture cache is optimized for 2D spatial locality; to that end, we have investigated the use of both 1D and 2D textures. A programming limitation when dealing with 2D textures, is that the maximum y-dimension is equal to 65,536. Therefore, in order to map large state tables, we split the initial table into several smaller (each of which contains $64K$ states at most) and align them sequentially. In order to find the transitions of a given state at the matching phase, it is first divided with 65,536 in order to find the subtable that resides.

When using 1D linear memory, the maximum transition table that can be mapped to texture memory is 512 MB (totalling 524,288 states, since each state holds 1024 bytes for transitions). The theoretical dimensions of the maximum 2D texture are equal to 64K × 32K elements, which are by far greater than the total amounts of memory that a modern GPU holds (i.e., up to 3 GB currently). Therefore, for cases that a single 1D state table is greater than 512 MB, we bind only the initial part of the table to the texture memory, leaving the rest in the global device memory.

Finally, one important optimization is related to the way the input data is loaded from the device memory. Since the input symbols belong to the ASCII alphabet, they are represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput would be reduced by a factor of up to 32. To utilize the memory more efficiently, we redesigned the input reading process such that each thread is fetching multiple bytes at a time instead of one. We have explored fetching 4 or 16 bytes at a time using the `char4` and `int4` built-in data types, respectively. The `int4` data type is the largest data type that can be used for texture alignment, allowing the utilization of about 50% of the memory bandwidth.

Finally, we tried to stage some data on the on-chip shared memory, but there was not any improvement due to the following reasons. First, the tradeoff of copying the data to the shared memory is worse than the benefit that the shared memory can provide, since each byte of the input is accessed only once. Second, by not using the shared memory, the performance of global memory accesses is boosted, since shared memory and L1-cache are implemented using the same physical memory. Therefore, in our implementation we do not take advantage of the shared memory. Nevertheless, input data are transferred in a 128-bit register using the `int4` built-in data type, and are accessed byte-by-byte, through the `.x`, `.y`, `.z` and `.w` fields.

3.4 Host Memory Optimizations

In addition to optimizing the device memory usage, we considered two other optimizations: the use of *page-locked* (or *pinned*) memory, and the reduction of the number of transactions between the host and the GPU device.

The page-locked memory offers better performance, as it does not get swapped. Furthermore, it can be accessed directly by the GPU through DMA. Hence, the use of page-locked memory improves the overall performance by reducing the data transferring costs to and from the GPU. The network packets are placed into a buffer allocated from page-locked memory through the CUDA driver. The buffer is then transferred through DMA from the physical memory of the host to the device memory of the GPU. To further improve performance, we use a large buffer to store the contents of multiple network packets, which is then transferred to the GPU in a single transaction. This results in a reduction of I/O transactions over the PCI Express bus.

Table 3. Memory requirements and properties of each pattern set.

	#Patterns	Min./Max./Avg. Pattern	DFA Size
SET1	2,000	5/34/19.42	33.02 MB
SET2	10,000	5/34/19.57	162.14 MB
SET3	30,000	5/34/19.57	478.36 MB
SET4	50,000	5/34/19.53	799.76 MB

4 Performance Evaluation

Hardware. For our testbed we used the NVIDIA GeForce GTX480 graphics card. The GPU is equipped with 480 cores organized in 15 multiprocessors and 1.5GB of GDDR5 memory. Our base system is equipped with an Intel Xeon E5520 Quad-core CPU at 2.66GHz with 8192KB of L3-cache, and a total of 12GB of memory. The GPU is interconnected using a PCIe 2.0 x16 bus.

Data Sets. We use synthetic network traces and synthetic patterns, in order to control the impact of the network packet sizes and the size of the patterns to the overall performance. The content of the packets, as well as the patterns, are completely random, following a uniform distribution from the ASCII alphabet. Table 4 shows the properties of the patterns and the memory requirements of the generated DFA state machine. In all experiments, the data to be scanned are transferred from the host memory, to the GPU memory space in batches.

The structure of the input data force the matching engine to exercise most of the code branches in all scenarios. Due to their high entropy, random workloads can be assumed quite representative for evaluating pattern matching [16]; specific scenarios, like virus detection or traffic classification may perform better, due to the lower entropy between the scanned content and the patterns themself.

