Multi-Pattern Matching Using GPU

Gayatri Darade[1], Hitesh Kajale[2], Rajesh Shirsath[3], Sumit Rajguru[4]

Department of computer Engineering
Mathoshri college of Engineering and Research centre, Eklahare, Nashik.

***Abstract*— Graphics processing units (GPUs) have attracted a lot of attention due to their cost-effective and enormous power for massive data parallel computing. In this paper, we propose a novel parallel algorithm for exact pattern matching on GPUs. A traditional exact pattern matching algorithm matches multiple patterns simultaneously by traversing a special state machine called an Aho-Corasick machine. Considering the particular parallel architecture of GPUs, in this paper, we first propose an efficient state machine on which we perform very efficient parallel algorithms. Also, several techniques are introduced to do optimization on GPUs, including reducing global memory transactions of input buffer, reducing latency of transition table lookup, eliminating output table accesses, avoiding bank-conflict of shared memory, coalescing writes to global memory, and enhancing data transmission via peripheral component interconnect express. We evaluate the performance of the proposed algorithm using attack patterns from Snort V2.8 and input streams from DEFCON. The experimental results show that the proposed algorithm performed on NVIDIA GPUs achieves up to 143.16-Gbps throughput, 14.74 times faster than the Aho-Corasick algorithm implemented on a 3.06-GHz quad-core CPU with the OpenMP. The library of the proposed algorithm is publically accessible through Google Code.**

Keywords— CUDA, GPU, Pattern Matching, Parallel Algorithm

I. INTRODUCTION

Antivirus programs can detect viruses essentially in two ways: by looking up virus signatures in the executable code, or by profiling the run time behavior, usually through emulation. Virus signatures are data patterns which uniquely describe the identity of a virus or of a family of viruses. The problem of identifying a signature inside binary code is a classic problem of pattern matching.

Although multiple pattern matches algorithms have been proposed, such as the Aho-

Corasick, and Commentz-Walter variants, real-time implementations still pose a challenge to reducing the scanning time of an executable.

Parallel approaches to both single and multiple pattern matching have been researched, however given the low number of cores available on most CPUs usually, the speed increase obtained is still causing significant bottlenecks in real-time implementations. Recent progress in the field of GPU technology, along with NVidia's CUDA architecture have made possible the ability to build hybrid, CPU/GPU-based solutions, that could benefit from the high degree of parallelism offered by the GPU hardware.

II. LITRATURE SURVEY

The pattern matching challenge has a long history in the computing era, dating back to the early stages in computing. The single pattern matching problem aims to find all occurrences of a given, non-empty keyword, into an input string, while later applications have extended the problem to find multiple occurrences of a finite, non-empty set of keywords into an input string.

Pattern matching algorithm can be classified in two types: Single pattern Matching and Multi pattern matching algorithms.

a) Single-Pattern Matching Algorithms

1. The Brute-Force Algorithm

The brute force algorithm is the simplest (and the slowest) of existing variants of multiple pattern matching. The brute-force pattern matching algorithm compares the pattern P with the text T for each possible shift of P relative to T, until either a match is found, or all placements of the pattern have been tried.

2. The Karp-Rabin Algorithm

Michael O. Rabin and Richard M. Karp came up with the idea of hashing the pattern and to check it against a hashed sub-string from the text in 1987. In general the idea seems quite simple, the only thing is that we need a hash function that gives different hashes for different sub-strings. Such hash function, for instance, may use the ASCII codes for every character. The hash function may vary depending on many things, so it may consist of ASCII char to number converting, but it can be also anything else. The only thing we need is to convert a string (pattern) into some hash that is faster to compare. Let's say we have the string "hello world", and let's assume that its hash is hash('hello world') = 12345. So if hash('he') = 1 we can say that the pattern "he" is contained in the text "hello world". Thus on every step we take from the text a sub-string with the length of m, where m is the pattern length. Thus we hash this sub-string and we can directly compare it to the hashed pattern. The Rabin-Karp algorithm has the complexity of O(nm) where n, of course, is the length of the text, while m is the length of the pattern.

3. The Boyer-Moore Algorithm

Boyer-Moore is an algorithm that improves the performance of pattern searching into a text by considering some observations. It is defined in 1977 by Robert S. Boyer and J Strother Moore and it consist of some specific features. The main idea of Boyer-Moore in order to improve the performance are some observations of the pattern. In the terminology of this algorithm they are called good suffix and bad-character shifts.

b) Multi pattern Matching algorithms.

Multi pattern matching algorithm can generally be classified into the following approach.

1. Prefix algorithms

The prefix searching algorithms use a tree to store the patterns, a data structure where each node represents a prefix u of one of the patterns. For a given position i of the input string, the algorithms traverse the tree looking for the longest possible suffix u of t0:::ti that is a prefix of one of the patterns. One of the most well known prefix multiple pattern matching algorithms is Aho-Corasick.

