Detecting Computer Viruses using GPUs

Alexandre Nuno Vicente Dias
Instituto Superior Técnico, No. 62580
alexandre.dias@ist.utl.pt

Abstract

Anti-virus software is the main defense mechanism against malware, which is becoming more common and advanced. A significant part of the virus scanning process is dedicated to scanning a given file against a set of virus signatures. As it is important that the overall scanning process be as fast as possible, efforts must be done to minimize the time spent in signature matching. Recently, graphics processing units have increased in popularity in high performance computation, due to their inherently parallel architecture. One of their possible applications is performing matching of multiple signatures in parallel. In this work, we present details on the implemented multiple string searching algorithm based on deterministic finite automata which runs on a graphics processing unit. Due to space concerns inherent to DFAs our algorithm only scans for a substring of each signature, thereby serving as a high-speed pre-filtering mechanism. Multiple optimizations were implemented in order to increase its performance. In our experiments with sets of test files, the implemented solution was found to have a speedup of around 28 when compared to the pattern matching portion of ClamAV, an open-source anti-virus engine.

Keywords: network security, virus, string matching, GPGPU Computing, CUDA

1. Introduction

As technology gets more advanced, malware writers are using a whole new set of tools to build malware that is more efficient and harder to detect. Malware is also becoming more prevalent in the web: For instance, anti-virus companies can receive malware samples at a rate of about one sample every 5 seconds. Due to this emergence, both IT professionals and regular users alike are facing various challenges when it comes to protection. An example of an effort to increase the defenses against viruses is the cooperation between anti-virus companies to detect a new variation of a virus (or a new virus itself): samples of recent viruses are placed into a shared repository to which the anti-virus companies have access. However, there is still a given time window (from the time where the virus first appears to the time where an anti-virus company updates their product) in which users are vulnerable.

Anti-virus products try to minimize the chance of infection of a machine, employing various techniques to do so. For instance, they might take a sample and analyze their behavior in run-time, in order to check if the sample does anything that it is not supposed to. However, this run-time analysis should not be the first thing to do to a newly arrived sample, as it is very time consuming. The first step in identifying a virus is usually done by scanning a file, and matching its body to a set of rules or patterns, called signatures. Each signature can correspond to a different virus or strain of one, and if it occurs in the body of a file, then we know that such file is malicious.

Anti-virus products and intrusion detection systems (which scan network packets against a rule set), employ various algorithms to speed up the scanning process. Such algorithms are part of a problem called string matching, which as the name indicates, is the problem of finding a given string in a text. The products also have a signature set, so they must scan for multiple strings inside the same text (which corresponds to the body of a file). This problem is a subclass of the aforementioned one, and it is called multi-pattern matching. Many techniques have been implemented to try to speedup multi-pattern matching, some of them having success in doing so. These techniques are mainly small changes to the algorithms, but some of them focus on implementing the algorithms on specific hardware, rather than on CPUs. A common choice in hardware devices to implement algorithms in is the Graphics Processing Unit (GPU).

The utilization of GPUs for general purpose computing is rapidly gaining popularity, due to the massive parallelism inherent to these devices. GPUs are specialized for computationally intensive and highly parallel tasks (namely graphics rendering), and as so, they are designed such that more transistors are devoted to data processing rather than data caching and flow control. A factor that helped GPUs gain

MD5 Hashes	Basic Strings	Regular Expressions	Others	Total
1,215,718	88,391	9,421	7,484	1,321,014

Table 1. Distribution of signature types in ClamAV.

execution time in seconds	number of times called	percentage of execution time	function name
37.16	2592	38.5%	cli_ac_scanbuff
36.68	2566	38%	cli_bm_scanbuff

Table 2. Results of profiling ClamAV by scanning a large file.

popularity in general purpose computing was the release of software development kits by NVIDIA and AMD, which are to be used in their respective GPUs. These SDKs provide much needed help for developers wanting to start programming in GPUs.

Currently, GPUs are used for a whole range of applications, from biomedical calculations to brute-forcing MD5 hashes. When it comes to multi-pattern matching, research efforts have proved that GPUs can bring significant speedups to the process. It is due to such conclusions, and due to the parallel nature of GPUs, that we have decided to adopt them in order to speedup the multi-pattern matching part of an anti-virus engine.

Considering that pattern matching is a process that can potentially be sped up, we decided to have as our goal the improvement of such a component in an anti-virus product (we chose to use ClamAV due to being open-source), so as to decrease the execution time of the overall virus scanning process.

We have found that our system provided a good improvement: speed-ups ranged from around 12 to 28 on different sets of files. On some other sets of files the speedup was lower: if, for example, our system was only to scan a single large file, then it would have an execution time larger than ClamAV.