4.1 Fetching the packets from the device memory

In our first experiment, we evaluate the performance of reading the packets from the memory of the GPU, using different sized word accesses. Figure 5 shows the performance achieved when fetching 1, 4, and 8 bytes at a time. The horizontal axis corresponds to the number of packets processed at once. Each packet is processed by one thread, hence the number of packets is equal to the number of threads. The network packets (1500-bytes long) reside on the global device memory. Each byte in the packet requires two memory

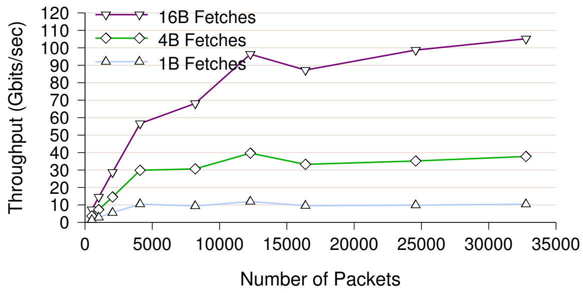


Figure 5. Impact of word accesses when fetching data from the global device memory.

accesses: one access for fetching the contents of the packet, and one access to the state machine, in order to find the next state to traverse.

As we increase the number of threads, the multithreaded capabilities of the GPU at hiding memory latencies, results to an increase in the throughput sustained by the GPU. An interesting observation is that performance levels-off proportionally to the size of the word accesses. For example, when fetching the input data one byte at a time, we observe that the throughput sustained by the GPU remains constant after processing 4096 packets at a time. Moreover, using the `char4` and `int4` data types for loading the data from the device memory has a positive impact to overall performance. When reading the data one-byte at a time, unused bytes are transferred for every device memory transaction, since the minimum size per transaction is 32 bytes. The `char4` type uses four bytes per transaction and boosts the performance up to four times. With 16 bytes per transaction using the `int4` type, an additional performance boost of 300% is achieved, while the plateau starts when using 24576 threads.

It is clear that reducing the number of memory transactions from the device memory, results in a significant increase of the processing throughput. Finally, we observe a performance degradation, when moving from 12288 to 16384 threads, which we speculate that is related to the internal GPU thread scheduler.

4.2 Evaluating Memory Hierarchies

Figure 6 shows the raw processing throughput obtained for different types of memories. The horizontal axis corresponds to the number of packets processed at once. Each thread process a different packet in isolation, fetching 16-bytes at once, which performs better as we have shown in the previous experiment. By storing the network packets and the state table to different types of memory, we measure how each type affects the processing throughput.

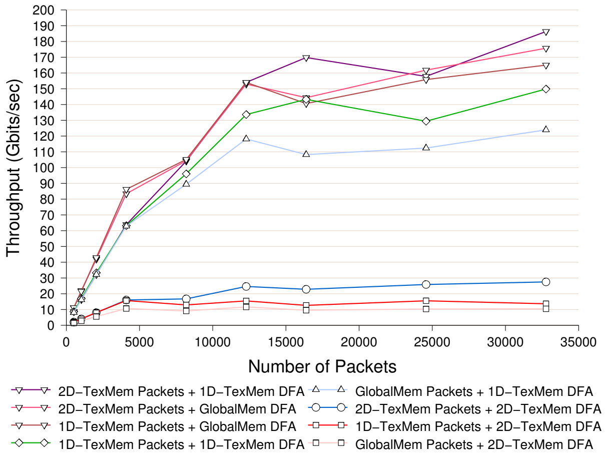


Figure 6. Memory accesses impact on DFA matching.

Storing the state table as a 2D texture significantly decreases the overall throughput. We speculate that state table accesses exhibits bad 2D spatial locality, hence the 2D optimized textured cache reduces the performance. In contrast, when accessing the network packets, the 2D textures sustain the best performance. All threads achieve coalesced reads when accessing packet data, in contrast to DFA matching that exhibits irregular memory accesses. This irregularity might lead to cache thrashing, and it results in very poor performance.

