BAREKENG: Journal of Mathematics and Its Applications
September 2025 Volume 19 Issue 3 Page 1725-1736
P-ISSN: 1978-7227 E-ISSN: 2615-3017
https://doi.org/10.30598/barekengvol19iss3pp1725-1736

APPLICATION OF GPU-CUDA PARALLEL COMPUTING TO THE SMITH-WATERMAN ALGORITHM TO DETECT MUSIC PLAGIARISM

Alfredo Gormantara[1*], Ferdianto Tangdililing[2], Sean Coonery Sumarta[3]

[1,3]*Informatics Engineering Study Program, Department of Information Technology, Universitas Atma Jaya Makassar*
[2]*Electrical Engineering Study Program, Department of Engineering, Universitas Atma Jaya Makassar*
Jln. Tanjung Alang No 23, Makassar, 90134, Indonesia

*Corresponding author's e-mail: * **alfredo_gormantara@lecturer.uajm.ac.id***

Article History:

Received: 15th January 2025
Revised: 10th February 2025
Accepted: 18th March 2025
Published: 1st July 2025

Keywords:

GPU-CUDA;
Music plagiarism;
Parallel Computing;
Smith-Waterman.

ABSTRACT

This study introduces the Smith-Waterman algorithm because the advantage of this algorithm is that it can determine the similarity from any position that corresponds to music plagiarism, considering that song similarities can occur in any part of a song. Plagiarism detection can be done by comparing the melody notes of 2 songs to determine whether or not there are similarities. Songs that are identified as plagiarism have similar melodies of 8 bars. However, the Smith-Waterman algorithm has a weakness, namely that the speed of this algorithm is relatively slow, so parallel computing is required to speed up the detection process. Parallel computing relies on the capabilities of multi-core GPUs that can be programmed using CUDA. Therefore, the innovation raised in this study is to speed up the computing process in detecting music plagiarism by applying parallel computing to the Smith-Waterman algorithm. The methodology stages begin with melody extraction, namely taking the song melody from the MIDI file along with the melody's tempo in the MIDI file and then transposing it to the basic tone of C. The study's results showed that using the GPU can speed up the execution time by up to 5.7 times compared to using the CPU. In addition, validation was carried out with real music plagiarism cases and validation of the results using the MIPPIA website. This shows that parallel computing has been successfully applied to the Smith-Waterman algorithm in detecting music plagiarism.

How to cite this article:

A. Gormantara, F. Tangdililing and S. C. Sumarta., "APPLICATION OF GPU-CUDA PARALLEL COMPUTING TO THE SMITH-WATERMAN ALGORITHM TO DETECT MUSIC PLAGIARISM," *BAREKENG: J. Math. & App.*, vol. 19, no. 3, pp. 1725-1736, September, 2025.

Journal homepage: *https://ojs3.unpatti.ac.id/index.php/barekeng/*
Journal e-mail: *barekeng.math@yahoo.com; barekeng.journal@mail.unpatti.ac.id*

Research Article · Open Access

1. INTRODUCTION

Parallel computing is a method that can speed up the computing process [1]. With parallel computing, the problem will be divided among several existing sources so that the problem can be solved simultaneously and the process of working on the problem can be resolved quickly [2]. Parallel computing involves dividing a larger problem into smaller, independent, and often similar components that can be executed simultaneously by multiple processors communicating through shared memory. The results of these executions are then combined upon completion as part of the overall algorithm.

As the need for parallel computing in various scientific and industrial applications increases, the limitations of CPUs in handling complex parallel tasks become increasingly apparent. Traditional CPUs are designed for sequential task execution and have a limited number of cores, making them less efficient in handling workloads that require high parallelization [3]. As a solution, the use of Graphics Processing Units (GPUs) has grown rapidly due to their architecture that supports massive parallel execution [4]. Originally designed for graphics rendering, GPUs are now widely used in general computing due to their ability to process thousands of threads simultaneously, enabling significant speedups in applications such as machine learning, scientific simulations, and big data analysis.

Parallel computing relies on the multi-core capabilities of the Graphic Processing Unit (GPU), where the GPU can be programmed via CUDA (Compute Unified Device Architecture). Using CUDA, GPU algorithms can perform better than multi-core Central Processing Units (CPU) [5]. Implementations of various Jacobi solver variants on GPUs using CUDA have been compared to speedups of up to 60 times over sequential implementations on CPUs [6]. This research [7] discusses the parallel simulation of pattern formation in a reaction-diffusion system of Fitzhugh-Nagumo Using GPU CUDA, which can speed up computing time by 12x. In addition, other research [8] shows that the use of GPU-CUDA is 7.34x faster than the use of OpenCL. Even in this research [9], optimizing the use of GPU-CUDA in conducting experiments using extensive data obtained results 30x faster than the previous process. The use of GPUs with CUDA in accelerating computer vision tasks has shown significant improvements in efficiency and speed, with some implementations achieving speedups of up to 8061 times over conventional methods [10].

Some research on song plagiarism [11] speeds up 3x the Smith-Waterman algorithm using GPUs; even in research [12], it can be up to 24x faster using GPUs. The primary objective of parallel computing is to enhance available computing power, thereby speeding up application processing and problem-solving [13]. Parallel computing is often used for problems or algorithms that require quite a long processing or computing time, such as machine learning [14], long-short term memory [15], deep learning [16] and smith-waterman [17].

The Smith-Waterman algorithm is a dynamic programming algorithm where dynamic programming can take similarities of any length in any order and determine whether optimal similarity can be achieved. The reason why this research uses the Smith-Waterman algorithm is that the advantage of this algorithm is that it can evaluate similarities from any position that matches music plagiarism, considering that song similarities can occur in any part of a song [18].

Research on the Smith-Waterman algorithm has been carried out in several case studies. Research [19] used the Smith-Waterman algorithm to detect Indonesian documents with more than 50% similarity results, which means that the results can be declared as plagiarism. Some compare document text with paraphrases using Smith-Waterman [20] and intelligent chatbot applications [21]. Meanwhile, music plagiarism cases have been studied using the Multi-MMLG feature by comparing MIDI files from a song [22].

