Parallelized Advanced Rabin-Karp Algorithm for String Matching

Omkar Sunil Joshi
Department of Computer Science and Engineering
Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham
Amrita University, India
omkar_21@yahoo.com

Bhargavi R. Upadhyay
Department of Computer Science and Engineering
Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham
Amrita University, India
u_bhargavi@blr.amrita.edu

Supriya M.
Department of Computer Science and Engineering
Amrita School of Engineering, Bengaluru
Amrita Vishwa Vidyapeetham
Amrita University, India
m_supriya@blr.amrita.edu

***Abstract*—String matching refers to the search of each and every occurrence of a string in another string. Nowadays, this issue presents itself in various segments in a great deal, starting from standard programs for text editing and processing, through databases and all the way to their various applications in other sciences. There are numerous different efficient algorithms to solve this problem. One of the efficient algorithms is Rabin-Karp algorithm which has complexity of O(m(n-m+1)) whereas the complexity of proposed advanced Rabin-Karp algorithm is O(n-m). However, the main focus of this research is to apply the concepts of parallelism to improve the performance of the algorithm. There are lots of parallel processing Application Programming Interfaces (APIs) available, like OpenMP, MPI, CUDA MapReduce, etc. out of these we have chosen OpenMP and CUDA to achieve parallelism. Comparison of the results of both serial and parallel implementations will give us insights into how performance and efficiency is achieved through various techniques of parallelism.**

Keywords—CUDA; OpenMP; Parallel programming; String matching

I. INTRODUCTION

In computer science, string matching algorithms are sometimes also called as string searching algorithms. The concept of string matching involves two strings; text T [0…n] and pattern P [0…m] where pattern is searched in the text for the exact match. There are two major techniques for string matching, exact string matching and approximate string matching [1]. In this research paper we'll focus on exact string matching because it gives more accurate results as compared to approximate string matching. Therefore exact string matching has numerous applications like, bioinformatics, intrusion detection, genetic computing, plagiarism detection, digital forensics, text mining, image and video retrieval, retrieval of music patterns and many more [2]. Performance becomes critical factor when huge amount of data are involved in such applications [3]. Parallel programming can be a solution to this problem.

Parallel programming is a type of programming where computation or the execution of processes happens simultaneously. New generation Central Processing Units (CPUs) and Graphics Processing Units (GPUs) provide multicore architecture which enables parallel programming [4]. In parallel programming, large computing tasks are divided into smaller tasks which then can be executed in parallel environment. There are many parallel programming APIs available like, MPI, OpenMP, MapReduce, CUDA, etc.

OpenMP which stands for Open Multi-Processing is an API which supports C, C++ and FORTRAN. It works on multiplatform shared memory multiprocessing architecture. OpenMP is designed with a set of preprocessor directives, environment variables and library functions that controls the runtime behavior of the program. OpenMP is controlled and run by non-profit technology consortium OpenMP Architecture Review Board (OpenMP ARB) mutually supported by top-tier software and hardware companies, including IBM, Intel, NVidia, AMD, HP, Red Hat, Oracle and more [5].

Computer Unified Design Architecture (CUDA) is an API and parallel computing platform model designed by NVidia [6]. It allows GPUs to be used for general purpose processing, commonly referred as GPGPU (General Purpose computing on Graphics Processing Unit) [7]. CUDA is designed to work as an embedded code in C, C++ and FORTRAN. This compatibility makes it easier for programmers to use GPU for parallel programming rather than using prior APIs [8].

Paper is categorized as follows. Section II narrates related work, section III describes implementation methodology for Rabin Karp and Advanced Rabin Karp algorithm, Section IV gives details about experimental environment, section V describes experimental results and section VI provides the conclusion of this research.

II. RELATED WORK

A brief description and working methodology of all popular string matching algorithms is provided in this section.

978-1-5386-4008-1/17/$31.00 ©2017 IEEE

A. Naïve (Brute force) string matching algorithm

The naïve string matching algorithm works on one by one character matching principle. It takes the pattern string and matches it with text in which the search is happening; one character at a time, shifting both pattern and text from left to write. It is the simplest but inefficient way of string matching [1], [9].

