Computing and Informatics, Vol. 43, 2024, 583–610, doi: 10.31577/cai_2024_3_583

PARALLEL NEAR-DUPLICATE DOCUMENT DETECTION USING GENERAL-PURPOSE GPU

Dimitar Peshevski, Vladimir Zdraveski

Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University
Rugjer Boshkovikj 16
1020, Skopje, North Macedonia
e-mail: `dimitar.peshevski@students.finki.ukim.mk`,
`vladimir.zdraveski@finki.ukim.mk`

Sashko Ristov

Department of Computer Science
University of Innsbruck
Technikerstraße 21a
A - 6020, Innsbruck, Austria
e-mail: `sashko.ristov@uibk.ac.at`

Abstract. In today's data-rich world, one of the most significant challenges is efficiently identifying near-duplicate data, especially when integrating data from various sources. Identifying near-duplicate documents applies to any content and has been widely used to enhance the efficiency of search engines, identify plagiarism or spam, and so on. Even smaller or specialized search engines can benefit from knowledge about near-duplicate documents. Shingling and MinHash are two state-of-the-art approaches to detecting near-duplicate documents. However, there are not many attempts to improve the performance of this locality-sensitive hashing technique. In this research paper, we propose a parallel implementation of the MinHash algorithm for near-duplicate document detection utilizing the immense parallelism offered by general-purpose GPUs. Experimental results show that the GPU-based parallel solution is far more cost-effective than the CPU-based sequential and parallel solutions.

Keywords: Near-duplicate, document, Shingling, similarity, locality-sensitive hashing, MinHash, fingerprint, parallelism, GPU, CUDA

Mathematics Subject Classification 2010: 68W10

1 INTRODUCTION

Duplicate document identification and removal are beneficial for a variety of reasons. Apart from others, document identification and removal brings benefits

1. for search engines [1] and web crawlers [2] to minimize their processing time,
2. to detect and fight against plagiarism and spam contents [3, 4], and
3. to filter out repeated news items from diverse information channels.

Although deduplication is mainly applied for text content, many techniques are used to deduplicate other multimedia documents, such as image detection [5], virus scanning [6], datasets deduplication [7], and handwriting recognition. In this paper, we focus on the detection of near-duplicate text documents.

There are numerous attempts to reduce the complexity of determining duplicates from a given document dataset. One such successful attempt is the MinHash algorithm which is a locality-sensitive hashing algorithm. It reduces the complexity by representing the document as a fingerprint signature with a limited length, called a MinHash signature [8]. In this way, instead of comparing the contents of each document with the contents of the other documents, it is solely necessary to compare their fingerprints, which, in principle, are small in size.

But that does not change the fact that the processing takes time $O(n\log(n))$, where n is the size of the document collection [1]. Doing this processing sequentially for a dataset consisting of thousands of documents will take an extremely long time. That is why we considered parallelization of the near-duplicate document detection process and making use of the general-purpose graphics processing unit (GPGPU).

The general concept of this research paper is to parallelize the document comparison and finding the near-duplicates. The goal is to achieve this by each thread block[1] of the graphics processor being responsible for one document at a time, comparing it to all the others, and identifying its near-duplicates. Experimental results show that the GPU-based parallel solution is far more cost-effective than the CPU-based sequential and parallel solutions. We evaluated our algorithm with various numbers of documents and achieved a speedup of up to 30× compared to the CPU-based sequential solution and a speedup of up to 4× compared to the CPU-based parallel solution using 10 000 documents.

[1] A thread block is a programming abstraction representing a group of threads that can be executed sequentially or concurrently.

2 DOCUMENT STRUCTURE

The rest of this paper is structured as follows. We outline the related work we have discovered in the literature in Section 3. In Section 4, we explore CPU and GPU architectures, offering insights into their parallelization techniques, distinctions in clock speeds, and the number of processing units. This section also draws specific references to the architectures used in our research. Section 5 describes the MinHash algorithm, along with a summary of the solution architecture. Section 6 provides an outline of the experiments conducted and the associated results. Finally, in Section 7, we conclude our work and outline our next steps for the future.

3 RELATED WORK

This section presents the related work split into two parts, the motivation for deduplication, state-of-the-art algorithms for deduplication, and approaches to parallelize the deduplication.

3.1 Motivation for Deduplication

These days, digital content is widespread and simply redistributable, either legally or illegally, making the world plagued by redundant and plagiarised data of every kind. For those reasons, automated robust methods for duplicate detection of documents and textual data are getting more attention recently. In [1], the author presents an effective technique for filtering near-duplicate content that has been used within the context of the AltaVista search engine. The basic approach provided in [1] for computing resemblance between documents involves two aspects:

1. the resemblance is defined as a Jaccard similarity coefficient, and
2. the relative size of the intersections is calculated via a random sampling process that can be done individually for each document.

Identifying the content automatically plays a significant role in detecting and fighting plagiarism and spam content in today's digital world connected through the Web. In the matter of duplicate document detection, document fingerprinting provides an alternate and efficient solution for managing and identifying near-duplicate content. Ho and Kim [3] propose a method using a trie-tree data structure to store a set of 64-bit strings, each of which is the fingerprint of a web document. After which, they use a meet-in-the-middle approach to detect near-duplicate documents. Also, document fingerprinting combined with sentence feature extraction and clustering has promising experimental results that demonstrate its validity and effectiveness [9].

On the other hand, detecting near-duplicate documents is fundamental to the content ecosystem of information flows. In [10], Montanari and Puglisi provide a new method for finding near-duplicate documents based on q-grams derived from the text. The proposed algorithm was evaluated in a multifeed news content management system to filter out repeated news items from diverse information channels.

Also, the presence of near-duplicates affects the performance of the search engines critically, because of their need to be managed and potentially removed from query answers to provide users with uncluttered search results. The authors of [4] look into the use of SimHash and shingle-based approaches for detecting near duplicates in CiteSeer^X. CiteSeer^X is a real-world digital library of academic papers that collects documents automatically through focused crawling. Similar versions of a document may exist on various sites on the Web, and the goal is for these multiple versions to be included in the collection automatically as a result of the automatic crawling and ingesting.

Web crawling is an integral piece of infrastructure for search engines and its performance depends on the elimination of near-duplicate web documents. Manku et al. [2] demonstrate that SimHash is practically useful for identifying near-duplicates in web documents from a multibillion-page repository. They come to the conclusion that the problem's scaling limits the solution to small-sized fingerprints. Fortunately, the SimHash approach with 64-bit fingerprints appears to work well in practice for an 8 B web page repository. It is worth mentioning that they also developed a technique for solving the Hamming distance problem.

3.2 Algorithms for Deduplication

As we see, there are many reasons why a lot of near-duplicate document detection algorithms are proposed nowadays. Among the most efficient such algorithms and solutions are effective and fast near-duplicate detection via signature-based compression metrics [11], duplicate text detection based on Longest Common Subsequence (LCS) algorithm [12], efficient near-duplicate document detection using consistently weighted sampling filter [13], adaptive near-duplicate detection via similarity learning [14], detecting near-duplicate documents using sentence-level features and supervised learning [15], etc.

In [11], Zhang et al. present SigNCD, a new near-duplicate approach that combines the signature extraction procedure with normalized compression distance to cope with immense collections of documents with a wide range of lengths. The solution demonstrated in [12] is an improvement of Broder's Shingling and MinHash approaches in detecting near-duplicate documents [1]. Specifically, it is a method that combines Shingling and LCS algorithm called SWLR (Shingling with Location Relationship). To quickly and accurately compute the similarity in large-scale datasets, Yuan et al. [13] build a threshold filter based on CWS and offer a quicker weighted hash similarity measurement algorithm. The authors of [14] describe a novel near-duplicate document identification algorithm that may be easily adjusted for a specific domain. Their method treats each document as a real-valued sparse k-gram vector, with weights learned to optimize for a certain similarity function, such as cosine similarity or the Jaccard coefficient. This improved similarity measure can accurately find near-duplicate documents. Similarly, Lin et al. [15] describe a novel method for finding near-duplicates in a huge document collection. Their method has three important components: feature selection, similarity measure, and discriminant derivation.

The most common approach, i.e. solution for near-duplicate document detection is a min-wise independent permutations locality-sensitive hashing scheme [8] also known as MinHash which is one of the most popular techniques for finding approximate nearest neighbor searches in high-dimensional spaces. Implementation of this algorithm proved to be an efficient way to detect duplicates between a corpus of news articles [16]. Furthermore, a study proposed Sectional MinHash (S-MinHash) [17], specifically designed for the detection of near-duplicate documents, which enhances the MinHash data structure with information about the location of the attributes in the document. Also, an extended version of the S-MinHash method for near-duplicate detection was proposed recently [18].

