

GPU-based Parallel Technique for Solving the N-Similarity Problem in Textual Data Mining

Mahmood Fazlali
*Department of Computer Science
Cyber Security and Computing
Research Group
University of Hertfordshire
Hertfordshire, UK
m.fazlali@herts.ac.uk*

Mina Mirhosseini
*Department of Computer and Data
Sciences
Faculty of Mathematical Sciences
Shahid Beheshti University
Tehran, Iran
mi_mirhosseini@sbu.ac.ir*

Mahyar Shahsavari
*AI Department
Donders Institute for Brain, Cognition
and Behaviour
Radboud University
Nijmegen, The Netherlands
mahyar.shahsavari@donders.ru.nl*

Alex Shafarenko
*Department of Computer Science
Cyber Security and Computing Research Group
University of Hertfordshire
Hertfordshire, UK
a.shafarenko@herts.ac.uk*

Mashallah Mashinchi
*Department of Statistics
Faculty of Mathematics and Computer
Shahid Bahonar University of Kerman
Kerman, Iran
mashinchi@uk.ac.ir*

Abstract— An important issue in data mining and information retrieval is the problem of multiple similarity or n-similarity. This problem entails finding a group of n data points with the highest similarity within a large dataset. Exact methods to solve this problem exist but come with high time and space complexities. Additionally, various metaheuristic algorithms have been proposed, including genetic algorithms, gravitational search algorithms, particle swarm optimization, imperialist competitive algorithms, and fuzzy imperialist competitive algorithms. These metaheuristics are capable of finding near-optimal solutions within a reasonable timeframe, although there is no guarantee of achieving exact results. In this paper, we employ a parallelization technique using CUDA to expedite the exact method. We conduct experiments on textual datasets to identify a group of n textual documents with the highest similarity to each other. The experimental results demonstrate that the proposed parallel exact method significantly reduces execution time compared to the best sequential approach and CPU multi-core implementation. Furthermore, it is evident that the proposed method requires less memory space than the exact method.

Keywords—multiple similarity, n-similarity, parallel programming, text document similarity

I. INTRODUCTION

The concept of similarity plays a crucial role in various applications, including clustering, classification, and diagnostic systems [1, 2, 3]. Specifically, 2-similarity, which measures the resemblance between two objects, is formally defined as a function: $2\text{-sim}: U \times U \rightarrow [0, 1]$ or $[-1, 1]$, where U represents the dataset. In this context, a 2-sim value of 1 signifies identical objects, while 0 or -1 denotes complete dissimilarity. Numerous research efforts have introduced different similarity measures, such as cosine similarity, Pearson correlation, Dice coefficient, and Jaccard index, for quantifying the similarity between two data points [4, 5]. Additionally, various fuzzy similarity measures have been proposed for handling fuzzy sets and Intuitionistic Fuzzy Sets; further details can be found in [6-11].

While 2-similarity measures focus on pairs of data points, there are scenarios where we need to assess the similarity among more than two objects and identify a group of n data points (where $n \geq 3$) with the highest similarity within a large dataset. This problem is known as n-similarity and was mathematically introduced by Keshavarzi et al. [3], with applications in classification. However, this method often entails prohibitively long execution times and high memory space requirements, rendering it impractical for large datasets.

One approach to reduce algorithm execution time and solve the n-similarity problem within a reasonable timeframe is to leverage metaheuristic algorithms. For instance, in [12], a binary genetic algorithm (BGA) was applied to address the n-similarity problem. Furthermore, [13] explored additional metaheuristics, such as the gravitational search algorithm (GSA), particle swarm optimization (PSO), imperialist competitive algorithm (ICA), and its fuzzy variant called FICA, to tackle the n-similarity problem on various datasets, including UCI datasets and Reuter's textual datasets. Comparative analysis revealed that FICA delivered the best precision results [13]. It is worth noting that while the applied metaheuristics in [12, 13] can provide 'near-optimal' solutions within a reasonable time, they do not guarantee exact or deterministic results.

Another technique to expedite the solution of certain problems is parallelization. Two common parallelization environments are OpenMP for multi-core CPUs and CUDA programming for GPUs. With the growing number of cores in modern processors, OpenMP enables efficient utilization of multiple cores to accelerate algorithms [14]. In a similar way, [15] explored the use of the OpenMP library to accelerate the solution of the n-similarity problem.

Manycore models have emerged as promising techniques in high-performance computing (HPC). Graphic processing units (GPUs), utilizing parallel computing architecture through CUDA, demonstrate the concept of general-purpose GPU (GPGPU) computing. GPUs, with their multitude of processing cores, offer a highly parallel computing environment [16].

CUDA programming is widely applied across various applications and problems to expedite execution times. For

instance, in [17], the authors employed GPGPU for perfusion imaging analysis. This approach not only maintained the quality of cerebral hemodynamic maps but also significantly accelerated the analysis compared to traditional methods. In [18], it is shown that the CUDA programming model can substantially reduce the runtime of next-generation sequencing (NGS) analysis tools, achieving a speedup of over 46 times.

