Journal of Information Security, 2012, 3, 314-318
http://dx.doi.org/10.4236/jis.2012.34038 Published Online October 2012 (http://www.SciRP.org/journal/jis)

State of the Art for String Analysis and Pattern Search Using CPU and GPU Based Programming

Mario Góngora-Blandón, Miguel Vargas-Lombardo
Centro de Investigación, Desarrollo e Innovación en Tecnologías de la Información y las Comunicaciones (CIDITIC) Grupo de Investigación en Salud Electrónica y Supercomputación (GISES), Technological University of Panama, Panama City, Panama
Email: mario.gongora@utp.ac.pa, miguel.vargas@utp.ac.pa

Received July 11, 2012; revised August 17, 2012; accepted August 26, 2012

ABSTRACT

String matching algorithms are an important piece in the network intrusion detection systems. In these systems, the chain coincidence algorithms occupy more than half the CPU process time. The GPU technology has showed in the past years to have a superior performance on these types of applications than the CPU. In this article we perform a review of the state of the art of the different string matching algorithms used in network intrusion detection systems; and also some research done about CPU and GPU on this area.

Keywords: GPU; String Matching; Pattern Matching

1. Introduction

Jack Dongarra [1,2], explains that GPU computing is the use of graphics processing unit together with a CPU to accelerate general-purpose scientific and engineering applications.

String matching algorithms allow string or pattern searching in a given text. These algorithms are used in many applications such as: word processors and in utilities like grep in UNIX [3] based operating systems.

The network based network intrusion detection systems also apply these algorithms, since most of the processing is found in pattern search.

Studies reveal that this process takes about 75% of the total CPU time in modern intrusion detection systems. For this reason the graphic processors, known as GPU, are studied to develop general purpose applications with the GPU [4]. The main reason is that the GPU are specialized in computationally intensive operations and highly parallel operations, required for graphic rendering, therefore are more adequate for data processing than for cache data storage and flow control. In this article we will be discussing different string matching algorithms and their application in intrusion detection systems in CPU as well as in GPU. The article is organized as follows: in Section II we described different string matching algorithms. In Section III we present a state of the art of the different studies in the network intrusion detection systems (NIDS) using string matching algorithms.

Section IV presents the state of the art of the studies done in the GPU field using string matching algorithms. In Section V the conclusions are presented.

2. String Matching Algorithms Used in Intrusion Detection Systems

The objective of the String Matching Algorithms is to locate and identify all the sub-strings, given a set of patterns in a specific text. To make the reading easier lets clarify the following terms when we refer to a string matching algorithm [3]:

- A string is a finite sequence of symbols.
- Where $K = \{y_1, y_2, \cdots, y_k\}$ is a finite set of strings usually called keywords.
- And x is a random string that we can call text string.

These algorithms can be classified in unique or multiple pattern algorithms. This means that if we have k patterns to search, the algorithm is repeated k times. Knuth-Morris-Pratt [5] and Boyer-Moore [6] are some of the most used unique pattern search algorithms. Multiple pattern search algorithms look simultaneously for a set of patterns in a text. This is achieved by applying preprocessing techniques to the set of patterns to get an automaton. The automaton is a state machine that's represented as a table, or a tree or a combination of both. Each text character will be searched once. Some of the multiple pattern search algorithms are: Aho-Corasick [7], Wu-Manber [8] and Commentz-Walter [9]. Next we will describe some of these algorithms.

2.1. Brute-Force Algorithm

The Brute-Force Algorithm [3] consists in comparing

Copyright © 2012 SciRes.

two strings of characters. This algorithm compares from left to right each word the user writes with each letter of the name of the file found inside of the route the user specifies. The process that this algorithm performs is the following [3]:

- Takes the character with which the pattern starts.
- Starts to compare it with each of the text characters, until the first match is found.
- It stops in said position and from there it starts to verify if the pattern matches with the rest of the text.

If the pattern matches, it stops the comparison and then the next file in the route is reviewed. If otherwise, the pattern is not equivalent it will compare again the initial position with the rest of the text characters until a match is found again.