Near-duplicate detection does not apply only to text documents. Also, it finds extensive application when it comes to multimedia. The paper [5] gives an overview of the computer vision techniques used for near-duplicate image detection. Its applicability spans even across data exploration, data integration, and data quality. In [7], the authors propose a technique for detecting quasi-duplicate datasets which is based on feature extraction and machine learning. In this paper, we focus on the MinHash algorithm.

3.3 Approaches to Parallelize Deduplication

Document processing is an evergreen topic in parallelization and has been the subject of extensive research and development over the years. Security is one area where parallelism has been very successful, with signature-based virus scanning [6] being a notable example. Similarly, parallelization has shown promise in speeding up and enhancing the accuracy of handwriting recognition systems using parallel neural network techniques [19]. Finally, general-purpose GPUs are being used to accelerate data compression [20], resulting in shorter processing times and faster data transfer and storage.

Although the problem of finding near-duplicates stretches across many different domains, and even though many algorithms tackle this problem, there are still not enough parallel or distributed solutions that would speed up these algorithms. In this paper, we focus on parallelizing MinHash algorithm using GPUs.

4 BACKGROUND

Modern computing is underpinned by the architectures of Central Processing Units (CPUs) and Graphics Processing Units (GPUs), each uniquely adapted to address various computational demands.

In this section, we describe general characteristics specifics for the infrastructures and platforms that are used in this paper and details for the specific types of infrastructure.

4.1 Infrastructure

4.1.1 CPUs

CPUs, at the heart of most computing systems, are designed for various applications, optimizing both reliability and adaptability. While their design is primarily focused on sequential processing, it is worth noting that CPUs can also be employed for parallel tasks when necessary using multithreading. For our research, we employ two different CPUs on two distinct platforms.

On Google Colaboratory, we utilize the Intel Xeon CPU, a member of Intel's Xeon processor family, characterized by its Broadwell microarchitecture. Operating at a frequency of 2.20 GHz, this CPU boasts a substantial cache size of 56.32 MB, encompassing L1, L2, and L3 caches. Its core attributes incorporate a solitary physical core supported by hyper-threading, facilitating the execution of two logical threads.

On our local bare-metal, we utilize the Intel Core i7-8700 CPU, which operates at a base clock speed of 3.20 GHz. This CPU features six physical cores and supports hyper-threading, allowing each core to handle two threads simultaneously, enabling efficient multitasking and parallel processing. Built on Intel's Coffee Lake microarchitecture, with its cache hierarchy encompassing L1, L2, and L3 caches, the Intel Core i7-8700 enhances data access speed and computational efficiency. This CPU is commonly found in desktop computers and workstations, making it ideal for various applications, including content creation, video editing, and general computing, where its balance of clock speed and core count delivers robust performance.

4.1.2 GPUs

In contrast, GPUs exhibit an architecture explicitly engineered for parallel computation, making them exceptionally proficient in tasks necessitating high throughput. Particularly, we use two different GPUs for our research, each employed on its respective platform.

The first is the NVIDIA Tesla T4 GPU, part of the Tesla series, which features the Turing architecture. This GPU boasts 16 GB GDDR6 memory designed for rapid and efficient data access and is utilized on Google Colaboratory. At the core of the Tesla T4's architecture, lie 40 Streaming Multiprocessors (SMs), each containing 64 CUDA cores. This results in a total of 2 560 CUDA cores, which excel at executing tasks concurrently across a multitude of threads. Despite the lower clock speed of around 1.6 GHz compared to CPUs, the sheer number of CUDA cores and the innovative architecture enable the GPU to achieve significantly higher computational throughput.

Additionally, we use the NVIDIA GTX 1060 GPU on our local bare-metal, built on the Pascal architecture. Equipped with 6 GB of GDDR5 video memory and 1280 CUDA cores, this GPU operates at a base clock speed of 1.5 GHz and is well-suited for GPU-accelerated tasks such as scientific simulations, machine learning, and 3D rendering.

4.1.3 CPUs vs. GPUs

Distinguishing between the Intel Xeon CPU and the NVIDIA Tesla T4 GPU, both manifest disparities in clock speed and processing units. The Intel Xeon CPU, operating at 2.20 GHz, excels in orchestrating complex computations, data management, and task coordination. In contrast, the NVIDIA Tesla T4, despite its lower clock speed of around 1.6 GHz, is equipped with 16 GB GDDR6 memory and an abundance of CUDA cores, designed explicitly for parallel tasks such as graphics rendering, machine learning, and scientific simulations. While clock speed is a crucial metric, the number of processing units and their parallel execution capabilities are equally pivotal in determining the computational ability of these two entities.

Comparing the Intel Core i7-8700 CPU and the NVIDIA GTX 1060 GPU, we see distinct strengths. The Intel Core i7-8700, with its 3.20 GHz base clock speed and six cores with hyper-threading, excels in multitasking and complex calculations, making it ideal for content creation, video editing, and general computing. In contrast, the GTX 1060 GPU, designed for graphics and parallel processing, offers 6 GB of GDDR5 video memory and around 1 280 CUDA cores, excelling in tasks like scientific simulations, machine learning, and 3D rendering. While its clock speed is lower than the Intel Core i7-8700 CPU, its parallel capabilities stand out in applications requiring massive parallelism. In summary, the CPU thrives in diverse computing tasks, while the GPU is optimized for parallel workloads and to various domains, including scientific research, artificial intelligence, and graphical simulations.

4.2 Platforms

4.2.1 Google Colaboratory

For our research, we use the platform offered by Google Colaboratory which consists of an Intel Xeon CPU with two virtual CPUs (vCPUs) and 13 GB of RAM. In addition to CPU and RAM, Google Colaboratory also offers access to NVIDIA Tesla T4 GPU.

4.2.2 The Local Bare-Metal

Our research also employs a local bare-metal with an Intel Core i7-8700 CPU running at 3.20 GHz and an NVIDIA GeForce GTX 1060 GPU with 6 GB of memory, along with 16 GB of system memory.

5 SOLUTION ARCHITECTURE

In this section, we provide an overview of the solution architecture used for near-duplicate document detection. Particularly, in Section 5.1, we provide a detailed overview of the MinHash algorithm and how it is used to determine the near-duplicates among a document dataset. The most attention is devoted to the modeling of the documents and the estimation of their similarity. In Section 5.2, we describe how the graphics processing unit is utilized for our purposes and what are the advantages of using it in that way. In Section 5.3, we present the programming language we used, and we give the details about the solution infrastructure. Lastly, in Section 5.4, we provide a link to our public repository where we published the code in order for our results to be repeatable and reproducible.

For this research paper, a near-duplicate document detection system was developed based on the locality-sensitive hashing algorithm MinHash making use of the massive parallelism the general-purpose graphics processing units (GPGPUs) are offering nowadays. The general idea, illustrated in Figure 1, is that with the help of a similarity estimate, the system searches for similar documents among a dataset, which computes how close the documents are to each other.

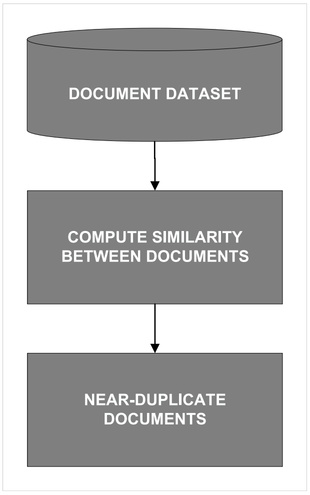


Figure 1. Solution architecture diagram

The similarity assessment between documents is done by first transforming each document into a fingerprint that characterizes it. Then each fingerprint is compared with the fingerprints of the other documents. If the similarity estimate is greater than a given threshold, those documents are considered near-duplicates. Graphically, the comparison process is shown in Figure 2.