Additionally, [19] implemented a novel parallel method for the breadth-first search algorithm in graph exploration, both on multi-core CPUs and GPUs, highlighting the superior performance of the hybrid approach. In [20], the authors developed a GPU-based solver for notoriously challenging problems like graph coloring and random SAT, resulting in accelerated execution times. [21] employed a hybrid parallelization technique combining CUDA and OpenMP to expedite the quadrivalent quantum-inspired gravitational search algorithm in solving wireless sensor networks (WSNs) problems. Furthermore, [22] presented the implementation of the gravitational search algorithm on GPUs, demonstrating that the proposed GPU implementation outperforms the CPU and achieves a speedup of more than 40 times. In [23], a GPU-based parallel implementation of the ant colony optimization (ACO) algorithm showed a remarkable speedup of approximately 21 times.

The aforementioned successes have motivated us to harness GPU-based parallelism to enhance the speed of the n -similarity algorithm. Keshavarzi's exact method [1] suffers from high space complexity and extended execution times. Additionally, the metaheuristic methods used in [12, 13] do not yield precise results. In Keshavarzi's exact method, thanks to the presence of data and instructions that can be processed independently, there is significant potential for parallel implementation. While in [15], the OpenMP library was employed to expedite the resolution of the n -similarity problem, we believe in the potential of GPU and CUDA programming to further increase the algorithm's speed. Therefore, our motivation is to reduce the execution time of Keshavarzi's exact method by leveraging CUDA parallelization and comparing the results with the application of OpenMP in [15]. Our method offers three key advantages over Keshavarzi's [1]:

- Faster execution time.
- Reduced memory space requirements.
- Precise solutions.

Furthermore, our proposed method boasts two advantages compared to the metaheuristics [12, 13]:

- It produces exact solutions.
- It computes the n -similarity values for all possible groups of n data points.

To evaluate the performance of our proposed parallel CUDA method, we compare its execution time with sequential implementation and the OpenMP parallelization method proposed in [15].

The remainder of this paper is organized as follows: Section II provides relevant definitions. Section III details the suggested parallel methods for solving the n -similarity problem. In Section IV, we present experimental results regarding text document resemblance. Finally, the paper concludes with Section V.

II. PRELIMINARIES

This section presents some needed definitions from [1, 12]. **Definition 1.** T-norm is a function $T: [0,1] \times [0,1] \rightarrow [0,1]$ which for all $x_1, x_2, x_3, x_4 \in [0,1]$ satisfies the following conditions:

1. Commutativity: $T(x_1, x_2) = T(x_2, x_1)$,
2. Monotonicity: $T(x_1, x_2) \leq T(x_4, x_3)$ if $x_1 \leq x_4$ and $x_2 \leq x_3$,
3. Boundary: $T(x_1, 0) = 0$, $T(x_1, 1) = x_1$,
4. Associativity: $T(x_1, T(x_2, x_4)) = T(T(x_1, x_2), x_4)$.

The minimum T-norm as $T_{min}(x_1, x_2) = \min(x_1, x_2)$ is a popular T-norms which for n -similarity definition used in [1, 2];

Definition 2. 2-similarity is defined as function $S: U \times U \rightarrow [0,1]$ on domain U , having the following properties:

1. Reflexivity: $\forall x_1 \in U, S(x_1, x_1) = 1$,
2. Symmetry: $\forall x_1, x_2 \in U, S(x_1, x_2) = S(x_2, x_1)$,
3. Transitivity: $\forall x_1, x_2, x_3 \in U, S(x_1, x_3) \geq S(x_1, x_2) \wedge S(x_2, x_3)$, in which \wedge denotes the minimum operator.

Definition 3. The 3-similarity is a function $S: U \times U \times U \rightarrow [0,1]$ meeting the following properties:

1. Reflexivity: $\forall x_1 \in U, S(x_1, x_1, x_1) = 1$,
2. $\forall x_1, x_2, x_3 \in U, S(x_1, x_2, x_3) = S(x_i, x_j, x_k)$ where i, j, k is a possible permutation of 1, 2 and 3.
3. Transitivity: $\forall x_1, x_2, x_3, p \in U, S(x_1, x_2, x_3) \geq S(p, x_2, x_3) \wedge S(x_1, p, x_3) \wedge S(x_1, x_2, p)$, where \wedge is the minimum T-norm.

