2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

DTW-Based Subsequence Similarity Search on AMD Heterogeneous Computing Platform

Sitao Huang, Guohao Dai, Yuliang Sun, Zilong Wang, Yu Wang, Huazhong Yang

E.E. Dept. TNLIST, Tsinghua University
{hst10, dgh10, sunyl09, wang-zl11}@mails.tsinghua.edu.cn,
{yu-wang, yanghz}@mail.tsinghua.edu.cn

***Abstract*—Subsequence similarity search is one of the most common subroutines in time series data mining algorithms. According to previous studies, Dynamic Time Warping (DTW) distance is the best distance measurement in many domains. However, the high computational complexity of DTW distance makes it a critical bottleneck in many subsequence similarity search applications. In some applications, the performance of software implementation still could not meet the high requirements of applications. Under the circumstance, some hardware implementations of DTW-based algorithms were proposed in the data mining community, using GPUs and FPGAs. In this paper, we propose a full system implementation for subsequence similarity search on AMD heterogeneous computing platform, including complete normalization pre-processing, two kinds of improved lower bound for pruning, and a novel segmented parallel DTW calculation process, which efficiently utilizes the capacity of CPU and GPU on the platform. Our work achieves one to two orders of magnitude speedup compared to software implementation and several times speedup to other GPU or FPGA implementations.**

Keywords—DTW; GPU; Subsequence Similarity Search; Time Series; Heterogeneous Computing

I. INTRODUCTION

As the need of discovering new knowledge from large databases increases, a series of methods that efficiently deal with big data are attracting more and more researchers' interest. Subsequence similarity search is one of the most common subroutines in data mining algorithms, because of its universality of revealing potential similarity. For example, in astronomy research, the time series data produced by aerial array every day is of a huge amount, and searching specified sequence pattern in the large database can help discovering typical astronomical phenomena. This is a typical subsequence similarity search problem.

In solving subsequence similarity search problem, an essential component that needs to be decided is the similarity measurement, or the distance measurement, of two subsequences of time series. An intuitive idea is to choose the Euclidean distance as the subsequence similarity measurement. However, because of the unsatisfactory conditions of reality, such as the noise in sensors, the measurement errors in experiments and so on, time series may be distorted, and the accuracy of similarity search may not be high enough. Under the circumstance, Dynamic Time Warping (DTW) distance was proposed. DTW distance allows time series to have a limited level of distortion (usually called *warping*), and DTW distance calculation algorithm even indicates how to restore the warped subsequence according to the given pattern. After exploration and application during a long time, DTW distance is now considered to be the best distance measurement [1].

However, the calculation of DTW distance is a computation-massive process. Classic DTW calculation takes so much time that the efficiency of applications based on DTW distance is strictly limited. To improve the performance of DTW-based applications, researchers have proposed many accelerating approaches so far. To reduce the number of actual DTW distance calculation times during a search as many as possible is a common approach to higher performance of subsequence similarity search in software. Accelerating approaches in software includes lower bound pruning, early abandoning strategy, indexing and so on. However, DTW distance calculation still takes too much time, up to more than 80% of the whole similarity search time [2].

Recently, some hardware accelerating approaches are proposed to further improve the performance of subsequence similarity search. The hardware used in these approaches includes FPGA and GPU. FPGA can realize calculation with fine-grained parallelism and the degree of parallelism could be very high. However, modification and reconfiguration on FPGA are relatively difficult, which prevent it from being used in the areas where problems are frequently updated. Besides, programming FPGA requires relevant electronic circuit design knowledge, and because of the requirements for professional knowledge in utilizing FPGA, only a few approaches using FPGA can really achieve efficient acceleration. The parallelism that GPU achieves is often coarse-grained, and the degree of parallelism achieved may not be as high as FPGA. But the programming model of GPU has been well developed, and it is convenient to implement algorithms on GPU. There are some versions of DTW implementation on GPU, but most of them do not efficiently utilize the potential parallelism in the algorithm. Besides, most of them use a single GPU to accelerate DTW distance calculation, without exploiting the full potential of the whole platform.

In this paper, we propose a new heterogeneous framework to perform subsequence similarity search, which can achieve the similarity search of high speed and of high flexibility. The major contribution of our study are as follows:

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.149

(1) We design and implement a full subsequence similarity search system on AMD heterogeneous computing platform, including complete normalization pre-processing, two kinds of improved lower bounds for pruning, and a novel segmented parallel DTW calculation process. We carefully select, improve and implement the available algorithms so that they best fit the heterogeneous platform and efficiently utilize the capability of both CPUs and GPUs on the heterogeneous platform.

(2) We propose a longer-length normalization method and improve lower bound techniques so that subsequences distribution is sparse but continuous in segments after lower bound pruning. Thus the longer-length normalization method can be applied on the segments, which enables the coarse-grained segmented parallel DTW distance calculation. In this way, the computation-reuse property of neighbor candidate subsequences is utilized and the DTW process is further parallelized.

(3) Our work achieves a high speedup while guaranteeing necessary flexibility and scalability. The speedup is up to one to two orders of magnitude when compared to software implementation and several times compared to the other GPU or FPGA implementations.

The remainder of this paper is organized as follows. Section II gives a review on recent works on subsequence similarity search and DTW distance. Section III introduces the process of subsequence similarity search based on DTW distance and defines symbols. In section IV, we present our subsequence similarity search system architecture, including heterogeneous structure and subsequence similarity search system implementation on heterogeneous computing platform. In section V, we introduce our experimental settings and present our experimental performance in detail. And in appendix, we introduce the architecture of the AMD GPU that we use and the OpenCL programming model.

II. RELATED WORKS

The DTW distance is widely used in subsequence similarity search algorithms. Recently, much work has been done attempting to accelerate subsequence similarity search. However, most of these attempts are software-based techniques.

