Accelerating k-Shape Time Series Clustering Algorithm Using GPU

Xun Wang, Ruibao Song, Junmin Xiao, Tong Li, and Xueqi Li, *Member, IEEE*

***Abstract*—In the data space, time-series analysis has emerged in many fields, including biology, healthcare, and numerous large-scale scientific facilities like astronomy, climate science, particle physics, and genomics. Clustering is one of the most critical methods in time-series analysis. So far, the state-of-art time series clustering algorithm k-Shape has been widely used not only because of its high accuracy, but also because of its relatively low computation cost. However, due to the high heterogeneity of time series data, it can not be simply regarded as a high-dimensional vector. Two time series often need some alignment method in similarity comparison. The alignment between sequences is often a time-consuming process. For example, when using dynamic time warping as a sequence alignment algorithm and if the length of time series is greater than 1,000, a single iteration in the clustering process may take hundreds to tens of thousands of seconds, while the entire clustering cycle often requires dozens of iterations. In this article, we propose a set of novel parallel strategies suitable for GPU's computation model, called Times-C, which is an abbreviation for Time Series Clustering. We define three stages in the analysis process: aggregation, centroid, and class assignment. Times-C includes efficient parallel algorithms and corresponding implementations for these three stages. Overall, the experimental results show that the Times-C algorithm exhibits a performance improvement of one to two orders of magnitude compared to the multi-core CPU version of k-Shape. Furthermore, compared to the GPU version of the k-Shape algorithm, the Times-C algorithm achieves a maximum acceleration of up to 345 times.**

***Index Terms*—Data space, time series analysis, time series clustering, GPU architecture, k-shape algorithm.**

Manuscript received 31 August 2022; revised 5 July 2023; accepted 18 July 2023. Date of publication 24 July 2023; date of current version 11 August 2023. This work was supported in part by the National Key R&D Program of China under Grants 2022YFB4500403, 2021YFA1000103, and 2021YFA1000100, in part by the NSF of China under Grants 61972416, 62202454, 62272479, and 62202498, in part by the Taishan Scholarship under Grant tsqn201812029, in part by the Foundation of Science and Technology Development of Jinan under Grant 201907116, in part by the Shandong Provincial Natural Science Foundation under Grant ZR2021QF023, in part by the Fundamental Research Funds for the Central Universities under Grant 21CX06018A, in part by the Spanish project under Grant PID2019-106960GB-I00, in part by the Juan de la Cierva under Grant IJC2018-038539-I, in part by the China National Postdoctoral Program for Innovative Talents under Grant BX2021320, and in part by the Chinese Academy of Engineering Strategic Research and Consulting Program under Grant 2023-XBZD-16. Recommended for acceptance by P. D'Ambra. *(Corresponding author: Xueqi Li.)*

Xun Wang is with the Department of Computer Science and Technology, China University of Petroleum (East China), Qingdao, Shangdong 266580, China, and also with High Performance Computer Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: wangsyun@upc.edu.cn).

Ruibao Song and Tong Li are with the Department of Computer Science and Technology, China University of Petroleum (East China), Qingdao, Shangdong 266580, China (e-mail: sruibao@s.upc.edu.cn; ltong@s.upc.edu.cn).

Xueqi Li is with the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: lixueqi@ict.ac.cn).

Junmin Xiao is with the High Performance Computer Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China (e-mail: xiaojunmin@ict.ac.cn).

Digital Object Identifier 10.1109/TPDS.2023.3298148

I. Introduction

IN THE data space, time series data refers to the dataset used for conducting time-series analysis. This type of data consists of observations or measurements collected at regular intervals over a sequence of equally spaced time points. Many data formats are stored as time series data [1], [2], [3], [4], [5], such as biology, healthcare (blood pressure and ECG measurement), and numerous large-scale scientific facilities like astronomy, climate science, particle physics, and genomics. Among all techniques applied to time-series data analysis, time-series clustering is one of the most critical ones because it does not rely on costly human supervision or time-consuming annotation of data. Through clustering, we can get the division of the data between different classes and the overall characteristics of the original data. In recent years, time series clustering has been applied in many fields [6], [7], [8], [9]. Also, with the development of big data technologies and data collection technologies, the data size has proliferated [12], [13], and timely and efficient time-series clustering algorithms have become even more important.

Recent decades have witnessed the development of clustering algorithms for time series clustering. Specifically, to effectively calculate the similarity between different series, the dynamic time warping (DTW) algorithm [27] has been developed, which calculates the similarity of each point in a sequence relative to multiple consecutive points in another series based on the distance matrix between sequences. The similarity of the aligned sequences is obtained by searching the distance matrix. However, in the DTW algorithm, the generation of the distance matrix is time-consuming because the next step of calculation depends on the results of the previous step, which results in a serial workflow and low parallelism.

Another domain-independent time series clustering algorithm often used is the k-Shape algorithm [11], which has proven to be more efficient in many fields [6], [7], [8]. Compared with DTW-based time series clustering algorithms, the k-Shape algorithm has a smaller calculation cost under the same accuracy and has no constraint on a back-and-forth workflow. Further, previous studies have shown that the k-Shape algorithm achieves 10× speedup [11]. However, when facing a surge in data size, the performance of k-Shape algorithm is still limited by two very time-consuming key stages. The first stage is the similarity calculation, which has to derive the offset of each time point in the sequence on the time axis. The second stage is the class center extraction, in which all the data in the class are treated as a single matrix, and the matrix eigenvector corresponding to the maximum eigenvalue is used as the class center.

In this paper, our *goal* is to accelerate the k-Shape algorithm using GPUs. We first analyzed the potential for parallelism in the two time-consuming stages. In the similarity computation stage, the input sequences can be computed independently, providing an opportunity for parallel implementation on GPUs. Furthermore, the computations between classes are independent, and the matrix operations within classes can also be executed in parallel on GPUs. Based on the two observations, we found that GPU is a potentially good platform for tapping into the parallelism of k-Shape algorithms. However, there are still three challenges that need to be addressed for achieving high performance.

- *Challenge 1:* A core of class center extraction is the data aggregation operation, i.e., aggregating data of the same class together. Parallel sorting is an efficient way to aggregate data, but since time series are usually used as high-dimensional vectors, which leads to a not-so-subtle time overhead for each move in memory, it is not simple to implement fast data aggregation.
- *Challenge 2:* Another core of class center extraction is class center computation, which is essentially solving for the eigenvectors corresponding to the largest eigenvalues of multiple real symmetric matrices. However, the current method [34] is either for a single matrix, not for the parallelism between the matrix and the matrix, or for all eigenvectors, which results in waste in computations. Therefore, how efficiently finding the eigenvectors corresponding to the maximum eigenvalues of multiple matrices will be a challenge.
- *Challenge 3:* Similarity calculation requires time series to be offset and aligned on the time axis, so it is not easy to effectively obtain the similarity between the offset sequence and the class center in parallel.

To overcome the three challenges, we propose a new parallel approach for GPU architecture, named *Times-C*, which stands for Time Series Clustering. We divided the analysis process into three stages: *Aggregation*, *Centroid*, and *Class assignment*, and propose efficient parallel algorithms and implementations for each stage. For *Aggregation*, we propose a parallel fast hash sorting method that is suitable for high-dimensional vectors. For *Centroid*, we design a two-level parallel structure to quickly determine the class centers of different classes. For *Class assignment*, we develop a parallel frequency domain similarity calculation method that uses three-dimensional threads to compute the similarity between offset sequences and class centers in parallel. Our main contributions in this paper are as follows:

- We designed a parallel data hash sorting method to improve the data locality and optimize memory access, which aggregates the same class of data quickly and ensures that the same class of data can be processed in parallel (Section IV-B).
- We propose a two-level parallel structure for the center extraction to compute eigenvectors for the new centroid of each class in parallel, which can effectively compute the eigenvectors corresponding to the maximum eigenvalues in parallel (Section IV-C).
- We designed an efficient parallel method for similarity calculation. Specifically, we designed a method that utilizes three-dimensional thread parallelism to compute Fourier multiplications. This approach enables fast calculation of the similarity between the data and the center. (Section IV-D).
- We conducted extensive experiments to evaluate the effect of different configurations on the k-Shape efficiency. Overall, compared with the state-of-the-art k-Shape algorithm, i.e., tslearn [40], the Times-C algorithm achieves one to two orders of magnitude acceleration. (Section V).

The rest of this paper is organized as follows: Section II introduces the k-Shape algorithm, the CUDA programming model, and the acceleration library used in this research. Section III provides a comprehensive analysis of the k-Shape algorithm. Section IV presents the details of the Times-C algorithm. Section V presents the experiment results. Section VI discusses the related work, and Section VII concludes the paper.

II. Preliminaries

A. Time Series Clustering

Given a dataset containing n time series $D = \{D_1, D_2, \ldots, D_n\}$, and a similarity measure F, we get $C = F(D)$, where $C = \{C_1, C_2, \ldots, C_k\}$, and $C_i \cap C_j = \emptyset$ $(i \neq j,\ i, j \in [1, k],\ k \ll n)$, C_i contains a homogeneous set of data D, $D = \cup_{i=1}^{k} C_i$. The process of mapping D to C through some similarity measure is called time series clustering.

Time series data is widely available in many fields such as biology[2], finance[3], energy[6], [8], market analysis[9], etc. Especially in recent years, with the rapid development of the internet, data storage, and other technologies, a significant amount of time series data is generated every day[12] (such as patients' ECG, EEG data, weather data, etc.). Time series data contains valuable information, such as weather data[10], which not only reflects the changes in the past but also enables us to predict future trends through analysis. Time series clustering, as an essential part of time series analysis, has attracted considerable attention. Currently, many time series analysis tasks require obtaining the class labels to which the series belong[1]. However, in real life, the collected time series data rarely comes with labels. Manual labeling is a time-consuming and energy-consuming task. Therefore, clustering algorithms are often relied upon to obtain the corresponding annotations for time series data.

B. k-Shape Algorithm

Similar to the k-means algorithm[14], k-Shape algorithm proposed by Paparrizos et al. [11] is an iterative process, which consists of two main parts: The first part is to calculate the center of each class. The second part is determining which class each data belongs to based on the class center, and both rely on similarity measures.

1) Time-Series Similarity: In most cases, even if two time series are similar overall, their shapes may not be aligned on the time axis. Therefore, one (or both) of them needs to be offset under the time axis to achieve better alignment before comparing their similarity. In the similarity measurement method of time series clustering, the widely known method is the DTW algorithm. However, DTW is not efficient because of its high computation cost. Therefore, k-Shape algorithm uses cross-correlation as a measure of similarity between time series data.