2. Background

2.1 ClamAV

ClamAV is an open-source anti-virus engine, used by many people in desktop computers, servers and mail gateways alike. The signature set of ClamAV contains about one million signatures in total. However, they are not all of the same format. The set is divided in various types, being the most prominent MD5 hashes and static strings. As of April 2012, the distribution of the signature types was as indicated by Table 1 (the “others" category refers to combinations of signatures, signatures related to archives, signatures related to spam e-mails, etc).

As we can see, more than 90% of the signatures are MD5 hashes of files. However, MD5 hashes are calculated by simply applying a hash function to a whole file. Scanning against this type of signatures does not take a significant amount of time, in contrast to scanning for regular expressions and basic strings, which must be scanned for inside the file. This is proven by profiling ClamAV: we did so by scanning a large file (around 600 MBytes), and found that two functions stood out from the others (considering their execution time). The results are shown in Table 2.

All of the other functions that are not shown in the table had an execution time below 2 seconds (except for a function related to the database, that took 13 seconds to execute). As such, the biggest fraction of the running time that we can optimize is the one related to the two functions shown above. These two functions shown in the table are related to the process of pattern matching, which is done by an anti-virus product in order to determine if a given file contains a known signature (which corresponds to a virus).

As just mentioned, ClamAV’s signature set is quite diverse, but some subset of it needs processing that is computationally expensive.

ClamAV’s signatures can be split into two different groups: signatures with wildcards, and signatures without them. Signatures without wildcards are not especially problematic: despite being fairly large both in number as in length, these can be applied to pattern matching algorithms which don’t have to worry about matching regular expressions (such algorithms take advantage of that fact to have better overall runtime complexity). Signatures with wildcards however, can be very problematic.

Due to the complexity of modern viruses, these wildcards are wildly used in signature sets. In the ClamAV signature set used (the same as mentioned above), we have found 5779 “*" wildcards (which correspond to the “.*" wildcard in the POSIX notation). So not only are such wildcards highly complex in terms of automaton representation, but they also exist in a high number. This leads to automata-based solutions having problems with ClamAV’s signature set.

2.2 GPGPU

Graphics Processing Units (GPUs) are hardware units specially designed to perform graphics operations (or rather, graphics rendering), which are very important for video games. Graphics rendering is by nature a parallel operation, as a value of a given pixel does not depend on the values of other pixels. Due to this, GPUs have been designed to perform these parallel operations as fast as possible. They are essentially a highly parallel, multithreaded, many core processor. As they evolve, they become more and more powerful, and thus also more and more attractive to programmers in need of implementing a highly parallel algorithm.

2.2.1 Comparing CPUs and GPUs

A few years ago, CPUs increased their performance by increasing their core frequency. However, increasing the frequency is limited by some physical restrictions such as heat and power consumption. Nowadays, CPUs' performance is increased by adding more cores. On computing hardware, the focus is slowly shifting from CPUs that are optimized to minimize latency towards GPUs that are optimized to augment the total throughput. CPUs that are optimized towards latency use the Multiple Instruction Multiple Data (MIMD) model, meaning that each of its cores works independently from the others, executing instructions for several processes.

GPUs on the other hand use a Single Instruction Multiple Data model, where each multiple processing elements execute the same instruction on multiple data simultaneously.

In an NVIDIA GPU, processors execute groups of 32 threads, called warps [1]. Individual threads in the same warp start together, but can then execute and branch independently.

Although threads can generally execute independent instructions, threads belonging to the same warp can only execute one common instruction at a time. If any thread in a warp diverges, then the threads in that warp are serially executed for each of the independent instructions, with barriers being inserted after the execution of such instructions so that threads are then re-synchronized. Thus, efficiency is achieved when all 32 threads of a warp share the same execution path.

This kind of divergence that forces hardware to serially execute instructions can only exist inside a warp, as different warps are running on different multiprocessors.

Another main difference between CPUs and GPUs are threads and how they are organized. On a typical CPU, a small number of threads can run concurrently on a core. On a GPU however, thousands can run concurrently per multiprocessor.

Switching between threads on a CPU can also take some significant time, whereas on a GPU this switching is done seamlessly and almost instantly. This is due to the fact that on a CPU, the operation system controls the thread switching behavior. On a GPU, thousands of threads are put into a queue (in groups of warps). If the hardware must wait on one warp of threads (because for instance it needed to perform a slow access to memory), then it just swaps that warp with another one that is ready to execute.

This is mainly possible due to the large amount of registers present on a GPU: separate registers are allocated to all of the threads, so no swapping of registers or state needs to be done when thread switching occurs (unlike a CPU, which has less registers and as such their state must be saved in thread switches).

2.2.2 Hardware Architecture

When it comes to memory, an NVIDIA GPU has different types of physical memory, which can also be virtually abstracted.There are three main memories on a GPU: device memory and shared memory. Device memory is the largest memory, but also the one with the largest access latency. It is equivalent to the DRAM of a CPU. Despite being slow, its main use comes from the fact that it is the only physical memory on the GPU that can be populated by the CPU before a kernel is launched.