Y. Sakurai et al. proposed the SPRING algorithm [3] to accelerate DTW by exploiting the potential relationship of adjacent subsequences. However, subsequences were not allowed to be normalized according to the strategy used in the SPRING algorithm, which led to the degradation of performance of DTW in some cases. S. H. Lim et al. applied indexing techniques to subsequence similarity search, and realized the acceleration of searching process [4]. But the length of the pattern(or the query) should be specified when constructing index, and the other possible lengths of the pattern were not supported during the querying and searching stage. E. Keogh et al. [5] proposed a multiple-indexing technique to deal with various lengths of patterns, which would become quite complex when the variation range of pattern is wide. One more aspect of limitation of index technique is that it is difficult to construct index on streaming data.

Another acceleration technique for subsequence similarity search is lower-bounding. Lower bound is a kind of estimation of DTW distance and it can be strictly proved that it is less than the exact DTW distance, so if the lower bound of DTW distance exceeds the threshold, the exact value of DTW distance must also exceed the threshold. Generally, the computational complexity of lower bound is much less than that of DTW distance, so the estimated lower bound of DTW could be used to effectively prune off those subsequences that are obviously not similar to the given pattern before actually calculating the DTW distance. A. Fu, E. Keogh et al. proposed a kind of lower bound of DTW, *LB_Keogh*, which can prune off a considerable number of subsequences [6]. However, the performance of the lower bound *LB_Keogh* depends on the tightness of Sakoe-Chiba band constraint, which we will demonstrate later, and this kind of lower bound has high performance only when the constraint is tight enough. Besides, the application of the lower bound technique requires a proper trade-off between the tightness and the computational complexity of the lower bound, which limits the further speedup when using lower bound techniques.

The acceleration techniques discussed above are all software acceleration techniques. To further improve the efficiency of subsequence similarity search, using hardware is an effective approach. Many hardware implementations have been proposed during the last decade. S. Srikanthan et al. used CPU clusters to speedup DTW calculation [7], and N. Takhashi et al. used multi-cores [8]. M. Grimaldi et al. proposed an acceleration design using GPU [9]. However, these three hardware implementations only utilized the simple coarse-grained parallelism, i.e. they just simply distributed patterns, or subsequence of time series, to different computing units. And every computing unit completed a whole and independent DTW distance computation task. Obviously, the improvement in the performance of these implementations was limited since the DTW calculation of a single subsequence and pattern was not accelerated. Besides, this simple distribution technique added a heavy burden to the data transmission in the system. And data transmission was likely to be a bottleneck of the system using this kind of hardware acceleration techniques.

Y. Zhang et al. proposed a different GPU implementation [2] which utilized the potential fine-grained parallelism of DTW distance calculation. They firstly used parallel computing units to generate the distance matrix, i.e. every computing units calculated the distance of two tuples from time series and pattern respectively, so the elements of distance matrix were generated at the same time. Secondly, based on the generated distance matrix, different computing units searched for the DTW path of different subsequence at the same time, i.e. the search on a distance matrix was completed by a single computing unit, and this was a serial process. The core idea of this GPU acceleration design is to separate the potential parallel processing part from the other serial processing parts, and fully parallelize the parallel processing part. This is a great progress in the attempts to accelerate DTW distance algorithm. However, their hardware implementation did not utilize the

computation-reuse property between neighbor subsequences, which leaves space for further performance improvement.

D. Sart et al. proposed a GPU implementation and a FPGA implementation of DTW-based subsequence similarity search [11]. Their GPU implementation adopted simple coarse-grained parallelism strategy. Their FPGA implementation was said be the first one to accelerate DTW using hardware circuits. Their design utilized the potential fine-grained parallelism inside DTW distance calculation. They place processing units along the diagonals of the DTW distance matrix while searching for the DTW distance path, so that calculation of elements along diagonals can be done in parallel. So their FPGA implementation achieves a higher speedup of DTW distance algorithm. However, their work is created using C-to-VHDL tools, and the flexibility and scalability of their system were cut down, due to the lack of carefully designed circuit structure which fully utilizes characteristics of the algorithm.

In our previous work [10], we proposed a new system architecture for subsequence similarity search, and we designed a new circuit structure, PE-ring, to break the strong computation dependence between neighbor subsequences. PE-ring structure utilizes the fine-grained parallelism inside the DTW algorithm, and it can efficiently accelerate the DTW distance calculation. This FPGA implementation achieves a significant speedup compared to the best software implementation and existing hardware implementations.

Although hardware implementations using FPGA properly can achieve a high performance, programming FPGA requires professional knowledge about circuits and the modification and reconfiguration on FPGA are relatively difficult. Compared with FPGA, GPU programming is relatively easier due to the maturity of development tools and the design can be quickly updated without much cost. Out of this consideration, in this paper, we propose a new implementation of subsequence similarity search system on AMD heterogeneous computing platform which consists of different types of computing devices, such as CPU and GPU.

III. Problem Definition and Algorithm

In this section, we give a concise introduction to the process of subsequence similarity search, including its mathematical description and algorithm description.

A. Problem Definition

Subsequence similarity search is a process finding the subsequences which are similar to the given patterns from time series. Related concepts can be defined as follows.

Time Series: A time series S is a one-dimension sequence with a specified order, $S = s_1, s_2, \ldots, s_N$, where N is the length of the time series in problem. The element of time series, s_i, $1 \le i \le N$, is usually called a tuple. Tuples could be of any data types as long as the concept of distance between tuples has been defined in the problem. In most cases, tuples are double-precision float point numbers.

Subsequence: A subsequence $S_{i,k}$ of time series S is a shorter sequence starts at position i and ends at position k of time series S, that is, $S_{i,k} = s_i, s_{i+1}, \cdots, s_k$. Note that in our problem, subsequence is assumed to be a segment of continuous tuples in time series. We do not consider the subsequences made of discontinuous tuples.

Pattern: A pattern P is a sequence specified by the need of applications, used as a subsequence search template, $P = p_1, p_2, \cdots, p_M$, where M is the length of pattern. The subsequences that are similar to the pattern are the targets we want to find out.