2. Suffix algorithms

The suffix algorithms store the patterns backwards in a suffix tree, a rooted directed tree that represents the suffixes of all patterns. At each position i of the input string the algorithms compute the longest suffix u of the input string that is a suffix of one of the patterns. Commentz-Walter combines a suffix tree with the good suffix and bad character shift functions of the Boyer-Moore algorithm. A simpler variant of Commentz-Walter is Set Horspool, an extension of the Horspool algorithm that uses only the bad character shift function. Suffix searching is generally considered to be more efficient than prefix searching since on average more input string positions are skipped following each mismatch.

3. Factor algorithms

The factor searching algorithms build a factor oracle, a tree with additional transitions that can recognize any substring (or factor) of the patterns. Dawg-Match and Multi BDM [6] were the first two factor algorithms. The Set Backward Oracle Matching and the Set Backward Dawg Matching algorithms are natural extensions of the Back-ward Oracle Matching and the Backward Dawg Matching [4] algorithms respectively for multiple pattern matching.

4.Hashing algorithms

The algorithms following this approach use hashing to reduce their memory footprint, usually in conjunction with other techniques. Wu-Manber is based on the Horspool algorithm. It reads the input string in blocks to effectively increase the size of the alphabet and then applies a hashing technique to reduce the necessary memory space.

5.The Wu-Manber Algorithm

Wu-Manber is a generalization of the Horspool algorithm for multiple pattern matching. It scans the characters of the input string backwards for the occurrences of the patterns, shifting the search. Window to the right when a mismatch or a complete match occurs. To perform the shift, the bad character shift function of the Horspool algorithm is used. During the pre processing phase, three tables are built from the patterns, the SHIFT, HASH and PREFIX tables. SHIFT is the equivalent of the bad character shift of the Horspool algorithm for blocks of characters, generalized for multiple patterns.

Algorithm-

Wu-Manber(P={p 1; p 2 pr },T={t 1; t2tn })
1: Preprocessing
2: Computation of B
3: Construction of hsh table SHIFT and HASH
4: Searching
5: pos = l min
6: While pos n Do
7: I = h1(t pos B+ 1t pos)
8: If SHIFT[i]=0 Then
9: list=HASH[h 2(t pos B+ 1t pos)]
10: Verify all pattern in the list one by one against all the text
11: pos=pos+1;
12: Else pos=pos + SHIFT[i]
13: End of If
14: End of While

6. The Aho-Corasick Algorithm

One of the most widespread algorithms used nowadays to solve the multiple pattern matching problem is that proposed by Aho and Corasick in .The Aho-Corasick algorithm was proposed in 1975 by Alfred V. Aho and Margaret J.Corasick[1] ,an remain to this day this is most effective multi pattern matching algorithm. Aho-Corasick (AC) is a Multi-string matching algorithm, meaning it matches the input against multiple strings at the same time. Multi-string matching algorithms generally pre-process the set of strings, and then search all of them together over the input text. The algorithm works in two step first is building a tree from set of pattern and second is searching text for keywords in previously build tree. Here tree also called State machine. Searching for a keyword is very efficient, because it only moves through the states in the state machine. If a character is matching, it follows goto() function otherwise it follows fail()function[1].In the Aho-Corasick automaton the actions are determined by three functions:

1. The goto function g(q,a) is the next state from the current state q, on receiving symbol 'a'.
2. The failure function f(q). for q* 0, is the next state in case of a mismatch.
3. The output function out(q) gives the set of patterns found at state q.

Algorithm1: Pattern matching machine

Input: A text string x = a 1a 2::: a n where each a is an input symbol and a pattern matching machine M with goto function g, failure function f, and output function output, as described above.
Output : Locations at which keywords occur in x.
Method
1: begin
2: state 0
3: for i 1 until n do
4: begin
5: while g (state,a i) = fail do state f(state)
6: state g (state, ai)
7: if output (state)6= empty then
8: begin
9: print i
10: print output (state)
11: end
12: end
13: end

Algorithm 2 : Construction of the goto function

Input: Set of keywords K = {yl ; y2; ::::: yk }.
Output: Goto function g and a partially computed output function
output.
Method
We assume output(s) is empty when state s is first created, and g(s, a) = fail if a is undefined or if g(s, a) has not yet been defined. The procedure enter(y) inserts into the goto graph a path that spells out y.
1: begin
2: new state 0
3: for i 1 until k do enter(yi)
4: for all a such that g(0, a) = fail do g(0, a) 0
5: end

6: procedure enter(a 1; a2am):
7: begin
8: state 0; j 1
9: while g (state, a j)6= fail do
10: begin
11: state g (state, a j)
12: j j + l
13: end
14: for p j until m do
15: begin
16: new state + 1
17: g (state, a p)new state
18: state new state
19: end
20: output(state) { a 1a 2a m }
21: end

Algorithm 3: Construction of the failure function.

