A Highly-Efficient Memory-Compression Approach for GPU-Accelerated Virus Signature Matching

Ciprian Pungila and Viorel Negru

West University of Timisoara,
Blvd. V. Parvan 4, Timisoara 300223, Timis, Romania
{cpungila,vnegru}@info.uvt.ro
http://info.uvt.ro/

Abstract. We are proposing an approach for implementing highly compressed Aho-Corasick and Commentz-Walter automatons for performing GPU-accelerated virus scanning, suitable for implementation in real-world software and hardware systems. We are performing experiments using the set of virus signatures from ClamAV and a CUDA-based graphics card, showing how memory consumption can be improved dramatically (along with run-time performance), both in the pre-processing stage and at run-time. Our approach also ensures maximum bandwidth for the data transfer required in the pre-processing stage, between the host and the device memory, making it ideal for implementation in real-time virus scanners. Finally, we show how using this model and an efficient combination of the two automata can result in much lower memory requirements in real-world implementations.

Keywords: gpu, gpu-accelerated, cuda, commentz-walter, aho-corasick, wu-manber, memory efficient, virus scan, malicious code detection.

1 Introduction

The problem of efficient virus detection has had a lot of attention over the past few years and, as the numbers of viruses grew exponentially, is now one of the most computationally-intensive tasks found in intrusion detection systems. With the performance boost that technology and, in particular, graphics cards have shown lately, the solution has been improved dramatically, by a good factor of times compared to normal CPU operational speed.

In essence, the problem of virus detection is related to the challenge of efficient pattern matching (given that there are multiple patterns in the database that need to be matched), both in terms of memory usage and processing time. Aho et al[1] were the first to propose an algorithm for exact multiple pattern matching (Aho-Corasick, or A-C) that is fast and runs in linear time. A-C is optimal in the worst case, however, as Boyer-Moore have shown in [2], in practice there are situations when the running time can be improved by jumping over redundant characters, an idea that was later on studied by Commentz-Walter in [3].

D. Gollmann and F.C. Freiling (Eds.): ISC 2012, LNCS 7483, pp. 354–369, 2012.

The result was an algorithm (Commentz-Walter, or C-W) that is not optimal in the worst case, but that can produce the same results in sublinear time in the average case. Better performance had been obtained by Wu et al in [4] (the Wu-Manber algorithm, or W-M) after implementing a hashing approach that can significantly reduce the number of overall operations performed. We relate in this paper to ClamAV[5], one of the most commonly used open-source antivirus programs today.

This paper focuses on performing memory and time-efficient virus signature matching in real-world systems by taking advantage of the massively parallel processing capabilities of GPUs, and proposes a highly-efficient memory-organization and virus signature scanning model that helps achieve these goals without compromising performance. Furthermore, we show how using our approach, an efficient combination of the A-C and C-W algorithms can result in much lower memory requirements. We discuss related work in section II, presenting the CUDA architecture, the A-C, C-W and W-M algorithms along with the ClamAV virus signature format and its implementation, while in section III we present our own approach and the proposed model and discuss it in depth. Section IV shows the experimental results we have obtained with a variety of implementation scenarios performed, both on the GPU and on the CPU.

2 Background

2.1 The CUDA Architecture and Programming Model

For this paper we have focused on the feasibility of adding GPU-accelerated virus scanning support to antivirus software in generic consumer hardware. We have used a Core i7 CPU and a CUDA-based (PCI-E 2.0) GTX 560Ti 1GB DDR5 graphics card, with the engine clock running at 900MHz, shader clock at 1800MHz, memory clock at 4200MHz and having 384 CUDA cores. The Compute Unified Device Architecture (CUDA) is described in [7] and is the basis for programming all CUDA-based GPU cards, offering an abstraction of the underlying hardware through additional libraries that can be easily called from C code. For our implementation, we have used Visual Studio 2010 and CUDA version 4.1.

The GTX 560Ti card is based on the GF114 chip and has two processing clusters with 4 multiprocessors each. Each multiprocessor consists of 48 stream cores and is accompanied by 8 texture-processing units (see Figure 1). Stream processors are operating on SIMD (Single Instruction Multiple Data) programs. CUDA programming allows the GPU to be used as a massively parallel execution machine, supporting a very high number of threads. Programs issued by the host computer to the GPU are called kernels and are executed on the device as one or more threads, organized in one or more blocks. Each multiprocessor executes one or more thread blocks, those active containing the same number of threads and being scheduled internally. It is possible to manage shared memory and the host can access global, constant and texture memory (constant and texture memories are cached, so therefore faster).