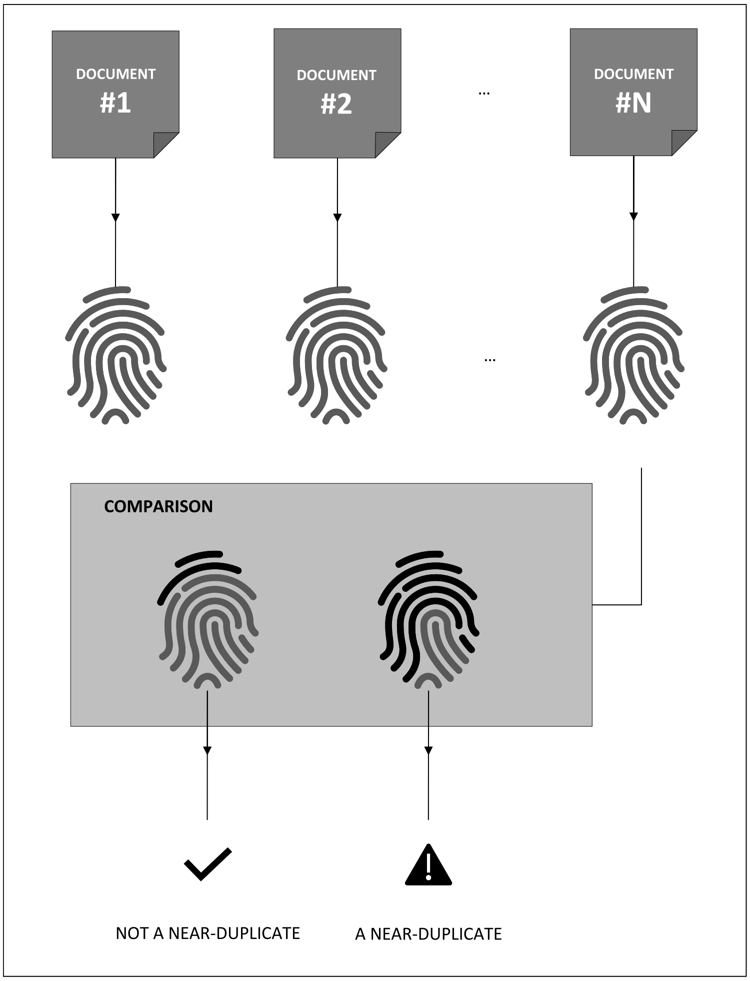


Figure 2. Process of similarity estimating between documents

5.1 MinHash Algorithm

MinHash is a locality-sensitive hashing technique suitable for approximate similarity searches of sparse sets. Here, a document is modeled as a sparse set of shingles. MinHash then allows performing a nearest-neighbor search among all such sparse sets within a document dataset.

When it comes to transforming a document into a set of shingles, the most common approach is to construct a set of all shingles appearing in the document dataset and then represent each document as a one-hot encoded vector regarding the shingles appearing in it. An alternative approach is to represent each document as a set of shingles appearing in it and then hash each value. This way, the hash value can be treated as an index of the corresponding shingle in the set of all shingles appearing in the documents, achieving a compact document representation that can further simplify calculations. In this research paper, we are using the CRC-32 hash function as we assume that the count of all different shingles in a document dataset cannot exceed $2^{32} = 4\,294\,967\,296$.

After the process of transforming each document into a set of hashed shingles, the next step is creating a MinHash signature. The collections of sets are permuted, hashed, and stored in a matrix known as the signature matrix. Let h_1, h_2, …, h_n be different MinHash functions (i.e. different permutations). Then the signature for a set S is:

$$[h_1(S), h_2(S), \dots, h_n(S)]. \tag{1}$$

Generally, the MinHash signatures have a fixed length. In the case of this research paper, 64 for each document in the dataset.

Here, we are facing the first issue regarding the MinHash algorithm. When the data dimensionality is too big, it becomes impractical (or too expensive) to use (and store) a permutation matrix for the random permutations required by the MinHash algorithm. One major limitation of GPUs is that they have fairly limited memory. Thus, we resort to simple hash functions. So, the solution is to use a random hash function (for row number) to simulate a permutation. Each hashing function h_i, $i = 1, 2, \dots, n$, has the following form:

$$h_i(k) = ((a \cdot k + b)\,\%\,p)\,\%\,m, \tag{2}$$

where k is the index of the k^{th} shingle in the set, a and b are randomly chosen integers, m is the count of shingles in the vocabulary and p is a prime number slightly bigger than m.

Various similarity measures exist to assess the resemblance between two documents. Commonly, the similarity between two documents is calculated considering the Jaccard similarity coefficient, which normalizes the number of identical objects by the number of distinct objects in the two sets. Let X and Y denote two documents, and S_X and S_Y their corresponding sets of shingles. Then, the Jaccard similarity coefficient is calculated regarding the equation:

$$J(S_X, S_Y) = \frac{|S_X \cap S_Y|}{|S_X \cup S_Y|}. \tag{3}$$

However, the Jaccard similarity coefficient can be easily estimated using the signature matrix with the advantage that we calculate it on fewer data, leading to better performance.

The final step is finding the near-duplicates among a document dataset. If two documents have a count of equal elements within their MinHash signatures greater than a given threshold, then they can be regarded as near-duplicates.

5.2 Utilizing the Advantages of GPUs

GPUs greatly outperform CPUs in terms of computing performance and memory bandwidth. However, because GPUs were built for graphics processing, the programming model (which incorporates massively parallel Single-Instruction-Multiple-Data (SIMD) processing and limited bus speeds for data transfers to/from main memory) is not ideal for arbitrary data processing applications.

The approach outlined in this research paper makes the most of this model by envisaging three main phases:

1. Sending compact representation of the documents and their corresponding indices to the GPU memory. The transformation to a compact representation is done upon a variant of the Compressed Row Storage (CRS) format.
2. Each thread block is responsible for one document. The hash values and accompanying minimum are then computed by threads in the corresponding block by applying all n hash functions to the data in the GPU and keeping the relevant minimum for each hash function and set.
3. Then once again each document is assigned to a particular block which compares that document's signature with the signatures of all succeeding documents finding all documents with a similarity estimate greater than a given threshold. And, then writes the results back to the main memory.

The benefits of this batch-style processing are numerous. Because the computation within the GPU itself scans through successive blocks of data in the GPU-internal memory (as opposed to random memory access patterns), we can take full use of coalesced access and the immense parallelism inherent in the GPU architecture by executing the same computation (with a different hash function) for each set entry n times.

But there are also downsides that need to be overcome. Writing the result from a similarity comparison performed in the GPU raises a few challenges. Since the size of the result is initially unknown, it is also not possible to know how much memory should be allocated from the GPU memory to hold the result. In addition, there may be conflicts between blocks when writing on the device's memory. For this reason, the GPU processes the documents twice. The first time it only determines the count of near-duplicates of each document. After that, it allocates the needed memory space for the result, and the second time each block writes the indices of the near-duplicates to the corresponding place in the allocated memory.

The five phases of the near-duplicate detection process mentioned before are illustrated in Figure 3. The outline of the process is summarized in Figure 4 with a flow chart. The document set is loaded first. Then, each document is assigned to a separate thread block, which performs the necessary processing to return the near-duplicates of the corresponding document as a result.