Input: Goto function g and output function output from Algorithm 2
Output: Failure function f and output function.
Method
1: begin
2: queue empty
3: for each a such that g(0, a) = s6=0 do
4: begin
5: queue queue [s
6: f(s) 0
7: end
8: while queue 6= empty do
9: begin
10: let r be the next state in queue
11: queue queue - {r}
12: for each asuch that g(r, a) = s6=fail do
13: begin
14: queue queue [{s}
15: state f(r)
16: while g (state, a) = fail do state f(state)
17: f(s) g(state, a)
18 output(s) output(s)[output(f(s))
19: end
20: end
21: end

III. PROPOSED SYSTEM

There are various measures on which performance of pattern matching system is depends, such as size of patterns, length of patterns, size of packet data, length of packet data ,size of constructed DFA. Performance analysis of the visual cryptography schemes are likely to be examined on the basis of Time required to search pattern in given data, Time required to copy data from CPU to GPU, Time required to construct DFA, position of found pattern in packet data. All this factor are important in pattern matching system. Since an improvement to any of these factors can result in a more effective Pattern matching system.

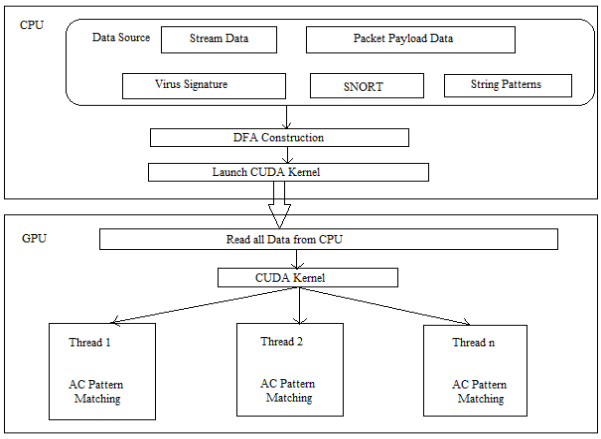

Fig. 1.System Architecture

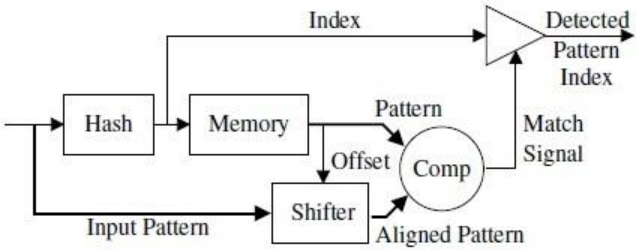

Fig. 2. Pattern Detection Module

IV. Result

The result of the Multi-Pattern Matching using GPU is as follows:

1. We came to the conclusion that the system with serial/ single pattern matching system which works on the CPU requires large amount of time for the pattern matching and detecting the virus signature.
2. Another system we studied in our project is Multi-Pattern matching using CPU with OpenMP. Which works on multiple cores of the system and is comparatively much more time saving than the previous system?
3. We studied and developed third system which is our main project concept i.e. Multi-Pattern Matching using GPU. The result of the system is really impressive in terms of the time required for the total operation. It takes a negligible time to perform tasks such as building DFA, matching patterns, identifying virus signatures etc.

REFERENCES

[1] A.V. Aho and M.J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. Communications of the ACM, 18(6):333-340, 1975.
[2] X. Chen, B. Fang, L. Li, and Y. Jiang. WM+: An Optimal Multi-pattern String Matching Algorithm Based on the WM Algorithm. Advanced Parallel Processing Technologies, pages 515-523, 2005.

[3] GNU Grep. Webpage containing information about the gnu grep search utility. Website, 2012. http://www.gnu.org/software/grep/.
[4] G. Navarro and M. Raffinot. Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, 2002.
[5] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Ryt-ter. Fast Practical Multi-pattern Matching. Information Processing Letters, 71(3-4):107 - 113, 1999.
[6] M. Crochemore and W. Rytter. Text Algorithms. Oxford University Press, Inc.,1994.
[7] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li. MDH: A High Speed Multi-phase Dynamic Hash String Matching Algorithm for Large-Scale Pattern Set. Information and Communications Security, 4861:201-215, 2007.
[8] Snort. Webpage containing information on the snort intrusion prevention and detection system. Website, 2010. http://www.snort.org/
[9] S. Dori and G.M. Landau. Construction of Aho Corasick Automaton in Linear Time for Integer Alphabets. Information Processing Letters, 98(2):66-72, 2006.
[10] N Wilt. The CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley Professional, 2013 CUDA Zone. Official webpage of the nvidia cuda api. Website, http://www.nvidia.com/object/cudahome.html
[11] A. Tumeo, S. Secchi, and O. Villa, "Experiences with String Matching on the Fermi Architecture," Proc. 24th Int'l Conf. Architecture of Computing Systems, 2011.
[12] G. Vasiliadis, M. Polychronakis, S. Antonatos, E.P. Markatos, and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection," Proc. 12th Int'l Symp. Recent Advances in Intrusion Detection, 2009.
[13] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA Pattern Matching on GPGPU Devices," SIGCOMM Computer Comm. Rev., vol. 40, pp. 20-26, 2010.
[14] GCC, http://gcc.gnu.org/, 2013.
[15] OpenMP, http://openmp.org/wp/, 2013.