

# GrAVity: A Massively Parallel Antivirus Engine

Giorgos Vasiliadis and Sotiris Ioannidis

Institute of Computer Science, Foundation for Research and Technology – Hellas,  
N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece  
{gvasil,sotiris}@ics.forth.gr

**Abstract.** In the ongoing arms race against malware, antivirus software is at the forefront, as one of the most important defense tools in our arsenal. Antivirus software is flexible enough to be deployed from regular users desktops, to corporate e-mail proxies and file servers. Unfortunately, the signatures necessary to detect incoming malware number in the tens of thousands. To make matters worse, antivirus signatures are a lot longer than signatures in network intrusion detection systems. This leads to extremely high computation costs necessary to perform matching of suspicious data against those signatures.

In this paper, we present GrAVity, a massively parallel antivirus engine. Our engine utilized the compute power of modern graphics processors, that contain hundreds of hardware microprocessors. We have modified ClamAV, the most popular open source antivirus software, to utilize our engine. Our prototype implementation has achieved end-to-end throughput in the order of 20 Gbits/s, 100 times the performance of the CPU-only ClamAV, while almost completely offloading the CPU, leaving it free to complete other tasks. Our micro-benchmarks have measured our engine to be able to sustain throughput in the order of 40 Gbits/s. The results suggest that modern graphics cards can be used effectively to perform heavy-duty anti-malware operations at speeds that cannot be matched by traditional CPU based techniques.

## 1 Introduction

The ever increasing amount of malicious software in today's connected world, poses a tremendous challenge to network operators, IT administrators, as well as ordinary home users. Antivirus software is one of the most widely used tools for detecting and stopping malicious or unwanted software. For an effective defense, one needs virus-scanning performed at central network traffic ingress points, as well as at end-host computers. As such, anti-malware software applications scan traffic at e-mail gateways and corporate gateway proxies, and also on edge compute devices such as file servers, desktops and laptops. Unfortunately, the constant increase in link speeds, storage capacity, number of end-devices and the sheer number of malware, poses significant challenges to virus scanning applications, which end up requiring multi-gigabit scanning throughput.

Typically, a malware scanner spends the bulk of its time matching data streams against a large set of known *signatures*, using a pattern matching algorithm. Pattern matching algorithms analyze the data stream and compare it

against a database of signatures to detect known malware. The signature patterns can be fairly complex, composed of different-size strings, wild-card characters, range constraints, and sometimes recursive forms. Every year, as the amount of malware grows, the number of signatures is increasing proportional, exposing scaling problems of anti-malware products.

To come up with the large signature sets, most approaches rely on the quickly, fast and accurate filtering of the “no-match” cases, based on the fact that the majority of network traffic and files is not supposed to contain viruses [9]. Other approaches are based on specialized hardware, like FPGAs and ASICs, to achieve high performance [15, 14]. Such hardware solutions are very efficient and perform quite well, however they are hard to program, complex to modify, and are usually tied to a specific implementation.

In contrast, commodity graphics processing units (GPUs) have been proven to be very efficient and highly effective at accelerating the pattern matching operations of network intrusion detection systems (NIDS) [26, 21, 27]. Driven by the ever-growing video game industry, modern GPUs have been constantly evolving to ever more powerful and flexible stream processors, specialized for computationally-intensive and highly parallel operations. The massive number of transistors devoted to data processing, rather than data caching and flow control, can be exploited to perform computations that up till now were handled by the CPU.

In this work, we explore how the highly parallel capabilities of commodity graphics processing units can be utilized to improve the performance of malware scanning programs and how they can assist and offload the CPU whenever possible.

From a high-level view, malware scanning is divided into two phases. First, all files are scanned by the GPU, using a combined DFA state machine that contain only a prefix from each signature. This results in identifying all potentially malicious files, but a number of clean files as well. The GPU then outputs a set of suspect matched files and the corresponding offsets in those files. In the second phase, all those files are rescanned using a full pattern matching algorithm.

The contributions of our work are:

- We have designed, implemented and evaluated a pattern matching algorithm on modern GPUs. Our implementation could be adapted to any other multi-core system, as well.
- We integrated our GPU implementation into ClamAV [12], the most popular and widely used open-source virus scanning software, proving that our solution can be used in the real-world.
- We developed and implemented a series of system level optimizations to improve end-to-end performance of our system.
- We implemented, experimented and analyzed our GPU-assisted virus scanning application with various configurations and we show that modern GPUs can effectively be used, in coordination with the CPU, to drastically improve the performance of anti-malware applications.

Our prototype implementation, called GrAVity, achieved a scanning throughput of 20 Gbits/s for binary files. This represents a speed-up factor of 100 from the single CPU-core case. Also, in special cases, where data is cached on the graphics card, the scanning throughput can reach 110 Gbits/s.

The rest of the paper is organized as follows. In Section 2, we present some background on general-purpose GPU (GPGPU) programming and introduce the related virus scanning architectures. The architecture and acceleration techniques are presented in Section 3. The performance analysis and evaluation are given in Section 4. The paper ends with an outline of related work in Section 5 and some concluding remarks in Section 6.

