# Accelerating Subsequence Similarity Search Based on Dynamic Time Warping Distance with FPGA

Zilong Wang[1], Sitao Huang[1], Lanjun Wang[2], Hao Li[2], Yu Wang[1], Huazhong Yang[1]

[1]E.E. Dept., TNLIST, Tsinghua University; [2]IBM Research China

{wang-zl11, hst10}@mails.tsinghua.edu.cn, {wangljbj, haolibj}@cn.ibm.com,

{yu-wang, yanghz}@mail.tsinghua.edu.cn

## ABSTRACT

Subsequence search, especially subsequence similarity search, is one of the most important subroutines in time series data mining algorithms, and there is increasing evidence that Dynamic Time Warping (DTW) is the best distance metric. However, in spite of the great effort in software speedup techniques, including early abandoning strategies, lower bound, indexing, computation-reuse, DTW still cost too much time for many applications, e.g. 80% of the total time. Since DTW is a 2-Dimension sequential dynamic search with quite high data dependency, it is hard to use parallel hardware to accelerate it. In this work, we propose a novel framework for FPGA based subsequence similarity search and a novel PE-ring structure for DTW calculation. This framework utilizes the data reusability of continuous DTW calculations to reduce the bandwidth and exploit the coarse-grain parallelism; meanwhile guarantees the accuracy with a two-phase precision reduction. The PE-ring supports on-line updating patterns of arbitrary lengths, and utilizes the hard-wired synchronization of FPGA to realize the fine-grained parallelism. It also achieves flexible parallelism degree to do performance-cost trade-off. The experimental results show that we can achieve several orders of magnitude speedup in accelerating subsequence similarity search compared with the best software and current GPU/FPGA implementations in different datasets.

## Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database application – *Data mining*

B.7.1 [**Integrated Circuits**]: Types and Design Styles – *Algorithms implemented in hardware, Gate arrays*

## General Terms

Algorithms, Design, Performance

## Keywords

DTW, FPGA, subsequence similarity search, time series

## 1. INTRODUCTION

There is an increasing interest in subsequence similarity search of time series in many domains, such as database and knowledge discovery. For example, in a bridge health monitoring database system, finding the bridge vibration which has some specific patterns may help maintenance personnel predict the bridge cracking problem. In subsequence similarity search, a key subroutine is to measure the distance between the subsequences and the pattern. In spite of dozens of similarity distance metrics which have been proposed in the last decades, there is increasing evidence that the classic **Dynamic Time Warping (DTW) is the best distance metric** [4].

Due to its unique ability to recognize a warping pattern, DTW has been widely used in many domains, like robotics, astronomy, speech recognition, gesture recognition, biometrics or medicine [1][2][3]. However, the application of DTW is greatly limited by its computational complexity. To solve this problem, dozens of techniques and thousands of papers have been proposed to speed up DTW. One of the most effective methods is lower bound, which tries to prune off most of the subsequences based on the lower bound of DTW distance estimated in a cheap way[4][5][8]. Another one is computation-reuse [12], which tries to reuse the temporary data between the neighboring DTW calculations to reduce calculation. The other techniques includes early abandoning strategies [10], data indexing [7][9], embedding. All these software based techniques focus on reducing the calling times of DTW, instead of accelerating DTW itself, while the DTW calculation still takes too much time, **accounting for about 80% of the total time** [17]. Therefore, even with so many techniques and several orders of magnitude speedup, the whole subsequence similarity search is still the bottleneck of many applications [2] [3].