Based on the definition above, we can draw a conclusion that if the length of a sequence is N, then it has $N - M + 1$ subsequences of length M. In the process of similarity search, a sliding window of size M is used to pick out every candidate subsequence. Then, the distances between pattern and each candidate subsequence are calculated, and based on the given distance threshold, candidate subsequence is determined to be similar or not to the given pattern.

There are various definitions of distance. The basic idea is to choose Euclidean distance as the distance measurement. As is mentioned in section I, because of the unsatisfactory conditions in reality, Euclidean distance was not so good to deal with time series similarity search, so Dynamic Time Warping distance was proposed. And there is increasing evidence that Dynamic Time Warping distance is the best distance measurement [1]. The definition of DTW distance and its calculation algorithm are illustrated below.

B. Algorithm and Parallelism Analysis

1) Dynamic Time Warping

Dynamic Time Warping (DTW) distance is a kind of distance measurement of two sequences. The main difference between DTW distance and Euclidean distance is that DTW distance also considers the cases where sequences are distorted.

For two sequences $S = s_1, s_2, \ldots, s_m$ and $P = p_1, p_2, \ldots, p_n$, the DTW distance of S and P is defined by the following recursive formula:

$$DTW(S, P) = d(m, n) \tag{1}$$

$$d(i, j) = dist(s_i, q_j) + \min \begin{cases} d(i-1, j) \\ d(i, j-1) \\ d(i-1, j-1) \end{cases} \tag{2}$$

$$d(0,0) = 0;\ d(i,0) = d(0,j) = \infty; \tag{3}$$

$$i = 1, 2, \ldots, m;\ j = 1, 2, \ldots, n$$

In the definition above, $dist(\cdot,\cdot)$ is the distance between two tuples, which could be defined as absolute distance or the squared distance, or other distance measurement.

The d in definition is a m-by-n matrix, whose elements can be regarded as the DTW distance of shorter subsequences of the two sequence.

The relationship between DTW distance, matrix d and the process of calculating DTW distance could be illustrated by Figure 1. It shows how the DTW distance is calculated and how DTW can recovery possible distortion of subsequences. The two series are pattern and candidate subsequence respectively. The calculation is equivalent to search the optimal path in a matrix, which is often called warping matrix. This optimal path in the warping matrix represents the correspondence of two tuples from pattern and candidate subsequence respectively. The green lines in the figure illustrate the boundary of the Sakoe-Chiba band constraint [1]. The Sakoe-Chiba band restricts the warping path in a band area, which is in fact a constraint on the distortion level of both pattern and candidate subsequence.

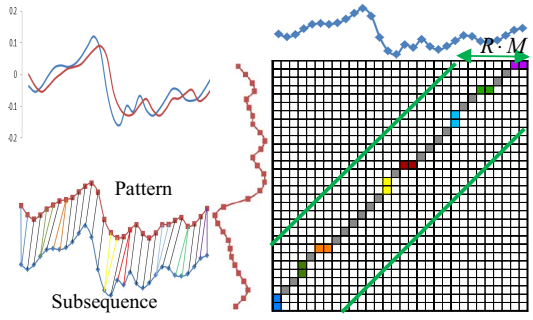

Figure 1. Illustration of DTW and Warping Matrix[10]

According to the definition of DTW distance above, the calculation process of DTW distance can be regarded as a kind of dynamic programming process. The data dependence in the calculation process is very strong, and the process tends to be serial. To accelerate the DTW distance calculation, we adopt a kind of strategy which exploits the potential parallelism in the DTW distance calculation and divide the calculation process into two different stages. The first stage is to calculate the distance of each pair of tuples and generate the distance matrix. The second stage is to find the DTW path in the generated DTW distance matrix. In the first stage, the calculation of every pair of tuples is completely independent, so it can be completed in parallel. The second stage of calculation is the serial part of the whole DTW distance calculation, and in this stage, the DTW distance matrix is searched and the DTW path is found serially. Based on the observation that the distribution of subsequences that need to calculate DTW distance are sparse but locally continuous in segments, we propose that several continuous subsequences to be joined as a segment and conducting DTW distance calculation together, which enables the segmented parallel DTW calculation. In this way, the potential parallelism will be further exploited. The work of Y. Zhang et al. in [2] also adopted similar DTW calculation strategy. However, they form a complete distance matrix for every candidate subsequence and search for the warping path respectively. Without the pruning of lower bound and adapted normalization, the computation-reuse property of neighbor subsequences is not utilized in their work, which limits the performance of their work.

2) Lower-Bound-based Pruning

Reducing the number of actual DTW distance calculation times is an effective strategy of software acceleration of DTW-based subsequence similarity search. One of the most common approaches is lower bound pruning.

Here "lower bound" means the lower bound of the DTW distance of subsequence and query, and it can be strictly proved in mathematics. In the process of subsequence similarity search, if the lower bound of a candidate subsequence exceeds the given threshold for the DTW distance of pattern and subsequences, the exact value of the DTW distance of that subsequence must be greater than the given threshold. The computational complexity of lower bound is much lower than that of DTW distance, so we can calculate the lower bound before actual DTW distance calculation and prune subsequences that are obviously different from the pattern.

There are several types of lower bound for DTW distance. We adopt the revised version of lower bound proposed by S.W. Kim et al. [12], which is called *LB_pDTW* in our previous work [10], and the lower bound proposed by E. Keogh [6], which is called *LB_Keogh*. The mathematical description of the two types of lower bound is shown below.

The lower bound *LB_pDTW* is the revised version of the lower bound proposed by S.W. Kim et al. Their lower bound uses the distance of the first and the last tuples of subsequence and pattern, and both the distance of the maximum and minimum values of subsequence and pattern. So the computational complexity of their lower bound is $O(n)$. To simplify the lower bound calculation and make pruning process more efficient, we only use the first c tuples and the last c tuples of the candidate subsequence and the pattern to calculate the lower bound. The parameter c can be adjusted according to the need of applications. We use these tuples to calculate a "partial" DTW distance, i.e. a DTW-like distance obtained by the first c tuples and the last c tuples of the candidate subsequence and the pattern, as is shown in the Figure 2.

