String Matching Algorithms

Project presentation

GPU Programming course

Referents:
Prof. Luca Sterpone, Prof. Bartolomeo Montrucchio,
Dr. Josie Esteban Rodriguez Condia, Dr. Corrado De Sio

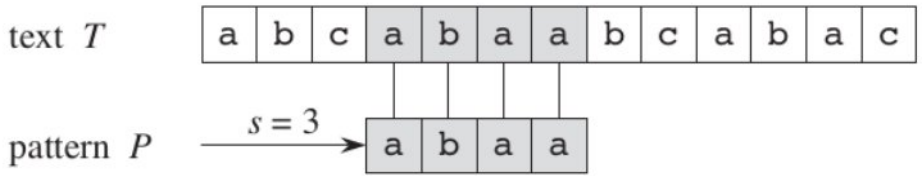
Presented by Isoldi Matteo, s318980

Outline

- Introduction
- Selected algorithms
- Code implementation
- Observed results

The string matching problem

Term used to refer to the problem of **finding the locations where one or more strings, called pattern, are positioned within a larger text**. It represents an essential problem in the computer science field, where frequent requests to scan texts, databases, websites and many other sources of information to find specific information are performed continuously, and bioinformatic fields, where molecular problem can be formulated as string matching problems.



Source: https://www.dcc.fc.up.pt/~pribeiro/aulas/alg1819/slides/8_stringmatching_11112018.pdf

The string matching problem

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)

Several algorithms have been designed over the years to improve the performance of the scan operation on the text. However, due to the importance of the problem, it is still topic of research to further improve existing solutions and to design better ones.

Source: https://d3i71xaburhd42.cloudfront.net/53b447ab70683336aebfafd1889bdb20bc61756a/2-TableI-1.png

Selected algorithms

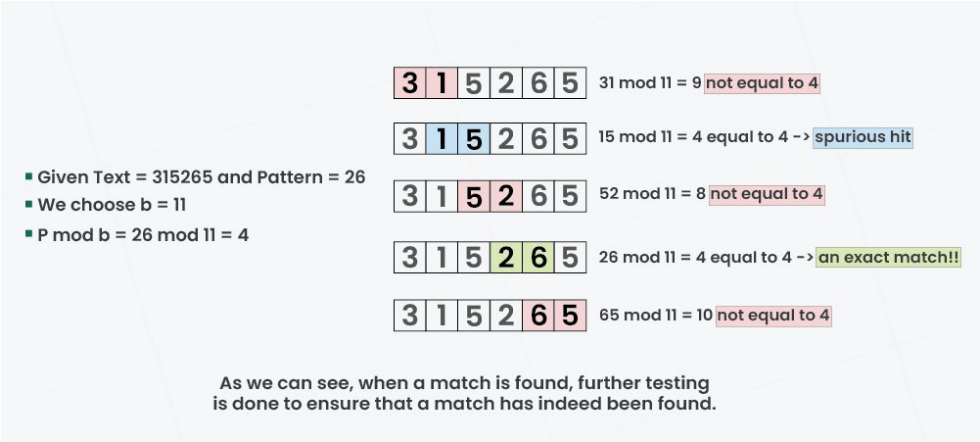
Three state-of-the-art algorithms were chosen based on two common properties: **the resulting solution has to be exact** and they have to **work on one pattern at a time** (single-pattern searches). The selected algorithms are:

- **Rabin-Karp** (RK);
- **Knuth-Morris-Pratt** (KMP);
- **Boyer-Moore** (BM).

Rabin-Karp

Based on speeding up the comparison process through a **rolling hash function**. Thus, every possible substring is mapped to an integer that is compared to the pattern's hash.

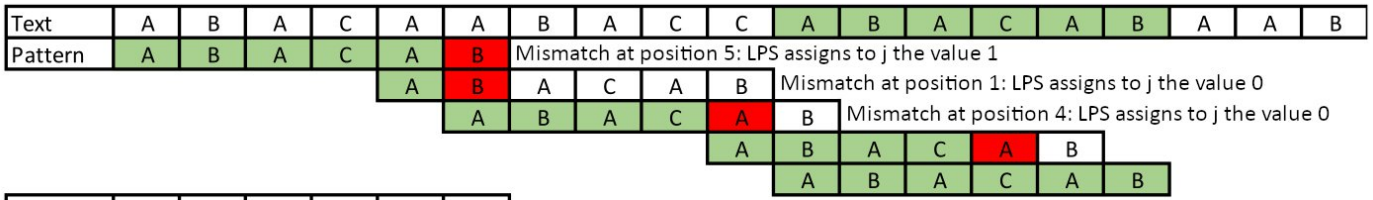
When two hashes match, the related strings are compared for a thorough check.



Source: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching

Knuth-Morris-Pratt

Based on skipping characters that were already checked after a mismatch occurs. To easily identify the length of the jump, **a prefix table, called LPS, has to be constructed and used**. The purpose of this table is to find prefixes similar to the suffix of the pattern's substring, ending with the character preceeding the mismatch, to move to the point of comparison.