## 2 Background

In this section, we briefly describe the architecture of modern graphics cards and the general-purpose computing functionality they provide for non-graphics applications. We also discuss some general aspects of virus-scanning techniques.

### 2.1 GPU Programming

For our work we selected the NVIDIA GeForce 200 Series architecture, which offers a rich programming environment and flexible abstraction models through the Compute Unified Device Architecture (CUDA) SDK [18]. The CUDA programming model extends the C programming language with directives and libraries that abstract the underlying GPU architecture and make it more suitable for general purpose computing. In contrast with standard graphics APIs, such as OpenGL and DirectX, CUDA exposes several hardware features to the programmer. The most important of these features is the existence of convenient data types, and the ability to access the DRAM of the device card through the general memory addressing mode it provides. CUDA also offers highly optimized data transfer operations to and from the GPU.

The GeForce 200 Series architecture, in accordance with its ancestors GeForce 8 (G80) and GeForce 9 (G90) Series, is based on a set of multiprocessors, each of which contains a set of *stream processors* operating on SIMD (Single Instruction Multiple Data) programs. When programmed through CUDA, the GPU can be used as a general purpose processor, 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*, and is executed on the device as many different *threads* organized in *thread blocks*. Each multiprocessor executes one or more thread blocks, with each group organized into *warps*. A warp is a fraction of an *active group*, which is processed by one multiprocessor in one batch. Each of these warps contains the same number of threads, called the *warp size*, and is executed by the multiprocessor in a SIMD fashion. Active warps are time-sliced: A thread scheduler periodically switches from one warp to another to maximize the use of the multiprocessors computational resources.

Stream processors within a processor share an instruction unit. Any control flow instruction that causes threads of the same warp to follow different execution paths reduces the instruction throughput, because different execution paths have to be serialized. When all the different execution paths have reached a common end, the threads converge back to the same execution path.

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 [18]. The constant and texture memory accesses are cached, so a read from them costs much less compared to device memory reads, which are not being cached. The texture memory space is implemented as a read-only region of device memory.

## 2.2 Virus Scanning and ClamAV

ClamAV [12] is the most widely used open-source virus scanner. It offers client-side protection for personal computers, as well as mail and file servers used by large organizations. As of January 2010, it has a database of over 60,000 virus signatures, and consists of a core scanner library and various command-line utilities. The database includes signatures for non-polymorphic viruses in simple string format, and for polymorphic viruses in regular expression format (polymorphic signatures).

The current version of ClamAV uses an optimized version of the Boyer-Moore algorithm [3] to detect non-polymorphic viruses using simple fixed string signatures. For polymorphic viruses, on the other hand, ClamAV uses a variant of the classical Aho-Corasick algorithm [1].

The Boyer-Moore implementation in ClamAV, uses a *shift-table* to reduce the number of times the Boyer-Moore routine is called. At start up, ClamAV preprocess every signature and stores the shift value of every possible block (arbitrarily choosing a block size of 3 bytes) to initialize a shift table. Then, at any point in the input stream, ClamAV can determine if it can skip up to three bytes by performing a quick hash on them. ClamAV also creates a hash table based on the first three bytes of the signature and uses this table at run-time when the shift table returns a match. Since this algorithm uses hash functions on all bytes of a signature, it is only usable against non-polymorphic viruses.

The Aho-Corasick implementation uses a trie to store the automaton generated from the polymorphic signatures. The fixed string parts of each polymorphic signature are extracted, and are used to build a trie. At the scanning phase, the trie will be used to scan for all these fixed parts of each signature simultaneously. For example, the signature ‘‘495243\*56697275’’ contains two parts, ‘‘495243’’ and ‘‘56697275’’, which are matched individually by the Aho-Corasick algorithm. When all parts of a signature are found, ClamAV also verifies the order and the gap between the parts, as specified in the signature. To quickly perform a lookup in this trie, ClamAV uses a 256 element array for each node. In the general case, the trie has a variable height, and all patterns beginning with the same prefix are stored under the corresponding leaf node.

However, in order to simplify the trie construction, the height is restricted to be equal to the size of the shortest part in the polymorphic signatures, which is currently equal to two. Thus, the trie depth is fixed to two and all patterns are stored at the same trie level. During the scanning phase, ClamAV scans an input file and detects occurrences of each of the polymorphic signatures, including partially and completely overlapping occurrences. The Aho-Corasick algorithm has the desirable property that the processing time does not depend on the size or number of patterns in a significant way.

The main reason that ClamAV uses both Boyer-Moore and Aho-Corasick is that many parts in the polymorphic signatures are short, and they restrict the maximum shift distance allowed (bounded by the shortest pattern) in the Boyer-Moore algorithm. Matching the polymorphic signatures in Aho-Corasick avoid this problem. Furthermore, compared with the sparse automaton representation of the Aho-Corasick algorithm, the compressed shift table is a more compact representation of a large number of non-polymorphic signatures in fixed strings, so the Boyer-Moore algorithm is more efficient in terms of memory space.