A recent work [10] combines all possible existing speedup techniques and achieved a very fast speed, which makes us believe that we have almost exploited all potential from software and we must turn to hardware if we want to get a further speedup. However, since DTW is a 2-dimension dynamic search, the high data dependency makes it not easy to be accelerated by parallel hardware. This may explain that in spite of the continuous interest in software speedup techniques, there are very few efforts exploiting the effectiveness of parallel hardware. Most of the existing hardware efforts are implemented in coarse-grained parallelism, i.e. they try to allocate DTW calculation of different subsequences into different processing elements, such as computer-cluster [14], multi-cores [13], or GPU [17]. Limited by the huge transfer load of subsequences, these methods cannot get further speedup, and they still do not accelerate DTW itself. The first and only implementation using FPGA [12] to accelerate DTW itself is generated by a C-To-VHDL tool. The lack of insight into the FPGA limits their achievement, and their work can only be applied in small-scale problems. Actually, there is still fine-grained parallelism potential in DTW, and we find that this parallelism can be only exploited by FPGA, instead of GPU or multi-cores. However, Unlike GPU or multi-cores, FPGA is considered to be not suitable for if-not-break algorithms because of the fixed pipeline structure, which means most of the software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*FPGA'13*, February 11–13, 2013, Monterey, California, USA.

Copyright ©2013 ACM 978-1-4503-1887-7/13/02 …$15.00.

speedup techniques may not be able to be utilized by FPGA. To address these problems and to explore how FPGA can go further than GPU or other parallel hardware in subsequence similarity search, we propose the subsequence similarity search based on DTW distance on FPGAs. The major contributions of our work are as follows:

1) We propose a stream oriented framework for subsequence similarity search. Based on the assumption that we can do normalization on subsequence of a little longer length, we realize coarse-grained parallelism by reusing data of different DTW calculations. This framework uses a two-phase precision reduction technique to guarantee accuracy while reducing resource cost.

2) We propose a PE-ring structure for DTW calculation. Benefited from the hard-wired synchronization only provided by FPGA, the PE-ring can exploit all the fine-grained parallelism of DTW. The flexible ring structure allows on-line updating patterns of arbitrary length without re-compilation or re-configuration. Besides, flexible parallelism degree is achieved to do performance-cost trade-off.

3) Our work achieves significant speedup. The experimental results show that we achieve several orders of magnitude speedup in accelerating subsequence similarity search, compared with the best software and current GPU/FPGA implementations in different datasets.

# 2. BACKGROUND AND RELATED WORK

## 2.1 Subsequence Similarity Search

Subsequence similarity search aims to find a subsequence which is similar to a certain pattern from a time series, using the following definition:

A *time series* $S$ is a sequence of tuples: $S = s_1, s_2 \ldots, s_N$, where $N$ is the length of the whole sequence.

A *tuple* $s_i$ consists of sample time $t_i$ and sample value $v_i$: $s_i = [t_i, v_i]$;

A *subsequence* $S_{ik}$, which starts from $i$ and ends at $k$, is a continuous subset of $S$: $S_{ik} = s_i, s_{i+1} \ldots, s_k$ where $1 \le i < k \le N$;

A *pattern* $P$ is a certain subsequence to be found: $P = p_1, p_2 \ldots p_M$, where $M$ is the length of the pattern.

Note that for a sequence of length $N$, there will be $N$-$M$ subsequences of length $M$, which are usually picked out by a sliding window. A distance metric is used to define how similar the two subsequences are. The DTW distance's definition can be found in Section 2.2. There are two kinds of subsequence similarity search:

***Best-match search***: Given sequence $S$ of length $N$ and pattern $P$ of length $M$, find the subsequence $S_{ik}$ whose distance from $P$ is the minimum among all the subsequences, that is to say: $Distance(S_{ik}, P) < Distance(S_{ab}, P)$ for any $1 \le a \le n$ and $b=1 \le b \le n$.

***Local-best-match search***: Given sequence $S$ of length $N$ and pattern $P$ of length $M$, find all the subsequences $S_{ik}$ whose distance from P is the minimal among their neighboring subsequences whose distance from the pattern $P$ is under the threshold.

We focus on the second one in our paper.