Regarding packet accesses, we observe that 1D texture memory improves about 20% the performance, while 2D textures provide a 50% improvement over global device memory. On the other hand, global device memory and 1D texture differ slightly for state table accesses, with global memory providing about 10% better performance. The GPU contains caches for both types of memory—a 12KB L1-cache per multiprocessor for texture memory, and a 16KB L1-cache per multiprocessor for global memory—hence the performance is almost the same for state table accesses. When there is a cache hit, the latency for a fetch is only a few cycles, against the hundreds of cycles required to access the global memory.

An interesting observation, is that texture memory seems to fit better for packet accesses, in contrast to state table accesses that performs better on global device memory. Although texture memory does not require to follow any coalescence patterns, it seems that is expose better cache performance.

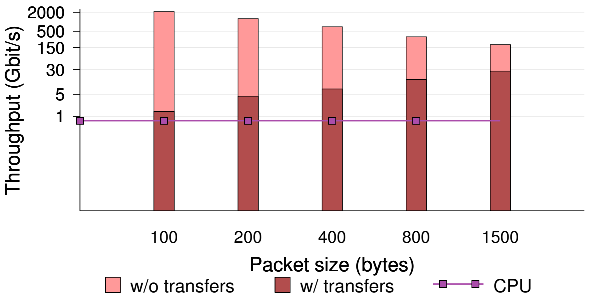


Figure 7. Throughput sustained including data transfers.

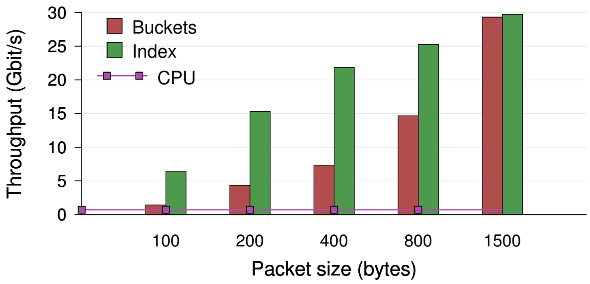


Figure 8. Buckets versus Index Packet Buffer.

4.3 Bucket-based vs. Index-based packet buffer

In the previous experiments, we evaluated the throughput sustained from the GPU alone, when packet data reside on the device memory. Unfortunately, graphics cards act as a coprocessor in computer systems, hence it is essential that data have to be transferred from the host memory, to the memory of the graphics card, over the PCIe bus. Therefore, the transferring scheme of the data plays a significant role in the overall performance.

Figure 7 shows the throughput of pattern matching, using the fixed-buckets transferring scheme used in previous works [18, 20]. We also include the throughput sustained when data reside on the GPU memory, for comparison reasons. When data transfers are not included, we can see that the throughput achieved decreases when packet size increase. For 100-byte packets, the GPU computational throughput raises to 2023.7 Gbit/s, that fall to 186 Gbit/s for 1500-byte packets. Smaller packets require less memory accesses, hence do not suffer from excessive memory latencies, as larger packets do. Unfortunately, the PCIe transfers introduce a significant overhead to the overall throughput, that is further influenced by the size of the packets.[2] For example, full-payload packets provide a 27.1 Gbit/s throughput, while 200-byte and 100-byte packets fall to 4.3 Gbit/s and 1.4 Gbit/s respectively, resulting to a 20x times deviation. The fixed-bucket buffer, shown in Figure 2(a), suffer from redundant data transfers, when small packets are collected. Since each bucket is 1500 bytes, only the $1/15$ of the total space is utilized for a 100-byte packet.

In Figure 8, we compare our novel indexed packet buffer scheme with the fixed-bucket array that was used in previous works. The performance for full-payload packets is the same, hence our index-based scheme does not affect the data parallelism at the scanning phase. For the fixed-bucket scheme, full-payload packets provide the best-case performance. Smaller packet sizes results to lower space utilization in the fixed-bucket array. On the other hand, our index-based scheme provides better space consumption, therefore the performance is higher for small packets. Even for tiny packets (i.e., 100 bytes length) the performance is about 5 times better, resulting to 6.49 Gbit/s throughput. The equivalent performance for the single CPU core implementation is 0.72 Gbit/s, hence our GPU version achieves the performance that (assuming an ideal parallelization) would be achieved using 41.2 CPU cores for 1500-byte packets, and 8.75 CPU cores for 100-byte packets.

