Cognitive Computation
https://doi.org/10.1007/s12559-019-09646-y

A Novel Algorithm for Online Inexact String Matching and its FPGA Implementation

Alessandro Cinti[1] · Filippo Maria Bianchi[2] · Alessio Martino[1] · Antonello Rizzi[1]

Received: 25 May 2018 / Accepted: 30 April 2019

Abstract
Among the basic cognitive skills of the biological brain in humans and other mammals, a fundamental one is the ability to recognize inexact patterns in a sequence of objects or events. Accelerating inexact string matching procedures is of utmost importance when dealing with practical applications where huge amounts of data must be processed in real time, as usual in bioinformatics or cybersecurity. Inexact matching procedures can yield multiple shadow hits, which must be filtered, according to some criterion, to obtain a concise and meaningful list of occurrences. The filtering procedures are often computationally demanding and are performed offline in a post-processing phase. This paper introduces a novel algorithm for online approximate string matching (OASM) able to filter shadow hits on the fly, according to general purpose priority rules that greedily assign priorities to overlapping hits. A field-programmable gate array (FPGA) hardware implementation of OASM is proposed and compared with a serial software version. Even when implemented on entry-level FPGAs, the proposed procedure can reach a high degree of parallelism and superior performance in time compared to the software implementation, while keeping low the usage of logic elements. This makes the developed architecture very competitive in terms of both performance and cost of the overall computing system.

Keywords FPGA · Approximate string matching · Dynamic programming · Systolic arrays

Introduction

Pattern recognition is an innate human skill of utmost importance in numerous activities [1]. Basic cognitive psychology categorizes pattern recognition approaches in three models [1, 2], namely template-based, prototype-based, and feature-based. According to the first model, humans store learned schema in their long-term memory and the new stimuli (input patterns) are best-matched with these templates. As per the second model, humans maintain average prototypes and the matching procedure is more flexible with respect to the former case as it allows variability in recognizing novel input stimuli. Finally, according to the latter model, humans match features extracted from the input pattern rather than the entire pattern with other patterns or prototypes. Regardless of what is currently considered the most reliable model, this kind of recognition via matching procedures leads to perception, which humans experience in everyday life.

Computer scientists and machine learning engineers tried to mimic this human capability by developing biologically inspired models. The most investigated application areas are image analysis and foreground–background scenario identification (also due to closeness to the most fundamental source of input stimuli: the human eyes) [3–6]. However, in computational sciences, recognizing visual objects is not the only application for models based on the human-inspired template matching procedure. In fact, much effort has been

✉ Filippo Maria Bianchi
filippo.m.bianchi@uit.no

Alessandro Cinti
alessandro.cinti@gmail.com

Alessio Martino
alessio.martino@uniroma1.it

Antonello Rizzi
antonello.rizzi@uniroma1.it

1 Department of Information Engineering, Electronics and Telecommunications, University of Rome “La Sapienza”, Via Eudossiana 18, 00184 Rome, Italy

2 Machine Learning Group, Department of Physics and Technology, UiT the Arctic University of Norway, Hansine Hansens veg 18, 9019 Tromsø, Norway

done to design matching procedures for several types of structured data, including graphs and sequences [7]. For example, the latter are particularly useful to represent text data, which is processed in natural language processing applications [8–11].

Approximate string matching (ASM) studies the problem of matching two generic sequences defined on the same alphabet and, unlike in exact string matching, a level of inexactness is allowed: two strings match if their dissimilarity is below a given threshold [12]. ASM is a particular case of subgraph matching [13, 14] that has been applied in different fields of science and technology, including computational biology, signal processing, and text retrieval. In computational biology, scientists try to detect frequent patterns into DNA sequences of nucleotides (motifs) using ASM to account for mutations and evolutionary alterations of the genome sequences [15–18]. In signal processing, ASM can be used to identify patterns which have been corrupted by noise during storage, transmission, processing, or conversion stages. The main application fields are message exchanges, wireless communications, audio, image, and video processing [19, 20]. ASM is extensively used for information retrieval in large text collections where, due to the large amount of data and the high variety of content, classical string matching procedures are usually not suitable [21–25].

ASM also plays a fundamental role in granular computing, which is a human-inspired computing and information processing paradigm that explores multiple levels of granularity in data [26]. The concept of granular computing arose from many branches of natural and social sciences [27–29], and it is at the basis of recently developed frameworks in computational intelligence [30–32]. In this context, an ASM technique that identifies frequent and meaningful motifs in sequence databases allows designing advanced machine learning systems such as symbolic histogram approaches [33–35], where each pattern (a sequence of objects/events) can be represented by a histogram of motif instances.

The dissimilarity between two strings can be evaluated as the cost of the edit operations required for transforming a string into the other. The most common dissimilarities are the Levenshtein, Hamming, Episode, and Longest Common Subsequence distance; they differ on the edit cost definition and on the type of the allowed edit operations, and their computational costs range from linear to NP-complete [36]. The matching procedure can be implemented following offline or online approaches, which differ on how sequences are searched and indexed [37]. Several algorithms for online pattern matching have been designed and they can be grouped into four main classes, namely the Dynamic Programming (DP), Automata, Bit-parallel, and Filtering approaches [12]. The main advantage of online ASM is providing detection in real time, which is essential in situations where a prompt response is required.

Cybersecurity is a prominent example of application of online ASM. In particular, network intrusion detection systems (NIDSs) are devices or software applications used to identify individuals who are using a computer system without authorization and whoever has legitimate access to the system, but is exceeding his privileges [38]. Traditional NIDSs relied on exact pattern matching to detect an attack. However, by changing the data used in the attack even slightly, it is possible to evade detection. Therefore, a more flexible detection system is required to scan efficiently *in real-time* the whole inbound and outbound traffic to match patterns from a library of known attacks, without compromising the overall network speed.

In molecular biology, online ASM is exploited in quantitative real-time polymerase chain reaction (PCR) [39]. Real-time PCR aims at amplifying a small target DNA sample during PCR, performing a quantitation step after amplification. This technique is useful when only a small amount of DNA is available, which is insufficient for performing an accurate analysis. In medicine, real-time PCR is applied to the discovery of tumor cells, to the diagnosis of infectious diseases, and to a plethora of other predictive medicine and diagnostic tasks [40]; it is widely used for studying genomes in several bacteria and protists which cannot be cultured; in legal medicine and forensics, it has been demonstrated to be crucial for analyzing fingerprints and stains found at crime scenes [41, 42].

To increase the time performance of string matching procedures, especially when dealing with big-data, their concurrent processes can be effectively parallelized at CPU level by executing microinstructions simultaneously, or at circuit level by re-implementing arithmetic and logic operations [43, 44]. Hardware implementation provides an alternative solution to improve the speed of ASM algorithms and several architectures have been proposed. The fastest implementations are custom architectures deploying a large number of processing elements designed ad hoc to execute a specific algorithm [45]. Other architectures, instead, are based on general-purpose processors that can be adopted to implement different algorithms [46, 47]. Configurable devices, such as the field-programmable gate array (FPGA), represent flexible hardware platforms that are used to accelerate the computation, providing both a high degree of programmability and reduced design time and costs [48, 49].

None of the existing methods for performing online ASM accounts for the presence of multiple shadow hits, which are overlapping hits that appear as a consequence to the flexibility in ASM. Usually, such shadow hits are filtered afterwards by means of a post-processing phase which increases the overall computational burden. Especially when dealing with data streams, this solution turns out to be unfeasible both in terms of memory and time complexity, hence the necessity to filter out shadow hits on the fly.

This paper introduces the online approximate string matching (OASM) algorithm for retrieving a set of inexact matches of a known pattern from a stream of symbols, while filtering shadow hits on the fly. The proposed method performs detection in real time and is suitable for real-time applications and big-data streams. OASM is mainly based on the evaluation of the Levenshtein distance between a target pattern and a set of sequences, obtained by shifting windows of pre-established lengths over the stream. OASM follows a DP approach that provides a remarkable speedup thanks to the massive parallelization capability deriving from the inherent simplicity and regularity of its structure. Both a software (SW-OASM) and a hardware (HW-OASM) implementation of the algorithm are proposed in this paper. HW-OASM is based on the systolic array principle for the computation of the Levenshtein distance that extends the work in [50]. Compared to its software counterpart, experimental results show the effectiveness of the HW-OASM especially when integrated in a multiple online approximate string matching (MOASM) system. The latter implements multiple instances of HW-OASM to perform simultaneously parallel searches of distinct patterns on a common input stream of symbols (not necessarily finite). The performance of the proposed algorithm is first evaluated in a controlled environment using synthetic data. Successively, the algorithm is applied to a case study in bioinformatics on real genome data. The analysis consists in mining part of the human genome for RNA-binding protein sites.

The remainder of the paper is organized as follows. Section "Related Works" reviews related ASM approaches, including hardware-based ones. Section "An Overview on the Levenshtein Distance" introduces basic concepts and details of the dissimilarity measure considered. Section "An Online Search Algorithm Based on Approximate String Matching" describes the OASM algorithm, and in the Section "Case Study," a real-world human genome case study is presented to highlight the features of SW-OASM, compared to other ASM approaches. Next, Section "Proposed Hardware Implementation" provides details of the hardware implementation, and in Section "Experiments," the performances of SW-OASM and HW-OASM are compared on synthetic data. Finally, in Section "Conclusions," conclusions are reported.

Related Works

In literature, there are two major approaches based on DP for performing ASM on dedicated hardware, namely FPGAs and (GP-)GPUs. In this paper, we focus on FPGAs and refer the interested reader to works such as [51–54] for ASM implementations on (GP-)GPUs. The first approach relies on local/global sequence alignment [55] (e.g., the Smith–Waterman's algorithm [56], the Needleman–Wunsch's algorithm [57], or Myers' fast bit-vector algorithm [58]), while the second approach exploits the Levenshtein distance (e.g., the Wagner–Fischer's algorithm [59]).

Implementation of the Smith–Waterman's algorithm for optimal local alignment on FPGAs can be found in [60], where the authors implement the plain Smith–Waterman's algorithm. In [61], the Smith–Waterman's algorithm has been implemented on FPGAs without relying on systolic arrays in order to exploit all processing units, whereas in [62], the authors used systolic arrays driven by OpenCL. In [63], the authors provide FPGA implementation for DP and BLAST routines [64]. It is noteworthy that since the Smith–Waterman's algorithm was primarily developed for nucleotide or protein sequence alignment, all works cited so far are with regard to bioinformatics-related applications. Conversely, in [65], the authors implement the Smith–Waterman's algorithm on FPGA for generic text search. Finally, in [66], the authors use an FPGA in order to speed up Myers' algorithm.

General-purpose applications are more frequently relying on the second approach, based on the Wagner–Fischer's algorithm. In [50], the authors adopt the plain Wagner–Fischer's algorithm, along with an improved version that better exploits each cell. In [67], the Wagner–Fischer's algorithm has been applied to multimedia information retrieval, also considering text search paradigms such as wildcards and idioms. In [68], the authors provide an FPGA implementation of the Wagner–Fischer's algorithm particularly suited for dealing with regular expressions.

FPGA implementations of systolic architectures have been proposed for ASM [50, 65] and, in bioinformatics, for DNA sequence alignment [60, 61, 63]. In the field of music information retrieval (MIR), circuits implementing ASM with dynamic programming cannot retrieve fragments with different sizes [67] and other architectures with higher flexibility only support symbol substitution [69]. Better performance was achieved by application-specific integrated circuit and FPGA with a comparable computational time on different MIR approaches [70–73]. An efficient FPGA implementation of ASM has been proposed for text mining, where a restricted class of regular expressions is used to define the patterns to search [68]. A work closely related to this paper is a successful implementation of online ASM on FPGA for NIDS applications [74].