One of the elements of a song is music, where a song is music that contains lyrics so it can be sung, where the presence of music will produce a song that is pleasant to listen to. The music has elements, namely melody, rhythm, tempo, etc [23]. herefore, this research was carried out by detecting song similarities from a melody contained in music where the music is one of the elements of a song.

Plagiarism detection can be done by comparing the melody notes of 2 songs to determine whether there are similarities. Songs identified as plagiarized have a melody similarity of 8 bars [24]. Also, plagiarism detection should have a feature to tell which parts of the song have been identified as plagiarized to make it easier for producers to change the melody. However, it is not only seen from how similar the 8-bar melody is; songs that are identified as plagiarism can also be seen from the 15-second similarity of the song. If the song's similarity is more or equal to 15 seconds, then it can be identified as plagiarism [25].

Identifying similarities in melodies or music can help song producers avoid the criminal law of plagiarism under the Copyright Law (UUHC) [26]. This is also based on the results of interviews conducted by researchers with a song producer, where plagiarism detection is essential because other people can steal expensive ideas from a producer and can help producers who want to release songs on digital platforms avoid plagiarism laws. The Smith-Waterman algorithm has a weakness, namely that the speed of this algorithm is relatively low, so parallel computing methods are required to speed up the detection process using the Smith-Waterman algorithm [27].

Therefore, this study discusses accelerating the Smith-Waterman algorithm in detecting music plagiarism. Applying parallel computing to detect music plagiarism in a song can accelerate the Smith-Waterman algorithm process. The novelty of this study is the use of the Smith-Waterman algorithm in a case study of music plagiarism accelerated using GPU-CUDA.

2. RESEARCH METHODS

This section discusses the methodology and research flow carried out during this research, as shown in Figure 1, which consists of four stages: data collection, data pre-processing, building code for the CPU and GPU, and analyzing the results.

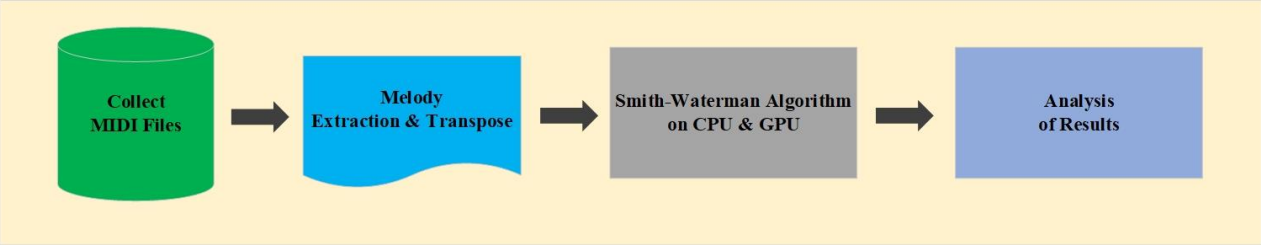

Figure 1. Research Flow

2.1. Data Collection

At this stage, data is collected through literature study and interviews. The interview process was carried out so that researchers could find out what music plagiarism means and what producers need to determine plagiarism in a song. Song plagiarism can be identified from the similarity of melody between songs. Not only that, but the similarity of the melody, among other things, must also be above 8 bars or more than 15 seconds in a row. In detecting songs, it is also best to have a feature that can tell you where the similarities between songs occur, whether at the beginning or in the middle, etc. So that producers can more easily find out and change the melody of songs that are identified as plagiarism.

Apart from that, MIDI file data was also collected with 8.000–10.000 files sourced from https://colinraffel.com/projects/lmd/ and https://bitmidi.com/. Apart from MIDI files, other data is also needed in the form of a song title, songwriter, song tempo, and song basic tone. The title of a song is needed as a sign of what song is being entered into the database or what song is being input for testing. The songwriter is the data entered as a marker of the song's identity. The song's tempo is used as input data because converting melody extraction into a MIDI file requires the song's tempo. The basic tone entered is used to transpose the basic tone to a predetermined basic tone to overcome the failure of song plagiarism detection if the basic tone is different.

2.2. Pre-Processing Data

This stage takes the song melody from a MIDI file along with the time at which the melody is located in the MIDI file. In this process, a library from Python called Mido is used to read MIDI files and extract the melody and the time of the melody location in seconds. On **Figure 2**, melody extraction simulations automatically take a sequence of frequency values according to the dominant melody line of a song where more than two notes can sound simultaneously (polyphonic music). In this function, the results of melody extraction are stored in a dictionary data type called data_per_track, which contains several notes/melodies, melody times, and melodies per MIDI track velocity. The results of notes/melodies in the MIDI library are stored as a MIDI number. Next, transpose the MIDI song melody to the basic note C so that all songs in the database have the same basic note. This is done by looking for the index of the basic note other than C in the notes_original variable. After getting the index of the basic tone of the song, the index will be considered as semitones.

