POLITECNICO MILANO 1863

NECST laboratory

# Accelerating exact string matching on GPU

Antonio Sgarlata

December 20, 2021

**Abstract**

This report describes a GPU-accelerated implementation of the Knuth–Morris–Pratt string-searching algorithm, developed with *CUDA*, a parallel computing platform and programming model offered by *NVIDIA*. The report will first discuss string-searching and its main applications. Then it will focus on describing the key points of the KMP algorithm and will outline related works on its parallelization. Finally, it will illustrate how the algorithm was adapted to leverage GPU resources and how the implementation could be further extended.

## 1 Introduction

String-searching (or string-matching) algorithms attempt to find occurrences of one or more strings, often called patterns, within a larger one, often called text. Both pattern and text are strings of a given alphabet $\Sigma$, i.e., sequences of characters from a finite set. Examples of $\Sigma$ are the English alphabet ($\Sigma = \{a, \dots, z\}$), the binary alphabet ($\Sigma = \{0, 1\}$), and the DNA alphabet ($\Sigma = \{A, C, G, T\}$).

More formally, we want to find all the occurrences of a pattern $x = [x_0 \dots x_{m-1}]$; $x_i \in \Sigma$; $i = 0, \dots, m-1$, in a text $y = [y_0 \dots y_{n-1}]$; $y_i \in \Sigma$; $j = 0, \dots, n-1$.

String-searching algorithms are relevant to many real-world problems, with applications like, inter alia, intrusion detection, plagiarism detection, bioinformatics, digital forensics, text mining research, and video retrieval [1].

The most simple string-searching algorithm is the so-called naive (or brute-force) one, which consists in sliding the pattern over the text one character at a time and checking for a match. Although convenient, as no preprocessing or extra space is required, this algorithm is slow, having a time complexity of $O(nm)$.

One of the most efficient string-searching algorithms was published jointly by Donald Knuth, James H. Morris and Vaughan Pratt in 1977 [2]. The fundamental idea behind the KMP algorithm is that whenever we detect a mismatch between the text and the pattern, the latter bears enough information to know whether we could skip some of the characters in the following comparison window. In order for this to happen, an auxiliary structure is needed to know exactly how far to slide the pattern relative to the text (i.e., how many characters to skip) at each mismatch. This structure serves as a *failure table* and gets constructed in a preprocessing phase by analyzing the pattern. In particular, it is an integer array as large as the pattern, which stores at index $i$ the length of the longest substring of the pattern (considered up to its $i^{th}$ position) which is both a proper prefix and a suffix. Thanks to this failure table we can avoid considering all those characters that would match anyway, quickening the research. Since the preprocessing and matching phases have, respectively, complexities of $O(m)$ and $O(n)$, the overall time complexity of the algorithm is $O(n + m)$.

# 2 Related work

Similar efforts to accelerate the KMP algorithm by executing it on GPUs are present in the literature.

A. Rasool and N. Khare give an overview of the parallelization of the algorithm [3]. In addition, they present a parallelized version for a SIMD architecture that follows an idea similar to the one here later described (i.e., dividing the text into different parts and looking for the pattern within them in parallel). However, their implementation suffers some complications when the pattern comes at what they call *data division part or connection point*.

X. Bellekens et al. further investigate the performance gains of GPU-accelerating the algorithm, with a particular focus on its application on Deep Packet Inspection and Intrusion Detection Systems [4]. They claim that their empirical results show a potential improvement by a factor of 29 when executing the KMP algorithm on GPUs instead of CPUs, using shared memory and loop unrolling. Their implementation of the algorithm is based on the CUDA framework as well, and follows a similar parallelization approach.

# 3 Implementation

The implementation's codebase can be found at this GitHub repository, which contains the `kmp.cu` CUDA source code file, together with requirements and instructions on how to to execute it. Both text (in the form of a text file) and pattern are passed to the program from the command line. The research is performed by considering one line of the file at a time. Some sample text files are present in the `text` folder included in the repository.

## 3.1 Preprocessing

The `main()` function is mostly used to read each line of the file and to invoke the `kmp()` **function on each of them**. Despite its name, this function represents only part of the actual algorithm. It is concerned with its orchestration and mostly serves as a wrapper function. First, `kmp()` calls `computeF()`, the subroutine which handles the computation of the failure table following the approach described above.