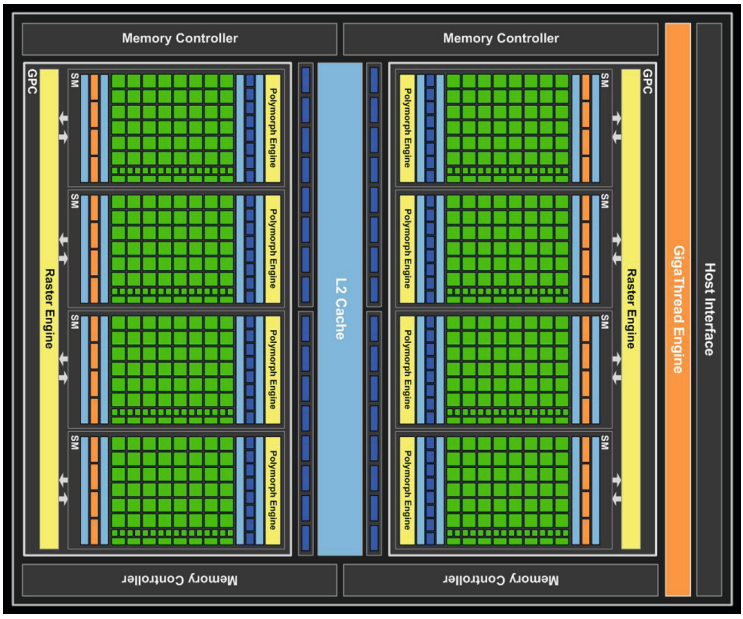

Fig. 1. The GF114 chip layout

2.2 Multiple Pattern Matching

The Aho-Corasick algorithm is based on the trie tree concept and is in fact a deterministic finite automaton (DFA) that has, at each level, a failure pointer associated to the node, which gets called in case of a mismatch in the DFA at the current state. The failure, for the root of the trie, as well as for all the nodes at the first level in it, is always pointing back to the root. For the other levels of the trie, the failure is being computed as being the longest possible suffix of the word (at the current node in the trie) that is also a word in the tree (see Figure 2) - if no such suffix exists, the failure points back to the root.

The Commentz-Walter algorithm however uses a different approach, based on the Boyer-Moore shift approach from simple pattern matching. It creates a trie tree from the set of reversed patterns, and assigns to each node two shift values, *shift1* and *shift2* and two additional functions, *out* and *char*. The C-W performance heavily relies on the length of the smallest keyword in the tree (w_{min}). *shift1* is being computed as the smallest between w_{min} and (if it exists) the distance between the current level of the current node (v) and all superiors levels of the other nodes, that have as suffix the word formed at the initial node *v*. *shift2* is computed similarly, except that in this case we are computing the smallest between w_{min} and (if it exists) the distance between the current level of the current node (v) and all superiors levels of the leaf nodes that have as proper suffix the word formed at the initial node *v*. *out* returns the pattern for the current node, if its word is a keyword in the tree, while *char(a)* is computed

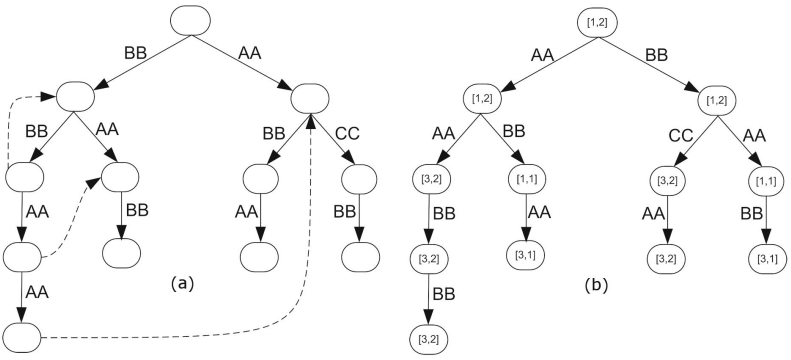

Fig. 2. a) The Aho-Corasick tree for the input set {BBBBAAAA, BBAABB, AABBAA, AACCBB}. Sample failures are marked dashes. b) The Commentz-Walter automaton for the same input set (includes distances as [shift1, shift2]).

as being the minimum of $w_{min}+1$ and the level of all nodes having a as label. The processing begins with a right-side comparison starting at position w_{min}; in case of a mismatch, the current position at node v is increased by $min\{shift2(v), max\{shift1(v), char(text[pos-j])-j-1\}\}$, where *pos* is the current position and j indicates the depth in the tree. Our experiments have shown that the *char* function is always 1 for the set of signatures used in ClamAV, which is why the shifting distance for C-W becomes $min\{shift1(v), shift2(v)\}$.

The Wu-Manber algorithm uses three tables built in the pre-computation stage, called *shift*, *hash* and *prefix*. The *shift* table is similar to the one precomputed for the Boyer-Moore bad-character skip table, while the other two are only used when *shift* table indicates not to shift (since there may be a potential match at the current position). Just like Commentz-Walter and Boyer-Moore, the performance highly depends on the length of the shortest pattern. The algorithm looks at blocks of text, and the authors suggest a size of 2 for short patterns and 3 for longer ones, leading to significant memory-requirements.

Numerous applications of multiple pattern matching have been proposed for intrusion detection systems: an A-C implementation based on the ClamAV signature format is proposed in [8], another A-C-based approach for detecting malicious code through system-call heuristics is proposed in [9], while in [10] bloom filters are used to speed-up the scanning process. Lin et al[11] have observed that the W-M approach can be used successfully on longer patterns from the ClamAV database, while A-C can still be used for regular-expression matching with shorter patterns, however their approach is lacking parallelism and is still relying on the performance of the A-C machine, and although W-M performs much better for longer patterns, A-C remains the primary bottleneck.