Regardless of the particular algorithm used to implement ASM (online and/or using dedicated hardware), none of the works previously discussed filters shadow hits on the fly. In fact, those are discarded a posteriori, increasing the overall computational burden. The proposed OASM jointly provides (i) the ability of performing simultaneously detection and shadow hits filtering; (ii) the ability of working with data streams, avoiding to store the entire input sequence and/or all multiple shadow hits. Contrarily to previous works, in OASM, not only is the plain Levenshtein distance parallelized, but so is the filtering procedure. This makes the proposed approach a perfect candidate for being implemented in FPGAs.

An Overview on the Levenshtein Distance

Let the pattern $\mathbf{p} = \langle p^{(0)}, p^{(1)}, \dots, p^{(l_p-1)} \rangle$ and the string $\mathbf{t} = \langle t^{(0)}, t^{(1)}, \dots, t^{(l_t-1)} \rangle$ be defined, respectively, as the concatenation of l_p and l_t symbols of a finite alphabet Σ and a generic substring $\mathbf{s} = \langle s^{(0)}, s^{(1)}, \dots, s^{(l-1)} \rangle = \langle t^{(i)}, t^{(i+1)}, \dots, t^{(i+l-1)} \rangle = \mathbf{t}[i, l]$ be a subset of l contiguous symbols in $\mathbf{t}$ starting from position i. The dissimilarity between strings $\mathbf{p}$ and $\mathbf{s}$ can be measured by means of the Levenshtein distance $\text{lev}(\mathbf{p}, \mathbf{s}) \in \mathbb{N}_0$, which is the minimum number of single-character edits (insertion, deletion, substitution) necessary to transform $\mathbf{p}$ into $\mathbf{s}$. Let $\mathbf{C}$ be a matrix of size $(l_p + 1) \times (l + 1)$, whose elements are:

$$c_{n,m} = \begin{cases} n, & \text{if } m = 0 \\ m, & \text{if } n = 0 \\ \min(c_{n-1,m} + 1, c_{n,m-1} + 1, & \\ c_{n-1,m-1} + \delta_{n,m}), & \text{if } n, m > 0 \end{cases} \tag{1}$$

where $\delta_{n,m} = \left(p^{(n)} \neq s^{(m)}\right)$, $n = 0, \dots, l_p$, $m = 0, \dots, l$. The Levenshtein distance between $\mathbf{p}$ and $\mathbf{s}$ corresponds to the element $c_{l_p,l}$ of the Levenshtein matrix $\mathbf{C}$. Equation 1 shows that, for $n, m > 0$, each element $c_{n,m}$ of the matrix $\mathbf{C}$ can be computed just by knowing its upper ($c_{n-1,m}$), left ($c_{n,m-1}$), and upper-left ($c_{n-1,m-1}$) neighbors. Applying the DP method to the Levenshtein distance computation consists in building column-wise (or row-wise) the matrix $\mathbf{C}$, element by element, by solving iteratively the same simple problem, whose result will be used as input of one of the successive iterations. Although the algorithm is $\mathcal{O}\left(l_p \cdot l\right)$ in time, the space complexity is only $\mathcal{O}\left(\min(l_p, l)\right)$ because only the previous column (or row) has to be stored to compute the new one.

Applying ASM between $\mathbf{p}$ and $\mathbf{t}$ consists in finding the set of all the substrings $\mathbf{s} \subset \mathbf{t}$ with length $l_s \in [l_p - K, l_p + K]$ that satisfy the condition:

$$\text{lev}(\mathbf{p}, \mathbf{s}) = k \leq K, \tag{2}$$

where the threshold K represents the maximum acceptable level of inexactness [75]. Figure 1(a) depicts the matrix $\mathbf{C}$ resulting from the computation of the Levenshtein distance between the pattern $\mathbf{p} = \langle ABCD \rangle$ and the substring $\mathbf{s} = \langle ABBDA \rangle$ from the string $\mathbf{t} = \langle ABBDABCDACDB \rangle$ defined over the alphabet $\Sigma = \{A, B, C, D\}$ for $K = 1$ and $i = 0$. The gray-shaded cells in Fig. 1(a) represent the values of the Levenshtein distances between $\mathbf{p}$ and the three substrings $\mathbf{t}[0, 3] = \langle ABB \rangle$, $\mathbf{t}[0, 4] = \langle ABBD \rangle$, and $\mathbf{t}[0, 5] = \langle ABBDA \rangle$. There is no need to compute three different distances since by computing only the distance between $\mathbf{p}$ and $\mathbf{t}[0, 5]$ (element $c_{4,5}$ of $\mathbf{C}$), the other two distances are available as intermediate computations (elements $c_{4,3}$ and $c_{4,4}$ of $\mathbf{C}$, respectively). By increasing i, step by step, all the substrings of $\mathbf{t}$ are generated.

The DP method can be applied to compute subsequently the Levenshtein distances between $\mathbf{p}$ and all the substrings obtained by shifting a fixed mask of length $l_p + K$ over $\mathbf{t}$ [76]. The DP approach reduces the overall number of computations compared to brute force approaches or when the same computation is repeated over and over [77].

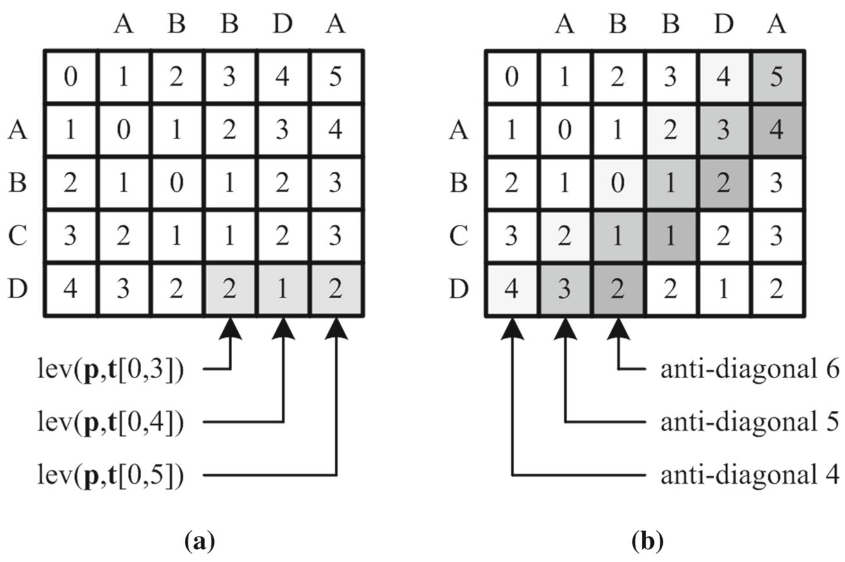


Fig. 1 Example of the matrix **C** computation with canonical (**a**) and wave-front (**b**) approaches

The DP approach for the Levenshtein distance computation exploits the relationship between each element $c_{n,m}$ in $\mathbf{C}$ and its three neighbors $c_{n-1,m}, c_{n,m-1}, c_{n-1,m-1}$. All the elements on an anti-diagonal j can be computed at the same time in a wave-front processing fashion, by combining the information contained in the anti-diagonals $j-1$ and $j-2$, together with the information related to the symbols of the two compared strings. Starting from the element $c_{0,0}$ and ending with the element $c_{l_p,l}$, the matrix $\mathbf{C}$ is filled in $2l_p + K - 1$ steps, as shown in Fig. 1(b). On the first step, the anti-diagonals 0 and 1 are combined to obtain the anti-diagonal 2, and so on, up to the step $2l_p + K - 1$, where the anti-diagonals $2l_p + K - 1$ and $2l_p + K$ are combined to obtain the anti-diagonal $2l_p + K + 1$.

An Online Search Algorithm Based on Approximate String Matching

This section presents the proposed online algorithm that combines the features of the DP method with a search criterion based on priority rules to find inexact occurrences of a known pattern $\mathbf{p}$ of length l_p within a continuous stream of symbols $\mathbf{t}$. All the found occurrences are stored in a set $\mathcal{S}$ to be used for further processing. Collecting all the substrings $\mathbf{s}$ of length $l \in [l_p - K, l_p + K]$ from the text $\mathbf{t}$ that verify Eq. 2 at each position i, may lead to multiple hits of the same occurrence of $\mathbf{p}$ [78]. Due to the online setting considered, it is not possible to decide whether an occurrence can be added to $\mathcal{S}$ at the time it is found, but it depends on the symbols of $\mathbf{t}$ that are not yet processed. For instance, if Eq. 2 is verified for two substrings $\mathbf{s}'$ and $\mathbf{s}''$ that start from two consecutive positions in $\mathbf{t}$, respectively i and $i+1$, the two hits will correspond to the same occurrence of $\mathbf{p}$. The next subsection introduces a set of rules that determine the priorities for accepting the hits found in the text $\mathbf{t}$ as valid occurrences.

Priority Rules

Not all the overlapping substrings verifying Eq. 2 should be considered as occurrences but, at the same time, the online nature of the problem complicates deciding on the fly which one should be kept or discarded. The proposed solution consists in assigning temporary priority values to the occurrences that verify Eq. 2. In the remainder of the paper, smaller values denote higher priorities and they depend on the degree of matching between $\mathbf{p}$ and $\mathbf{s}$, and on the position of the substrings in $\mathbf{t}$.

Three rules define a priority scheme that allows to build a greedy online algorithm with low complexity to solve the actual NP-complete problem.

R1: Consider that the substring $\mathbf{s}' = \mathbf{t}[i', l']$ for which $\text{lev}(\mathbf{p}, \mathbf{s}') = k' \leq K$ has already been found and assigned with a priority level k'. Any other overlapping substring $\mathbf{s}'' = \mathbf{t}[i'', l'']$ for which $\text{lev}(\mathbf{p}, \mathbf{s}'') = k'' \leq K$ becomes an occurrence only if $k'' < k'$.

R2: If $\mathbf{s}'$ and $\mathbf{s}''$ have the same priority, the first encountered substring becomes an occurrence, the other one is discarded.

R3: If $\mathbf{s}'$ and $\mathbf{s}''$ have the same priority and start at the same position in $\mathbf{t}$, the shorter becomes an occurrence.

```
R1:
  if then s' ∩ s'' ≠ ∅ and (k'' < k')
    s'' is an occurrence with priority k''
else
    s'' is discarded
R2:
  if then s' ∩ s'' ≠ ∅ and k' = k''
     if then i' < i''
       s' is an occurrence, s'' is discarded
    else
       s' is discarded, s'' is an occurrence
R3:
  if s' ∩ s'' ≠ ∅ and k' = k'' and i' = i'' then
     if l' < l'' then
       s' is an occurrence, s'' is discarded
    else
       s' is discarded, s'' is an occurrence
```

Validation Process

It is possible to notice that the three rules do not resolve completely the prioritization of the occurrences in presence of overlaps. For example, consider three overlapping substrings $\mathbf{s}'$, $\mathbf{s}''$, and $\mathbf{s}'''$, so that:

– $i' < i'' < i'''$;
– $k' > k'' > k'''$;
– $\mathbf{s}' \cap \mathbf{s}'' \neq 0, \mathbf{s}'' \cap \mathbf{s}''' \neq 0, \mathbf{s}' \cap \mathbf{s}''' = 0$

As $\mathbf{t}$ is processed symbol by symbol, priority $k' = 2$ is assigned to the occurrence relative to $\mathbf{s}'$. Then, the substring $\mathbf{s}''$ becomes an occurrence with priority $k'' = 1$ verifying rule R1. As more symbols are processed, $\mathbf{s}'''$ is found and according to rule R1 it becomes an occurrence with priority $k''' = 0$. Due to the condition $\mathbf{s}' \cap \mathbf{s}''' = 0$, only the occurrence relative to $\mathbf{s}''$ should be discarded. To ensure this behavior, it is necessary to introduce a further validation procedure, which decides when it is possible to safely discard an occurrence in presence of overlapping substrings.

The validation process proposed here uses one counter $r(k)$ per priority level:

– When an occurrence with priority $k < K$ is found, its validation begins and $r(k)$ starts counting from 0 up to a value equal to the occurrence length;
– If overlapping occurrences with higher priority $k^* < k$ are encountered, $r(k)$ keeps counting until the end of the occurrences.