B. Boyer Moore algorithm

This algorithm is developed in 1977 by Robert S. Boyer and J. Strother Moore. In this algorithm equivalence of pattern in text is determined from right to left whereas, the pattern moves from left to write. If the text character being compared with right most patter character doesn't appear in the pattern at all then the text is shifted by m (length of pattern) positions from right to left [10], [11].

C. Knuth Morris Pratt algorithm

In 1974 D. Knuth, J. Morris and V. Pratt designed a string matching algorithm called Knuth Morris Pratt (KMP) algorithm. This is the only known linear time algorithm and hence the most popular one. This algorithm works from left to right and uses notion border of the string in case of ill-match or exact match with the pattern. It makes sure that the string search won't take more than n (length of text) comparisons for string matching [12], [13].

D. Needleman Wunsch Algorithm

This algorithm is created in 1970 by Saul B. Needleman and Christian D. Wunsch. It was developed for biological sequence comparisons based on dynamic programming approach. Needleman Wunsch algorithm performs a comprehensive alignment on protein or nucleotide chains [14].

E. Smith Waterman

Temple F. Smith and Michael S. Waterman created this string algorithm in 1981. It guarantees the highest confined positioning which is a variation of the Needleman Wunsch algorithm. It works on local sequence alignment which detects similar regions between strings of nucleoid acid sequences or protein sequences [15].

F. Rabin Karp algorithm

Michael O. Rabin and Richard M. Karp developed this string matching algorithm in 1987. It works based on hashing mechanism; it calculates the hash value of the pattern and equates it with hash value of substring of text which is of same length as pattern length. If pattern hash value matches with text hash value then there will be a character by character string matching just like naïve string matching algorithm [16].

The worst case time complexity of the various string matching algorithms is shown in Table I [2]. Our work focuses on the improvisation of the existing Rabin Karp algorithm which is explained in the next section. In order to compare, the existing algorithm is described first which is then followed by the new modified approach which has running time complexity of O(n-m).

TABLE I. COMPARISON OF COMPLEXITIES

Algorithm	Complexity
Naïve string matching	O(mn)
Boyer Moore	O(mn)
Knuth Morris Pratt	O(m+n)
Needleman Wunsch	O(mn)
Smith Waterman	O(mn)
Rabin Karp	O(mn)
Boyer Moore Horspool	O(mn)
Quick search	O(mn)

III. IMPLEMENTATION METHODOLOGY

A. Rabin Karp algorithm

Fig. 1 explains the working of Rabin Karp algorithm where hash value is calculated using mod 13 and search pattern is 31415, first row contains the text in which string searching is happening whereas second row represents the hash values for respective substrings.

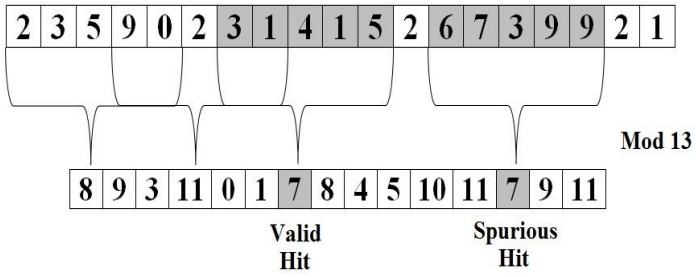

Figure 1. Rabin Karp algorithm.

Pseudo code for Rabin-Karp String Matching

```
n = length(Text)
m = length(Pattern)
mod = Mod(Pattern)
for i from 0 to (n – m + 1) {
      if Mod(Text[i]) == mod {
            for j from 0 to m {
                  if Text[i] == Pattern[j]
                        print(Valid)
                  else
                        print(Spurious)
            }
      }
}
```

Running time complexity of this algorithm is O(m(n-m+1)) or O(mn) which is an exponential value. It works like brute force string matching algorithm when hash value of pattern matches with hash values of substring of text, which is a major drawback of this approach. Another disadvantage of using

Rabin Karp algorithm is it gives spurious hits. When the hash value comparison gives a hit but string matching result is not matching, it is called as spurious hit or false hit.