## 2.2 Dynamic Time Warping

DTW is a measure of the distance between two subsequences. Suppose we have two time series: $P = p_1, p_2, p_3 \ldots p_M$, and $S = s_1, s_2, s_3 \ldots s_M$, where $P$ is the pattern and $S$ is the subsequence picked out by the sliding window. The DTW distance between $P$ and $S$ is defined as follows:

$$DTW(S, P) = D(M, M);$$

$$D(i, j) = dist(s_i, p_j) + min \begin{cases} D(i-1, j) \\ D(i, j-1) \\ D(i-1, j-1) \end{cases} \quad (1)$$

$$D(0,0) = 0;\ D(i, 0) = D(0, j) = \infty,\ 1 \le i \le M, 1 \le j \le M;$$

where $M$ is the length of pattern, *dist()* is the distance between two tuples. Either absolute difference or squared difference is fine.

The calculation process of DTW can be illustrated in the following intuitive way: to align two sequences of length $M$ using DTW, an $M$-by-$M$ warping matrix is generated, with the $(i, j)$ cell being the distance between $s_i$ and $p_j$, i.e., $dist(s_i, p_j)$. The DTW algorithm aims to search for an optimal path from the left-bottom cell to the right-top cell, as shown in Figure 1. The accumulated value of the cells in this path should be the minimum of all paths. The accumulated value is defined as the DTW distance, and the path implies the mapping relationship between the tuples of these two sequences. In addition, researchers often constrain the path that it cannot deviate more than $R*M$ cells from the anti-diagonal, as the green line in Figure 1, which is called the Sakoe-Chiba band constraint [4]. $R$ can be seen as the allowed warping degree of the pattern, ranging from 0 to 1.

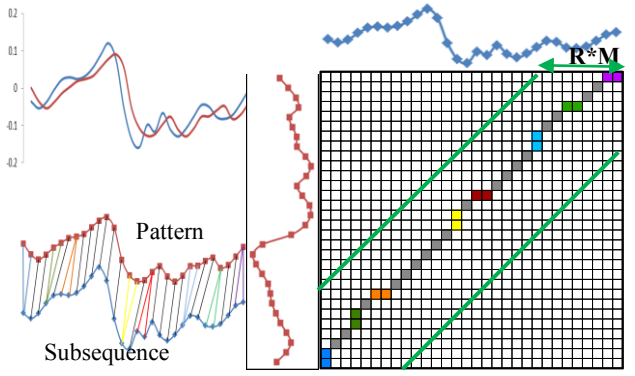



**Figure 1: *(left-top)* two similar time series; *(right)* search for the optimal path in the warping matrix; *(left-bottom)* the final mapping between two time series.**

Obviously, DTW is actually a 2-Diminson dynamic path search, whose time complexity is $O(M*M)$. For a sequence of length $N$, there will be $N$-$M$ subsequences of length $M$, so the time complexity is up to $O(N*M*M)$ for the whole sequence search. If Sakoe-Chiba path constraint is used, it is reduced to $O(N*M*M*R)$, which is still too large. However, DTW is considered as the best distance metric [4], since it allows the subsequence to have some stretching, shrinking, warping, or even length difference compared with the pattern. Many research efforts would like to use DTW, but find it very time-consuming [5,14,15,16]. Therefore, DTW is the right measure to accelerate.

## 2.3 Related Work

Accelerating DTW attracts thousands of research work in these decades, and almost all of them are software-based techniques.

Y. Sakurai et al. propose a computation-reuse algorithm called SPRING [11]. However, the data-reuse requires the sequence not

to be normalized, which limits its applications. S. H. Lim et al. use indexing techniques to speed up the search, which need to specify the length of pattern beforehand [9]. E. Keogh et al. build a multiple index for various length patterns [7], which may be quite complex if the patterns have a wide range of length. Besides, no one can index on an infinitely-long streaming data. A. Fu, E. Keogh et al. try to estimate the lower bound of DTW distance in a cheap way, called LB_Keogh. If the lower bound has exceeded the threshold, the DTW distance also exceeds the threshold, so the subsequence can be pruned off. Although the lower bound technique is widely used and effective, the conflict of tightness and complexity limits the further speedup. The same research group claims to be able to search data even larger than all the other work in a recent paper [10], which can be seen as one of the best software implementations. It utilizes almost all the software techniques to do speedup.