When the validation process of the occurrence with the highest priority is completed (its relative counter $r(k)$ reaches the target value), it can be decided which of the validated occurrences can be added to $\mathcal{S}$. Starting from the occurrence with the highest priority and going down to the validated occurrence with the lowest priority, the substring relative to the highest priority is added to $\mathcal{S}$. Moving downwards, the validated occurrence with lower priority k is added to $\mathcal{S}$ if the following condition is verified:

$$r(k) - l(k^*) > l(k) \quad (3)$$

where $k^* < k$ represents the index (priority) of the occurrence relative to the last substring added to $\mathcal{S}$.

Algorithm 1 Online approximate string matching.

```
Input: pattern $\mathbf{p}$, text $\mathbf{t}$, threshold $K$
Output: set of occurrences $\mathcal{S}$
1: $l_p = |\mathbf{p}|, i = 0, \mathcal{S} = \emptyset, idx = K, ins = 0, acc = 0$, reset(mem)
2: while true do
3:     for $l : l_p - K \to l_p + K$ do
4:         $\mathbf{s} = \mathbf{t}(i; l)$
5:         $k = \mathrm{lev}(\mathbf{p}, \mathbf{s})$
6:             if R1 is TRUE then
7:                 $idx = k, ins = 1, \mathbf{mem}[idx][:] = [i, l, 1]$
8:         else  if R2 is TRUE or R3 is TRUE then
9:                 $ins = 1, \mathbf{mem}[idx][:] = [i, l, 1]$
10:       if $ins == 1$ and $\mathbf{mem}[:][3] == 0$ then
11:           for $j : idx \to K$ do
12:                 if $j == idx$ or $\mathbf{mem}[j][3] - acc < \mathbf{mem}[j][2]$ then
13:                     $\mathcal{S} = \mathcal{S} \cup \mathbf{t}(\mathbf{mem}[j][1]; \mathbf{mem}[j][2])$
14:                     $acc = \mathbf{mem}[j][2] + acc$
15:           $idx = K, ins = 0, acc = 0$, reset(mem)
16:       if $\mathbf{mem}[idx][3] \neq \mathbf{mem}[idx][2]$ then
17:           for $j : idx \to K$ do
18:               $\mathbf{mem}[j][3] = \mathbf{mem}[j][3] + 1$
19:     $i = i + 1$
```

The pseudo-code in Algorithm 1 summarizes the whole OASM procedure. The storage variable **mem** is a matrix of size $(K + 1) \times 3$, whose content is reset by calling "reset(**mem**)." **mem** stores in his columns three quantities for each one of the $K + 1$ possible priority levels of the occurrence: its position i (column 1), its length l (column 2), and its associated validation counter r (column 3). The variable *idx* contains the priority of last found occurrence and the variable *ins* is a flag that enables the counting check in the validation process. The counters $r(k)$ of the occurrences in **mem** are incremented by one each time a new symbol of **t** is processed. Finally, the variable *acc* is used to implement (3) and accumulates all the lengths l of the validated occurrence that has just been added to $\mathcal{S}$. Every time substrings are added to $\mathcal{S}$, the variables *idx*, *ins*, *acc*, and **mem** are reset. The symbol ":" selects all the indexes across one dimension.

Example The OASM algorithm is executed on the following example, where $\mathbf{p} = \langle ACBDA \rangle$ and $\mathbf{t} = \langle CCCCDACCBDACBDAA\ldots \rangle$ are defined over the alphabet $\Sigma = \{A, B, C, D\}$ and $K = 2$. Figure 2 illustrates the evolution through time of the variables stored within **mem**: three occurrences (one per priority) relative to different substrings are overlapping. As the counter of the occurrence with priority 0 reaches 5, the validation process is complete. The substring $\mathbf{s} = \mathbf{t}[10; 5] = \langle ACBDA \rangle$ is the one with the highest priority and it represents the occurrence that is validated. For the substring $\mathbf{s} = \mathbf{t}[7, 4] = \langle CBDA \rangle$, instead, Eq. 3 has to be evaluated. Substituting numerical values in $r(1) - l(0) > l(1)$ gives $8 - 5 > 4$, which is false, so the occurrence with priority 1 is discarded. Finally the substring $\mathbf{s} = \mathbf{t}[3, 3] = \langle CDA \rangle$ relative to the occurrence with priority 2 has to be added because evaluating $r(2) - l(0) > l(2)$ gives $12 - 5 > 3$, which is true. In the last evaluation, $l(0)$ is present rather than $l(1)$, since the last validated occurrence is the one with priority 0, whereas the one with priority 1 was discarded.

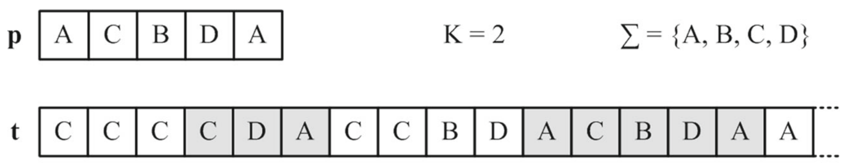
Complexity Analysis

The price to pay for filtering out unaccepted occurrences with the proposed validation process is that the validated occurrences are not instantaneously stored in $\mathcal{S}$.

To analyze the complexity of the algorithm, we consider the worst case scenario for a generic occurrence with priority k. It occurs when (i) overlapping occurrences with all priority values from 0 to K are found, (ii) all the occurrences have a maximum length, and (iii) each occurrence with priority $k - 1$ begins just before the end of the occurrence with priority k. In this case, the substring corresponding to the occurrence with the lowest priority k_{min} cannot be added to $\mathcal{S}$ before $d = l_p(k_{min} + 1) + k_{min}(k_{min} - 1)/2$ symbols of **t** are processed.

The space complexity corresponds to the size of the storage variable **mem**. According to the discussed worst case scenario, the upperbound for the space required to store **mem** is $\mathcal{O}\left(l_p(K + 1) + \frac{K(K-1)}{2}(2K + 1)\right)$.

Fig. 2 Content of **mem** over time, in the proposed example



index_stream	k = 0	k = 1	k = 2
0	---, ---, ---	---, ---, ---	---, ---, ---
1	---, ---, ---	---, ---, ---	i = 1, l = 5, r = 1
2	---, ---, ---	---, ---, ---	i = 2, l = 4, r = 1
3	---, ---, ---	---, ---, ---	i = 3, l = 3, r = 1
4	---, ---, ---	---, ---, ---	i = 3, l = 3, r = 2
5	---, ---, ---	i = 5, l = 6, r = 1	i = 3, l = 3, r = 3
6	---, ---, ---	i = 6, l = 5, r = 1	i = 3, l = 3, r = 4
7	---, ---, ---	i = 7, l = 4, r = 1	i = 3, l = 3, r = 5
8	---, ---, ---	i = 7, l = 4, r = 2	i = 3, l = 3, r = 6
9	---, ---, ---	i = 7, l = 4, r = 3	i = 3, l = 3, r = 7
10	i = 10, l = 5, r = 1	i = 7, l = 4, r = 4	i = 3, l = 3, r = 8
11	i = 10, l = 5, r = 2	i = 7, l = 4, r = 5	i = 3, l = 3, r = 9
12	i = 10, l = 5, r = 3	i = 7, l = 4, r = 6	i = 3, l = 3, r = 10
13	i = 10, l = 5, r = 4	i = 7, l = 4, r = 7	i = 3, l = 3, r = 11
14	i = 10, l = 5, r = 5	i = 7, l = 4, r = 8	i = 3, l = 3, r = 12
15	---, ---, ---	---, ---, ---	---, ---, ---

Case Study

microRNAs (miRNAs) [79, 80] are small non-coding primary transcripted molecules (approx. 22 nucleotides long) which serve as regulators of gene expression and are essential components of normal organism development. Found in plants, animals, and unicellular eukaryotes, miRNAs control diverse biological functions by promoting degradation or translation inhibition of target messenger RNAs or by carrying out post-transcriptional regulation of gene expression. miRNA genes are dispersed in various genomic locations (intronic, exonic, or intergenic regions) and can be transcribed independently or as a part of other host genes.

miRNAs are encoded within the genome and are often transcribed by RNA polymerase II or, rarely, by RNA polymerase III as long precursor transcripts (several hundreds or thousands of nucleotides), named primary-microRNAs (pri-miRNAs) [81, 82]. Those pri-miRNAs have the characteristic hairpin (or stem-loop) structure.

Within the nucleus, pri-miRNAs are cleaved by the microprocessor complex formed by proteins DROSHA and DGCR8 (also known as PASHA[1]) and the shorter hairpin structure (approx. 72 nucleotides) formed by this cleavage is referred to as precursor-microRNAs (pre-miRNAs).

The pre-miRNAs are transported from the nucleus to the cytoplasm by Exportin-5 together with Ran-Guanine TriPhosphatase [80]. In the cytoplasm, DICER, a cytoplasmic RNase III-type protein, dices the transported pre-miRs near the hairpin loop, yielding a mature miRNA-duplex of approx. 22 nucleotides. The duplex is loaded onto the Argonaute protein (*AGO*). One strand of the duplex (the passenger strand) is discarded, while the other strand (the guide strand or mature miRNA) remains in *AGO* to form an RNA-induced silencing complex [80].

[1]Partner of droSHA

The end positions of mature miRNAs are generally assumed as DROSHA cleavage sites. However, pre- and mature-miRNAs are often subject to end modification/processing such as trimming and tailing, which hinders the exact identification of DROSHA cleavage sites [83]. Processing of pri-miRNA stem-loops by the DROSHA/DGCR8 complex is the initial step in miRNA maturation and crucial for its function. Nonetheless, the underlying mechanism that determines the DROSHA cleavage site of primary transcripts has remained unclear: while the mechanisms of pre-miRNA recognition and cleavage by DICER are well characterized [84], an understanding of how DROSHA/DGCR8 selectively recognizes and precisely cleaves pri-miRNAs remains unclear [85]. The problem of identifying cleavage sites for DROSHA relies on the fact that failures at DROSHA processing step might lead to miRNAs downregulation, a phenomenon observed in cancer patients [86].

Baseline Algorithms for ASM

To show the effectiveness of the proposed OASM algorithm, it is compared to two baseline ASM techniques.

The first, referred as *Fully Naïve*, works in a brute force fashion [87] according to the following three steps:

1. From the input stream, extracts all possible adjacent substrings of any possible length;
2. Evaluates all pairwise Levenshtein distances between substrings and the target sequence;
3. Discards all substrings whose distance with respect to the target sequence is greater than the threshold K.

While being the most straightforward method, Fully Naïve does not consider multiple shadow hits and all the occurrences found must be stored in order to be filtered in a post-processing stage. Further, due to its brute force nature, it is unsuitable for processing large input streams, since the number of possible adjacent substrings of any possible length grows quadratically with the length of the text. Specifically, let l be the length of the text, the number of non-empty adjacent substrings of any possible length is $l(l+1)/2$. Accordingly, the time and space complexity for evaluating the Levenshtein distance with Wagner–Fischer's algorithm, as described in Section "An Overview on the Levenshtein Distance," is $\mathcal{O}(l \cdot l_p)$ with l and l_p being the input and target string lengths, respectively. Thus, the overall complexity in the best case is $\mathcal{O}(l_p)$, whereas in the worst case is $\mathcal{O}(l^2)$. Finally, the filtering procedure must scan all possible substrings generated in step 1, and its complexity is again $\mathcal{O}(l^2)$. By strictly following the three steps above, step 1 might lead to early out-of-memory errors due to the storage of $\mathcal{O}(l^2)$ possible substrings. To avoid this, for each extracted substring, one can evaluate the distance with respect to the target and discard the substring if the distance is above the user-defined threshold.