B. Advanced Rabin Karp algorithm

To overcome the drawbacks of Rabin Karp algorithm, we have developed this new algorithm called advanced Rabin Karp algorithm. This algorithm works on mathematical division rule (1).

$$\text{Dividend} = \text{Divisor} * \text{Quotient} + \text{Remainder} \quad (1)$$

In this proposed algorithm, along with the hash value we are computing the quotient of the pattern which is equated with the quotient value of the text. This computation of quotient of text and comparison takes place only if the hash value of pattern gives a positive result with the hash value of text. Such additional comparison avoids the character by character string matching like naïve string matching algorithm which is a drawback of the Rabin Karp algorithm. The proposed advanced Rabin Karp algorithm also guarantees that, there will be only valid hits, meaning no more spurious hits which is represented in Fig. 2. Time complexity of this proposed advanced Rabin Karp algorithm is O(n-m) which is a linear time as compared to quadratic time of Rabin Karp algorithm.

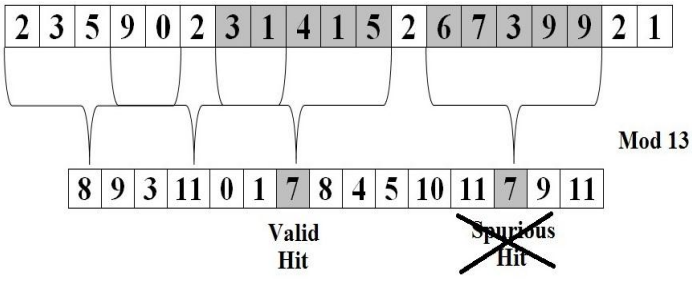

Figure 2. Advanced Rabin Karp algorithm.

Pseudo code for advanced algorithm

```
n = length(Text)
m = length(Pattern)
mod = Mod(Pattern)
quotient = Div(Pattern)
for i from 0 to (n – m + 1) {
        if (Mod(Text[i]) == mod) && (Div(Text[i]) == quotient)
                print(Hit)
}
```

IV. EXPERIMENTAL ENVIRONMENT

In order to test the performance and efficiency of the above described algorithms, an experimental environment has been created. Here, all the experiments were executed on Intel core i7 – 4777 @ 3.4GHz, 8 core processor with 16 GB RAM and Red Hat 7.3 edition open client operating system configured machine. For sequential as well as OpenMP program execution GCC 4.8.5 was used and CUDA programs were executed on NVidia GeForce GTX 1060, 6 GB GDDR5 graphics with 1280 CUDA cores graphics card with NVCC 8. Throughout these experiments only usual ideal background processes ran. To decrease the random variation all the time results are collected by taking average of 100 test runs per program per data sample. Pseudo codes propounded in the above section were implemented using ANSI C programming language. Program's hash value and/or quotient value comparison module was executed in parallel manner. For the purpose of testing of these algorithms we used part of DNA chain from a fish Zebrasome flavences family of sea fish. This is the same dataset which is used in [17].

V. EXPERIMENTAL RESULTS

For the better understanding of performance of all these algorithms, time results representing graphs are divided into two categories as small data and large data. Fig. 3 to 8 shows the comparison of the existing and the modified Rabin Karp algorithms. In all the graphs given below, x-axis represents the size of text whereas y-axis shows time in seconds.

