Implementation of exact-pattern matching algorithms using OpenCL and comparison with basic version

Andrii Rozumnyi
Faculty of Science and Technology
Computer Science
University of Tartu
andriiro@ut.ee

***Abstract*—In big text-processing tasks, the exact pattern-matching problem still remains time consuming. As algorithms asymptotically faster than existing ones cannot be developed, there is a need to use another approach to promote efficiency. Thus, parallel computing is able to significantly speed up the process of the exact pattern-matching problem solving. That is why the current work is focused on parallelized version of the most famous algorithms using OpenCL framework. The code base for the project and suggested approach were reviewed and approved by the International Workshop on OpenCL 2016.**

***Keywords*—exact pattern-matching, Knuth-Morris-Pratt algorithm, Boyer-Moore-Horspool algorithm, OpenCL, PyOpenCL**

I. Introduction

First of all, the exact pattern-matching problem (which implies recognition of all the occurrences of a pattern within the text given), nowadays has many different applications such as parsers, word processors, spam filters, DNA applications in Computational Biology, etc. In some cases, when the string length is relatively small, the problem can be efficiently solved by using classical algorithms with linear time complexity. However, in some areas such as Bioinformatics, this task is still a problem due to the huge length of the genomic data (for instance, human genome appear to be around three billion characters) and many patterns (with millions of patterns possible) processing is time consuming.

Nowadays the volume of work cannot be processed only by the mere power of computational units in reasonable time. Even though linear time complexity algorithms were developed in the previous century, it needs to be improved. As from the asymptotic point of view, it cannot be faster than linear time, the computer science society moved to heterogeneous computing, where GPUs, CPUs and other processing units act together as co-processors.

OpenCL is a powerful standard for task-parallel and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs [1]. It allows the use of wide range of devices in order to perform parallel computing. Therefore, in comparison to existing alternative programming toolkits, this makes it much easier for a developer to begin with a correctly functioning OpenCL program tuned for one architecture and produce a correctly functioning program optimized for another architecture [2].

In this work, we made an attempt to implement naive approach, Knuth-Morris-Pratt and Boyer-Moore-Horspool algorithms and compare them with the same algorithms, but adopted for concurrent processing using the power of OpenCL.

II. Related Works

The topics about string matching algorithms are quite popular. The already conducted researches present wide ranges of focusing. Aragon et al. devoted their special attention to energy consumption [3], while Nhat-Phuong focused mostly on memory efficiency [4]. There is bunch of works investigating the performance of exact-pattern matching algorithms on particular hardware architecture [5-8]. At the same time, some studies analyze algorithms in terms of particular application such as Network Security [9, 10].

Pandiselvam et al. in [11] analyzed set of algorithms for pattern-matching problem using Biological sequencing data.

III. Theory

A. Naive Approach

As a baseline, the naive approach had been chosen to tackle the exact pattern-matching problem. The idea of the algorithm implies the following: For each possible starting position of pattern inside a given string letter by letter is compared. If a mismatch occurs then we move the pattern along the string in one position and repeat the same procedure. If all letters of the pattern had been matched, then we add a starting position of a pattern relatively to the text to the result, move in one position ahead and the same procedure is repeated over again.

The disadvantage of the algorithm is that it always shifts the pattern only in one position. The naive approach does not use the information on checked characters. Thus, the algorithms complexity is $O(nm)$, where n stands for string length and m for pattern length. At the same time this process proves to be very easy to implement and debug. Furthermore, no additional pattern or string processing are required.

B. Knuth-Morris-Pratt (KMP) algorithm

In contrast to naive approach, KMP makes preprocessing of a pattern. There are a pattern P of a length m and a text T of length n. Consider comparing strings on position i, where pattern $P[0, m-1]$ is compared to a part of text $T[i, i+m-1]$. Let us assume that a first mismatch occurred between $T[i+j]$ and $P[j]$, where $1 < j < m$. Then $T[i, i+j-1] = P[0, j-1] = Pr$ and $a = T[i+j] \neq P[j] = b$.

