Analytical and Comparative Study of Pattern matching with Knuth-Morris-Pratt algorithm in CPU and GPU

Rahul (Grad Student) ; Dr. Mary Adedayo (Prof)
Computer Science Department, The University of Winnipeg

I. INTRODUCTION

String matching is an important problem in text processing and is commonly used to locate the appearance of one dimensional arrays (the so-called pattern) in an array of equal or larger size. The string matching problem can be defined as: let Σ be an alphabet, given a text array T[n] and a pattern array P[m], report all locations [i] in T where there is an occurrence of P, i.e. T[i + k] = P[k] for m ≤ n. String matching is the classical and existing problem in the real world whose application is implemented in every other domain or problem. In general terms, one or several strings called "Pattern" is to be searched with a well-built string or text. Some of the applications of string matching are Spell checkers, spam filters, Intrusion Detection System, Search Engines, Plagiarisms, Information retrieving system, Bioinformatics, Digital forensics and many more applications. Combinatorial pattern matching solves the issue of matching and searching patterns such as graphs, arrays and point sets. The area of string matching is always expected to grow due to increase in demand of speed and efficiency in field of biology, information retrieval, pattern recognition, compiling, security, program analysis and data compression.

II. RELATED WORKS

There exists various traditional pattern matching methodologies each having their scope and environment where they perform best as compare to the others. Some of the methodologies are like Naïve Brute force, Boyer Moore, Dynamic algorithms.

Naive Brute force- It is one of the basic algorithm of pattern matching which completely scans the text and pattern multiple times therefore having the complexity of O(m n) where m is the length of the pattern P and n is the length of Text T.

Rabin Karp Algorithm- It uses hashes to find any one of a set of pattern strings in a text. Rabin karp can rapidly search through a paper for instances of sentences from the given source material. Average and best case complexity of this algorithm is O(m + n) and worst case complexity is O(nm) where m is length of pattern P and n is the length of text T.

Boyer-Moore Algorithm- It performs larger shift-increment whenever mismatch is detected. It differs from Naïve in the way of scanning. It scans the string from right to left; unlike Naive i.e. P is aligned with T such that last character of P will be matched to first character of T. If character is matched then pointer is shifted to left to very rest of the characters of the pattern. If a mismatch is detected at say character c, in T which is not in P, then P is shifted right to m positions and P is aligned to the next character after c. If c is part of P, then P is shifted right so that c is aligned with the right most occurrence of c in P. The worst complexity is O (m+n)

Dynamic Programming Algorithm- It is the oldest and widely used algorithm. It includes Needleman Wunsch and Smith waterman algorithm. These are much more complex than exact pattern matching involves solving successive recurrence relations which means smaller problems are solved in succession to solve the main problem.

Smith Waterman Algorithm: It is a well-known algorithm for performing local sequence alignment; that is, for determining similar regions between two nucleotide or protein sequences. Instead of looking at the total sequence, the Smith Waterman algorithm compares segments of all possible lengths and optimizes similarity measure

Needleman Wunsch Algorithm: It performs a global alignment on two sequences. It is commonly used in bioinformatics to align protein or nucleotide sequences. The Needleman Wunsch algorithm is an example of dynamic programming, and was the first application of dynamic programming to biological sequence comparison

Algorithm using Levenshtein Distance: Levenshtein distance is a metric for measuring the amount of difference between two sequences (i.e. an edit distance). The term edit distance is often used to refer specifically to Levenshtein distance. The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.

III. DISCUSSIONS

Pattern matching is the process of verifying that a specific pattern is present in a text having the same or a larger length. The pattern and the text are usually represented as a one dimensional array. Let n denote the length of the text S, m the length of the pattern, and P the pattern. Furthermore, we assume at all time that the length of the pattern is smaller or equal to the length of the text; that is, m <= n. In order to give an idea of what pattern matching is, the brute force algorithm, and the KMP algorithm are described below.

The brute force algorithm is one of the simplest approaches that works as follows. It compares the first element of the pattern P with the first element of the text S and tries to match the successive position until the entire pattern is matched.

Only after complete match or a mismatch does the algorithm move to the next position. This algorithm does not require any pre-processing, but it exhibits slow performance due to its computational inefficiency. The time complexity of this algorithm is O(nm) where n is length of the text and m is the length of the pattern.

String_Matching (text, pattern):

```
    patLen := pattern Size
    strLen := string size

    for i := 0 to (strLen - patLen), do
        for j := 0 to patLen, do
            if text[i+j] ≠ pattern[j], then
                break the loop
    done

    if j == patLen, then
        display the position i, as there pattern found
```

The KMP algorithm has been designed **to reduce unnecessary comparisons**, such as those performed in the brute force algorithm. The KMP uses the LPS (Longest proper prefix which is also suffix) table to avoid comparing the text and the pattern at superfluous positions and skip the characters that have already been matched. This allows the algorithm to be significantly **more efficient and hence faster** than the brute force algorithm described previously. The algorithm operates as follows.

The LPS table is initialized for the pattern being searched. The LPS function is built by analyzing the pattern of interest and the repetition of its own first characters as shown in Table. This enables the search function to calculate the number of characters that should be skipped during the searching phase.