Shared memory is a fast memory that is present on every multiprocessor, and it is shared by all of that multiprocessor's threads. Its speed comes from the fact that it is present on-chip rather than off (in other words, shared memory does not need a special bus to be accessed, unlike global memory): no special buses need to be used to access it. Uncached shared memory latency can be up to 100 times lower than the global memory's latency (as long as there are no conflicts in the thread accesses). In order to achieve such low latency, shared memory is divided into banks. Any memory read or write access made of some number of memory addresses that are in their own distinct memory banks can therefore be done simultaneously, effectively increasing the bandwidth. However, if different threads need to access a same bank, then conflicts arise and accesses need to be serialized (not a big issue as it remains on-chip).

As for the virtual abstractions, they are all related to the device memory. Two important ones are global and texture memory: Global memory is quite slow. When a given warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more memory transactions, depending on the size of the word accessed and the requested memory addresses. If requested memory addresses are far from each other, the warp must effectively perform a single transaction for each memory address, which causes them to be se-

rialized (and unused words to be transferred). If they however are completely sequential, then a single transaction can usually be done for all of the addresses: the hardware performs that transaction and then broadcasts the values to the requesting threads. In a worst-case scenario with a small number of threads and a very random access pattern, the bottleneck of the execution becomes the global memory: threads are continuously waiting for others to finish their memory accesses so that they can perform their own.

The texture memory is optimized for 2-dimensional spatial locality, and has a dedicated cache. Threads that read texture addresses close together will achieve good performance. While it is read-only, texture memory can be greatly advantageous for algorithms that exhibit spatial locality in their access patterns.

3. Related Work

Pattern matching algorithms are dedicated to the problem of finding a pattern or a set of patterns on a text, and they can be divided in two classes: single and multi-pattern matching algorithms.

Single-pattern matching algorithms perform matching of one pattern at a single time. For k patterns to be matched, the algorithm must be repeated k times.

Multiple-pattern matching algorithms, in contrast to single-pattern ones, can perform matching of a set of patterns in a single run. For k patterns to be matched, the algorithm only runs once.

Some multi-pattern matching algorithms are based on finite automata. Finite automata used in pattern matching algorithms are either non-deterministic (NFA) or deterministic (DFA). Both kinds are augmented, directed graphs that have an alphabet, a set of states, and a transition function (that for each sate, computes a next state, given a character from the alphabet).

Matching is done by, starting at the initial state, computing the value of the transition function for the current input character, and feed that value to be the next state at the next iteration (where the transition function is again executed but for the next input character). When one of the states given by the transition function is final, the algorithm reports that there was a match at the current input position.

NFA are automata that have no restrictions on their transitions. A character can have many transitions out of the same state, and epsilon - the empty character - is a possible transition (so it is always taken each time it is present).

The execution model of NFA uses some auxiliary data when processing the input text. As in an NFA the system may be at more than one state at a single time, usually a vector is kept, which represents a list of active states. Each time one of the active states is final, the algorithm reports a match for the possible final states.

Matching on an NFA is a somewhat different (and more complex) process than matching on a DFA: instead of only following a single transition on each input character, multiple transitions are followed, which leads to several states being active at any one time.

A DFA on the other hand, has no transitions by the epsilon character, and each state only has one transition by a given character. This simple model has two main advantages: first, matching is straightforward and fast, as it requires only a single table lookup (or pointer dereference) per input character. Second, DFAs are composable, meaning that a set of DFAs can be composed into a single composite DFA (eliminating the need to transverse multiple DFAs instead of a single one).

Unfortunately, DFAs often do not interact well when combined, yielding a composite DFA whose size may be exponential in input and often exceeds available memory.

They do however provide a big advantage over NFAs: as computers are deterministic machines, NFAs are harder to simulate in them (due to the possibility of being in more than one state at the same time), which causes larger execution times and greater implementation complexity.

Regular expressions are gaining widespread use in applications which use searches with expressive and flexible queries (virus scanning and intrusion detection are examples of such applications). Using Thompson's algorithm [8], an NFA can be converted from a regular expression, and then converted to an equivalent DFA.

As regular expression use becomes more and more common in packet and file scanning applications, it is imperative that regular expression matching be as fast as possible, as to keep up with line-speed packet header processing. The inefficiency in regular expression matching is largely due to the fact that the current matching solutions do not efficiently handle some complex features of regular expressions used in such applications: many patterns use multiple wildcards (such as '.' and "*") and some patterns can contain over ten such wildcards.