After shifting we might assume that prefix of a pattern P coincide with some suffix of Pr. The length of the longest prefix, which is at the same time a suffix, is the value of prefix function from string P and index j.

It leads to the next algorithm: Let us have $\pi[j]$ the value of the prefix function from string $P[0, m-1]$ for index j. Then after shifting we can continue comparing from $T[i+j]$ and $P[\pi[j]]$ without any missing of pattern occurrences. It can be shown that the table π can be calculated in $\theta(m)$ time. As the text will be analyzed only once, the total complexity will be $\theta(m+n)$, where n - length of the T. The KMP algorithm performs at most $2n-1$ text character comparisons during the searching phase [12].

C. Boyer-Moore-Horspool (BMH) algorithm

The algorithm is a modification of Boyer-Moore algorithm, which was invented earlier. It based on the next principals. The algorithm start compare characters from the pattern end to the beginning. If the mismatch was found then the pattern shifts to the right on amount of positions based on bad-character shift rule. In case of BMH we take the character from the text (stop-symbol) which is aligned with the last character in pattern. Then we shift pattern in such a way that it aligns with the right-most position of a stop-symbol inside a pattern. It is implemented using the shifting table: For each symbol of the alphabet, we store the maximal possible shift, which does not skip any occurrences of stop-symbols. Thus, having 1-based indexing $shift(c) = |pattern| - lastpos(c, pattern[1 .. |pattern|-1])$, where $lastpos$ the last position of a symbol in a pattern, $pattern[a .. b]$ is operation of taking a substring.

For symbols, which do not occur in a pattern, the shift amount is set to the pattern length. The last symbol of a pattern does not take under consideration; otherwise, the algorithm might be cycled.

IV. Implementation

The research was conducted using PyOpenCL, which is a wrapper for OpenCL framework. It has completely the same functionality as OpenCL.

As it was mentioned in the beginning, all measurements were done using the biological data. The first 10,000,000 characters from the 1-st human chromosome serves as a text. Half of the patterns are taken from the DNA string (in order to be sure that it occurs in a text), while another half is randomly generated by means of DNA alphabet. There are two groups of patterns based on the length: $25, 50, ..., 200$ and $250, 500, ..., 2000$. There is 100 patterns of each length.

All algorithms were implemented in both versions: Basic one-threaded and adopted for concurrent processing using PyOpenCL. The exact pattern-matching problem can be easily parallelized as the algorithm might compare pattern with different parts of the text simultaneously using the concurrent programming approach. We divided the text into 1000 splits and analyzed each of them separately.

All measurements have been done under Windows 8.1 operating systems using the following hardware: Intel Core i5, CPU 2.2 GHz, 8 GB RAM. The chosen Integrated development environment is PyCharm from JetBrains. As laptop has CPU and GPU, measurements were done for both devices (Table I).

TABLE I. Hardware for Measurements

	CPU	GPU
Name	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz	Intel(R) HD Graphics 5500
Number of units	4	24

There have been implemented the basic one-threaded version for Rabin-Karp and Boyer-Moore algorithms as well. As work on concurrent version is still in progress, we did not include information on them in measurements section.

V. Measurements

All further information about measurements is an average time in seconds for finding all occurrences of one pattern inside a given string. The benchmarks for basic version are in Table II and Table III.

Even though we do not see the differences in order of magnitude, KMP and BMH are faster than naive approach. If we measure on data of practical size in Computational Biology then the differences will be much more significant.