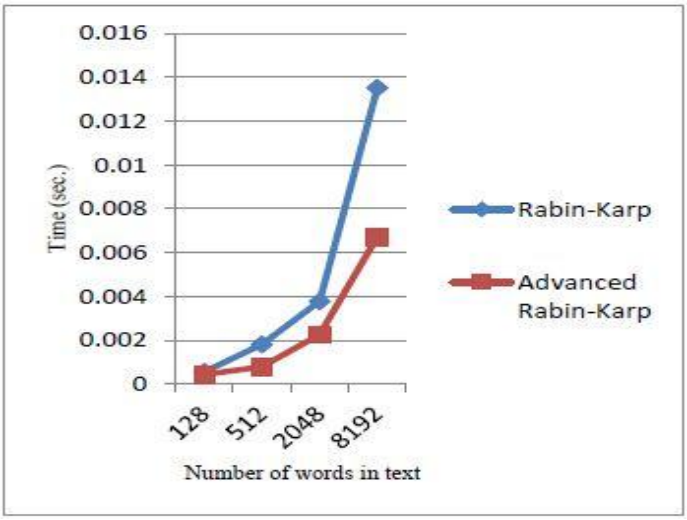

Figure 3. Sequential Rabin Karp on small data.

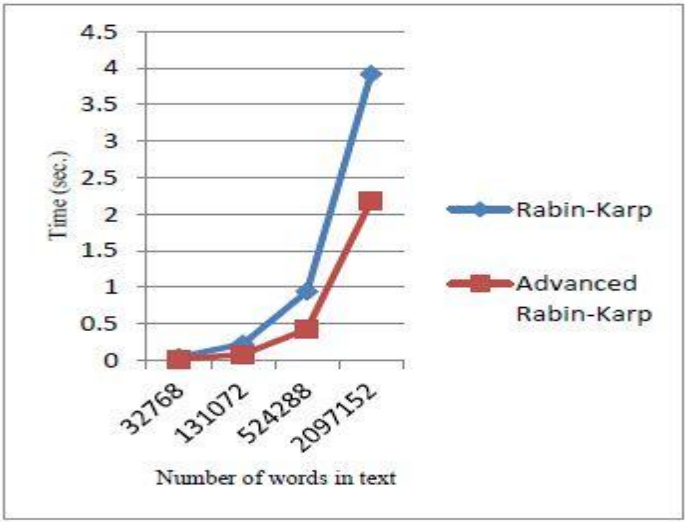

Figure 4. Sequential Rabin Karp on large data.

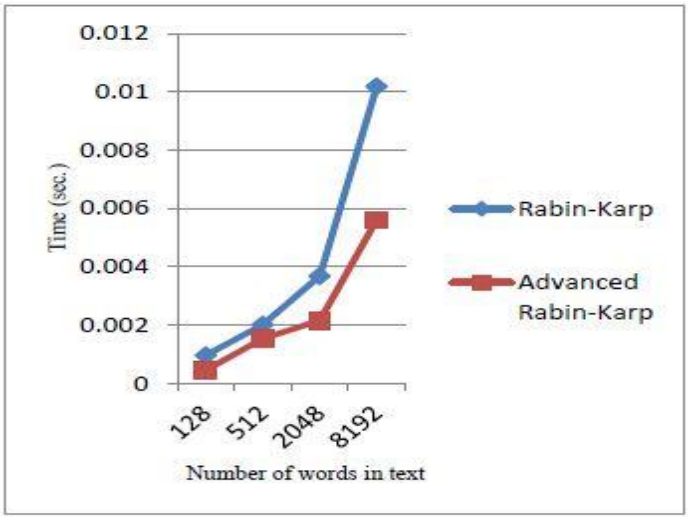

Figure 5. OpenMP Rabin Karp on small data.

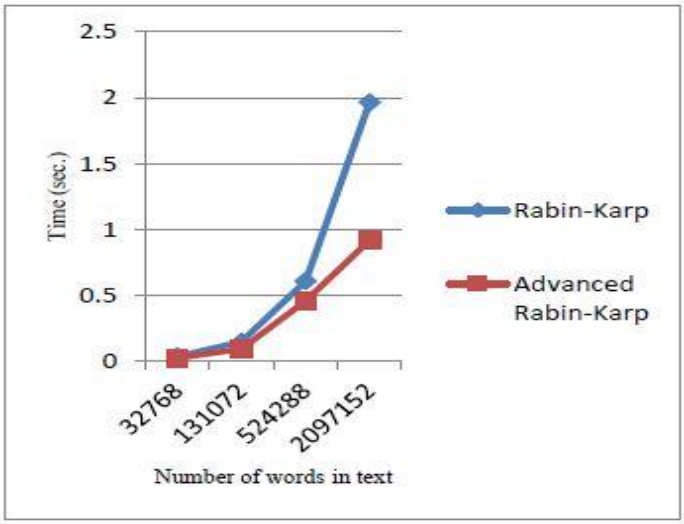

Figure 6. OpenMP Rabin Karp on large data.

