Soft Computing (2024) 28:3465–3477
https://doi.org/10.1007/s00500-023-08687-8

APPLICATION OF SOFT COMPUTING

GPU-based similarity metrics computation and machine learning approaches for string similarity evaluation in large datasets

Aurel Baloi[1,2] · Bogdan Belean[3] · Flaviu Turcu[3] · Daniel Peptenatu[1,2]

Accepted: 26 May 2023 / Published online: 14 June 2023

Abstract

The digital era brings up on one hand massive amounts of available data and on the other hand the need of parallel computing architectures for efficient data processing. String similarity evaluation is a processing task applied on large data volumes, commonly performed by various applications such as search engines, biomedical data analysis and even software tools for defending against viruses, spyware, or spam. String similarities are also used in musical industry for matching playlist records with repertory records composed of song titles, performer artists and producers names, aiming to assure copyright protection of mass-media broadcast materials. The present paper proposes a novel GPU-based approach for parallel implementation of the Jaro–Winkler string similarity metric computation, broadly used for matching strings over large datasets. The proposed implementation is applied in musical industry for matching playlist with over 100k records with a given repertory which includes a collection of over 1 million right owner records. The global GPU RAM memory is used to store multiple string lines representing repertory records, whereas single playlist string comparisons with the raw data are performed using the maximum number of available GPU threads and the stride operations. Further on, the accuracy of the Jaro–Winkler approach for the string matching procedure is increased using both an adaptive neural network approach guided by a novelty detection classifier (aNN) and a multiple-features neural network implementation (MF-NN). Thus, the aNN approach yielded an accuracy of 92% while the MF-NN approach achieved an accuracy of 99% at the cost of increased computational complexity. Timing considerations and the computational complexity are detailed for the proposed approaches compared with both the general-purpose processor (CPU) implementation and the state-of-the-art GPU approaches. A speed-up factor of 21.6 was obtained for the GPU-based Jaro–Winkler implementation compared with the CPU one, whereas a factor of 3.72 was obtained compared with the existing GPU implementation of string matching procedure based on Levenstein distance metrics.

Keywords String similarity score · String matching · Parallel computation · GPU · CUDA kernel

1 Introduction

The amount of digital data produced and transmitted around the world is increasing at an exponential rate. Consequently, the data processing tasks are more and more complex, leading to the need of hardware resources with increased parallel computing capabilities. As referred to the current technologies used for intensive processing tasks, the Moore law did not survive, meaning that in the past years, the number of decisional elements in a compact integrated circuit was more than doubled every 18 months. A few years ago,

Aurel Baloi and Bogdan Belean contributed equally to this work.

✉ Flaviu Turcu
flaviu.turcu@itim-cj.ro

✉ Daniel Peptenatu
daniel.peptenatu@geo.unibuc.ro

Aurel Baloi
aurel.baloi@ingr.ro

Bogdan Belean
bogdan.belean@itim-cj.ro

[1] Research Center for Integrated Analysis and Territorial Management, University of Bucharest, 4-12 Regina Elisabeta, 030018 Bucharest, Romania

[2] Department, Intergraph Computer Services, 22-24 Putul lui Zamfir, 011683 Bucharest, Romania

[3] Center for Research and Advanced Technologies for Alternative Energie, National Institute for Research and Development of Isotopic and Molecular Technologies, 67-103 Donat, 400095 Cluj-Napoca, Romania

Moore underlined potential paths for further technology improvement, such as the use of the speed of light and the atomic nature of materials (Moore 2015). State-of-the-art processing technologies in the context of the big-data paradigm represent the solution for designing effective big-data processing systems (Chen 2023). Field-programmable gate arrays (FPGAs), graphic processing units (GPUs), and multi-core processor architectures are recognized as the key technologies. Currently, GPUs take the lead, overcoming the disadvantages of the FPGAs and last generation CPUs. The main drawbacks were the difficulty of programming and the increased time to market in case of the FPGAs and the reduced number of processing threads (i.e., two orders of magnitude) of the CPUs as compared with the GPUs. All these powerful computing resources are used to tackle the main problems brought up by the big-data paradigm.

1.1 String similarities in large datasets

A key issue, considering the available data on a tremendous variety of subjects, is to find the occurrences of a given string or pattern within large datasets. The aforementioned task, also known as string similarity evaluation or approximate string matching, is considered to be the heart of application from various fields such as network intrusion detection systems, stock market estimation, web searching and even computational biology (Navarro 2001; Baúto et al. 2018; Binoue et al. 2012). The importance of this task is underlined as follows. In terms of network security, the undesired traffic cannot be filtered by the header information because most threats target the application layer (Villa et al. 2008; Ongliang and Xu 2018). Deep packet inspection and payload checking are needed to assure protection against viruses, malware, spyware, spam and deliberate incursion efforts, making this task increasingly difficult, leading to the design of security filters with lower processing time using GPUs and FPGAs (Samuel et al. 2021). Moreover, big-data and distributed machine learning approaches are used for scalable malware detection system (Kumar 2022). Bioinformatics applications, where a considerable number of reads must be mapped (or aligned) to a longer reference or database sequence, also make use of string similarities evaluation (Ayad et al. 2016; Maayah et al. 2022). Several exact string-matching tools are available, one of the most successful being the burrows wheeler transform, which also benefits from efficient implementations using GPUs (Salavert-Torres 2012). Moreover, to allow the presence of some mismatches and structural variations, approximate string matching is also performed in bioinformatics applications (Mitani et al. 2017). Considering both exact and approximate string-matching approaches applied on large scale data of RNA/DNA sequences, GPUs were successfully used for efficient implementation in terms of processing time and throughput (Zou et al. 2015). Thus, the Smith–Waterman dynamic programming algorithm is typically accelerated by GPU-based approaches for sequence alignment because of its regularity. However, it becomes impractical when comparing long length sequences due to its quadratic complexity; hence, heuristic methods are needed to condense the search field. In Perez-Wohlfeil et al. (2023), a CUDA version of the GECKO algorithm for sequential seed-and-extend sequence comparison is proposed to deal with long sequence alignments. Another type of applications where algorithms for the determination of string similarities were successfully applied is stock market estimation (Baúto et al. 2018). In this case symbolic aggregate approximation was used as a pattern recognition technique for identifying market and/or stock trends on increased size financial time series. Once the importance of the string similarity evaluation is underlined, we further on proceed to the description of the state-of-the-art algorithms used for this purpose.