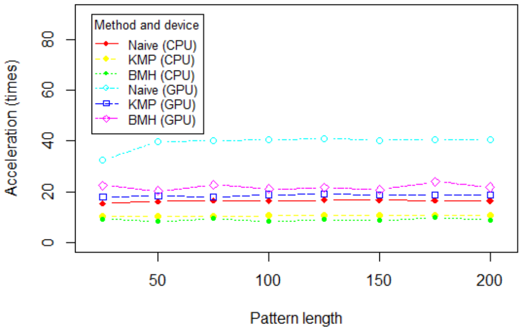

Fig. 1. Acceleration for the 1-st group of patterns on both devices

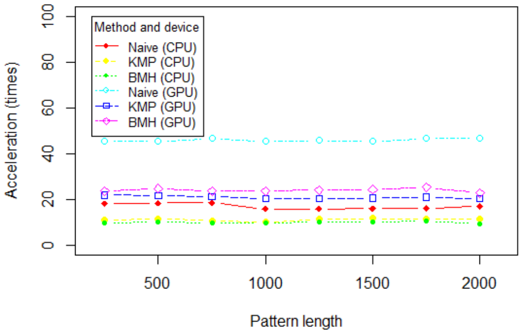

Fig. 2. Acceleration for the 2-nd group of patterns on both devices

TABLE II. MEASUREMENTS FOR BASIC IMPLEMENTATION FOR THE 1-ST GROUP OF PATTERNS

Length	25	50	75	100	125	150	175	200
Naive, sec	5.02182	5.038513	5.031702	5.019916	5.06889	5.072226	5.013521	5.006208
KMP, sec	3.195677	3.209536	3.189474	3.30718	3.355372	3.387918	3.386511	3.399092
BMH, sec	2.861145	2.593235	2.911317	2.699552	2.793429	2.699568	3.053905	2.771127

TABLE III. MEASUREMENTS FOR BASIC IMPLEMENTATION FOR THE 2-ND GROUP OF PATTERNS

Length	250	500	750	1000	1250	1500	1750	2000
Naive, sec	5.574916	5.649528	5.698257	5.659674	5.658158	5.663552	5.713852	5.855411
KMP, sec	3.851339	3.755037	3.715272	3.680036	3.668797	3.683142	3.693518	3.698741
BMH, sec	3.065156	3.137926	3.002268	3.004524	3.158191	3.111886	3.242042	2.932839

From the figure 1 and figure 2, we see the next pattern: On GPU we received bigger value in comparison to CPU. Accelerations are similar for different sizes, as we did not create the situation (which is rare case in practical) close to the worst.

In addition, we clearly see that naive approach accelerated even more (up to two times) in comparison to KMP and BMH. Quite foreseeable, as in last two algorithms we have additional tables from preprocessing phase, which we need to pass as a parameter to a function on each kernel. Thus, it is stored in global memory and it takes more time to get access to it.

More detailed benchmarks (for different number of text splits) may be well found in appendix A and GitHub repository https://github.com/JaakTree/pattern_matching/tree/test.

VI. CONCLUSION

This paper elucidates the research conducted on the exact-pattern matching issue. The most widely used algorithms had been hence considered and implemented as well as achieved acceleration had been studied on CPU and GPU devices for specific hardware.

There is an urgent need to continue the current research in both directions: Implement and adopt other algorithms for OpenCL platform and conduct measurements on more powerful hardware. Beside that, consumed energy consumption of running the searching procedure can be eventually measured.

In some areas such as Bioinformatics, there is a need to tackle inexact pattern-matching problem. It is more complicated problem; however, the current work might be used as a baseline for solving it.

ACKNOWLEDGMENT

The study of the author of this article is supported by the Estonian Foreign Ministry's Development Cooperation and Humanitarian Aid funds.

REFERENCES

[1] A. Munshi. (2008). OpenCL Specification Version 1.0 [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[2] J. Stone, D. Gohara, G. Shi, ”A Parallel Programming Standard for Heterogeneous Computing Systems”, Computer Science & Engineering, vol. 12, pp. 66-73, 2010. doi: [10.1109/MCSE.2010.69]

[3] E. Aragon, J. Jimenez, A. Maghazeh et al., ”Pattern matching in OpenCL: GPU vs CPU energy consumption on two mobile chipsets”, IWOCL, Bristol, UK, 2014. doi: [10.1145/2664666.2664671]

[4] N.-P. Tran, ”Memory Efficient Parallelization for Aho-Corasick Algorithm on a GPU”, High Performance Computing and Communication, Liverpool, UK, 2012. doi: [10.1109/HPCC.2012.65]

[5] N.-P. Tran, M. Lee, S. Hong et al., ”Multi-stream Parallel String Matching on Kepler Architecture”, MUSIC, pp. 307-313, 2013.