2.3 ClamAV and GPU-Accelerated Virus Scanning

Cha et al in [12] have observed that 95% of the scanning time of ClamAV is focused on matching the regular expressions, although their number is very small

(only about 16% of the number of signatures). ClamAV's A-C tree is limited in depth, as shown in Figure 6a [13]. The majority of the ClamAV signatures (specifically, about 96.6% in the version of the *main.cvd* ClamAV database we used) uses three types of constraints: {*n*-}, for specifying at least *n* characters, {*n*-*m*} (for specifying between *n* and *m* characters) and {-*m*} (for specifying at most *m* characters). The remaining signatures could be easily transformed into this type of constraints, since for instance *?* can be replaced with {*1-1*}, * with {*0,*-} and subexpressions of type {ABCD(EE|FF)00} can be distributed into separate patterns as follows: {ABCDEE00} and {ABCDFF00}. In our our tests, from the 62,302 signatures we had used for testing from the ClamAV database, only 6,831 contained regular expressions (about 11%).

A GPU-based pattern matching approach for virus scanning has been presented by Vasiliadis et al in [6] as GrAVity, with significant improvements over the performance of a single-core CPU, by limiting the depth of the Aho-Corasick tree used for scanning. Their approach however uses significant amount of memory (each node in their implementation uses 1KB, which leads to 400 MB of memory for 400,000 nodes). Tuck et al have presented a compression-approach for the A-C automaton through path-compression in [14] for the Snort network intrusion detection system[15], improved even further through bitmapped nodes by Zha et al in [16].

2.4 Motivation of Our Work

The motivation of our work is based on two important aspects for malware detection: the first is that static signatures are still the only completely accurate way of detecting malicious code (and according to antivirus manufacturers they will not disappear anytime soon, with so many thousands of viruses in the wild), and the second is that the process of exact matching signatures is usually the slowest, compared to dynamic heuristics. The recent improvements in hybrid CPU/GPU solutions have shown that GPU-based solutions outperform greatly CPU-based ones, however the main problem remains the data transfer between the host and the device, a problem we aim to solve in this paper. For real-world implementations, an antivirus based on GPU acceleration should offer a few important advantages: low-memory footprint, good performance and WDDM TDR-compliance.

The ClamAV implementation was modified to allow GPU-processing in GrAVity[6], however the approach used a very inefficient implementation of the DFA since every node was using 1 KB of data (given that at each node, there are a maximum of 256 possible transitions, and that each transition is a pointer, that will lead to at least 4*256 =1,024 bytes per node). That leads to 400 MB of usage for the 400,000 nodes that the authors had obtained when using a tree limited to depth 14. While Tuck et al[14] have shown how memory requirements can be improved through path-compression using Snort signatures[15] (they had replaced consecutive nodes having a single child with one single compressed node), an approach which was also used by Zha et al in [16] for reducing memory usage even further through bitmapped nodes, a real performance bottleneck in GPU-based implementations is also the way the memory is allocated (see Figure 3).

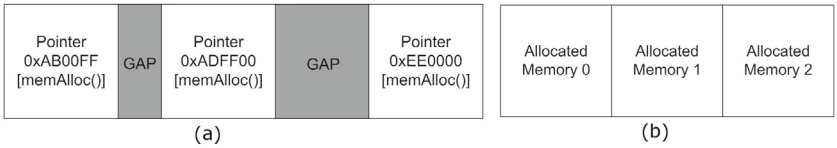

Fig. 3. a) Sparse, consecutive memory allocations and gaps that appear in GPU memory; b) A single, sequentially allocated memory block in the GPU device

Vasiliadis et al with MIDeA[17] have presented an architecture for intrusion detection which uses a compacted state table representation, that only stores for each state the elements between the lower and the upper limits. While this approach does offer improvements over the one presented in [6], it does still store additional redundant information if the states are sparse enough and a low number of children exist for each state. In Gnort[18], the authors present a solution based on Snort for implementing network IDS using GPUs, however the actual implementations use the default Snort approaches (full, sparse, banded, trie, etc.), still for all of them there is room for improvement in terms of space efficiency. PFAC[19] is an open-source library that implements string matching on the GPU that applies perfect hashing to the parallel failureless A-C, building one or two-level perfect hashing for efficient storage, at the expense of processing time. The final aim of PFAC is to build a perfect hash that stores only valid transitions in the automata, however the process required to compute and build the hash is laborious and quite expensive (as the authors observe, the *mod* operation is expensive to compute on the GPU), plus the actual memory requirements may be higher (authors propose an inequality for determining the upper bound for space usage, while in our approach the space usage is always constant).

ClamAV uses a depth-limited A-C tree (with a depth of 2, since the smallest virus signature contains a subexpression of just 2 bytes) along with an implementation of W-M. The W-M algorithm however uses large amounts of memory because of the shift and hash tables. In our implementation, for a block size of 3, we obtained $2^{24} = 16.7$ million elements in the hash table, or, considering that two 32/64-bit integer (we used 64-bit in our implementation) values are stored into each element of the table, a total amount of 64 MB or 128 MB memory, not counting additional structures needed.

In order for code that runs in the GPU through the CUDA architecture to access the data it needs, this needs to be transferred from the host (the RAM memory) to the device (in the GPU memory). The device to host bandwidth can reach high peaks given the PCI-E 2.0 x16 architecture, however the bandwidth is higher as the memory block being transferred is larger. That means that for small transfers, such as repeated 1 KB transfers, the transfer is highly inefficient. There is also a problem of the extra space being left-over as a gap by the library-based memory allocation routines, for small-footprint allocations. In order for the GPU to be able to take on the matching tasks from the CPU, it must have access to the memory area that needs to be scanned or passed through the A-C DFA, along with the data structures required to perform the

processing. Sparse, repeated small-size memory allocations will create unwanted gaps between memory locations, which may significantly increase the memory usage, even though that may not be desired or expected. Although it may be possible that future allocations will fit the gaps left-over, there is no guarantee that it will happen eventually.