The reason the performance of our index-based buffer still degrades when packet sizes decrease, is the extra cost spent for transferring and processing the matching results on the CPU side—since we allocate a separate space for each packet in order to store the matches, increasing the number of packets results to larger results transfers and more overhead at iterating through the returned matches.

4.4 Scalability to number of patterns

In the next experiment we evaluate how our experiment scales with the number of patterns. We used the sets of patterns shown in Table 4, and gave as input full-payload packets. Figure 9 shows the maximum throughput achieved for our pattern matching implementation to perform searches through rule-sets of sizes 2,000 up to 50,000 rules. We observe that the throughput remains constant independently of the number of patterns, a behavior expected for a DFA approach.

4.5 Scaling Factor

To measure how our optimized GPU pattern matching implementation has improved during the evolution of GPU models, we used three additional older-generation graphics

[2] The size of the packets include the headers of all encapsulated protocols (i.e., TCP/UDP, IP, and Ethernet), summing to 40 bytes size. At pattern matching, headers are ignored and only the payload is scanned.

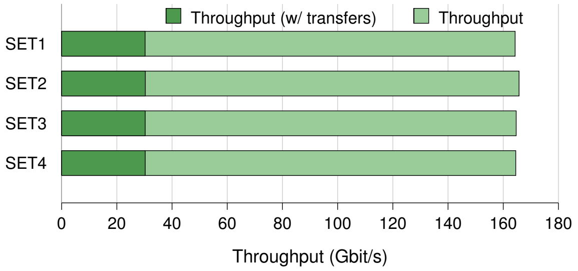


Figure 9. Overall throughput for different pattern numbers.

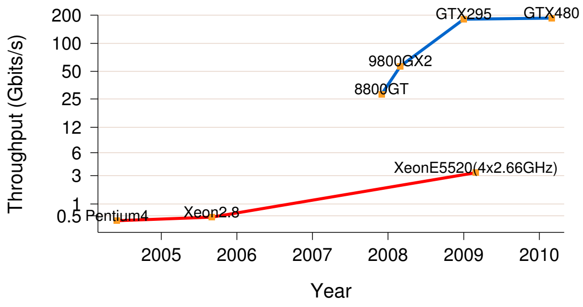


Figure 10. Performance sustained by our pattern matching implementation with all optimizations described in Section 3.3, on different generation of GPU and CPU models.

cards: a GeForce 8800GT, which was released on December 2007, a GeForce 9800GX2 released 4 months later, on March 2008, and a GTX295 released on January 2009. We only measure the raw computational throughput achieved on the GPU (no data transfers over the PCIe are included). Figure 10 shows that in less than two years, the computational throughput has increased about 6 times, from 28.1 Gbit/s to over 180 Gbit/s.

It must be noted that GTX295 consists of two printed circuit boards (PCBs). Since we measure execution time for both boards, the computational throughput achieved reaches the performance of GTX480. However, if we include the data transfers, the throughput of GTX295 matches half of the throughput of GTX480, since both PCBs use the same PCIe bus. For comparison reasons, we also measured and include the respective numbers for various generations of CPUs.

5 Related Work

Improving the performance of pattern matching algorithms has been extensively studied over the years. Specifically, the advent of general-purpose graphics processors has led researchers toward implementing pattern matching algorithms on GPUs. Gnort [18, 20] was the first attempt that sufficiently utilized the graphics processors for string searching and regular expression matching operations. For performance issues, Gnort utilized a DFA for pattern matching, at the cost of high memory utilization. GrAVity [19] reduces the memory consumptions of the resulting DFA, by compiling only the prefices (i.e., the first n-bytes) of the virus signatures.

Other approaches, adopt non-deterministic automata, allowing the compilation of very large and complex rule sets that are otherwise hard to treat [7]. Furthermore, Gnort takes advantage of DMA and the asynchronous execution of modern GPUs, to partially hide the data transfer costs from the host memory to the device memory, and vice versa. Therefore, Gnort imposes concurrency between the operations handled by the CPU and the GPU. Many other approaches followed the above scheme [9], without significant differences in the architecture and the performance benefits.