Cross-Correlation Measure: Given two time series $\vec{x} = (x_1, x_2, \ldots, x_m)$ and $\vec{y} = (y_1, y_2, \ldots, y_m)$, we indicate that the offset $\vec{x}_{(s)}$ of $\vec{x}$ relative to $\vec{y}$ is

$$\vec{x}_{(s)} = \begin{cases} (\overbrace{0, \ldots, 0}^{|s|}, x_1, x_2, \ldots, x_{m-s}), & s \geq 0 \\ (x_{1-s}, \ldots, x_{m-1}, x_m, \underbrace{0, \ldots, 0}_{|s|}), & s < 0 \end{cases}. \quad (1)$$

Where $s \in [-m, m]$, then the cross-correlation measure of $\vec{x}$ and $\vec{y}$ can be defined as $CC_w(\vec{x}, \vec{y}) = (\vec{c}_1, .., \vec{c}_t, ..\vec{c}_{2m-1})$, where

$$\vec{c}_t = \sum_{k=1}^{m} y_k \cdot \vec{x}_{(t-m),k} \quad t \in [1, 2m-1]. \quad (2)$$

Considering the differences between sequences, normalization is required. The k-Shape algorithm uses coefficient normalization to normalize similarity and z-score normalization to normalize sequences. The normalized similarity can be defined as

$$SBD(\vec{x}, \vec{y}) = \max_w \left(\frac{CC_w(\vec{x}, \vec{y})}{\sqrt{c_m(\vec{x}, \vec{x}) \cdot c_m(\vec{y}, \vec{y})}} \right). \quad (3)$$

For the above formula, if the length of the sequence is m, then $2 \cdot m - 1$ offsets are required, and after each offset, m-times multiplication is needed, so the time complexity of the whole process is $\mathbf{O}(m^2)$. This is unacceptable, but from the convolution theorem[23], the cross-correlation calculation of the time domain is equal to the product of the frequency domain, i.e.,

$$CC(\vec{x}, \vec{y}) = \mathcal{F}^{-1}\{\mathcal{F}(\vec{x}) * \mathcal{F}(\vec{y})\}. \quad (4)$$

Where $\mathcal{F}(\vec{x}), \mathcal{F}(\vec{y})$ represents $\vec{x}$ and $\vec{y}$ for Fourier transformation operation, $\mathcal{F}^{-1}()$ represents the inverse Fourier transformation operation[28], and $*$ for $\mathcal{F}(\vec{x})$ multiplied by the conjugate complex of $\mathcal{F}(\vec{y})$.

By fast Fourier transform operation (FFT) proposed by Cooley et al. [29], the time complexity can be reduced to $\mathbf{O}(mlog(m))$.

Fig. 1 describes the process of similarity calculation. First, we fill the time series to the specified length by padding zero. The calculation method for this length is as follows:

$$length = 2^{\text{nextpower2}(2 \cdot \text{length}(x) - 1)}. \quad (5)$$

Then, we perform fast Fourier transform on the padded data and perform the multiplication and inverse Fourier transform described in (4). At this time, we get the similarity measure of different offsets between sequences. For convenience, we call it $similarity\ array$.

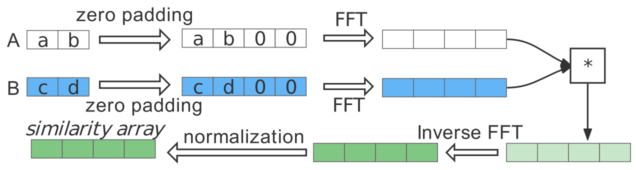


Fig. 1. Example of similarity calculation.

2) Time Series Centroid Extraction: In the process of clustering, we often use one data to represent one class, which is often called the center of the class. When we divide the samples into k classes, each sample will be closest to the center of its class. Among them, the extraction of the center depends heavily on the choice of similarity measurement. This part shows how the k-Shape algorithm extracts the center of the class.

The simplest way to extract the class center from a set of sequences is to represent each coordinate of the class center as the arithmetic mean of the corresponding coordinates of all sequences in the class. However, for time series data, these centers can not effectively capture class features[11]. The k-Shape algorithm transforms the center extraction into an optimization problem. For each class, the goal is to find the central sequence $\vec{\mu}_k^*$ that has the maximum similarity NCC_c with all time series in the class. For convenience, k-Shape algorithm optimizes the square of the similarity, i.e.,

$$\vec{\mu}_k^* = \underset{\vec{\mu}_k}{\text{argmax}} \sum_{\vec{x}_i \in P_k} \left(\frac{\max_w CC_w(\vec{x}_i, \vec{\mu}_k)}{\sqrt{c_m(\vec{x}_i, \vec{x}_i) \cdot c_m(\vec{\mu}_k, \vec{\mu}_k)}} \right)^2. \quad (6)$$

Where $\vec{\mu}_k$ represents the center of the kth class and P_k represents the data set in the kth class. Combined with the (2), (6) is transformed into

$$\vec{\mu}_k^* = \underset{\vec{\mu}_k}{\text{argmax}} \sum_{\vec{x}_i \in P_k} \left(\frac{\sum_{l \in [1,m]} x_{il} \cdot \mu_{kl}}{\sqrt{c_m(\vec{x}_i, \vec{x}_i) \cdot c_m(\vec{\mu}_k, \vec{\mu}_k)}} \right)^2. \quad (7)$$

At this stage, since sequences are already aligned to a reference sequence before computing the centroid. The denominator in (7) can be ignored. To handle the centering of $\vec{\mu}_k$, k-Shape algorithm set $\vec{\mu}_k = \vec{\mu}_k \cdot Q$, $S = \sum_{\vec{x}_i \in P_k} (\vec{x}_i \cdot \vec{x}_i^T)$, where $Q = I - \frac{1}{m} O$, I is the identity matrix, and O is a matrix with all ones. Besides, to ensure $\vec{\mu}_k$ has a unit norm, the k-Shape algorithm divides (7) by $\vec{\mu}_k^T \cdot \vec{\mu}_k$. Finally, we obtain

$$\begin{aligned} \vec{\mu}_k^* &= \underset{\vec{\mu}_k}{\text{argmax}} \frac{\vec{\mu}_k^T \cdot Q^T \cdot S \cdot Q \cdot \vec{\mu}_k}{\vec{\mu}_k^T \cdot \vec{\mu}_k} \\ &= \underset{\vec{\mu}_k}{\text{argmax}} \frac{\vec{\mu}_k^T \cdot M \cdot \vec{\mu}_k}{\vec{\mu}_k^T \cdot \vec{\mu}_k}, \end{aligned} \quad (8)$$

where $M = Q^T \cdot S \cdot Q$. In this case, the optimized objective function is known as the Rayleigh quotient maximization problem[30], and the maximized $\vec{\mu}_k$ can be reduced to the eigenvector corresponding to the maximum eigenvalue of the real symmetric matrix M.

Algorithm 1: $[IDX, C] = k\text{-}Shape(X, k)$.

```
Input:
  $X$ is an $m$-by-$n$ matrix containing m time series.
  $k$ is the number of clusters.
Output:
  IDX is an $m$-by-1 vector containing the assignment of
  $m$ time series to $k$ clusters (initialized randomly).
1: $iter = 0$
2: $IDX = random(1, k)$
3: $IDX' = [\,]$
4: while $IDX \mathrel{!}= IDX'$ and $iter < maxIter$ do
5:   $IDX' = IDX$
6:   for $j = 1$ to $k$ do
7:     $X' = [\,]$
8:     for $i = 1$ to $m$ do
9:       if $IDX(i) == j$ then
10:        $A = similarity\ array(X(i), C(j))$
11:        $p = argmax(A)$
12:        $s = p - length(X(i))$
13:        Shift $X(i)$ by (1), the shift parameter is $s$
14:        $X' = [X'; X(i)]$
15:      end if
16:    end for
17:    Solve the eigenvector of $X'$ according to (8) to
       obtain the class center $C$
18:  end for
19:  for $i = 1$ to $m$ do
20:    $maxdist = -\infty$
21:    for $j = 1$ to $k$ do
22:      $Arr = similarity\ array(X(i), C(j))$
23:      $dist = \max(Arr)$
24:      if $dist > maxdist$ then
25:        $maxdist = dist$
26:        $IDX(i) = j$
27:      end if
28:    end for
29:  end for
30:  $iter = iter + 1$
31: end while
```

3) k-Shape Time Series Clustering Algorithm: k-Shape is an iterative refinement process where the maximum number of iterations is defined by us. The algorithm execution process is shown in Algorithm 1. In each iteration, k-Shape performs two steps: (i) In the class center extraction step, the algorithm extracts the center of each class according to the data contained in each class (Lines 5-18); (ii) In the assignment step, the algorithm updates the class to which the member belongs by comparing each time series with all class centers and assigning each time series to the nearest center (Lines 19-31). The algorithm repeats these two steps until the class membership does not change or reaches the maximum number of iterations allowed. In the assignment step, k-Shape mainly depends on the similarity measure, while in the extraction step, it mainly depends on the center calculation method.

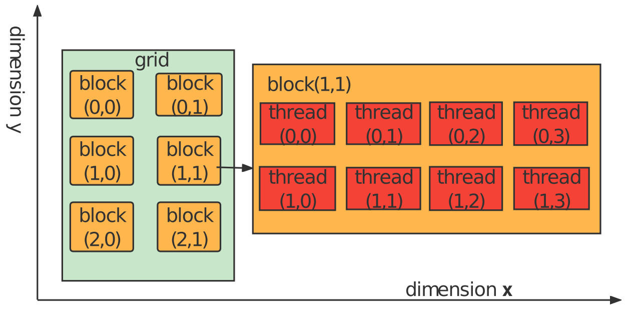


Fig. 2. Example of two-dimensional thread block and two-dimensional thread, in which the dimension of thread block is 2x3 and the dimension of thread is 4x2.

C. CUDA Programming Model and Libraries

Using GPU as a universal hardware accelerator has become the preferred choice today, whether in scientific computing[20] or in the fields of biology[19], medicine[21], chemistry[22], artificial intelligence[24], [25], computer graphics[26], etc. The key to GPU computing is that it provides many computational cores in hardware and an abstract programming model based on those cores.

The CUDA programming model consists of three levels: *grids*, *blocks*, and *threads*. Blocks and threads can be organized into 3-dimensional structures (x, y, z), within a grid and a block, respectively. This three-dimensional organization allows for more flexibility in addressing and accessing data, as well as better utilization of the parallel processing capabilities of GPUs. Thread is the finest granularity in our task parallelism. Blocks are composed of several threads as the basic scheduling unit of the Streaming Multiprocessor(SM). Threads within the same thread block run in the same SM, and they can synchronize in the block. The grid is composed of several thread blocks and serves as the basic dispatching unit at the GPU level. The dimensions of the blocks and threads (no more than three dimensions) can be defined by us. Fig. 2 shows an example of the layout of two-dimensional threads and thread blocks. In addition, we can define the size of the grid and thread blocks. On the current GPU, a block may contain up to 1,024 threads, each thread we define has its own index that can be accessed. CUDA also provides a number of basic algorithm libraries, such as

cuFFT[36]: Fourier transform is important mathematical transform in the field of digital signal processing, transforming signals from time domain to frequency domain. The cuFFT library allows users to quickly leverage the floating-point power and parallelism of the GPU in a highly optimized and tested FFT library.

cuSOLVER[34]: The NVIDIA cuSOLVER library provides a dense and sparse set of linear algebra solvers, which provide significant acceleration for computer vision, CFD, computational chemistry, and linear optimization applications. The cuSolver consists of the following three components.

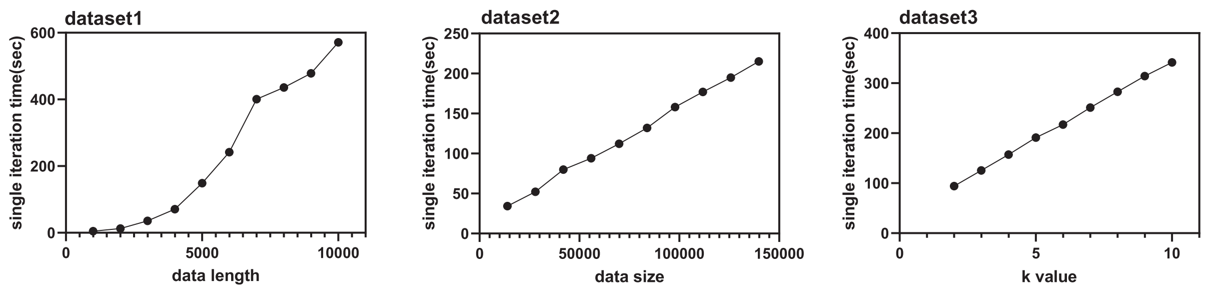


Fig. 3. Average execution time of a single iteration of the k-Shape algorithm.