String matching algorithms have been explored extensively utilizing a variety of methodologies, including dynamic programming, finite state machines, bit parallelism, filtering, and indexing (Yu et al. 2016). The literature survey led to the classification of these algorithms into automata-based algorithms, similarity metrics-based algorithm and artificial intelligence approaches. The automata-based algorithms can be split into deterministic finite automata (DFA) and nondeterministic ones (NDFA) (Mitani et al. 2017). In the first category, Knuth–Morris–Pratt (Wang et al. 2018) and Aho–Corasick (AC) (Najam-ul Islam et al. 2022) algorithms have been widely used for exact string matching. Meanwhile, the typical nondeterministic approaches namely the bit-parallel algorithms are used for both approximate and exact string matching (Kusudo et al. 2015; Sadiq and Yousaf 2023). The number of memory references is defined only by the text and pattern lengths in the bit-parallel algorithms. To put it another way, the bit-parallel approach has the same best- and worst-case time complexity, allowing it to deliver very reliable throughput while dealing with a variety of patterns with different text and pattern contents. Taking advantage of this bit parallelism approach, the number of operations to be performed is reduced by a factor of size w, corresponding to the computer word size. Moreover, in addition to bit parallelism, parallel computation strategies using GPUs were implemented to achieve increased computational speed (Lin et al. 2014; Bhat et al. 2022). Considering the DFAs, the parallel failureless Aho–Corasick (PFAC) algorithm (Thambawita et al. 2016) extends the AC algorithm for efficient parallelization on GPUs. Nevertheless, traditional methods, such as the AC algorithm, fail to meet the current need for efficiency (Ongliang and Xu 2018). Performance degradation occurs if the text has many partially matching patterns, and their matching lengths are relatively long. Such unstable search throughput is not desirable in case of large datasets. Another DFA algorithm is the Wu–Manber algorithm (Wu

and Manber 1992) which uses the hash function to eliminate impossible matching. This considerable validation step reduces the algorithm efficiency. Another application of the GPU implementation of the string matching algorithms is proposed in Naderalvojoud and Ozsoy (2021) to improve word embeddings, whereas in Sitaridi and Ross (2016) GPUs are successfully used for efficient implementation of string matching operators common in SQL queries. Considering long strings where computing the edit distance has been hampered by quadratic time complexity with respect to string length, the wavefront alignment algorithm was implemented onto GPU for efficient edit distance computation between strings having hundreds of millions of characters (Castells-Rufas 2023).

String similarity metrics are known for increased efficiency, and they are divided into two categories: string-based, where syntactic similarities are calculated, and language-based, where semantic similarities are evaluated (Budanitsky and Hirst 2006). As opposed to the first category which computes similarities based on characters appearance and sequence, semantic similarities are evaluated based on the likeness of their meaning or semantic content rather than lexicographical similarity (Sun et al. 2015). According to Santos et al. (2018), the Jaro–Winkler algorithm is a heuristic character-based approach which delivers increased accuracy and precision in string similarity evaluation, close to the artificial intelligence approaches. Its 65.17% accuracy and 78% precision when applied on five million pairs of toponyms make the algorithm an important candidate for string similarity evaluation. Regarding GPU-based implementations for string similarity evaluation using specific metrics, the approximate string-matching algorithm under various distances was performed using FPGAs (Cinti et al. 2020), multi-core processors (Watanuki et al. 2013) and GPUs (Tran et al. 2016; Ho et al. 2016). In Ho et al. (2016), the implementation idea is on the use of warp-shuffle operations to eliminate the access of global memory or shared memory. The proposed implementation outperformed the previous parallel approach on GPUs.

The machine learning approaches come into play when computing a similarity score is not necessary enough, for a match/mismatch decision. This means that, besides the string likeness, the context of their use is also important. For example, in case of author-level scientometric indicators, supervised machine learning (ML) approaches are used to disambiguate author names in the Web of Science publication database (Rehs 2021). The main parameters to describe the ML approach are the precision to quantify the number of positive class predictions that belong to the positive class, the recall to quantify the number of positive class predictions made of all positive examples in the dataset and the F-Measure which provides a single score that balances both the concerns of precision and recall in one number. In Rehs (2021), the random forest and logistic regression machine learning algorithms are used to achieve feature assessment as opposed to classic feed-forward neural networks together with increased precision and recall rates. In Santos et al. (2018) randomized trees, support vector machines and random forests were used for string similarity evaluation and increased accuracy was obtained on a large dataset as compared with classic approaches such as Jaro–Winkler and Damerau–Levenshtein approaches. Moreover, in Castellanos et al. (2021) an unsupervised machine learning approach is used for data reduction in string space.

1.2 String similarities in musical industry

In the musical market, inequality still exists in establishing royalty rights for broadcasting the recorded music. The intellectual rights owners (e.g., performers, producers, artists) look for a transparent and accelerated remuneration process in the context of TVs and radios broadcasts. Sometimes the remuneration comes after more than one year from broadcasting. Diving into the context, we simplify the business flow as follows: producers record media content on different supports (e.g., audio and audio-video) which are used by users such as TVs and radios which are obliged to pay a royalty, according to the national legislation and to the existing records within the collective management organizations that represents the intellectual rights owners. The right owner is defined here as a record characterized by the following information: performer/artist, title, and producer. Each record has also a specific timestamp and a broadcasting duration. *The repertory* represents all the right owner records managed by the collective management organization, and it is composed of audio and/or video records together with their corresponding record attributes: performer/artist, title, and producer. On the other hand, *the playlist* is a table of records for all broadcasted audio and/or video recordings of a TV or radio. Based on the received playlists and on the existing repertoire of each collective management organization, the remuneration is established for each right owner. The repertoire is established based on the contractual terms of representation between collective management organizations and intellectual rights owners. Records are described also by the three attributes: title, artist, and producer. The key process in establishing the remuneration is the matching between the user delivered playlists records, described by their attributes (performer/artist, title, and producer) and the repertoire of the collective management organization. This process must be transparently, reliable, and its results have to be efficiently delivered in terms of computational time. The current paper is developed based on an IT system that has been implemented for a collective management organization, that is meant to manage the repertoires, ingest the playlists and determine the matchings and create transparency and a space of dialog

for key stakeholders in the process of intellectual property royalties' distribution. The software application deals with large playlists composed of multiple string records that must be matched with repertory records in order to assure both copyright protection of all broadcast materials and royalty rights identification. The variety of string similarity evaluation algorithms together with state of the art computing capabilities and machine learning algorithms show great potential developing an efficient solution in terms of both accuracy and computational time.

To answer the previous demands, the current paper provides a novel Jaro–Winkler algorithm implementation that takes advantage of the GPU parallel computing capabilities. The Jaro–Winkler approach was chosen for string similarity evaluation considering the precise quantification of string similarities compared with approximate and exact string-matching algorithms and its comparable accuracy performances with ML algorithms (Santos et al. 2018). As underlined in Kusudo et al. (2015), GPU constraints regarding the data type and their fixed length have to be met, in spite of the variable string lengths for which the similarity evaluation needs to be performed on. Thus, the proposed implementation complies with the previous constraints, delivering increased throughput compared with state of the art approaches. Moreover, to improve the accuracy of the aforementioned implementation, two machine learning approaches were proposed. Considering the large datasets they are applied on, the computational complexity is analyzed, whereas a compromise between the algorithms complexity and their string matching accuracy is discussed.

The remainder of this paper is organized as follows. Section 2 introduces the available computing tools of the GPU architecture together with the Jaro–Winkler algorithm description and the proposed implementation for the string similarity evaluation with increased throughput. Moreover, a simple threshold-based algorithm is implemented onto the GPU for matching playlist records with repertory records, whereas machine learning approaches are used to increase the accuracy of the record matching procedure. The results and discussion including the obtained throughput and accuracy as compared with state-of-the-art approaches are described in Sect. 3. Finally, conclusions are drawn in Sect. 4.