In order to speed-up the pattern matching computation on the GPU, Smith et al. [14] and Tumeo et al. [16, 17], redesigned the packet reading process, such that each thread is fetching four bytes at a time, instead of one. Since the input symbols belong to the ASCII alphabet, they are represented with 8 bits. However, the minimum size for every device memory transaction is 32 bytes. Thus, by reading the input stream one byte at a time, the overall memory throughput may be reduced by a factor of up to 32. To overcome this limitation, the authors use the `char4` built-in data type (4-byte size), to read the content of each packet. Unfortunately, they do not experiment with larger word accesses (e.g., 16-byte offered by the `int4` data type).

Our work differs from previous work in that we extensively explore *all* available memory hierarchies that modern GPUs are provide. In previous works, the authors used the texture memory for storing the state transition table, as well as the network packets. However, Fermi architecture provides caches for both texture and global memories, hence using both types of memories results to better cache performance.

In addition, most of the previous works do not concentrate on data transfers to the memory of the GPU. Although the pattern matching on the GPU can reach several hundreds of Gigabits per second (depending on the packet size), the limit factor of the end-to-end performance is the PCIe bus and the trasfer throughput that can sustain. Therefore, an important design decision for packet processing on the GPU is the transferring of the packets.

Previous works that used a static buffer scheme (i.e., a constant space for each packet), resulted to low transfer throughput, especially for low packet sizes, where the packet buffer is transferred to the GPU almost empty. In contrast, we explore a novel schema, that enforce two properties: first, it ensures that *no redundant* data are transferred to the GPU every time. Second, it does not affect the data parallelism at the scanning phase. Our scheme can sustain a 6.49 Gbit/s throughput for small packets, and a 29.7 Gbit/s for full-payload packets. Comparing with the Tesla C2050 throughput, reported in [16], our implementation is about three times faster.

6 Conclusion

In this paper we presented an efficient DFA implementation of both string searching and regular expression matching on GPU architectures. We evaluated our implementation using the different memory hierarchies provided by modern GPUs and explored the various trade-offs. We also presented several optimizations for efficiently implementing the matching algorithms to the GPU, as well as a packet buffering scheme that improves the transferring costs with the high parallelization on the GPU. Our index-based packet buffer is able to reduce the large overheads due to small packets, which are incurred in previous implementations.

As part of our future work, we plan to explore alternative designs of the state table with a focus on memory space efficiency. Some potential directions for achieving this goal include the conversion of the state transition table into other data structures, such as a banded matrix.

Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS; by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007; and by the project i-Code, funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission – Directorate-General for Home Affairs under Grant Agreement No. JLS/2009/CIPS/AG/C2-050. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein. Giorgos Vasiliadis is also with the University of Crete.

References

[1] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net/.

[2] Cisco IOS Netflow. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[3] Clam AntiVirus. http://www.clamav.net/.

[4] MSCG Hierarchical DPI Solution. http://www.huawei.com/products/datacomm/catalog.do?id=1219.

[5] Snort IDS/IPS. http://www.snort.org.

[6] SpamAssassin. http://spamassassin.apache.org/.

[7] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review*.

[8] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM, 2010.

[9] 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 *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops*, AINAW, 2008.

[10] R.-T. Liu, N.-F. Huang, C.-H. Chen, and C.-N. Kao. A Fast String-matching Algorithm for Network Processor-based Intrusion Detection System. *ACM Transactions on Embedded Computing Systems*, 3(3):614–633, 2004.

[11] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the 19th Usenix Security Symposium (USENIX Sec '10)*, 2010.

[12] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS, 2007.

[13] NVIDIA. CUDA Programming Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[14] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2009.

[15] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[16] A. Tumeo, S. Secchi, and O. Villa. Experiences with string matching on the fermi architecture. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, ARCS, 2011.

[17] A. Tumeo, O. Villa, and D. Sciuto. Efficient pattern matching on GPUs for intrusion detection systems. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF, 2010.

[18] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. 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*, RAID, 2008.

[19] G. Vasiliadis and S. Ioannidis. GrAVity: A Massively Parallel Antivirus Engine. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection*, RAID, 2010.

[20] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID, 2009.

[21] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS, 2011.