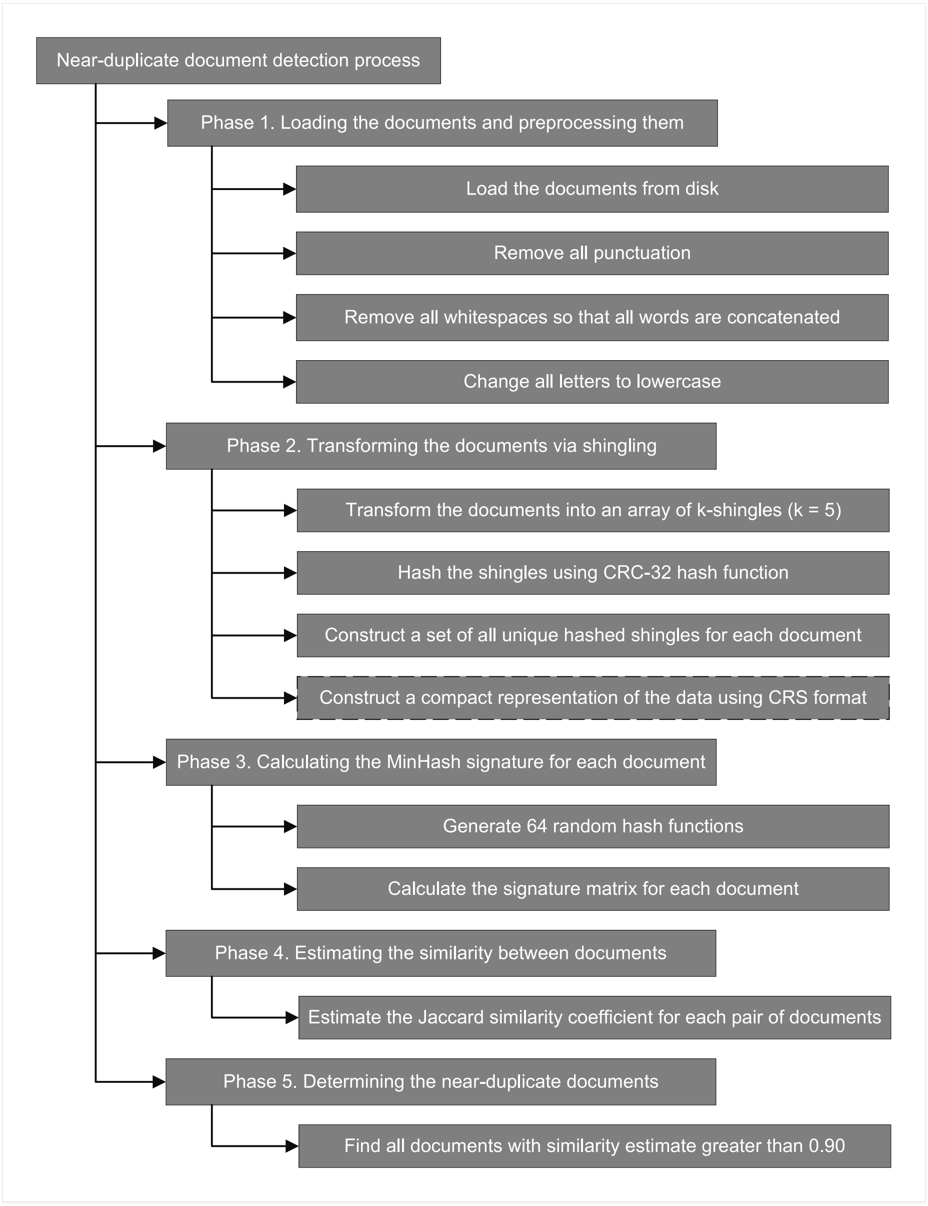

Near-duplicate document detection process
Phase 1. Loading the documents and preprocessing them
Load the documents from disk
Remove all punctuation
Remove all whitespaces so that all words are concatenated
Change all letters to lowercase
Phase 2. Transforming the documents via shingling
Transform the documents into an array of k-shingles (k = 5)
Hash the shingles using CRC-32 hash function
Construct a set of all unique hashed shingles for each document
Construct a compact representation of the data using CRS format
Phase 3. Calculating the MinHash signature for each document
Generate 64 random hash functions
Calculate the signature matrix for each document
Phase 4. Estimating the similarity between documents
Estimate the Jaccard similarity coefficient for each pair of documents
Phase 5. Determining the near-duplicate documents
Find all documents with similarity estimate greater than 0.90

Figure 3. Phases of the near-duplicate detection process

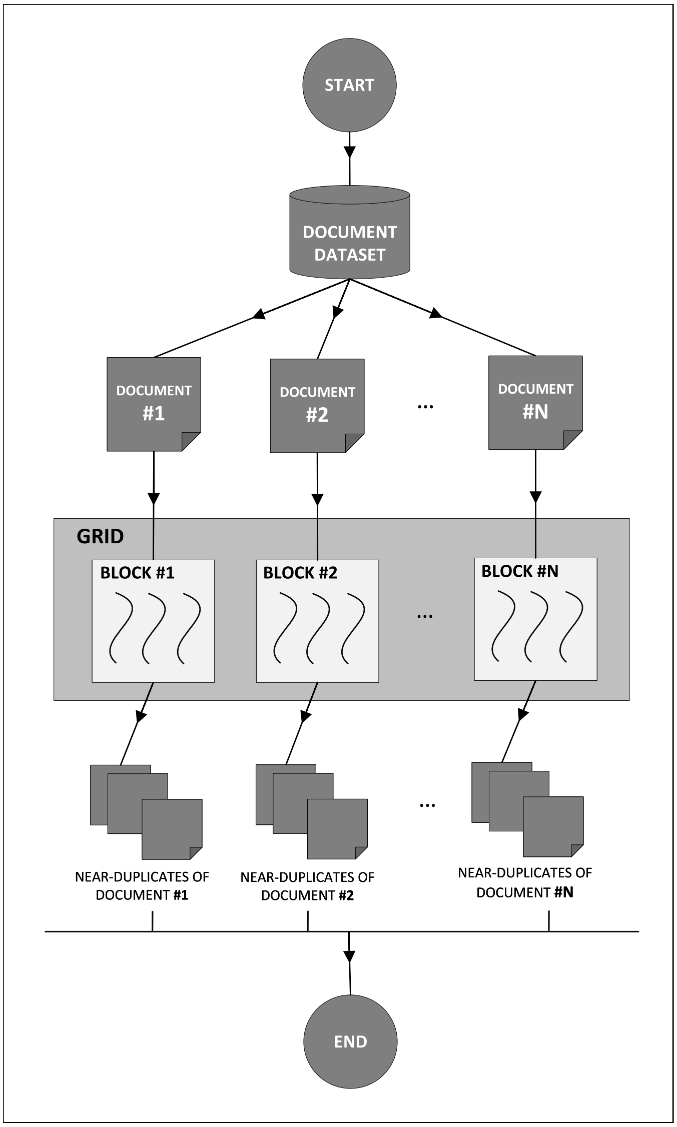


Figure 4. Flow chart of the near-duplicate detection process

5.3 Programming Language and Solution Infrastructure

For the approach presented in this paper, the programming language Python was used because it is powerful, flexible, and easy to use. When it comes to utilizing the GPU for the parallelization of tasks, Numba was used as it supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model making it possible to leverage massively parallel GPU computing to achieve faster results and accuracy. With Numba, one can write kernels directly with (a subset of) Python, and Numba will compile the code on the fly and run it. Numba was also utilized for the parallel implementation on the CPU because of its support for multithreading.

Numba's optimization process is a noteworthy aspect that deserves mention. Specifically, when a function is invoked for the first time using Numba, it employs the LLVM compiler infrastructure to translate the Python code into efficient machine code. This translation process significantly enhances the execution speed of the code. In fact, the performance of code compiled by Numba can often rival that of programs written in C, C++, or Fortran. This optimization process by Numba is a dynamic and just-in-time (JIT) compilation technique. It means that rather than interpreting Python code line by line, Numba compiles the code into machine-level instructions on the fly, allowing for faster execution.

More precisely, we use Numba for GPU programming in such a way that a separate kernel is built for each of the last three phases outlined in Figure 3. We launch and run each of the kernels one after the other. Meanwhile, after launching each of the kernels we synchronize to ensure that the kernel finishes its work, and then we measure the time taken for each phase.

The solution was evaluated with different sizes of document dataset and we concluded that the speedup grows along with increasing the size of the document dataset. The details of the experiments performed and the results obtained from them are given in the next section.

5.4 Implementation

In order to make our results reproducible, we published our code in a publicly available GitHub repository. The code for all three implementations: 1. CPU-based sequential implementation, 2. CPU-based parallel implementation, and 3. GPU-based parallel implementation, is available at the following link: `https://github.com/peshevskidimitar/ParallelNearDuplicateDocumentDetection`.

6 RESULTS

In this section, we present the results of the evaluation. We first describe the experimental setup in Section 6.1 in order to be repeatable and reproducible. Then, we discuss the achieved results and their implications in Section 6.2 and Section 6.3 according to the respective platform.

6.1 Experiments

6.1.1 Experiment Infrastructure

The first platform utilized for our experiments is Google Colaboratory, featuring an Intel Xeon CPU running at 2.20 GHz and an NVIDIA Tesla T4 GPU equipped with 16 GB of GDDR6 memory. Detailed specifications and characteristics of this platform can be found in Section 4.

Our second platform employs a local bare-metal, equipped with an Intel Core i7-8700 CPU and an NVIDIA GeForce GTX 1060 GPU. Further information about this platform can also be found in Section 4.

6.1.2 Experiment Execution

The time measurements were divided into 5 phases according to the execution of the algorithm illustrated in Figure 3. Only phases 3 through 5 were parallelized utilizing the advantages of GPU.

The final measurements are determined through the calculation of the mean from a total of 101 individual runs for each respective phase. For each phase, we present both the mean (referred to as AVG) and the relative standard deviation (abbreviated as RSD).