2 Materials and methods

The interest for efficient string similarity evaluation especially in case of large datasets was detailed in the previous section. Besides search engines, stock market estimations and computational biology applications, we underlie another task, royalty rights identification, where string similarity evaluation is mandatory. The main objective can be resumed to the matching between the playlist and repertoire records, each of them characterized by three attributes: title, artist and producer. As referred to the size of the datasets, in a particular case, we have to compare 80,000 repertoire records with more than 15,000,000 playlist records on the 3 attributes in order to establish the audio recordings royalties. This leads to multiple playlists per month to be evaluated, each evaluation involving tens of MB of input data for the processing algorithms. Let us also mention that the main challenge posed by the data is that the records are not uniquely identified by a key publicly available, and the playlist is completed by various users, based on different methods, which allow artists name or producers to be reversed or shortcut, whereas titles can be altered or mistype. These premises lead to a challenging matching process, also known in the literature as author name disambiguation (Rehs 2021). String similarity metrics computation commonly involves a threshold which defines the match or mismatch decision for two given strings. However, considering the input data variation, string matching technique accuracy must be improved on one hand, because of the string similarity metrics accuracy limited to 80% and on the other hand because of the data variance problem which also decreases accuracy. Consequently, the proposed approach to address these issues is to firstly use a lower threshold for the string similarity metrics in order to determine preliminary matches between records from the repertoire and playlist records for each author. The resulted set of preliminary matches includes false-positive ones due to the reduced similarity metrics threshold used for string matching procedure. Thus, before using an adaptive neural network for associating each playlist record to a repertory one for each of the authors, a one-class classifier for novelty detection excludes the false-positive matches, defined as the records that do not correspond to any repertory records of the given given author. The proposed methods are meant to increase the accuracy of the Jaro–Winkler string matching procedure, and they are detailed within the following sub-sections.

2.1 String records matching using the Jaro–Winkler similarity metric

We resume further on the challenge of finding the most appropriate approach for string matching which better suits our task. The scientific literature identifies the following units of analysis in matching between two texts: characters and tokens (words, n-grams). These main units are generally used in quantitative approaches for string matching tasks. The following table summarizes the two types of approaches, with the mention that hybrid approaches are often adopted. Character-based approaches for string similarity make use of specific distances between two strings and they are employed in applications where semantic meaning is not as important as the similarity. Considering the token-based similarities, similar results were obtained using Jaccard N-grams and

Table 1 String similarity evaluation

	Accuracy	Precision	Recall
Quantitative approach			
Character-based			
Jaro–Winkler	65.17	78	42.26
Levenshtein	65.07	78.65	41.36
Token-based			
Jaccard N-grams	61.72	71.5	71.5
Monge-Elkan	59.57	65.83	39.79
Qualitative approach			
Machine learning			
SVM	72.38	69.17	80.76
Random forest	78.67	78.03	79.80
Machine learning	85.4		

Monge-Ekan algorithms. The best results in terms of accuracy obtained with Jaro–Winkler (Santos et al. 2018) and Damerau–Levenshtein distances (Lazreg et al. 2020). The Jaro–Winkler distance is a string similarity metric that measures the similarity between two strings. Its computational complexity is generally quadratic, $O(n^2)$, where n is the length of the strings being compared. On the other hand, the Levenshtein distance is a string edit distance metric that calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. The computational complexity of the Levenshtein distance algorithm is $O(n^3)$, where n is the length of the strings. This means that it is less efficient than the Jaro–Winkler distance for long strings. Considering their similar accuracy (Table 1) and the reduced computational complexity of the Jaro–Winkler approach, the last one was chosen for our task, and it is detailed further on.

2.1.1 Jaro–Winkler string similarity algorithm description

Let us consider two strings denoted by $s_1 = a_1 a_2 \cdots a_N$ and $s_2 = b_1 b_2 \cdots b_L$, where a_i and b_j represent the characters included in the s_1 and s_2 string, respectively. A character a_i in s_1 is common for the two strings in case a character $b_j = a_i$ exists in s_2, such that $\mid i - j \mid < min(\mid s_1 \mid, \mid s_2 \mid)/2$. Let $s_1' = a_1' \cdots a_m'$ be the common characters in s_1 and let $s_2' = b_1' \cdots b_m'$ be the common characters in s_2, a transposition of s_1' and s_2' is defined as a position i, such that $a_i \neq b_i$. Considering m the number of the common characters and $t/2$ half the number of transpositions, the Jaro distance is defined as:

$$\text{Jaro}(s_1, s_2) = \begin{cases} 0, & \text{for } m = 0 \\ 3(m/\mid s_1 \mid + m/\mid s_2 \mid + (2m - t)/2m), & \text{otherwise} \end{cases} \quad (1)$$

An improved variant of aforementioned metrics was proposed by Winkler, where the constraint of a common prefix is added to the similarity metrics as denoted by:

$$\text{JW}s(s_1, s_2) = \text{Jaro}(s_1, s_2) + \max(4, l')p(1 - \text{Jaro}(s_1, s_2)), \quad (2)$$

where l' is the longest common prefix of s_1 and s_2 and $p < 0.25$ (commonly set to 0.1).

Let us consider each record i from our database described by the triplet of variables author (a), title (t) and producer (p), denoted by (a, t, p). There are two sets of records called repertoire and playlist, denoted by $R = (a_r, t_r, p_r)$ and $Y = (a_y, t_y, p_y)$, respectively, where R and Y represent their size. We extracted from the repertoire 3 subsets of size R, containing authors, song titles and producers, denoted by A_R, T_R and P_R and another 3 subsets A_V, T_V and P_V of size Y each, from the playlist. The strings included in the set A are evaluated in terms of similarity with all the strings from the set A_V, using the Jaro–Winkler algorithm. Considering a selected threshold k_1, all the entities from the set A_V having a higher JW similarity score than k_1 are selected according to the Eq. (3).

$$\text{JW}s(a, a_v) > k_1 \quad (3)$$

In case of the selected records, the values for the title variable t are further on checked for similarities with the title variable t_V in a similar manner based on the threshold k_2. The pseudocode for the proposed algorithm is described in Fig. 2.

Please note that, in round numbers, the number of selected entities after the k_1 threshold is applied is equal or greater than R. This approximation is used further on for the computational complexity estimation. The computational complexity of our proposed approach, as well as the input data size, has a significant impact on the implementation to be developed. For this reason, we evaluate the computational complexity of our approach, which is given by the order of growth for the total computational cost in case of the Jaro Winkler algorithm $O_J(mn)$, where m is the length of string s_1 and n is the length of string s_2. The number of instances the JW algorithms is applied is rv in case of the authors' similarities evaluation between the two sets A and A_v. Considering string similarities evaluation of titles and producers from repertoire with the ones from playlist, these are performed only for the entities that meet the criterion expressed by Eq. (3). For these last cases, the number of JW algorithm instances is approximately rr, which is an order of magnitude lower than the number of instances used for the determination of author similarities. This leads to a total computational complexity of

$$O(VRmn + R^2MN) \approx O(VRmn). \quad (4)$$

The V and R parameters in our application have values of up to $1.000K$ and $100K$, respectively. On a general-purpose computer, this results in an increase in computational time of up to several hours, even if multithreading capabilities are used in implementation. Results and discussion section details these timing considerations. Thus, a GPU-based implementation is further on used in order to reduce the computational time. Before proceeding to the implementation, the specific tools for parallel computations available through the GPU architectures are summarized. It should be noted that this still does not address the issue of the low accuracy for the existing similarity metrics (see Table 1). One can say that the solution can be found among the large variety of machine learning techniques. Nevertheless, the large datasets and the large variety of categorical variables and large number of categories to be identified may lead to complex training procedures and increased computational time. Taking all of this into account, we can conclude that employing both GPUs and machine learning techniques represents the solution for determining string similarity in the application at hand. But first, let's have a look at the GPU-based implementation of the Jaro–Winkler algorithm.