Position	0	1	2	3	4	5	6
Character	A	A	A	B	B	A	B
LPS index	0	1	2	0	4	1	4

a) KMP ALGORITHM

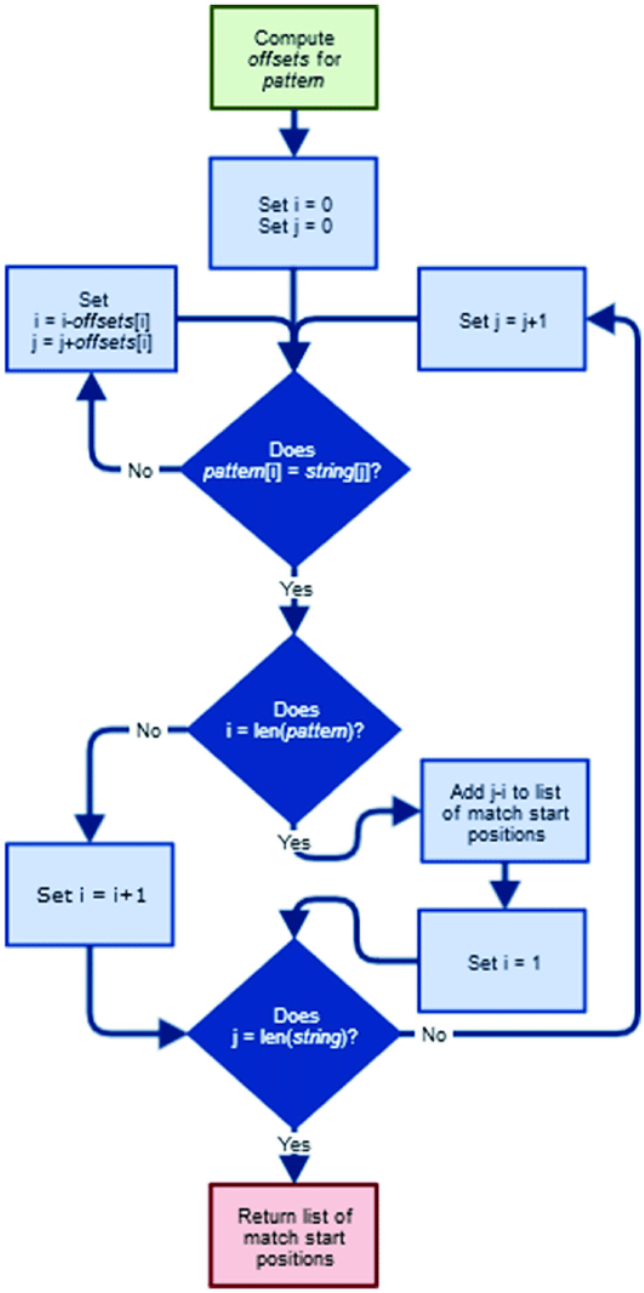
The algorithm calculates prefix array. Input to the array is pattern, length of pattern m, array prefArray/ LPS table to store calculated prefix values

```
calculatePrefixArray(pattern, m, prefArray )
  length := 0
  prefArray[0] := 0

  for all character index 'i' of pattern, do
    if pattern[i] = pattern[length], then
      increase length by 1
      prefArray[i] := length
    else
      if length ≠ 0 then
        length := prefArray[length - 1]
        decrease i by 1
      else
        prefArray[i] := 0
```

```
KMP( text, pattern)
  n := size of text
  m := size of pattern
  calculatePrefixArray(pattern, m,
prefArray)

  while i < n, do
    if text[i] = pattern[j], then
      increase i and j by 1
    if j = m, then
      print the location (i-j) as there is the
pattern
      j := prefArray [j-1]
    else if i < n AND pattern[j] ≠ text[i] then
      if j ≠ 0 then
        j := prefArray [j - 1]
      else
        increase i by 1
```



Flowchart image by https://www.researchgate.net/figure/A-flowchart-of-the-Knuth-Morris-Pratt-string-searching-algorithm-KMP-is-a_fig2_303799704

b) ALGORITHM DESCRIPTION

The preprocessing algorithm processes the array with of longest proper prefix which is also the suffix. The array size created is same as size of pattern. The array indicates longest proper prefix which is also a suffix. The proper prefix is prefix excluding the whole string.
Proper prefix for “ABABA” are “A”, “AB”, “ABA”, “ABAB”
Suffix for “ABABA” are “A”, “BA”, “ABA”, “BABA”, “ABABA”
The array stores matching proper prefix which is also a suffix of the sub-pattern pattern[0…i]. Therefore prefArray[i] is the longest prefix of pattern[0..i] which is also a suffix of pattern[0..i]. This algorithm’s pre-computation creates an array with information about how the keyword matches against shifts of itself.
The below table is generated for a pattern “AABAACAABAA” following the steps of an algorithm

Position	0	1	2	3	4	5	6	7	8	9	10
Character	A	A	B	A	A	C	A	A	B	A	A
LPS index	0	1	0	1	2	0	1	2	3	4	5

The searching algorithm doesn’t match the character already matched in the pattern and skip the execution of matching again and again which is done is brute force algorithm. It uses prefArray[] generated from preprocessing algorithm to decide the number of characters needed to be missed while matching the pattern.
The algorithm starts matching the pattern[j] with j=0 with the characters of current window of text. The algorithm works in such a way that j starts with 0 and increment only if match occurs. The algorithm keeps matching characters of text[i] and pattern[j] and keep incrementing I and j while pattern[j] and text[i] keep matching.
When there is a mismatch

- It is clear that characters pattern[0…j-1] match with text[i-j,….i-1]
- It is also clear that prefArray[j-1] is the count of characters of pattern[0….j-1] that are both prefix and suffix
- We can deduce that we don’t need to match characters of prefArray[j-1] with the characters of text[i-j…i-1] because we know that these characters will anyway match

The example of the above searching algorithm is described below

Text[]= “XXXXYXXXYX”
Pattern[]=”XXXX”

Position(i)	0	1	2	3
Character	X	X	X	X
prefArray[i]	0	1	2	3

In first iteration i=0 and j=0
Text[]= “**X**XXXXYXXXYX”
Pattern[]=”**X**XXX”
As we can see that both text[i] and pattern[j] matches, the algorithm increments i and j. Therefore i becomes 1 and j becomes 1 in next iteration.

In second iteration i=1 and j=1
Text[]= "X**X**XXXYXXXYX"
Pattern[]="X**X**XX"
As we can see that both text[i] and pattern[j] matches, the algorithm increments i and j. Therefore i becomes 2 and j becomes 2 in the next iteration.

In third iteration i=2 and j=2
Text[]= "XX**X**XXYXXXYX"
Pattern[]="XX**X**X"
As we can see that both text[i] and pattern[j] matches, the algorithm increments i and j. Therefore i becomes 3 and j becomes 3 in the next iteration

In second iteration i=3 and j=3
Text[]= "XXX**X**XYXXXYX"
Pattern[]="XXX**X**"
As we can see that both text[i] and pattern[j] matches, the algorithm increments i and j. Therefore i becomes 4 and j becomes 4.

Now i=4 and j=4
Since j has reached to the size of pattern that is j = m now, pattern matched in the text and the reset is made to j to check whether there is more matches in the string and reset is made in such a way it doesn't compare the characters already compared.
Now j = prefArray[j -1]
As j=4, we make j as the value of prefArray[3] = 3
Therefore i=4 and j=3
Text[]= "XXXX**X**YXXXYX"
Pattern[]="XXX**X**"
As we can see that both text[i] and pattern[j] matches, the algorithm increments i and j. Therefore i becomes 5 and j becomes 4.