The solution presented in this research paper was tested on multiple subsets of a document dataset of size 1 000, 2 500, 5 000, 7 500, and 10 000 documents. The average document length in the given dataset was 1 589.53 characters.

6.2 Experimental Results on Google Colaboratory

Our experimental results indicate that parallelization reduces the time required to compute near-duplicates from a dataset of text documents.

In Table 1, the mean of each phase execution time for the CPU-based sequential solution is given in detail by phases, and in the last row, the total time required for the whole process of near-duplicate document detection is calculated. In Table 2, we provide the relative standard deviation of each phase execution time for the CPU-based sequential solution. Analogously to the measurements for the CPU-based sequential solution, we present the measurements for the CPU-based parallel solution in Tables 3 and 4, and the measurements for the GPU-based parallel solution in Tables 5 and 6.

As we can see from Tables 1, 3, and 5, the GPU-based parallel solution is from 18 to 30 times faster than the CPU-based sequential solution and from 3 to 4 times faster than the CPU-based parallel solution. In both (CPU-based and GPU-based) parallel implementations, there is overhead in phase 2 because of the need for

#	1 000	2 500	5 000	7 500	10 000
1. phase	32.87	65.18	139.55	211.20	277.74
2. phase	654.12	1 607.31	3 199.91	4 871.18	6 320.39
3. phase	11 313.42	27 779.84	54 723.24	83 410.12	109 835.53
4. phase	912.59	5 312.78	22 763.59	49 968.58	90 482.11
5. phase	45.27	219.65	1 074.63	2 452.95	4 242.08
Total	12 958.27	34 984.76	81 900.92	140 914.03	211 157.85

Table 1. Mean (in milliseconds) of each phase execution time for the CPU-based sequential implementation on Google Colaboratory

#	1 000	2 500	5 000	7 500	10 000
1. phase	26.32	23.07	26.26	25.64	25.78
2. phase	26.75	23.41	23.92	17.39	14.54
3. phase	3.73	2.09	3.96	1.79	1.87
4. phase	22.13	14.58	4.14	2.33	4.01
5. phase	7.13	11.01	13.89	13.06	13.52

Table 2. Relative standard deviation (in percentages) of each phase execution time for the CPU-based sequential implementation on Google Colaboratory

a more compact representation of the documents for transferring them to the GPU memory and making most of the multithreading on the CPU. On average, it occupies 0.11 % of the total time required for the execution of the CPU-based parallel solution and 0.37 % of the total time required for the execution of the GPU-based parallel solution, so we can conclude that the overhead is negligible. The improvement with the GPU-based parallel solution is even better illustrated by Figure 5, where all implementations are compared.

The performance improvement is most evident from the ratio of the time required for the CPU-based sequential and parallel solution to the time needed for the GPU-based parallel solution presented in Table 7 and Table 8 for each document subset size. For 10 000 documents, the GPU-based parallel solution is 30 times faster than the CPU-based sequential solution and 4 times faster than the CPU-based parallel solution. The same data from Table 7 and Table 8 are plotted in Figure 6 and

#	1 000	2 500	5 000	7 500	10 000
1. phase	32.87	65.18	139.55	211.20	277.74
2. phase	658.79	1 611.51	3 207.50	4 889.22	6 347.85
3. phase	1 575.43	4 046.80	8 365.61	13 766.12	18 792.61
4. phase	35.50	200.75	873.39	1 899.43	3 413.83
5. phase	1.17	3.12	13.24	39.79	52.18
Total	2 303.76	5 927.36	12 599.29	20 805.76	28 884.21

Table 3. Mean (in milliseconds) of each phase execution time for the CPU-based parallel implementation on Google Colaboratory

#	1 000	2 500	5 000	7 500	10 000
1. phase	26.32	23.07	26.26	25.64	25.78
2. phase	25.39	22.13	21.95	17.46	14.72
3. phase	18.50	19.08	11.43	5.14	3.74
4. phase	31.13	25.33	27.72	23.09	18.04
5. phase	5.98	10.90	4.38	29.98	6.48

Table 4. Relative standard deviation (in percentages) of each phase execution time for the CPU-based parallel implementation on Google Colaboratory

#	1 000	2 500	5 000	7 500	10 000
1. phase	32.87	65.18	139.55	211.20	277.74
2. phase	658.79	1 611.51	3 207.50	4 889.22	6 347.85
3. phase	9.15	16.22	30.11	45.81	62.20
4. phase	5.36	18.22	71.30	123.97	317.93
5. phase	1.26	1.39	1.74	2.01	2.61
Total	707.43	1 712.52	3 450.20	5 272.21	7 008.33

Table 5. Mean (in milliseconds) of each phase execution time for the GPU-based parallel implementation on Google Colaboratory

Figure 7, where it becomes clear that the larger the dataset, the more efficient the GPU-based parallel solution is.

We expect the speedup to increase for larger cardinalities of the document set, but up to a certain point where it reaches its limit and starts decreasing. Specifically, we expect a linear speedup for a collection ranging from 10 000 to 100 000 documents when the speedup reaches its limit. Our predictions are primarily based on the practices already demonstrated for modeling the speedup for scalable systems [21], the fact that we utilized a GPU with an array of 40 streaming multiprocessors for the parallel solution and that we already obtained a speedup of 30 times for a collection of 10 000 documents.

It is worth mentioning that even though GPUs are designed primarily for working with numerical data and mathematical operations rather than processing textual data, we managed to achieve a speedup of 30 times for a dataset consisting of 10 000 documents.

#	1 000	2 500	5 000	7 500	10 000
1. phase	26.32	23.07	26.26	25.64	25.78
2. phase	25.39	22.13	21.95	17.46	14.72
3. phase	23.61	23.74	18.83	13.25	10.59
4. phase	11.57	10.21	4.68	5.64	4.57
5. phase	29.37	26.62	25.86	25.37	22.99

Table 6. Relative standard deviation (in percentages) of each phase execution time for the GPU-based parallel implementation

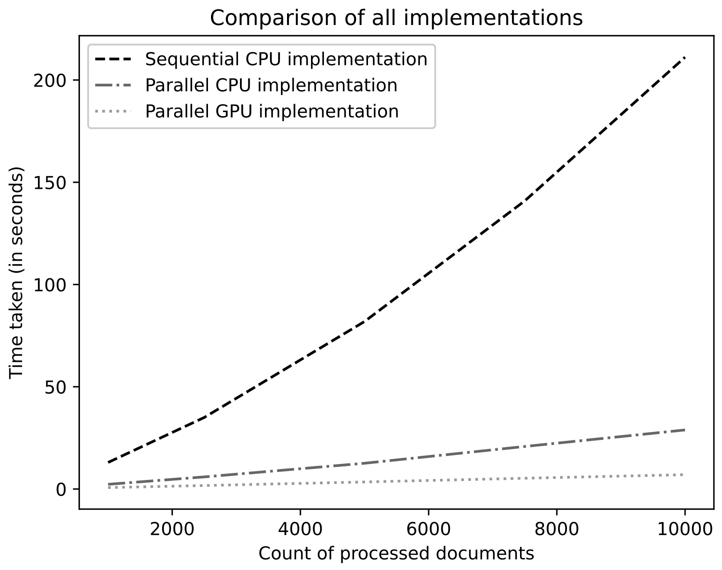


Figure 5. Comparison of all implementations on Google Colaboratory

#	1 000	2 500	5 000	7 500	10 000
Speedup	18.32	20.43	23.74	26.73	30.13

Table 7. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based sequential implementation on Google Colaboratory

6.3 Experimental Results on the Local Bare-Metal

Similarly to the preceding section, where we showcased results from Google Colaboratory, in this section we provide the results obtained from our local bare-metal.