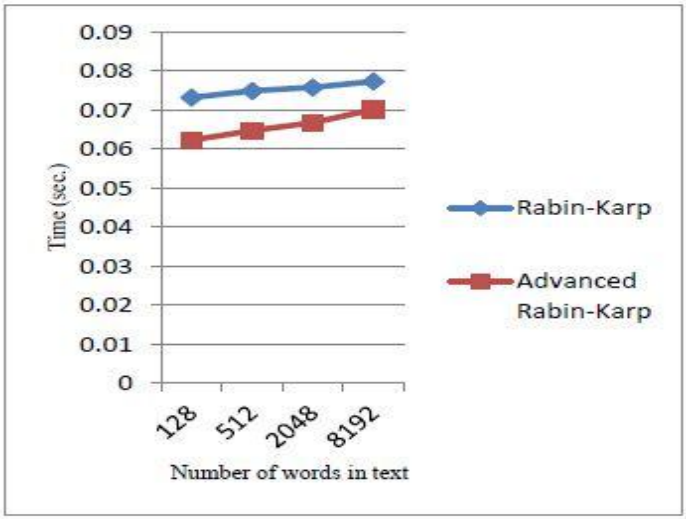

Figure 7. CUDA Rabin Karp on small data.

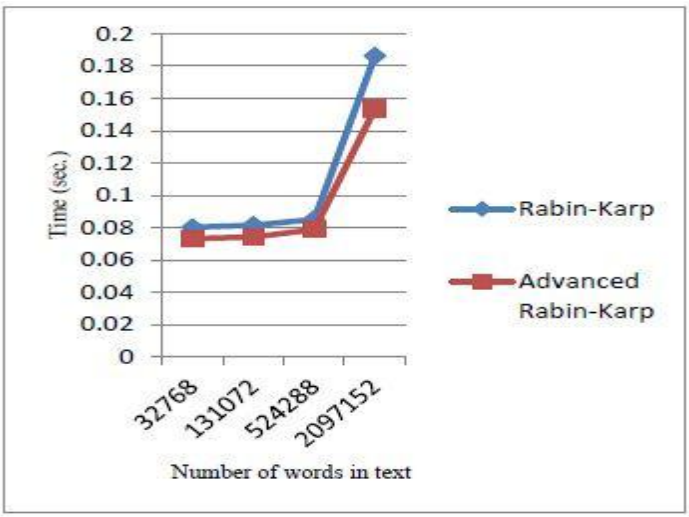

Figure 8. CUDA Rabin Karp on large data.

VI. CONCLUSION

We have explored all the popular string matching algorithms and did in depth research on Rabin Karp. We used this analysis as a base for the advanced Rabin Karp algorithm. Pseudo code shows the better time complexity for the proposed algorithm. Hence we can conclude that the advanced Rabin Karp algorithm's time complexity is the best so far, even better than KMP algorithm. Experimental results show better results for proposed algorithm than the existing algorithm irrespective of the implementation environment and data size.

Experimental results also give us insight into performance of different implementation methodologies. Fig. 9 to 12 concludes that sequential execution is a faster variant for data size less than 100 KB whereas OpenMP gives best results when the data size is in the range 100 KB – 1 MB. But when the data size increases beyond 1 MB which is the usual case in most of the real-time applications then CUDA programming is the most efficient solution.