## 3.2 Matching

Next, `kmp()` launches the `patternMatch<<<>>>()` kernel, the function which performs in the GPU the actual matching process. Being a CUDA kernel, an execution configuration must be provided to specify the thread hierarchy of its launch. The thread hierarchy defines the number of thread groupings (called blocks), as well as how many threads to execute in each block. Settling on a choice of threads and blocks is not an easy task. One would be tempted to parallelize as much as possible, and this would generally yield good performances. However, threads could eventually start conflicting, slowing down the execution. During most of the development, for simplicity's sake, I opted for an execution configuration of 2 blocks of 4 threads each. However, several configurations were tested and their resulting execution times are later provided. The basic idea behind the parallelization of the algorithm is to take the text and divide it by the total number of threads in the grid. Each thread will be therefore assigned a portion of text and will separately and asynchronously execute the algorithm on it. To determine where its portion starts and ends, a thread only needs to know the length of a text portion and its unique index within the grid. Before presenting the required formulas, we introduce some useful CUDA-provided variables which refer to the current execution. `gridDim.x` is the number of blocks in the grid. `blockIdx.x` is the index of the current block within the grid. `blockDim.x` describes the number of threads in a block. `threadIdx.x` describes the index of the thread within a block. Hence, the formula to compute the length of a text portion (called `sublength`) is `ceilf((float) N / (gridDim.x * blockDim.x))`, while the formula to compute a thread's unique index within the grid is `threadIdx.x + blockIdx.x * blockDim.x`. At this point, each thread executes the KMP algorithm indexing the text string from position `idx * sublength` (included) to position `idx * sublength + sublength` (excluded). It may happen that the pattern is located across two or more different threads' portions of text. Yet, the implementation guarantees that only one thread will be able to match it, namely the first one (i.e., the one whose portion contains the starting character of the pattern). To collect the algorithm's outcome, an integer array of the same length of the text is used. The array is first initialized to 0. Whenever a match is found in the text from position `i` onwards, a 1 is stored on the outcome array at that same position.

### 3.3 Verification and output of results

Once all the threads have returned from the kernel (they operate asynchronously and must be waited for via an explicit synchronization point), `kmp()` invokes a subroutine that applies the naive string-searching algorithm to the same text line and pattern. The obtained results are stored in an additional array with the same structure as the one described above and are used as a baseline for verification. Verification and output of the algorithm's results are performed together by analyzing in parallel the two outcome arrays. If both the arrays store a 1 at the same index, a match in the text from that same index was found by both algorithms and we call it a *verified* match. Instead, if the two arrays differ at a given index, the implementation yielded some error (the naive algorithm is always assumed to be correct). A match found by the naive algorithm, but not by the KMP one, is defined as *missed*. Conversely, it is defined as *unverified*. While scanning the arrays and following the logic just described, on the one hand, the program prints the position of the given match (tagging it with one of the three definitions), on the other hand, it updates accordingly two integer variables used to count the total number of verified and missed matches within the whole file. When all the lines of the text file have been read, the `main()` prints the total number of verified (and missed, if any) occurrences of the pattern within the text. Instead, if no occurrence was detected, it will communicate so.

## 4 Algorithm analysis

Considering the execution of this GPU-accelerated version of the algorithm to search for a pattern of length $m$ on a single line of text of length $n$, performances are greatly improved compared to the sequential version. Indeed, assuming to have $t$ threads in the grid and negligible overhead, the algorithm's time complexity is $O(\frac{n}{t} + m)$.

## 5 Experimental results

The experiments focused on comparing the runtime of the `patternMatch<<<>>>()` kernel execution when looking for the pattern $ACGT$ within the `DNA.txt` sample text file. They were performed on a workstation with an eight-core i7-6700 CPU running at 3.40GHz, 32 GB of RAM, and featuring an Nvidia GeForce GTX 960 GPU. The OS was Red Hat Enterprise Linux Server 7.9. The CUDA version was 11.3.

The chosen execution configuration ranged from 1 block of 1 thread (basically a sequential execution) to $\frac{N+31}{32}$ blocks of 32 threads, where $N$ is the number of characters in the text (which means that there were at least N threads in the grid, but only 1 block's worth extra, thus ensuring optimal parallelization).

| Execution configuration | Total time (ns) |
|---|---|
| 1,1 | 20,466,655 |
| 1,2 | 11,077,641 |
| 2,4 | 3,162,202 |
| 4,8 | 979,338 |
| 8,16 | 496,755 |
| 16,32 | 360,860 |
| (N+31)/32,32 | 331,095 |

## 6 Conclusion and future work

This report presented a detailed and usable implementation of the KMP string-searching algorithm based on the CUDA framework offered by NVIDIA. The experimental results confirmed the performance improvements of accelerating string-searching algorithms with GPUs, especially as the parallelization increases.

Future work could focus on accelerating also the computation of the algorithm's auxiliary structure and further improving and extending the implementation's scope.

# References


[1] Vidya SaiKrishna, Akhtar Rasool, and Nilay Khare. String matching and its applications in diversified fields. *International Journal of Computer Science Issues (IJCSI)*, 9(1):219, 2012.

[2] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[3] Akhtar Rasool and Nilay Khare. Parallelization of kmp string matching algorithm on different simd architectures: Multi-core and gpgpu's. *International Journal of Computer Applications*, 49(11), 2012.

[4] X Bellekens, I Andonovic, RC Atkinson, C Renfrew, and T Kirkham. Investigation of gpu-based pattern matching. In *The 14th Annual Post Graduate Symposium on the Convergence of Telecommunications, Networking and Broadcasting (PGNet2013)*. Citeseer, 2013.