The second procedure, referred as *Less Naïve*, tweaks the Levenshtein matrix $\mathbf{C}$ [12, 68]. Again, let l and l_p be the lengths of the input stream and target string, respectively. According to Section "An Overview on the Levenshtein Distance," to keep track of the prefixes, one initializes the matrix $\mathbf{C}$ with a dummy row (0 to l) and a dummy column (0 to l_p), as shown in Fig. 1. Instead, if the dummy row is initialized with all zeros, the Wagner–Fischer's algorithm computes the Levenshtein distance between the target and substrings in the input stream. Therefore, rather than picking the bottom-right element, one takes all positions in the last row that are below the threshold K as the ending positions of the hits.

Qualitative Analysis of the Results

After consultation with field experts,[2] several human pre-miRNAs of interest have been selected for the analysis. As discussed above, since the end positions of mature miRNAs cannot be considered as reliable cleavage sites, mining pre-miRNAs neighborhood regions (both upstream and downstream) can give some further insights to biologists.

To evaluate the capability of the proposed algorithm to filter shadow hits on the fly, five of the suggested pre-miRNAs have been considered, along with the corresponding chromosomes. All data are freely available on ad hoc biological online databases: the human miRNAs can be found at miRBase [88], whereas the human chromosomes (assembly GRCh38) can be retrieved from GenomeBrowser [89]. The three algorithms, *Fully Naïve*, *Less Naïve*, and OASM have been evaluated on the same data using the same inexactness threshold $K = 2$. The number of hits and their positions are reported in Table 1 and Fig. 3 visually depicts the amount of overlapping occurrences found in each position.

From Table 1 and Fig. 3 clearly emerges that OASM detects each occurrence accurately just one time since, in contrast to the other two algorithms, it is able to discard those detected sequences that are overlapping. Indeed, neither Fully Naïve nor Less Naïve can deal with shadow hits, thus requiring a mandatory additional post-processing phase to validate the occurrences. Such a post-process results in additional computation and, most importantly, contrarily to the proposed OASM algorithm, the detection cannot be done online.

[2]Dr. Giulia Piaggio and Dr. Aymone Gurtner. Regina Elena Institute for Cancer Research, Rome, Italy.

Table 1 Number of hits across 5 different pre-miRNAs by considering 200 nucleotides neighborhood (both upstream and downstream)

Input stream	Target	Fully Naïve	Less Naïve	OASM
`hsa-mir-218-1` ± 200 nt (510 nt)	`aaaaaaaa`	29	13	4
`hsa-mir-515-1` ± 200 nt (483 nt)	`gcaacc`	64	39	19
`hsa-mir-519a-1` ± 200 nt (485 nt)	`acgttgca`	8	6	4
`hsa-mir-105-1` ± 200 nt (481 nt)	`aaccttgg`	6	6	3
`hsa-mir-1-2` ± 200 nt (485 nt)	`ctcattca`	7	7	5

The total number of nucleotides is shown in brackets

Table 2 summarizes the time complexity of the ASM algorithms. Execution times are not reported since the post-processing filtering operation, which is required when using Fully Naïve and Less Naïve algorithms, is not univocally defined. In fact, it can be implemented in several ways and executed at different times, in accordance to the requirements of the task at hand. A detailed discussion and analysis of the offline filtering operations for the naïve algorithms is beyond the scope of this work.

Proposed Hardware Implementation

Top-Level Instance: *LEV CORE*

The LEV CORE module computes the multiple Levenshtein distances between the *pattern* **p** and all the *substrings* **s** extracted from a stream of symbols **t** and returns only the list of occurrences *result* below a threshold K, according to the algorithm described in Section "An Online

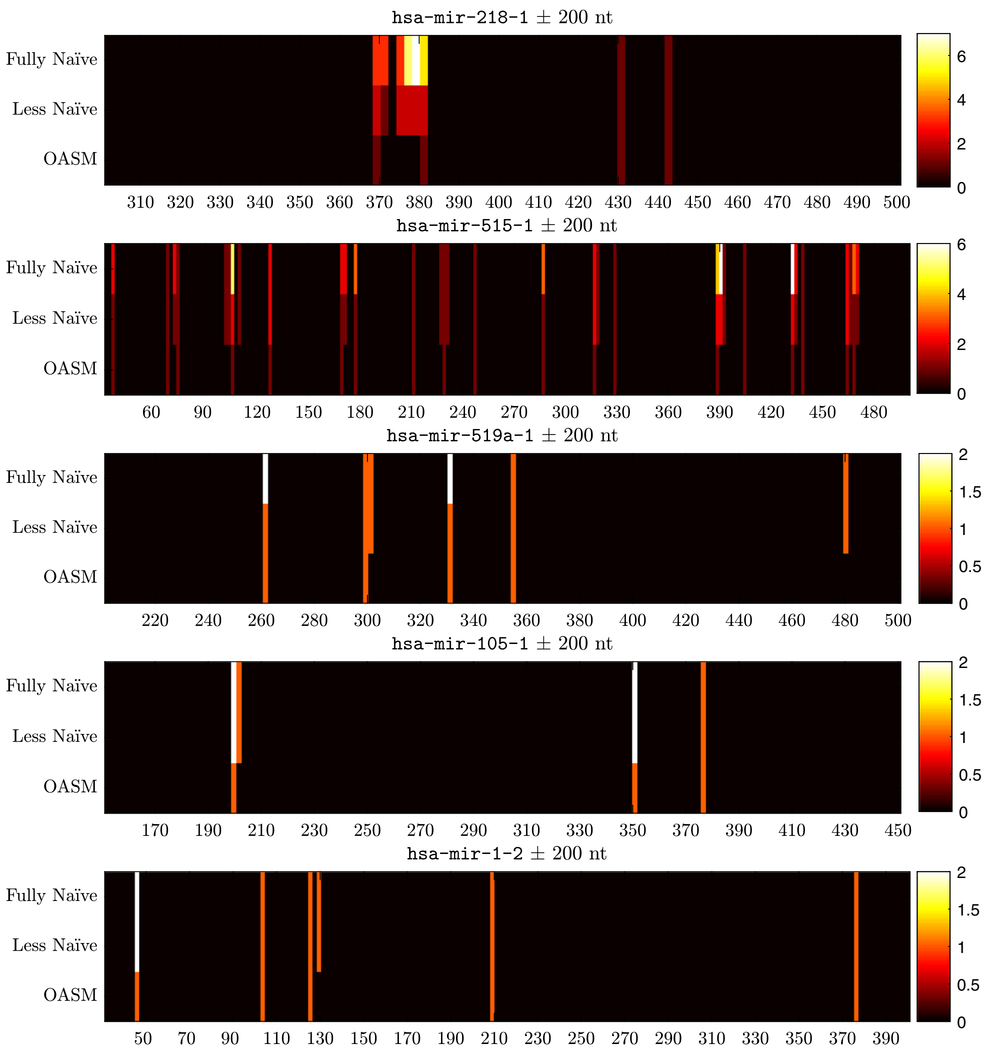


Fig. 3 Hit positions for experiments in Table 1. The color map indicates the number of overlapping occurrences found in a given position

Table 2 Computational complexities of the ASM algorithms

Algorithm	Time complexity
Fully Naïve	$\mathcal{O}(l^2) \cdot [\mathcal{O}(l_p) \rightarrow \mathcal{O}(l^2)] + \mathcal{O}(l^2)$ + offline filtering cost
Less Naïve	$\mathcal{O}(l \cdot l_p)$ + offline filtering cost
OASM	$\mathcal{O}\left(l \cdot \left(l_p(K+1) + \frac{K(K-1)}{2}\right)\right)$

Herein, let l and l_p be the length of the input stream and the length of the pattern to be searched within the input stream, respectively

Search Algorithm Based on Approximate String Matching.” Figure 4(a) depicts the main I/O ports and the two main sub-modules of the LEV CORE:

- LEV CALC computes at every new *index_stream* value the distances between **p** and the $2K+1$ substrings $\mathbf{s} \in \{\mathbf{t}[i, l_p - K], \ldots, \mathbf{t}[i, l_p + K]\}$.
- LEV SEARCH elaborates the received data to search for all the possible occurrences of **p** and to validate the found occurrences that satisfy the priority scheme described in Section “An Online Search Algorithm Based on Approximate String Matching”, discarding undesired shadow hits.

After the pattern **p** is loaded into the proper registers, the elaboration starts as the continuous stream of data **t** flows into the module. In Fig. 4(b), a time diagram of the main signals (port-level and internal) is shown.

Distance Computation: *LEV CALC*

The systolic nature of the DP algorithm used to compute the Levenshtein distance is highly parallelizable and particularly suitable for direct hardware implementation. A systolic architecture [90] consists of identical processing elements arranged in an array that process data synchronously, executing a short invariant sequence of instructions without an intervening memory to store and exchange results across the pipelined array. The advantage of this architecture is

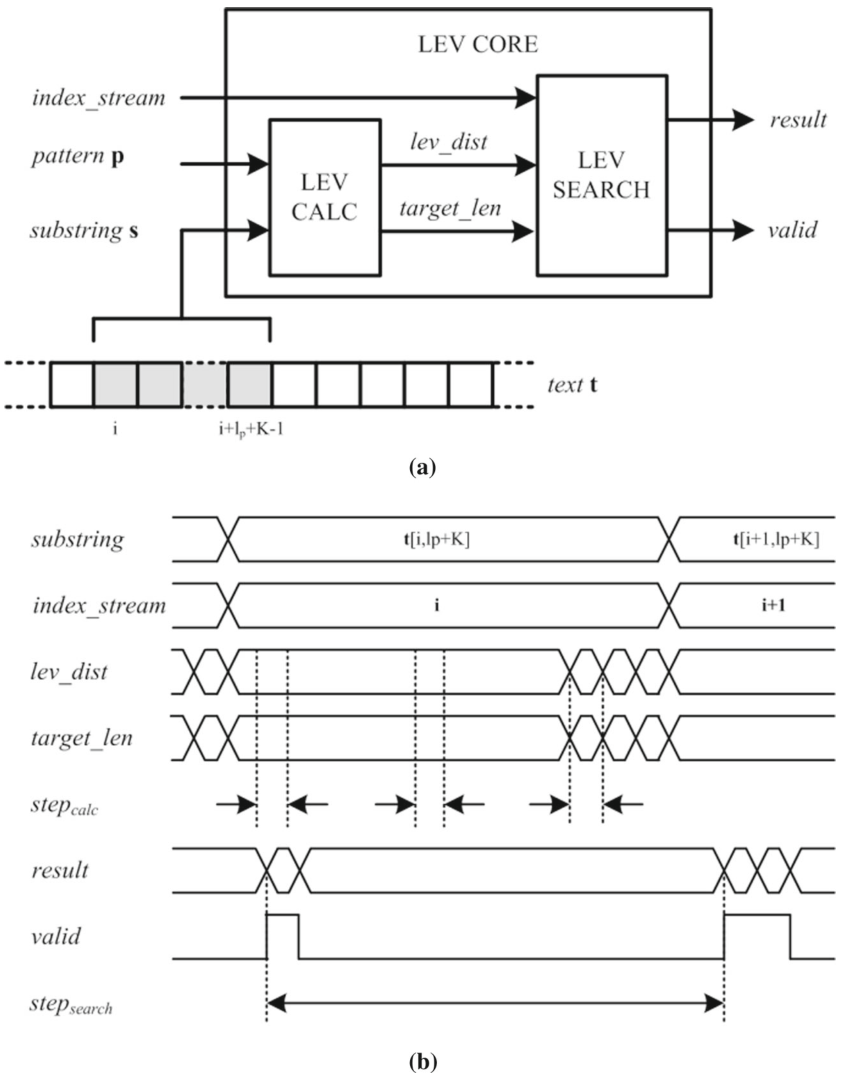


Fig. 4 Conceptual block scheme for LEV CORE module (**a**) and relative timing for the main signals (**b**)

to make both the space and time of the calculation of the Levenshtein matrix linear by parallelizing independent processes. In the ordinary systolic architectures used in string matching applications [91], two strings are shifted on each other to compare step-by-step each pair of symbols. To compute the distance, each symbol needs to be associated to an additional (stored) information, which is the number that identifies the position of the symbol within the stream **t**. The LEV CALC sub-module here introduced implements a counter-based systolic architecture that avoids the storage of a priori known data. It computes the Levenshtein distances in a wave-front fashion returning all the elements of the current anti-diagonal of **C**, filling the matrix in $2l_p + K - 1$ steps, each one from now on referred to as step$_{\text{calc}}$ (Fig. 4(b)). Figure 5 illustrates the conceptual architecture of two consecutive processing elements of the systolic array, on which the LEV CALC sub-module is based on.