TABLE I
DATASETS FOR ANALYSIS

Dataset	Data Size	Length	Number of Classes
dataset1	8732	1000-10000	10
dataset2	13988-139883	3750	2
dataset3	55912	3750	2-10

TABLE II
METRICS OF ALGORITHM 1

SM occupancy	gld_throughput	gst_throughput
2.08%	0.94Gb/s	0.73Gb/s

- *cuSolverDN:* It handles dense matrix factorization and solves routines such as LU, QR, SVD, and LDLT.
- *cuSolverSP:* cuSolverSP provides a new set of sparse routines based on sparse QR factorization.
- *cuSolverRF:* cuSolverRF is a sparse re-factorization package that can provide very good performance when solving a sequence of matrices where only the coefficients are changed, but the sparsity pattern remains the same.

cuBLAS[35]: The cuBLAS library provides the GPU accelerated implementation of the basic linear algebra subroutine (BLAS), which has made great contributions to accelerating AI and HPC applications.

III. PROFILING OF ORIGINAL K-SHAPE ALGORITHM

We first analyze the performance bottleneck of the original k-Shape algorithm, including the key parameters that affect the performance. Then we propose a new approach for *stage division* and present the reasons.

A. Analysis of k-Shape Algorithm

The k-Shape algorithm is a domain-independent algorithm that can be applied to many fields and can directly calculate on the original data without preprocessing the data. However, when dealing with large-scale datasets, the computational efficiency of recent time-series clustering suffers from high overhead on data transfer and calculation.

As shown in Fig. 3, we tested the average execution time of a single iteration of the k-Shape algorithm in ten iterations on multiple datasets (detailed in Table I). Fig. 3 shows the time cost of the k-Shape algorithm with the growth of data length (time series length), data size (number of time series), and k value. It can be seen that the running time of the k-Shape algorithm increases superlinearly with the increase of data length and linearly with the increase of data size and k-value. The reason for this difference is that the increase in data length will lead to the increase of data volume of each time series in the dataset, while the increase in data size is only an incremental increase. The increase of the k value means that more class center extraction calculations are required. As shown in the figure, when the amount of data increases, a single iteration of the k-Shape algorithm will take hundreds of seconds. If we set up more than ten iterations, the program may take several hours or even more than ten hours to execute. The existing data, both in terms of data length and data size, are much larger than those years ago. In the future, the data volume may become larger. Therefore, it is necessary to parallelize the k-Shape algorithm for improved performance.

B. Three Key Stages in k-Shape Algorithm

The k-Shape algorithm is originally divided into two stages: *class center extraction* and *class assignment*. According to the characteristics and parallelizability of the two steps, we have subdivided the class center extraction into two parts: (1) data aggregation and alignment; (2) class center calculation. Among them, data alignment can be achieved by some methods of class assignment, but considering the order of execution, we combine it with data aggregation. Data aggregation is an essential part of class center computing. However, directly migrating Algorithm 1 from CPU to GPU can not fully use the GPU resources. Table II shows some performance metrics of Algorithm 1 when it is directly migrated to GPU, where $gld\_throughput$ stands for global memory loads throughput, and $gst\_throughput$ represents global memory stores throughput. It is clear that Algorithm 1 has a low affinity for the GPU during data aggregation, and the SM occupation of GPU is only about 2%. This poor affinity means we need to optimize data aggregation, so we separate data aggregation from class center computing.

We then test the time distribution of the three stages with multiple datasets and caculate the average execution time. As shown in Table III, the three parts bear the main time cost of the algorithm. Further, it makes sense to provide different

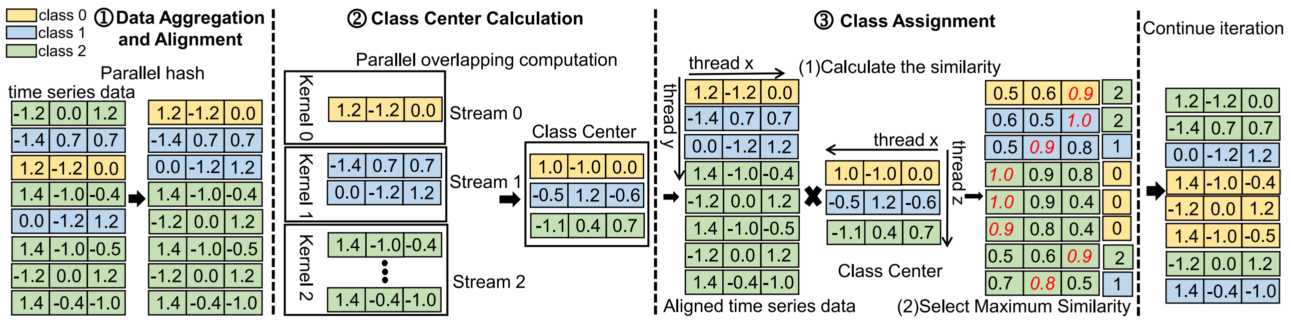


Fig. 4. Example of the Times-C's workflow.

TABLE III
TIME DISTRIBUTION OF THE THREE STAGES IN K-SHAPE

Aggregation and Alignment	Center Calculation	Assignment
19.5%	27.6%	52.7%

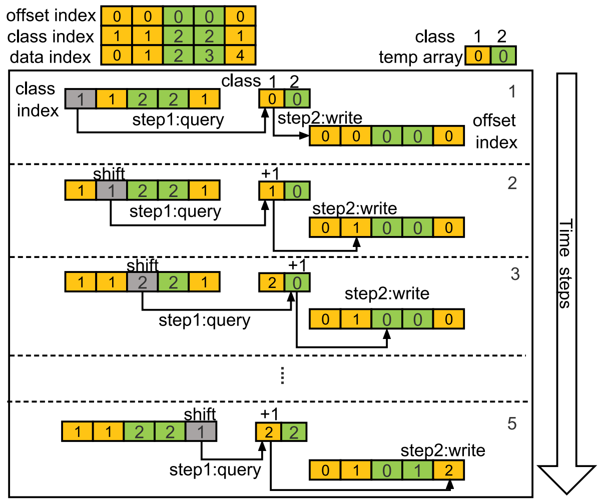


Fig. 5. Serial method to get the intra-class offset of time series.

parallelism strategies or methods for the three stages. Based on the observation, in order to reduce the execution time of the whole algorithm, it is necessary to design different acceleration strategies for *each of the 3 components*.

IV. TIMES-C: A GPU-BASED CROSS-LAYER PARALLEL OPTIMIZATION METHOD

A. Overview

Fig. 4 provides an example showing the workflow of Times-C algorithm using GPU. The algorithm includes three main phases: 1) data aggregation and alignment, 2) class center Calculation, and 3) class assignment. The basic description of optimizing the three parts is as follows:

First, Algorithm 1 used the simplest two-level loop traversal to aggregate data, while its parallel efficiency is low. A better choice is to sort the data according to its class. However, due to the high dimensionality of the time series and the multiple movements of each sequence in the sorting process, this improvement is still inefficient. In this work, we will design an efficient hash algorithm to minimize data movement and fully use the L1 cache during the movement process.

Second, for the class center computation, we try to fully explore the parallelism of this phase. The discussion in Section II shows that the calculation of each class is independent. Hence, the calculation within each class can be executed in parallel. However, the number of classes is often small, the amount of computation within the class is large, and the number of threads is limited for each thread block, which makes it inefficient to make a thread block correspond to a class. To enhance the parallelism of the class center extraction, we adopt CUDA streams to build a two-level parallel structure between intra-parallel and inter-parallel. The first level of parallelism is intra-class parallelism, and the second level of parallelism is inter-class parallelism.

Third, for the class assignment phase, we observe that the similarity calculation is equivalent to the multiplication of high-dimensional vectors. To fit the calculation task to the GPU hardware architecture, we use three-dimensional blocks and threads to calculate the similarity of the offset sequence.

B. Parallel Data Aggregation and Alignment

To effectively utilize GPU, we have designed a parallel hashing method for the data aggregation phase, which consists of two steps. Step 1 is to determine the location of each data hash in parallel, and Step 2 executes a parallel hash sorting at each time point.

1) Serial Intra-Class Offset Calculation: For hash sorting, it includes two parts: intra-class offset calculation and data mapping. We first introduce the serial version of intra-class offset calculation. Specifically, serial intra-class offset calculation is to calculate the offset of each data in each class.

Fig. 5 shows a serial method to get the intra-class offset of time series, which needs the index of each time series and its class belongs($data\ index$ and $class\ index$ at the upper left of Fig. 5). This method initializes the $offset\ index$ to zero and declares a temporary variable table (the $temporary\ array$ at the top right of Fig. 5). Next, it sequentially queries the table

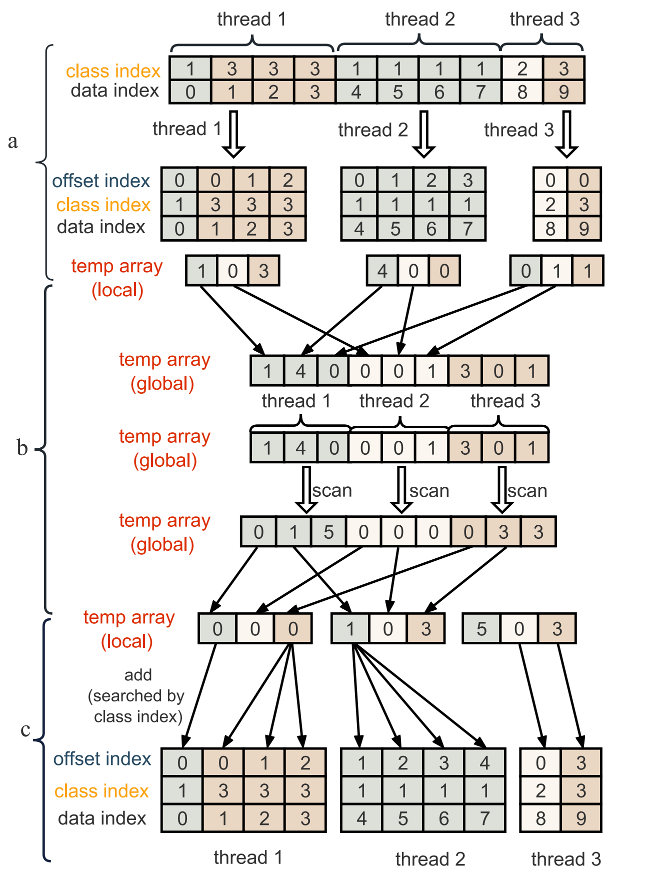


Fig. 6. Example of parallel acquisition of time series data offsets. (a) The method shown in Fig. 5 is executed by each thread. (b) The *temp array* are reordered and performing the scanning operation. (c) Each *offset index* is added to its corresponding *temp array*.

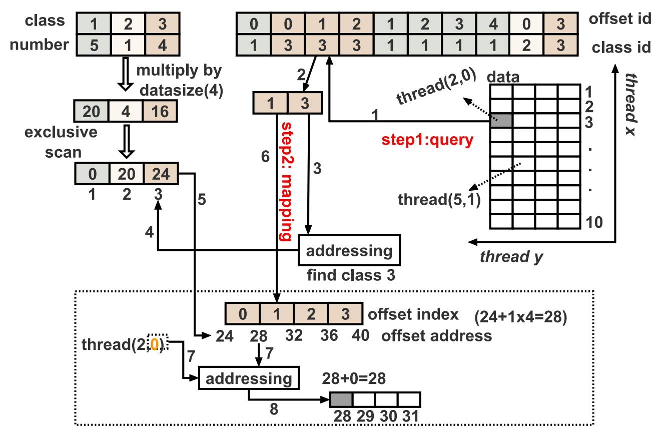


Fig. 7. Example of parallel hash sort. *number* represents the number of sequences in the class. *class index* represents the class to which each time series belongs. Here, we build threads into two dimensions so that one thread corresponds to one time point.

according to *class index* to which the data belongs and then assigns the queried data value to the offset of the current data. Further, the value of the corresponding class is added one in the table. After obtaining the intra-class offsets of all the data, the time series within a class are hashed using the class index and the intra-class offset of the data.