In the case of DFAs, where the number of nodes can reach several millions, this is becoming a real problem as it can create huge memory losses. The second problem with repeated allocations is that they consume a large amount of time - in our tests, copying all DFA nodes from the host device to the GPU device took over 2 hours for about 350,000 nodes. Of course, the ideal approach is to copy large areas of memory at once and benefit from the full PCI-Express bandwidth. This poses additional challenges since tree-based structures are generally comprised of unevenly distributed pointers throughout memory, simply creating a linear list from these pointers will not suffice, as all data structures must also update their internal pointers to accommodate the new locations. Our approach using multiple queues will achieve a perfectly linear, gap-free distribution of nodes, as we will show in Section III. Using our model, the performance of parsing the automata is not compromised and the space storage required is always linear in S, the total number of states.

3 Implementation

We used 62,302 signatures from the ClamAV database for our tests. We have implemented the A-C, a constraint-based A-C, C-W and W-M algorithms and we began testing on the CPU, then moved on to the GPU. The constraint-based A-C is an adaptation of the A-C algorithm to support the three types of constraints that ClamAV signatures contain. For the CPU implementations, we used path-compression and bitmapped nodes for the A-C and C-W automatons (Figure 4 shows the data structure formats used during implementation), to minimize memory usage, while for the GPU part we have built our own memory model for aligning the automaton so that we can maximize the host-to-device bandwidth when copying it into the GPU. We used a data-parallel approach for performing scanning on the GPU.

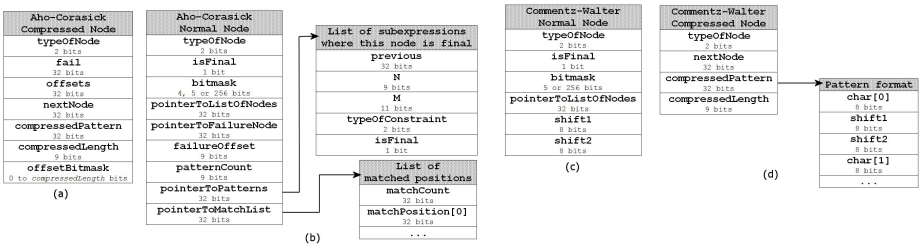

Fig. 4. a) A-C compressed nodes; b) A-C normal nodes in the constraint-based automata supporting RegEx signatures; c) C-W normal nodes; d) C-W compressed nodes

Figure 5 *b, c, d* shows results for single-core CPU benchmarks of individual algorithms, while Figure 5*f* shows the results obtained when attempting to implement a dual-scanning system (first, AC-AC used A-C in one thread for scanning RegEx signatures and again A-C in the other for scanning regular signatures, and the second, AC-CW used A-C in one thread for scanning RegEx signatures and C-W in the other for scanning regular signatures). The results obtained showed that AC-CW is performing faster than AC-AC, however given that C-W performs best when the shifting distance is higher, we have split the set of signatures into two parts: the first, containing regular expressions and patterns shorter in length than *d*, and the second containing patterns longer than *d* (where *d* was chosen as 16, 32, 64, 96 and 128) (Figure 5a shows the signature distribution after the split, showing how many signatures were scanned by the RegEx signature scanner, and how many by the regular signature scanner in the different scenarios). We also tested a two-thread scanner comprised of A-C for RegEx scanning and W-M for regular scanning (referred to as AC-WM), and results in Figure 5e show that memory usage is much lower in the AC-CW approach compared to AC-WM.

In GrAVity, the authors limit the depth of the tree in order to reduce memory usage[6]. This may produce false positives, but once a potential match is detected an extra step is employed at the end to fully verify that a full match is present. It is also stated that for a depth-limited tree of 8, the number of false positives when scanning is less than 0.0001%, which is why in our approach we have also used the same depth for our tree (however we also performed memory and bandwidth benchmarks for depths 12 and 16). Our tests were performed on 50 MB of randomly generated binary data and we have used 128 threads per block (a 66% occupancy rate offered the best results after performing initial testing with 96, 128, 160 and 192 threads) and 8,192/4,096/1,024 blocks (leading to a total of 1,048,576/524,288/131,072 threads) for each kernel running.

3.1 The Constraint-Based Aho-Corasick Automata

We have implemented the default A-C approach and also a constraint-based version, where we added basic support for the three types of constraints found in the ClamAV signatures. The process is performed as follows: a regular expression signature is being parsed and trimmed into its subexpression parts (e.g. *11AA22*{*8-*}*33FF*{*-12*}*DDEE* will insert in the tree, separately, the patterns *11AA22*, *33FF* and *DDEE*), with pointers existing from the leaf of a subexpression to the leaf of the subexpression preceding it. Whenever a leaf of any subexpression is reached, the leaf of the previous subexpression is checked - if all leaves were visited, this is considered a partial match and is checked at the end for a full match. Our constraint-based version of A-C ensures accurate matching by adding match position arrays at every leaf of every subexpression, and verifying constraints on the spot whenever a partial match is found (Figure 6b).