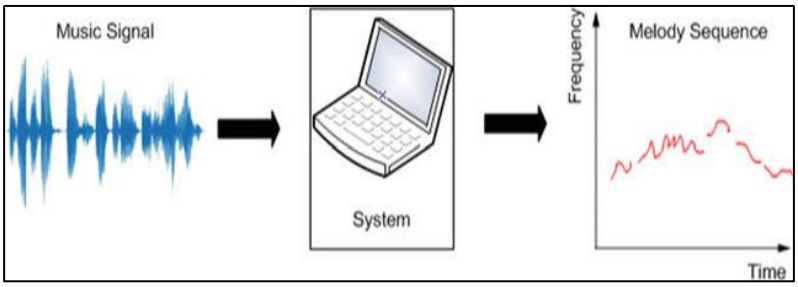

Figure 2. Melody Extraction Process

Semitones are intervals of a note or melody where the interval is only half of the melody (example: D to C has 2 semitones, or D to C# has 1 semitone). After getting the semitones from the basic note of the song, the MIDI number will be subtracted by semitones to get the MIDI number that has been transposed. In this research, the MIDI number that has been transposed will be the basic note C.

```
def get_notes_from_midi(midi_file_path, bpm):
    data_per_track = {}
    mid = MidiFile(midi_file_path)
    ticks_per_beat = mid.ticks_per_beat
    for i, track in enumerate(mid.tracks):
        notes = []
        current_time = 0  # Default tempo in microseconds per beat
        for msg in track:
            current_time += msg.time
            if msg.type == 'set_tempo':
                tempo = msg.tempo
                 # Default tempo in microseconds per beat
            if msg.type == 'note_on':
                if msg.velocity > 0:  # Note is played (velocity > 0)
                    try:
                        time_seconds = mido.tick2second(current_time, ticks_per_beat, tempo)
                    except:
                        tempo = (60000 // int(bpm)) * 1000
                        time_seconds = mido.tick2second(current_time, ticks_per_beat, tempo)
                    notes.append({
                        'note': msg.note,
                        'velocity': msg.velocity,
                        'time': time_seconds
                    })
        data_per_track[f"Track {i + 1}"] = {
            'notes': notes
        }
    return data_per_track
```

Based on the code above, get_notes_from_midi function reads the MIDI file and extracts the pitch information from each track, including the note (the note played), velocity (the intensity of the note), and time (the time in seconds). This function converts the time from ticks to seconds using the tempo from the

MIDI file or the default BPM if the tempo is unavailable. Each track is analyzed by summing the time of each MIDI message and storing the notes with a velocity > 0. The extracted data is organized into a dictionary that groups the notes by track, allowing for musical pattern analysis or melody matching, for example, in music plagiarism detection.

Next, the song information is retrieved or extracted from the midi name by the format artist_name – song_title – song_release_date – basic_nada – bpm_lagu.mid. Finally, all information is stored in the database in dictionary form. This is because MongoDB is a NoSQL database which stores data in key and value form so a dictionary data type is needed before entering data in the database in **Table 1**.

Table 1. Data Dictionary

Key	Description
artist	Name of the artist or singer of the song
title	Song title
release	Song release date
key	The basic chords of the song
bpm	Song tempo
pitches_before_transpose	The pitch number of the song before transposing
pitches_after_transpose	The pitch number of the song after transposing
transpose_notes	Song notes after transposing
notes_before_transpose	Song notes before transposing
times	The location of the song notes is in the song in the form of seconds
length_notes	The total length of the notes in the song
file	Location of MIDI files

2.3. CPU vs GPU

This stage builds program code to detect music plagiarism using the Smith-Waterman algorithm in the Python language.

$$H_{ij} = \max \begin{cases} H_i - 1, \mathrm{j} - 1 + \mathrm{W}(\mathrm{a}_i, \mathrm{b}_j) \\ \max\{H_i - \mathrm{k}, \mathrm{j} - \mathrm{g}\} \\ \max\{H_i, \mathrm{j} - \mathrm{l} - \mathrm{g}\} \end{cases} \quad (1)$$

H_{ij} is the maximum similarity of two segments ending in a_i and b_j. The algorithm takes two input sequences (for example sequence A = a1, a2, a3, … an, = a1, a2, a3, ... an, and sequence B = b1, b2, b3, ... bn). Weight matrix $\mathrm{W}(\mathrm{a}_i,\mathrm{b}_j)$ where $\mathrm{W} \leq 0$ if $\mathrm{a}_i \neq \mathrm{b}_j$ dan $\mathrm{W} > 0$ if $\mathrm{a}_i = \mathrm{b}_j$ dan Gap penalty is $\mathrm{G} < 0$ which is issued to start or expand the gap. For gap penalty, the computing time of the algorithm is O(mn) if $1 \leq \mathrm{i} \leq \mathrm{m}$ and $1 \leq \mathrm{j} \leq \mathrm{n}$; and if $\mathrm{i} < 1$ or $\mathrm{j} < 1$, then H_{ij} =0.

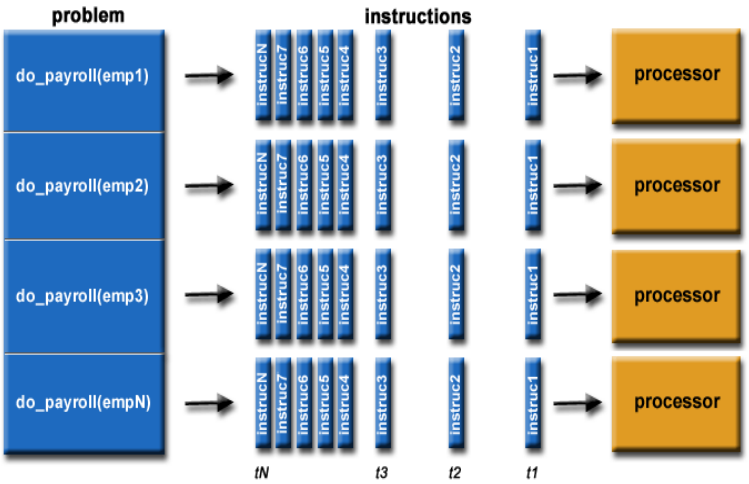

Figure 3. Illustration of Parallel Computing

Figure 3 illustrates the concept of parallel processing, which can also be applied to speed up the execution of the Smith-Waterman algorithm for sequence alignment. The concept of parallel processing,

where processing tasks (do_payroll(emp1), do_payroll(emp2), etc.) are broken down into instructions that are executed simultaneously on multiple processors. By splitting the instructions and implementing them in parallel, the system can increase efficiency and speed up execution time compared to sequential processing.

The most important information used to detect music plagiarism is the basic key of the song's BPM and the MIDI file. BPM is used to determine the position of the song melody in time in seconds; this is very important because if the song's BPM is different, then the position of the song melody or onset time will also be different.

Using the available Python library, the program code will be built in 2 types, namely in the CPU and GPU-CUDA versions. Parallel computing uses a Python library called number. Numba is an open-source library [28]. Just-in-time Python compiler that developers can use to speed up functions on the CPU and GPU using standard Python functions. Numba helps optimize the function of the Smith-Waterman algorithm, parallelize the iterations in the function, and run the Smith-Waterman function on the GPU. Parallel computing uses 3 decorators, which are used to activate numbers for each function, including @jit, @cuda, and Prange. With this decorator, the data containing the song melody must be converted into an array, which is then sent to the device or GPU. That way, the array can be run across multiple threads and blocks per grid.