### 3 Design and Implementation

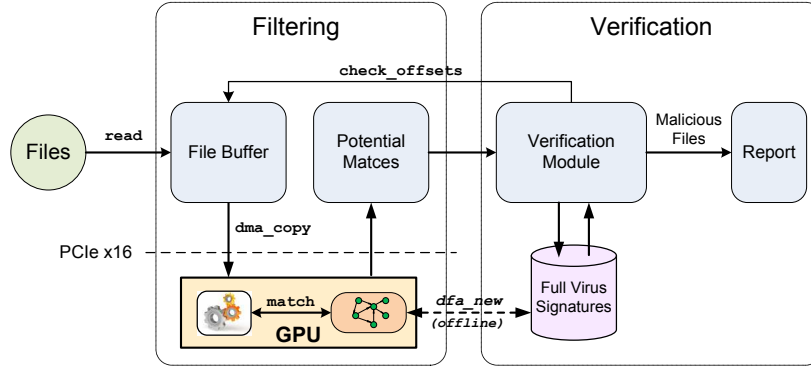
GrAVity utilizes the GPU to quickly filter out the data segments that do not contain any viruses. To achieve this, we have modified ClamAV, such that the input data stream is initially scanned by the GPU. The GPU uses a prefix of each virus signature to quickly filter-out clean data. Most data do not contain any viruses, so such filtering is quite efficient as we will see in Section 4.

The overall architecture of GrAVity is shown in Figure 1. The contents of each file are stored into a buffer in a region of main memory that can be transferred via DMA into the memory of the GPU. The SPMD operation of the GPU is ideal for creating multiple search engine instances that will scan for virus signatures on different data in a massively parallel fashion. If the GPU detects a suspicious virus, that is, there is prefix match, the file is passed to the verification module for further investigation. If the data stream is clean, no further computation takes place. Therefore, the GPU is employed as a first-pass high-speed filter, before completing any further potential signature-matching work on the CPU.

#### 3.1 Basic Mechanisms

At start-up, the entire signature set of ClamAV is preprocessed, to construct a deterministic finite automaton (DFA). Signature matching using a DFA machine has linear complexity as a function of the input text stream, which is very efficient. Unfortunately, the number of virus signatures, as well as their individual size is quite very large, so it may not be always feasible to construct a DFA machine that will contain the complete signature set. As the number and size of matching signatures increase, the size of the automaton also increases.

To overcome this, we chose to only use a portion from each virus signature. By using the first  $n$  symbols from each signature, the height of the corresponding



**Fig. 1.** GrAVity Architecture. Files are mapped onto pinned memory that can be copied via DMA onto the graphics card. The matching engine performs a first-pass filtering on the GPU and return potential true positives for further checking onto the CPU.

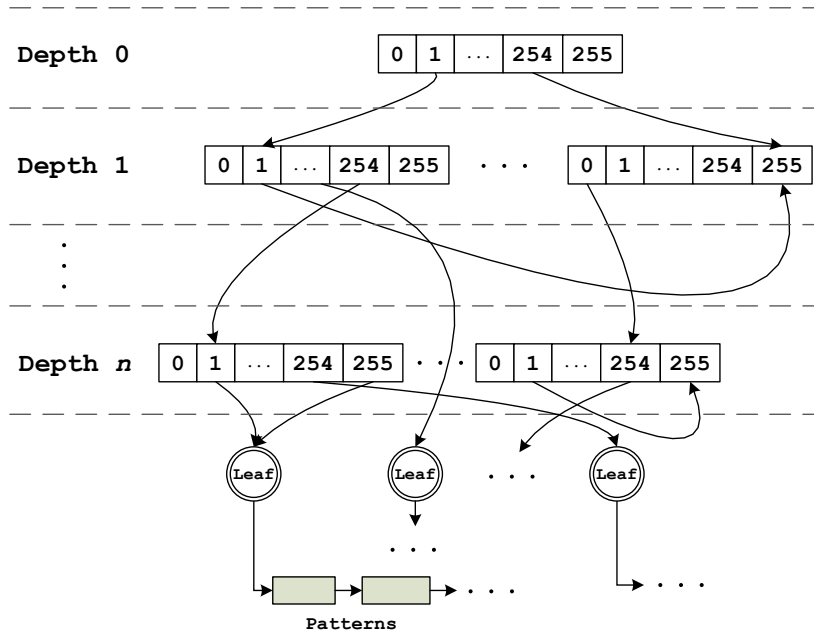
DFA matching machine is limited to  $n$ , as shown in Figure 2. In addition, all patterns that begin with the same prefix are stored under the same node, called *final node*. In case the length of the signature pattern is smaller than the prefix length, the entire pattern is added. A prefix may also contain special characters, such as the wild-characters  $*$  and  $?$ , that are used in ClamAV signatures to describe a known virus.

At the scanning phase, the input data will be initially scanned by the DFA running on the GPU. Obviously, the DFA may not be able to match an exact virus signature inside a data stream, as in many cases the length of the signature is longer than the length of the prefix we used to create the automaton. This will be the first-level filtering though, designed to offload the bulk of the work from the CPU, by drastically eliminating a significant portion of the input data that need to be scanned.

It is clear that the longer the prefix, the fewer the number of false positives at this initial scanning phase. As we will see in Section 4, using a value of 8 for  $n$ , can result to less than 0.0001% of false positives in a realistic corpus of binary files.