2.1.2 GPU approach for parallel computing

Within the large variety of computing applications, graphics processing technology has evolved to provide unique benefits when parallel computation capabilities are needed. Thus, in addition to their intended use (i.e., accelerate the rendering procedure of the 3D graphics), the latest graphics processing units (GPUs) become flexible and programmable and serve as important tools used to significantly speed-up complex tasks in high-performance computing, deep learning, and many other fields. These hardware devices are composed of tens of *streaming multiprocessors* (SM). Each SM represents an individual computing unit which executes multiple 32-wide vector instructions in parallel. The CUDA application programming interface (API) used for building software applications which employ the GPUs for general purpose processing, refers to the vector instructions as warps. A warp is also known as 32 independent *threads*, where a thread can be defined as the smallest subunit of a computer program that can be executed independently. A group of warps is called a *thread block*, and it is associated with a SM. Considering the first GPU architectures from version 1 to Pascal v6.2, the warps were executed synchronously, an implicit synchronization procedure being applied in case of any thread's divergence within the same warp (Nvidia 2018). Starting with the Volta architecture, warp threads are executed asynchronously, whereas any threads synchronization must be explicitly programmed if needed (Nvidia 2019). In order to perform processing operations on the given data, the execution threads require memory to work with. The memory architecture of the GPUs is divided into three categories: global memory, shared memory, and local memory (Dominguez et al. 2020). The global memory is in the order of GBs, accessible by all the SM threads and used to store the application data and the expected post-processing results. Given the increased global memory size, it is preferable that consecutive threads access memory addresses in consecutive order. This approach is known as coalesced memory access, and it ensures optimal bandwidth utilization and low latency. The shared memory is on the order of MBs in size and has a lower latency than the global memory; however, it can only be shared within thread blocks. The local memory is the fastest, but it is also the smallest (in the order of MBs), and it can only be accessed by threads within a warp. Having multiple threads and their corresponding memory available, there is one more step to build software routines which make use of GPU parallel computing capabilities for data processing: writing CUDA *kernel functions*. A CUDA kernel is a GPU function which takes as arguments the input data arrays and arrays corresponding to the output data. Thus, these functions do not return any values, their results being written in the output data arrays. Moreover, each kernel has its own structure of threads (i.e., the number of thread blocks and the number of threads per block) declared each time they are executed. Important tools in kernel execution are the stride and atomic operations. In case similar independent computational steps are performed, they can be assigned to multiple threads executed at once. In case there are more steps to be performed than available threads, the *stride operation* comes in handy. Thus, after the execution of all available threads, the stride operation feeds new data to the same threads for performing the next computational steps. When results from one execution thread need to be used by another thread, *atomic operations* are used, which schedule the execution of threads sequentially.

2.1.3 Parallel implementation of the Jaro–Winkler-based record matching algorithm using GPU

Both Jaro–Winkler score computation and record matching algorithms from Figs. 1 and 2, respectively, are implemented using parallel computing capabilities of the GPUs. For this purpose the two kernels described in Figs. 3 and 4, respectively, are used. In the case of the first kernel, the subsets A, T and P, where the lower indices R or Y denote their correspondence to the repertoire or the playlist triplets, are copied within the GPU global memory. The similarity scores are computed for the pairs (A_R, A_Y), (T_R, T_Y) and (P_R, P_Y) using the same kernel. As shown in Fig. 3, for each a_j element in A_R, similarities with several a_i elements, with $i = 1 \cdots t_c$ in A_Y are computed in parallel using all t_c threads specified within the kernel configuration. The a_j similarity scores for the a_i elements with $i = t_c$. are computed using stride

Algorithm – Jaro Winkler

Input: s1 and s2 strings
Output: similarity score JWs (s1, s2)

```
# Variable initialization
1:   len1, len2     - s1 and s2 string sizes
2:   max_dist       - max(len1,len2)/2
3:   match          - count of matches
4:   h_s1, h_s2     - hash array of size len1 and len2
# Compute matches
5:   for i ← 1 to len1
6:       for j ← max(0,i-maxdist) to min(len2, i+maxdist)
7:           if (s1[i] == s2[j] and hash_s2[j] == 0) {
8:                   hash_s1[i] = 1; hash_s2[j] = 1
9:                   match += 1  }
# Compute score
10:   t = 0; point = 0;
11:   for i ← 1 to len1 {
12:       if (hash_s1[i])  {
13:           while (hash_s2[point] == 0)
14:                point += 1
15:           if (s1[i] != s2[point])   {   point += 1
16:                              t += 1  }
17:           else
18:               { point += 1 }
19:       t = t/2  }
20:   JWs =  ((match / len1 + match / len2 +
          (match - t) / match ) / 3.0)
```

Fig. 1 Computation of the string similarity score JWs for the (s_1, s_2) pair of strings using Jaro–Winkler algorithm

Algorithm – Entity match

Input: Repertoriu string sets (A,T,P)
Phonograma string sets (A_V, T_V, P_V)
Output: Entity matches list M(Repertoriu, Phonograma)

```
1:   for i ← 1 to size (A)
2:   for j ← 1 to size (AV)
3:       compute JWs(A[i], Av[j])
4:       if JWs (A[i], Av[j]) > k1
5:           store i and j indices as s_i, s_j
6:           for s_i and s_j indices
7:               compute JWs(T[s_i], Tv[s_j])
8:               compute JWs (P[s_i], Pv[s_j])
9:               if JWs(T[s_i], Tv[s_j]) > k2  && JWs
                          (P[s_i], Pv[s_j])>k3
10:                  store entity matches M(Repertoy,
                                            Playlist)
```

Fig. 2 String record matching procedure for the records included in the playlist (Phonograma) set with the records included in the repertory (Repertoriu) set, based on the similarity scores computed using the algorithm described in Fig. 1

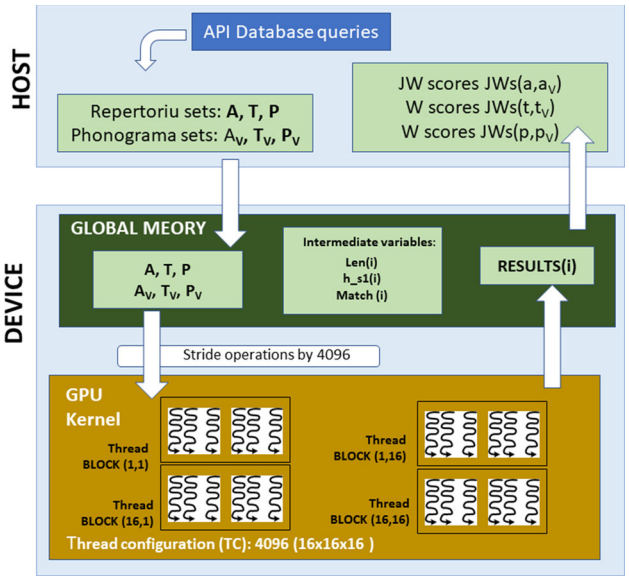

Fig. 3 Kernel 1: GPU global memory organization and threads configuration for performing parallel computation and stride operations for the determination of similarity scores sim_i in case of the pairs (A_R, A_Y), (T_R, T_Y) and (P_R, P_Y), representing repertory and playlist records