Table 2. Hardware Specification

	Specification
RAM	16 GB DDR4
SSD	512 GB
CPU	Intel Core i7-11700
GPU	NVIDIA GeForce RTX 3060 12GB

Table 2 explains the hardware specifications used in this research. The NVIDIA GeForce RTX 3060 GPU helps the Smith-Waterman algorithm detect song plagiarism. This GPU is equipped with 12GB GDDR6, which is connected using a 192-bit memory interface and has a total of 3584 cores. This GPU operates at a frequency of 1.32 GHz, which can be increased up to 1.78 GHz.

Writing Python program code using Visual Studio code. In this stage, the researcher will also explain more clearly the differences between writing code that will be run with CPU and writing code that will be run with GPU.

```
@jit(target_backend="cuda", nopython=True)
def smith_waterman_gpu(string1, string2):
    # Initialize the score matrix
    score_matrix = [[0] * (len(string2) + 1) for _ in prange(len(string1) + 1)]

    # Initialize variables for storing the maximum score and its position
    max_score = 0
    max_pos = None

    # Perform the dynamic programming step
    for i in prange(1, len(string1) + 1):
        for j in prange(1, len(string2) + 1):
            if string1[i - 1] == string2[j - 1]:
                score = [
                    score_matrix[i - 1][j - 1] + 1,
                    score_matrix[i - 1][j] - 10,
                    score_matrix[i][j - 1] - 10,
                    0
                ]
                score_matrix[i][j] = max(score)

                if max(score) > max_score:
                    max_score = max(score)
                    max_pos = (i, j)
            else:
                score_matrix[i][j] = 0
```

Above is the code for performing GPU parallel computing in the Smith-Waterman algorithm. For the smith-waterman algorithm itself, the code is written based on the formula from smith-waterman itself, which will take the maximum value from the score_matrix. For the penalty and gap values in this study, the value

is -10. This is done so that Smith-Waterman will only take strings where the strings are one hundred percent the same between string 1 and string 2. Meanwhile, the match value is +1. After getting the maximum value, the program will save the position of the value/string from the score_matrix and then save it in the max_pos variable, take that value, and save it in the max_score variable.

The iteration code is written using a for loop in Python using prange and not range-like iteration in general. This is because the researcher uses range from Numba, which means parallel range where all iterations will be executed in parallel so that it can speed up the iteration process. In this section, the iteration body will be scheduled in a separate thread and run in the context of nopython numba. Then, prange will automatically handle privacy and data reduction.

In the validation stage, this study conducted a trial on 10 real cases that have been considered to have committed plagiarism. This stage compares two songs that have been regarded as cases of song plagiarism. In addition, validation also compares the results of the music plagiarism check website.

3. RESULTS AND DISCUSSION

In the implementation stage, an application display is created that facilitates plagiarism testing as shown in Figure 4 below. Several inputs are requested in the application, namely the artist's name, song title, basic tone, BPM, number of blocks and threads, and music MIDI files. After being processed, the application displays the results of detecting song similarities between the songs entered by the user and the songs in the database. This display will also show the computation time of the Smith-Waterman algorithm.