Now i= 5 and j=4
Since j has reached to the size of pattern that is j = m now, pattern matched in the text and the reset is made to j to check whether there is more matches in the string and reset is made in such a way it doesn't compare the characters already compared.
Now j = prefArray[j -1]
As j=4, we make j as the value of prefArray[3] = 3
Therefore i=5 and j=3
Text[]= "XXXXX**Y**XXXYX"
Pattern[]="XXX**X**"
As we can see text[i] and pattern[j] doesn't matches therefore the algorithm change only j to prefArray[j-1]= prefArray[2]=2 and starts matching similarly
In above similar way, algorithm moves forward until it traverse whole text string.

The KMP algorithm knows what the appropriate shift should be, and may shift over multiple positions without missing any potential matches. The trick to doing this correctly is in the pre-computation step where the prefix function for the keyword is built into an array. If we define x as the first r characters of x that have been matched at any point, the prefix function array table at position r (table[r]) contains the length of the longest prefix of x that is a proper suffix of x r. Using this information the shift to the next position is always possible to calculate as (the current position) + (the number of matched characters before the mismatch (r)) - (table[r]).

c) COMPLEXITY ANALYSIS

COMPLEXITY ANALYSIS OF PREFIX TABLE

The run time of the algorithm is decided by both the algorithms which includes calculating the array for longest proper prefix which is also a suffix. In our case, calculatePrefixArray calculates the pre-processing array which is further used in pattern matching. The algorithm for calculating pre-processing table iterates for length of pattern number of times. All the other lines in our algorithm takes linear time. Therefore cost associated with computation lines of our algorithm except the loop is O(1). The cost associated with loop is O(length of pattern). Assuming that length of pattern is m, the cost is **O (m)**.

COMPLEXITY ANALYSIS OF MATCHING ALGORITHM

The actual string matching algorithm makes use of processing prefix table algorithm and also iterates through whole text string to match the text with the pattern. The algorithm is given below

```
KMP( text, pattern)
  n := size of text                                          } Takes constant amount of time O(1)
  m := size of pattern                                       }
  calculatePrefixArray(pattern, m, prefArray)  ------------> O(m) (discussed in above paragraph)
  while i < n, do  ----------------------------------------> n times
    if text[i] = pattern[j], then                            }
      increase i and j by 1                                  }
    if j = m, then                                           }
      print the location (i-j) as there is the pattern       }
      j := prefArray [j-1]                                   }  n times
    else if i < n AND pattern[j] ≠ text[i] then              }  (cost associated is constant)
      if j ≠ 0 then                                          }
        j := prefArray [j - 1]                               }
      else                                                   }
        increase i by 1                                      }
```

Therefore the complexity is sum of complexity of both the algorithms used in KMP pattern matching. The run time complexity is given by **O(m + n)** where m is length of the pattern to be found and n is length of text which is traversed.

SPACE COMPLEXITY ANALYSIS

The space overhead of KMP string matching algorithm is due to the extra space used by the algorithm to calculate longest proper prefix which is also a suffix. The space is used to store the values of prefix function. In our case, we used an array of size of the pattern to be found to store the result values of preprocessing. The size of pattern in the given is 4 and array to store preprocessing result is also 4. Therefore the space complexity is upper bounded by size of the pattern which is m in our case. It clearly shows that the space complexity of KMP is **O(m)**

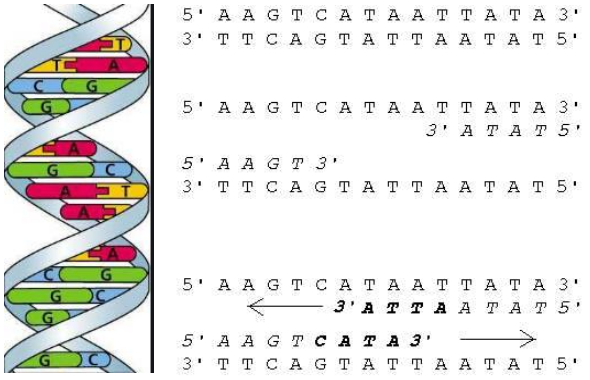
Position(i)	0	1	2	3
Character	X	X	X	X
prefArray[i]	0	1	2	3

IV. REAL WORLD APPLICATIONS OF KMP

Pattern matching is one of the important algorithm to solve many real world problems. Many applications uses pattern matching methods directly or indirectly. Every search engine uses different string matching algorithm or enhanced versions of existing string matching algorithm for the efficient search behavior. In real applications list of items often are records of list implemented as an array of objects. The search engine deals with different type of data such as videos, audios, documents and matches the user inputted pattern with the existing data for efficient matching result. Apart from search engine, let us discuss all the real world application where string matching algorithm especially KMP algorithm (as it is focus of our research) is applied in real world application.

A) COMPUTATIONAL BIOLOGY

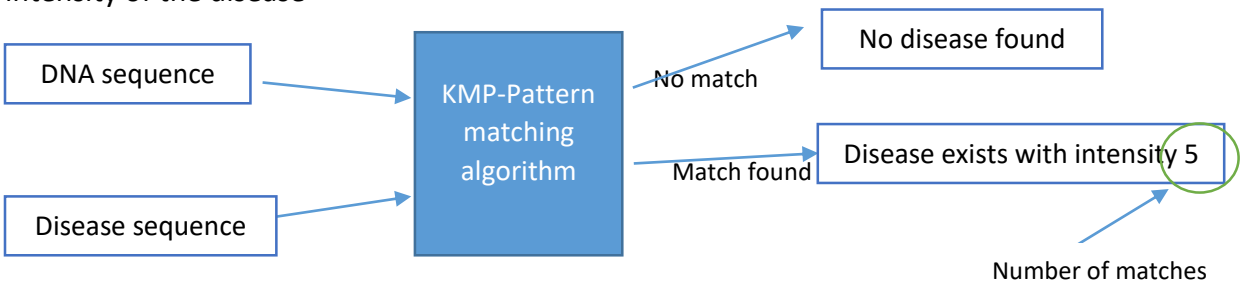
In Computational Biology, Deoxyribonucleic Acid (DNA) is a nucleic acid that contains genetic instructions. The four bases in DNA are Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). Thymine and adenine always come in pairs. Likewise, guanine and cytosine bases come together too. Every human has his/her unique genes. Genes are made up of DNA. DNA and protein sequences can be depicted by long texts constructed by specific alphabets such as: (A, C, G, T), that represent genetic code of living organisms. Although these sequences can be very long, scientists search for specific parts over



those texts, in order to assemble the DNA chain from the pieces that were obtained by experiments, look for specific characteristics inside them or compare different parts. These requirements are modeled as searching for patterns in a large text file. However, on such applications approximate matching is much more useful than exact matching, as the experimental measures have errors, and even if they are correct they may have small differences, due to mutations and evolutionary alterations. Every disease has its own words and mutated sequence which can be searched in DNA for its existence.