Table 9 displays the mean of the execution times for each phase of the CPU-based sequential solution, along with the total time for the entire near-duplicate document detection process in the last row. Moving to Table 10 it presents the relative standard deviation of the execution times for each phase for the CPU-based sequential solution. Similarly, Tables 11 and 12 present the measurements for the CPU-based parallel solution, while Tables 13 and 14 present the measurements

#	1 000	2 500	5 000	7 500	10 000
Speedup	3.26	3.46	3.65	3.95	4.12

Table 8. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based parallel implementation on Google Colaboratory

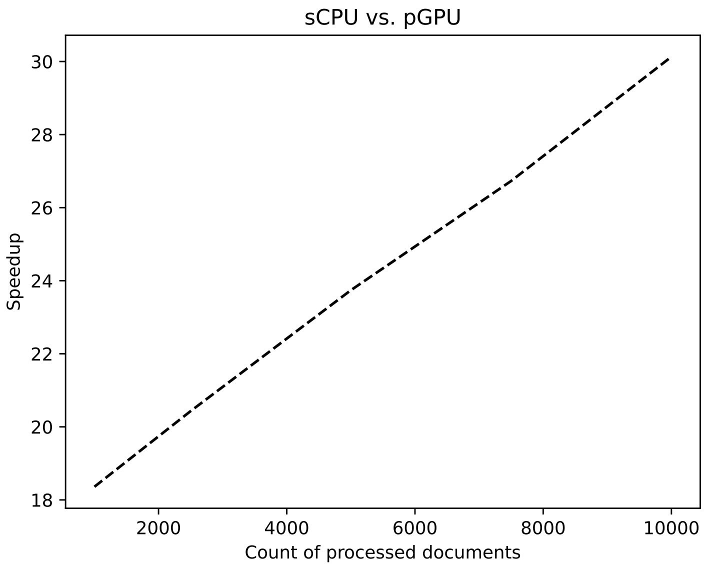


Figure 6. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based sequential implementation on Google Colaboratory

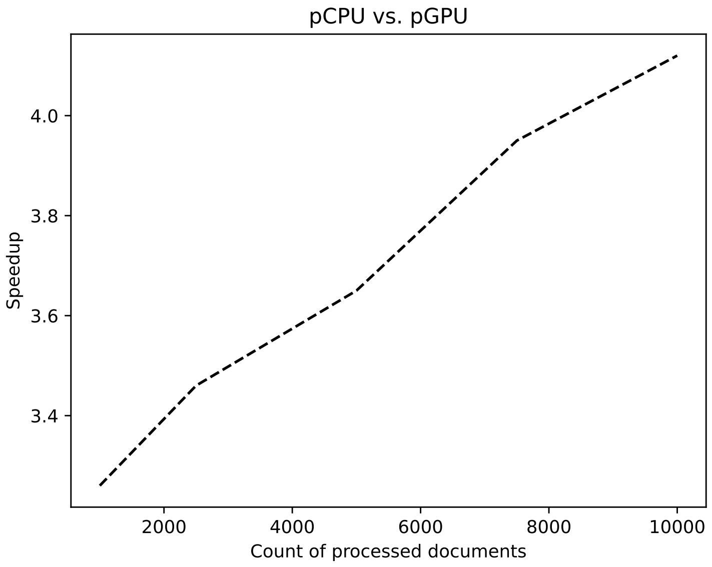


Figure 7. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based parallel implementation on Google Colaboratory

for the GPU-based parallel solution, mirroring the structure and format used for Tables 9 and 10.

#	1 000	2 500	5 000	7 500	10 000
1. phase	18.31	47.25	94.16	142.91	192.44
2. phase	520.19	1 316.32	2 627.26	3 889.01	5 193.00
3. phase	7 853.52	19 437.31	39 067.19	58 908.81	78 238.03
4. phase	604.31	3 800.90	17 215.09	35 477.77	63 472.59
5. phase	22.47	144.13	609.38	1 358.66	2 566.11
Total	9 018.80	24 745.91	59 613.08	99 777.16	149 662.17

Table 9. Mean (in milliseconds) of each phase execution time for the CPU-based sequential implementation on the local bare-metal

#	1 000	2 500	5 000	7 500	10 000
1. phase	3.82	6.50	2.25	4.60	1.48
2. phase	1.79	2.58	2.00	1.55	1.36
3. phase	2.98	0.73	0.45	0.24	0.22
4. phase	1.86	2.11	1.12	0.79	0.67
5. phase	10.10	2.18	2.21	1.96	6.33

Table 10. Relative standard deviation (in percentages) of each phase execution time for the CPU-based sequential implementation on the local bare-metal

#	1 000	2 500	5 000	7 500	10 000
1. phase	18.31	47.25	94.16	142.91	192.44
2. phase	531.01	1 330.79	2 663.22	3 940.20	5 286.77
3. phase	200.81	500.84	991.47	1 494.46	1 997.40
4. phase	1.94	10.87	41.38	91.30	161.81
5. phase	0.18	0.89	3.71	8.43	14.99
Total	752.25	1 890.64	3 793.94	5 677.30	7 653.41

Table 11. Mean (in milliseconds) of each phase execution time for the CPU-based parallel implementation on the local bare-metal

Observing the data presented in Tables 9, 11, and 13, it is evident that the GPU-based parallel solution outperforms the CPU-based sequential solution by a factor of 12 to 20 and is 1.3 times faster than the CPU-based parallel solution. As previously mentioned, both parallel implementations exhibit overhead during phase 2 due to the requirement for a more efficient document representation, facilitating their transfer to GPU memory and optimizing CPU multithreading. On average, this overhead consumes only 1.10 % of the total execution time for the CPU-based parallel solution and 1.38 % for the GPU-based parallel solution, reaffirming its

#	1 000	2 500	5 000	7 500	10 000
1. phase	3.82	6.50	2.25	4.60	1.48
2. phase	4.09	1.42	1.33	1.22	1.17
3. phase	4.20	4.59	2.76	3.30	3.01
4. phase	8.76	7.91	8.02	6.42	5.45
5. phase	11.11	12.36	5.39	5.10	3.54

Table 12. Relative standard deviation (in percentages) of each phase execution time for the CPU-based parallel implementation on the local bare-metal

#	1 000	2 500	5 000	7 500	10 000
1. phase	18.31	47.25	94.16	142.91	192.44
2. phase	531.01	1 330.79	2 663.22	3 940.20	5 286.77
3. phase	18.69	43.48	84.48	125.48	166.21
4. phase	3.58	17.35	67.92	153.36	257.88
5. phase	0.65	0.97	1.46	2.45	3.77
Total	572.24	1 439.84	2 911.24	4 364.40	5 907.07

Table 13. Mean (in milliseconds) of each phase execution time for the GPU-based parallel implementation on the local bare-metal

negligible impact. The enhanced performance achieved with the GPU-based parallel solution becomes even more evident when examining Figure 8, which provides a comparison of all implementations.

We can also observe the performance improvement by comparing the time required for the CPU-based sequential and parallel solutions to the time taken by the GPU-based parallel solution, as displayed in Tables 15 and 16 for various document subset sizes. The same data from Tables 15 and 16 are plotted in Figures 9 and 10.

In this scenario, the speedup of the GPU-based parallel solution compared to the CPU-based sequential solution steadily increases, while it remains relatively constant compared to the CPU-based parallel solution. We anticipate that as the dataset size grows, the speedup achieved by the CPU-based parallel implementation compared to the CPU-based sequential implementation will eventually reach a saturation point. Furthermore, we believe that the speedup for the GPU-based parallel

#	1 000	2 500	5 000	7 500	10 000
1. phase	3.82	6.50	2.25	4.60	1.48
2. phase	4.09	1.42	1.33	1.22	1.17
3. phase	6.90	4.23	2.94	1.71	0.94
4. phase	11.17	4.38	4.20	3.05	2.46
5. phase	16.92	11.34	13.01	19.18	9.02

Table 14. Relative standard deviation (in percentages) of each phase execution time for the GPU-based parallel implementation on the local bare-metal

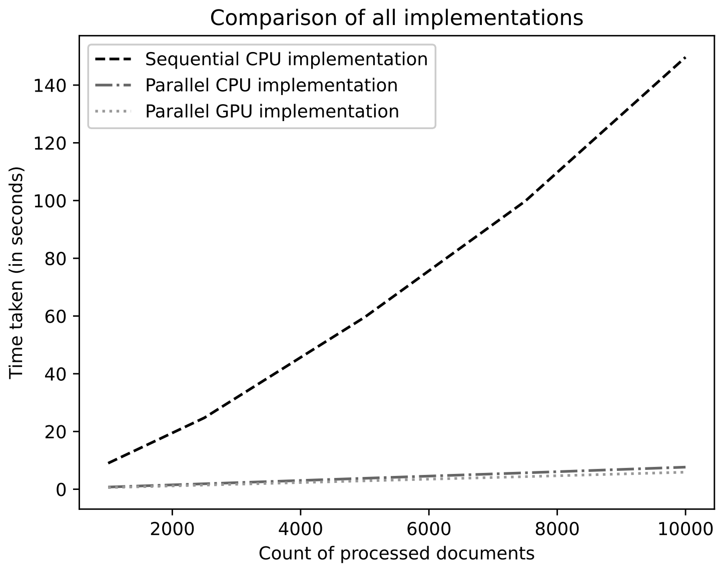


Figure 8. Comparison of all implementations on the local bare-metal

implementation compared to the CPU-based parallel implementation will become more apparent with larger datasets.