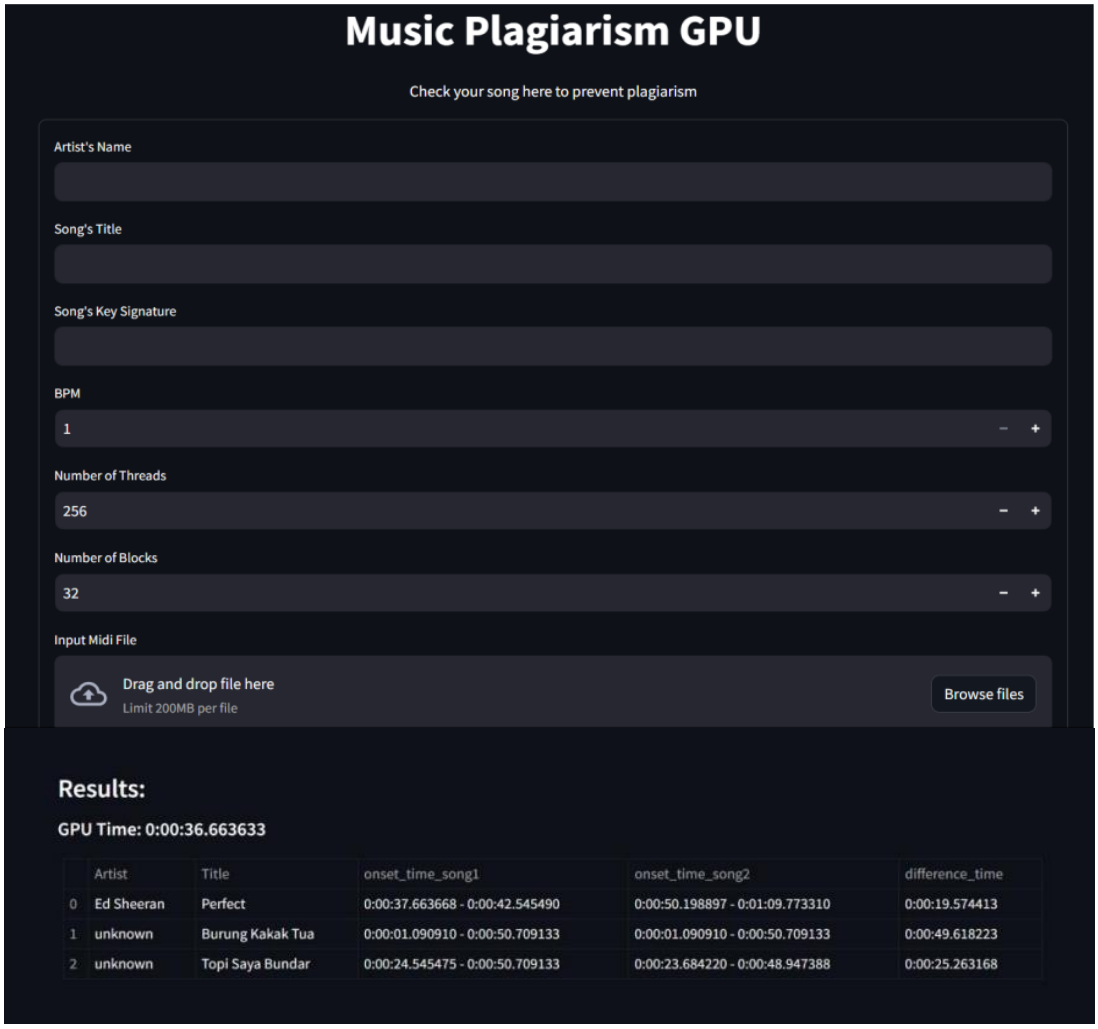

Figure 4. Application Implementation

In the initial stage, code was developed using the Smith-Waterman algorithm to detect song plagiarism. The algorithm will receive input in the form of 2 strings, string1 and string2, as the music is compared. After

that, the algorithm performs dynamic programming to check the strings on string1 and string2. Furthermore, the algorithm will only take characters from the string with the highest similarity level. This research conducted experiments on 10 pairs of songs that were cases in the real world of song plagiarism and had harmed certain parties financially and legally. It can be seen in that out of 10 cases in the real world, 7 cases were declared plagiarized, and 3 cases were declared not plagiarized. Meanwhile, the research results using the Smith-Waterman algorithm provide the same results as the actual results. So, the algorithm that was built can be used in development using CPU or GPU to detect song plagiarism. In addition, validation was carried out using the MIPPIA site, which checks for plagiarism in a song. From the MIPPIA results in **Table 3**, out of 10 cases, 7 results were the same as the existing case, 2 results were different, and 1 Indonesian song was not detected.

Table 3. Song Plagiarism Detection Results

No	Title Song 1	Artist 1	Title Song 2	Artist 2	Original Results	Research Results	MIPPIA
1.	Sweet Little Sixteen	Chuck Berry	Surfin' USA	The Beach Boys	Plagiarism	Plagiarism	Very High
2.	Burung Kakak Tua	R.C Hardjosubroto	Topi Saya Bundar	Soerjono	Plagiarism	Plagiarism	-
3.	Run Through the Jungle	Creedence Clearwater Revival	The Old Man Down the Road	John Fogerty	Not Plagiarized	Not Plagiarized	Low
4.	My Sweet Lord	George Harisson	He's so Fine	Chiffons	Plagiarism	Plagiarism	No
5.	Shaker Maker	Oasis	I'd Like to Teach the World to Sing	New Seekers	Not Plagiarized	Not Plagiarized	Very High
6.	Ice ice baby	Vanilla Ice	Under Pressure	Queen	Plagiarism	Plagiarism	Very High
7.	Stay With me	Sam Smith	I won't come back down	Tom Petty	Plagiarism	Plagiarism	No
8.	Every Breath You Take	The police	I'll be Missing You	P diddy	Plagiarism	Plagiarism	High
9.	All Day and All of the Night	The Kinks	Hello I Love You	Doors	Plagiarism	Plagiarism	Very High
10.	If I Could Fly	Joe Satriani	Viva La Vida	Coldplay	Not Plagiarized	Not Plagiarized	Safe

This research conducted several experiments using CPU and GPU with different data sets. The comparison of code using CPU is in iteration only using the range where this code is the basic code for the Python programming language. This is one of the code comparisons when using parallel computing against for-loop iteration in Python.

Based on the results obtained in **Table 4**, using CPU, the required computational execution time increases as the number of data sets increases. The data for 50 songs takes 7 minutes 3 seconds to 5000 songs takes 33 hours 45 minutes 3 seconds. So, 1 song requires ± 8 – 24 seconds in the song recognition process in the database.

The following experiment uses GPU code where in iteration using a for loop in Python, it uses range and not range like iteration on CPU. This is because Numba's prange, which means parallel range, does everything in parallel to speed up the iteration process. This section will schedule the iteration body in a separate thread and execute in the nopython numba context. Then, prange will automatically handle privacy and data reduction. Prange in Numba is the same as parallel in OpenMP, where the number of iterations will be divided into several threads.

Write the @cuda decorator in the smith_waterman_gpu function, where this decorator helps run this function on the GPU. With this decorator, the data containing the song melody must be converted into an array, which is then sent to the device or GPU. The array can be run across multiple threads and blocks per grid. Blocks on the GeForce RTX 3060 GPU have a limit, namely a maximum of 1024 blocks; blocks also have regulations, namely that they can only use block sizes in multiples of 32. If you set the block to only 32, it can limit the occupancy level, but if you use a block size of 1024, it can also limit the performance of the GPU.

Therefore, choosing a block size between 128 – 512 is best. In this study, researchers used 32 threads and 64 blocks because the maximum total registers per block on the GeForce RTX 3060 GPU is 65536, so 256 x 256 = 65536. Likewise, with the number Only 32 threads were used, this is because the maximum value of threads per block on the GeForce RTX 3060 GPU is 1024, so the researchers used 32 threads, so the total threads in one block were 32 x 32 = 1024. So, the researchers decided to use a block size of 64.

The performance results of the GPU can be seen in Table IV above, where GPU usage fluctuates or becomes uncertain when using a GPU with a block number of 256. It is known that GPU usage fluctuates from 0 to 100 over time. The same thing also happens when using block 128. This is because the use of the GPU does not exceed the maximum limit of the threads and blocks on the GPU used so that there is no excess weight, which can cause GPU usage to reach the limit of up to 100% until the program has finished executing.