2) Parallel Hash Sort: Based on the serial method shown in Fig. 5, we design a parallel method on GPUs. In order to obtain the intra-class offset of each time series, a simple idea is to individually assign one thread for one data. However, as multiple threads may read or write the same offset in the temporary variable table at the same moment, this idea inevitably results in memory-access conflict. Although the atomic operation can be used to avoid the conflict, the atomic operation is usually time-consuming, and multiple instructions can not be executed simultaneously by one atomic operation. Inspired by the idea of division and merger, we propose an efficient parallel method for obtaining the time series data offsets based on bucket segmentation.

As shown in Fig. 6, the data is divided into several buckets. Each bucket has its fixed size, and each bucket is assigned to a thread. Assume that there are 10 time series data, and the number of classes is 3. The class index is denoted by *class index* in Fig. 6. First, we calculate the intra-class offset of the data according to the data index and its class index. In the process of obtaining the offset within the class, we set the number of buckets to three, and each bucket is assigned to an individual thread. Each thread applies the serial method shown in Fig. 5 to get the intra-class offset independently, which leads to the relative offset of each data (Fig. 6(a)). Next, we reorder *local temporary array* in each thread (Fig. 6(b)), and aggregate the *local temporary array* of the same class to obtain *global temporary array* which is updated by the exclusive scan operation [31]. The element in the updated *global temporary array* represents the starting index of the same class's data (with the same color) in different buckets. Further, *global temporary array* is remapped back to *local temporary array* for each thread. In order to obtain the required global offsets, the offset array *offset index* is added with *local temporary array* (Fig. 6(c)). Specifically, each thread traverses the data in each bucket in turn, and queries the class i corresponding to the data, and then adds the offset corresponding to the data and the ith value in the *local temporary array*. Finally, the desired offset array is obtained.

Assume that the length of each time series data is 4. Based on global offset arrays, we can perform parallel data hash sorting, as shown in Fig. 7. In order to realize the parallel sorting, We distribute all the threads along two dimensions, where the dimension in the x direction corresponds to each row, and the dimension in the y direction corresponds to each column. Each number in the data can be determined by the thread id in the x direction and y direction. The position of each data will be determined by querying the offset index and the class index. For example, to hash the first number of the third time series data, we query the offset index and class index corresponding to the third data. Its class index is 3, and the offset index is 1. We find the starting address of the third class based on the class index, where the starting address of each class is obtained by performing thrust exclusive scan [32] on all classes. Next, we use offset index to determine the intra-class offset address of the data to be mapped. The intra-class offset of the queried data is 1. We multiply the offset by the data size and add the queried class start address (24) and the thread y dimension id to get the position after the data hash. It should be noted that the address corresponding to each number can be calculated in parallel. As the number of each data is adjacent, the memory access can be consolidated.

Algorithm 2: $X_s, indexScan = gpuMove(X, C, index)$.

```
Input:
  $X$ is an m-by-n matrix containing $m$ time series.
  $class\_index$ is a 1-by-$m$ vector containing the classes of each time series.
  $C$ is the old center of each class.
Output:
  $X_s$ is the time series after aggregation and offset.
  $indexScan$ represents the corresponding starting position of each class.
1: //step 1 Data Aggregation.
2: $offset = Get\_offset(class\_index)$
3: $numclass = count\_each\_class(class\_index)$
4: $indexScan = exclusive\_scan(numclass)$
5: $X_s = hash(class\_index, indexScan, offset, X)$
6: //step 2 Data Alignment.
7: for $i = 1$ to $k$ do
8:    $begin = indexScan[i-1]$
9:    $end = indexScan[i]$
10:   //gpuNCC and gpuAssign show in Algorithm 4 and Fig. 11, it will be described in Section IV-D
11:   if $begin != end$ then
12:     $X_t = gpuNCC(X_s[begin : end], C_i)$
13:     $\_, s = gpuAssign(X_t)$
14:     //Set $end - begin$ threads, each thread corresponds to a time series data in $X_s[begin : end]$. For time series data $j$, $j \in [begin, end]$, perform the following operations in parallel(Next two lines)
15:     $s[j] = s[j] - length(X_s[begin + j])$
16:     Shift $X_s[begin + j]$ by (1), the shift parameter is $s[j]$
17:   end if
18: end for
```

3) Algorithm Details: Algorithm 2 shows the process of data aggregation and alignment. First, we obtain the intra-class offset for each time series (Line 2). Second, we calculate the size of each class in the $class\_index$ and store it in the array $numclass$ (Line 3). Third, we perform parallel scanning on $numclass$ to obtain the starting position $indexscan$ of each class (Line 4). Using $indexscan$, $index$, and $class\_index$, we can execute the parallel hash sorting approach to get the sorted data X_s (Line 5). Next, the alignment of the data is performed. Initially, we determine the presence of data within the class (Lines 8-11). If data is found within the class, the functions $gpuNCC$ and $gpuAssign$ are utilized to obtain the corresponding position s for each time series that requires an offset (Line 13). Subsequently, the data is offset based on the obtained positions s (Lines 14-16). The

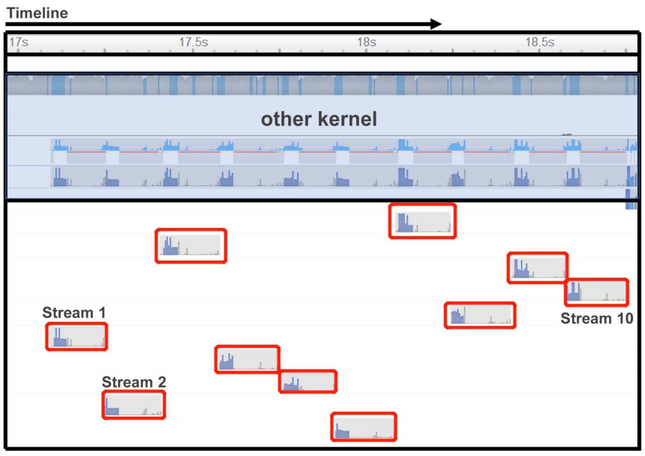


Fig. 8. Profiling on the stream schedule of using cuSOLVER to calculate class centers.

detail functions $gpuNCC$, and $gpuAssign$ will be presented in Section IV-D.

C. Parallel Class Center Calculation

The essence of class center calculation is an eigenvector calculation. Although NVIDIA has provided a linear algebra library cuSOLVER, it is still inefficient to directly use cuSOLVER to solve the class center due to two facts.

1) cuSOLVER for finding the matrix eigenvectors is based on either Jacobi iteration or QR decomposition. Jacobi iteration has better performance on small and medium matrices, while QR decomposition is more suitable for large matrices. As the size of each class center matrix is closely related to the length of the time series, which is determined by the high-dimensional characteristics, the center matrix is often large, which means that it is more appropriate to use QR decomposition to deduce the class center. But, current cuSOLVER does not provide any QR function for a batched eigenvector calculation of large matrices. For this, we can use CUDA streams to overlap different cluster center calculations. In fact, in Fig. 8, each red box represents a stream. It is obvious that the stream schedule still can not achieve a good overlapping effect, because each QR decomposition consumes large shared-memory resources of GPU, which limits the parallelism.

2) As only the eigenvector corresponding to the maximum eigenvalues is useful for each class center calculation, it is unnecessary to calculate all the eigenvectors. However, cuSOLVER does not support the dedicated computation of the eigenvector corresponding to the maximum eigenvalue, which will inevitably cause computational waste.

Consequently, our design combines the power iteration [33] and cuSOLVER. Specifically, the power iteration can directly find an eigenvector corresponding to the eigenvalue with the maximum absolute value. As the power iteration is not memory-consuming, the CUDA stream can be used to optimize the pipeline.

Algorithm 3: $C_n = gpuCenter(X_s, indexScan)$.

```
Input:
  $X_s$ is the time series after aggregation and offset.
  $indexScan$ represents the starting position of each class.
Output:
  $C_n$ is the new center of each class.
1: //$I$ is the identity matrix, $O$ is a matrix with all ones
2: $Q = cuBLAS(I - \frac{1}{m} \cdot O)$
3: for $i = 1$ to $k$ do
4:   $begin = indexScan[i-1]$
5:   $end = indexScan[i]$
6:   if $begin != end$ then
7:     //All tasks within this $if$ statement will be assigned
       to the $i$th stream.
8:     $X = X_s[begin : end]$
9:     $S = cuBLAS(X^T \cdot X)$
10:     $M = cuBLAS(Q^T \cdot S \cdot Q)$
11:     $len = length(time\ series)$
12:     $v = random\_init()$
13:     //Power iteration
14:     for $j = 1$ to $len$ do
15:       $v = cuBLAS(M \cdot v)$
16:       $v = \frac{v}{\|v\|_2}$
17:     end for
18:     $lda = \frac{M \cdot v \cdot v}{v \cdot v}$
19:     $left = M \cdot v$
20:     $right = lda \cdot v$
21:     if $!sign\_equal(left, right)$ then
22:       $lda = -1 \cdot lda$
23:       $left = -1 \cdot left$
24:     end if
25:     if $lda \leq 0$ or $right \neq left$ then
26:       $v = Max\_cuSOLVEREig(M)$
27:     end if
28:     $C_n[i] = v$
29:   end if
30: end for
```

Algorithm 3 shows the process of solving the class center. First, we iterate through each class of data to determine if the ith class contains data (Lines 2-6). If the data is present, we then assign the task of processing this data to the ith stream (Line 7). Next, we obtain the matrix M using (8) (Lines 8-10) and use the power iteration method to find the eigenvector corresponding to the largest absolute eigenvalue of M (Lines 11-17). For the detail of power iteration, please refer to [33]. Moreover, we choose the length of the time series as the number of iterations of the power iteration. Given that the maximum eigenvalue of the matrix may be a repeated root, we need to check both the accuracy of the eigenvalue and the accuracy of the corresponding eigenvector computation (Lines 18-25). If the eigenvalue is less than or equal to zero or the eigenvector calculation is incorrect (Line 25), we use cuSOLVER to find the eigenvector corresponding to the maximum eigenvalue (Line 26) and use it as the new centroid for the class.

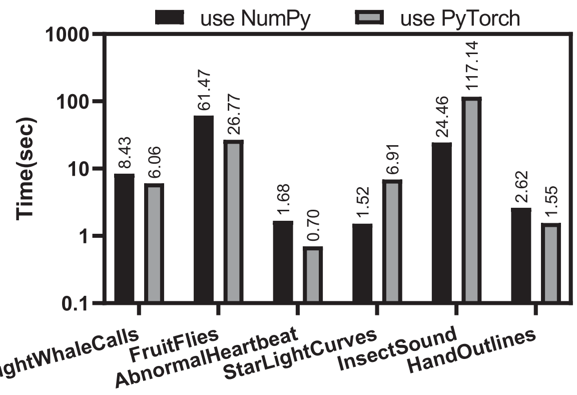


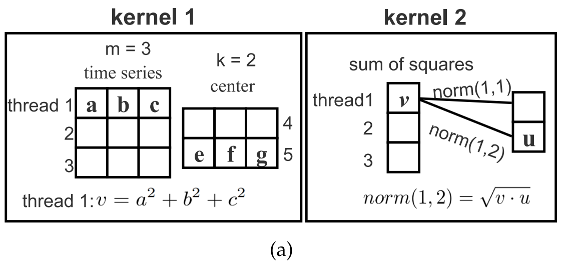
Fig. 9. Comparing the runtime of the k-Shape algorithm on an NVIDIA A100 GPU using PyTorch and on a multi-core CPU using NumPy for similarity calculation. The dataset is described by Table V in Section V.