Using the minimum T-norm, the 3-similarity is defined in [1] as Eq. (1) in which, S_3 denotes 3-similarity and S_2 is 2-similarity.

$$S_3(x_1, x_2, x_3) = \min(S_2(x_1, x_2), S_2(x_1, x_3), S_2(x_2, x_3)). \quad (1)$$

Definition 4. The n -similarity is defined as a function $S: U \times U \times \dots \times U \rightarrow [0,1]$ meeting these properties:

1. Reflexivity: $\forall x_i \in U, S(x_i, x_i, \dots, x_i) = 1$,
2. Symmetry: for all possible permutations (i_1, i_2, \dots, i_n) as $(1, 2, \dots, n)$ $S(x_1, x_2, \dots, x_n) = S(x_{i_1}, x_{i_2}, \dots, x_{i_n})$,
3. Transitivity: $\forall x_1, x_2, \dots, x_n, p \in U, S(x_1, x_2, \dots, x_n) \geq \min\{S(p, x_2, \dots, x_n), \dots, S(x_1, x_2, \dots, x_{n-1}, p)\}$.

The n -similarity is defined in [1] as Eq. (2), satisfying all mentioned properties. In this equation, S_{n-1} denotes the $(n-1)$ -similarity that itself satisfies all above conditions.

$$S_n(x_1, x_2, \dots, x_n) = \min(S_{n-1}(x_2, x_3, \dots, x_n), S_{n-1}(x_1, x_3, \dots, x_n), \dots, S_{n-1}(x_1, x_2, \dots, x_{n-1})). \quad (2)$$

By this way, to compute the n -similarity, it is needed to compute and save previous similarities $(n-1)$, $(n-2)$, ..., 3 and 2. So, this method has high space complexity and is unable to run for datasets with more than 100 objects. In addition, for using several nested *for* loops, this algorithm suffers from long execution time. The next section suggests how to address these problems.

III. THE PROPOSED METHOD

As it was mentioned before in Section 1, the exact method introduced in [1] suffers from high memory complexity and

the executing time; and this method is practically limited for large datasets.

In [1] for computing the n -similarity for dataset in the size of N , the space complexity would be $O(N^n)$. Because, computing the n -similarity depends on the computing and storing $(n-1)$ -similarity, $(n-2)$ -similarity, $(n-3)$ -similarity until 2-similarity, that each one respectively stores in a matrix in the size of N^{n-1} , N^{n-2} , N^{n-3} and so on. So, the required memory space to run this method is equal to $N^n + N^{n-1} + N^{n-2} + \dots + N^3 + N^2$.

To reduce the required memory to compute the n -similarity, we suggest using the following relations. In these relations, for summarizing we show the 2-similarity, 3-similarity, 4-similarity and n -similarity by S_2, S_3, S_4, S_n respectively. For example, we have from Definition 4:

$$S_4(x_1, x_2, x_3, x_4) = \min(S_3(x_1, x_2, x_3), S_3(x_1, x_2, x_4), S_3(x_1, x_3, x_4), S_3(x_2, x_3, x_4)) \quad (3)$$

We replace S_3 (Eq. (1)), to this relation, and subsequently S_4 would be achieved in terms of S_2 as Eq. (4).

$$S_4(x_1, x_2, x_3, x_4) = \min\left(\begin{matrix} S_2(x_1, x_2), S_2(x_1, x_3), S_2(x_1, x_4), S_2(x_2, x_3), \\ S_2(x_2, x_4), S_2(x_3, x_4) \end{matrix}\right). \quad (4)$$

Similarly, S_n would be achieved as Eq. (5) just in terms of S_2 compared with Eq. (2). Therefore, by this way, S_n can be computed just using 2-similarity matrix in the size of N^2 , and despite [1], it is not needed to compute $(n-1)$ -, $(n-2)$ -, ..., 3-similarities. So, the required memory would be $N^2 + N^n$.

$$S_n(x_1, x_2, \dots, x_n) = \min(S_2(x_1, x_2), S_2(x_1, x_3), \dots, S_2(x_{n-1}, x_n)). \quad (5)$$

Moreover, to additionally decrease the execution time of the algorithms, we use CUDA based parallelism technique or GPU implementation.

In CUDA programming, the problem is broken into many sub-problems which are solved in parallel by blocks of threads. The CUDA instructions are organized in functions named kernel which are run cooperatively by threads of a block. In GPU, there is an array of streaming multiprocessors or SMs which can run several blocks simultaneously. Kernels can be involved a 2-dimensional grid of blocks that each block has a unique pair of indices. Also, threads of a block can be 2- or 3-dimensional with a unique set of indices [21].

Algorithm I presents this proposed implementation in which $size3$ is the size of dataset, $nblocks$ represents the size of 3-similarity matrix, $size3$ represents the size of 3-similarity matrix, variable $nblocks$ is the number of blocks, and $nthreads$ is the number of threads. In this algorithm, three kernels have been used. The first one is Calc-3-Sim which gets a GPU device pointer to a matrix $N \times N$ called 2-similarity containing the similarity value between all pair-waists; and stores the result in a GPU device pointer matrix 3-similarity in the size of $N \times N \times N$. The grid and block configuration of this kernel is presented in this algorithm.