Some approaches have been proposed to counter the state-space explosion that affects DFAs constructed from regular expressions with complex wildcards: HFA [3], hybrid finite automata, split the DFA under construction at the point where state-space explosion would happen. Lazy DFAs [5] keep a subset of the DFA that matches the most common patterns in memory; for uncommon strings, they extend the subset from the corresponding NFA at runtime. This causes the Lazy DFA to be smaller than the corresponding complete DFA, while providing good performance for common patterns. However, a malicious sender can input widely different patterns into the system, causing it to be dynamically rebuilding a DFA (and thus slowing down the overall system).

3.1 Pattern Matching on GPUs

Smith et al. [7] implemented a pattern matching system on a GPU based on XFA, which are automata which use less memory than DFAs but require additional computation at each state. While XFA might seem like a good improvement, they introduce serious challenges for SIMD architectures. Having more variable updates involve memory accesses in a manner different than the access patterns used for traversing an automaton. In a GPU memory accesses should be as few and simple as possible, as they pose one of the most common bottlenecks of an algorithm.

In their implementation, the authors found that the percentage of SIMD instructions that diverged was more than 50%. Modern GPUs do support divergence, but at a (sometimes large) performance cost.

The authors concluded that signature matching requires memories that support efficient regular accesses, some fast accesses to a smaller memory and capabilities to hide the latency of irregular accesses. These are all characteristics that are present on modern GPUs, so DFA processing (which has a somewhat predictable control flow) should be efficiently supported.

Measurements from their work proved the disadvantages of XFA: their measured speedup with XFA was of about 6.7, while a DFA implementation reached 8.6.

They proposed caching the XFA on the GPU in order to improve throughput, but this could also be done for DFA, so the performance issue should not be tackled from an implementation point-of-view, but rather from a design one.

Vasiliadis et. al proposed a solution in which the Aho-Corasick [2] algorithm was ported to the GPU, in order to perform high-speed intrusion detection [9].

In the authors' implementation, a separate detection engine instance is used to match the patterns belonging to a rule group. In order to improve the CPU to GPU memory transfer throughput, each rule group is copied using a buffer that was allocated using page-locked memory. The CPU to GPU transfer is also performed using direct memory access, thus freeing the CPU during this time. They have also used a double-buffering scheme, so that when the first buffer becomes full, it is sent to the texture memory so that the GPU can process it. While this processing is in place, the CPU is copying newly arrived packets in the second buffer.

The processing that the GPU performs is the execution of the Aho-Corasick algorithm. The automaton used by Aho-Corasick is stored in a tradition two-dimensional array, where each cell contains not only the information of the next state to move to, but also information regarding whether that state is final or not. As to improve memory access efficiency, the automaton is stored in the GPU's texture memory.

The authors found that overall, due to the cost of divergence in the execution, the best parallelization method was assigning chunks of packets to different threads.

They found that the system's performance was quite good, as it yielded a speedup of 2.4. The CPU was not completely offloaded though, so this opens up the system to targeted attacks. An attacker could craft packets that would need to be able to be verified by the CPU (after the GPU processing), thus slowing down the system.

Lin et al. [6] also proposed a port of the Aho-Corasick algorithm to the GPU, but with a somewhat different approach than the one by the work described above. A problem with running the Aho-Corasick algorithm on a GPU (while dividing the file(s) to be scanned amongst threads, which is a direct implementation of the algorithm) is boundary detection. This problem can be resolved by having threads process overlapped computation on the boundaries, but the overhead of such overlapped computation can degrade the performance.

Due to this issue, the authors opted for another approach: instead of assigning a packet's chunk to a thread, they assign each byte of the input stream in order to identify any pattern that may start at the thread's starting location.

This idea of allocating each byte of an input stream to a thread to identify any virus pattern starting at the thread starting location has an important implication on the efficiency. First, in the conventional Aho-Corasick automaton, the failure transitions are used to back-track the state machine when a mismatch occurs, so as to identify the patterns starting at any location of an input stream. Since in the proposed algorithm a GPU thread only needs to check for patterns starting at a particular location, back-tracking is not needed. Therefore, the failure transitions of the Aho-Corasick automaton can all be removed. And also, as each thread is allocated a different input stream byte to search for, there is no need for border detection.

The authors' algorithm allocates a large number of threads, and most of these have a high probability of terminating early. This means that threads will have to wait for others in order to proceed, which can lead to some overall performance loss.

Their results found that the system had an overall 2 to 2.5 speedup over the CPU version (taking into account memory transfers). Although a nice improvement in performance, the authors' proposed system has a simplistic work division, leading to possible uneven work between threads (depending on the input data).

Cascarano et al. [4] proposed an approach based on NFA. The adoption of a NFA instead of a DFA allows the system to efficiently store a very large regular expression set in a limited amount of memory. Instead of the regular representation of automata (a table which in every cell contains the number of the next state to transition to), the authors opted for a "symbol-first" representation: a list of

(source, destination) tuples representing transitions is kept, sorted by their triggering symbol. This list can grow to be very large, so it must be stored in the GPU's global memory, along with an auxiliary data structure that records the first transition for each symbol (in order to allow easy random look-ups).