DISEASE DETECTION IN DNA USING KMP

The occurrence of un-wanted or mutated words cause the disease in DNA sequence. Every disease has its own words sequences and the intensity of disease depends on occurrence of the mutated sequence in the DNA. Especially cancer is caused due to the uncontrolled growth of cells is the result of alterations or mutation in genetic material. When particular sequence of disease is known to us, we can trace the disease existence using pattern matching algorithm in DNA sequence. We can also find the number of sequences existing in the DNA sequence to decide the intensity of the disease



B) COMPUTER SECURITY SYSTEM

The several areas of computer security uses pattern matching techniques in order to achieve various demanding activities such as intrusion detection, file hash matching, virus scans and spam filtering. In Intrusion Detection System data packets that contain intrusion related keywords are found by applying string matching strategy. All the malicious code is stored in the database and every incoming data is compared with stored data. If match found then alarm is generated. It is based on exact string matching algorithms where we have to capture each and every intruded packet and they must be detected.

INTRUSION DETECTION USING KMP

Network Intrusion detection system detects the intrusion in network traffic. Snort is free and open source network intrusion detection system built on the pattern matching algorithm. Snort is rule driven language that performs the pattern matching.

Structure of Intrusion Detection System rules

Rule Header	Rule Options

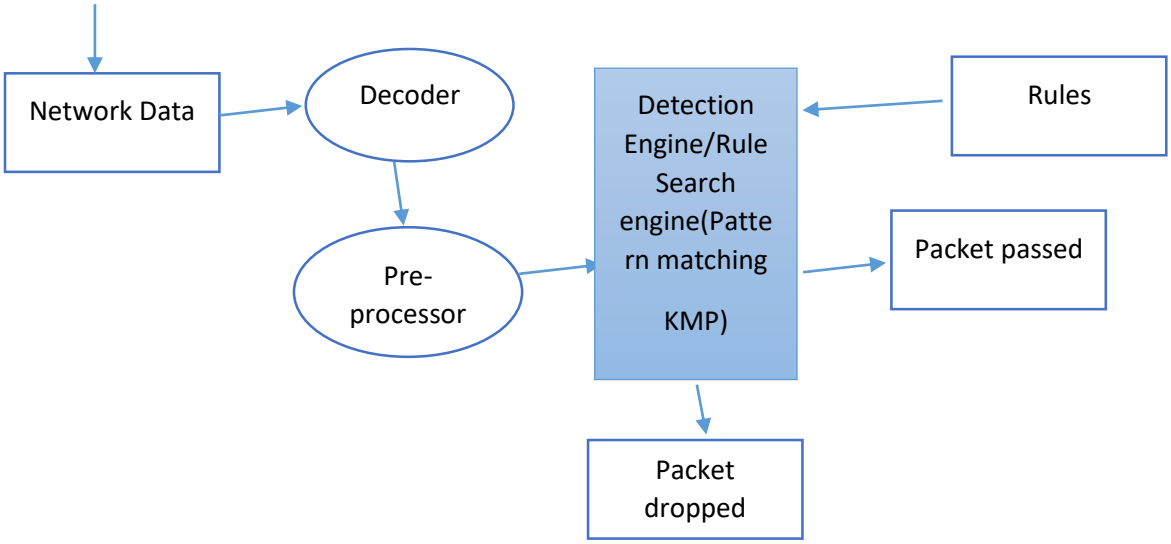
Every rule of IDS has two logical parts which is rule header and rule options. The rule header has information related to the action a rule takes and also, it contains criteria which is followed while matching rule with the data packet. Rule options part contains an alert message and information about which part of packet should be used to generate the alert message. It also contains additional criteria for matching rule against data packets. A rule may detect one type or multiple types of intrusion activity.

The general structure of rule header is shown below

Action	Protocol	Address	Port	Direction	Address	Port

The Action part determines type of action taken when criteria are met and rule is exactly matched against data packet. The protocol part is used to apply rule on particular protocol only. The address part defines source and destination addresses. Port part has information related to ports of source and destination. Direction in the rule header shows direction of packet between two addresses.

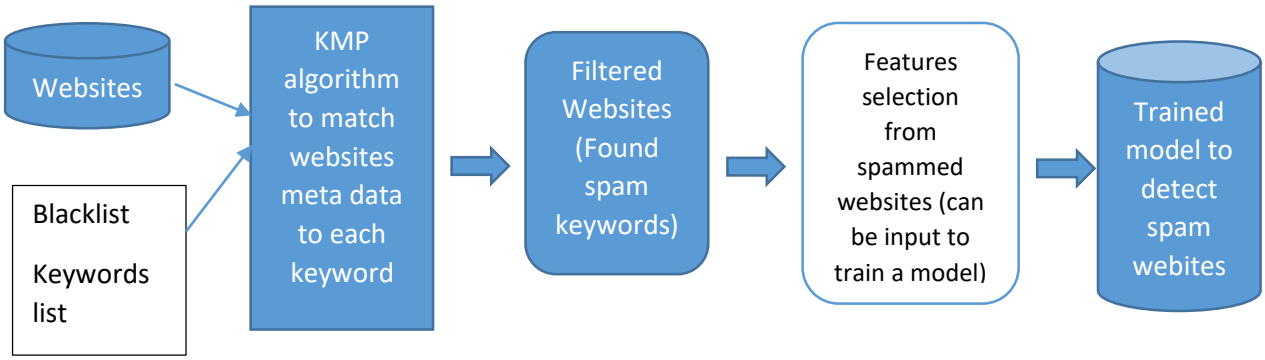
How snort works and which part uses pattern matching algorithm



C) SPAM WEBSITES DECTECTION USING KMP

Spam websites are very common in today advancement of web and technologies and it is becoming hard to differentiate and discover that website we are using may be spam. There are multiple ways to create an algorithm to detect a website which is spam. However, we can use KMP pattern matching algorithm to find the spam website. Application using KMP algorithm to match pattern can be useful in extracting patterns in the website and find the same. Keywords which are commonly used in spam websites can be one of the effective parameters to identify the nature of spam websites. For an example, most of the spam websites use specific keywords, KMP can be used to find the matching patterns with website data by using dictionary of specific spam words. Therefore, it can help further analysis of features using decision tree or

classification algorithm. Once websites with spam keywords are detected, it can help to train a model after feature selection and extraction.