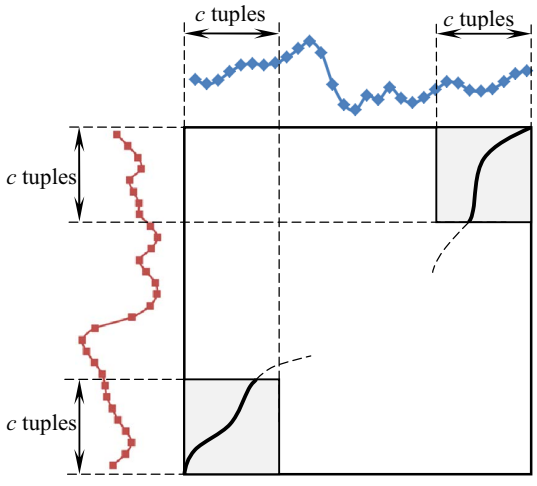

Figure 2. Illustration of *LB_pDTW* Calculation

As is illustrated in the figure above, the *LB_pDTW* calculation process is equivalent to search for a part of warping path only within the area formed by the first and the last tuples in pattern and candidate subsequence. So *LB_pDTW* is strictly a lower bound for DTW. In applications, the value c is usually very small ($1 \le c \le 6$ in our experiments), so we can easily calculate the lower bound *LB_pDTW* in a short time.

The computational complexity of *LB_pDTW* is $O(1)$ since the computing time needed is constant and not related to the length of the subsequence and the pattern. The calculation of *LB_pDTW* of every candidate subsequence and pattern is independent so the calculation can go on at the same time. Because the *LB_pDTW* calculation is relatively simple, there is no much space for parallelism acceleration inside *LB_pDTW* calculation.

The lower bound *LB_Keogh* creates an upper envelope and a lower envelope of the given pattern, and use the sum of the part that candidate subsequence exceeds these two envelopes to calculate the lower bound of DTW distance. The lower bound *LB_Keogh* can be calculated as follows.

The i^{th} tuple in the upper envelope UW and lower envelope LW of the pattern Q are defined as:

$$UW_i = \max(Q_{\max(1,i-r)}, \cdots, Q_{\min(i+r,n)}) \tag{4}$$

$$LW_i = \min(Q_{\max(1,i-r)}, \cdots, Q_{\min(i+r,n)}) \tag{5}$$

where $r = R \cdot M$ is the width of Sakoe-Chiba band. Note that lower bound *LB_Keogh* works only when Sakoe-Chiba band constraint exist, i.e. r does not equal to the length of pattern. So in the applications where Sakoe-Chiba band constraint should not be considered, the lower bound *LB_Eeogh* needs not to be calculated. With these two envelopes, the lower bound *LB_Eeogh* of DTW distance can be obtained using the following formula:

$$LB\_Keogh(C,Q) = \sum_{i=1}^{m} \begin{cases} (C_i - UW_i)^2, & C_i > UW_i \\ (C_i - LW_i)^2, & C_i < LW_i \\ 0, & otherwise \end{cases} \tag{6}$$

where C_i is the i^{th} tuple in the candidate subsequence.

During a searching process through a long time series, the given pattern does not change. Thus once the upper envelope and the lower envelope of the pattern are constructed, these two envelopes can be used throughout the whole searching process. For each candidate subsequence, *LB_Keogh* accumulates the part that candidate subsequence exceeds the upper envelopes or the lower envelopes. According to the definition of *LB_Keogh* above, the calculation of lower bound of a candidate subsequence can be accelerated exploiting its potential of fine-grained parallelism. Although the final accumulation process is serial, the distance calculation of every pair of tuples is independent to each other, and can be completed in parallel. The number of tuples is very large, so this is a high degree parallelism. We will explain our design in detail in section IV.

3) Normalization

In applications, due to the instability of the output of sensors, the time series may have some level of time-variant distortion. The distortion of time series may lead to the higher mistake rate and missing rate of subsequence similarity search. Consequently, the normalization preprocessing is necessary for both pattern and incoming data. The type of normalization used is usually z-normalization. In general, according to the definition of z-normalization, for a sequence of length n, $S = s_1, s_2, \cdots, s_n$, the elements of the normalized sequence $S_{normalized} = s_1', s_2', \cdots, s_n'$ could be calculated using the following formula,

$$s_i' = \frac{s_i - \mu}{\sigma},\ 1 \le i \le n, \tag{7}$$

where μ is the mean of the sequence S and σ is the standard deviation of the sequence S. According to the above definition, the mean of the sequence must be 0 and the standard deviation of the sequence must be 1 after z-normalization. In this way, the mean and the standard deviation of the pattern and the time series will be the same after normalization, so they can be compared and further processed together.

In our implementation, there are two stages containing the normalization processing. The first time normalization conducted is when pattern and time series data flow into the system and before lower-bound-based pruning. There normalization ensures that the input of lower-bound-based pruning is effective. To reduce redundant computation, the value of μ and σ could be updated according to the following formula, as the tuples in pattern and time series flow into our system one after another.

$$\mu = \frac{1}{n}\left(\sum_{i=1}^{k} s_i - \sum_{i=1}^{k-n} s_i\right) \tag{8}$$

$$\sigma^2 = \frac{1}{n}\left(\sum_{i=1}^{k} s_i^2 - \sum_{i=1}^{k-n} s_i^2\right) - \mu^2 \tag{9}$$

In this form, the computational complexity of normalization of subsequences except the first one is constant, not related to the length of subsequences.

The other time normalization conducted is after lower-bound-based pruning and those continuous not-pruned subsequences are joined into segments. In this stage, all tuples in the same segment are regarded as a longer subsequence and are normalized as a whole, which is based on the assumption that the statistical characteristics are of the same for the subsequences in one segment [10]. Here normalization is conducted on a segment of time series, containing more than one subsequence. This is a little different from the common normalization which is conducted on every single subsequence. However, in most practical applications, the effect of these two kinds of normalization is almost the same [10].