D. Parallel Class Assignment

After the center of each class is gotten, we have to divide each data into its own class. This process requires the similarity measure. Similarity calculation is essentially an element-wise multiplication operation of multiple high-dimensional vectors. Although PyTorch has supported the element-wise multiplication for high-dimensional vectors on GPUs, the direct usage of PyTorch needs frequent kernel calls during the similarity calculation of each time series and each class center, which would involve the heavy overhead of kernel startup. For example, assume that the number of time series is 10,000, and the number of class centers is 10. Using PyTorch will call the kernels almost 100,000 times. Such expensive kernel calls will seriously degrade the performance. Fig. 9 shows the comparison of k-Shape execution time between using PyTorch on GPU and using NumPy on multi-core CPU for parallel class assignment. Compared to using NumPy on CPU, using PyTorch on GPU does not yield a significant acceleration effect and, in some cases, even takes more time. In this work, we develop a batched method to solve time series similarity. Specifically, we use three-dimensional blocks and three-dimensional threads for the element-wise multiplication of vectors to achieve high performance on GPU.

1) Parallel Similarity Calculation: During the similarity calculation, the cross-correlation can be translated to an element-wise vector multiplication using fast Fourier transform. Although cuFFT library [36] can be applied for the parallel fast Fourier transform and inverse Fourier transform, their overheads are trivial compared with the element-wise multiplication of vectors. Hence, our design for the similarity calculation focuses on the efficient implementation of element-wise vector multiplication on GPUs.

To reduce the kernel startup overhead, we combine the normalization operation and element-wise complex vector multiplication. According to (3), we multiply the L_2 norm of each time series data and the L_2 norm of each class center in parallel (Fig. 10(a)) before the vector multiplication. Specifically, we set two kernels, the first kernel sets $m + k$ threads where m is the number of time series data, and k is the number of class centers to



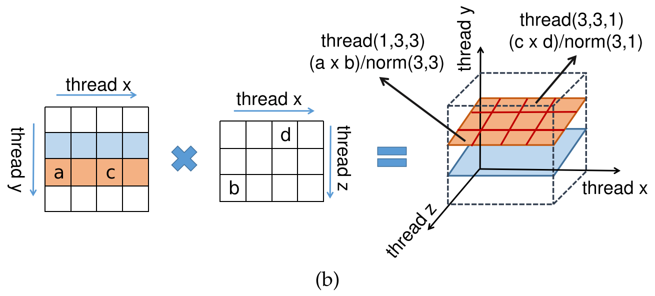


Fig. 10. Similarity calculating using three-dimensional blocks and threads. (a) Calculate the L_2 norm product of time series and class center. (b) The similarity calculations of all time series data and all class centers are parallelized.

obtain the sum of squares of each time series and class center. In the second kernel, we set m threads ($m >> k$) to multiply each time series by the L_2 norm of all class centers. Fig. 10(b) shows the parallel process of solving the similarity of all time series data. Here, we set the block and thread to three dimensions. The direction x of the block is the mapping of the length of the time series, the direction y of the block is a mapping of the number of time series data, and the direction z of the block represents the mapping of the number of class centers. The term $norm(i, j)$ denotes the L_2 norm between the ith time series data and the jth class center. To achieve the appropriate parallel granularity, each thread corresponds to a complex multiplication.

Algorithm 4: $X_t = gpuNCC(X, Y)$.

```
Input:
  $X$ is an m-by-n matrix containing $m$ time series.
  $Y$ is an k-by-n matrix containing $k$ time series centriod.
Output:
  $X_t$ is a $m$-by-$k$-by-$w$ tensor containing the Similarity of
  each time series with each center.
1: $length = 2^{nextpower2(2 \cdot length(x)-1)}$
2: Get the $L_2$ norm of $X$ and $Y$
3: $X_f = cuFFT(X, length)$
4: $Y_f = cuFFT(Y, length)$
5: $X_t = complex\_dot3D(X_f, Y_f)$
6: $X_t = cuFFTInv(X_t)$
```

Algorithm 4 presents the calculation process of similarity array. First, for time series data X and class center data Y, we obtain the L_2 norm between each data and each center, which is used as the normalized denominator (Line 2). Second, we perform fast Fourier transform on both X and Y using cuFFT library, and then implement the element-wise vector

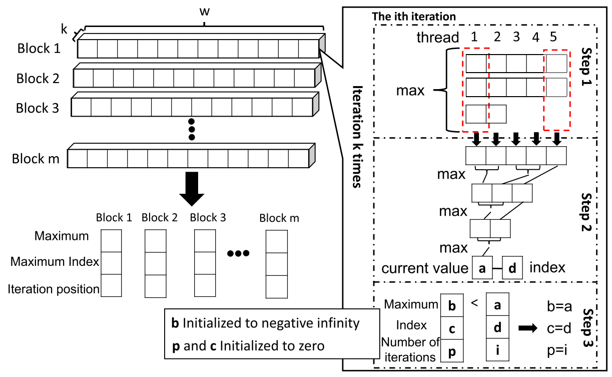


Fig. 11. gpuAssign: Get maximum similarity and its index. We assume that the number of time series is m, the length of the similarity array w is equal to 12, the number of classes is k, and the number of threads in a thread block is 5.

multiplication in the frequency domain using the approach above based on a three-dimensional block and thread on GPUs (Lines 3-5). Finally, through the inverse Fourier transform (Line 6), we obtain the similarity array which describes the distances between each time series and different class centers.

2) Maximum Similarity Extraction: The similarity array owns the size of $m \cdot k \cdot w$, where m is the number of time series data, k is the number of class centers, and w is the length transformed according to (5). The array represents the similarity measurement of each data in X and Y under all offsets. To determine which class each data belongs to, we set up $m \cdot bsize$ threads, where $bsize$ is the size of a thread block. Each thread block processes a data block with size of $k \cdot w$ in the similarity array, as shown in Fig. 11. In the figure, we assume that the size of $bsize$ is 5.

For each thread block, $bsize$ threads search the maximum value and its corresponding index in the array with the size of w in parallel. It is an iterative process. Each block is iterated for k times. Each iteration is divided into three steps. In the first step, each thread looks for the maximum value of its $id + j \cdot bsize(j = 0, 1, 2 \ldots ; id + j \cdot bsize < w)$ position. In the second step, every two adjacent threads get a maximum value and obtain the maximum value of this iteration and its index. In the third step, we compare the current iteration maximum with the historical maximum. If the current maximum is larger, we will save the current maximum value, index and iteration number. It is worth noting that in the function $gpuMove$, the value of the passed-in parameter k is one. However, in the class assignment phase, the parameter k is the number of clusters to be clustered. In addition, we will obtain two arrays. The first array $shift$ records the relative position j ($0 \leq j < w$) of the maximum similarity, which is applied to $gpuMove$. The second array $index$ records the iterative position i ($0 \leq i < k$) of the maximum similarity, which is applied to class assignments.

E. Parallel k-Shape Algorithm

Combined with the previous sub-problem processing, we propose a parallel k-Shape algorithm, i.e., Times-C, which is shown in Algorithm 5. First, we set the class centers of k classes as zero

Algorithm 5: $index = gpuKshape(X)$.

```
Input:
  $X$ is an m-by-n matrix containing $m$ time series.
Output:
  $index$ is a 1-by-$m$ vector containing the classes of each
  time series.
1: $C = \{0\}$
2: $index = rand()$
3: for $i = 1$ to $maxIter$ do
4:     $oldIndex = index$
5:     $X_s, indexScan = gpuMove(X, C, index)$
6:     $C_n = gpuCenter(X_s, indexScan)$
7:     $X_t = gpuNCC(X, C_n)$
8:     $index, \_ = gpuAssign(X_t)$
9:     if $index == oldIndex$ then
10:        $break$
11:    end if
12: end for
```

TABLE IV
ALGORITHMS AND HARDWARE DETAILS

configurations	algorithms
(1) NVIDIA A100-PCIE-40GB(Ampere), 6,912 CUDA cores @ 1.41 GHz, CUDA Version: 11.6. (2) Intel(R) Xeon(R) Gold 6354, 36 processors @ 3.00GHz, 256 GB Memory.	(1) k-Shape-CPU [11], (2) k-Shape-GPU [45], (3) tslearn-kshape [40], (4) dtw-kmeans [40], (5) Times-C (this work).

(Line 1), and then randomly initialize the class of each time series to a number between 1 and k (Line 2). Next, an iteration process is executed. In each iteration, the center of each class is calculated in parallel (Lines 5-6), and then each time series is re-grouped into a new class according to the similarity between it and the class centers (Lines 7-8). Further, the iteration process above is repeated until the iteration termination condition is met (Lines 9-10). The termination condition of the iteration is that either two-adjacent iterations achieve the same clustering result, or the iteration reaches $maxIter$ times.

V. EVALUATION

A. Experiment Setup

In this section, we make a comprehensive evaluation of the Times-C algorithm. We use the UCR and UAE databases developed by Ruiz et al. [37], the UCR database developed by Dau et al. [38], and urbansound8k dataset[39] commonly used for time series classification. These datasets have become an essential resource for the time series data mining community. At least one dataset from the archives is used in at least thousands of published papers, and they collect time series data from different fields, including electrocardiograms, medical images, spectra, sensors, animal sounds, etc. We have selected some classic datasets for our performance evaluation and analysis. The details are shown in Table V.

TABLE V
BENCHMARK DATASETS

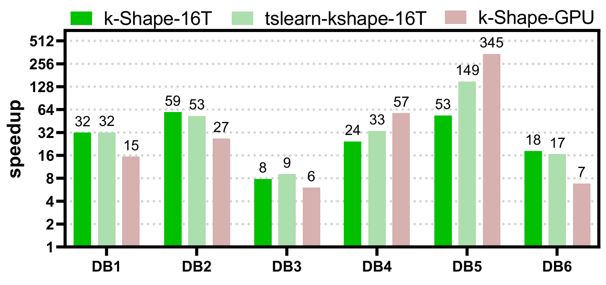
Dataset	Data Size	Length	No. of Classes
RightWhaleCalls	12896	4000	2
FruitFlies	34518	5000	3
AbnormalHeartbeat	606	3053	5
StarLightCurves	9236	1024	3
InsectSound	50000	600	10
HandOutlines	1370	2709	2
urbanSound8k	8732	-	10
MosquitoSound	139883	3750	-

The calculation cost of the k-Shaped algorithm is affected by three parameters: the size of the dataset, the length of the time series, and the number of clusters. Therefore, we selected different configurations to test the overall performance improvement of Times-C algorithm. For example, For the dataset $InsectSound$, the data size is much larger than the length. We measure the acceleration ratio when the data size is dominant. On the contrary, if the composition of data is small-scale, high dimension (dataset $AbnormalHeartbeat$, data size is much smaller than length). We measure the acceleration ratio when the data length is dominant. $RightWhaleCalls$ and $FruitFlies$ tests the speedup of algorithm execution on large-scale datasets.

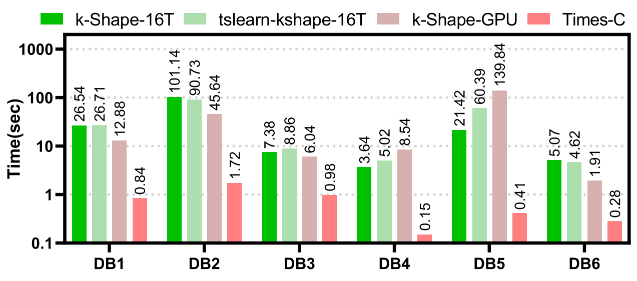
We compared the CPU and GPU implementations of Algorithm 1 provided by Paparrizos et al. [11], with the GPU version implemented using PyTorch. Furthermore, we also compared the k-Shape algorithm and the dtw-kmeans algorithm implemented in the popular time series analysis framework tslearn [40]. It is worth noting that the k-Shape algorithm implemented in tslearn differs from Algorithm 1 in the initial iteration. Table IV shows the details of the hardware configuration and the algorithms tested. The dtw-kmeans algorithm will be used for accuracy comparison in Section V-D. Additionally, for the CPU algorithm, we selected the Intel(R) Xeon(R) Gold 6354 18-core, 36-thread as the hardware testing environment. Since the original paper of k-Shape algorithm compared it with other algorithms in a 16-thread environment, we selected 16 threads as the baseline for CPU comparison. In addition, to more comprehensively compare CPU algorithms, we also added a comparison of the execution time of CPU algorithms under different thread numbers in Section V-G. For the GPU algorithm, we used the NVIDIA A100 as the hardware platform.