The outer loop is executed at most $n + m - 1$ times, and the inner loop m times, for each iteration of the outer loop. Therefore, the running time of this algorithm is in $O(nm)$ in the worst case. **Algorithm 1** shows the Brute-Force algorithm.

```
Naive-String-Matcher (T,P)
   n = T.length
   m= P.length
   for s = 0 to n – m
       if P[1..m] == T[s + 1..s + m]
       print “Pattern occurs with shift” s
```

Algorithm 1. Brute force.

2.2. Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt [5] developed KMP, an algorithm that has the primary objective to search for the existence of a pattern inside a text string. In the algorithm it is used the information based on the previous data capture mistakes, taking in advantage the information that the search word has on it itself (a table of values is calculated about it), to determine where the next finding could be, without the need of analyzing more than once the character string where it’s been searched.

The KMP locates the start position of a character string inside another. The first step is to locate a string, immediately a table of values is calculated (known as fault or error table). Next the strings are compared with each other and are used to make hops when an error is located.

For example, in a pre-calculated table, both strings start the comparison using an advance pointer for the string that is been searched (pattern), if an error occurs instead of returning to the position next to the first match, it hops the pattern and it places it aligning the character where the error occurred and then it continues verifying the matches. This process is executed until the pattern matches completely with the text. The Knuth-Morris-Pratt algorithm reaches an execution time of $O(n+m)$, which is optimal in the worst case scenario, where each text character and pattern has to be examined at least once. **Algorithm 2** shows the Knuth-Morris-Pratt algorithm.

```
KMP-Matcher(T,P)
      n = T.lenght
      m = P. lenght
      p = Compute-Prefix-Function(P)
      q = 0
      for i = 1 to n
            while q > 0 and P[q + 1] <> T[i]
                  q = p[q]
                  if P[q + 1] == T[i]
    q = q + 1
    if q == m
          print “Pattern occurs with shift”   i - m
          q = p[q]
                return p
```

Algorithm 2. Knuth-morris-pratt.

2.3. Boyer-Moore Algorithm

The Boyer-Moore algorithm [6] is considered the most efficient string processing algorithm on usual applications. A simplified version or the complete algorithm are frequently implemented on text editors for the search and replace commands.

This algorithm consists on aligning the pattern in a text window and comparing from right to left the characters in the window with the ones in the pattern. If there is a mismatch a secure displacement, is calculated, which allows the displacement of the window to in front of the text without the risk of omitting a match. If the start of the window is reached and there are no mismatches, then a match is reported and the window is displaced.

The Booyer-Moore algorithm as presented in the original paper has worst case running time of $O(n+m)$ only if the pattern does not appear in the text. When the pattern does occur in the text the running time of the original algorithm is $O(nm)$ in the worst case. In the best case the complexity of this algorithm is in $O(n/m)$. In **Algorithm 3** we present the Boyer-Moore Algorithm.

```
Boyer-Moore-Matcher(T,P,E)
      n = T.length
      m = P.length
      l = Compute-Last-Ocurrence-Function(P, m, E)
      y = Compute-Good-Suffix-Function(P, m)
      s = 0
      while s <= n – m
            do j = m
     while j > 0 and P[j] = T[s + j]
          do j = j – 1
                if j = 0
         print “Pattern occurs at shift” s
         s = s + y[0]
               else
         s = s + max(y[j],j - l[T[s+j]])
```

Algorithm 3. Boyer-moore.

2.4. Aho-Corasick Algorithm

The Aho-Corasick [7], algorithm it's a search algorithm created by Alfred V. Aho and Margaret J. Corasick. Is a dictionary search algorithm that locates the elements of a finite set of strings (dictionary) within an input text. The complexity of the algorithm is linear to the length of the patterns, plus the length of the searched text, plus the number of matches that the output provides. It should be noted that due to the fact that all the matches are located, there can be a quadratic number of coincidences if each sub-string matches.

The algorithm builds a finite state machine that resembles to a tree with additional links between the different intern nodes. These internal links allow fast transitions between the matching patterns without the need to take steps back. When the dictionary pattern it's known beforehand the building of the automaton can be done once it's off-line and the compiled automaton stored for future use.