To reduce the number of transitions to be stored (and also to speed up execution), the system adopts a special representation for self-looping states (those with an outgoing transition to themselves for each symbol of the alphabet). These states are marked as persistent in a dedicated bit-vector and, once reached during a traversal and marked as active, will never be reset. To improve memory access throughput, the bit-vectors containing current and future active state sets are stored in shared-memory.

As for the work division in the GPU, every thread in each block executes the main scanning algorithm. After the execution of some instruction, threads explicitly wait for the others to reach the same point before proceeding. Parallelism is exploited not only because at any given time multiple blocks are active to process multiple packets, but also because for each packet, each thread examines a different transition among those pending for the current symbol. In this way, a large number of transitions can be processed in parallel, and if there are enough threads, then the time spent in processing a single symbol will be effectively equal to what would be required for a DFA.

The overall speedup when compared to a CPU execution was not as good as a typical DFA approach, which can be explained by the fact that NFA still have a higher per-byte processing cost. The proposed system becomes ideal when signature sets become extremely large (at which point DFA solutions have their memory usage become too high).

3.2 Using ClamAV signatures

Vasiliadis et al. [10] proposed a solution in which ClamAV was modified to run on a GPU-based DFA matching engine. DFAs are the automata with the highest performance, being able to guarantee O(n) scanning time complexity. However, they also have a high memory cost, causing some researches to find alternatives with lower performance. The proposed system took into account the fact that most approaches rely on fast and accurate filtering of the "no-match" case, as usually the majority of network traffic and files are not malicious. The system contains two major phases: a high-speed filtering engine on the GPU, and a lower speed verification engine that is run on the CPU. The system starts by preprocessing the DFA from the signature set of ClamAV. Due to the large size of the virus signature set of ClamAV, constructing a single DFA is infeasible. In order to overcome this, they chose to only use a portion from each virus signature. This causes the GPU scanning to become a first-level filtering, effectively offloading a significant portion of work from the CPU. When it comes to assigning input data to each thread, the simplest approach would be to use multiple data streams, one for each thread, and in separate memory areas. However, this would result in asymmetrical processing for each shared multiprocessor. As such, they decided to have each thread scan a different portion of the input data stream. Using such a strategy means of course that boundary detection is necessary. The authors found that the best approach was to have each thread scan further from its chunk until it reaches a fail or final-state. This avoids communication between threads regarding boundaries in the input data buffer. The measurements done reflect the efficiency of a DFA-based solution: they found that the system had a maximum throughput of around 20 Gpbs, which constituted a speedup of around 100 over the CPU-only version.

4. Implementation

ClamAV's signature set can be very problematic for some automata-based systems (designed for sub-string matching): the existence of a high number of the “*" wildcard, means that if we are to build a full DFA, then at each point where we must represent the “*" wildcard there will be a high number of additional states to represent the possibility of all of the other patterns occurring at that point.

Due to the design choice of using a $\Theta(n)$ scanning algorithm (having transitions on the DFA in between prefixes of patterns), we found that the above problem was present on our signature set, and as such we could not include the whole signature set to be scanned, both in the case of regular expressions and static signatures. In our system we have two available GPUs, and our total DFA size was larger than the sum of both of the GPU's available space.

We have thus decided that for each signature, only part of it would be considered as to reduce the memory requirements. This will lead to occurrences of false-positives, but later in this section we will describe a mechanism to greatly reduce them.

Our process for extracting substrings from signatures was the following: For static signatures, we simply chose a string of a given size that is present in the middle of the signature. The position in the signature in which to choose the string is the middle: usually this is the part with the most variance, as both the beginning and the end of signatures most times contain characters that are present in regular files. For instance, a Portable Executable signature is very likely to contain the “DOS_HEADER" substring in its beginning. For regular expressions, we first check if the signature has a substring with no wildcards. If it does, then we use that string. If it doesn't, then we try to find the largest substring containing the least possible amount of wildcards.

This mechanism of extracting smaller parts from signatures still yielded some false positives. In order to reduce them, we opted to introduce another step in the matching algorithm: each signature has a given file type associated with it. Instead of reporting all of the occurring matches of signatures for all of the file types, we check if the current file being matched has the same type as the signature that was just matched (we have modified ClamAV to expose this file type detection functionality, and we perform this detection before scanning a file). If it does, then we report the match. If not, then we carry on with the algorithm.

Even after this optimization to reduce false positives, some were still present. This resulted in we having manually chosen the subsets of signatures with a low matching rate on a set of test files (corresponding to the */usr/bin* folder in a Linux distribution). A more automatic method would be to choose a set of test files, and for each signature, find all of its subsets. Then, scan the test files for each of the subsets, and whichever subset has the smallest matching rate, is the one chosen to be used. Such a process, although automatic, can be very costing in terms of execution time if the signature set is very large.