As is mentioned above, the first normalization processing of every subsequence could be completed within a constant time which is not related to the length of the subsequences, as tuples being read from the data source and flowing into our system. So this is a serial process and do not need further

parallelism design consideration. The second time of normalization processing is aimed at the segments made up of the continuous subsequences not pruned. So we can use different processing units to conduct the normalization of every segment separately, exploiting the potential coarse-grain parallelism of the normalization of different segments. However, inside each segment, normalization is a serial process. It is because that normalization processing requires the mean and the standard deviation of the tuples in the segment, and these two quantities can be directly obtained after a serial scan through the segment, which can be merged into the process of discovering segments of continuous subsequences. Although the mean and standard deviation can also be obtained using parallel techniques, it could be costly both in time and hardware resources to do so here. We will show normalization processing in detail in section IV.

IV. IMPLEMENTATION

In this section, we introduce our heterogeneous system structure of subsequence similarity search, including our GPU mapping of DTW calculation algorithm and scalability strategies. Some GPU relevant concepts and terminology are explained in Appendix.

A. System Architecture

The following figure illustrates the complete process of subsequence similarity search and our system architecture.

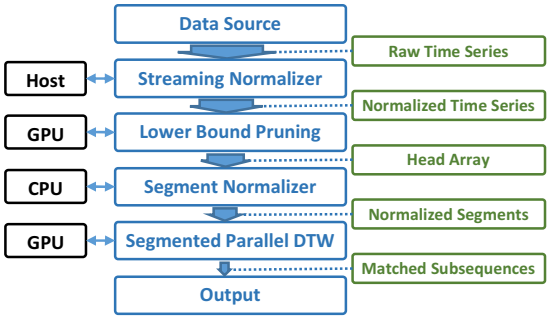

Figure 3. Our Heterogeneous System Architecture and Complete Process of Subsequence Similarity Search

As is shown in the figure above, the computation of the subsequence similarity search is achieved in both GPUs and CPUs. The GPU on our platform is the AMD Radeon HD 7970 GPU. And the processor of host in our experiment is a AMD A10-5800K APU with Radeon HD Graphics.

The process of subsequence similarity search can be easily divided into different stages, which correspond to the hierarchy of system architecture. This kind of hierarchical architecture guarantees the flexibility and scalability of the system. The system can well adapt to the different input parameters and the scale of the system can be adjusted according to the need of applications, which will be discussed later.

Different data structures are well designed to serve as the effective connection between different stages of the processing and the carrier of information between different parts of the system. Basically, subsequence similarity search could be regarded as a process of data filtering and information extraction. A specific component of the system takes in the outputted result of the previous processing component, and performs the specific process, and output the condensed information in a suitable data structure to the next processing component in the system.

B. Longer-Length Normalization

As is mentioned in the section III, there are two stages that need the normalization process. The normalizer is used to calculate the mean and standard deviation of every subsequence of the incoming time series. It should be noted that we does not really change the raw time series, but store the mean and standard deviation of every subsequences of the incoming time series, along with the raw time series. It is because that the subsequences of the time series are heavily overlapped, and it will cost a large amount of additional memory space to store those normalized time series. So we choose to only store the mean and standard deviation of the subsequences for the further analysis.

Out of consideration for the form of input data, which is inputted in sequence, we realize the normalizer associated with the input module, which means that the normalizer deals with raw time series as the time series coming in. As is mentioned in Section III, the cost of this implementation is very low, so a high data input rate could be easily realized. Because normalization is a serial process, we use host-end CPU to perform normalization. The experiments show that this strategy is good for improving processing speed.

C. Lower-Bound-Based Pruning

After the mean and standard deviation of every subsequence is calculated, the process of lower-bound-based pruning starts. The calculation is performed on GPU, out of consideration that it is a highly parallel process. Time series and the mean and standard deviation of every subsequence are sent to GPU, and every work-item performs the lower bound calculation for a subsequence. Considering the memory source on GPU is strictly limited, we divide the incoming time series into the blocks of about 100,000 to 150,000 tuples, with an overlap of length of the query. This length of the block is proved to be proper for the efficient performance of the GPU in our experiments. There are two types of lower bound pruning in our implementation, lower bound proposed *LB_pDTW* and lower bound *LB_Keogh* proposed by E. Keogh. Because the lower bound of every subsequence is calculated in parallel by specified work-item, it is very easy to develop a new version of kernel of different types of lower bound according to the need of applications.

According to the experiment results in [10], after pruning process using lower bounds, the distribution of left candidate subsequences is sparse. Figure 4 illustrates the pruning effect of lower bounds. The red points along the threshold line (distance = 40 in this case) indicate the subsequences that do not pruned off by any lower bound, which need further actual DTW distance calculation. We can find that those subsequences not pruned off usually distribute in different groups.

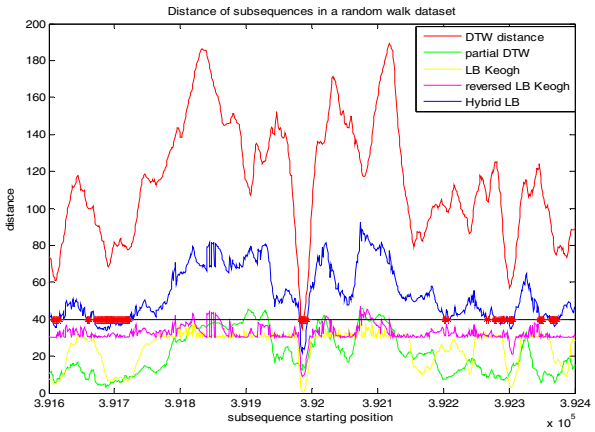

Figure 4. Pruning Effect of Lower Bounds[10]

We design a special data representation for the output of lower bound module. The output data structure of the lower bound kernel is a binary array. If an element in the array is "1", it means the corresponding position in time series is a head of subsequence that needs DTW distance calculation, otherwise, a "0" means the corresponding subsequence has been pruned off by the lower bound pruning process. Then, the binary array is interpreted by host-end CPU to an array that records the positions of the subsequences need to calculate DTW distance, which we call Head Array. It should be noted that the data returned by the lower bound kernel is a binary array, rather than an array records position. The process from a binary array to Head Array is performed by the host-end CPU, because it is a serial process, and CPU would perform better in this process than GPU. At the same time, the binary array flowing in is also used to form the segments that normalization will be performed on. And the mean and standard deviation of every segments are also calculated during this time. This process also counts the number of the subsequences that need to be calculated DTW distance with the query. This number will be used in the configuration of the number of work-items of the second kernel, which is used for DTW distance calculation.