In this kernel, we use multi-dimensional blocks. The first thing to do for that is determination of the x and y axis indexes like row and col numbers. Then, to prevent unnecessary threads from operating, the if condition checks the row and col does not exceed from their bounds. In fact, this kernel assigns one thread to calculate one cell of matrix 3-similarity. Each thread loads from global memory, one row and

one column of matrix 2-similarity and compute and store the 3-similarity value using a for loop and Eq. (1).

Algorithm I. The pseudo codes of computing the maximum of 3-similarities using CUDA

```

Input: 2-sim Matrix
Output: MAX
__global__ Calc-3-Sim(2-sim,3-sim)
{
row=blockIdx.y*blockDim.y+threadIdx.y;
col=blockIdx.x*blockDim.x+threadIdx.x;
if(row<size && col<size && col>row)
for i = col+1 to size
3-sim(row,col,i)=min(2-sim(row,col),2-sim(col,i),2-sim(row,i));
}
__global__ MaxOfEachBlock(size3,global_BlockMax, 3-sim)
{
__shared__ double shared_ThreadResult[nthreads];
globalThreadId = blockIdx.x * blockDim.x + threadIdx.x;
localThreadId = threadIdx.x;
blockId = blockIdx.x;
start=globalThreadId*(size3/(block_num*thread_num));
end=(globalThreadId+1)*(size3/(block_num*thread_num));
for i = start To end
if(3-sim[i]>shared_ThreadResult[localThreadId])
shared_ThreadResult[localThreadId] = 3-sim[i];
__syncthreads();
for i = blockDim.x/2 To 0 i=i/2
{
if (localThreadId < i)
if(shared_ThreadResult[localThreadId]<shared_ThreadResult[localThreadId+i])
shared_ThreadResult[localThreadId]=shared_ThreadResult[localThreadId+i];
__syncthreads();
}
if(localThreadId == 0)
global_BlockMax[blockId] = shared_ThreadResult[0];
}
__global__ MaxOfBlockResults(MAX, global_BlockMax)
{
localThreadId = threadIdx.x;
for i = blockDim.x/2 To i > 0 i=i/2
{
if (localThreadId < i)
if(global_BlockMax[localThreadId]<global_BlockMax[localThreadId+i])
global_BlockMax[localThreadId]=global_BlockMax[localThreadId+i];
__syncthreads();
}
if(localThreadId == 0)
MAX=global_BlockMax[0];
}
main()
{
read 2-sim matrix
grid_rows=(size+BLOCK_SIZE - 1)/BLOCK_SIZE;
grid_cols = (size+ BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 dimGrid(grid_cols, grid_rows);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

Calc-3-Sim<<<dimGrid,dimBlock>>>(2-sim, 3-sim);
MaxOfEachBlock<<<nblocks, nthreads>>>(size3, global_BlockMax,
sim_3);
max_block_results<<<1,nblocks>>>(MAX, global_BlockMax);
}

```

Then, the 3-similarity matrix is given as input to the next kernel to compute the maximum. The output of this kernel is $global_BlockMax$ in which the block results are stored. The size of $global_BlockMax$ is equal to number of blocks. Finding the maximum of threads of each block is performed on shared memory. In fact, for each block, maximum of threads results is computed on shared memory and then block result is transferred on the global memory.

Each thread has a global and local ID, respectively as $\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$ and threadIdx.x . Each thread computes its start and stop indexes in order to divide the job between threads. The variables of *start* and *stop* are computed as $\text{globalThreadId} \times (\text{size3}/(\text{block_num} \times \text{thread_num}))$ and $(\text{globalThreadId} + 1) \times (\text{size3}/(\text{block_num} \times \text{thread_num}))$ in which *size3* is the size of 3-sim matrix. In other words, the total number of threads is $\text{block_num} \times \text{thread_num}$ and each global thread executes $\text{step_num}/(\text{block_num} \times \text{thread_num})$ steps. All threads find the maximum of $3 - \text{sim}[i]$ in which *i* is between its *start* and *stop* indexes and store their maximum on *shared_ThreadResult[local_ThreadId]*. The `__syncthreads()` instruction is caused to wait for all threads to finish their own calculation.

Then, for each block, the maximum of array *shared_ThreadResult[0...nthreads - 1]* is calculated using binary reduction method, and after synchronization, their maximum is finally stored in *shared_ThreadResult[0]*. The binary reduction method is done in $\log(\text{nthreads})$ stages. *shared_ThreadResult[0]* as the found maximum of each block is stored on global memory by *global_BlockMax[blockId]*, to calculate the global maximum or the maximum of block results in the next kernel.

The next kernel namely *MaxOfBlockResults* gets *global_BlockMax* array as input containing the found maximum inside of each block and find their maximum using the binary reduction method in $\log(\text{nblocks})$ step and transfer the global maximum on variable *MAX*.