False-positives were still not fully eliminated though, due to some signatures being especially problematic: some of them are very small, such as 3 or 4 bytes in length. This means that they have a very high probability of occurring anywhere in a file. In fact, all of these signatures have a given file position associated with them, which indicates that their match is only to be reported if they occur at that position. Constantly checking positions in files is definitely not ideal for a GPU though, as it would either massively increase the complexity (for each position in a file, check for the signatures associated with it), or either bring divergence to the algorithm (some signatures can actually be present in intervals of positions, so checking them would not be a trivial process, causing other threads to be stalled waiting for such process to complete).

The choice of using only a signature's subset however is not enough to mitigate the memory space issues (it could be enough, but in order to do so, we would have to consider a tiny subset of each signature, which would massively increase the false-positive rate, at a point where manual interaction would not yield less false positives). Another design choice done to resolve this issue, was to divide the final signatures DFA into several smaller DFAs. As these different DFAs are composed of a lesser number of signatures, the subset construction algorithm doesn't have to keep track of as many emulated states, which not only allows it to run faster, but also causes the resulting automatons to be smaller.

In order to further reduce the number of states in each automaton corresponding to a signature subset, we have decided to group signatures (in order to form a subset) based on shared prefixes. Signatures that share the same prefix will also share the same DFA states.

Due to not being able to use complete signatures, some false-positive verification mechanism has to take place: it is possible that a given signature's part matches, but the whole of it does not. If a match occurs, then that signature needs to be verified, and in our system that is done by simply passing the corresponding file to ClamAV (this means that we only scan up until finding a match in a file). If however no match occurs, which should be most of time (considering that most of the files are not malicious), then no further verification is needed. In the case that the files are in fact malicious or are specifically constructed in order for a match to occur in the pre-filtering phase, then the execution time greatly increases.

It was straightforward to decide that files to be scanned should reside in a GPU for as long as the matching process takes place. This does limit the size of the automatons that are present on each GPU, but it is certainly a better option than constantly transferring the files from the CPU (or reading them from the CPU's DRAM, which has a large latency when accessed from the GPU).

Some space in the GPU's global memory is therefore allocated to files. But it is possible that the size of the files to be scanned is larger than this allocated space. In order to overcome this problem, we chose to divide files into chunks (or parts). Each file to be scanned is divided into several parts, which will separately be sent to the GPU to be scanned. Having files split into parts means that the GPUs need to keep track of the states (and results) of the automatons in between the parts. For example, if the matching process of a given file's first part ends at state 1, then the next part should start to be matched at state 1 instead of state 0 (the initial state), as to ensure that possible pattern matches between a file's parts are reported.

When it comes to the work division, taking into account our GPU's (Tesla C2050) characteristics, we have chosen to assign each of the smaller automata to a different multiprocessor: as they are already independent from each other, we can exploit that fact to have threads process files in different automata in parallel. Our GPU has 14 multiprocessors, so there are 14 smaller automata for each of the GPUs.

Assigning each automaton to a different multiprocessor, and taking into account the fact that most of the e-mail attachments present in e-mail gateways are small files, we have decided to assign each of the threads in a multiprocessor to a different file. A different work distribution could be had, such as having multiple threads process different parts of the same file, but this would not be ideal as only a small number of files could be scanned at a time; this would also mean that memory transfers from the CPU to the GPU would be an overhead to consider, as they would be done very often. Having multiple files being scanned at a time

leads us to performing less overall memory transfers (performing a single transfer of the files to memory instead of performing multiple smaller transfers of single files).

Overall, the process of verifying a set of files is:

- 14 Automatons are built for both the static strings signature subset and the regular expressions signature subset. The process takes around 2 hours for each, so it is expensive. Signatures aren't updated that often though, so this is not a problem. The automatons reside in each of their GPU's texture memory, until they are eventually updated with new ones.
- At some point in time, the CPU is given a set of files. If the set's total size is less than the available remaining space in the GPU's memory, then the files are simply sent to the GPU (in groups of 1024 - our GPU supports a maximum of 1024 threads per block, and remember that each block is processing files for a different automaton). If the set's size is bigger than the available size, then files are divided into parts and groups of parts are sent to the GPU.
- Once the GPUs receive the data to be processed, they run the matching algorithm: states are fetched from the state table resident in texture memory, and checked if they are final. If they are, their corresponding signature file type is verified against the file type of the file present in the current thread (these file types are saved in shared memory). If these match, then the current file is marked as being suspicious and the thread responsible for it stops processing.
- Once all of the threads in both GPUs stop processing, then their results (files marked as suspicious) are aggregated and sent to ClamAV for a full verification.