The generic jth processing element contains both sequential and combinational logic. The register **p_reg**(j) is configured with the jth symbol $p^{(j)}$ of the pattern **p** (if unused, filled with special symbol $\$_1$). The register **sh_reg**(j) contains the shifting symbols sh(j) of the substring $\mathbf{s} = \mathbf{t}[i, l_p + K]$ and at each step$_{\text{calc}}$, contains a different symbol of **s**. The special symbol $\$_2$ is used to fill the unused **sh_reg** positions during the shifting. The special symbols $\$_1$ and $\$_2$ belong to the alphabet Σ but cannot be used both in **p** and in **t**. The register **a_reg**(j) stores the result of the jth processing element produced by **l_comb**(j) during the current step$_{\text{calc}}$. The register **a_reg_d**(j) represents a delayed version of **a_reg**(j). The combinational block **l_comb**(j) elaborates both the information contained in the abovementioned registers and the state of the upward counter *cnt* to yield the jth element of the anti-diagonal of the matrix **C**.

The counter is used both to allow the shifting process of the symbols sh(j) through the jth processing block, and to provide the suitable n or m values of Eq. 1 to **l_comb**(j) block when the corresponding processing element represents an element $c_{n,0}$ or $c_{0,m}$ of **C** respectively. The jth element of the anti-diagonal in the current step$_{\text{calc}}$ implementing a rearranged, yet equivalent, version of Eq. 1 is computed as:

$$\mathbf{l\_comb}(j) = \min \begin{bmatrix} c_{\text{left}}(j), \\ c_{\text{upper}}(j), \\ c_{\text{upper-left}}(j) \end{bmatrix} + \phi(j), \tag{4}$$

where

$$c_{\text{left}}(j) = \begin{cases} \mathbf{a\_reg\_d}(j), & \text{if } j < cnt \\ cnt, & \text{otherwise} \end{cases}$$

$$c_{\text{upper}}(j) = \begin{cases} cnt, & \text{if } j = 0 \\ \mathbf{a\_reg}(j-1), & \text{if } j > 0 \end{cases}$$

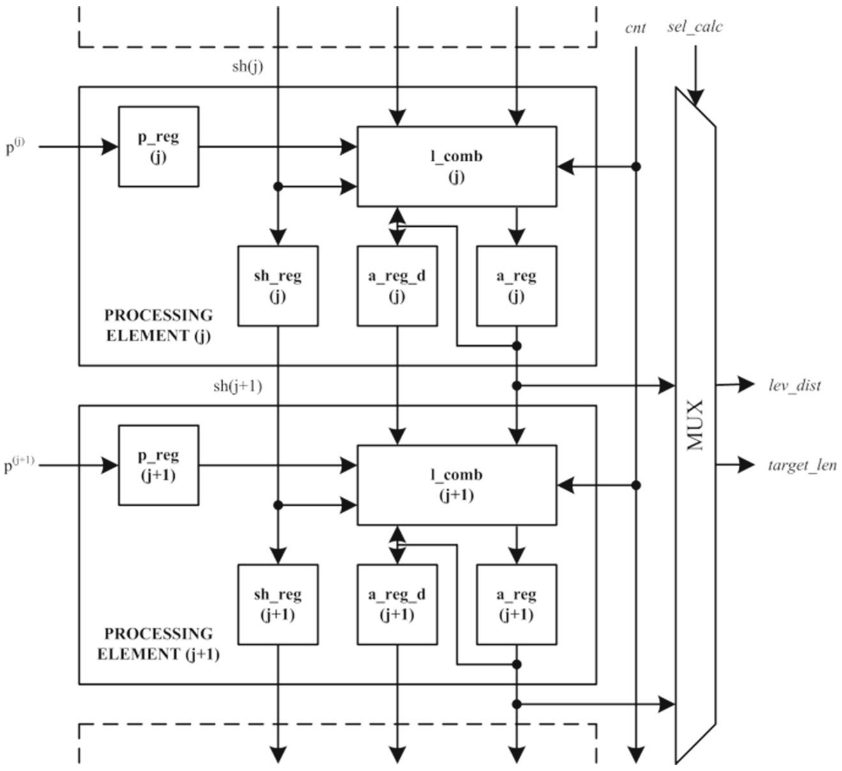


Fig. 5 Details of the systolic architecture of LEV CALC module

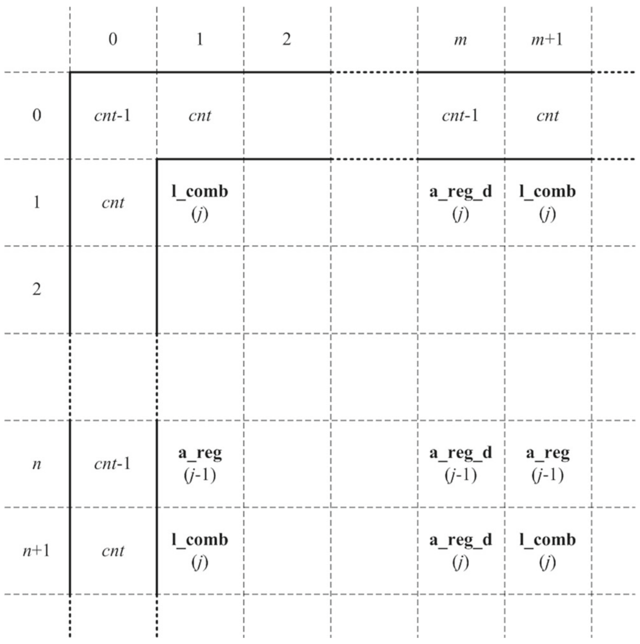


Fig. 6 Representation of the neighbors in the C matrix to compute **l_comb**(j)

$$c_{\text{upper-left}}(j) = \begin{cases} \mathbf{a\_reg\_d}(j-1), & \text{if } j < cnt \\ cnt - 1, & \text{otherwise} \end{cases}$$

$$\phi(j) = \begin{cases} 1, \text{ if } c_{\text{upper-left}}(j) > \min \begin{bmatrix} c_{\text{left}}(j), \\ c_{\text{upper}}(j), \\ c_{\text{upper-left}}(j) \end{bmatrix} \\ \mathbf{p\_reg}(j) \neq \mathbf{sh\_reg}(j), \text{ otherwise.} \end{cases}$$

In Fig. 6, the four possible combinations of neighbors for the computation of **l_comb**(j) are located in the **C** matrix frame to show how Eq. 1 turns into Eq. 4.

Finally, the MUX selector *sel_calc* is constant and uniquely determined with the length of the pattern **p** and the value of the threshold K. Figure 7 shows the content of **p_reg** and **sh_reg** in the inherent shifting mechanism for the example discussed in Section "An Overview on the Levenshtein Distance."

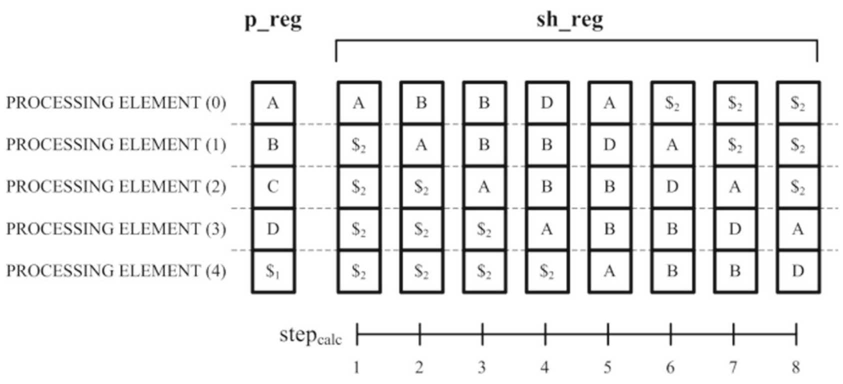
Search and Validation: *LEV SEARCH*

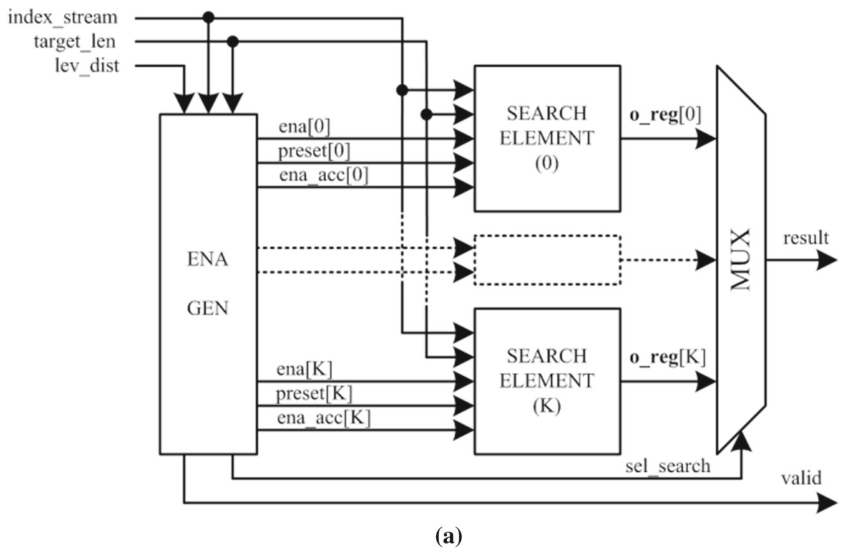
The outputs generated by LEV CALC, together with *index_stream*, represent the inputs of the LEV SEARCH sub-module whose architecture is schematized in Fig. 8(a). This module implements the online search algorithm described in Section "An Online Search Algorithm Based on Approximate String Matching" and formalized in Algorithm 1.