Figure 5. Maximum GPU Usage Performance

However, if you use several blocks and threads that exceed the maximum limit of the GPU, it can reach 100% in **Figure 5** without decreasing to 0%. This is because the use of threads and blocks has exceeded the limit of the GPU used, which also affects the acceleration performance—resulting from. Suppose usage exceeds the maximum limit (more than 64 blocks and more than 32 threads). In that case, the acceleration provided is not very significant compared to the acceleration of the resulting blocks, namely 64, and threads, namely 32. Where in the song dataset, it is 50 if using blocks 512 and threads 256, the computing time is around 00:15:59 (12 minutes 36 seconds), but if you use blocks 64 and threads 32, the computing time is only around 00:01:24 (1 minute 24 seconds). So, in this research, the optimal GPU usage is 64 blocks and 32 threads.

In the results of parallel computing, researchers took 8 case study examples where each case study has a different dataset. The following are the results of GPU parallel computing using Numba for the Smith-Waterman algorithm. In this trial, researchers tested with 50, 200, 400, 600, 800, 1000, 2000 and 5000 datasets. Next, the execution time was measured for various sequence lengths. The formula was then used to determine the performance enhancement of the GPU relative to the CPU. This improvement is quantified by the acceleration rasio **[29]**.

$$Speed\ up = \frac{Time\ taken\ by\ CPU\ (Serial\ version)}{Time\ taken\ by\ GPU\ (Parallel\ version)} \tag{2}$$

The speedup ratio is defined as the ratio of CPU and GPU execution time. In this case, it is the ratio of time required by the CPU to the GPU. Below in Table V is a comparison of acceleration between using CPU and GPU computing:

Table 4. Comparison Between Execution Time Taken by CPU and GPU

Amount of Data	CPU	GPU	SpeedUp
50	00:07:03	00:01:24	5.0
200	00:32:57	00:06:00	5.5
400	01:23:47	00:14:49	5.6
600	02:09:54	00:23:12	5.6
800	03:17:13	00:34:45	5.7
1000	04:16:37	00:46:17	5.5
2000	08:15:03	01:28:34	5.6
5000	33:45:03	06:06:25	5.5

Table 4 and Figure 6 are the results of a comparison of execution time between CPU and GPU for 8 scenarios with different datasets. The best results were obtained with an acceleration level of 5.7 times GPU usage compared to CPU usage. This result is because dynamic programming has many data dependencies, meaning that the calculation for one cell in the matching matrix depends on the results of the previous calculation [30]. This makes it challenging to perform parallelization efficiently compared to more parallel-friendly algorithms. Dynamic programming often requires high synchronization between threads, so even though the GPU has many cores, not all can be fully utilized efficiently [31]. Smith-Waterman uses a large matrix to store the matching score values. If the matrix is not small enough to fit into shared memory or registers, access to global memory (GPU VRAM) becomes frequent and hinders performance. Global memory access on the GPU is slower than registers or shared memory, which can reduce parallelization efficiency.

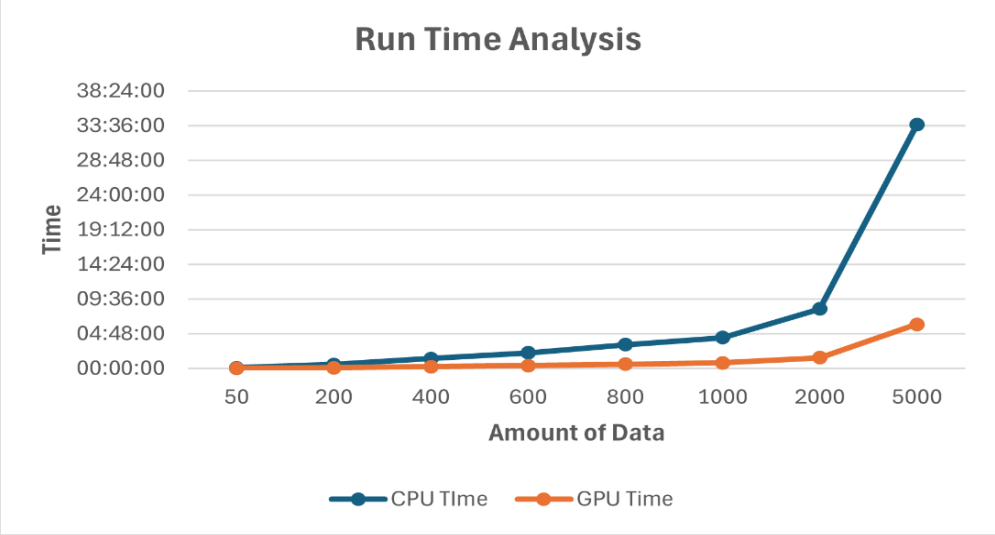

Figure 6. Comparison between execution time taken by CPU and GPU

Another research [32] used 50 MIDI files to identify similarities in song melodies with the Smith-Waterman algorithm using a CPU, and the result was a computing time of 33 minutes. Meanwhile, this research using a CPU only takes around 15 minutes, and using GPU parallel computing, the computing time of Smith-Waterman with 50 MIDI files only takes 3 minutes, so this proves that parallel computing has succeeded in speeding up the computing process of the Smith-Waterman algorithm to detect song similarities.

4. CONCLUSIONS

In terms of detecting music plagiarism using the Smith-Waterman algorithm, it requires a long execution time. The long time is because the Smith-Waterman algorithm is a dynamic programming algorithm. One way to reduce computing execution time is to utilize parallel computing, namely GPU CUDA. In this research, GPU utilization succeeded in making the process execution time 5.7 times faster than CPU usage. The different amounts of data influence this increase in speed, so the greater the computational load carried out by the GPU, the greater the performance. This shows that parallel computing has been successfully applied to the Smith-Waterman algorithm for detecting music plagiarism. However, despite the significant speedup, there are still several challenges in implementing this method. The Smith-Waterman algorithm has high data dependencies, which limits parallelization efficiency and requires frequent synchronization between threads.

Additionally, memory access bottlenecks occur when large matrices do not fit into shared memory or registers, forcing frequent global memory access, which reduces performance. Future research could optimize memory usage, improve load balancing, and explore alternative heuristic methods to enhance efficiency. Apart from that, you can commit plagiarism not only on music but also on song lyrics.