B. Overall Performance Comparison

In the experiment, we set the maximum number of iterations to ten and used the average running time of a single iteration under ten iterations as the comparison baseline. Moreover, all datasets of our experiment reached the maximum number of iterations during the execution. Fig. 12 shows the speedup achieved by Times-C compared to other algorithms. $RightWhaleCalls(DB1)$, $InsectSound(DB5)$, and $FruitFlies(DB2)$ are large datasets. Compared to other datasets, the Times-C algorithm has achieved better performance on these datasets, indicating that the Times-C algorithm performs better on large datasets. The data sizes of the



(a)



(b)

Fig. 12. (a) The acceleration achieved by the Times-C algorithm compared to other algorithms. (b) The execution time of different algorithm. 16T stands for CPU with 16 threads. DB1-DB6 represent the first six datasets in the benchmark dataset.

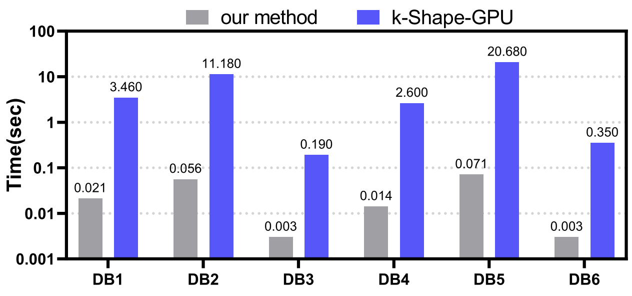


Fig. 13. Our algorithm's data aggregation and alignment time is compared with that of the k-Shape-GPU algorithm. DB1-DB6 represent the first six datasets from the benchmark dataset collection.

datasets $StarLightCurves(DB4)$ and $InsectSound(DB5)$ are much larger than their data lengths. Compared to the k-Shape-GPU algorithm, the Times-C algorithm obtained a higher speedup on these two datasets. In addition, the length of the $AbnormalHeartbeat(DB3)$ dataset is much larger than its size. Compared to the k-Shape-GPU algorithm, the speedup obtained by the Times-C algorithm is not ideal, indicating that Times-C may have higher performance gain in data size rather than data length. Therefore, in Section V-E, we conducted two separate tests. The first test aimed to evaluate the speedup based on data length, using the $urbanSound8k$ dataset for comparison. The second test focused on the speedup related to data size, utilizing the $MosquitoSound$ dataset for comparison.

C. Optimization Strategy Analysis

Although we employ some acceleration libraries to improve the performance of the algorithm, however, the performance improvement is closely related to our proposed hash sorting method, the two-level parallel structure, and the similarity of three-dimensional thread computation. To demonstrate this, we compared our algorithm with current popular GPU algorithms. Additionally, in this section, we used the first six datasets from the benchmark dataset collection.

First, we compared the data aggregation and alignment parts of the Times-C algorithm with those of the k-Shape-GPU algorithm. Fig. 13 shows the average runtime of a single iteration for both algorithms over ten iterations. It can be seen that, the

TABLE VI
PERFORMANCE OF DATA AGGREGATION

Dataset	L1 Hit Rate	L2 Hit Rate	SM Occupancy	gld_throughput
DB1	92.82%	92.85%	74.73%	800.97Gb/s
DB2	92.31%	92.34%	73.1%	875.68Gb/s
DB3	92.18%	92.67%	73.06%	848.89Gb/s
DB4	90.77%	92.97%	71.13%	780.55Gb/s
DB5	92.83%	92.89%	76.64%	862.69Gb/s
DB6	90.75%	92.78%	71.94%	784.88Gb/s

TABLE VII
PERFORMANCE OF SIMILARITY CALCULATION

metrics	L1 Hit Rate	SM Occupancy	GFLOPS	Proportion
ours	91.04%	78.23%	252.84	-
PyTorch_1	37.31%	78.12%	200.27	<4%
PyTorch_2	9.7%	6.13%	7.86	>96%

Times-C algorithm can speed up data aggregation and alignment by more than a hundred times compared to the k-Shape-GPU algorithm on different datasets. The results indicate that the optimization of data layout in Times-C has achieved a significant acceleration compared to the k-Shape-GPU algorithm. This further reflects the effectiveness of the Times-C algorithm.

Table VI shows the L1/2 cache hit rate, SM occupancy rate, and throughput obtained in the data aggregation section. It is worth noting that we only measured the metrics for data aggregation. As the data alignment part can be achieved through similarity calculation, the metric for data alignment can be considered as the metric for similarity calculation in Table VII that we will discuss later. The dataset we use is the first six datasets of the benchmark dataset. It can be seen from the table that our proposed method has fully utilized L1 cache, which is why we do not use shared memory in data aggregation. In fact, since each data is only moved once, configuring shared memory does not lead to performance improvement.

Fig. 14 shows the average running time of the Times-C algorithm and the eigenvector calculation in a single iteration using only cuSOLVER under ten iterations. From the figure, it can be seen that on different datasets, compared to the algorithm that only uses cuSOLVER to compute the cluster centroids, the Times-C algorithm achieves the lowest acceleration of 1.6 times

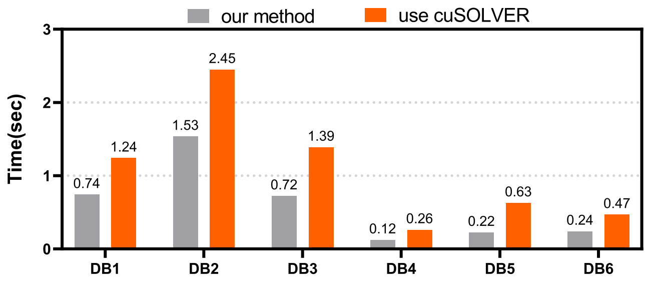


Fig. 14. Comparing our method with the time spent calculating class centers using cuSOLVER only.

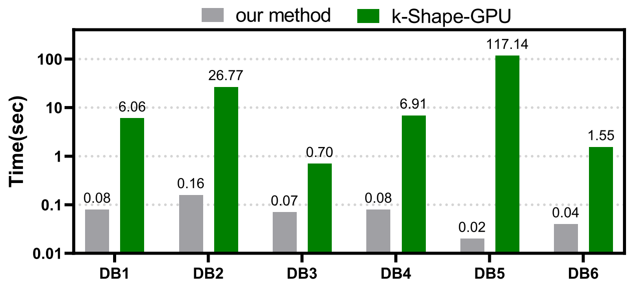


Fig. 16. Compare our method with the time it takes to calculate the similarity using only PyTorch.

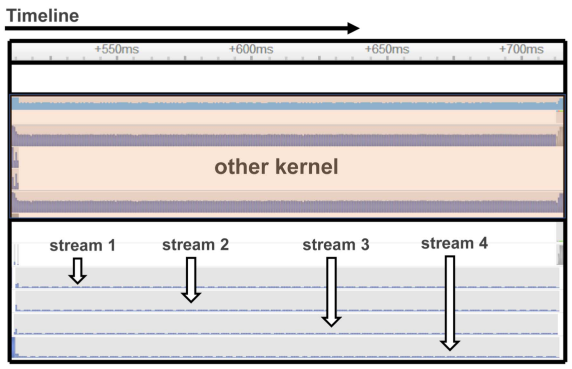


Fig. 15. Our class center extraction method overlaps on the time axis, with different gray areas representing different streams.

TABLE VIII
RAND INDEX OF DIFFERENT ALGORITHMS

Dataset	Times-C	Times-C-tsl	dtwKmeans	tslearn-Kshape
DB7	0.84	0.85	0.82	0.85
DB8	0.82	0.84	0.82	0.83
DB9	0.86	0.85	0.85	0.85
DB10	0.97	0.97	0.96	0.97
DB11	0.95	0.96	0.96	0.96
DB12	0.79	0.79	0.51	0.80
DB13	0.89	0.93	0.94	0.94
DB14	0.80	0.84	0.83	0.86
DB15	0.86	0.88	0.91	0.88
DB16	0.82	0.90	0.91	0.90
DB17	0.95	0.80	0.95	0.79
DB18	0.96	0.91	0.50	0.90

on the $FruitFlies$ dataset, and the highest acceleration of 2.86 times on the $InsectSound$ dataset. Compared with cuSOLVER, the reason why the algorithm we proposed has no significant performance improvement is due to two aspects: on the one hand, the number of clusters is often relatively small, and the degree of overlapping computation of the two-level parallel structure we involved is related to the number of clusters. On the other hand, when the eigenvalues obtained by the power iteration are less than zero, our method will call cuSOLVER additionally, and the additional calls will also generate a lot of time overhead. However, in general, our method can still achieve nearly three times the acceleration compared with cuSOLVER.

Fig. 15 shows the parallelism of Times-C class center extraction methods when solving different classes. The gray areas in the figure represent different class center solutions. It can be seen that compared with the case where cuSOLVER almost does not overlap, our proposed method overlaps well on the time axis.

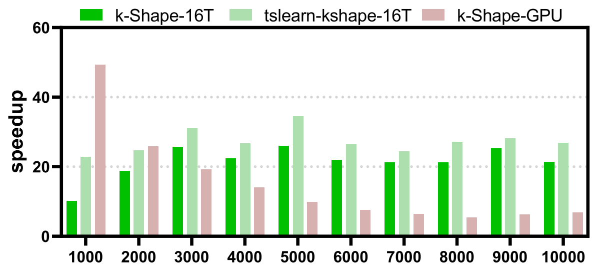
Fig. 16 displays the average runtime for similarity computation between Times-C algorithm and k-Shape-GPU algorithm (i.e., using PyTorch element-wise operations) for ten iterations. It can be seen from the figure that the Times-C algorithm has a significant performance improvement compared with the PyTorch algorithm on different datasets. The reason for such high acceleration is that PyTorch will frequently call Element-Wise operation in the process of calculating similarity. This frequent calling will cause significant overhead, particularly when dealing with large size datasets. In such cases, the performance of PyTorch may even be worse than that of the CPU. However, the algorithm we proposed can avoid this shortcoming and obtain better performance.

Table VII shows the performance comparison between PyTorch and our similar calculation methods. Here, we select the average performance of the benchmark dataset to show. In the analysis of PyTorch, because a kernel function needs to be executed many times, the results of each execution are not the same. In the analysis process, we observed two manifestations of PyTorch in the execution process. The performance indicators displayed by these two manifestations are shown in the table PyTorch_1 and PyTorch_2. The time proportion of these two forms of representation is shown in the table, as can be seen from PyTorch_2 is much higher than PyTorch_1. This also means that the inefficient kernel function in the execution process dominates PyTorch. On the other hand, even with the more efficient PyTorch_1, its performance also lags behind the method we proposed.