All the former software techniques can be seen as pre-processing techniques, as they mainly focus on how to reduce the calling times of DTW calculation, instead of accelerating DTW itself. However, because it is too complex, DTW still costs a large part of the total time in many applications, which limits the further speedup of software techniques.

S. Srikanthan et al. exploit the possibility of CPU clusters to speed up DTW. They send subsequences starting from different positions of the time series to different processors, and each processor executes DTW calculation in the naive way [13]. N. Takhashi et al. choose multi-cores [14]. They separate different patterns into different cores, and each subsequence will be sent to different cores to be compared with different patterns in the naive way. As a subsequence consists of several hundreds or thousands of tuples, the data transfer becomes the bottleneck of the whole system in both implementations. M. Grimaldi et al. propose a GPU implementation [6] similar to [13].

All the former works using parallel hardware are implemented in coarse-grained parallelism, as they focus on how to separate different DTW calculations into different processing units. However, the cost of coarse-grained parallelism is very high. A sequence of length $N$ includes ($N$-$M$) subsequences of length $M$, If we would like to achieve a parallelism up to $W$, the data transfer bandwidth will be $W*M$ tuples/cycle, which will soon become the bottleneck as the $W$ or $M$ increases, so it is hard to get speedup from coarse-grained parallelism.

Y. Zhang et al. propose another GPU implementation [17]. Multi threads are used to generate the warping matrix in parallel, but they used another thread to do path search serially. This implementation exploits partial fine-grained parallelism of DTW itself. Because it separates the matrix generation phase and the path search phase into two kernels, it needs to store and transfer the whole warping matrix for each DTW calculation, which is also a heavy burden of the whole system. D. Sart et al. claim themselves to be "the first to present hardware acceleration techniques for similarity search in streams under the DTW measure" [12], and to the best of our knowledge, this is also the only work using FPGA to accelerate DTW. Their GPU implementation is similar to [13]. Their FPGA implementation consists of two modules: *Normalizer* and *Warper* (DTW calculation). Though their system is generated by a C-to-VHDL tool called ROCCC, based on their performance, we think the tool has smartly exploited the fine-grained parallelism of the DTW. However, the lack of insight into the FPGA makes their work suffers from lacking flexibility and scalability: it must be re-compiled if updated to a new pattern of different length, the data precision is only 8 bits, and it cannot support patterns longer than 1024. Their work focuses on the naive subsequence similarity search, and do not exploit any pre-processing techniques, which also limits them to get further speedup.

# 3. ALGORITHM

Both software solution and our hardware solution share the similar algorithm framework, consisting of three modules: normalization, lower bound, and DTW calculation, as shown in Figure 2. In normalization step, we make two assumptions to enable the data-reusability for DTW calculation and guarantee the accuracy while using lower precision representation in FPGAs. We try to construct a hybrid lower bound structure based on the existing lower bounds techniques to do performance-resource trade-off in different applications. After lower bound's pruning, the rest subsequences are usually located in continuous interval, so we modify the SPRING algorithm to calculate the DTW distances of multiple continuous subsequences in single search.

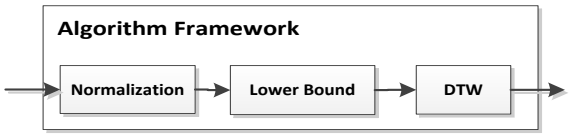



**Figure 2: The algorithm framework**

## 3.1 Normalization

In some applications, the subsequences must be normalized to get a meaningful result, which is emphasized in some existing works [10][12]. This is mainly caused by the time-varying offset (or amplitude) of the sequence, which has a quite bad impact on the accurate distance calculation.