ACKNOWLEDGMENT

The author would like to thank the Directorate of Research, Technology and Community Service, Director General Higher Education, Research and Technology, Ministry of Education, Culture, Research and Technology, the Republic of Indonesia, funded this research through the novice lecturer research grant scheme (PDP) and to Atma Jaya Makassar University which has assisted with the administrative process and research permits.

REFERENCES

[1] N. T. S. Saptadi *et al.*, *PENGANTAR TEKNOLOGI INFORMASI*, vol. 1, no. 1. 2023.
[2] A. Sabiq, H. Yugaswara, and H. Wicaksono, "SISTEM KOMPUTASI PARALEL MENGGUNAKAN GPU BERBASIS NVIDIA CUDA," *ORBITH*, vol. 17, no. 2, pp. 158–164, 2021.
[3] S. Zhu *et al.*, "INTELLIGENT COMPUTING: THE LATEST ADVANCES, CHALLENGES, AND FUTURE," *Intell. Comput.*, vol. 2, 2023, doi: https://doi.org/10.34133/icomputing.0006.
[4] M. Vaithianathan, M. Patil, S. F. Ng, and S. Udkar, "COMPARATIVE STUDY OF FPGA AND GPU FOR HIGH-PERFORMANCE COMPUTING AND AI," *Int. J. Adv. Comput. Technol.*, vol. 1, no. May 2023, pp. 37–46, 2023, doi: 10.56472/25838628/IJACT-V1I1P107.
[5] M. Schubiger, G. Banjac, and J. Lygeros, "GPU ACCELERATION OF ADMM FOR LARGE-SCALE QUADRATIC PROGRAMMING," *J. Parallel Distrib. Comput.*, vol. 144, pp. 55–67, 2020, doi: https://doi.org/10.1016/j.jpdc.2020.05.021.
[6] M. Aslam, O. Riaz, S. Mumtaz, and A. D. Asif, "PERFORMANCE COMPARISON OF GPU-BASED JACOBI SOLVERS USING CUDA PROVIDED SYNCHRONIZATION METHODS," *IEEE Access*, vol. 8, pp. 31792–31812, 2020, doi: https://doi.org/10.1109/ACCESS.2020.2973669.
[7] A. Gormantara and Pranowo, "PARALLEL SIMULATION OF PATTERN FORMATION IN A REACTION-DIFFUSION SYSTEM OF FITZHUGH-NAGUMO USING GPU CUDA," *AIP Conf. Proc.*, vol. 2217, no. April, 2020, doi: https://doi.org/10.1063/5.0000667.
[8] A. Asaduzzaman, A. Trent, S. Osborne, C. Aldershof, and F. N. Sibai, "IMPACT OF CUDA AND OPENCL ON PARALLEL AND DISTRIBUTED COMPUTING," *2021 8th Int. Conf. Electr. Electron. Eng. ICEEE 2021*, pp. 238–242, 2021, doi: https://doi.org/10.1109/ICEEE52452.2021.9415927.
[9] P. R. Kommera, S. S. Muknahallipatna, and J. E. McInroy, "OPTIMIZED CUDA IMPLEMENTATION TO IMPROVE THE PERFORMANCE OF BUNDLE ADJUSTMENT ALGORITHM ON GPUs," *J. Softw. Eng. Appl.*, vol. 17, no. 04, pp. 172–201, 2024, doi: https://doi.org/10.4236/jsea.2024.174010.
[10] M. Afif, Y. Said, and M. Atri, "COMPUTER VISION ALGORITHMS ACCELERATION USING GRAPHIC PROCESSORS NVIDIA CUDA," *Cluster Comput.*, vol. 23, no. 4, pp. 3335–3347, 2020, doi: https://doi.org/10.1007/s10586-020-03090-6.
[11] K. Kaur, S. Chakraborty, and M. K. Gupta, "ACCELERATING SMITH-WATERMAN ALGORITHM FOR FASTER SEQUENCE ALIGNMENT USING GRAPHICAL PROCESSING UNIT," in *Journal of Physics: Conference Series*, 2022, vol. 2161, no. 1, doi: https://doi.org/10.1088/1742-6596/2161/1/012028.
[12] S. Petti *et al.*, "END-TO-END LEARNING OF MULTIPLE SEQUENCE ALIGNMENTS WITH DIFFERENTIABLE SMITH–WATERMAN," *Bioinformatics*, vol. 39, no. 1, 2023, doi: https://doi.org/10.1093/bioinformatics/btac724.
[13] L. You *et al.*, "GPU-ACCELERATED FASTER MEAN SHIFT WITH EUCLIDEAN DISTANCE METRICS," in *Proceedings - 2022 IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC 2022*, 2022, pp. 211–216, doi: https://doi.org/10.1109/COMPSAC54236.2022.00037.
[14] A. Gormantara, "Analisis Sentimen Terhadap New Normal Era di Indonesia pada Twitter Analisis Sentimen Terhadap New Normal Era di Indonesia pada Twitter Menggunakan Metode Support Vector Machine," no. July, pp. 0–5, 2020.
[15] M. N. A. Putera Khano, D. R. S. Saputro, S. Sutanto, and A. Wibowo, "SENTIMENT ANALYSIS WITH LONG-SHORT TERM MEMORY (LSTM) AND GATED RECURRENT UNIT (GRU) ALGORITHMS," *BAREKENG J. Ilmu Mat. dan Terap.*, vol. 17, no. 4, pp. 2235–2242, 2023, doi: https://doi.org/10.30598/barekengvol17iss4pp2235-2242.
[16] A. A. Nareswari and D. T. Utari, "ADVANCEMENTS IN ALZHEIMER'S DIAGNOSIS THROUGH MRI USING BAYESIAN CONVOLUTIONAL NEURAL NETWORKS AND VARIATIONAL INFERENCE," *BAREKENG J. Ilmu Mat. dan Terap.*, vol. 18, no. 4, pp. 2423–2434, 2024, doi: https://doi.org/10.30598/barekengvol18iss4pp2423-2434.
[17] R. Hidalgo, A. Devito, N. Salah, A. S. Varde, and R. W. Meredith, "INFERRING PHYLOGENETIC RELATIONSHIPS USING THE SMITH-WATERMAN ALGORITHM AND HIERARCHICAL CLUSTERING," *Proc. - 2022 IEEE Int. Conf. Big Data, Big Data 2022*, no. December, pp. 5910–5914, 2022, doi: https://doi.org/10.1109/BigData55660.2022.10020454.
[18] M. Risnasari, M. A. Effindi, P. Dellia, L. Cahyani, N. Aini, and N. Aini, "COMPUTER BASED TEST USING THE FISHER-YATES SHUFFLE AND SMITH WATERMAN ALGORITHM," *KnE Soc. Sci.*, vol. 2021, pp. 353–360, 2021, doi: https://doi.org/10.18502/kss.v5i6.9224.
[19] B. Sari and Y. Sibaroni, "DETEKSI KEMIRIPAN DOKUMEN BAHASA," *Ind. J. Comput.*, vol. 4, no. 3, pp. 87–98, 2019, doi: 10.21108/indojc.2019.4.3.365.
[20] F. Alvi, M. Stevenson, and P. Clough, "PARAPHRASE TYPE IDENTIFICATION FOR PLAGIARISM DETECTION USING CONTEXTS AND WORD EMBEDDINGS," *Int. J. Educ. Technol. High. Educ.*, vol. 18, no. 1, 2021, doi: https://doi.org/10.1186/s41239-021-00277-8.
[21] A. Khozaimi, H. Husni, Y. D. Pramudita, M. F. Adlim, F. H. Rachman, and I. O. Susanti, "CHATBOT CERDAS SEBAGAI HELPDESK OBJEK WISATA MENGGUNAKAN ALGORITMA SMITH-WATERMAN," *J. Teknol. Inf. dan Terap.*, vol.