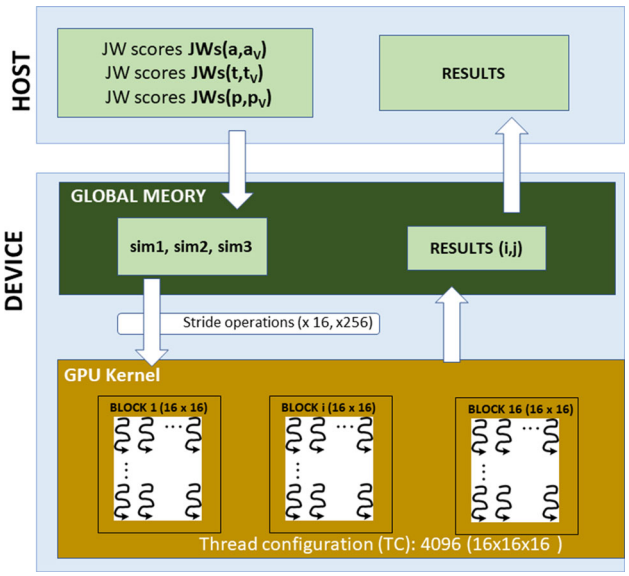

Fig. 4 Kernel 2: GPU global memory organization and threads configuration for performing the comparison of similarity scores sim_i with the thresholds $k1$, $k2$ and $k3$ described in Fig. 2 in order to determine a match or mismatch between the playlist and repertory records

operations. These computational steps support block-based data partitioning and coalesced memory access on GPU, for reduced computational time. The resulting similarity scores for the a_j with a_i elements with $i = 1 \cdots Y$ are stored within the global memory and further on copied to the destination HOST for storage. It is to be mentioned that intermediate variable arrays of size Y are stored in the global memory and used in similarity scores computation. The source code for the GPU implementation is available at Kaggle data repository together with a selection of entities from repertory and playlists sets (Kaggle 2019). Considering the second kernel, it implements part of the algorithm described in Fig. 2 to determine the list of matches between entities from the repertory and the ones from the playlist based on the simi-

larity scores computed by Kernel 1. Thus, it can be observed that the computational tasks corresponding to the pseudocode lines 3, 7 and 8 from Fig. 2 are performed by the CUDA Kernel 1. Their results are delivered as similarity score matrices $sim1$, $sim2$ and $sim2$ from the HOST to the GPU Device. Further on, the comparisons corresponding to the if statements from pseudocode lines 4 and 9 are executed in parallel by the Kernel 2. Thus, the proposed kernel configuration (i.e., 16 block threads, each containing several 16×16 threads) achieves 4096 comparisons executed at once, whereas stride operation delivers another 4096 similarity scores from the sim_i matrices to the same thread's configuration for processing. Coalesced memory access is assured for efficient memory access and reduced computational time. This algorithm section is also chosen for GPU implementation due to increased execution time of conditional branches generated by if statements in case of general-purpose CPUs.

2.2 Adaptive neural network for increased string similarity evaluation accuracy

Despite the parallel computation capabilities used for the string similarity evaluation, the main disadvantage remains the reduced efficiency of the Jaro–Winkler algorithm. A precision of 78% and 68% accuracy is achieved considering the threshold of 0.75 used in implementation. As referred to our database, using 0.75 threshold values lead to significantly increased number of false-positives matches. Empirically determined, we employed a threshold value of 0.9 for the authors and 0.8 threshold value for the titles and producers' information to use the JW similarity score as a stand-alone approach for similarity evaluation. In order to increase the accuracy, threshold values $k < 0.75$ are used to preliminary associate matches for each of the authors from the repertory to the ones from the playlist. Let M be the set of matches for a given author a_i. Even if significant false-positive results are included, a neural network approach is used to further on classify all the matches from the set M to the repertory entities corresponding to the author a_i, denoted by N. The main issue is that both the repertory and the playlist are updated over time. This addition of new data has an impact on the neural network structure to be used for classification, an issue also known as incremental computing. The general idea used for our adaptive neural network (aNN) to cope with the incremental computing is to use multiple neural networks, one for each author a_i and its corresponding titles. In this manner, when a new song n is added in the repertory, only one network aNN_i is updated in terms of its size and training, according to Fig. 5a. Regarding network training, a procedure which uses automatic data generation is used for training to classify input data corresponding to the new item n.

The number of our aNN and their size are determined based on our next observations related to the existing dataset. In the case of an 80k repertory size and 1.5 M playlist size, there is a set K of approximately 10k matches returned by the JW algorithm. Please note that the size of K is similar with the repertory size, nevertheless, only a few of a_n authors (e.g., in round numbers $a_n = 500$) are present in the playlist, due to the fact that the same title is associated with the same author multiple times. Also, it can be observed that an average of $a_k = 40$ items per author are present in the repertory. The range of parameters such as items per author and the number of similarities between the two sets lead to a number of a_n neural networks with the structure presented in Fig. 5. Considering a given a_i author from the repertory, each neural network aNN_i classifies the set M from the playlist, into one of the entities from the repertory, corresponding to the same author a_i. The total input nodes are $i_n = a + t + p + c + l$, where all the terms are constant values. The output layer is composed of $U_j = 1 \cdots a_k$ nodes, one for each of the repertory records corresponding to the author a_i. The output layer is used to represent the classification scores, which represent real numbers, each giving the probability of the input data to be associated with one of the a_k repertory entities. Two hidden layers are considered for the neural network. Ideally the number of hidden layer units is between N and $2N$, where N is the number of units in the output layer. Our choice is $3N/2$ nodes, considering the relatively increased number of predicted outputs. The node connections within network layers are unidirectional, leading to a feed forward topology. As for the activation function used by network elements, the softmax function is applied:

$$f(x_i) = e^{x_i} \Big/ \sum e^{x_j} \tag{5}$$

It uses the standard exponential function to each element of the input vector and normalizes these values by dividing them with the sum of all these exponentials, assuring that the sum of the output vector components is 1. The aforementioned output can be interpreted as the probability for the input data values to be associated with one of the output entities. Further on, the network training and the outliers detection are discussed as the key elements of the implementation. For training purposes, besides the existing playlists and repertory record entries, synthetic datasets are generated by introducing reversed and shortcuted titles, producer, and performer names together with typos, for each aNN_i classifier. On the other hand, considering the test data, there will be two types of inputs: inputs similar to the training data and inputs that differ in some respect from the data that are available during training. This classification task is also known as novelty detection, and it will be used in our case for detecting the abnormal data at the neural network input. For this purpose, according to Siddiqui and Boukerche (2021) and Song et al. (2021), an auto-detector is built for the identification of the novelty inputs, which correspond to the playlist