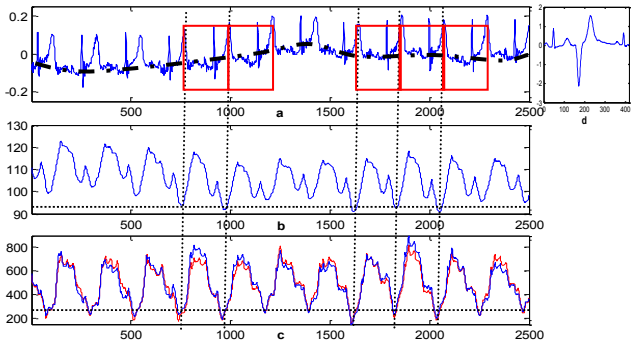



**Figure 3: (a) the blue line is the electrocardiogram (ECG), and the dotted line is the approximate offset; (b) the DTW distance without normalization, only five heartbeats can be recognized; (c) the DTW distance with normalization on subsequence of length $M$ (red) and $M$+$M$(blue), all the heartbeats have been recognized in both cases; (d)the pattern we want to find.**

As shown in Figure 3 (a), the offset of the heartbeat wave starts below zero, then goes up and finally descends. If we directly do DTW calculation without normalization, we can get a result like Figure 3 (b). Some correct subsequences are missed or some false subsequences are recognized, no matter how we set the threshold. If we do normalization just once on the whole sequence, we will find that the sequence has not changed, because the mean of the whole sequence is still about zero. The right way, just like what they do in [10][12], is to do normalization in every subsequence, for example, $S'_{i,i+M} = normalize(S_{i,i+M})$ and $S'_{i+1,i+M+1} = normalize(S_{i+1,i+M+1})$. In fact, it implies an assumption that the offset (or the amplitude) can be approximately seen as time-

invariant in a short length of $M$, where $M$ is the length of pattern, so bad impact of the offset can be removed by normalization. If the offset can't be seen as time-invariant, we cannot get meaningful result even with normalization. This motivates us that it is the time-varying offset leading to the accuracy degradation, instead of the nonstandard offset itself. In fact, if several subsequences share the same nonstandard offset, the most similar one to the pattern will still be the most similar one to the pattern after normalization. Since our objective is to find the subsequence which is most similar to the pattern in a local interval, we can normalize these several subsequences once if they share the same offset. Although these subsequences' offset may still be nonstandard after such normalization, it makes no change to us to find the most similar one to the pattern. Here we make a little stronger assumption, which will be used in Section 3.3.

**Assumption 1**: the offset or the amplitude can be approximately seen as time-invariant in a little longer length of $M+C$, where $M$ is the length of pattern, and $C$ is a constant no larger than $M$.

With this assumption, we can pick out $M+C$ tuples once and do normalization on these tuples. Note that $M+C$ tuples include $C+1$ subsequences of length $M$, so we actually normalize $C+1$ subsequences once. The experimental results show that this assumption is effective in most real world time series. As shown in Figure 3 (c), the red curve is the DTW distance after normalization in every subsequence, which means normalized by every $M$ tuples, while the blue curve is the DTW distance after normalization in every $M+M$ tuples, where we set $C$ to be $M$. There is little difference between each other, but the correct subsequences similar to the pattern have been recognized in both cases. Anyway, the purpose is to find the similar subsequence in a local interval under the threshold, not to calculate the exact distance between the subsequence and pattern.

To reduce computation and process the streaming data, we do subsequence normalization based on the following equations:

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

$$\sigma^2 = \frac{1}{m}\left(\sum_{i=1}^{k} s_i^2 - \sum_{i=1}^{k-m} s_i^2\right) - \mu^2 \tag{3}$$

$$s_i = \frac{s_i - \mu}{\sigma} \tag{4}$$

With formula (2) and (3), the mean and standard deviation of the subsequence in the sliding window can be updated only with the new-come tuple and the last tuple in the window.

Because floating point operation or wide fixed point operation cost much resource in FPGAs and most applications don't need such high precision, it's necessary to explore how many bits is enough to represent a number without sacrificing accuracy. Here we make another assumption.

**Assumption 2:** the subsequence can be represented with a shorten width fixed point number after normalization, that is to say, low precision number.