10, no. 1, pp. 36–46, 2023, doi: https://doi.org/10.25047/jtit.v10i1.312.
[22] J. Zhao, D. Taniar, K. Adhinugraha, V. M. Baskaran, and K. S. Wong, "MULTI-MMLG: A NOVEL FRAMEWORK OF EXTRACTING MULTIPLE MAIN MELODIES FROM MIDI FILES," *Neural Comput. Appl.*, vol. 35, no. 30, pp. 22687–22704, 2023, doi: https://doi.org/10.1007/s00521-023-08924-z.
[23] R. Kubik *et al.*, "DEVELOPMENT OF AN INTELLIGENT SYSTEM FOR SELECTING SONGS ACCORDING TO THE USER NEEDS," *CEUR Workshop Proc.*, vol. 2604, pp. 1251–1279, 2020.
[24] J. N. Muhammad Irfan Reza Mahendra, "PERLINDUNGAN HUKUM PREVENTIF DAN REPRESIF TERHADAP PERBUATAN PLAGIARISME CIPTAAN LAGU ATAU MUSIK," *Nusant. J. Ilmu Pengetah. Sos.*, vol. 9, no. 4, pp. 1483–1490, 2022.
[25] Z. Yin, F. Reuben, S. Stepney, and T. Collins, "'A GOOD ALGORITHM DOES NOT STEAL – IT IMITATES': THE ORIGINALITY REPORT AS A MEANS OF MEASURING WHEN A MUSIC GENERATION ALGORITHM COPIES TOO MUCH," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2021, vol. 12693 LNCS, pp. 360–375, doi: https://doi.org/10.1007/978-3-030-72914-1_24.
[26] A. D. Ongriwalu and Y. Yunanto, "KARYA CIPTA LAGU DAN MUSIK DALAM BINGKAI ASAS PERLINDUNGAN HUKUM," *AL-MANHAJ J. Huk. dan Pranata Sos. Islam*, vol. 5, no. 2, pp. 1369–1374, 2023, doi: https://doi.org/10.37680/almanhaj.v5i2.2781.
[27] H. El Haji and L. Alaoui, "A CATEGORIZATION OF RELEVANT SEQUENCE ALIGNMENT ALGORITHMS WITH RESPECT TO DATA STRUCTURES," *Int. J. Adv. Comput. Sci. Appl.*, vol. 11, no. 6, pp. 268–273, 2020, doi: https://doi.org/10.14569/IJACSA.2020.0110635.
[28] A. Widjaja, T. K. Gautama, S. F. Sujadi, and S. R. Harnandy, "HIGH PERFORMANCE COMPUTING ENVIRONMENT USING GENERAL PURPOSE COMPUTATIONS ON GRAPHICS PROCESSING UNIT," *J. Tek. Inform. dan Sist. Inf.*, vol. 7, no. 2, 2021, doi: https://doi.org/10.28932/jutisi.v7i2.3715.
[29] K. Kaur, S. Chakraborty, and M. K. Gupta, "ACCELERATING SMITH-WATERMAN ALGORITHM FOR FASTER SEQUENCE ALIGNMENT USING GRAPHICAL PROCESSING UNIT," *J. Phys. Conf. Ser.*, vol. 2161, no. 1, 2022, doi: https://doi.org/10.1088/1742-6596/2161/1/012028.
[30] B. Schmidt, F. Kallenborn, A. Chacon, and C. Hundt, "CUDASW ++ 4 . 0 : ULTRA - FAST GPU - BASED SMITH – WATERMAN PROTEIN SEQUENCE DATABASE SEARCH," *BMC Bioinformatics*, 2024, doi: https://doi.org/10.1101/2023.10.09.561526.
[31] Z. Xia *et al.*, "A REVIEW OF PARALLEL IMPLEMENTATIONS FOR THE SMITH – WATERMAN ALGORITHM," *Interdiscip. Sci. Comput. Life Sci.*, vol. 14, no. 1, pp. 1–14, 2022, doi: https://doi.org/10.1007/s12539-021-00473-0.
[32] V. Krishnakumar, "DETECTION OF SIMILAR MELODIES BY REPURPOSING ALGORITHMS FOR SEQUENCE ALIGNMENT AND STRING SEARCHING VIHAAN KRISHNAKUMAR," *Res. Arch. Rising Sch.*, pp. 1–7, 2023, doi: https://doi.org/10.58445/rars.814.