In this situation, its execution time is linear in the input length plus the number of matching inputs. In this way, we can conclude that the Aho-Corasick algorithm is $O(n)$ and the pre-processing of the string is linear with the size of the pattern $O(m)$. **Algorithm 4** shows the Aho-Corasick algorithm.

```
begin
    state = 0
    for i = 1 to n
        begin
        while g(state, a1) = fail do state = f(state)
        state = g(state, a1)
        if output(state) <> empty
            begin
      print i
      print output(state)
            end
       end
end
```

Algorithm 4. Aho-corasick.

2.5. Karp-Robin Algorithm

The Karp-Rabin algorithm [10] searches for a pattern in a text by hashing. So we preprocess *p* by computing its hash code, then compare that hash code to the hash code of each substring in *t* if we find a match in the hash codes, we go ahead and check to make sure the strings actually match (in case of collisions). The best case and average case time for this algorithm is in $O(n+m)$ (m time to compute hash (*p*) and *n* iterations through the loop). However, the worse case time is in $O(nm)$, which occurs when we have the maximum numbers of collisions. Karp-Rabin is inferior for single pattern searching to many other options because of its slow worst case behavior. However, it is excellent for multiple pattern searches. If we wish to find one of some large number, say *k*, fixed length patterns in a text, we can make a small modification that uses a hash table or other set to check if the hash of a given substring of *t* belongs to the set of hashes of the patterns we are looking for. In this way, we can find one *k* patterns in $O(km+m)$ time (*km* for hashing the patterns, *n* for searching). In **Algorithm 5** we present the Karp-Robin algorithm.

```
KarpRabin(T, P)
    n = T.length
    m = P.length
    hpatt = hash(P)
    htxt = hash(T[0..m–1])
    for i = 0 to n
        if(htxt == hpatt)
            if(t[i..i + m – 1] == P
            return i
            htxt = hash(T[i + 1..i + m])
        print "not found"
        return -1
```

Algorithm 5. Karp-robin algorithm.

After describing each one of the algorithms in **Table 1**, the execution times of each algorithm are shown. The string matching processing time is defined for the worst case and best case respectively.

In the worst case scenario, the Aho-Corasick algorithm with a $O(n)$ runtime has the best execution time among the analyzed algorithms. Although for simple string matching cases, it does not performs very well, but when there are multiple patterns or pattern matching is done at regular expression level, it is one of the best options.

3. String Matching Algorithm Applied to Intrusion Detection Systems

String processing is a highly intensive computational process; studies demonstrate that the total processing time in a CPU reaches 75% in modern intrusion detection systems. For this reason, is necessary to count on string

Table 1. Comparison between the execution times. String matching algorithms. Where *m* is the length of the string, *n* the length of the text that is been searched, *z* is the amount of string matches and Σ the used alphabet.

Algorithm	Pre-processing	String matching	
		CaseWorst	BestCase
Brute force	No preprocessing	$O(nm)$	$O(n)$
KMP	$O(m)$	$O(nm)$	$O(n)$
Boyer moore	$O(m + \Sigma)$	$O(nm)$	$O(n/m)$
Aho corasick	$O(m)$	$O(n + z)$	$O(n)$
Karp rabin	$O(m)$	$O(nm)$	$O(n + m)$

matching algorithms capable of processing high amounts of information.

Most of the network intrusion detection systems use finite automata and regular expressions for string matching. Both Fisk and Vagese [11] optimized the Boyer-Moore-Horspool algorithm for it to process a set of rules (strings) simultaneously.

An innovative proposal is offered in the Set-Wise Boyer-Moore-Horspool which demonstrated to be faster than the Aho-corasick algorithm and the Boyer-Moore algorithm for pattern sets smaller than 100. At the same time, about this work, Coit, Stainford and MacAlemey [12] implemented a new version of Gunsfield in the Commentz-Wlater algorithm using suffix trees for the heuristics of good suffix. The algorithm was improved in the performance of Snort [13] combining the keyword tree of the Aho-Corasick algorithm with the hop characteristic of the Boyer-Moore algorithm.