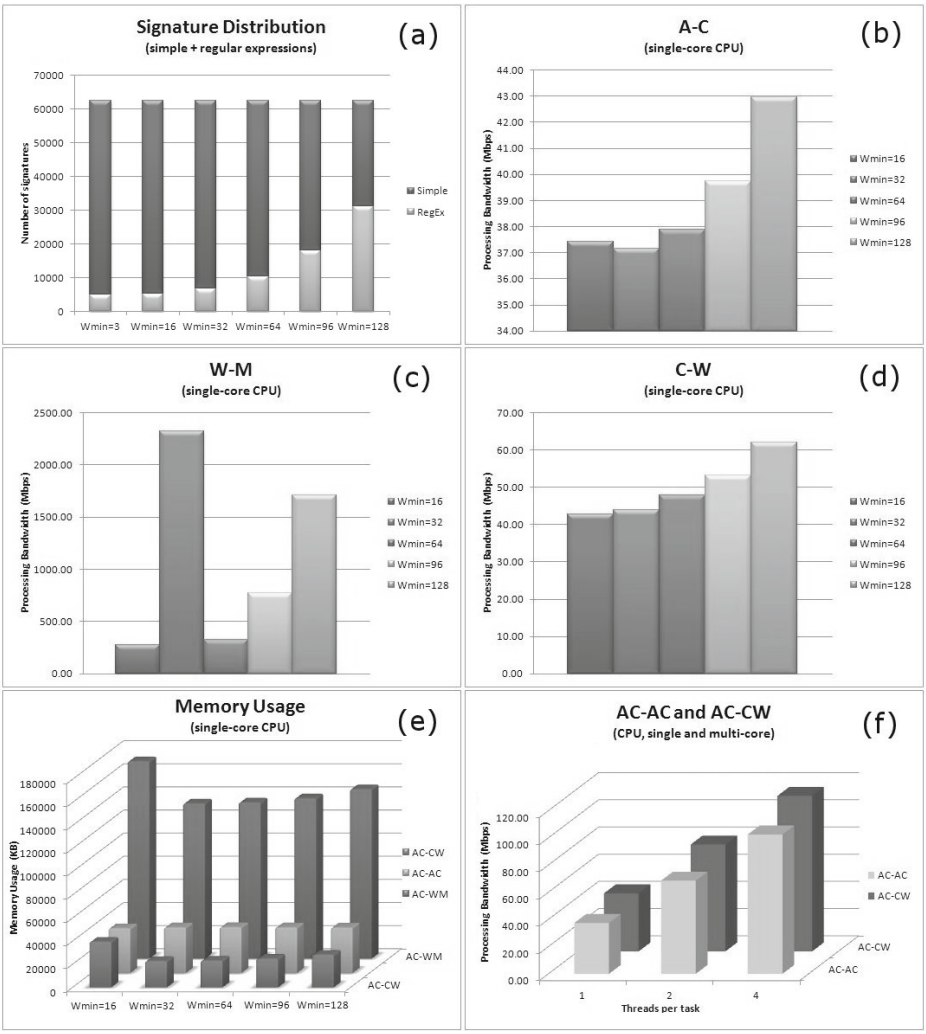

Fig. 5. Signature distribution when performing dual-scanning and single-core CPU performance results for tested algorithms

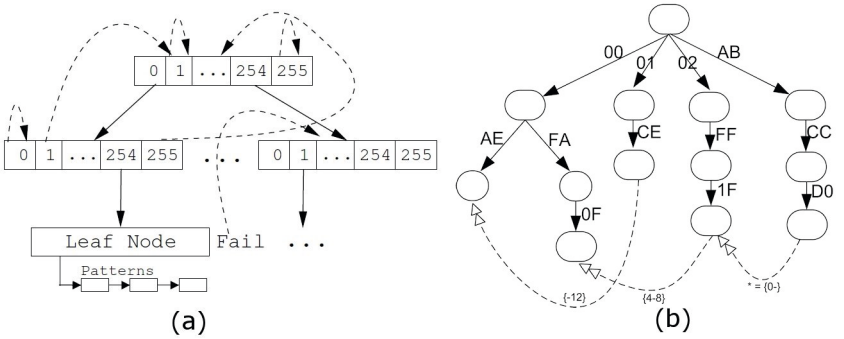

Fig. 6. a) ClamAV implementation of the A-C algorithm [13]; b) The constraint-based A-C automata for the input set {00FA0F{4-8}*02FF1F*ABCCD0, 00AE{-12}01CE}. Failure transitions are not shown.

3.2 Our Proposed Memory Compression Model

We have used stacks for rebuilding the automaton, replacing the pointers that exist in the usual data structures with offsets in the stacks pointing to the elements of interest. Basically, we are serializing the tree in a continuous, gap-free memory block, but using offsets in the stacks for locating elements, instead of pointers. For the standard A-C and C-W versions we use two stacks, *nodes*, *offsets*: the first points to the list of serialized nodes, stored as structures, while the second points to a list of offsets (Figure 7). The *offset* member of the node structure indicates to the position where all the children of that node are stored - the number of children is given by the population count of the bitmap in the structure. The model proposed always stores all children of a node in a sequential manner in the stack, while the offset in the stack corresponding to the first child is stored in the parent node. Therefore, the child at index i in a node n is accessible through a single reference: *nodes[offsets[n->offset+i]]*.

The constraint-based automata uses a similar approach and a third stack, *links*, which holds structures that contain an offset to a node in *nodes* (this offset indicates the leaf of a previous subexpression in a regular expression pattern).