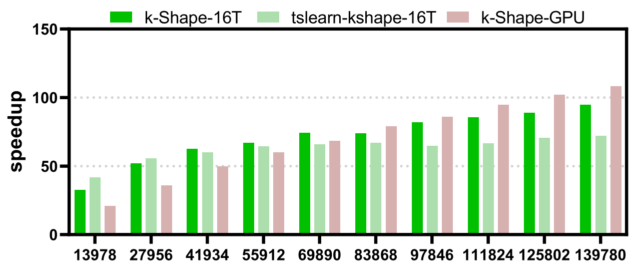
D. Clustering Results

In this section, we will evaluate the clustering results of Times-C. First, we used the benchmark dataset for clustering performance tests and compared it with the k-Shape algorithm and dtw-kmeans algorithm. However, during the test, we found that the DTW algorithm has a long execution time (for example, when testing the dataset $InsectSound$, it takes several days for one iteration). Therefore, we selected some small datasets from the UCR database for the test.

Table VIII shows the comparison of clustering scores between the Times-C algorithm and other algorithms. Whereas



(a)



(b)

Fig. 17. (a) The speedup achieved by the Times-C algorithm for different sequence lengths. (b) The speedup achieved by the Times-C algorithm for different time series size.

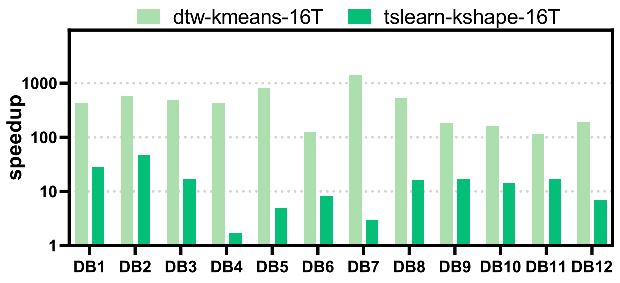


Fig. 18. Compared to the tslearn-kshape and dtw-Kmeans algorithms, our algorithm achieves an acceleration ratio. DB1 to DB12 represent the top-to-bottom datasets in Table VIII.

DB7-DB18 represent the datasets uWaveGestureLibrary_Z, uWaveGestureLibrary_Y, uWaveGestureLibrary_X, NonInvasiveFatalECG_Thorax2, NonInvasiveFatalECG_Thorax1, MoteStrain, MALLAT, InsectWingbeatSound, FacesUC, FaceAll, Adiac, and ECGFiveDays from the UCR Time Series Archive database, respectively. Besides, we use the rand index[41] as the evaluation index, which is a value between zero and one. The closer the rand index is to one, the better the clustering effect.

It can be seen that the Times-C algorithm has little difference in clustering results from dtw-kmeans algorithm in most cases, and sometimes even better than the DTW algorithm. The difference between the Times-C algorithm and tslearn-kshape is due to the difference in initial iteration. Specifically, in the first iteration, instead of randomly initializing the class to which each time series belongs, tslearn-kshape randomly selects k time series as the center of each class. By using the similarity measure in Algorithm 1, the center j ($j <= k$) closest to time series i is determined, and time series i is divided into j classes, which are then iterated repeatedly in the extraction of class centers and division of time series in the Algorithm 1. For the Times-C algorithm, in the first step of the iteration, the classes for each time series are randomly initialized first. On this basis, duplicate iteration class center extraction and time series allocation are performed. This difference in initialization results in differences in calculation results. When we use the same initialization method as the tslearn-kshape strategy(Times-C-tsl), we obtain results that are similar but not identical to tslearn-kshape. These slight differences are due to the random initialization.

Fig. 18 shows the acceleration ratio obtained by the Times-C algorithm, and it can be seen from the figure that the Times-C algorithm is typically hundreds of times faster than the dtw-kmeans algorithm. Compared to tslearn-kshape, the Times-C algorithm is typically more than ten times faster.

E. Impact of Data Size and Data Length

In this part, we use $urbanSound8k$ and $Mosquitosound$ datasets to test the impact of data length and data size on the algorithm execution time, respectively. We still set the maximum number of iterations as ten and take the average single iteration time for comparison.

$urbanSound8k$ is the original audio dataset. We sample it through the librosa library[42] of python. Starting from the length of 1,000, the sampling stride is 1,000, and the sub-datasets with the sequence length of 1,000 to 10,000 are obtained respectively. We compared our algorithm with both CPU and GPU algorithms. Fig. 17(a) shows the speedup obtained by the Times-C algorithm compared to other algorithms. As can be seen from the figure, the speedup of Times-C gradually decreases as the sequence length increases. However, when the sequence length reaches 7,000, the speedup obtained by the Times-C algorithm stabilizes and does not increase with the increase of the sequence length, which indicates that the Times-C algorithm has a certain stability. On the other hand, when the sequence length is relatively short, the performance gain obtained by the Times-C algorithm over the k-Shape-GPU algorithm is high, while decreasing the sequence length will make the sequence size more prominent. This indicates that when the sequence size is dominant, the k-Shape-GPU algorithm cannot achieve good performance benefits.

$Mosquitosound$ dataset is a dataset with a size of 139,883 and a length of 3,750. In order to test the impact of data size on the execution time of the algorithm, we segment the dataset and divide the original dataset into its sizes of $\frac{1}{10}, \frac{2}{10} \cdots \frac{10}{10}$. Similarly, we compared our algorithm with both CPU and GPU algorithms. We chose $k = 2$ for the test. Fig. 17(b) shows the speedup obtained by the Times-C algorithm compared to other algorithms. It can be seen from the figure that as the sequence size increases, the speedup of Times-C gradually increases. In addition, Times-C obtains a much higher performance gain in sequence size than in sequence length. Compared to the k-Shape-GPU algorithm, the acceleration speed obtained by the Times-C algorithm increases more significantly as the sequence size increases. This finding confirms our analysis of the similarity calculation in Section V-C, which suggests that

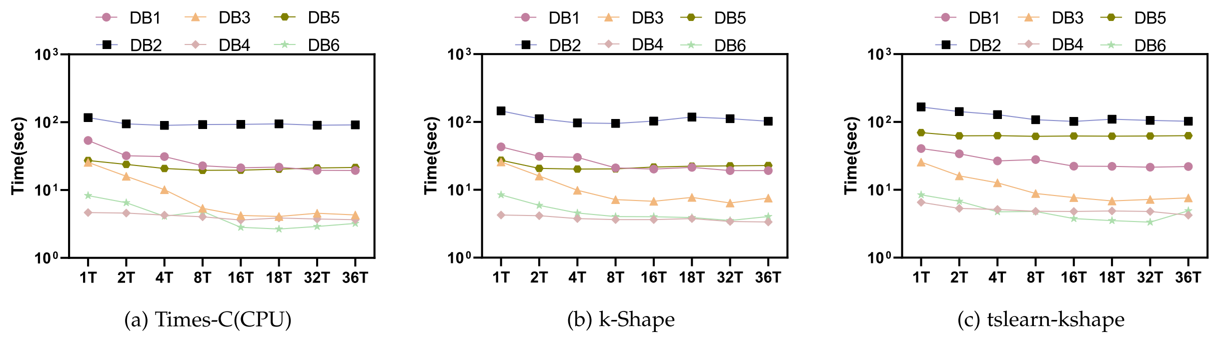


Fig. 19. Runtime of CPU version algorithm under different threads, DB1 to DB6 represents the first six benchmark datasets.

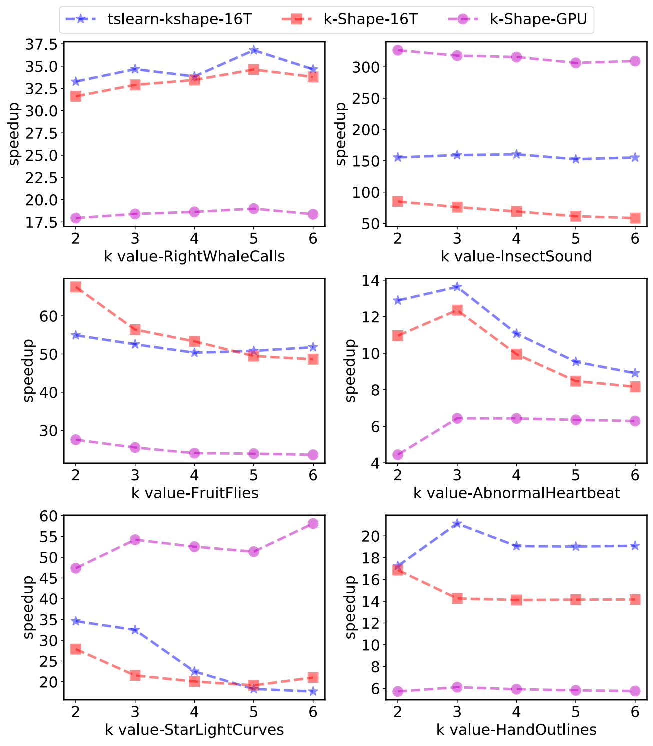


Fig. 20. Speedup achieved by the Times-C algorithm for different k value.

the k-Shape-GPU algorithm performs poorly in environments where the sequence size is large.

F. Impact of k-Value

Considering that the number of clusters also has a great impact on the execution time of the algorithm. In this part, we tested the speedup of the Times-C algorithm compared to the CPU algorithm and GPU algorithm under different numbers of clusters. Fig. 20 shows the speedup of the Times-C algorithm when the number of clusters increases from 2 to 6. It can be seen that on most datasets, when k increases from 3 to 6, the speedup of most datasets does not fluctuate greatly. This reflects the stability of the Times-C algorithm. Generally, our implemented Times-C algorithm on large-scale datasets achieves a speedup of usually $30\times$ compared to the CPU algorithm. For example, datasets $RightWhaleCalls$, $FruitFlies$, and $InsectSound$. Moreover, compared to the k-Shape-GPU algorithm, our implemented Times-C algorithm achieves a speedup of over $10\times$ on large-scale datasets. In addition, on small datasets, compared with the CPU algorithm, our proposed Times-C algorithm achieves over $6\times$ speedup, while compared with the k-Shape-GPU algorithm, it achieves over $4\times$ speedup.

G. The CPU Version of Times-C

Although our proposed Times-C algorithm was designed based on some abstract models of GPUs, the methods we proposed are not limited to GPUs. The parallel method of class center extraction we proposed can be migrated to CPUs. To provide a more comprehensive comparison, we extended our proposed parallel class center extraction method to CPUs and compared it with the other algorithms running on CPUs. Fig. 19 shows the average running time per iteration of the algorithms under different threads.

From the figure, it can be seen that under a single thread, Times-C and k-Shape have similar performance. However, as the number of threads increases, the performance of the Times-C algorithm is slightly better than that of the k-Shape algorithm, but not significantly so. This is because our designed parallel similarity calculation relies on GPU-specific programming models. Therefore, we optimized only the extraction of class centers on the CPU, while keeping the similarity calculation part consistent with the original k-Shape algorithm. The similarity calculation is also a time-consuming part of the clustering process. Moreover, CPUs typically have a limited number of threads available for parallelization. In class center calculation, our design optimizes the data layout so that the calculation of each class can be parallelized independently from the others. However, using NumPy to calculate a single class on CPUs can occupy all available threads, and parallelizing between classes does not yield significant performance gains. Nevertheless, by optimizing the data layout, our proposed algorithm achieved performance improvements on CPUs as well.

H. Code Availability

The proposed Times-C GPU version algorithm is implemented in C++ language using the CUDA framework. The

Times-C CPU version algorithm is implemented in Python. The source code is available at[1]

VI. Related Work

k-medoids[43] and k-means[14] are very popular time series clustering algorithms in shape-based time series clustering methods. Unlike k-means, which uses the average value of each cluster selected as the new center, k-medoids instead uses the centrally located object in the cluster, the center point (medoids), as the reference point. This makes k-medoids more robust than k-means in the presence of noise and outliers. However, since k-medoids need to constantly find the minimum distance from each point to all other points to correct the cluster center, this greatly increases the time for cluster convergence. In addition, the euclidean distances used by traditional k-medoids and k-means in sequence similarity calculation also lead to them not capturing the sequence offset effectively on the time axis[46], [47], [48]. Because DTW algorithm [27] is a good time series alignment algorithm, many time series clustering algorithms optimize and enhance the DTW algorithm on the basis of k-means and k-medoids, and k-DBA[44] averages the center of gravity on the basis of DTW. To get the center of the aligned sequence. However, the time complexity of aligning two time series using DTW algorithm is $\mathbf{O}(n^2)$, which leads to high computational costs. Therefore, in recent years, some work has been done to accelerate DTW algorithm.