### 3.2 Parallelizing DFA matching on the GPU

During scan time, the algorithm moves over the input data stream one byte at a time. For each byte, the scanning algorithm moves the current state appropriately. The pattern matching is performed byte-wise, meaning that we have an input width of 8 bits and an alphabet size of  $2^8 = 256$ . Thus, each state will contain 256 pointers to other states, as shown in Figure 2. The size of the DFA state machine is thus  $|\#States| * 1024$  bytes, where every pointer occupies 4 bytes of storage.



**Fig. 2.** A fragment of the DFA structure with  $n$  levels. The patterns beginning with the same prefix are stored under the same final node (leaf).

If a final-state is reached, a potential signature match has been found. Consequently, the offset where the match has been found is marked and all marked offsets will be verified later by the CPU. The idea is to quickly weed-out the dominant number of true negatives using the superior performance and high parallelism of the GPU, and pass on the remaining potential true positives to the CPU.

To utilize all streaming processors of the GPU, we exploit its data parallel capabilities by creating multiple threads. An important design decision is how to assign the input data to each thread. The simplest approach would be to use multiple data input streams, one for each thread, in separate memory areas. However, this will result in asymmetrical processing effort for each processor and will not scale well. For example, if the sizes of the input streams vary, the amount of work per thread will not be the same. This means that threads will have to wait, until all have finished searching the data stream that was assigned to them.

Therefore, each thread searches a different portion of the input data stream, at the matching phase. To best utilize the data-parallel capabilities of the GPU, we create a large number of threads that run simultaneously. Our strategy splits the input stream in distinct chunks, and each chunk is processed by a different thread. Figure 3 shows how each GPU thread scans its assigned chunk, using

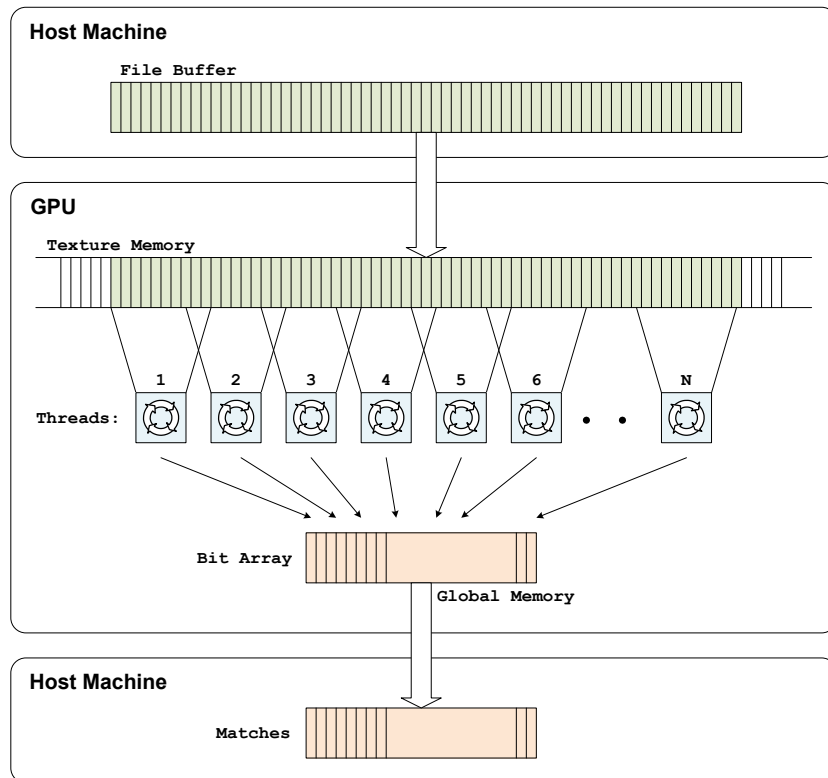
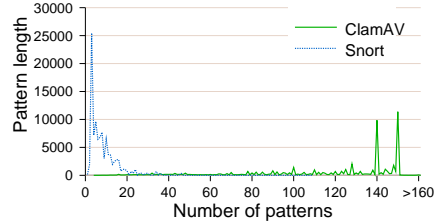


Fig. 3. Pattern matching on the GPU.

the underlying DFA state table. Although they access the same automaton, each thread maintains its own state, eliminating any need for communication between them.

A special case, however, is for patterns that may span across two or more different chunks. The simplest approach for fixed string patterns, would be to process in addition,  $n$  bytes, where  $n$  is the maximum pattern length in the dictionary. Unfortunately, the virus patterns are usually very large, as shown in Figure 4 for the ClamAV, especially when compared with patterns in other pattern matching systems like Snort. Moreover, a regular expression may contain the wild card character  $*$ , thus the length of the patterns may not be determined. To solve this problem, we used the following heuristic: each thread continues the search up to the following chunk (which contains the consecutive bytes), until a fail or final-state is reached. While matching a pattern that spans chunk boundaries, the state machine will perform regular transitions. However, if the state machine reaches a fail or final-state, then it is obvious that there is no need to process the data any further, since any consecutive patterns will be matched



**Fig. 4.** ClamAV pattern length distribution.

by the thread that was assigned to search the current chunk. This allows the threads to operate independently and avoid any communication between them, regarding boundaries in the input data buffer.