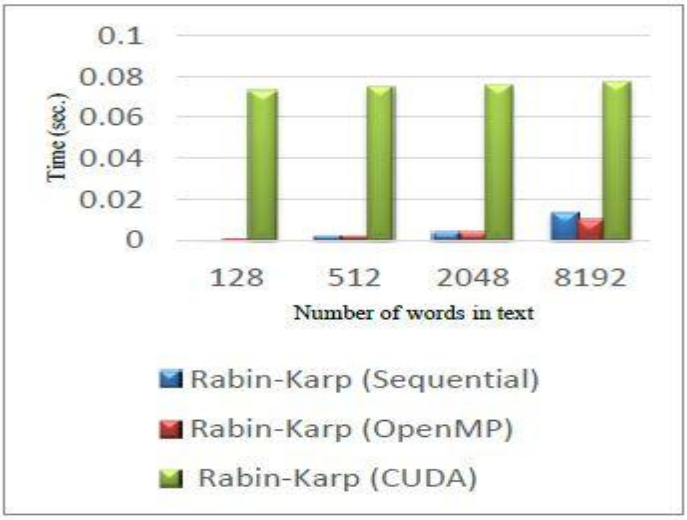

Figure 9. Rabin Karp algorithm on small data.

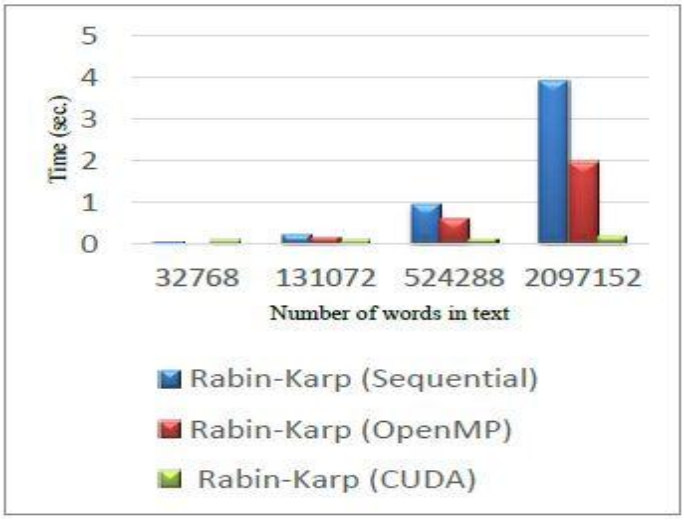

Figure 10. Rabin Karp algorithm on large data.

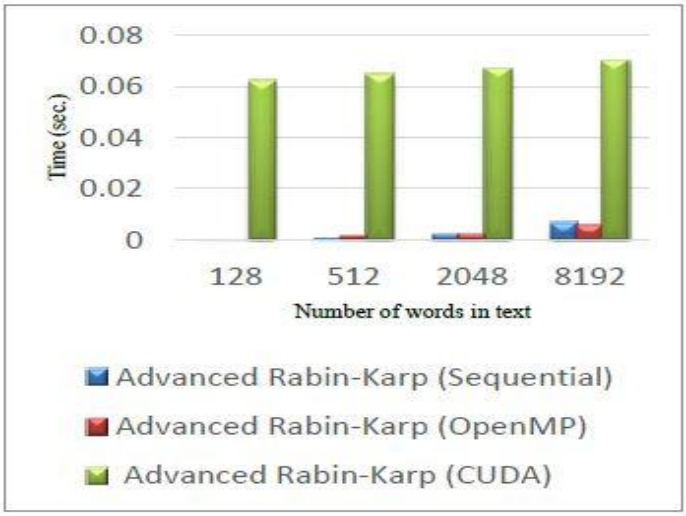

Figure 11. Advanced Rabin Karp small data.