On CPU: Srikanthan et al. [15] exploited the use of CPU clusters to accelerate DTW. They divide a separate long sequence into subparts and assign them to different processors in the CPU cluster. Each processor executes the DTW algorithm in parallel. Takahashi et al. [16] designed a parallel DTW algorithm using multi-core processors. In their work, each sub time series will be assigned to different cores, and the pairwise comparison between sequences will be carried out in each core. In these two parallel implementations, data transmission limits the performance of the algorithm because a sub sequence needs to be transmitted in different cores.

On GPU: Zhu et al. [17] proposed a new algorithm to effectively find similar local subsequences between time series. This algorithm is parallelized on the GPU, and the diagonal sequence technology is used to optimize the memory access mode in the GPU. Zhang et al. [18] proposed a lower-bound DTW search algorithm on GPU. They combined DTW and KNN search to obtain the k best matches. The matrix generation stage and path search stage are separated into two kernels. However, this approach, while offering finer-grained parallelism, can lead to increased communication overhead. This is because the result matrix needs to interact with the path search stage.

Considering that in the process of generating the result matrix by DTW algorithm, the next step of calculation needs to rely on the results of the previous step, which means that parallelism can only occur in a single step, so this feature fundamentally limits the execution efficiency of the algorithm. Therefore, we consider accelerating the k-Shape algorithm. Compared with DTW algorithm, k-Shape algorithm has a smaller calculation cost under the same accuracy as DTW algorithm and has no front-back constraints on parallelism. This means that compared with DTW algorithm, k-Shape algorithm can achieve finer parallel granularity.

[1][Online]. Available: https://github.com/URSA-OpenXPU/Times-C

VII. Conclusion

Time series data analysis is increasingly becoming an important data processing method not only in everyday life but also in large-scale scientific facilities. In this paper, we propose Times-C, a GPU-based cross-layer optimization method and efficient implementation of the k-Shape algorithm. We analyze the k-Shape algorithm and divide it into three main tasks: data aggregation, class center extraction, and similarity calculation. We have performed parallel optimization for these three tasks, achieving a significant acceleration effect. Moreover, the parallel method we propose is not limited to time series clustering. For instance, certain techniques in class center extraction can be employed to solve problems related to maximum eigenvalues and their corresponding eigenvectors. Finally, through comprehensive evaluations, Times-C demonstrates significant improvements compared to the software framework running on CPU and GPU. We believe that the method we propose will become one of the key approaches for analyzing time series data in the data space.

References

[1] R. H. Shumway, D. S. Stoffer, and D. S. Stoffer, *Time Series Analysis and its Applications*, New York, NY, USA: Springer, 2000.

[2] Z. Bar-Joseph et al., “A new approach to analyzing gene expression time series data,” in *Proc. 6th Annu. Int. Conf. Comput. Biol.*, 2002, pp. 39–48.

[3] E. J. Ruiz et al., “Correlating financial time series with micro-blogging activity,” in *Proc. 5th ACM Int. Conf. Web Search Data Mining*, 2012, pp. 513–522.

[4] K. Uehara and M. Shimada, “Extraction of primitive motion and discovery of association rules from human motion data,” in *Progress in Discovery Science*, Berlin, Germany: Springer, 2002, pp. 338–348.

[5] T. W. Liao, “Clustering of time series data–A survey,” *Pattern Recognit.*, vol. 38, no. 11, pp. 1857–1874, 2005.

[6] J. Yang et al., “k-shape clustering algorithm for building energy usage patterns analysis and forecasting model accuracy improvement,” *Energy Buildings*, vol. 146, pp. 27–37, 2017.

[7] J. Thalheim et al., “Sieve: Actionable insights from monitored metrics in distributed systems,” in *Proc. 18th ACM/IFIP/USENIX Middleware Conf.*, 2017, pp. 14–27.

[8] T. Jarábek, P. Laurinec, and M. Lucká, “Energy load forecast using S2S deep neural networks with k-shape clustering,” in *Proc. IEEE 14th Int. Sci. Conf. Informat.*, 2017, pp. 140–145.

[9] G. Bello-Orgaz et al., “Marketing analysis of wineries using social collective behavior from users’ temporal activity on Twitter’ temporal activity on Twitter,” *Inform. Process. Manag.*, vol. 57, no. 5, 2020, Art. no. 102220.

[10] Z. Karevan and J. A. K. Suykens, “Transductive LSTM for time-series prediction: An application to weather forecasting,” *Neural Netw.*, vol. 125, pp. 1–9, 2020.

[11] J. Paparrizos and L. Gravano, “k-shape: Efficient and accurate clustering of time series,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1855–1870.

[12] C. W. Tan, G. I. Webb, and F. Petitjean, “Indexing and classifying gigabytes of time series under time warping,” in *Proc. SIAM Int. Conf. Data Mining*, 2017, pp. 282–290.

[13] P. Huijse, P. A. Estevez, P. Protopapas, J. C. Principe, and P. Zegers, “Computational intelligence challenges and applications on large-scale astronomical time series databases,” *IEEE Comput. Intell. Mag.*, vol. 9, no. 3, pp. 27–39, Aug. 2014.

[14] J. MacQueen, “Classification and analysis of multivariate observations,” in *Proc. 5th Berkeley Symp. Math. Statist. Probability*, 1967, pp. 281–297.

[15] S. Srikanthan, A. Kumar, and R. Gupta, “Implementing the dynamic time warping algorithm in multithreaded environments for real time and unsupervised pattern discovery,” in *Proc. Int. Conf. Comput. Commun. Technol.*, 2011, pp. 394–398.

[16] N. Takahashi, T. Yoshihisa, and Y. Sakurai, "A parallelized data stream processing system using dynamic time warping distance," in *Proc. Adv. Intell. Syst. Comput.*, 2009, pp. 1100–1105.
[17] H. Zhu et al., "Developing a pattern discovery method in time series data and its GPU acceleration," *Big Data Mining Anal.*, vol. 1, no. 4, pp. 266–283, 2018.
[18] Y. Zhang, K. Adl, and J. Glass, "Fast spoken query detection using lower-bound dynamic time warping on graphical processing units," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2012, pp. 5173–5176.
[19] M. S. Nobile et al., "Graphics processing units in bioinformatics, computational biology and systems biology," *Brief. Bioinf.*, vol. 18, no. 5, pp. 870–885, 2017.
[20] S. Peng and S. X. D. Tan, "GLU3.0: Fast GPU-based parallel sparse LU factorization for circuit simulation," *IEEE Des. Test*, vol. 37, no. 3, pp. 78–90, Jun. 2020.
[21] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Med. Phys.*, vol. 38, no. 5, pp. 2685–2697, 2011.
[22] C. Ma, L. Wang, and X. Q. Xie, "GPU accelerated chemical similarity calculation for compound library comparison," *J. Chem. Inf. Model.*, vol. 51, no. 7, pp. 1521–1527, 2011.
[23] Y. Katznelson, *An Introduction to Harmonic Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[24] S. Chetlur et al., "cuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[25] G. Lu, W. Zhang, and Z. Wang, "Optimizing depthwise separable convolution operations on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 70–87, Jan. 2022.
[26] S. Popov, J. Günther, and H. P. Seidel, "Stackless KD-tree traversal for high performance GPU ray tracing," *Comput. Graph. Forum*, vol. 26, no. 3, pp. 415–424, 2007.
[27] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proc. 3rd Int. Conf. Knowl. Discov. Data Mining*, 1994, pp. 359–370.
[28] R. N. Bracewell and R. N. Bracewell, *The Fourier Transform and its Applications*, New York, NY, USA: McGraw-Hill, 1986.
[29] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
[30] G. H. Golub and C. F. Van Loan, *Matrix Computations*. Baltimore, MD, USA: JHU Press, 2013.
[31] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, Nov. 1989.
[32] NVIDIA, thrust, 2023. [Online]. Available: https://docs.nvidia.com/cuda/thrust/index.html
[33] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*. Boston, MA, USA: Cengage Learning, 2015.
[34] NVIDIA, cuSOLVER, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cuSOLVER/index.html
[35] NVIDIA, cuBLAS, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html
[36] NVIDIA, cuFFT, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html
[37] A. P. Ruiz et al., "The great multivariate time series classification bake off: A review and experimental evaluation of recent algorithmic advances," *Data Min. Knowl. Discov.*, vol. 35, no. 2, pp. 401–449, 2021.
[38] H. A. Dau et al., "The UCR time series archive," *IEEE/CAA J. Automatica Sinica*, vol. 6, no. 6, pp. 1293–1305, Nov. 2019.
[39] J. Salamon, C. Jacoby, and J. P. Bello, "A dataset and taxonomy for urban sound research," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 1041–1044.
[40] R. Tavenard et al., "Tslearn, A machine learning toolkit for time series data," *J. Mach. Learn. Res.*, vol. 21, no. 118, pp. 1–6, 2020.
[41] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *J. Amer. Statist. Assoc.*, vol. 66, no. 336, pp. 846–850, 1971.
[42] B. McFee et al., "librosa: Audio and music signal analysis in Python," in *Proc. 14th. Python Sci. Conf.*, 2015, pp. 18–25.
[43] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ, USA: Wiley, 2009.
[44] F. Petitjean, A. Ketterlin, and P. Gançarski, "A global averaging method for dynamic time warping, with applications to clustering," *Pattern Recognit.*, vol. 44, no. 3, pp. 678–693, 2011.
[45] J. Paparrizos and L. Gravano, "k-Shape source code," 2023. [Online]. Available: https://github.com/TheDatumOrg/kshape-python
[46] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah, "Time-series clustering–A decade review," *Inf. Syst.*, vol. 53, pp. 16–38, 2015.
[47] A. S. Shirkhorshidi, S. Aghabozorgi, and T. Y. Wah, "A comparison study on similarity and dissimilarity measures in clustering continuous data," *PLoS One*, vol. 10, no. 12, 2015, Art. no. e0144059.
[48] P. Esling and C. Agon, "Time-series data mining," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–34, 2012.

Xun Wang received the degree from the University of Tsukuba, Tsukuba, Japan, in 2015. She is currently an associate professor with the China University of Petroleum, Qingdao, China. Her current research interests include DNA computing, cell signaling, bioinformatics and high-performance computing.

Ruibao Song is currently working toward the master's degree with China Petroleum University, Qingdao, China. His research interests include machine learning and parallel computing.

Junmin Xiao received the doctor of philosophy degree in computational mathematics from the Institute of Computational Mathematics and Scientific/Engineering Computing, Chinese Academy of Sciences. He is currently an associate professor with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include high performance computing, parallel optimization, communication avoiding, parallel and distributed deep learning.

Tong Li is currently working toward the master's degree with China Petroleum University, Qingdao, China. Her research interests include parallel computing and bioinformatics.

Xueqi Li (Member, IEEE) received the PhD degree from ICT under the supervision of Prof. Ninghui Sun. He is currently an assistant professor with ICT, CAS. From 2018 to 2020, he was also co-advised by Prof. Yuan Xie as a joint PhD student with the SEAL Lab, UCSB. He also serves as a research assistant for strategic studies with CAS & CAE. His research interests include Domain-specific PIM accelerators and AI for Life Sciences. He has published several papers in various conferences and journals (MICRO, HPCA, DAC, PPoPP, SC, *Journal of Computer Science and Technology*). He is a member of the ACM.