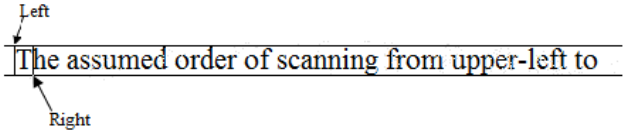


D) OPTICAL CHARACTER RECOGNITION

Optical Character Recognition is process of converting the printed document or scanned document into the computer understood ASCII characters. KMP algorithm plays important role in this field due to its important feature of storing longest proper prefix which is also a suffix in a reference table and optical character is recognized using this algorithm. The range of applications include postal code recognition, automatic data entry into large administrative systems, banking, automatic cartography and reading devices for blind. Accuracy, flexibility and speed are main features of good optical character recognizer. KMP can play important role in efficient optical character recognizer.

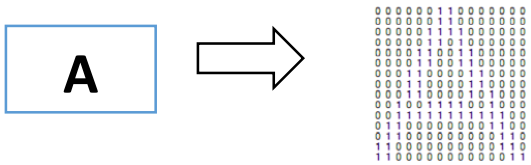
The components of OCR System are

i. Feature Extraction – It is process of getting information about object or group of objects in order to do classification. The document is scanned for the initial darker pixel to be named as top of row and next blank line is detected as bottom of row. The area between them defines characters to be recognized.

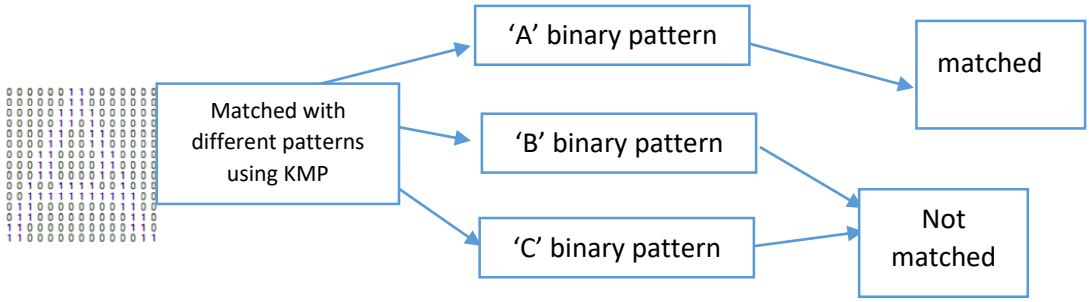


ii. The character from scanned image is obtained and normalized into particular pixel size such as 15 pixel each side.

iii. Now image of character is mapped against binarized array on both the dimensions. Darker place is substituted by 1 and lighter place is replaced with 0 as given below



iv. There are many ways to match the pattern of binarized array with existing pattern and find which character it matches. One of the optimized method to verify this character is KMP algorithm which matches pattern against text data given. The above array of pixels can be converted to 1D array and matched with existing patterns of different characters



V. EXPERIMENT ANALYSIS(CPU Version)

All the below experiments codebase,input data and output results are available at https://github.com/RahulSindwani1/KnuthMorrisPratt_PatternMatching

ANALYSIS OF DETECTION OF DISEASE IN DNA

Machine Configuration:

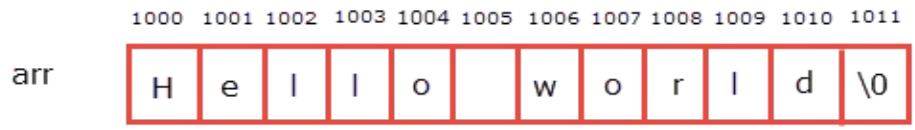
In order to perform the analysis, the execution time of algorithm with various size of inputs was measured. To match a pattern of DNA with the existing text file with different size of input, the implementation of an algorithm was done in C++ language. The system used to do the experiments is Intel i5 Core processor running on 2.4 GHz, 4 GB RAM, 64 bit Window 8.

Input Data Set:

To generate the input data of DNA having {A,C, G,T}, the implementation was done in the python code. Using python code , multiple text files were generated with varying sizes to test the efficiency of algorithm of various data sizes. The same test data was used for both CPU and GPU (discussed in next section). The text file generated were with text size of 1000, 10000, 100000, 200000, 500000, 1000000, 3000000, 5000000 and many more. The implementation of generating input data and input data set used for all the experiments is attached with this report

Data Structure: A character array is a sequence of characters, just as a numeric array is a sequence of numbers. A typical use is to store a short piece of text as a row of characters in a character

vector. The text file is read as a string and then it is mapped to a character array to execute KMP algorithm on it to find the matching patterns



12 bytes of memory is allocated to store 12 characters

EFFICIENCY ANALYSIS

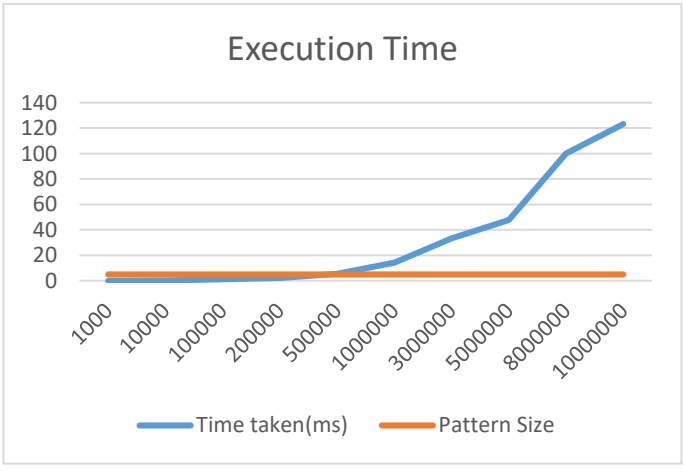
The methodology used was the simulation based in order to observe the efficiency of our implementation. Simulation method deals with turning the features of the system on and off to test and observe independent behavior of the features.