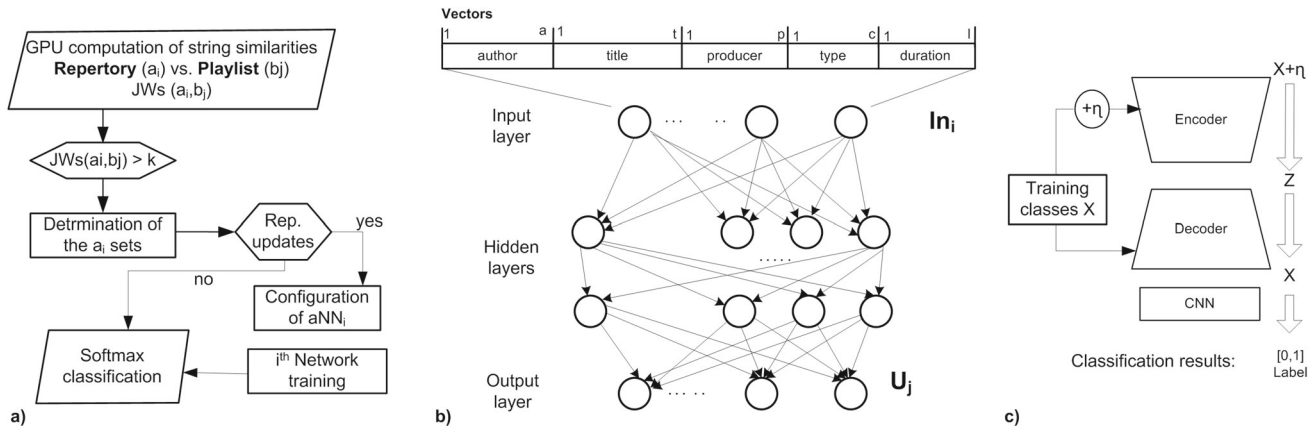

Fig. 5 **a** Adaptive neural network (aNN) for the classification of records corresponding to the ai author, **b** Neural network topology for the aNN, **c** auto-encoder for the identification of authors from the playlists which are not part of the ai group of records

records that are not part of the group of records corresponding to the a_i artist, due to the low Jaro–Winkler threshold used for record matching. Thus, a preliminary refinement of the input data applied to a given aNN_i network is performed with the help of the adversarially learned one-class classifier for novelty detection proposed in Sabokrou et al. (2018). Input records which do not correspond to the a_k classes of the aNN_i network corresponding to the author in question are classified as intrusions and excluded from the next processing step, namely, the adaptive neural networks for increased accuracy of the records matching procedure.

Let us describe further on the network training and validation procedures for the adaptive neural networks aNN_{PS} and aNN_{NP} used for classification of the playlist Records two well-known Romanian artists, namely (PS) and (NP). The number of output classes for the two authors are 105 and 176 for PS and NP, respectively. The network topology was previously described and it can be seen in Fig. 5b. At each aNN input layer composed of i_n nodes, a set of information vectors of approximately 10k size, representing playlist records for the corresponding a_i author, is delivered for training. The information vectors set is composed of both the existing playlists and repertory record and the synthetic datasets generated as described previously. As far as for the validation set, a number of 1k information vectors taken from playlist, repertory records and also from the generated synthetic datasets. Figure 6 shows the training and validation results where a faster accuracy convergence towards 93% is observed in case of the networks having a smaller number of inputs.

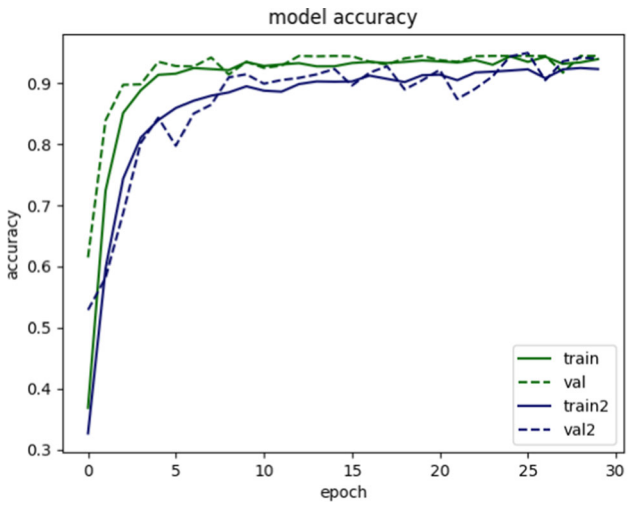

Fig. 6 Network training and validation accuracy considering the adaptive neural networks aNN_{PS} (train, val) and aNN_{NP} (train2, val2) corresponding to the two well-known Romanian artists, namely PS and NP

3 Results and discussion

In this section, the performances of both the proposed GPU implementation for string similarity metrics computation and the proposed machine learning approaches for string similarity evaluation by means of classification are presented and compared with existing approaches. Consequently, processing throughput and speed-up are analyzed considering multi-threading CPU and GPU implementations, whereas in terms of string similarity evaluation accuracy, specific quality measures such as accuracy, precision and recall scores are computed for classic algorithms such as Jaro–Winler and also for the machine learning algorithms used for performing the string-matching task. Please note that the proposed work envisages string matching by means of similarity metrics and neural network-based classification. Our experimental setup includes an Intel(R) i7-9750H CPU workstation, with 6 cores, 12 logical processors at 2.60 GHz, 32 GB of RAM and an NVIDIA GeForce RTX 2060 with 6 GB RAM. We used Visual Studio Code running on Windows 10 with python

numba libraries for CUDA programming and the Keras API for implementing neural network classification algorithms.

3.1 Evaluation of the GPU-based string similarity metrics computation

Throughput comparison for several input packets applied to both Jaro Winkler and Levenstein algorithms for similarity metrics computation is presented in Table 2. Our implementation used the Jaro Winkler approach whereas the implementation from Ho et al. (2016) used the Levenstein distance for similarity estimation. The accuracy of the two approaches is described in Sect. 2 and Santos et al. (2018). The throughput is calculated as the number of bytes in the input strings and the total processing time. Note that search throughput does not include data transfer time between the CPU and GPU. Table 2 shows a throughput comparison of the proposed approach with a similar one for string similarity evaluation. The resemblance of the two approaches is underlined in terms of computational complexity. Thus, the computational complexity is estimated based on the input data and string length for all the approaches accounted for string similarity evaluation. The number of computational steps performed by the full workflow used for the proposed record matching approach is given by $O(3VRn^2)$, according to Eq. (4) where M and N were each replaced with the average string length n. Regarding the Levenshtein distance computation, regular implementations also have a computational complexity of $O(n^2)$, where n represents the average string size. This leads to a total computational complexity of $O(Mn^3)$ for the overall approximate string-matching algorithm with k differences under the Levenshtein distance. It can be observed that, by adopting the GPU-based parallel execution, the performance was greatly enhanced compared to the CPU approach. Even though the computational complexity is similar, the increased throughput of our approach is mainly due to independent computing tasks performed by our implementation, compared with the approach presented in Ho et al. (2016).