In the main function, the $2 - \text{sim}$ matrix is read from the file and *Calc - 3 - Sim* is called by a 2D grid and 2D blocks to calculate all 3-similarities. The size of block is defined as *BLOCK_SIZE* variable and set to 16 in both dimension. Also the grid row and column size is set as $(\text{size} + \text{BLOCK_SIZE} - 1)/\text{BLOCK_SIZE}$ in which *size* is the size of dataset.

Then, the kernel *MaxOfEachBlock* is called to find the maximum inside each block. The size of grid and block is respectively set as *nblocks* = 64 and *nthreads* = 256. Finally, *max_block_results* kernel is run to find the maximum of block results, as the final output *MAX*. This kernel runs by 1 block involving *nblocks* = 64 threads. In fact, the number of threads in this kernel is equal to the number of blocks of previous kernel.

Algorithm II. The pseudo codes of a kernel to compute the 4-similarity values

```

__global__ Calc-4-Sim(2-sim,4-sim)
{
row=blockIdx.y*blockDim.y+threadIdx.y;
col=blockIdx.x*blockDim.x+threadIdx.x;
if(row<size && col<size && col>row)
for i = col+1 to size
for j=i+1 to size
4-sim(row,col,i,j)=min(2-sim(row,col),2-sim(row,i),2-sim(row,j), 2-
sim(col,i), 2-sim(col,j), 2-sim(i,j));
}

```

The computation of 4-similarity is as the same as Algorithm I, except in the case of first kernel. This kernel is rewritten as Algorithm II. The 4-similarity is computed as Eq. (4), so, in this kernel the 4-similarity is achieved by taking the minimum of 2-similarity of all six possible pair waist of 4 objects. In the case of problems needed a space larger than

the global memory, the problem can be divided into smaller sub-problems.

IV. EXPERIMENTAL RESULTS

In this section, we conduct some experiments on textual documents to find *n* documents among a dataset in the size of *N*, in such away they have the highest similarity to each other. In the case of textual datasets, some preprocessing [24] are needed to do, to reach the 2-similarity matrix. Then, by applying this matrix, the 3-similarity, 4-similarity, etc. can be achieved. The main steps of preprocessing are as follows.

Step 1- The firstly, all symbols and digits are eliminated and the strings between *space*, *.*, *:*, *;*, *-*, *?*, *!*, *(*, *)*, *[*, *]* etc. are tokened as a word.

Step 2- All stop-words including *a*, *the*, *are*, *is*, *do*, etc. are deleted applying a list called “*Weka machine learning workbench*” [25] composed of 527 stop-words.

Step 3- In this step, stemming is performed to map the derivationally related forms of a word to a stem as a common base form. In this purpose, the *Porter’s suffix-stripping algorithm* [26] is used.

Step 4- Then, a numerical weight is assigned to each word. Here the TFIDF weighting relation [24] has been used.

Step 5- This step removes the words with the weight less than a threshold.

Step 6- The similarity value between all pair-waist documents is computed by a similarity measure. The most popular similarity measure for textual documents is Cosine coefficient defined in [27]. This step generates the 2-similarity matrix which gives as input to 3-similarity, 4-similarity or *n*-similarity algorithms.

The experiments of this research are performed on selected documents of Re0 dataset extracted from the Reuters repository [28]. This dataset is composed of 31 classes, 1,504 newspaper articles and 2,886 keywords.

The sequential and multi core simulations are performed on a cluster with 50 cores and 128 GB RAM. Our CUDA parallel design is developed on GeForce GTX 1080 Ti with 3584 cores, 11GB RAM and 50KB shared memory per block.

The simulation results reports the execution time and the speedup of sequential implementation, OpenMP [15] and CUDA based parallelism techniques for 3-similarity and 4-similarity as Tables I to IV. In these tables, T_{seq} represents the running time of sequential implementation, T_4 , T_8 , T_{16} , T_{32} and T_{50} show the execution time of OpenMP [15] penalization using 4, 8, 16, 32 and 50 cores respectively and T_{CUDA} is the execution time of CUDA implementation.

Moreover, S_4 , S_8 , S_{16} , S_{32} and S_{50} are the achieved speedup using parallel OpenMP based technique [15] by 4, 8,16, 32 and 50 cores respectively and S_{CUDA} represents the speedup applying GPU and CUDA programming. The column *size* shows the size of selected documents among datasets. In this method for using the binary reduction, the size of dataset would be better to be power of two as 128, 256, 512 and 1024. The last row shows the average result of each implementation for different size of dataset. The reported times are in terms of second, and the results are averaged over 5 independent runs.