After implementing our system according to the decisions described in the previous section, we found that its performance was not satisfactory. Namely, little or no improvement was had over an equivalent CPU version of the algorithm.

In order to improve the system's performance, we implemented some optimizations, which are now described.

Texture memory has a separate cache, and it is designed to be used in algorithms with 2D locality. Automata-based algorithms can exhibit such locality (depending on the input data, the algorithm can be constantly fetching the same states), so the texture memory was a natural choice to include our automaton in, instead of global memory. Global memory is in fact a bad choice for the automatons to be present in. Automatons have random access patterns, as the states to be fetched depend on the input data, which cannot be predicted. Global memory does not work very well with random access patterns: if two threads request values from two memory addresses that are distant from each other, then the GPU performs two serial memory accesses. However, if the two memory addresses are sequenced, then a single memory access is performed, and the two values are broadcast to each requesting thread. Global memory is therefore a better suit for linear access patterns. We have one data structure in our algorithm that has such a pattern: the input data.

As described above, we need to keep in memory the types corresponding to each signature in order to reduce the rate of false-positives. But keeping these values in global memory would not be ideal: they are only accessed in the case that we have found a final state, and they are not accessed in an orderly fashion (so no coalescing of accesses could be done).

In order to save these memory accesses, the same idea is applied: due to the choice of using texture memory, we can only have 130,000 states in each automaton. This means that in each cell of the state-transition matrix, there won't be a number larger than 130,000. However, we still need 4 bytes to keep this value in, and lots of bits will be unused (as the value 130,000 does not use them all to be represented).

Taking this into account, we decided to represent final states as negative ones. Checking if a state is final is now a trivial operation, compared to accessing a value in the GPU's memory.

The type corresponding to a given signature is also embedded in that signature's final state number: as ClamAV only has 10 types, from 0 to 9, every time we reach a final state in the NFA to DFA conversion process, we simply multiply that state's number by 10, and add the corresponding signature's type.

Conversely, checking a type for a given final state is now reduced to getting the remainder of the division of that state's number by 10.

GPUs benefit greatly from aligned memory accesses. If accesses to global memory in a warp are not to sequential memory accesses, then the GPU's hardware will have to perform one separate memory transaction (occupying the bus) for each access. If they are aligned then a single transaction can be performed, and the fetched values broadcast to the warp threads.

In our system, as mentioned earlier, each thread in a multiprocessor is assigned to a different file, and naturally they all start processing the file at its beginning. Having each of the threads accessing contiguous memory locations means that the files can not be stored contiguously: they need to be interleaved, so that all of threads access the same position at the same time, and in contiguous addresses. The process of interleaving files can be seen as performing a transpose operation on them.

Transposing a non-square matrix takes some time in the CPU. If we were to perform this transposing every time we

want to send files to the GPU, then some inefficiency would be present: the GPU would be waiting for the CPU to finish the transposing before being able to process the data.

Likewise, sending the data from the CPU to the GPU also takes some time, although far less than the one needed for the transposing (in our card, sending around 600MB takes around 40 milliseconds).

In order to cut these times, we have chosen to partition the data to be sent: the GPU will effectively process all of the data as before, but not all at once.

We divide each file into parts, and we only send parts of files at once. The smaller the parts (and thus the bigger the number of parts of a file), the less time we spend performing the transpose, but the more launches we perform on the GPU.

Separating the files into parts allows us to perform background execution on the CPU: while the GPU is processing a given batch of parts of files, the CPU is already transposing them; the CPU even sends them to the GPU while it is processing the previous batch (this is possible due to the use of page-locked memory on the CPU).

Once a batch is finished, the GPU then has to wait a very small amount of time (compared to waiting for a transpose) before it can start processing the next one. This waiting is due to the fact that splitting files into parts means that we have to keep track of states in between file parts, as not to loose any possible match information. So each time a batch is finished, the GPU copies the last processed states and results onto the next batch.

5. Evaluation

For our experiments, we use a computer equipped with two Tesla C2050 cards, an Intel i7 950 processor and 12GB of RAM. All of the times for ClamAV were obtained by calling the *clamd* daemon and then scanning files using *clamdscan*. This way, the time used to initialize ClamAV's data structures is not counted in the total scan time. Additionally, we could not reliably profile the pattern matching portion of ClamAV. Thus, considering that, by Table 2, the pattern matching portion of ClamAV uses approximately 80% of its time, we have taken the times given by *clamdscan* and multiplied them by 0.8 in order to obtain an approximation of the time spent in pattern matching. The execution times for the GPU are given by the largest execution time of both of the GPUs: if a GPU finishes processing before the other, then it waits before the verification module is executed. The times for the CPU are measured with an equivalent version of the algorithms on the GPUs: two threads run in parallel, and each of them respectively processes files on the automatons for the regular expression signatures and the static signatures. When they are both done, the verification module is called for any file which matched a signature.