For having a better image of the computational power introduced by the GPU, the entire workflow including CPU to GPU data transfer and vice versa is considered for speed-up evaluation of the proposed implementation compared with multi-threading-based CPU implementation and a classic CPU one. The hardware resources used for the implementation are specified in the beginning of the current section. Taking into account the repertory and playlist sizes of 100k records, the speed-up factor introduced by using a number of 10 threads reached to the value of 4.38×, compared to the classic CPU implementation. When using the GPU implementation, the speed-up factor compared with the classic CPU-based approach is 21.6×, considering the entire processing workflow. Considering these results, one can say the GPU approach for computing the string similarities can be used as a stand-alone procedure for the determination of playlist matches with the given repertory records. The powerful computation capabilities of the GPU architecture were used to compare one by one the repertory records with the playlist ones.

3.2 Evaluation of the string matching accuracy using string similarity metrics and neural networks approaches

Aside from the aforementioned straightforward approach (GPU-SA), an adaptive neural network approach aNN was introduced for string similarity evaluation. Its main benefits are the reduced number of string comparisons (i.e., string similarity metrics computation) together with the improved accuracy and the possibility of computing task deployment on multiple machines. The reduced number of operations can be observed by looking at the order of growth for the computational complexity of the aNN approach compared with the GPU-SA and a naive NN implementation namely the multi-features neural network (MF-NN) (see Table 3). The MF-NN approach was inspired by a high-accuracy model recognition method of mobile device based on weighted feature similarity proposed in Li et al. (2022). Herein, a number of 20 features were extracted from the network traffic and device physical attributes and a feature similarity metric rules was designed to calculate target device similarity with known devices. Before discussing the computational complexity of the three approaches in more detail, the MF-NN is shortly summarized. Thus, each record p_i from the playlist is compared in terms of string similarity with the repertory r_i ones. Multiple metrics (i.e., Jaro–Winkler, Levenshtein distance, permuted Jaro–Winkler, Jaccard N-grams and unique character count) are computed on the record strings and they represent features describing the (p_i, r_i) pair similarities. A multilayer perceptron classifier with the input layers corresponding to the previous features is trained and used to classify the (p_i, r_i) pair either as a match or as two different records. It becomes naturally that using the multiple similarity features and machine learning approaches for evaluating the records similarities the accuracy and the precision are high. Nevertheless, the computational complexity leads to an increased amount of hardware resources to perform the similarity evaluation in a reasonable time frame. Thus, considering a number of 5 features for the MF-NN implementation, the order of growth for the computational complexity is $15VRn^2$ for the features computation and VRn^5 for the machine learning classification, leading to the order $O(VRn^5 + 15VRn^2)$. Considering the aNN approach, let g be the number of authors in the playlist for which the JW score $\mathrm{JW}s(a_i, a_j) > k$ and a_n be the total number of authors present in the playlist. Considering the ga_n factor, the

Table 2 String records description together with string matching procedure throughput in case of both CPU and GPU implementations

String records description				Throughput (Gb/s)	
Input	Avg. length	Matches	Complexity	CPU	GPU
Hardware resources (Ho et al. 2016)—string matching under Levenstein distance				Intel Xeon E31270	Nvidia GPU GeForce GTX 660
12.88 (Castells-Rufas 2023)	16	198k	$O(Mn^3)$	0.025	1.98
22.87 (Ho et al. 2016)	16	580k	$O(Mn^3)$	0.026	1.99
Hardware resources—Jaro–Winkler score computation (Fig. 1)				Intel(R) i7-9750	NVIDIA RTX GeForce 2060
18 Mb	25	350k	$O(VRn^2)$	0.47	31.04
Record matching algorithm full workflow (Fig. 2)				Intel(R) i7-9750	NVIDIA RTX GeForce 2060
54 Mb	25	350k	$O(3VRn^2)$	0.34	7.41

computational complexity of the aNN approach is 3 orders of magnitude times lower than the MF-NN. Moreover, the computational complexity term VRn^2 is also significantly higher than the corresponding term for similarity metrics computation used in the aNN approach. Consequently, the increased accuracy of the MF-NN comes with the price of significantly increased computational complexity for large datasets.

Commonly, the efficiency of a machine learning classifier or any other classifier is described by the accuracy, precision and recall parameters computed based on true-positives, true-negatives, false-positives and false-negatives classifications performed by the method in question. It can be observed that, in the case of the GPU-SA approach, using a threshold value of 0.92, the accuracy reached 0.83 in case of the full workflow of the entity match algorithm (Fig. 4, 54 Mb input data). Due to the increased number of false negative classifications. Comparing the aNN with the GPU-SA procedure, the efficiency is significantly improved, whereas the computational complexity makes the approach feasible in terms of hardware resources and computational time.

4 Conclusions

String similarity metrics can be successfully used for string matching decisions by tuning the similarity threshold values. When large datasets are involved, high performance computing tools such as GPUs are available for similarity metrics computation aiming to obtain increased processing throughput. Consequently, we propose a string records matching procedure based on the Jaro–Winkler similarity metric for royalty rights identification and copyright protection of broadcast materials in the musical industry. The procedure was implemented onto a GPU, which leads to a processing throughput of 31.04 Gbps for the Jaro–Winkler similarity metric computation, and 7.41 Gbps for the full string record matching algorithm which includes the thresholding procedures. A matching accuracy of 0.83 was obtained, but there are still the following limitations to be mentioned. First, the manual tuning of similarity thresholds needs to be eliminated from the processing workflow, whereas the accuracy needs to be increased. Consequently, we proposed two machine learning approaches for string records matching, both starting from the parallel implementation of the Jaro Winkler similarity metrics. The first one entitled the multi-features neural network approach delivers increased accuracy (up to 0.99), for the record matching procedure, but the computational complexity is also increased, which makes it unsuitable for large datasets. On the other hand, the second approach, namely the adaptive neural network, leads to an accuracy of 0.92, whereas the computational complexity is approximately 3 orders of magnitude lower than the previous one.

Table 3 String matching procedure accuracy and computational complexity for the simple GPU implementation based on the Jaro–Winkler metrics and for the adaptive neural network and the multiple-features neural network implementations

Method	Accuracy	Precision	Recall	Complexity
GPU-SA	0.83	0.80	0.76	$O(3VRn^2)$
MF-NN	0.99	0.98	0.99	$O(VRn^5 + 15VRn^2)$
aNN	0.92	0.93	0.92	$O(ga_nn^5 + VRn^2)$

This fact qualifies it as an efficient solution for string records matching in case of large datasets.

Author Contributions All authors contributed to the material preparation, data collection, study conception and design. Algorithms development and data analysis were performed by AB and BB. The first draft of the manuscript was written by AB and BB, whereas all authors contributed on previous versions of the manuscript.

Funding This work was supported by: the Romanian Ministry of Education and Research through the Nucleu Programme, Project no. PN 23 24 02 01, the CNCS/UEFISCDI Project no. PN-III P4-ID-PCE-2020-1076, the Ministry of Research, Innovation and Digitization through CNCS/CCCDI-UEFISCDI Project no. PN-III-P2 2.1-SOL-2021-0084, and through the Program Funding Projects for Excellence in RDI, Contract No. 37PFE-CONSOL, 2021.

Data availability Enquiries about data availability should be directed to the authors.

Declarations

Conflict of interest The authors declare there are no conflict of interest.

Ethical approval The authors declare that the submitted work is original and have not been published elsewhere in any form or language. All authors read and approved the final version of the manuscript.

Informed consent Not applicable; the manuscript does not contain information concerning identifying details of any individuals.