Building the A-C/C-W automata and preprocessing it is not part of the virus scanner itself, since database signatures are usually delivered (through the update mechanism) in a ready-to-be-used form. Given that our automata is one continuous block of memory, the transfer between the host and the device is the fastest possible. The algorithm we used to build the node stack for our model is presented as follows:

– *Initialization*
  1: $topOfNodeStack \leftarrow 1$ (top of node stack)
  2: $currentPosition \leftarrow 0$ (position in the stack)
  3: $topOfOffsetStack \leftarrow 0$ (top of the stack of offsets)
  4: *node* (the currently processed node)

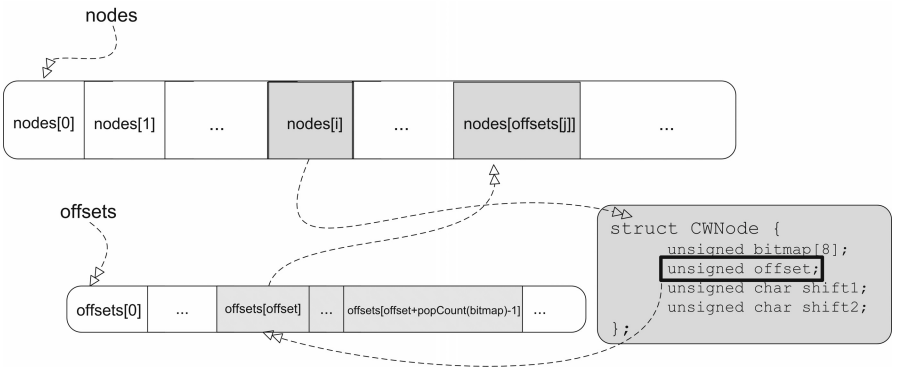

Fig. 7. The stack-based memory model used for efficient storage of the C-W automata in the GPU. A similar model is used for the basic A-C implementation.

– *function addNode(node, currentPosition)*
  1: $nodeStack[currentPosition] \leftarrow node$
  2: $nodeStack[currentPosition].offset \leftarrow topOfOffsetStack$
  3: add to hash ($key \leftarrow node$, $value \leftarrow currentPosition$)
  4: $pc \leftarrow popCount(node.bitmap)$
  5: **for** $i \leftarrow 0$ to $pc - 1$ **do**
  6: $\quad offsetsStack[topOfOffsetStack] \leftarrow topOfNodeStack + i$
  7: $\quad topOfOffsetStack \leftarrow topOfOffsetStack + 1$
  8: **end for**
  9: $old \leftarrow topOfNodeStack$
  10: $topOfNodeStack \leftarrow topOfNodeStack + pc$
  11: **for** $i \leftarrow 0$ to $pc - 1$ **do**
  12: $\quad$ addNode($node.child[i]$, $old + i$)
  13: **end for**

To further reduce memory usage, the *offsets* stack can be completely discarded, assuming that offsets are stored in the node itself. The code for serializing the automata while using a single stack of nodes is the following:

– *Initialization*
  1: $topOfNodeStack \leftarrow 1$ (top of node stack)
  2: $currentPosition \leftarrow 0$ (position in the stack)
  3: *node* (the currently processed node)
– *function addNode(node, currentPosition)*
  1: $nodeStack[currentPosition] \leftarrow node$
  2: $nodeStack[currentPosition].offset \leftarrow topOfNodeStack$
  3: add in hash ($key \leftarrow node$, $value \leftarrow currentPosition$)
  4: $pc \leftarrow popCount(node.bitmap)$
  5: $old \leftarrow currentPosition$

6: $topOfNodeStack \leftarrow topOfNodeStack + pc$
7: **for** $i \leftarrow 0$ to $pc - 1$ **do**
8: addNode($node.child[i]$, $old + i$)
9: **end for**

In order to efficiently restore the pointers in the new structures, a hash is created in the first step that holds a list of the nodes in the automaton, together with their new corresponding offset in the GPU memory-representation. A second processing step is then employed, which parses the GPU tree in pre-order and restores pointers by locating them in the hash and replacing them with the offset. While path-compression and bitmapped nodes may be applied to the automata at this stage to further reduce the memory usage on the GPU, there are a few elements that suggest this option should be avoided in general: first, memory rellocation in the GPU is a time-consuming process (memory manipulation routines, such as *memmove()* and *realloc()* in C, have significant impact over the overall performance of the running thread) and second, coding complexity greatly reduces, therefore reducing performance penalties in the GPU.

A simple theoretical analysis shows that in order to implement, using our model and bitmapped nodes, a basic Aho-Corasick pattern matching automata with an alphabet of size Σ and a total of N nodes/states, the total space required for storage is $N \times (2log_2N + \Sigma)$ bits (for each node, the following was considered: one failure offset to indicate the new state in case of a mismatch, using log_2N bits, the offset pointer to indicate to the list of children for the current node which is also occupying log_2N bits, and a bitmapped representation of the alphabet occupying Σ bits). Comparing this result to PFAC[19], where the AC-Compact approach used by the authors for 1,703,023 nodes occupied 24.18 MB (about 15 bytes per node, assuming an alphabet size of $\Sigma = 32$), our approach (without any performance penalty, unlike AC-Compact) would require a storage space of only $2 \times 21 + 32 = 47$ bits per node, or 10 bytes per node and a total of 15.02 MB of memory (1.6 times less memory) to store the complete automata in memory.