Every time a match is found, it is stored to a bit array. The size of the bit array is equal to the size of the data that is processed at concurrently. Each bit in the array represents whether a match was found in the corresponding position. We have chosen the bit array structure, since it is a compact representation of the results, even in the worst case scenario where a match is found at every position.

### 3.3 Optimized Memory Management

The two major tasks of DFA matching, is determining the address of the next state in the state table, and fetching the next state from the device memory. These memory transfers can take up to several hundreds of nanoseconds, depending on the traffic conditions and congestion.

Our approach for hiding memory latencies is to run 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 tested GrAVity to determine how the computational throughput is affected by the number of threads. As discussed in Section 4.2 the memory subsystem is best utilized when there is a large number of threads, running in parallel.

Moreover, we have investigated storing 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, in contrast to global memory, where the access patterns must be coalesced. This feature can be very useful for algorithms like DFA matching, which exhibit irregular access patterns across large data sets. Furthermore, texture fetches are cached, increasing the performance when read operations preserve locality. As we will see in Section 4.2, the usage of texture memory can boost the computational throughput up to a factor of two.

### 3.4 Other Optimizations

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

The page-locked memory offers better performance, as it does not get swapped (i.e. non-pageable memory). Furthermore, it can be accessed directly by the GPU through Direct Memory Access (DMA). Hence, the usage of page-locked memory improves the overall performance, by reducing the data transferring costs to and from the GPU. The contents of the files are read into a buffer allocated from page-locked memory, through the CUDA driver. The DMA then, transfers the buffer from the physical memory of the host, to the texture memory of the GPU.

To further improve performance, we use a large buffer to store the contents of many files, that is transferred to the GPU in a single transaction. The motivation behind this feature, is that the matching results will be the same, whether we scan each file individually or scanning several files back-to-back, all at once. This results in a reduction of I/O transactions over the PCI Express bus.

## 4 Performance Evaluation

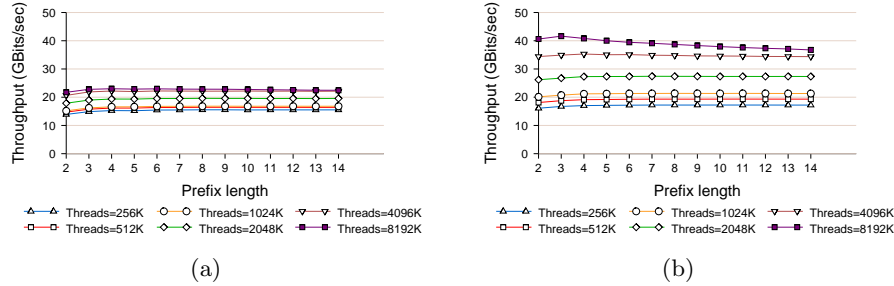
In this section, we evaluate our prototype implementation. First, we give a short description of our experimental setup. We then present an overall performance comparison of GrAVity and ClamAV, as well as detailed measurements to show how it scales with the prefix length and the number of threads that are executing on the GPU.

### 4.1 Experimental Environment

For our experiment testbed, we used the NVIDIA GeForce GTX295 graphics card. The card consists of two PCBs (Printed Circuit Board), each of which is equipped with 240 cores, organized in 30 multiprocessors, and 896MB of GDDR3 memory. Our base system is equipped with two Intel(R) Xeon(R) E5520 Quad-core CPUs at 2.27GHz with 8192KB of L2-cache, and a total of 12GB of memory. The GPU is interconnected using a PCIe 2.0 x16 bus.

We use the latest signatures set of ClamAV (main v.52, released on February 2010). The set consists of 60 thousand string and regular expression signatures. As input data stream, we used the files under `/usr/bin/` in a typical Linux installation. The directory contains 1516 binary files, totalling about 132MB of data. The files do not contain any virus, however they exercise most code branches of GrAVity.

In all experiments we conducted, we disregarded the time spent in the initialization phase for both ClamAV and GrAVity. The initialization phase includes the loading of the patterns and the building of the internal data structures, so there is no actual need to include this time in our graphs.



**Fig. 5.** Sustained throughput for varying signature prefix. Higher number of threads achieve higher performance as memory latencies are hidden. We demonstrate the effect of different GPU memory types on performance. (a) uses global device memory to store the DFA state table, where (b) uses texture memory.

## 4.2 Microbenchmarks

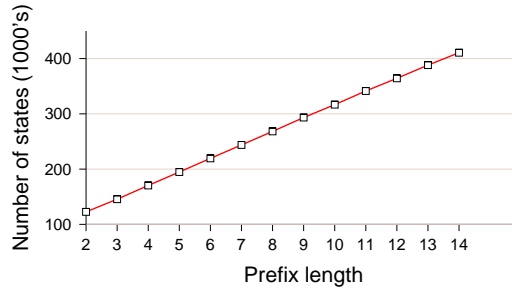
Figure 5 shows the matching throughput for varying signature prefix lengths. We explore the performance that different types of memory can provide, by using global device and texture memory respectively to store the DFA state table. The horizontal axis shows the signature prefix length. We also repeated the experiment using different number of threads. As the number of threads increases, the throughput sustained by the GPU also increases. When using eight millions threads, which is the maximum acceptable number of threads for our application, the computational throughput raises to a maximum of 40 Gbits/s.