References

Ayad L, Pissis S, Retha A (2016) libFLASM: a software library for fixed-length approximate string matching. BMC Bioinform 17:454

Baúto J, Canelas A, Neves R, Hort N (2018) Parallel SAX/GA for financial pattern matching using NVIDIA's GPU. Expert Syst Appl 105:77–88. https://doi.org/10.1016/j.eswa.2018.03.026

Bhat R, Thilak RK, Vaibhav RP (2022) Hunting the pertinency of hash and bloom filter combinations on GPU for fast pattern matching. Int J Inf Technol 14(5):2667–2679

Binoue K, Shimozono S, Yoshida H, Kurata H (2012) Application of approximate pattern matching in two dimensional spaces to grid layout for biochemical network maps. PLOS ONE 7:e37739. https://doi.org/10.1371/journal.pone.0037739

Budanitsky A, Hirst G (2006) Evaluating wordnet-based measures of lexical semantic relatedness. Comput Linguist 32(1):13–47

Castellanos FJ, Valero-Mas J, Calvo-Zaragoza J (2021) Prototype generation in the string space via approximate median for data reduction in nearest neighbor classification. Soft Comput 25:15403–15415

Castells-Rufas D (2023) GPU acceleration of Levenshtein distance computation between long strings. Parallel Comput 116(103):019

Chen S (2023) Design of computer big data processing system based on genetic algorithm. Soft Comput 27(11):7667–7678

Cinti A, Bianchi F, Aea Martino (2020) Novel algorithm for online inexact string matching and its FPGA implementation. Cogn Comput 12:369–387

Dominguez C, Moure-Lopez J, Bartrina-Lapesta J, Auli-Llinas F (2020) GPU-oriented architecture for an end-to-end image/video codec based on JPEG2000. IEEE Access 8:68474–68487

Ho TL, Oh SR, Kim HJ (2016) A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations. PLOS ONE 11(10):e0163535

Jea Salavert-Torres (2012) Using GPUs for the exact alignment of short-read genetic sequences by means of the burrows-wheeler transform. IEEE/ACM Trans Comput Biol Bioinform 9(4):77–88. https://doi.org/10.1016/j.eswa.2018.03.026

Kaggle (2019) String records matching. https://www.kaggle.com/datasets/ioanbogdanbelean/videograme-v9

Kumar M (2022) Scalable malware detection system using big data and distributed machine learning approach. Soft Comput 26(8):3987–4003

Kusudo K, Ino F, Hagihara K (2015) A bit-parallel algorithm for searching multiple patterns with various lengths. J. Parallel Distrib. Comput. 76:76–81

Lazreg M, Goodwin M, Granmo OL (2020) Combining a context aware neural network with a denoising autoencoder for measuring string similarities. Comput Speech Lang 60:101–028

Li R, Wang X, Luo X (2022) High-accuracy model recognition method of mobile device based on weighted feature similarity. Sci Rep 12(1):21–865

Lin CH, Wang GH, Huang CC (2014) Hierarchical parallelism of bit-parallel algorithm for approximate string matching on GPUs. Proc. Symp. Comput. Appl. Commun. 6:323–350

Maayah B, Arqub OA, Alnabulsi S, Alsulami H (2022) Numerical solutions and geometric attractors of a fractional model of the cancer-immune based on the Atangana–Baleanu–Caputo derivative and the reproducing kernel scheme. Chin J Phys 80:463–483

Mitani Y, Ino F, Hagihara K (2017) Parallelizing exact and approximate string matching via inclusive scan on a GPU. IEEE Trans Parallel Distrib Syst 28(17):1989–2002

Moore G (2015) Gordon Moore: the man whose name means progress, the visionary engineer reflects on 50 years of Moore's law. IEEE Spectrum

Naderalvojoud B, Ozsoy A (2021) A non-sequential refinement approach to improve word embeddings using GPU-based string matching algorithms. Clust Comput 24(4):3123–3134

Najam-ul Islam M, Zahra F, Aea Jafri (2022) Auto implementation of parallel hardware architecture for Aho–Corasick algorithm search. Des Autom Embed Syst 26:29–53

Navarro G (2001) A guided tour to approximate string matching. ACM Comput Surv 33:31–88. https://doi.org/10.1145/375360.375365

Nvidia (2018) Warp level primitives. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

Nvidia (2019) Warp level primitives. https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

Ongliang D, Xu J (2018) Massive fishing website URL parallel filtering method. IEEE Access 6:2378–2388

Perez-Wohlfeil E, Trelles O, Guil N (2023) Irregular alignment of arbitrarily long DNA sequences on GPU. J Supercomput 79(8):8699–8728

Rehs A (2021) A supervised machine learning approach to author disambiguation in the web of science. J Inform 15(3):101–166
Sabokrou M, Khalooei M, Fathy M, Adeli E (2018) Adversarially learned one-class classifier for novelty detection. In: IEEE/CVF conference on computer vision and pattern recognition, pp 3379–3388
Sadiq MU, Yousaf MM (2023) Space-efficient computation of parallel approximate string matching. J Supercomput 79(8):9093–9126
Samuel P, Subbaiyan S, Bea Balusamy (2021) A technical survey on intelligent optimization grouping algorithms for finite state automata in deep packet inspection. Arch Comput Methods Eng 28:1371–1396
Santos R, Murrieta-Flores P, Martins B (2018) Learning to combine multiple string similarity metrics for effective toponym matching. Int J Digit Earth 11(9):913–938
Siddiqui A, Boukerche A (2021) Adaptive ensembles of autoencoders for unsupervised IoT network intrusion detection. Computing 103(6):1209–1232
Sitaridi EA, Ross KA (2016) GPU-accelerated string matching for database applications. VLDB J 25(5):719–740
Song Y, Hyun S, Cheong Y (2021) Analysis of autoencoders for network intrusion detection. Sensors 21(13):4294
Sun Y, Ma L, Wang S (2015) A comparative evaluation of string similarity metrics for ontology alignment. J Inf Comput Sci 12(3):957–964
Thambawita R, Roshan V, Elkaduwe D (2016) Parallel failure-less Aho–Corasick algorithm for DNA sequence matching. In: IEEE international conference on information and automation for sustainability
Tran T, Liu Y, Schmidt B (2016) Bit-parallel approximate pattern matching: Kepler GPU versus Xeon Phi. Parallel Comput 54:128–138
Villa O, Scarpazza S, Petrini F (2008) Accelerating real-time string searching with multicore processors. Computer 41(4):42–50
Wang K, Sadredini E, Skadron K (2018) Hierarchical pattern mining with the automata processor. Int J Parallel Prog 46:376–411
Watanuki Y, Tamura K, Kitakami H, Takahashi Y (2013) Parallel processing of approximate sequence matching using disk-based suffix tree on multi-core CPU. In: IEEE sixth international workshop on computational intelligence and applications (IWCIA) pp 137–142
Wu S, Manber U (1992) Fast text searching allowing errors. Commun ACM 35(10):83–91
Yu M, Li G, Deng D (2016) String similarity search and join: a survey. Front Comput Sci 10:399–417. https://doi.org/10.1007/s11704-015-5900-5
Zou Q, Hu Q, Guo GW (2015) HAlign: fast multiple similar DNA/RNA sequence alignment based on the centre star strategy. Bioinformatics 31(15):2475–2481