In an infinite loop cycling on i (corresponding to a new *index_stream*), the Levenshtein distances between **p** and all the possible $2K + 1$ substrings $\mathbf{s} \in \{\mathbf{t}[i, l_p - K], \ldots, t[i, l_p + K]\}$ returned by LEV CALC, which satisfy both the described priority rules (R1, R2, and R3) and validation process, are stored in $\mathcal{S}$. The logic relative to the various checks performed by the algorithm is located into the ENA GEN sub-block. There, $[i, l, k]$ = [*index_stream*, *target_len*, *lev_dist*] are processed by the priority rules (see Section "An Online Search Algorithm Based on Approximate String Matching"), and Eq. 3 is evaluated on the values of the counters $r(k)$ during the validation process. Within the same block, the variables *idx*, *ins*, and *acc* defined in Algorithm 1 are implemented as a status register, a flag, and an accumulation register, respectively, and are used to generate the signals ena, preset, ena_acc, and valid. Figure 8(b) depicts a detail of the sequential section which represents the state of the algorithm (SEARCH ELEMENTs). The two-dimensional array **o_reg** of size $(K + 1) \times 3$ implements the storage variable **mem** in Algorithm 1 and it allows to retrieve information relative to the current eligible occurrences. The procedure executed in LEV CALC consists in a constant number of step$_{\text{calc}}$, whereas the algorithm implemented in LEV SEARCH may have a different duration depending on the validation process that is executed only if eligible occurrences are found. A LEV SEARCH cycle is called here step$_{\text{search}}$ and its duration coincides with the number of step$_{\text{calc}}$ necessary to complete one LEV CALC cycle multiplied by its duration:

$$T_{\text{step}_{\text{search}}} = (2 \cdot l_p + K - 1) T_{\text{step}_{\text{calc}}}. \tag{5}$$

Fig. 7 Evolution of the content of **sh_reg** step by step to implement the string shifting





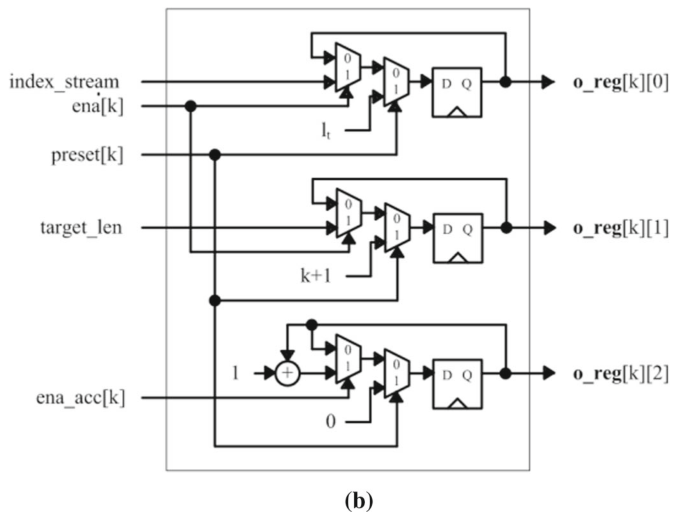


Fig. 8 Conceptual scheme of the LEV SEARCH architecture (**a**) and detail of the kth SEARCH ELEMENT sub-block (**b**)

The proposed hardware architecture executes one step$_{\text{calc}}$ in one clock cycle (T_{clk}), so the LEV CALC block generates the first *lev_dist* sample after $2l_p - K - 1$ clock cycles and in the same cycle the LEV SEARCH block starts processing it:

$$\text{lat}_{\text{calc}} = 2l_p + K - 1$$

$$\text{lat}_{\text{search}} = \begin{cases} K + 1 & \text{without validated occurrences} \\ 2K + 1 & \text{with validated occurrences} \end{cases}$$

Although the latency of the LEV SEARCH is data dependent, the operations of the two modules can be pipelined. The total execution time depends on the latency of the two blocks:

$$T_{\text{exec}} \approx T_{\text{step}_{\text{search}}} l_t = (2l_p + K - 1) \cdot T_{\text{clk}} l_t \qquad (6)$$

where T_{clk} is the clock period. It should be noticed that Eq. 6 is valid when $K < 2(l_p - 1)$, which is always verified since the threshold K is always lower than l_p. Figure 9 depicts the pipelining process.

Resource Usage and Parallel Search

The LEV CORE has been completely described in VHDL and parameterized in terms of:

- $l_{\text{p\_max}}$: Maximum number of symbols per pattern;
- l_{symb}: Number of bits per symbol;
- K: Maximum threshold.

Table 3 reports synthesis results in terms of logic elements (LE[#]) used, when varying the design parameters. The largest synthesized LEV CORE modules ($l_{\text{symb}} = 16$, $l_{\text{p\_max}} = 32$, $K = 4$) use only 3% of the total available resources on Altera Cyclone IV E FPGA (114800 LE). Figure 10 depicts the results of Table 3, providing a clearer view of the resource usage as the design parameters vary. The planar shape of the surfaces and the lack of intersections show the linear relation between resource usage and design parameters.

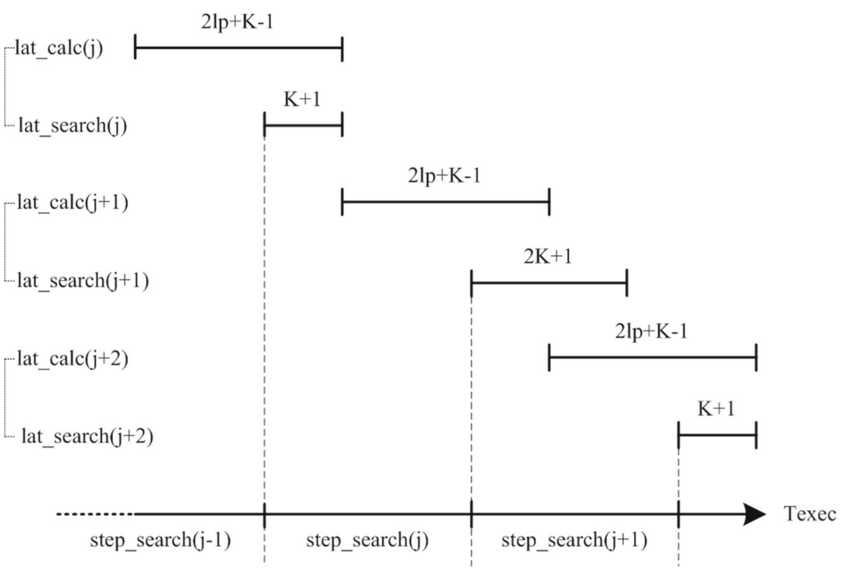


Fig. 9 Time execution diagram in the pipelined structure

Table 3 LE utilization relationship with design parameters

l_{symb}	l_{p_max}	K	LE (nos.)	l_{symb}	l_{p_max}	K	LE (nos.)
4	8	1	557	8	8	1	626
		2	732			2	799
		3	822			3	895
		4	880			4	953
	16	1	1122		16	1	1266
		2	1286			2	1443
		3	1391			3	1536
		4	1472			4	1610
	24	1	1687		24	1	1936
		2	1840			2	2025
		3	1960			3	2151
		4	2064			4	2217
	32	1	2252		32	1	2530
		2	2394			2	2869
		3	2529			3	3002
		4	2656			4	3068
12	8	1	701	16	8	1	765
		2	867			2	940
		3	962			3	1031
		4	1019			4	1090
	16	1	1403		16	1	1543
		2	1579			2	1706
		3	1670			3	1810
		4	1738			4	1865
	24	1	2165		24	1	2349
		2	2233			2	2438
		3	2352			3	2547
		4	2402			4	2616
	32	1	2790		32	1	3067
		2	3163			2	3428
		3	3269			3	3537
		4	3359			4	3616

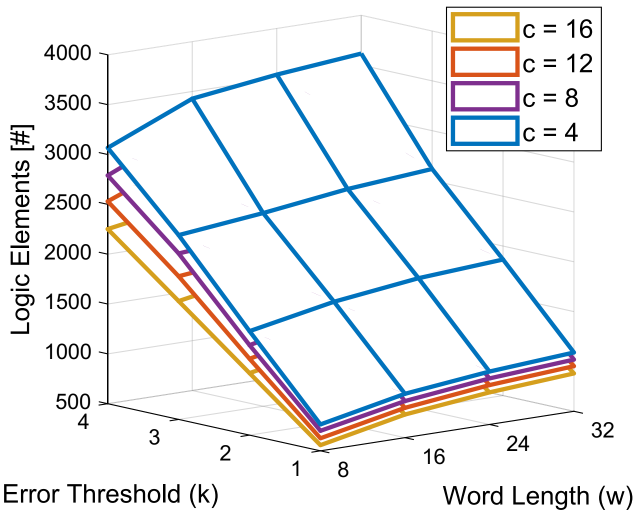


Fig. 10 Linear relationship between design parameters and number of LEs

A parallel search of different patterns over the same text **t** can be done efficiently, by deploying multiple instances of LEV CORE. This is implemented by the multiple OASM (MOASM) system, whose architecture is schematized in Fig. 11. The synthesis of a MOASM system with 40 LEV CORE modules ($l_{\text{symb}} = 8$, $l_{\text{p\_max}} = 16$, $K = 4$) and additional control logic resulted in 62,936 LE, which corresponds to 55% of the resource usage on the same target FPGA.

Experiments

In this section are reported experiments on synthetic data in order to compare the speed performances of the software (SW-OASM) and hardware (HW-OASM) implementation of the proposed OASM algorithm. SW-OASM has been implemented in C++ and executed on a CPU Intel Core i7-4700MQ @ 2.40 GHz. HW-OASM has been described in VHDL-1993 and implemented on the FPGA Altera Cyclone IV E mounted on the board Terasic DE2-115. Two tests are performed using the same setup with randomly generated sequences **t** of $l_t = 3104$ symbols defined over the alphabet $\Sigma = \{A, B, C, D\}$, and randomly generated patterns **p** that are processed by both the implementations, for different values of patterns length l_p and threshold K.

Recent FPGAs commonly host a high-speed link (e.g., PCIe 2.0 with 5 GT/s and the per-lane throughput 500 MB/s) so, to compensate for the lack of a high-speed link on the available FPGA, a simple link emulator for the online input data transfer has been designed. This solution can be considered acceptable since the purpose of the tests is measuring the processing time of the LEV CORE block. Since, two special symbols $\$_1$ and $\$_2$ are added to the alphabet Σ, each symbol in **p** and **t** is coded with $l_{\text{symb}} = 3$ bits. The text **t** is stored into the FPGA in 1 Mbit embedded memory and is sent, symbol by symbol, to the processing logic that operates at 100 MHz (300 Mbit/s). The resulting data flow is stored into another 1 Mbit embedded memory element and is sent to a PC by a low-speed USB link, to be analyzed once the test is completed. Figure 12 schematizes the HW-OASM architecture implemented for the tests, whose details are the following.

- LEV CORE: Processing block with hardwired pattern **p**, maximum pattern length $l_p = 15$, alphabet size = 3 bit (extended due to special symbols), maximum threshold $K = 5$.
- LINK EMULATOR: 1 Mbit ROM, containing the text **t** (5 symbols per memory location) plus control logic. It sends a new symbol to the LEV CORE block at every new incoming request pulse.
- RAM: One 1 Mbit RAM to dump the results (1 Mbit), 16-bit index (sufficient to recognize an occurrence), 3-bit threshold, 5-bit length (max. detected length is $l_p + K = 20$).

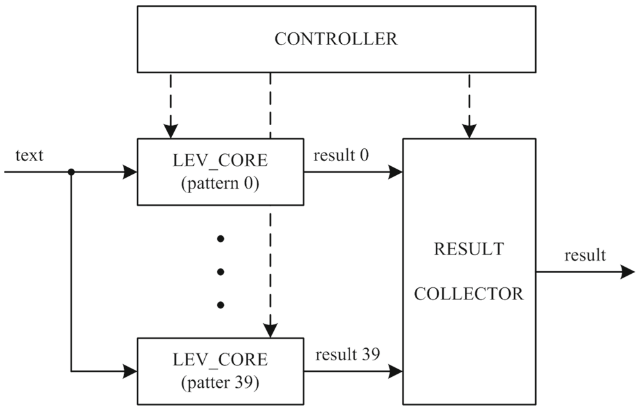


Fig. 11 Conceptual scheme of a MOASM system

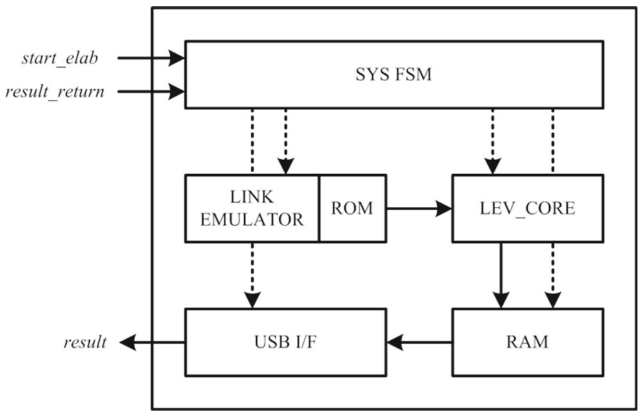


Fig. 12 System implemented for executing the experiments

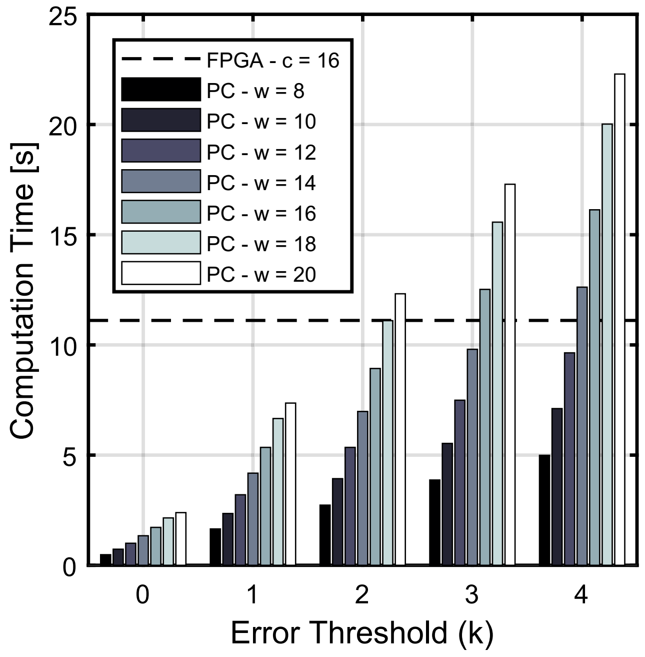


Fig. 13 SW-OASM computation time for different word length (w) and error threshold (k) pairs. HW-OASM computation time (dashed) is constant on the same set of parameters used for SW-OASM

– USB I/F: USB device interface to transfer the dumped results to an external PC.
– SYS FSM: finite state machine to coordinate the I/O operations. Main control signals are *start_elab* (start of elaboration) and *result_return* (command to start returning the data stored in RAM via USB I/F).