Number of files	Total size (bytes)	Largest file (bytes)	GPU (ms)	Clam AV (ms)	Speed up
10	114,870	44,552	40	34	0.9
100	17,815,178	3,974,440	2,744	830	0.3
1000	57,702,247	663,936	956	1,829	2.7
2000	85,604,681	340,376	674	10,218	15.1
3431	97,291,831	170,176	527	14,691	27.8

Table 3. Execution times for files with different sizes.

We tested our algorithm on a somewhat realistic set of files: files from the */usr/bin* folder of a Linux distribution, which contains files with different sizes. We changed the number of files that were scanned, along with the total amount of data and the size of the largest file, to see what each of these characters had an impact on our system's performance.

Table 3 shows the results.

We can see that as expected, the biggest speed-up was had when scanning a large amount of files with a relatively small size: it appears that ClamAV does not handle a large amount of files very well, but our system does. Scanning a small amount of files however led to our system having a worse performance than ClamAV. As such, we can conclude that our system should only be present on situations where many files have to be scanned, and ideally, these files should have limited sizes. This is the case in e-mail gateways: they contain many e-mail attachments, and usually e-mail servers limit attachments to a given size.

We mentioned the possibility of false-positives being present in the system, and we described a mechanism to verify them: we simply use ClamAV itself to verify files that had a given signature match in them.

This characteristic of our system is in fact one of its weak points, as it can be attacked by a malicious user constructing specially crafted files. In order to see the impact of such an attack on the system, we measured the execution time for the above 1024 (real) files, but while continuously adding a signature to them, so that a match would be reported by the GPU. With an increasing number of files (out of 1024) having a match, then the most work the verification module (ClamAV) must perform.

Chart 1 shows the results.

In the case where only one false-positive out of 1024 is present, our system performs well as expected (ClamAV only has to scan a single file, which is a fast process). How-

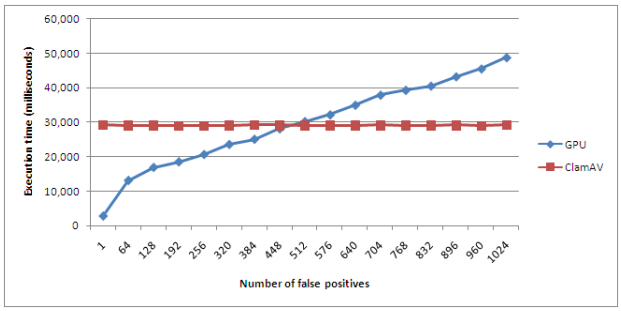

Figure 1. Execution times for different numbers of false positives.

ever, as the number of files needing to be verified increases, so does our system's total execution time. At the point where more than 512 files out of 1024 need to be verified, our system's performance is worse than ClamAV's, thus eliminating the purpose of having implemented our system.

Despite being an extreme case (as the vast majority of files is not malicious), this test helps understand what would happen in the case of a targeted attack.

6. Conclusion

We started by showing the problem that we proposed ourselves to solve, along with background on it. We gave an introduction to general purpose computing on GPUs, and showed how they can be utilized to achieve high performance. We then presented related work on the area, which served as a basis for our own.

Details were shown for our system, an $\Theta(n)$ algorithm running on two GPUs which pre-processes files in a high-speed manner in order to detect if possible virus signatures are present in them. It differs from other related work using ClamAV as it is capable of scanning a high quantity of files in parallel.

The system was found to have a good speed-up on the case that it was designed for: an e-mail gateway with a large amount of files with limited sizes. A number of optimizations were implemented in order to achieve this performance, which was not possible with a first implementation.

Our solution did not perform so well on some cases, and it is open to targeted attacks. More work can be done to mitigate these issues, such as having a different work distribution for large files.

References

[1] Nvidia cuda programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, June 1975.

[3] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 1:1–1:12, New York, NY, USA, 2007. ACM.

[4] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA pattern matching on GPGPU devices. *SIGCOMM Computer Communication Review*, 40(5), oct 2010.

[5] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29:752–788, December 2004.

[6] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu. Accelerating String Matching Using Multi-threaded Algorithm on GPU. In *GLOBECOM 2010 - 2010 IEEE Global Communications Conference*, pages 1–5. IEEE.

[7] 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.

[8] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[9] G. Vasiliadis, S. Antonatos, and M. Polychronakis. Gnort: High performance network intrusion detection using graphics processors. ... *in Intrusion Detection*, 2008.

[10] G. Vasiliadis and S. Ioannidis. GrAVity: a massively parallel antivirus engine. In *RAID'10: Proceedings of the 13th international conference on Recent advances in intrusion detection*. Springer-Verlag, sep 2010.