4 Experimental Results

We have conducted a series of tests for evaluating the performance of our approach on the GPU using two types of tests: single-scanning, where the whole input data set was scanned using the A-C and constraint-based A-C machine (Figure 8), and dual-scanning, where we have used the A-C and AC-CW approaches for performing scanning in parallel with regular expressions (in the A-C machine) and simple patterns (through C-W, see Figure 9).

In Figure 8a, the total number of nodes obtained after applying the depth-limited implementation of our model (we used three different depths of 8, 12 and 16 for testing various scenarios) is directly proportional to the amount of memory used by the A-C automata in Figure 8b. The GPU throughput of both the standard A-C and the constraint-based A-C used in RegEx scanning (in Figure

8 c, d), for the three different depths shows that the default A-C works about twice as fast as the constraint-based A-C. RegEx scanning is mainly limited by the large number of small signatures that get matched partially, since ClamAV's smallest signature is comprised of only 2 bytes. In particular, for a depth of 8, when using the basic A-C machine, our automata was using only 18.63 MB of memory for about 352,921 nodes, compared to the approach used in GrAVity where the amount was about 345 MB (almost 19 times lower memory usage). However, given that the A-C approach we had used in this test was affected by the existence of many nodes that included redundant information (such as the one used for leaves in subexpressions of regular expressions), the AC-CW approach (which was using even less memory for the C-W automata than the equivalent A-C) consumed only 14.75 MB of memory (lowest amount in our test) for w_{min}=32, compared to almost 315 MB in the previous solution, leading to almost 22 times lower memory usage in this scenario.

The last step of our experiment involved implementing the dual-scanning system on the GPU, based on the observations made *a-priori* that using a combination of the Aho-Corasick and Commentz-Walter automata produces the lowest memory usage, while maintaining the throughput similar to that of the Aho-Corasick RegEx scanning approach alone. In our experiment, in one kernel ran the A-C algorithm and in the other the C-W machine. We limited the depth of the automata at a distance of 8, as discussed before, and the number of total nodes for A-C (running in kernel 1) is shown in Figure 9a (we used different test scenarios involving different values for W_{min} for determining the best compromise between memory usage and throughput performance), while the remaining

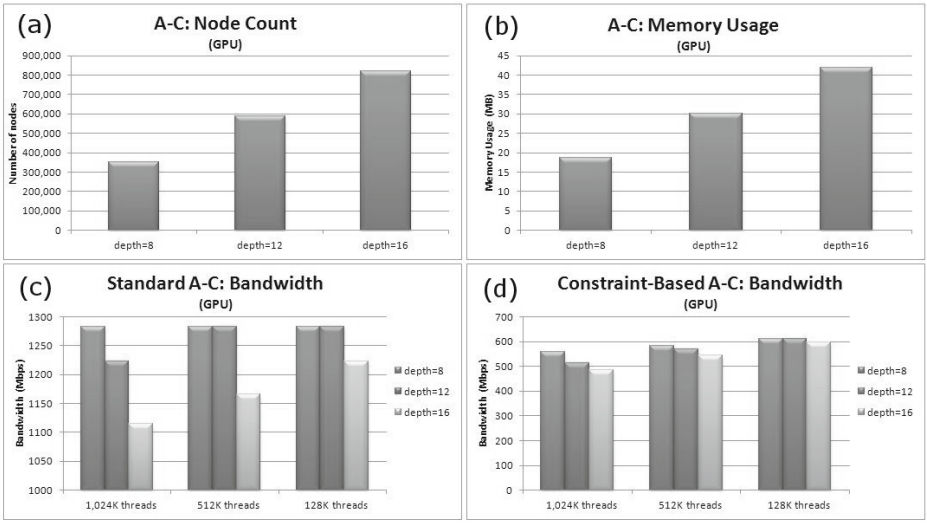

Fig. 8. Results obtained during the experimental phase when using a single A-C kernel and the entire experimental set of ClamAV signatures

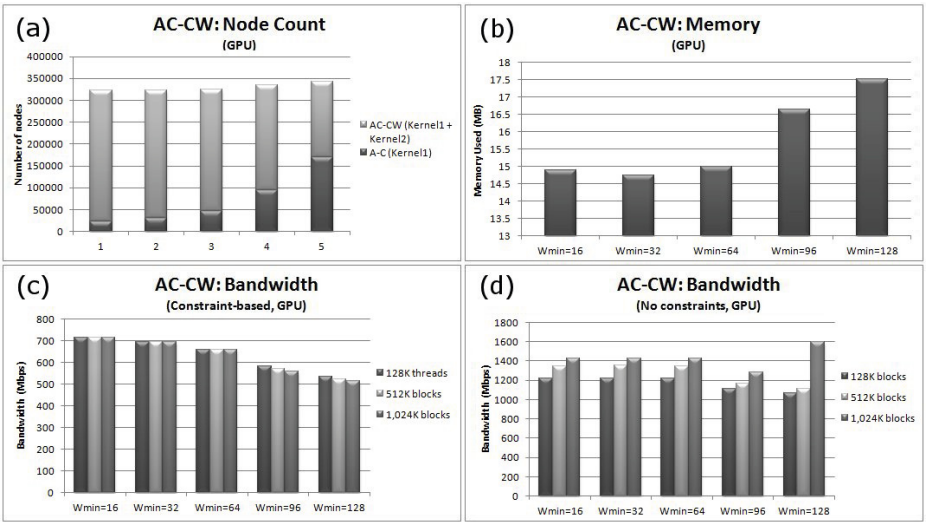

Fig. 9. Results obtained during the experimental phase when using dual-scanning with one A-C kernel scanning for regular expressions and an AC-CW kernel for scanning simple patterns