Same pattern size but different input text data size

In the given experiment, the pattern size was kept constant as 5 and input text data of DNA sequence was varied. The main focus of experiment is to compute the time metric for different set of inputs and eventually observing the efficiency of KMP pattern matching in disease finding in DNA. The graph is plotted on basis of above observations. It can be clearly seen that with the increase in size of input text in which pattern is searched, the execution time increases. The execution time with 100000 size of text is 1.22138 ms and it is 2.16136 ms for 200000 input text. Therefore, execution time increases with input size.

Experiment clearly shows that **with increase in size of input text which is n, the time complexity of a KMP algorithm also increases.** KMP has time complexity as O (n + m). As we can see in our experiment, the pattern size m was kept constant as 5. The time increased with increase in number of input text and decreased with less input size. **The running time is directly proportional to input size which is proved in our experiment.**

Text file size	Pattern Size	Time taken in ms	Match found
1000	5	0.011975	3
10000	5	0.109479	8
100000	5	1.22138	87
200000	5	2.16136	186
500000	5	5.40597	496
1000000	5	10.7581	963
3000000	5	33.3151	3016
5000000	5	47.8544	4767
8000000	5	99.9739	7819
10000000	5	123.24	9831



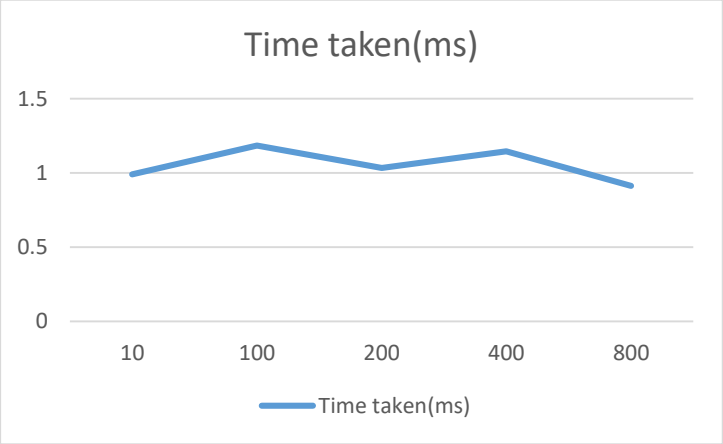
Different pattern size but same input text data size

In the given experiment, the input text data size of DNA sequence was kept constant as 100000 and pattern size was varied.

The main focus of experiment is to compute the time metric for different set of patterns and eventually observing that how the efficiency of KMP pattern matching for disease finding in DNA depends on disease pattern or in other words how efficiency depends on pattern size.

The table clearly shows that there is no certain behavior of time consumed on change of pattern size. The running time is not directly or inversely proportional to pattern size. But still, if we compare this table with the above experiment result table**, we can derive that size of pattern adds to the complexity of KMP algorithm**

Text file size	Pattern Size	Time taken in ms
100000	10	0.99046
100000	100	1.18546
100000	200	1.03492
100000	400	1.14654
100000	800	0.913896
100000	1600	3.11503

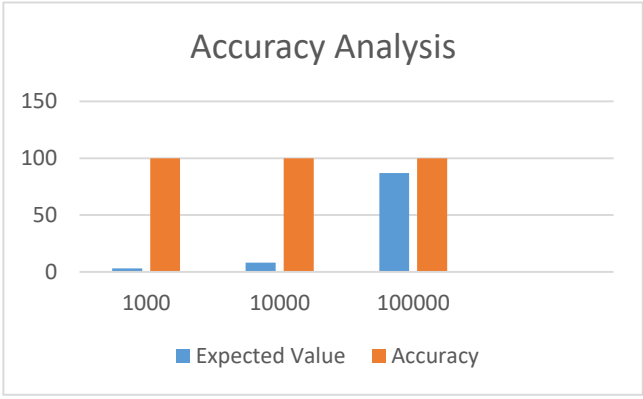


We can clearly derive from above two experiments that running time complexity depends on both input text data size and pattern size. Therefore it is given as O(m + n)

ACCURACY ANALYSIS

To find the accuracy of matches found by the implementation of KMP pattern matching algorithm, number of patterns were intentionally added into the search input text in above experiment input files and the result was verified with the number of DNA matching in the text file with the result of our algorithm. They all correctly matched with expected result.

Text file size	Expected value	Match found	Accuracy
1000	3	3	100 %
10000	8	8	100 %
100000	87	87	100 %



It was found in the observations that KMP resulted in accurate number of pattern searches as compare to expected value. It showed 100 % accuracy in case finding disease undesired behavior or disease in DNA of different sizes of text files.

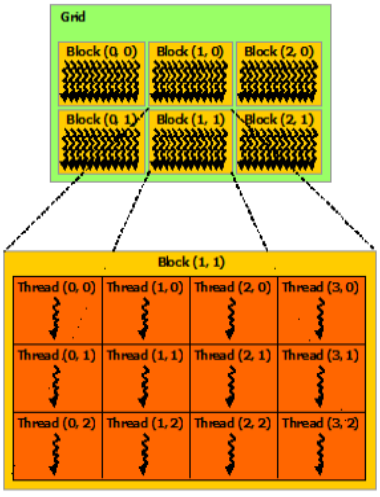
VI. EFFICIENT PARALLELISED VERSION WITH GPU

GPU BACKGROUND

The architecture of the GPUs evolved in 2001, when for the first time programmers were given the opportunity to program individual kernels in the graphics pipeline by using programmable vertex and fragment shaders. Later GPUs (as the G80 series by NVIDIA) substituted the vertex and fragment processors with a unified set of generic processors that can be used as either vertex or fragment processors depending on the programming needs. On each new generation, additional features are introduced that move the GPUs one step closer to wider use for general purpose computations. The use of a GPU instead of a CPU to perform general purpose computations is known as General Purpose computing on Graphics Processing Units (GPGPU).

CUDA ARCHITECTURE

CUDA is a parallel computing architecture developed by NVIDIA. The aim is to provide a programming framework for general purpose computations on Graphics Processing Units. CUDA programming model assumes that each CUDA thread has its own local memory and is running on one of the stream processors of the GPU multiprocessor sharing on-chip memory with the other threads running on the multiprocessor. Unlike CPU, which follows sequential pattern of execution of code, all the GPUs follow streaming data parallel programming model using SPMD architecture with single program and multiple data. Thread is the basis of parallelization in CUDA. The idea of implementing KMP in GPU is not new as many of researchers have used it in many areas of computing including mathematics. The focus now is on analyzing the efficient GPU version as compare to version of CPU discussed above.