Table I presents the execution time for 3-similarity problem by the proposed parallel CUDA-based method in comparison

with the sequential and multi-core implementation by different number of cores as 4, 8, 16, 32 and 50. It shows that the execution time for 128 documents is decreased from 0.0123 to 0.0025 seconds by changing the number of cores from 1 to 32, and the execution time is increased by 50 cores to 0.0057. It shows that there is a saturation point between 32 and 50 for this execution. The execution time of CUDA implementation is 0.00044. In this table, for dataset in the size of 256, we have a decrement from 0.06549 to 0.00232 in execution time for sequential to CUDA implementation. Similarly, for dataset consisting of 512 documents, the running time regularly reduces from 0.38936 seconds to 0.03895 seconds by changing the number of cores from 1 to 50 and decreased to 0.02163 seconds for CUDA parallelization. This behavior is repeated for dataset in the size of 1024, in which the execution time is decreased from 2.78661 to 0.1968 from sequential to CUDA implementation. The last row shows the average of results on different size of dataset. The average execution time for sequential run is 0.81344 and for CUDA implementation, it is averagely reduced to 0.05529. Table I clearly illustrates the decreasing execution time using CUDA based parallelism in comparison with sequential and multi core implementation.

TABLE I. COMPARISON OF THE EXECUTION TIME (SEC.) USING OPENMP [15] AND CUDA BASED PARALLELISM ON 3-SIMILARITY PROBLEM

Size	$T_{seq.}$	T_4	T_8	T_{16}	T_{32}	T_{50}	T_{CUDA}
128	0.0123	0.0088	0.0051	0.0034	0.0025	0.0057	0.00044
256	0.06549	0.03985	0.02468	0.01779	0.01100	0.01028	0.00232
512	0.38936	0.23520	0.15246	0.08636	0.04658	0.03895	0.02163
1024	2.78661	1.61311	0.933686	0.52286	0.30079	0.23062	0.1968
Avg.	0.81344	0.47424	0.27898	0.15760	0.09021	0.07138	0.05529

TABLE II. COMPARISON OF SPEEDUP USING OPENMP [15] AND CUDA BASED PARALLELISM ON 3-SIMILARITY PROBLEM

Size	S_4	S_8	S_{16}	S_{32}	S_{50}	S_{CUDA}
128	1.3963	2.3995	3.5571	4.8781	2.1672	27.9545
256	1.6432	2.6529	3.6810	5.9520	6.3706	28.2284
512	1.6554	2.5539	4.5085	8.3589	9.9966	18.0009
1024	1.7274	2.9845	5.3295	9.2641	12.0831	14.1596
Avg.	1.6056	2.6477	4.2690	7.1132	7.6543	22.0858

Table II shows the speedup for 3-similarity using the proposed method by CUDA (S_{CUDA}) and OpenMP [15] implementation for different size of Re0 dataset from 128 to 1024 by changing the number of cores from 4 to 50 (showing by S_4 to S_{50}).

As it can be seen in Table II, for dataset with the size of 128, the speedup is increased from 1.3963 to 27.9545 for multi core implementation by 4 cores and CUDA parallelization. Also, for dataset composed of 256 documents, the speedup is averagely increased from 1.6432 to 28.2284. In the case of 512 documents, the speedup grows from 1.6554 to 18.0009 times. Also, for dataset consisting of 1024 documents, the speedup starts from 1.7274 using 4 CPU cores and reaches to 14.1596 using the proposed CUDA program. The last row of this table shows that the averaged speedup for different size

of dataset. The averaged speedup for using 4, 8, 16, 32 and 50 CPU cores is 1.6056, 2.6477, 4.2690, 7.1132, 7.6543 respectively, and is 22.0858 times using the proposed CUDA implementation. This table clearly shows the effectiveness of the proposed CUDA implementation in accelerate solving 3-similarity problem.

Table III reports the execution time for 4-similarity problem by the proposed CUDA parallelization compared with the sequential and OpenMP implementation using different number of cores. It is observable that the execution time for 128 documents is decreased from 0.3001 to 0.0203 seconds from sequential to CUDA implementation. Also, for dataset in the size of 256, the running time reduces from 4.2894 to 0.2421 for sequential and CUDA implementation. In addition, we have a reduction in running time from 65.838 seconds to 4.0138 in the case of 512 documents and from 1077.262 seconds to 72.2119 seconds for dataset in the size of 1024. The row Avg. shows the averaged execution time on various size of dataset. The average running time for sequential code is 286.9223 and for CUDA program is 19.1220. In fact, Table III clearly shows the decrement of running time using CUDA compared with sequential and multi core implementation.