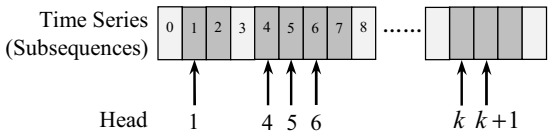

Figure 5. Illustration of Head Array

The meaning of head array is illustrated in Figure 5. The darker part of the array shows the tuples of subsequences not pruned off during lower bounds pruning stage, which need to be calculated DTW distance. The head array records the starting position of all these subsequences that need DTW distance calculation. The head array is one of the input parameters of DTW kernel, where head array tells each work-item which subsequence it should process.

The second kernel calculates the DTW distance between query sequence and all the subsequences listed in the Head Array. The process will be illustrated in detail below. This kernel will output all the position of subsequence whose DTW distance to the query is under a specified threshold.

D. Segmented Parallel Dynamic Time Warping

The calculation of Dynamic Time Warping (DTW) distance is the most time-consuming process in software implementations. The DTW distance calculation process is a kind of dynamic programming, and the data in different stages of calculation have strong dependency, so it is hard to calculate in parallel. The performance of the DTW module is critical for the overall performance of the whole system.

We separate the process of DTW calculation of subsequences into two stages: distance matrix formation and warping path searching. The separating strategy can extract the potential parallel operations. This strategy dealing with warping matrix is also used in [2], and it is proved to be efficient. However, the warping matrix in [2] is formed by a single subsequence and the query sequence, so the effect of acceleration is not significant enough.

As is mentioned in Section III, after the process of lower bound module, the distribution of the subsequences that need DTW distance calculation has a significant characteristic: they form small segments in the whole time series. Based on this observation, we try to simplify the process of normalization, that is to normalize the subsequences by segments. After normalization, we calculate the distance matrix using work-items in parallel, and each work-item calculates the distance between a pair of tuples.

Based on the generated distance matrix, we use different computing units to search for the warping path of different subsequence at the same time, i.e. the searching on a distance matrix of a subsequence is completed by a single computing unit. So it is a serial process inside a single distance matrix.

V. PERFORMANCE EVALUATION

In this section, we will present our experimental setup and results, then evaluate the performance of our system design.

A. Experimental Setup

Our subsequence similarity search system is implemented on the heterogeneous computing platform, which is a workstation computer consists of AMD GPU and APU. The GPU on the platform are AMD Radeon HD 7970 and the processor is 3.8GHz AMD APU A10-5800K with Radeon HD Graphics. The operating system is Windows 7. And the development and compilation environment is Visual Studio 2010 and AMD Accelerated Parallel Processing (APP) SDK, which supports OpenCL.

In our experiments, the data source is the raw time series stored in the memory in advance. In applications, the data source could be in the form of data record coming from hard disks or time series stream flowing in via hardware interface, or other possible forms. To eliminate the influence of inconsistency of data form between different applications, in our experiments we assume that data has been stored in the memory, which can best simulate the data source in most applications in reality.

B. Experiment Results

1) Comparison with Software

There are various software acceleration techniques towards subsequence similarity search algorithms, and the software implementations differ in many aspects. The software implementation developed by T. Rakthanmanon et al. adopts almost all of the most effective software acceleration techniques and it achieves a high performance [15]. This implementation can be regarded as the best software implementation so far, which we compare our implementation with. In our experiments, we use the source code they provide to test the performance of their implementation.

In our experiments, we use strictly the same problem definition for both software implementation and our implementation. In detail, it is to search the inputted time series and find out all the subsequence whose DTW distance to the pattern is less than the given threshold. It should be noted that the source code of [15] on their website is a version searching for a best-match subsequence and the DTW distance threshold in their implementation keeps updating, so we do some little modifications to their source code so that the threshold keeps constant.

We use several different datasets to test the performance of our system, including random walk dataset, speech dataset and so on. As is mentioned in section III, the tightness of Sakoe-Chiba band constraint (R) differs in specific application, and R will have a wide range (e.g. from 0.05 to 0.9). We consider the variety of R and test the system performance under different value of R. Besides, the length of pattern is also an important variable in applications, so we also consider the different length of pattern in our experiments and evaluation.

a) Random walk dataset

Random walk is a kind of useful data model, and the time series in many applications can be modeled using random walk. We use the random walk data that used in [15] to evaluate the performance of best software implementation and our system. We try different R and different length of pattern, and observe the difference in performance when each of these parameters varies. Because of the limitation of GPU device memory capacity on our experiment platform, our implementation on current platform will be difficult to deal with patterns that are too long. However, this problem can be easily overcome when using GPU devices of larger memory capacity or using clusters of heterogeneous computing devices.

Figure 6 shows the result of the experiments in which the length of pattern is 128.

In Figure 6, two curves present the performance of the best software implementation UCR_DTW and the performance of our work. As is indicated by the experiment results, in the case where the length of pattern is 128, our work outperforms the best software implementation over all R values. On average, our work achieves a speedup of 5.5 times in this test case. And the maximum speedup of 6.67 times is achieved when $R = 0.1$. The experiment results also indicate that the performance of our work is not so sensitive to the change of R, when compared with software implementation.