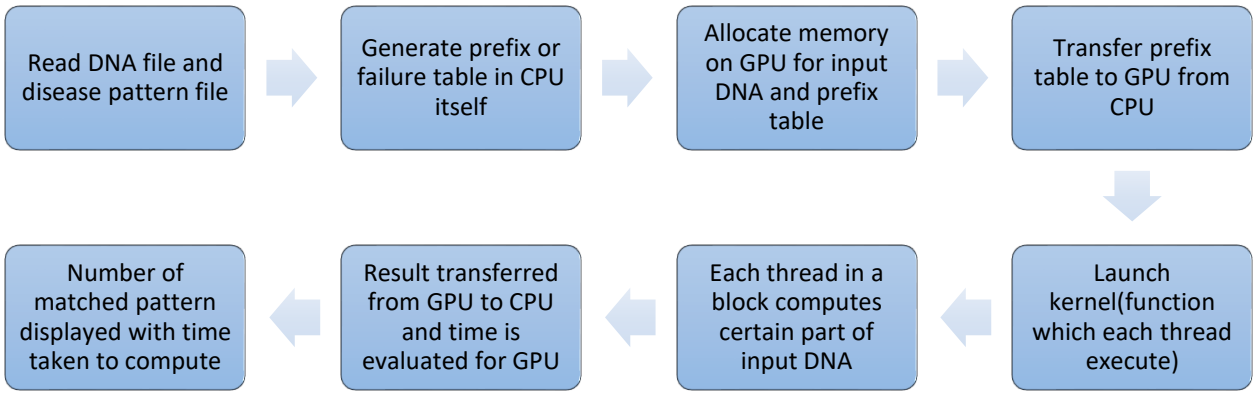
THREAD HIERARCHY

Thread is the basis of parallelization in GPU. GPUs have a parallel multi-core architecture, with each core containing thread processors that are capable of running hundreds of threads simultaneously. The threads are grouped into a number of blocks that in turn are grouped into grids. The threads in each block can be synchronized and exchange information using shared memory. All the blocks that belong to a grid can be executed in parallel while a hardware thread manager is responsible to manage threads automatically at runtime. *threadIdx* is the three component vector can be used to identify position of thread with

respect to block. On current GPUs, a thread block may contain up to 1024 threads. Each block within the grid can be identified by a one dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the builtin blockIdx variable. The dimension of the thread block is accessible within the kernel through the built in blockDim variable.

KNUTH-MORRIS-PRATT IN GPU

In implementation of Knuth-morris-pratt algorithm in GPU, the text, prefix table which is also known as preprocessed table/ failure table, the pattern array were transferred from CPU memory (host memory) to GPU memory(device memory) and kernel(function in GPU) is launched which is executed by all the threads in parallel fashion. Each thread computed specific part of text data to check the pattern matching and updated the result. Every thread accesses the preprocessed table to follow the KMP behavior of matching pattern in text. The flow of our implementation in GPU is shown by chart below



COMPLEXITY ANALYSIS

The implementation of algorithm in GPU is less complex in terms of time taken for execution if all the threads in a block efficiently distribute the input data amongst them and process the certain amount of data in parallel manner. The algorithm in efficient if size of input data is large enough to leverage the parallelism of GPU. The time complexity is less in case of GPU and is given as

$$O(m + \frac{n}{(B * T)})$$

Where m is size of pattern to be found and n is size of input text data. B in the complexity is the number of blocks and T is number of threads in each block. The above complexity defines the upper bound of run time Knuth morris pratt algorithm in GPU. It is assumed that all the threads execute at same time but it completely depends on our GPU hardware which has limit of executing certain number of threads concurrently in Stream multiprocessors. The number of resources required for execution also limits the execution time of application.

VII. EXPERIMENT ANALYSIS(GPU Version)

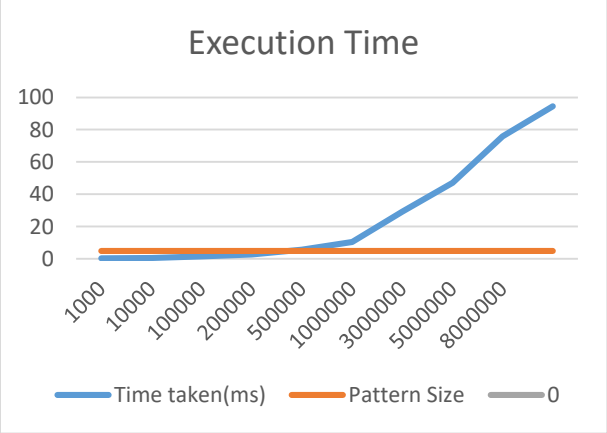
The machine on which code was executed and experiments were done is a CUDA capable device "Tesla K80" with 7.5 CUDA driver version/ Runtime version. The machine has 11520 Mbytes of global memory and 13 Multiprocessors with 192 CUDA cores per multiprocessors with total of 2492 CUDA Cores. GPU max clock rate of device is 824 MHz and memory clock rate is 2505 Mhz.

Size of L2 cache of the machine on which code is executed is 1572864 bytes. Total amount of shared memory per block is 49152 bytes whereas total amount of constant memory is 65536 bytes. Machine is capable enough to execute our solution and perform an experiments

ANALYSIS OF DETECTION OF DISEASE IN DNA

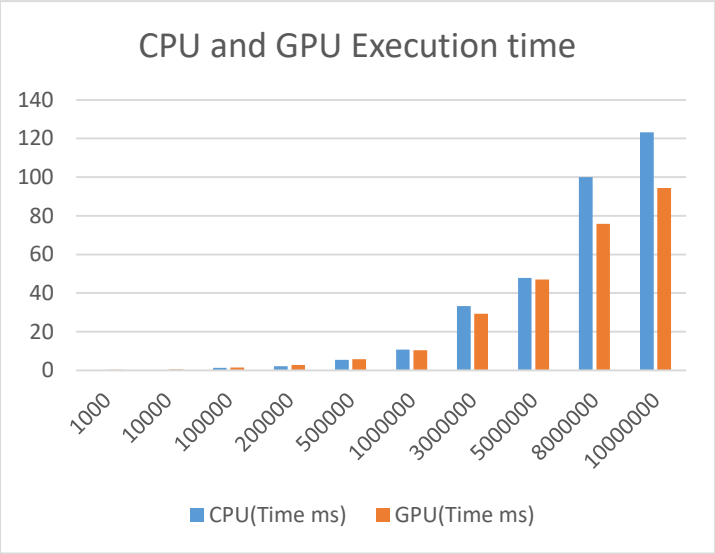
The algorithm was executed on various input size of text and fixed pattern size of 5. The execution time was observed for different input size. The graph is plotted on basis of above observations. It can be clearly seen that with the increase in size of input text in which pattern is searched, the execution time increases. The execution time with 100000 size of text is 1.5345 ms and it is 2.77069 ms for 200000 input text. Therefore, execution time increases with input size in GPU, the similar pattern was observed in CPU.