In brief, they only measured the performance of a single set-wise algorithm, while Fisk and Vaghese [11] measured multiple algorithms and obtained better measurements without sacrificing the semantic of the rules used by Snort. Tuck [14] optimized the Aho-Corasick algorithm applying bitmap nodes and path compression.

4. State of the Art of Applications Based on String Matching in GPU

The continuous growth of traffic and signature databases make the performance of these systems increasingly more defying and important, is for this reason that the researchers are developing technologies that involve the Graphic Processing Units more every time. The main reason resides in that the GPU specializes in calculation of highly intensive and parallel operations, and therefore, are designed in such a way that more transistors are dedicated to data processing instead of cache data storage and flow control [4]. The following works [15-22], are based in GPU high performance computing.

One of the first works in the GPU field was PixelSnort [15], a version of the intrusion detection system Snort which processed the string matches with a NVIDIA GPU. The GPU programming was complicated, because this video card doesn't support general purpose programming models for GPU. The system coded the Snort rules and packages to textures and did string searches using the Knuth-Morris-Pratt algorithm. However, PixelSnort did not get satisfactory results in normal load conditions. In addition, it doesn't have any multiple pattern matching algorithms adapted to GPU. This represents a serious limitation because the multiple pattern matching algorithms are Snort's by default.

For Marziale [16] the GPU shaping tool performance was evaluated. The system was implemented in a G80 architecture [23] and the results showed that the GPU usage increased substantially in the performance of the digital forensic software analysis, which is based in binary string search. Both Nottingham and Irwin [17] designed gPF: a package classification program based in GPU. In Smith [18] a programmed signature matching system in a GPU G80 [23] based in SIMD (Simple Instruction Multiple Data) was implemented. This system outperforms a Pentium 4 until 9X and a 32 thread system based in Niagara until 2.3X demonstrating that the GPU are promising candidates for signature matching. In their work they evaluated two signatures matching mechanism based in finite automata, these are:

- Deterministic Finite Automaton (DFA [19]: it recognizes the exact type of regular expression.
- Extended Finite Automaton (XFA) [20,24], it reduces the DFA memory requirements.

On the other hand, Vasiliadis and Ioannidis developed GrAVit [21], an antivirus engine, using the architecture of an NVIDIA GPU. They designed, implemented and evaluated pattern matching algorithms, integrated their GPU implementation in the ClamAV [25], antivirus, a very popular open source antivirus. GrAVity reached an end to end performance in the 20 Gbits order, a 100 times the performance of ClamAV using only CPU.

In [4] an intrusion detection system was designed based in Snort, which potentiates the computational power of the video cards (GPU). Its prototype, called Gnort, reached maximum processing rates of traffic of 2.3 Gbits using synthetic tracks, while using an Ethernet interface; it surpassed Snort by a factor of two. Its results demonstrate that modern video cards can be used effectively to accelerate the intrusion detection systems, as well as other systems that involve string matching operations. Seaman and Alexander [22] presented ways to build a special type of regular expressions used by ClamAV in a GPU. Phar and Fernando [26] show a review of some high performance applications adapted to GPU.

This state of the art allowed us to identify string matching algorithms with better performance that the ones described previously. Also, it was demonstrated that exists a very wide research field on GPU, specifically in pattern analysis in intrusion detection systems. These researches have given evidence that the usage of GPU give better performance than the CPU.

5. Conclusion

In this article, we present a state of the art of different algorithms used for pattern matching in network intrusion detection systems. We compare the execution time of these algorithms. Also, we discuss different studies that presented proposals to improve the algorithms based in string matching. Finally, we present a state of the art on some studies on pattern search and package signing

Copyright © 2012 SciRes.

using GPU technology. We can state that in the next years the high performance application development using GPU will increase, displacing CPU eventually.