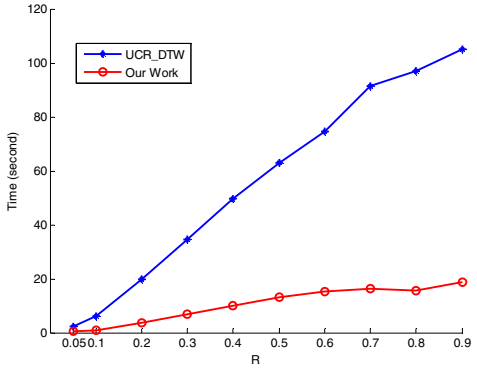

Figure 6. Performance Comparison on Random Walk Dataset

We use AMD APP Profiler to analyze the time consuming of every part of our implementation. The following table illustrates a typical profiling result when pattern length is 128 and R = 0.05.

TABLE I. A TYPICAL PROFILING RESULT

Function Name	% of Time
clBuildProgram	48.81%
clWaitForEvents	17.58%
clEnqueueWriteBuffer	10.31%
DTW	9.49%
clEnqueueReadBuffer	5.49%
clCreateCommandQueue	4.15%
LowerBound	1.74%
Distance	1.15%
clEnqueueNDRangeKernel	1.05%
clReleaseMemObject	0.17%

Table I presents the time used by every part of the whole program. The part with grey background is the time consumed by kernels in our program, while the rest is that of OpenCL API functions. As we can find in the table, for the cases where pattern length and R are not very large, the time used by the GPU device is only a small part of the total time used, and the host-end file reading or data transmission use relatively longer time.

b) ECG dataset

Electrocardiogram (ECG) dataset is a kind of typical medical monitoring dataset. The size of monitoring datasets is usually large, because of the continuous working requirement of common monitoring tasks. So there are great needs to reduce the time of discovering a specific pattern from the given monitoring dataset. We use the ECG dataset in [15] to test the performance of our work. This dataset consists of around 8.5×10^9 tuples and the length of pattern is 360. In the implementation, $R = 0.05$. Our work achieves 1.87 times speedup compared to their work when using this dataset.

TABLE II. PERFORMANCE COMPARISON USING ECG DATASET

UCR_DTW[15]	**Our work**	**Speedup**
18.0 minutes	9.61 minutes	1.87

2) Comparison with Hardware

During the last decade, different kinds of hardware acceleration techniques are proposed. D. Sart et al. proposed GPU implementation and FPGA implementation respectively in [11]. And they can be regarded as typical representations of hardware implementation. Besides, one of our recent works [10] realizes significant speedup of subsequence similarity search using FPGA, which can be regarded as the best hardware implementation so far.

We use the dataset in [11] to evaluate the performance of these hardware implementations. This is an Electrical Penetration Graph (EPG) dataset. It contains 1,499,000 tuples and the length of pattern is 360. As is stated in [11], their GPU implementation needs 80.39s and FPGA implementation needs 2.24s, while our system can complete the search task in only 1.205s, achieving 66.70 times speedup compared to their GPU implementation and 1.86 times to their FPGA implementation. In this test case, our work even outperforms their FPGA implementation. There are two main reasons. On the one hand, their FPGA implementation is created using C-to-VHDL tool, and the lack of utilization of data reuse and combination of lower-bound techniques reduce the performance of their FPGA implementation.

TABLE III. PERFORMANCE COMPARISON USING EPG DATASET

	D. Sart[11]	**Our Work**	**Speedup**
GPU	80.39s	1.205s	66.70
FPGA	2.24s		1.86

Our recent work of accelerating subsequence similarity search [10] achieves a significant speedup using FPGA. This FPGA implementation needs only 0.011s to search in the above dataset, which is around 100 times faster than our GPU implementation. However, programming FPGA requires professional knowledge about circuits and the modification and reconfiguration on FPGA are relatively difficult, while GPU can be easily programmed and the design can be quickly updated without much cost. Besides, GPU is more common in the computing platform and is more widely supported by software. Considering these aspects, GPU and FPGA can be regarded as two kinds of computing devices whose application areas are different, and the performance of GPU and FPGA should be evaluated separately.

VI. CONCLUSION AND FUTURE WORK

We have presented a DTW-based subsequence similarity search system on the heterogeneous computing platform. We have carefully adapted the state-of-art DTW-based similarity search software algorithm so that it well fits the characteristics of each kind of devices on the heterogeneous computing platform, and the hardware implementation achieves a higher performance than other GPU implementations and even some of FPGA implementations. As is indicated by experiment results, our work achieves one to two orders of magnitude speedup compared to software implementation and several times speedup to other GPU implementations and even some FPGA implementations. Our similarity search system uses lower bound technique to prune off most of the candidate subsequences, and based on the pruned-off results, we proposed a novel merge technique so that computation-reuse features between neighbor subsequences are fully utilized. Besides, due to the potential properties of heterogeneous computing platform, our system is scalable and can be easily adjusted to deal with different data sizes.

Note that the processor on our heterogeneous computing platform is an AMD APU, which consists of a CPU and a GPU. However, our current implementation utilizes only the CPU on the processor and the independent GPU board. And we are considering about proposing a proper design that fully utilize the GPU on the processor. For example, move the task of lower bound pruning to the GPU on the APU processor. In this way, the lower bound pruning process and the DTW calculation process will go on at the same time, since the GPU on processor and the independent GPU can run at the same time.

OpenCL implementation guarantees the concurrent control over all of the heterogeneous computing devices, so we can effectively utilize every computing device on the platform at the same time, including CPUs, GPUs, or even FPGAs or other possible devices in the future. Based on this knowledge, we are also considering about designing a similarity search system that can sense all the available computing devices on the platform and automatically balance the load of every computing devices. In this way, the computing resources are fully utilized and the highest performance is achieved.

ACKNOWLEDGMENT

We would like to thank Dr. Lanjun Wang, from IBM Research China, for her helpful suggestions on our research and paper writing. This work was supported by IBM, 973 project 2013CB329000, National Science and Technology Major Project (2013ZX03003013-003), National Natural Science Foundation of China (No. 61373026, 61028006), and Tsinghua University Initiative Scientific Research Program.