Comparing the two types of memory available in the graphics card, we observe that the texture memory significantly improves the overall performance by a factor of two. The irregularity of memory accesses that DFA matching exhibits, can be partially hidden when using texture memory. Texture memory provides a random access model for fetching data, in contrast with global memory where access patterns have to be coalesced. Moreover, texture fetches are cached, which offers an additional benefit.

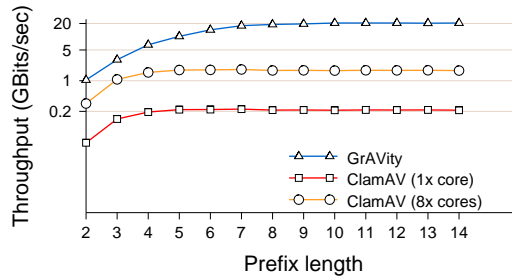
The total memory requirements for storing the DFA, independently of the memory type, is shown in Figure 6. We observe that the total number of states of the DFA machine is growing linearly to the length of the prefix. Using a value of 14 as a prefix length, results in a DFA machine that holds about 400 thousands states. In our DFA implementation this is approximately 400MB of memory — each state requires 1KB of memory.

## 4.3 Application Performance

In this section, we evaluate the overall performance of GrAVity. Each experiment was repeated a number of times, to ensure that all files were cached by the operating system. Thus, no file data blocks were read from disk during our



**Fig. 6.** Memory requirements for the storage of the DFA as a function of the signature prefix length.



**Fig. 7.** Performance of GrAVity and ClamAV. We also include the performance number for ClamAV running on 8 cores. The CPU-only performance is still an order of magnitude less than the GPU-assisted. The numbers demonstrate that additional CPU cores offer less benefit than that of utilizing the GPU.

experiments. We have verified the absence of I/O latencies using the `iostat(1)` tool.

**Throughput** In this experiment we evaluate the performance of GrAVity compared to vanilla ClamAV. Figure 7 shows the throughput achieved for different prefix lengths. The overall throughput increases rapidly, raising at a maximum of 20 Gbits/s. A plateau is reached for a prefix length of around 10.

As the prefix length increases, the number of potential matches decreases, as shown in Figure 9. This results to lower CPU post-processing, hence the overall application throughput increases. In the next section, we investigate in more detail the breakdown of the execution time.

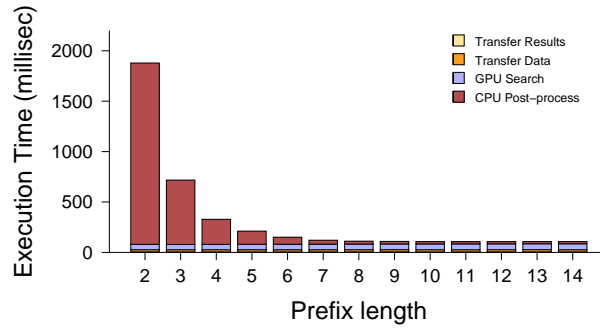


Fig. 8. GrAVity execution time breakdown.

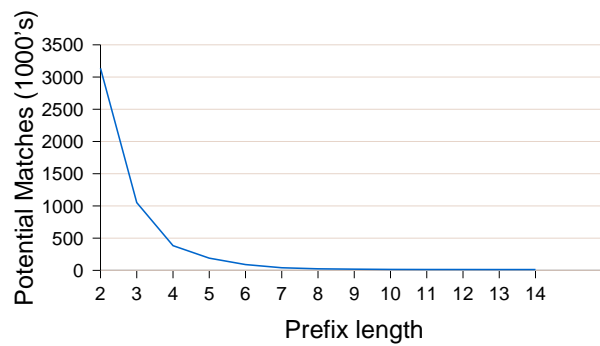
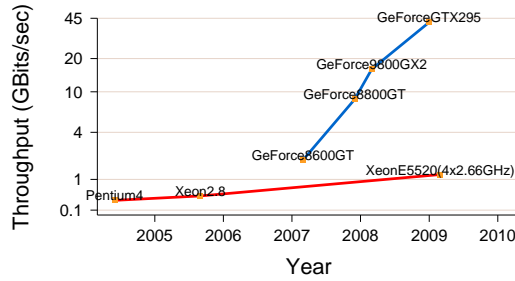


Fig. 9. Number of matches as a function of the signature prefix length.

**Execution Time Breakdown** We measure the execution time for data transfers, result transfers, CPU and GPU execution. We accomplish this by adding performance counters before each task.