Text file size	Pattern Size	Time taken in ms(GPU)	Match found
1000	5	0.315808	3
10000	5	0.512288	8
100000	5	1.5345	87
200000	5	2.77069	186
500000	5	5.7456	496
1000000	5	10.4727	963
3000000	5	29.3293	3016
5000000	5	47.0677	4767
8000000	5	75.833	7819
10000000	5	94.4426	9831



COMPARATIVE ANALYSIS OF KNUTH MORRIS PRATT IN GPU AND CPU

To measure the improvement achieved by the GPU version of the Knuth-Morris-Pratt algorithm, the execution time was used. This metric is the total time needed for the algorithm to find all patterns searched for in the input text, and includes the transfer time between the host and the GPU (excluding the pre-processing time). The GPU and CPU version of our application was executed on multiple size data set and time was compared for both the versions for same set and same pattern. The table is shown to analyze the efficient algorithm in case of different input size.



The result clearly indicates that if data is less, for examples if data is below 1000000, then CPU performs much better as compare to GPU for performing Knuth-Morris-Pratt algorithm provided that pattern size is on an average of 5 number of characters. GPU substituted each iteration with a thread executing on specific part of data parallel. **For data less than 1000000, CPU is efficient in executing Knuth-morris-pratt algorithm for matching the pattern.(in our testing environment)**

It can be clearly seen that when there is large amount of file data to be processed and matched, then GPU is efficient environment for high execution speed and less execution time. As we can see in the table, that time taken by CPU is 123.24 ms for text size of 10000000 whereas GPU took only 94.4426 ms to process same amount of time.

Text file size	Time taken in ms(CPU)	Time taken in ms(GPU)	Efficient between CPU and GPU
1000	0.011975	0.315808	CPU
10000	0.109479	0.512288	CPU
100000	1.22138	1.5345	CPU
200000	2.16136	2.77069	CPU
500000	5.40597	5.7456	CPU
1000000	10.7581	10.4727	GPU
3000000	33.3151	29.3293	GPU
5000000	47.8544	47.0677	GPU
8000000	99.9739	75.833	GPU
10000000	123.24	94.4426	GPU

The speedup achieved is

Time taken by CPU/Time taken by GPU

=123.24/94.4426

=1.3 (approx.)

VIII. CONCLUSIONS

This paper presented analytical and comparative study of Knuth-morris-pratt pattern matching algorithm. Initial sections described the algorithm idea and its working. Examples were discussed to demonstrate the working of an algorithm. The paper also discussed various fields in our real world where this algorithm plays important role and solves lot of real world problems. The implementation of KMP algorithm was done in finding disease in DNA sequence and various set of data was executed against the implemented algorithm.

The paper discussed various fields in real world where our code is adaptable and will perform efficiently such as intrusion detection system, web spam filters, spell checkers and many more.

The implementation of KMP algorithm in the application was done using C++ programming language and efficiency was measured by executing it on various size of input in CPU. Similar version was implemented in CUDA C(similar to C++, only few difference) which employed threads executing in a parallel fashion to speedup the task.

The findings were that GPU gives efficient result in terms of time taken to execute an application after certain input size. The CPU is efficient if size of input is not too large because there is overhead in GPU which is added due to transfer of data from CPU and GPU memories in case of GPU implementation.

The cost of overhead added due to transfer of data from different memory systems in case of GPU implementation becomes negligible when data becomes too large and CPU takes long time to execute. **Therefore GPU implementation performs efficient in terms of time taken when input data is large in size otherwise CPU gives better performance.**

IX. FUTURE WORKS

Future work on optimization of KMP algorithm is suggested on CPU itself. There is scope of improvement in existing algorithm in both CPU and GPU to maximize its execution throughput. Future research can be done on string matching of patterns in more than one dimension to increase the execution speed and solving lot of real world 2 dimensional pattern matching problems with new future version of KMP on both CPU and GPU.
In GPU Implementation of KMP algorithm, there is scope of improving the efficiency of an algorithm by reducing the memory transfer time between CPU and GPU. This can be attempted and analyzed using unified memory where memory transfer is not needed explicitly.

X. REFERENCES

[1] "https://courses.acs.uwinnipeg.ca/7101/" course by Dr. Mary Adedayo

[2] "https://courses.acs.uwinnipeg.ca/4306/" course by Dr. Christopher Henry

[3] *Thomas H Corman, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein "Introduction to Algorithms-String matching", EEE Edition, 3nd Edition*

[4] *Alfred v. aho and margaret j. corasick,"efficient string matching: an aid to bibliographic search" communication of acm, vol. 18, june 1975.*

[5] *Faisal Mohammad, Jyoti Anarase, Milan Shingote, Pratik Ghanwat "Optical Character Recognition Implementation Using Pattern Matching"*

[6] *Nvidia, "Cuda c programming guide," 2012. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA C Programming Guide.pdf*

[7] *Charalampos S. Kouzinopoulos and Konstantinos G. Margaritis" String Matching on a multicore GPU using CUDA"*

[8] *KNUTH, D. E, MORRIS JR J. H AND PRATT V. R,"Fast pattern matching in strings", In the procd. Of SIAM J.Comput.Vol. 6, 1, pp. 323–350, 1977.*

[9] *Alberto Apostolico and ZviGalil," Pattern Matching Algorithms" Published in Oxford University Press, USA, 1st edition, May 29, 1997*

[10] *Vidya SaiKrishna1, Prof. Akhtar Rasool2 and Dr. Nilay Khare3 "String Matching and its Applications in Diversified Fields"*

[11] *"String Matching Algorithms and their Applicability in various Applications" Nimisha Singla, Deepak Garg*

[12] *"Parallel Approaches to the String Matching Problem on the GPU" Ashkiani, Saman Amenta, Nina Owens, John D*

[13] *"Performance Efficient DNA Sequence Detection on GPU Using Parallel Pattern Matching Approach" by Rahul Shirude1 Valmik B. Nikam2 B.B. Meshram3*