REFERENCES

[1] S. Tomov, J. Dongarra and M. Baboulin, "Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems," *Parallel Computing*, Vol. 36, No. 5-6, 2010, pp. 232-240. doi:10.1016/j.parco.2009.12.005

[2] NVIDIA, "What Is GPU Computing," 2012. http://www.nvidia.com/object/what-is-gpu-computing.html

[3] D. Gusfield, "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology," Cambridge University, Cambridge, 1997. doi:10.1017/CBO9780511574931

[4] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors," *Proceedings of the* 11*th International Symposium on Recent Advances in Intrusion Detection*, Cambridge, 15-17 September 2008, pp. 116-134.

[5] D. E. Knuth, J. H. Morris Jr. and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, Vol. 6, No. 2, 1977, pp. 323-350.

[6] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Communications of the ACM*, Vol. 20, No. 10, 1977, pp. 762-772.

[7] A. V. Aho and M. J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, Vol. 18, No. 6, 1975, pp. 333-340.

[8] S. Wu and U. Manber, "A Fast Algorithm for Multi-Pattern Searching," Technical Report TR-94-17, University of Arizona, Tucson, 1994.

[9] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," *Automata, Languages and Programming*, Vol. 71, 1979, pp. 118-132.

[10] R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development*, Vol. 31, No. 2, 1987, pp. 249-260.

[11] M. Fisk and G. Varghese, "Applying Fast String Matching to Intrusion Detection," University of California, San Diego, 2004.

[12] C. J. Coit, S. Staniford and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *DARPA Information Survivability Conference and Exposition*, Vol. 1, No. 2, 2001, pp. 367-373.

[13] M. Roesch, *et al.*, "Snort-Lightweight Intrusion Detection for Networks," *Proceedings of the* 13*th USENIX Conference on System Administration*, Seattle, 7-12 November 1999, pp. 229-238.

[14] N. Tuck, T. Sherwood, B. Calder and G. Varghese, "Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection," *INFOCOM*, Vol. 4, 2004, pp. 2628-2639.

[15] N. Jacob and C. Brodley, "Offloading IDS Computation to the GPU," *Proceedings of the* 22*nd Annual Computer Security Applications Conference*, Washington DC, 11-15 December 2006, pp. 371-380.

[16] L. Marziale, G. G. Richard III and V. Roussev, "Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools," *Digital Investigation*, Vol. 4, 2007, pp. 73-81.

[17] A. T. Nottingham and B. Irwin, "gPF: A GPU Accelerated Packet Classification Tool," *Southern African Telecommunications Networks and Applications Conference*, Royal Swazi Spa, 30 August-2 September 2009, pp. 339-344.

[18] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam and C. Estan, "Evaluating GPUs for Network Packet Signature Matching," *IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, 26-28 April 2009, pp. 175-184. doi:10.1109/ISPASS.2009.4919649

[19] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection," *Proceedings of the* 2006 *ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, California, 4-5 December 2006, pp. 93-102. doi:10.1145/1185347.1185360

[20] R. Smith, C. Estan, S. Jha and S. Kong, "Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata," *SIGCOMM Computer Communication Review*, Vol. 38, No. 4, 2008, pp. 207-218.

[21] G. Vasiliadis and S. Ioannidis, "Gravity: A Massively Parallel Antivirus Engine," *Recent Advances in Intrusion Detection*, Vol. 6307, 2011, pp. 79-96.

[22] E. Seamans and E. Alexander, "Fast Virus Signature Matching on the GPU," *GPU Gems*, Vol. 3, No. 1, 2007, pp. 771-783.

[23] J. Owens, "GPU Architecture Overview," *ACM SIGGRAPH*, Vol. 1, No. 2, 2007, pp. 5-9.

[24] R. Smith, C. Estan and S. Jha, "XFA: Faster Signature Matching with Extended Automata," *Proceedings of the* 2008 *IEEE Symposium on Security and Privacy*, Oakland, 18-21 May 2008, pp. 187-201.

[25] T. Kojm, "ClamAV," 2004. http://www. clamav. net

[26] M. Pharr and R. Fernando, "2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)," Addison-Wesley Professional, Boston, 2005.