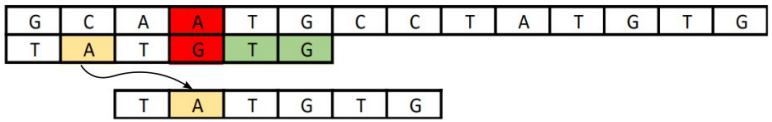


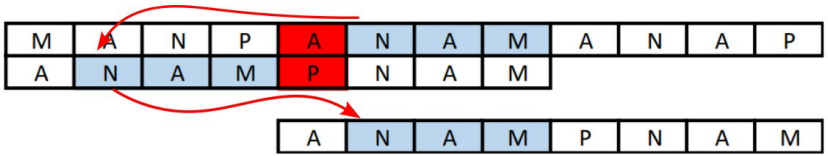
j	0	1	2	3	4	5
Pattern	A	B	A	C	A	B
LPS	0	0	1	0	1	2

Boyer-Moore

The algorithm's idea is to skip portions of the text based on two strategies:

- **Bad-Character rule**: finds the character which caused the mismatch inside the pattern and shifts it to the comparison point;
- **Good-Suffix rule**: finds a substring similar to the suffix of the mismatch and place it after the failed comparison point. It has different variations of the placing based on the scenario.



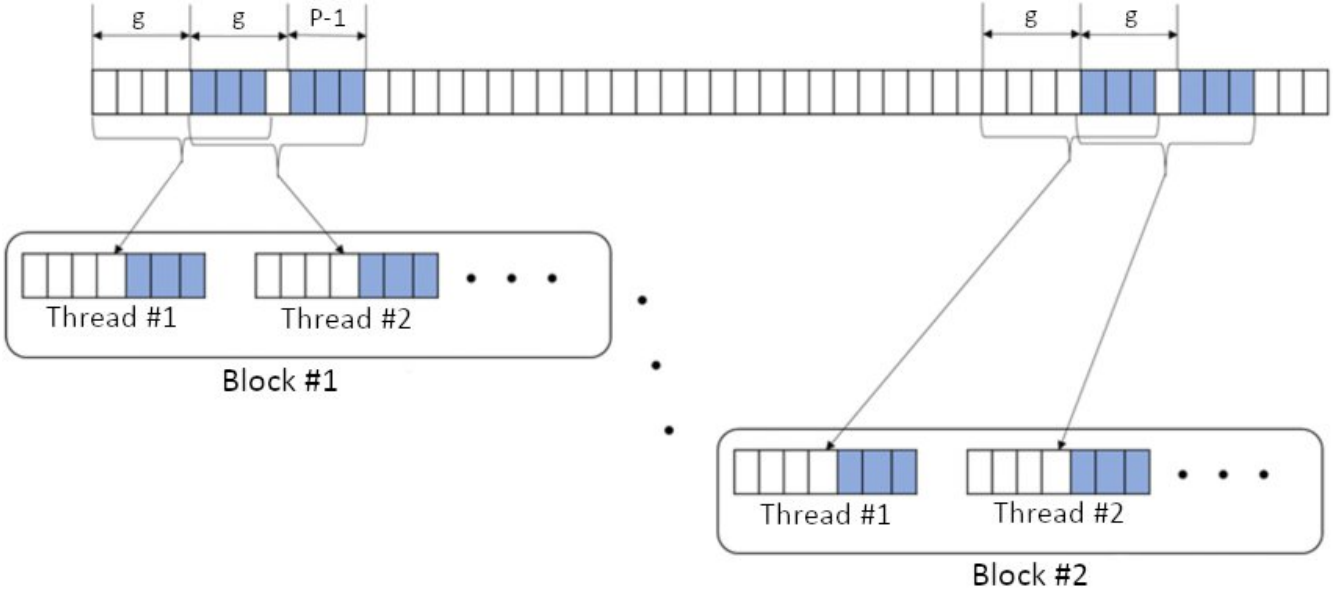


Code implementation

Given the strong sequential nature of the algorithms, the kernels were built based on a **“Divide and Conquer” strategy**. Basically, the scan problem was divided among the different threads. Doing so, allowed each of them to analyze only a small part of the overall text, thus reducing elaboration time.

The length of the substring each thread has to take care of is determined by the user at start-up. This value is called **granularity** (*g*). However, to properly check the substrings related to the last characters, every kernel launch has to consider at least *P* - 1 more symbols, where *P* is the pattern string’s length.

Code implementation



Code implementation

Two kernel versions were developed for each algorithm:

- a **naive** implementation, where each algorithm's structure has been maintained the same as their sequential counterpart. The only modifications introduced were related to the thread's identification of the portion of the text to control for possible matches;
- a **memory optimized** implementation, where the pattern's information and pre-processed tables are moved inside the shared memory of each block for faster load/store operations. To further enhance their performance, these versions make use of stream structures. The number of streams to launch is a user defined parameter, requested at start-up.