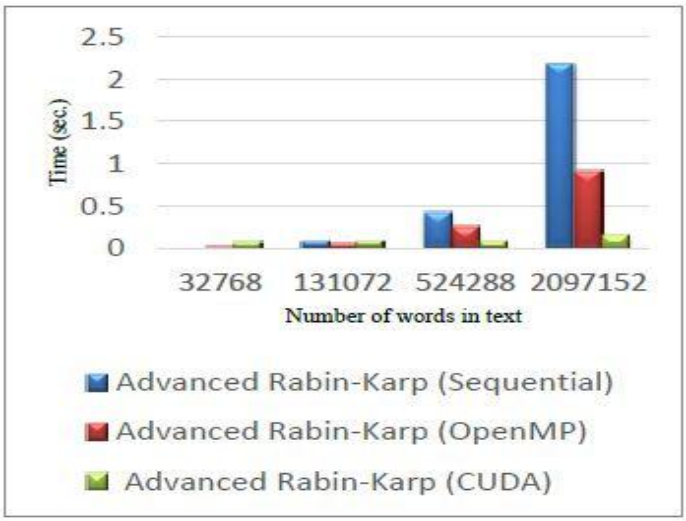

Figure 12. Advanced Rabin Karp on large data

REFERENCES

[1] Minal Suthar, Amit Patel, Shivali Shah "A Survey Paper on String Matching." International Journal for Scientific Research & Development ISSN [online]: 2321-0613 Vol. 3, Issue 05, 2015

[2] Nimisha Singla, Deepak Garg "String Matching Algorithms and their Applicability in various Applications." International Journal of Soft Computing and Engineering ISSN: 2231-2307, Volume-I, Issue-6, January 2012

[3] Abarna K., Rajamani M., Shriram K Vasudevan "Big data analytics: A detailed gaze and a technical review." International Journal of Applied Engineering Research, Research India Publications, Volume 9, Number 11, p.1735-1751 (2014)

[4] A. Baskar, Shriram K. Vasudevan, P. Prakash "Advanced investigation and comparative study of graphics processing unit-queries countered." Research Journal of Applied Sciences, Engineering and Technology, Volume 8, Issue 15, Number 2, p.1766-1771 (2014)

[5] "The OpenMP API specification for parallel programming." [online]. Available: http://www.openmp.org

[6] "CUDA parallel computing platform." [online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[7] Charalampos S. Kouzinopoulos, Konstantinos G. Margaritis "String Matching on a multicore GPU using CUDA." 13th Panhellenic Conference on Informatics. 10-12 Sept. 2009

[8] "NVidia Accelerated Computing." [online]. Available: http://www.developer.nvidia.com/cuda-zone

[9] Koloud Al-Khamaiseh, ShadiALShagarin, "A Survey of String Matching Algorithms." International Journal of Engineering Research and Applications, ISSN: 2248-9622, Vol. 4, Issue 7 (Version 2), July 2014, pp.144-156.

[10] Boyer Robert S., Moore J. Strother. "A Fast String Searching Algorithm." Comm. ACM. New York, NY, USA: Association for Computing Machinery. ISSN 0001-0782, October 1977

[11] Zhengda Xiong, "A Composite Boyer-Moore Algorithm for the String Matching Problem." 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies. 492 – 496. 8-11 Dec. 2010.

[12] D. E. Knuth, J. H. Morris and V. R. Pratt "Fast pattern matching strings." SIAM Journal on Computing, 6(1): 323-350, 1977.

[13] Dana Shapira, Ajay Daptardar "Adapting the Knuth–Morris–Pratt algorithm for pattern matching in Huffman encoded texts." Data Compression Conference 2004. 23-25 March 2004.

[14] Saul B. Needleman, Christian D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins." Journal of Molecular Biology, Volume 48, Issue 3, 28 March 1970, Pages 443-453

[15] Smith, Temple F., Waterman, Michael S. "Identification of Common Molecular Subsequences." Journal of Molecular Biology, 147: 195–197. 1981

[16] Karp Richard M., Rabin Michael O. "Efficient randomized pattern-matching algorithms". IBM Journal of Research and Development, 31 (2): 249–260. March 1987

[17] Predrag Brodanac, Leo Budin, Domagoj Jakobovic "Parallelized Rabin-Karp Method for Exact String Matching." Proceedings of the ITI 2011 33rd Int. Conf. on Information Technology Interfaces (2011): 585-90. Web.2 Oct.2014