[6] A. Rasool, N. Khare, ”Parallelization of KMP String Matching Algorithm on Different SIMD architectures: Multi-Core and GPGPUs”, IJCA, vol. 49, pp. 26-28, Jul, 2012.

[7] M. Assael (2013, March). String Matching on hybrid parallel architectures, an approach using MPI and NVIDIA CUDA [Online]. Available: http://www.yannisassael.com/publications/assael_uom_dissertation.pdf

[8] A. Rasool, N. Khare, ”Generalized Parallelization of String Matching Algorithms on SIMD Architecture”, IJCSIS, vol. 11, no. 5, pp. 6-16, May, 2013.

[9] M. Jaiswal, ”Accelerating Enhanced Boyer-Moore String Matching Algorithm on Multicore GPU for Network Security. International Journal od Computer Applications”, IJCA, vol. 97, pp. 30-35, Jul, 2015.

[10] J. Sharma, M. Singh, ”CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU”, IJCNIS, vol. 10, pp. 70-77, Sept, 2015. doi: [10.5815/ijcnis.2015.10.08]

[11] P. Pandiselvam, T. Marimuthu, R. Lawrance. ”A Comparative Study on String Matching Algotthms of Biological Sequences” in International Conference on Intelligent Computing, Pattaya, 2014, pp. 37-43.

[12] C. Christian, L. Thierry, ”Knuth-Morris-Pratt algorithm,” in Handbook of exact string-matching alorithms Christian Charras, 1-st ed. London, UK: College Publications, 2004, ch. 7, pp. 47-50.

APPENDIX A
DETAILED MEASUREMENTS

TABLE IV. MEASUREMENTS FOR THE 1-ST GROUP OF PATTERNS ON CPU

Length	25	50	75	100	125	150	175	200
Naive, sec	0.323474007	0.309220052	0.303894157	0.305363889	0.303506517	0.303291125	0.303220315	0.302899404
KMP, sec	0.311757803	0.311536798	0.310448842	0.311428857	0.313387036	0.310848417	0.31105927	0.31536191
BMH, sec	0.307938929	0.307091517	0.308409023	0.318847704	0.307207699	0.308077908	0.308912005	0.308469448

TABLE V. MEASUREMENTS FOR THE 2-ND GROUP OF PATTERNS ON CPU

Length	250	500	750	1000	1250	1500	1750	2000
Naive, sec	0.305012364	0.304769855	0.305656362	0.355974698	0.357422271	0.35072722	0.353427548	0.343660817
KMP, sec	0.341282878	0.328195052	0.341405015	0.36579186	0.322969494	0.313570914	0.325357518	0.318204336
BMH, sec	0.309918842	0.308242202	0.307004466	0.30654108	0.306764998	0.307552299	0.306757245	0.307265573

TABLE VI. MEASUREMENTS FOR THE 1-ST GROUP OF PATTERNS ON GPU

Length	25	50	75	100	125	150	175	200
Naive, sec	0.154338431	0.126586719	0.125607944	0.123725219	0.12330503	0.125738597	0.123485093	0.123104715
KMP, sec	0.177230687	0.173500938	0.177494836	0.17481236	0.175059247	0.180018783	0.179453397	0.179965053
BMH, sec	0.12739718	0.126916838	0.12774735	0.127901235	0.1284091	0.129958887	0.126963749	0.126885729

TABLE VII. MEASUREMENTS FOR THE 2-ND GROUP OF PATTERNS ON GPU

Length	250	500	750	1000	1250	1500	1750	2000
Naive, sec	0.122794113	0.124260712	0.12234282	0.124456096	0.123756008	0.124711165	0.122507715	0.124859056
KMP, sec	0.17527607	0.172276807	0.174231391	0.18058732	0.179690127	0.178845682	0.174552774	0.18151938
BMH, sec	0.128773861	0.126733084	0.125951414	0.126282744	0.130474501	0.12736515	0.127961454	0.128504658