#	1 000	2 500	5 000	7 500	10 000
Speedup	11.99	13.09	15.71	17.57	19.55

Table 15. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based sequential implementation on the local bare-metal

#	1 000	2 500	5 000	7 500	10 000
Speedup	1.31	1.31	1.30	1.30	1.30

Table 16. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based parallel implementation on the local bare-metal

6.4 Discussion

The local measurements differ when compared to those obtained on Google Collaboratory. Several factors contribute to this difference, with infrastructure being a significant one. On Google Collaboratory, we have access to a single core of Intel Xeon with two threads, whereas locally, we benefit from the processing power of six Intel Core i7-8700 cores, each featuring two threads. This implies a clear advantage

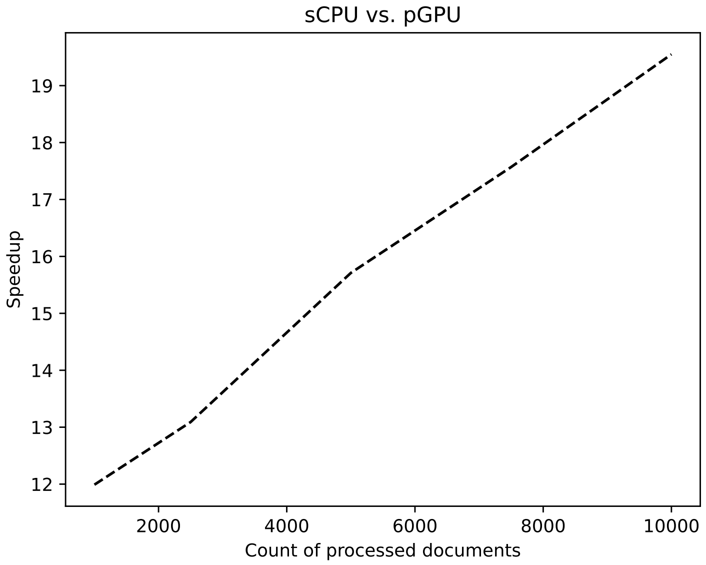


Figure 9. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based sequential implementation on the local bare-metal

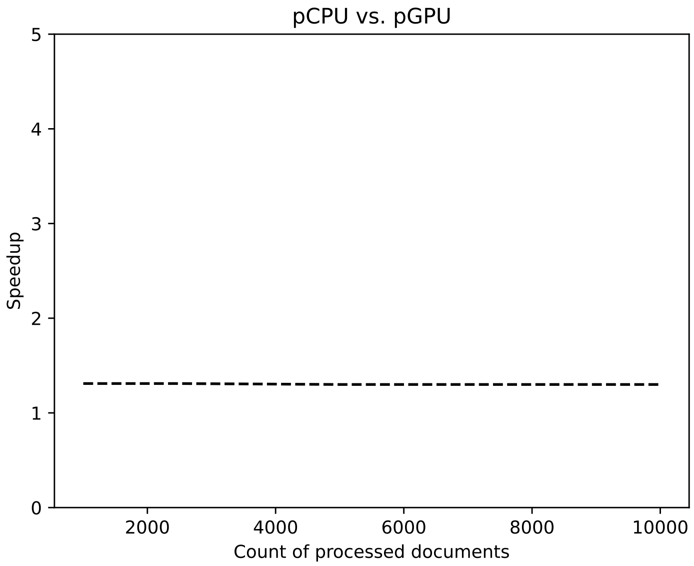


Figure 10. Speedup achieved by the GPU-based parallel implementation compared to the CPU-based parallel implementation on the local bare-metal

in terms of CPU performance when experimenting locally. However, the situation differs when it comes to the graphics processing unit (GPU). Locally, we rely on the NVIDIA GeForce GTX 1060, whereas Google Collaboratory offers the NVIDIA Tesla T4 with 13 GB of GPU memory. This disparity in GPU resources can also impact the observed trends in our measurements.

The local bare-metal CPU outperforms the CPU on Google Colaboratory, leading to better results in Tables 9 and 11 compared to Tables 1 and 3. However, there is a noteworthy contrast when it comes to GPU performance. Despite Google Colaboratory having a faster GPU than the local bare-metal setup, Table 5 does not show better results than Table 13. This difference can be attributed to the initial two non-parallelized phases, significantly contributing to the overall execution time. The slower CPU on Google Colaboratory causes these phases to dominate much more in terms of time than on the local bare-metal, ultimately impacting the final results negatively. Additionally, we believe that the higher relative standard deviation (RSD) of execution times for the GPU-based parallel implementation on Google Colaboratory (Table 6) compared to the GPU-based parallel implementation on the local bare-metal (Table 14) likely plays a substantial role in this outcome.

6.5 Threats to Validity

Google Colaboratory operates on a strategy facilitated through dynamic usage limits and a lack of guarantee for unlimited resources. This dynamic approach results in periodic fluctuations in several essential aspects, such as overall usage limits, idle timeout durations, maximum virtual machine (VM) lifetimes, available GPU types, and other relevant factors.

In such a shared-resource scenario, the performance of a job may be influenced by the simultaneous execution of other tasks on the same hardware, as resource allocation is dynamically managed by the platform. This could result in variations in the available computing power, memory, and GPU resources, affecting the execution time of a given job. Consequently, there is a threat to validity when interpreting execution time measurements obtained in such an environment, as they may not accurately reflect the intrinsic performance of the code or algorithms being tested.

In addition to Google Colaboratory, we employ our local bare-metal as another platform for our proof-of-concept work. This approach allows us to compare results obtained on a non-virtualized local bare-metal with those from a virtualized environment on Google Colaboratory, thereby gaining a more comprehensive understanding of our measurements and ensuring the robustness of our findings.

7 CONCLUSION AND FUTURE WORK

In this research paper, we parallelized part of the near-duplicate document detection process. In particular, phases 3 through 5. The parallelization of these three phases of the near-duplicate document detection process proved worthwhile, considering the results of the conducted experiments. For 10 000 documents, the GPU-based parallel solution was 30 times faster than the CPU-based sequential one and 4 times faster than the CPU-based parallel one.

Despite this, we believe there is still additional potential for parallelizing a greater part of the process. Specifically, there is a possibility of parallelization of phase 2, in which documents are transformed via shingling. The transformation process is independent for each document and this indicates an opportunity for parallelization. But the challenges that arise should not be neglected. The size of the set of unique shingles that represents the document is initially unknown, so the size of the memory space, that needs to be allocated to accommodate the document representation cannot be known in advance. Furthermore, GPUs are designed primarily for working with numerical data and mathematical operations, rather than processing textual data. Hence, finding a technique to translate text processing to mathematical operations on numerical data will be necessary.

In our future work, we will try to develop the algorithm on a serverless platform as a workflow. We intend to use the new type of serverless infrastructures which has emerged, also known as the Function-as-a-Service model. The authors of [22] assess the FaaS model's applicability for compute- and data-intensive scientific workflows and discuss potential ways to repurpose serverless infrastructures for scientific workflow execution. Also, in [23], Ristov et al. introduce a scalable middleware service xAFCL that can schedule and execute different functions of the same FC across multiple FaaS systems. Considering these possibilities, we plan to build the solution according to the FaaS model, making the most of the advantages it offers.

We also intend to build the solution on a more robust architecture that can handle larger document datasets. So we can run experiments with datasets ranging in size from 10 000 to 1 000 000 documents to see if we can achieve a linear speedup with the number of streaming multiprocessors. We also want to achieve greater scaling and so exceed the current GPU limitation in terms of the number of streaming multiprocessors.

If we succeed in overcoming these challenges, we believe we will achieve even more significant results. Near-duplicate document detection has numerous potential applications in various domains. Its improvement through parallelization increases those possibilities considering that most of today's computers have general-purpose GPUs.

REFERENCES

[1] Broder, A. Z.: Identifying and Filtering Near-Duplicate Documents. In: Giancarlo, R., Sankoff, D. (Eds.): Combinatorial Pattern Matching (CPM 2000). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 1848, 2000, pp. 1–10, doi: 10.1007/3-540-45123-4_1.