As expected, Figure 8 shows that for small prefix sizes most of the time is dominated by the cost of the CPU, verifying the possible matches reported back by the GPU. For example, for a prefix length equal to 2, approximately 95% of the total execution time is spent on the CPU to validate the potential matches. For a prefix length equal to 14, the corresponding CPU time results in just 20% of the total execution time, and in actual time signifies a reduction of *three orders of magnitude*, while the GPU consumes 54% of the total execution. As the prefix length increases, this overhead decreases and the GPU execution time becomes the dominant factor. For verification, in Figure 9 we plot the number of potential matches reported in accordance with the signature prefix length.



**Fig. 10.** Performance sustained by our pattern matching implementation on different generation of GPU and CPU models.

#### 4.4 Scaling Factor

To measure how our GPU pattern matching implementation has improved during the evolution of GPU models, we used three additional older-generation graphics cards. Specifically, we utilized a GeForce 8600GT, which was released early on March 2007, a GeForce 8800GT released on December 2007, and a GeForce 9800GX2 that released 4 months later, on March 2008.

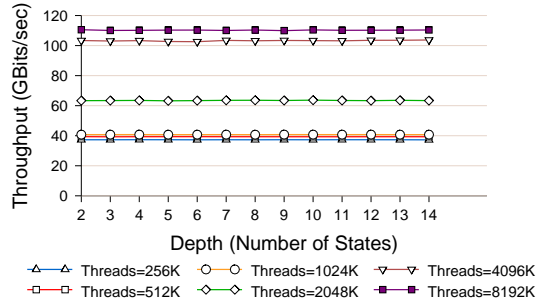
Figure 10 shows that in less than two years, the computational throughput has raised 20 times, from about 2 Gbits/sec to over 40 Gbits/sec. For comparison reasons, we also calculated and included the respective numbers of various generations of CPUs.

#### 4.5 Peak Performance

In the final experiment we explore the ideal performance our GPU implementation can achieve. For this reason, we created a large file containing the NULL character, to ensure that no state transitions will be performed at the matching phase. The automaton will remain always at the same state, which will be cached. Moreover, no matches will be reported, that would trigger an expensive memory write at the global device memory. In this “best-case” scenario, our throughput reached an order of 110 Gbits/s. This demonstrated the top end performance the hardware can support. GrAVity’s end-to-end performance reaches a very respectable 20% of this upper bound.

## 5 Related Work

Multi-pattern matching algorithms is one of the core operations used by applications in many domains. In the networking area, the most important applications, that primarily rely on pattern matching, are intrusion detection systems and malware scanners.



**Fig. 11.** Peak performance sustained by our pattern matching implementation on the GPU.

Many approaches rely on the hardware implementation of pattern matching algorithms, like FPGAs [20, 22, 10, 2], CAMs [24, 29, 23] and Network Processors [6, 7]. Most of these studies have focused primarily on network intrusion detection systems, which are quite different from virus scanning applications [9].

Recently, however, several efforts have been made to improve the performance of ClamAV [16, 9, 15, 14]. Many approaches rely on the simple, fast and accurate filtering of the input data stream, as software implementations running on generic processors [16, 9], or more complex approaches using specialized hardware [15, 14].

Recent software implementations have adapted Bloom filters for use in virus scanning as a first-level filter before the exact pattern matching algorithm occurs [9, 5]. A fragment of constant length is extracted from every signature and inserted into a Bloom filter. At the scanning phase, a window of the same size slides over the files to be examined, and its content at every position is tested against the filter. A Bloom filter is the most compact structure that can store a dictionary and is used to determine whether a string belongs to that dictionary or not. A major drawback of Bloom filters, however, is that they cannot be used for regular expressions matching. A possible solution is to select an invariant fragment (i.e. a fixed byte sequence) from a wild-card containing signature and put it in the filter. Unfortunately, the fact that the fragments have to be of the same length, will shorten the hashing window to the shortest signature or fragment, and will increase the false positive rate. Several approaches have been used Bloom filters efficiently in specialized hardware, for example with FPGAs [8, 17, 4]. Hardware implementations provide better performance, although with a high, and often prohibitive, cost for many organizations.

Besides specialized hardware solutions, commodity multi-core processors have begun gaining popularity, primarily due to their increased computing power and low cost. It has been shown that fixed-string pattern matching implementations on SPMD processors, such as the IBM Cell processor, can achieve a computational throughput of up to 2.2 Gbits/s [19], while regular expression matching

up to 7.5 Gbits/s [13]. In the context of network intrusion detection systems, graphics processors have been used to accelerate their performance [26, 27, 21, 11, 28, 25]. Specifically, work in [26, 27] significantly improved the performance of Snort by offloading the string searching and regular expression matching operations to the GPU. The work in this paper, exploits and extends some of those ideas and applies them in a hybrid, GPU-CPU malware detection architecture, with a drastic improvement in performance.

## 6 Conclusions

In this paper, we presented GrAVity, a massively parallel antivirus engine that utilizes the GPU to offload the bulk of pattern and regular expression matching from a popular antivirus system. Our system exploits the highly threaded architecture of modern graphics processors, as well as the embarrassingly parallel nature of virus scanning to achieve end-to-end throughput in the order of 20 Gbits/s. This result is 100 times faster than the unmodified ClamAV running on a modern CPU. Our benchmarks also showed that our approach completely offloads the CPU and frees it to perform other tasks. Finally, our micro-benchmarks showed that it is possible to achieve throughput in the order of 40 Gbits/s in cases where data is pre-cached on the graphics card, showing that solving data transfer bottlenecks can lead to doubling of performance.