REFERENCES

[1] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., and Keogh, E. J. 2008. Querying and mining of time series data: experimental comparison of representations and distance measurements. *PVLDB* 1, 2, 1542-52.

[2] Zhang, Y., Adl, K., and Glass, J. 2012. Fast spoken query detection using lower-bound Dynamic Time Warping on Graphical Processing Units. *ICASSP*, 5173 – 5176.

[3] Sakurai, Y., Faloutsos, C., and Yamamuro, M. 2007. Stream Monitoring under the Time Warping Distance. *ICDE*, 1046-1055.

[4] Lim, S. H., Park, H., and Kim, S. W. 2007. Using multiple indexes for efficient subsequence matching in time-series databases. *Inf. Sci.* 177, 24, 5691-5706.

[5] Keogh, E. J., Wei, L., Xi, X., Vlachos, M., Lee, S. H., and Protopapas, P. 2009. Supporting exact indexing of arbitrarily rotated shapes and periodic time series under Euclidean and warping distance measurements. *VLDB J.* 18, 3, 611-630.

[6] Fu, A., Keogh, E. J., Lau, L., Ratanamahatana, C., and Wong, R. 2008. Scaling and time warping in time series querying. *VLDB J.* 17, 4, 899-921.

[7] Srikanthan, S., Kumar, A., and Gupta, R. 2011. Implementing the dynamic time warping algorithm in multithreaded environments for real time and unsupervised pattern discovery. *IEEE ICCCT*, 394-398.

[8] Takhashi, N., Yoshihisa, T., Sakurai, Y., and Kanazawa, M. 2009. A Parallelized Data Stream Processing System using Dynamic Time Warping Distance. *CISIS* 1100-1105.

[9] Grimaldi, M., Albanese, D., Jurman, G., and Furlanello, C. 2009. Mining Very Large Databases of Time-Series: Speeding up Dynamic Time Warping using GPGPU. *NIPS Workshop*.

[10] Wang, Z., Huang, S., Wang, L., Li, H., Wang, Y., and Yang, H. 2013. Accelerating subsequence similarity search based on dynamic time warping distance with FPGA. Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays. *ACM*, 2013: 53-62.

[11] Sart, D., Mueen, A., Najjar, W., Niennattrakul, V., and Keogh, E. J. 2010. Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs. *ICDM*, 1001-1006.

[12] Kim, S., Park, S., and Chu, W. 2001. An index-based approach for similarity search supporting time warping in large sequence databases. *ICDE*, 607–61.

[13] AMD Accelerated Parallel Processing OpenCL Programming Guide. Advanced Micro Devices, Inc. Dec. 2012.

[14] The OpenCL Specification v1.2. Khronos OpenCL Working Group. Nov. 2012.

[15] Rakthanmanon, T., Campana, B. J. L., Mueen, A., Batista, G. E. A. P. A., Westover, M. B., Zhu, Q., Zakaria, J. and Keogh, E. J. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. *KDD*, 262-270.

APPENDIX: GPU ARCHITECTURE AND PROGRAMMING MODEL

The GPU on our experiment platform belongs to the AMD Radeon HD 7900 series. In this section, we briefly introduce the architecture of the AMD GPU that we use and the OpenCL programming model. Please refer to [13] for a detailed information.

A. AMD GPU Architecture

Figure 7 illustrates the generalized AMD GPU compute device structure. As is shown in Figure 7, the GPU compute device contains a number of Compute Units (CUs), and each CU contains one Scalar Unit and four Vector Units, each of which contains an array of 16 Processing Elements. The four Vector Units use SIMD execution of a scalar instruction [13]. For AMD Radeon HD 79XX series, each GPU compute device contains 32 CUs.

Figure 8 shows two compute units of AMD Radeon HD 79XX family. One compute unit contains Scalar Unit, Scalar Unit, Level 1 data cache (L1) and Local Data Share (LDS), and there are 32 compute units. The SC cache is the scalar unit data cache, and the Level 2 cache contains instructions and data.

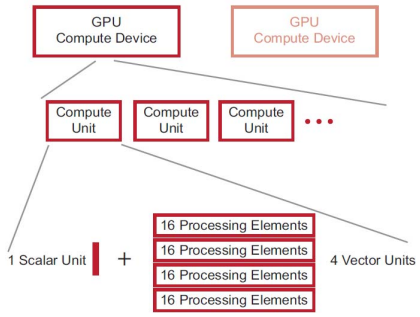

Figure 7. Generalized AMD GPU Compute Device Structure for Southern Islands Devices [13]

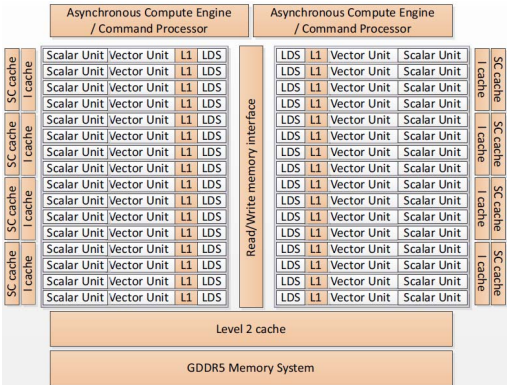

Figure 8. AMD Radeon HD 79XX Device Partial Block Diagram[13]

B. The OpenCL Programming Model

We use the APIs provided by OpenCL to program AMD GPU. OpenCL is an open standard for parallel programming of heterogeneous systems. Detailed information about OpenCL could be found in [14].

In OpenCL, the program executed on GPU is called a *kernel*. And the unit of parallel execution is called a *work-item*. Every work-item has its own ID and executes the same kernel. The work-items are automatically mapped onto the processing units on GPU by OpenCL, and this is the basic idea how parallelism is realized. The number of work-items can be specified by host-end program, and that which work-item should be active and execute kernels is determined and managed by OpenCL. So even though the number of work-item specified by the host-end program is larger than actual parallel processing units, the program can be executed normally. OpenCL supports two programming models, data parallel programming model and task parallel programming model.