[2] Manku, G. S.—Jain, A.—Das Sarma, A.: Detecting Near-Duplicates for Web Crawling. Proceedings of the 16th International Conference on World Wide Web (WWW '07), ACM, 2007, pp. 141–150, doi: 10.1145/1242572.1242592.

[3] Ho, P. T.—Kim, S. R.: Fingerprint-Based Near-Duplicate Document Detection with Applications to SNS Spam Detection. International Journal of Distributed Sensor Networks, Vol. 10, 2014, No. 5, Art. No. 612970, doi: 10.1155/2014/612970.

[4] Williams, K.—Giles, C. L.: Near Duplicate Detection in an Academic Digital Library. Proceedings of the 2013 ACM Symposium on Document Engineering (DocEng '13), 2013, pp. 91–94, doi: 10.1145/2494266.2494312.

[5] Thyagharajan, K. K.—Kalaiarasi, G.: A Review on Near-Duplicate Detection of Images Using Computer Vision Techniques. Archives of Computational Methods in Engineering, Vol. 28, 2021, No. 3, pp. 897–916, doi: 10.1007/s11831-020-09400-w.

[6] Dimitrioski, A.—Gusev, M.—Zdraveski, V.: Parallelism in Signature Based Virus Scanning with CUDA. In: Poulkov, V. (Ed.): Future Access Enablers for Ubiquitous and Intelligent Infrastructures (FABULOUS 2019). Springer, Cham, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 283, 2019, pp. 413–422, doi: 10.1007/978-3-030-23976-3_36.

[7] Chevallier, M.—Rogovschi, N.—Boufarès, F.—Grozavu, N.—Clairmont, C.: Detecting Near Duplicate Dataset. In: Abraham, A., Engelbrecht, A., Scotti, F. et al. (Eds.): Proceedings of the 13th International Conference on Soft Computing and Pattern Recognition (SoCPaR 2021). Springer, Cham, Lecture Notes in Networks and Systems, Vol. 417, 2022, pp. 394–403, doi: 10.1007/978-3-030-96302-6_36.

[8] Broder, A. Z.—Charikar, M.—Frieze, A. M.—Mitzenmacher, M.: Min-Wise Independent Permutations (Extended Abstract). Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC '98), 1998, pp. 327–336, doi: 10.1145/276698.276781.

[9] Kumar, J. P.—Govindarajulu, P.: Near-Duplicate Web Page Detection: An Efficient Approach Using Clustering, Sentence Feature and Fingerprinting. International Journal of Computational Intelligence Systems, Vol. 6, 2013, No. 1, pp. 1–13, doi: 10.1080/18756891.2013.752657.

[10] Montanari, D.—Puglisi, P. L.: Near Duplicate Document Detection for Large Information Flows. In: Quirchmayr, G., Basl, J., You, I., Xu, L., Weippl, E. (Eds.): Multidisciplinary Research and Practice for Information Systems (CD-ARES 2012). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7465, 2012, pp. 203–217, doi: 10.1007/978-3-642-32498-7_16.

[11] Zhang, X.—Yao, Y.—Ji, Y.—Fang, B.: Effective and Fast Near Duplicate Detection via Signature-Based Compression Metrics. Mathematical Problems in Engineering, Vol. 2016, 2016, Art. No. 3919043, doi: 10.1155/2016/3919043.

[12] Yu, J.—Li, M.—Zhang, D.: Duplicate Text Detection Based on LCS Algorithm. Proceedings of the 2nd Information Technology and Mechatronics Engineering Conference (ITOEC 2016), Atlantis Press, 2016, pp. 5–9, doi: 10.2991/itoec-16.2016.2.

[13] Yuan, X.—Wang, S.—Peng, C.—Zhang, C.: Efficient Near-Duplicate Document Detection Using Consistent Weighted Sampling Filter. International

Journal of Network Security, Vol. 21, 2019, No. 6, pp. 947–956, doi: 10.6633/IJNS.201911_21(6).08.

[14] Hajishirzi, H.—Yih, W. T.—Kolcz, A.: Adaptive Near-Duplicate Detection via Similarity Learning. Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '10), 2010, pp. 419–426, doi: 10.1145/1835449.1835520.

[15] Lin, Y. S.—Liao, T. Y.—Lee, S. J.: Detecting Near-Duplicate Documents Using Sentence-Level Features and Supervised Learning. Expert Systems with Applications, Vol. 40, 2013, No. 5, pp. 1467–1476, doi: 10.1016/j.eswa.2012.08.045.

[16] Rodier, S.—Carter, D.: Online Near-Duplicate Detection of News Articles. In: Calzolari, N., Béchet, F., Blache, P. et al. (Eds.): Proceedings of the Twelfth Language Resources and Evaluation Conference (LREC 2020). European Language Resources Association, 2020, pp. 1242–1249, `https://aclanthology.org/2020.lrec-1.156`.

[17] Hassanian-Esfahani, R.—Kargar, M. J.: Sectional MinHash for Near-Duplicate Detection. Expert Systems with Applications, Vol. 99, 2018, pp. 203–212, doi: 10.1016/j.eswa.2018.01.014.

[18] Shayegan, M. J.—Faizollahi-Samarin, M.: An Extended Version of Sectional MinHash Method for Near-Duplicate Detection. The Journal of Supercomputing, Vol. 78, 2022, No. 13, pp. 15638–15662, doi: 10.1007/s11227-022-04447-x.

[19] Todorov, D.—Zdraveski, V.—Kostoska, M.—Gusev, M.: Parallelization of a Neural Network Algorithm for Handwriting Recognition: Can We Increase the Speed, Keeping the Same Accuracy. 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), 2021, pp. 932–937, doi: 10.23919/MIPRO52101.2021.9597042.

[20] Ristovski, K.—Zdraveski, V.: Accelerating Data Compression Using General Purpose GPUs. The 19th International Conference on Informatics and Information Technologies – CIIT 2022, 2022, pp. 144–147, `http://hdl.handle.net/20.500.12188/25705`.

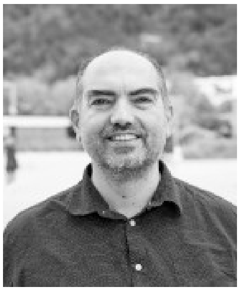
[21] Ristov, S.—Gusev, M.—Velkoski, G.: Modeling the Speedup for Scalable Web Services. In: Bogdanova, A. M., Gjorgjevikj, D. (Eds.): ICT Innovations 2014. Springer, Cham, Advances in Intelligent Systems and Computing, Vol. 311, 2015, pp. 177–186, doi: 10.1007/978-3-319-09879-1_18.

[22] Malawski, M.—Gajek, A.—Zima, A.—Balis, B.—Figiela, K.: Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. Future Generation Computer Systems, Vol. 110, 2020, pp. 502–514, doi: 10.1016/j.future.2017.10.029.

[23] Ristov, S.—Pedratscher, S.—Fahringer, T.: xAFCL: Run Scalable Function Choreographies Across Multiple FaaS Systems. IEEE Transactions on Services Computing, Vol. 16, 2023, No. 1, pp. 711–723, doi: 10.1109/TSC.2021.3128137.

Dimitar Peshevski is a third-year bachelor's student at the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, in Skopje, North Macedonia, where he is pursuing a degree in computer science. In his studies, he focuses on the theoretical and algorithmic foundations of computing, parallel and distributed processing, data science, artificial intelligence, and its sub-domain machine learning. His research interests include algorithm design and optimization, performance optimization of parallel and distributed systems, and high-performance computing. Upon completion of his undergraduate studies, he intends to apply to graduate school to further his knowledge of computer science and progress toward a career as a researcher.

Vladimir Zdraveski graduated in 2010 in the field of computer science and engineering and obtained his Master's degree in e-technologies and computer networks in 2012, and his Ph.D. in computer science and engineering in 2017. Currently, he is Associate Professor at the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University in Skopje. He participated in a few international and local research projects, published several papers, co-authored one book and two patents. His ongoing research is focused on engaging high-performance computing tools in different domains. His academic and industrial interests are also in web technologies, parallel programming and enterprise software development.



Sashko Ristov is Assistant Professor at the University of Innsbruck, Austria. His research interests include performance modeling and optimization of parallel and distributed systems, serverless computing, cloud engineering, and cloud federation. He received his Ph.D. in computer science from Ss. Cyril and Methodius University, Skopje, North Macedonia, where he was an Assistant Professor (2013–2017). He received the IEEE Cloud Summit best paper award in 2022.