The wall-clock time of the SW-OASM execution is not deterministic, since there are usually other processes running in the background that consume computational resources. Therefore, each test run with SW-OASM for a specific configuration has been repeated 100 times with independently randomly generated datasets and the results are reported as mean and standard deviation. Regarding the test runs with HW-OASM, the results prove the effectiveness of the linear deterministic model for the execution time computation described in Eq. 6 with $T_{\mathrm{clk}} = 10$ ns. The experimental results slightly depart from the theoretical ones because of implementation details that prevent $T_{\mathrm{step}_{\mathrm{search}}}$ in Eq. 5 to be just dependent on l_p and K. In Fig. 13, the speed performance is reported, when different thresholds K are used.

Table 4 Speed performance comparison between SW-OASM and HW-OASM implementation for the first test

l_p	SW-OASM (s)	HW-OASM (s)
5	46.51±0.74	0.0039
7	59.991±1.42	0.0051
10	78.973±3.54	0.0069
15	97.004±1.76	0.0099

Table 5 Speed performance comparison between SW-OASM and HW-OASM implementation for the second test

K	SW-OASM (s)	HW-OASM (s)
2	9.731±0.50	0.0096
3	15.179±0.45	0.0099
4	30.858±2.90	0.0102
5	67.953±3.14	0.0105

Two different tests are performed. In the first, whose results are in Table 4, random patterns with lengths $l_p = \{5, 7, 10, 15\}$ are generated and a threshold $K = 3$ is used.

In the second test, with results reported in Table 5, random patterns with fixed length $l_p = 5$ are generated and different thresholds $K = \{2, 3, 4, 5\}$ are used.

Conclusions

A basic cognitive skill always present in human beings, as well in other animal species, is the generalization of patterns at different abstraction levels. Speech recognition, text understanding, and scene analysis are the most notable examples, as the ability to recognize inexact patterns in a sequence of objects or events. Approximate string matching (ASM) is a particular case of subgraph matching and is a fundamental tool in many practical applications, such as bioinformatics, cybersecurity, diagnostic systems, and financial trading. However, when using a tolerance threshold, the ASM can produce shadow hits, i.e., it can identify close multiple hits that, in turn, introduce unwanted uncertainty in the overall template matching procedure and thus in the knowledge discovery task. Of course, the shadow hits could be filtered out offline afterwards in a post-processing step. However, when dealing with online applications, an alternative approach must be followed.

This paper introduces the online ASM (OASM) algorithm to filter out shadow hits online, which is based on a validation procedure that takes advantage of side information provided during Levenshtein distance computations. OASM is designed in plain dynamic programming and is characterized by a high degree of parallelism, which can be leveraged to design an efficient FPGA implementation. The proposed architecture allows performing such a complex retrieval procedure relying on inexpensive hardware, such as the adopted entry-level Altera Cyclone IV E FPGA, where only 3% of available logic elements are used. This envisages further opportunities to increase the parallelism of the whole procedure, by allocating on the FPGA more

LEV CORE units to concurrently search for instances of multiple motifs on the same sequence, or even on different sequences. The execution time of the hardware implementation of such an enhanced implementation (MOASM system) is independent of the number of the instantiated LEV CORE modules, whereas the computational time in the software counterpart scales linearly with the number of pattern searched. Especially when the MOASM system is implemented on high-end FPGA chips that allow the instantiation of many LEV CORE modules, the I/O interface (usually a PCI-Express bus) may easily become congested, due to the possible huge number of results convoyed on a single communication bus. This underlines the importance of the proposed approach to filter out unwanted shadow hits directly on the FPGA, which can greatly reduce the overall I/O throughput. An important future work regards the implementation of a complete MOASM system and the transfer of the Intellectual Property on a high-performance FPGA, such as the Stratix V GX.

Compliance with Ethical Standards

Conflict of Interest All authors declare that they have no conflict of interest.

Ethical Approval This article does not contain any studies with human participants performed by any of the authors.

References

1. Pi Y, Liao W, Liu M, Lu J. Theory of cognitive pattern recognition. In: Pattern recognition techniques, technology and applications. InTech. 2008.
2. Shugen W. Framework of pattern recognition model based on the cognitive psychology. Geo-spatial Inf Sci. 2002;5(2):74–8. https://doi.org/10.1007/BF02833890.
3. Vasamsetti S, Mittal N, Neelapu BC, Sardana HK. 3d local spatio-temporal ternary patterns for moving object detection in complex scenes. Cognitive Computation. 2018;ISSN 1866-9964. https://doi.org/10.1007/s12559-018-9594-5.
4. Li C, Hua T. Human action recognition based on template matching. Procedia Eng. 2011;15:2824–30. https://doi.org/10.1016/j.proeng.2011.08.532. http://www.sciencedirect.com/science/article/pii/S1877705811020339. CEIS 2011. ISSN 1877-7058.
5. Abe Y, Fujita K, Kashimori Y. Visual and category representations shaped by the interaction between inferior temporal and prefrontal cortices. Cogn Comput. 2018;10(5):687–702. https://doi.org/10.1007/s12559-018-9570-0. ISSN 1866-9964.
6. Ragusa E, Gastaldo P, Zunino R, Cambria E. Learning with similarity functions: a tensor-based framework. Cognitive Computation. 2018;ISSN 1866-9964. https://doi.org/10.1007/s12559-018-9590-9.
7. Pang J, Zhao Y, Xu J, Gu Y, Yu G. Super-graph classification based on composite subgraph features and extreme learning machine. Cogn Comput. 2018;10(6):922–36. https://doi.org/10.1007/s12559-018-9601-x. ISSN 1866-9964.
8. Justo R, Alcaide JM, Torres MI, Walker M. Detection of sarcasm and nastiness: new resources for spanish language. Cogn Comput. 2018;10(6):1135–51. https://doi.org/10.1007/s12559-018-9578-5. ISSN 1866-9964.
9. Yang H-C, Lee C-H, Wu C-Y. Sentiment discovery of social messages using self-organizing maps. Cogn Comput. 2018;10(6):1152–66. https://doi.org/10.1007/s12559-018-9576-7.
10. Lauren P, Qu G, Yang J, Watta P, Huang G-B, Lendasse A. Generating word embeddings from an extreme learning machine for sentiment analysis and sequence labeling tasks. Cogn Comput. 2018;10(4):625–38. https://doi.org/10.1007/s12559-018-9548-y. ISSN 1866-9964.
11. Ma Y, Peng H, Khan T, Cambria E, Hussain A. Sentic lstm: a hybrid network for targeted aspect-based sentiment analysis. Cogn Comput. 2018;10(4):639–50. https://doi.org/10.1007/s12559-018-9549-x. ISSN 1866-9964.
12. Navarro G. A guided tour to approximate string matching. ACM Comput Surv (CSUR). 2001;33(1):31–88.
13. Livi L, Rizzi A. The graph matching problem. Pattern Anal Applic. 2013;16(3):253–83.
14. Tran H-N, Cambria E, Hussain A. Towards gpu-based commonsense reasoning: using fast subgraph matching. Cogn Comput. 2016;8(6):1074–86. https://doi.org/10.1007/s12559-016-9418-4. ISSN 1866-9964.
15. Buhler J, Tompa M. Finding motifs using random projections. J Comput Biol. 2002;9(2):225–42.
16. Eskin E, Pevzner PA. Finding composite regulatory patterns in dna sequences. Bioinformatics. 2002;18(suppl 1):S354–63.
17. Pavesi G, Mereghetti P, Mauri G, Pesole G. Weeder web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. Nucleic Acids Res. 2004;32(suppl 2):W199–203.
18. Sinha S, Tompa M. Ymf: a program for discovery of novel transcription factor binding sites by statistical overrepresentation. Nucleic Acids Res. 2003;31(13):3586–88.
19. Typke R, Wiering F, Veltkamp RC, et al. A survey of music information retrieval systems. In: ISMIR. 2005. p. 153–60. 2005.
20. Bertini M, Del Bimbo A, Nunziati W. Video clip matching using mpeg-7 descriptors and edit distance. In: Image and video retrieval. Springer; 2006. p. 133–42. 2006.
21. Ziviani N, De Moura ES, Navarro G, Baeza-Yates R. Compression: a key for next-generation text retrieval systems. Computer. 2000;33(11):37–44.
22. Boukharouba A, Bennia A. Recognition of handwritten arabic literal amounts using a hybrid approach. Cogn Comput. 2011;3(2):382–93. https://doi.org/10.1007/s12559-010-9088-6. ISSN 1866-9964.
23. Sahi M, Gupta V. A novel technique for detecting plagiarism in documents exploiting information sources. Cogn Comput. 2017;9(6):852–67. https://doi.org/10.1007/s12559-017-9502-4. ISSN 1866-9964.
24. Gravano L, Ipeirotis PG, Koudas N, Srivastava D. Text joins in an rdbms for web data integration. In: Proceedings of the 12th international conference on World Wide Web. ACM; 2003. p. 90–101.
25. Maiorino E, Possemato F, Modugno V, Rizzi A. Noise sensitivity of an information granules filtering procedure by genetic optimization for inexact sequential pattern mining. In: Computational intelligence. Springer; 2016. p. 131–50.
26. Yao Y-Y. The rise of granular computing. Journal of Chongqing University of Posts and Telecommunications (Natural Science Edition). 2008;20(3):299–308.
27. Howard N, Lieberman H. Brainspace: relating neuroscience to knowledge about everyday life. Cogn Comput. 2014;6(1):35–44.
28. Bargiela A, Pedrycz W. Granular computing. In: Handbook on computational intelligence: volume 1: fuzzy logic, systems, artificial neural networks, and learning systems. World Scientific; 2016. p. 43–66.