TABLE III. COMPARISON OF THE EXECUTION TIME (SEC.) USING OPENMP [15] AND CUDA BASED PARALLELISM ON 4-SIMILARITY PROBLEM

Size	$T_{seq.}$	T_4	T_8	T_{16}	T_{32}	T_{50}	T_{CUDA}
128	0.3001	0.2187	0.1462	0.0758	0.0390	0.0258	0.0203
256	4.2894	2.7831	1.7774	0.9331	0.4851	0.3101	0.2421
512	65.838	44.5858	27.8171	14.9167	7.8976	5.1297	4.0138
1024	1077.262	733.357	441.7463	247.8341	133.1242	83.8182	72.2119
Avg.	286.9223	195.2361	117.8717	65.9399	35.3864	22.3209	19.1220

TABLE IV. COMPARISON OF SPEEDUP USING OPENMP [15] AND CUDA BASED PARALLELISM ON 4-SIMILARITY PROBLEM

Size	S_4	S_8	S_{16}	S_{32}	S_{50}	S_{CUDA}
128	1.3718	2.0515	3.9572	7.6894	14.8341	14.7832
256	1.5412	2.4133	4.5968	8.8423	13.8285	17.7174
512	1.4766	2.3668	4.4137	8.3363	12.8346	16.4029
1024	1.4689	2.4386	4.3467	8.0921	12.8523	14.9180
Avg.	1.4646	2.3175	4.3286	8.2893	13.5873	15.9554

Table IV gives the speedup for 4-similarity problem using the proposed CUDA implementation denoted by S_{CUDA} and multi core CPU implementation with different number of cores denoted by S_4 , S_8 , S_{16} , S_{32} and S_{50} and various sizes of dataset.

As Table IV shows, for dataset with the size of 128, the speedup is increased from 1.3718 to 14.7832 and for dataset in the size of 256; the speedup is averagely grows d from 1.5412 to 17.7174. Also, for dataset composing of 512 documents, the speedup is improved from 1.4766 to 16.4029 times. In the case of 1024 documents, the speedup for 4 CPU cores is 1.4689 and reaches to 14.9180 times by CUDA programming. The average row shows the averaged speedup on various size of dataset. The averaged speedup for using OpenMP is 1.4646 for 4 cores, 2.3175 using 8 cores, 4.3286 in the case of 16 cores, 8.2893 for 32 cores, and 13.5873 using

50 cores. Also, the speedup is 15.9554 times using the proposed CUDA parallelization. This table indicates that the proposed CUDA implementation is able to accelerate solving 4-similarity problem more, in comparison with other implementations.

Fig. 1 illustrates the average speedup for OpenMP by 4, 8, 16, 32 and 50 cores and the proposed CUDA parallelization for 3-similarity and 4-similarity problems. An increasing trend is clearly observable. This means in these problems generally by increasing the number of CPU cores, the speedup is increased, also, the CUDA implementation give the best speedup.

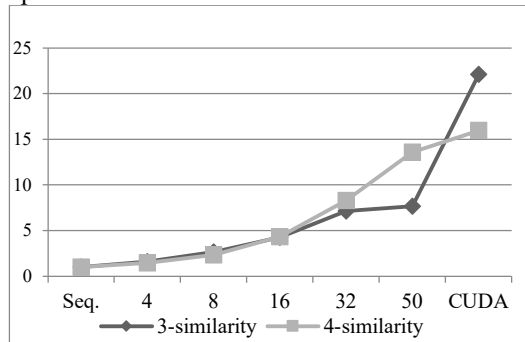


Fig 1. The averaged speedup of for 3-similarity and 4-similarity.

V. CONCLUSIONS

In this paper, we have introduced a parallelized version of an exact method for solving the n-similarity problem within a practically feasible timeframe. Furthermore, our proposed method requires less memory space compared to the previous exact approach. To validate our approach, we conducted experiments using a dataset consisting of 1,504 newspaper articles, 2,886 keywords, and 31 classes, focusing on text document similarity. Our results were obtained by varying the dataset size and the number of textual documents used for both 3-similarity and 4-similarity problems.

The results indicate that our CUDA-based parallelization achieves an average speedup of 22.0858X for 3-similarity problems and 15.9554X for 4-similarity problems. In future work, we plan to further optimize memory usage and explore opportunities for enhancing speed even more.

REFERENCES

- [1] M. Keshavarzi, M. A. Dehghan, M. Mashinchi, Applications of classification based on similarities and dissimilarities, *Fuzzy Information and Engineering*, Vol. 4, No. 1, pp. 75-92, 2012.
- [2] M. Keshavarzi, M. A. Dehghan, M. Mashinchi, Classification based on 3-similarity, *Iranian Journal of Mathematical Sciences and Informatics*, Vol. 6, No. 1, pp 7-21, 2011.
- [3] M. Keshavarzi, Classification based on similarity and dissimilarity, PhD thesis, Shahid Bahonar University of Kerman, Iran, 2010.
- [4] S. Theodoridis, K. Koutroumbas, *Pattern recognition*, Academic Press, 2008, ISBN: 9780080949123.
- [5] L. Kaufman, P. J. Rousseeuw, *Finding Group in Data*, Wiley, New York, 1990, ISBN:9780470316801.
- [6] W. J. Wang, New similarity measure on fuzzy sets and their applications, *Mathematical and Computer Modeling*, Vol. 53, pp. 91-97, 2011.
- [7] H. Rezaei, M. Emoto, M. Mukaidono, New similarity measure between two fuzzy sets, *Journal of Advanced Computational Intelligence and Intelligent Informatics*, Vol. 10, No. 6, pp. 946-953, 2006.
- [8] J. Ye, Cosine similarity measures for intuitionistic fuzzy sets and their applications, *Mathematical and Computer Modeling*, Vol. 53, pp. 91-97, 2011.