To achieve such high performance, we tuned our system and performed a number of optimizations. Since virus signatures are both very long and more numerous compared to other signature matching systems, like network intrusion detection systems, we build our engine as a pre-filter, that uses prefixes of the actual signatures. These prefixes are used to create the DFAs used in the actual pattern matching on the GPU. Our architecture also takes advantage of the physical memory hierarchies of graphics processors, as well as, bulk data transfers using DMA.

As future work we plan to investigate how to port our engine to commercial antivirus software, as well, other tools such as antispymware. In terms of architecture, we plan to overlap GPU and CPU matching phase, as right now our system is serialized in that respect. Finally we plan on utilizing multiple GPUs instead of a single one. Modern motherboards, such as the one we used in our evaluation, support multiple GPUs on the PCI Express bus. In our case it would be possible to utilize up to four such cards. Such a system would require a more thorough investigation of communication and synchronization between multiple GPUs.

## Acknowledgments

This work was supported in part by the Marie Curie Actions – Reintegration Grants project PASS. Giorgos Vasiliadis and Sotiris Ioannidis are also with the University of Crete.

## References

1. A. V. Aho and M. J. Corasick. Efficient String Matching: an Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.
2. Z. K. Baker and V. K. Prasanna. Time and area efficient pattern matching on FPGAs. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*, pages 223–232, New York, NY, USA, 2004. ACM.
3. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the Association for Computing Machinery*, 20(10):762–772, October 1977.
4. F. Braun, J. Lockwood, and M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable hardware. *IEEE Micro*, 22(1):66–74, 2002.
5. S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. SplitScreen: Enabling efficient, distributed malware detection. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, Apr. 2010.
6. C. R. Clark, W. Lee, D. E. Schimmel, D. Contis, M. Kon, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, editors, *Network Processor Design: Issues and Practices, Volume 3*, pages 99–118. Morgan Kaufmann, 2005.
7. W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. SafeCard: a Gigabit IPS on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Hamburg, Germany, September 2006.
8. S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.
9. O. Erdogan and P. Cao. Hash-AV: Fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks*, 2(1/2):50–59, 2007.
10. J. T. L. Ho and G. G. Lemieux. PERG-Rx: a hardware pattern-matching engine supporting limited regular expressions. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 257–260, New York, NY, USA, 2009. ACM.
11. 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 *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pages 62–67, 25-28 2008.
12. T. Kojm. Clamav. <http://www.clamav.net/>.
13. F. Kulishov. DFA-based and SIMD NFA-based regular expression matching on Cell BE for fast network traffic filtering. In *SIN '09: Proceedings of the 2nd international conference on Security of information and networks*, pages 123–127, New York, NY, USA, 2009. ACM.
14. Y.-D. Lin, P.-C. Lin, Y.-C. Lai, and T.-Y. Liu. Hardware-Software Codesign for High-Speed Signature-based Virus Scanning. *IEEE Micro*, 29(5):56–65, 2009.
15. Y.-D. Lin, K.-K. Tseng, T.-H. Lee, Y.-N. Lin, C.-C. Hung, and Y.-C. Lai. A platform-based SoC design and implementation of scalable automaton matching for deep packet inspection. *J. Syst. Archit.*, 53(12):937–950, 2007.
16. Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

17. J. Moscola, J. Lockwood, R. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–38, Napa, CA, USA, Apr. 2003.
18. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 3.0. [http://developer.download.nvidia.com/compute/cuda/3\\_0/toolkit/docs/NVIDIA\\_CUDA\\_ProgrammingGuide.pdf](http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf).
19. D. P. Scarpazza, O. Villa, and F. Petrini. Exact multi-pattern string matching on the cell/b.e. processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 33–42, New York, NY, USA, 2008. ACM.
20. R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM01)*, 2001.
21. 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*, 2009.
22. T. Song, W. Zhang, D. Wang, and Y. Xue. A Memory Efficient Multiple Pattern Matching Architecture for Network Security. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 166–170, 13-18 2008.
23. I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 258–267, Washington, DC, USA, 2004. IEEE Computer Society.
24. I. Sourdis, D. N. Pnevmatikatos, and S. Vassiliadis. Scalable multigigabit pattern matching for packet inspection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):156–166, 2008.
25. A. Tumeo, O. Villa, and D. Sciuto. Efficient pattern matching on GPUs for intrusion detection systems. In *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*, pages 87–88, New York, NY, USA, 2010. ACM.
26. G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.
27. G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of 12th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.
28. C. Wu, J. Yin, Z. Cai, E. Zhu, and J. Chen. A Hybrid Parallel Signature Matching Model for Network Security Applications Using SIMD GPU. In *APPT '09: Proceedings of the 8th International Symposium on Advanced Parallel Processing Technologies*, pages 191–204, Berlin, Heidelberg, 2009. Springer-Verlag.
29. F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit Rate Packet Pattern-Matching Using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP '04)*, pages 174–183, Washington, DC, USA, October 2004. IEEE Computer Society.