29. Yao Y. A triarchic theory of granular computing. Granular Comput. 2016;1(2):145–57.
30. Singh PK. Similar vague concepts selection using their euclidean distance at different granulation. Cogn Comput. 2018;10(2):228–41.
31. Lin TY, Yao YY, Zadeh LA. Data mining, rough sets and granular computing, volume 95 Physica. 2013.
32. Bianchi FM, Livi L, Rizzi A, Sadeghian A. A granular computing approach to the design of optimized graph classification systems. Soft Comput. 2014;18(2):393–412.
33. Rizzi A, Del Vescovo G, Livi L, Mascioli FMF. A new granular computing approach for sequences representation and classification. In: The 2012 International joint conference on neural networks (IJCNN). IEEE; 2012. p. 1–8.
34. Bianchi FM, Scardapane S, Rizzi A, Uncini A, Sadeghian A. Granular computing techniques for classification and semantic characterization of structured data. Cogn Comput. 2016;8(3):442–61. https://doi.org/10.1007/s12559-015-9369-1. ISSN 1866-9964.
35. Martino A, Giuliani A, Rizzi A. Granular computing techniques for bioinformatics pattern recognition problems in non-metric spaces. Computational intelligence for pattern recognition. In: Pedrycz W and Chen S-M, editors. Cham: Springer International Publishing; 2018. p. 53–81, https://doi.org/10.1007/978-3-319-89629-8_3. ISBN 978-3-319-89629-8.
36. Andoni A, Krauthgamer R, Onak K. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In: 2010 51st Annual IEEE symposium on foundations of computer science (FOCS). IEEE; 2010. p. 377–86.
37. Boytsov L. Indexing methods for approximate dictionary searching: comparative analysis. J Exper Algor (JEA). 2011;16:1–1.
38. Di Pietro R, Mancini LV. Intrusion detection systems, volume 38. Springer Science & Business Media; 2008.
39. Heid CA, Stevens J, Livak KJ, Mickey Williams P. Real time quantitative pcr. Genome Res. 1996;6(10):986–94.
40. MJ Espy JR, Uhl LM, Sloan SP, Buckwalter MF, Jones EA, Vetter JDC, Yao NL, Wengenack JE, Rosenblatt FR, et al. 3 Cockerill Real-time pcr in clinical microbiology: applications for routine laboratory testing. Clin Microbiol Rev. 2006;19(1):165–256.
41. Madel M-B, Niederstätter H, Parson W. Trixy—homogeneous genetic sexing of highly degraded forensic samples including hair shafts. Forens Sci Int: Gen. 2016;25:166–74. https://doi.org/10.1016/j.fsigen.2016.09.001. ISSN 1872-4973.
42. Niederstätter H, Coble MD, Parsons TJ, Parson W. Characterization of mtdna snp typing using quantitative real-time pcr with special emphasis on heteroplasmy detection and mixture ratio assessment. Int Congress Series. 2006;1288:1–3. https://doi.org/10.1016/j.ics.2005.09.021. ISSN 0531-5131. Progress in Forensic Genetics 11.
43. Rasool A, Khare N. Parallelization of kmp string matching algorithm on different simd architectures: multi-core and gpgpu's. Int J Comput Appl. 2012;49(11):26–8.
44. Zhong C, Chen G-L. A fast determinate string matching algorithm for the network intrusion detection systems. In: 2007 International conference on machine learning and cybernetics, volume 6. IEEE; 2007. p. 3173–77.
45. Crochemore M, Iliopoulos CS, Pinzon YJ, Reid JF. A fast and practical bit-vector algorithm for the longest common subsequence problem. Inf Process Lett. 2001;80(6):279–85.
46. Leighton FT. Introduction to parallel algorithms and architectures: arrays · trees · hypercubes. Elsevier. 2014.
47. Michailidis PD, Margaritis KG. A programmable array processor architecture for flexible approximate string matching algorithms. In: International conference workshops on parallel processing, 2005. ICPP 2005 Workshops. IEEE; 2005. p. 201–9.
48. Antonik P, Haelterman M, Massar S. Online training for high-performance analogue readout layers in photonic reservoir computers. Cogn Comput. 2017;9(3):297–06. https://doi.org/10.1007/s12559-017-9459-3. ISSN 1866-9964.
49. Vásquez JL, Pérez ST, Travieso CM, Alonso JB. Meteorological prediction implemented on field-programmable gate array. Cogn Comput. 2013;5(4):551–557. https://doi.org/10.1007/s12559-012-9158-z. ISSN 1866-9964.
50. Mikami S, Kawanaka Y, Wakabayashi S, Nagayama S. Efficient fpga-based hardware algorithms for approximate string matching. ITC-CSCC. 2008;2008:201–4.
51. Mitani Y, Ino F, Hagihara K. Parallelizing exact and approximate string matching via inclusive scan on a gpu. IEEE Trans Parallel Distrib Syst. 2017;28(7):1989–2002. https://doi.org/10.1109/TPDS.2016.2645222. ISSN 1045-9219.
52. Xu K, Cui W, Hu Y, Guo L. Bit-parallel multiple approximate string matching based on gpu. Procedia Comput Sci. 2013;17:523–9. https://doi.org/10.1016/j.procs.2013.05.067. First International Conference on Information Technology and Quantitative Management.
53. Nunes LSN, Bordim JL, Nakano K, Ito Y. A fast approximate string matching algorithm on gpu. In: 2015 Third international symposium on computing and networking (CANDAR); 2015. p. 188–92. https://doi.org/10.1109/CANDAR.2015.29.
54. Ho T, Oh S-R, Kim HJ. A parallel approximate string matching under levenshtein distance on graphics processing units using warp-shuffle operations. PLOS ONE. 2017;12(10):1–15, 10. https://doi.org/10.1371/journal.pone.0186251.
55. Van Court T, Herbordt MC. Families of fpga-based algorithms for approximate string matching. In: 15th IEEE International conference on application-specific systems, architectures and processors, 2004. Proceedings. IEEE; 2004. p. 354–64.
56. Smith TF, Waterman MS. Identification of common molecular subsequences. J Mol Biol. 1981;147(1):195–7. https://doi.org/10.1016/0022-2836(81)90087-5. ISSN 0022-2836.
57. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Molec Biol. 1970;48(3):443–53.
58. Myers G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. J ACM (JACM). 1999;46(3):395–415.
59. Wagner RA, Fischer MJ. The string-to-string correction problem. J ACM (JACM). 1974;21(1):168–73.
60. Yu CW, Kwong KH, Lee K-H, Leong PHW. A smith-waterman systolic cell. In: New algorithms, architectures and applications for reconfigurable computing. Springer; 2005. p. 291–300.
61. Dydel S, Bała P. Large scale protein sequence alignment using fpga reprogrammable logic devices. In: Field programmable logic and application. Springer; 2004. p. 23–32.
62. Sirasao A, Delaye E, Sunkavalli R, Neuendorffer S. Fpga based opencl acceleration of genome sequencing software. System. 2015;128(8.7):11.
63. Herbordt MC, Gu Y, Sukhwani B, VanCourt T. Single pass, blast-like, approximate string matching on fpgas. In: 14th Annual IEEE symposium on field-programmable custom computing machines, 2006. FCCM'06. IEEE; 2006. p. 217–26.
64. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J Molec Biol. 1990;215(3):403–10. https://doi.org/10.1016/S0022-2836(05)80360-2. http://www.sciencedirect.com/science/article/pii/S0022283605803602.
65. West B, Chamberlain RD, Indeck RS, Zhang Q. An fpga-based search engine for unstructured database. In: Proc. of 2nd workshop on application specific processors, vol. 12, p. 25–32. 2003.
66. Hoffmann J, Zeckzer D, Bogdan M. Using fpgas to accelerate myers bit-vector algorithm. XIV Mediterranean conference on

medical and biological engineering and computing 2016. In: Kyriacou E, Christofides S, and Pattichis CS, editors. Cham: Springer International Publishing; 2016. p. 535–541. ISBN 978-3-319-32703-7.
67. Blüthgen H-M, Noll TG. A programmable processor for approximate string matching with high throughput rate. In: IEEE International conference on application-specific systems, architectures, and processors, 2000. Proceedings. IEEE; 2000. p. 309–16.
68. Utan Y, Wakabayashi SI, Nagayama S. An fpga-based text search engine for approximate regular expression matching. In: 2010 International conference on field-programmable technology (FPT). IEEE; 2010. p. 184–91.
69. Park JH, George KM. Parallel string matching algorithms based on dataflow. In: Proceedings of the 32nd Annual Hawaii international conference on systems sciences, 1999. HICSS-32. IEEE; 1999. p. 10–pp.
70. Ou C-M, Yeh C-Y, Su Y-L, Hwang W-J, Chen J-F. Fpga implementation of content-based music retrieval systems. In: International conference on embedded software and systems symposia, 2008. ICESS Symposia'08. IEEE; 2008. p. 96–103.
71. Smith MJS. Application-specific integrated circuits. Addison-Wesley Professional. 2008.
72. Brown S. Fpga architectural research: a survey. Des Test Comput IEEE. 1996;13(4):9–15.
73. Bondalapati K, Prasanna VK. Reconfigurable computing systems. Proc IEEE. 2002;90(7):1201–17.
74. Kawanaka Y, Wakabayashi S, Nagayama S. A systolic regular expression pattern matching engine and its application to network intrusion detection. In: FPT, p. 297–300. 2008.
75. Levenstein V. Binary codes capable of correcting spurious insertions and deletions of ones. Probl Inf Transm. 1965;1(1):8–17.
76. Sellers PH. The theory and computation of evolutionary distances: pattern recognition. J Algor. 1980;1(4):359–73. https://doi.org/10.1016/0196-6774(80)90016-4. ISSN 0196-6774.
77. Ukkonen E. Algorithms for approximate string matching. Inf Control. 1985;64(1):100–18. https://doi.org/10.1016/S0019-9958(85)80046-2. ISSN 0019-9958. International Conference on Foundations of Computation Theory.
78. Matsui T, Uno T, Umemori J, Koide T. A new approach to string pattern mining with approximate match. In: Discovery science. Springer; 2013. p. 110–25.
79. Lee Y, Jeon K, Lee J-T, Kim S, Narry Kim V. Microrna maturation: stepwise processing and subcellular localization. EMBO J. 2002;21(17):4663–70.
80. Winter J, Jung S, Keller S, Gregory RI, Diederichs Sn. Many roads to maturity: microrna biogenesis pathways and their regulation. Nat Cell Biol. 2009;11(3):228.
81. Borchert GM, Lanier W, Davidson BL. Rna polymerase iii transcribes human micrornas. Nate Struct Molec Biol. 2006;13(12):1097.
82. Lee Y, Kim M, Han J, Yeom K-H, Lee S, Baek SH, Narry Kim V. Microrna genes are transcribed by rna polymerase ii. EMBO J. 2004;23(20):4051–60.
83. Kim B, Jeong K, Narry Kim V. Genome-wide mapping of drosha cleavage sites on primary micrornas and noncanonical substrates. Molec cell. 2017;66(2):258–69.
84. Park J-E, Heo I, Tian Y, Simanshu DK, Chang H, Jee D, Patel DJ, Narry Kim V. Dicer recognizes the 5' end of rna for efficient and accurate processing. Nature. 2011;475(7355):201.
85. Landthaler M, Yalcin A, Tuschl T. The human digeorge syndrome critical region gene 8 and its d. melanogaster homolog are required for mirna biogenesis. Curr Biol. 2004;14(23):2162–67.
86. Thomson JM, Newman M, Parker JS, Morin-Kensicki EM, Wright T, Hammond SM. Extensive post-transcriptional regulation of micrornas and its implications for cancer. Genes & Devel. 2006;20(16):2202–7.
87. Hasan SS, Ahmed F, Khan RS. Approximate string matching algorithms: a brief survey and comparison. Int J Comput Appl, 120 (8). 2015.
88. Kozomara A, Griffiths-Jones S. mirbase: annotating high confidence micrornas using deep sequencing data. Nucleic Acids Res. 2013;42(D1):D68–73.
89. Kent JW, Sugnet CW, Furey TS, Roskin KM, Pringle TH, Zahler AM, Haussler D. The human genome browser at ucsc. Gen Res. 2002;12(6):996–06.
90. Kung HT, Leiserson CE. Algorithms for vlsi processor arrays. Introduct VLSI Syst, 271–92. 1980.
91. Lipton RJ, Lopresti D. A systolic array for rapid string comparison. In: Proceedings of the Chapel Hill conference on VLSI, p. 363–76. 1985.