nodes were being used by regular pattern scanning in kernel 2. The memory used by the GPU implementation is shown in Figure 9b and one can see that while W_{min} increases, the total storage required also increases, since the number of RegEx signatures increases (the data structures used in RegEx scanning occupy of course more space than those used in regular signature matching). Figure 9c shows performance obtained when using RegEx scanning, while Figure 9d shows performance obtained when dropping the RegEx scanning entirely and only performing regular A-C matching in kernel 1. The bandwidth reached 1,420 Mbps when using 1 million threads (almost 34 times faster than the AC-CW throughput on the CPU), for $W_{min} = 128$, but at the expense of memory usage (about 17.5MB required). The best compromise in terms of the $\frac{Bandwidth}{Memory}$ ratio is offered by the AC-CW implementation for $W_{min} = 16$ and $W_{min} = 32$, both having a ratio value of 48.

5 Conclusion

We have proposed a highly-efficient memory compression model for implementing GPU-accelerated virus signature matching and have tested it on the GPU, in single and dual-scanning modes. Experimental results have shown that our model is ideal for implementation in real-time monitoring systems based on GPU hardware acceleration. Tests performed have revealed high performance improvements over previous known attempts, using almost 22 times less memory than related implementations used in GrAVity, while achieving up to 38

times higher bandwidths than single-core CPU implementations. In our experiment, the most memory-efficient approach for implementing virus signature matching relied on dual-threaded scanning, where one thread was scanning regular expressions through the Aho-Corasick machine, and the other thread was scanning regular patterns using the Commentz-Walter algorithm.

Future work includes a GPU-based, WDDM TDR-compliant driver model for performing real-time virus scanning as part of the Windows operating system.

Acknowledgements. This work was partially supported by the grant of the European Commission FP7-REGPOT-CT-2011-284595 (HOST), and Romanian national grant PN-II-ID-PCE-2011-3-0260 (AMICAS). Special thanks go to Sean Baxter and Tim Murray at NVIDIA Corp. for their valuable help and insight on the CUDA architecture.

References

1. Aho, A., Corasick, M.: Efficient string matching: An Aid to blbiographic search. CACM 18(6), 333–340 (1975)
2. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20, 762–772 (1977)
3. Commentz-Walter, B.: A String Matching Algorithm Fast on the Average. In: Maurer, H.A. (ed.) ICALP 1979. LNCS, vol. 71, pp. 118–132. Springer, Heidelberg (1979)
4. Wu, S., Manber, U.: A fast algorithm for multi-pattern searching. Technical Report TR-94-17, University of Arizona (1994)
5. Clam AntiVirus, `http://www.clamav.net`
6. Vasiliadis, G., Ioannidis, S.: GrAVity: A Massively Parallel Antivirus Engine. In: Jha, S., Sommer, R., Kreibich, C. (eds.) RAID 2010. LNCS, vol. 6307, pp. 79–96. Springer, Heidelberg (2010)
7. NVIDIA: NVIDIA CUDA Compute Unified Device Architecture Programming Guide, version 4.1, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`
8. Lee, T.H.: Generalized Aho-Corasick Algorithm for Signature Based Anti-Virus Applications. In: Proceedings of 16th International Conference on Computer Communications and Networks, ICCN (2007)
9. Pungila, C.: A Bray-Curtis Weighted Automaton for Detecting Malicious Code Through System-Call Analysis. In: 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC, pp. 392–400 (2009)
10. Erdogan, O.: Hash-AV: fast virus signature scanning by cache-resident filters. International Journal of Security and Networks 2(1/2) (2007)
11. Lin, P.C., Lin, Y.D., Lai, Y.C.: A Hybrid Algorithm of Backward Hashing and Automaton Tracking for Virus Scanning. IEEE Transactions on Computers 60(4), 594–601 (2011)
12. Cha, S.K., Moraru, I., Jang, J., Truelove, J., Brumley, D., Andersen, D.G.: Split Screen: Enabling Efficient, Distributed Malware Detection. In: Proc. 7th USENIX NSDI (2010)
13. Miretskiy, Y., Das, A., Wright, C.P., Zadok, E.: Avfs: An On-Access Anti-Virus File System. In: Proceedings of the 13th USENIX Security Symposium (2004)

14. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM, vol. 4, pp. 2628–2639 (2004)
15. Snort, `http://www.snort.org/`
16. Zha, X., Sahni, S.: Highly Compressed Aho-Corasick Automata For Efficient Intrusion Detection. In: IEEE Symposium on Computers and Communications, ISCC, pp. 298–303 (2008)
17. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: MIDeA: A Multi-Parallel Intrusion Detection Architecture. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS, pp. 297–308 (2011)
18. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 116–134. Springer, Heidelberg (2008)
19. Liu, C.H., Chien, L.S., Chang, S.C., Hon, W.K.: PFAC Library: GPU-based string matching algorithm. In: PU Technology Conference, GTC (2012)