- [9] G. Coletti, B. Bouchon-Meunier, Fuzzy similarity measures and measurement theory, *2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, New Orleans, LA, USA, 2019, pp. 1-7.
- [10] R. Verma, A. Mittal, Multiple attribute group decision-making based on novel probabilistic ordered weighted cosine similarity operators with Pythagorean fuzzy information, *Granular Computing*, Vol. 8, pp. 111-129, 2023.
- [11] M. Kirişci, New cosine similarity and distance measures for Fermatean fuzzy sets and TOPSIS approach. *Knowledge and Information Systems*, Vol. 65, no. 2, pp. 855-68, 2023.
- [12] M. Mirhoseini, M. Mashinchi, H. Nezamabadi-pour, Improving n-Similarity problem by genetic algorithm and its application in text document resemblance, *Fuzzy Information and Engineering*, Vol. 6, pp. 263-278, 2014.
- [13] M. Mirhoseini, H. Nezamabadi-pour, Metaheuristic Search Algorithms in Solving the n-Similarity Problem, *Fundamenta Informaticae*, vol. 152, no. 2, pp. 145-166, 2017.
- [14] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling Parallel Real-Time Tasks on Multi-core Processors, *2010 31st IEEE Real-Time Systems Symposium*, San Diego, CA, 2010, pp. 259-268.
- [15] M. Mirhoseini, M. Fazlali, Parallel and Exact Method for Solving n-Similarity Problem, *Journal of Electrical and Computer Engineering Innovations (JECIEI)*, vol. 8, no. 2, pp.193-200, 2020.
- [16] M. Khairy, A. G. Wassal, M. Zahran, A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity, *Journal of Parallel and Distributed Computing*, vol. 127, pp. 65-88, 2019.
- [17] F. Zhu, D. R. Gonzalez, T. Carpenter, M. Atkinson, J. Wardlaw, Parallel perfusion imaging processing using GPGPU, *Computer Methods and Programs in Biomedicine*, vol. 108, no. 3, pp. 1012-1021, 2012.
- [18] J. Chae Na, I. Lee, J-K Rhee, S-Y Shin, Fast single individual haplotyping method using GPGPU, *Computers in Biology and Medicine*, vol. 113, 2019.
- [19] S. Hong, T. Oguntebi, K. Olukotun, Efficient Parallel Graph Exploration on Multi-Core CPU and GPU, *2011 International Conference on Parallel Architectures and Compilation Techniques*, Galveston, TX, pp. 78-88, 2011.
- [20] F. Molnár, Sh. R. Kharel, X. Sh. Hu, Z. Toroczka, Accelerating a continuous-time analog SAT solver using GPUs, *Computer Physics Communications*, vol. 256, 2020.
- [21] M. Mirhoseini, M. Fazlali, G. Gaydadjiev, A Parallel and Improved Quadrivalent Quantum-Inspired Gravitational Search Algorithm in Optimal Design of WSNs, High-Performance Computing and Big Data Analysis. *TopHPC 2019. Communications in Computer and Information Science*, vol. 891, Springer, Cham, pp. 352-366, 2019.
- [22] A. Zarrabi, Kh. Samsudin, E. K. Karuppiah, Gravitational search algorithm using CUDA: a case study in high-performance metaheuristics, *The Journal of Supercomputing*, vol. 71, pp. 1277-1296, 2015.
- [23] R. Skinderowicz, Implementing a GPU-based parallel MAX-MIN Ant System, *Future Generation Computer Systems*, vol. 106, pp. 277-295, 2020.
- [24] Z. Rahimi, M. M. Homayounpour, The impact of preprocessing on word embedding quality: A comparative study. *Language Resources and Evaluation*, Vol. 57, no. 1, pp. 257-291, 2023.
- [25] <http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop>.
- [26] M. F. Porter, An algorithm for suffix stripping, *Program*, Vol. 14, No. 3, pp. 130-137, 1980.
- [27] C. Qimin, G. Qiao, W. Yongliang, W. Xianghu, Text clustering using VSM with feature clusters, *Neural Computing and Applications*, Vol 26, pp. 995- 1003, 2015.
- [28] D. D. Lewis, Reuters-21578 text categorization test collection distribution 1.0, <http://www.Research.att.com/lewis/reuters21578.html>, 1999.