Code implementation

To analyze the case where multiple searches have to be performed inside the text, a second operative mode was introduced. In this scenario, the algorithms' implementation remains the same, while the control code presents some differences:

- only the optimized version of the Rabin-Karp can be launched;
- a maximum number of eight patterns can be searched concurrently, where each search is tied to a specific stream.

Test system

Two hardware devices were used to test and evaluate the developed program's performance:

- **NVIDIA Jetson Nano,** with an 128-core Maxwell GPU, 4 GB of RAM and a quad-core ARM Cortex-A57 CPU;
- **NVIDIA RTX 3070,** with 5888 CUDA cores, divided between 46 streaming multiprocessors, and 8 GB of RAM.

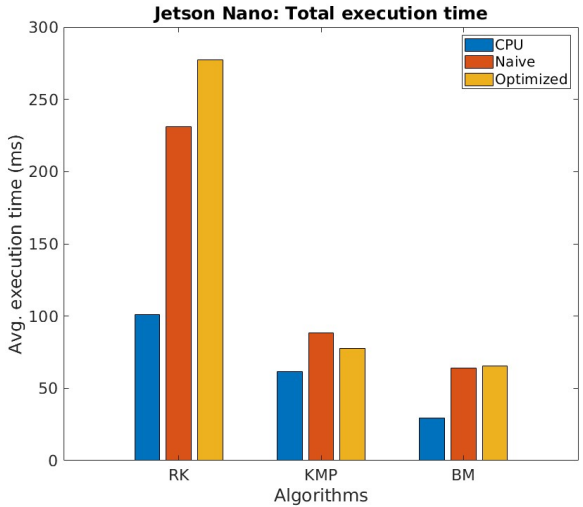
Test parameters

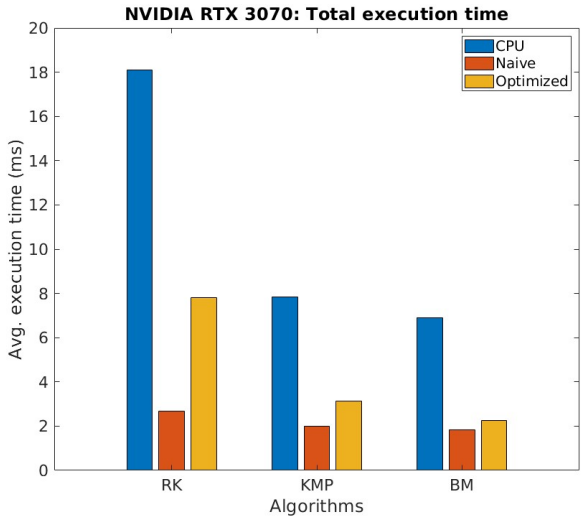
For the testing of the program the same parameters were used for both the hardware platforms, to easily compare the results. For the optimized kernel's launch, eight streams were used, based on the CUDA_DEVICE_MAX_CONNECTIONS environment variable.

As for g, a simple test was performed to identify its optimal value: the same pattern had been searched with every algorithm while increasing the granularity value from 100 to 10000. At the end of the process, it was found that 1000 characters of length represents an optimal trade-off between the length of substring to analyze and the performance of the sequential strategies employed.

The text used for the following metrics is the Bible. Five different patterns, of different length and frequency of matching, have been sampled to be used as input by the algorithms.

Observed results





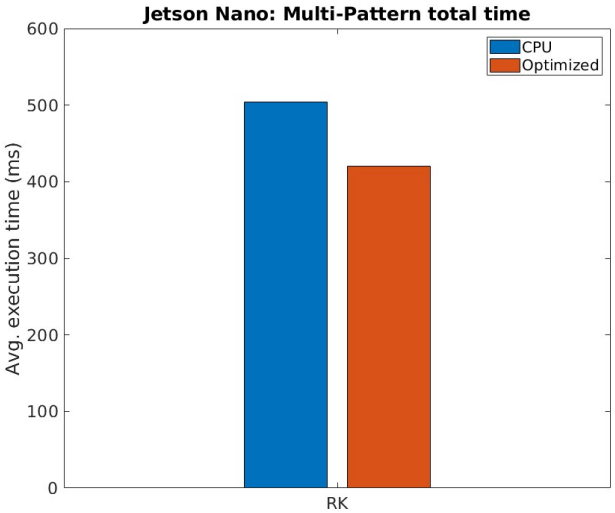
The results shown on the left are the average times computed after searching the five patterns, for each algorithm.

The total times contain both the memory transfer times and execution time of the kernels.

Observed results

The reduction of load operations requested to the global memory is reported in the table at the bottom. On the right, the performance difference of the multi-pattern search's execution time on the GPU and on the CPU.

	Global memory accesses (naive)	Global memory accesses (opt)	Accesses reduction (%)	Shared memory accesses (opt)
Rabin-Karp	$\sim 4.05M$	$\sim 3.96M$	2.21%	$\sim 325K$
KMP	$\sim 1.16M$	$\sim 558K$	52.07%	$\sim 21K$
Boyer-Moore	$\sim 473K$	$\sim 98K$	79.38%	$\sim 5.7K$



Thank you for your attention!

More information can be found on the project report.