Note the fact that the dynamic range of the data is usually very small after offset removed and amplitude compressed, this assumption is rather trivial in most applications. We select "short" type (16 bit fixed point number) as the default data format, but it can be replaced by any wider fixed point number if needed. The experiment also shows that this assumption only leads to a little difference in distance, without sacrificing the matching accuracy.

Unlike lower bound technique or DTW calculation of the mathematical completeness, normalization is usually considered as an empirical pre-processing technique. There is no theoretical evidence that which normalization and how to do normalization is the best choice. For example, z-normalization as formula (4) is used based on the assumption that the sequence follows the normal distribution, while we can also simply set the maximum to be "1" as another kind of normalization. Whether we do normalization on the whole sequence, or on every subsequence, or on several continuous subsequences as assumption 1 is dependent on whether the accuracy can be improved. The time-varying offset (or amplitude) can also be removed by many other techniques, such as Kalman-filter, which is widely used in sensors for motion detection. However, we do normalization not only for improving accuracy, but also for the data representation in hardware, as shown in section 4.1.2.

## 3.2 Lower Bound

A classic way to speed up subsequence similarity search based on an expensive distance metric such as DTW is to use a cheap-to-compute distance to estimate the lower bound of the real distance[4][7]. If the lower bound has exceeded the threshold, the DTW distance will exceed the threshold, too. There are many lower bound techniques, and their tightness and complexity are all different. T. Rakthanmanon et al. propose a cascade lower bound, [10] consisting of LB_Kim [8], LB_Keogh [4][9] and reversed-LB_Keogh. The complexity of these three lower bounds is increasing as well as the tightness. If a former lower bound's result is not large enough to prune off a subsequence, the latter lower bound will be executed. As such if-not-break algorithm is not suitable for FPGA, we propose a hybrid structure consisting of LB_partial DTW (LB_pDTW), LB_Keogh and reversed-LB_Keogh. Actually, this can been seen as a combination of lower bound technique and early abandoning technique.

LB_pDTW is defined as the DTW distance between the partial subsequence and the partial pattern, for example, the DTW distance between the first X tuples of the subsequence and the first X tuples of the pattern. It can be seen as a strengthened version of LB_Kim. LB_Keogh constructs an upper envelope and a lower envelope, and accumulate the distance falling out of the envelope as the lower bound. Reversed-LB_Keogh uses the same algorithm as LB_Keogh, only reversing the position of the pattern $P$ and the subsequence $S$. The definition is as follows:

$$U_i = max\ \{p_{i-R}, p_{i-R+1} \ldots p_{i+R-1}, p_{i+R}\};$$

$$L_i = min\ \{p_{i-R}, p_{i-R+1} \ldots p_{i+R-1}, p_{i+R}\}; \tag{5}$$

$$D_i = \begin{cases} s_i - U_i & if\ s_i > U_i \\ L_i - s_i & if\ L_i > s_i \\ 0 & else \end{cases} \tag{6}$$

$$LB(P_{1,Y}, S_{1,Y}) = \sum D_i \tag{7}$$

where $1 \le i \le Y$, and $R$ is the parameter indicating the allowed warping degree, defined as the green line in Figure 1.

In our hybrid lower bound, LB_pDTW processes the first $X$ tuples of every subsequence, while LB_Keogh and reversed-LB_Keogh process the following $Y$ tuples. Note that $Y$ is usually set to be $M-X$, where $M$ is the length of the pattern. The final lower bound is defined as ***LB_pDTW + max{LB_Keogh, reversed-LB_Keogh}***. LB_pDTW is a rather tight lower bound, but its time complexity is up to O($X$*$X$). LB_Keogh has the highest performance-cost-ratio, which costs only about O($Y$) time complexity to prune a lot of subsequences. The effectiveness of LB_Keogh highly depends

on the parameter *R*, which is usually set to a small value, such as 5%~10%. If we set *R* to a little larger value, for example, 20%, the performance will significantly degrade, and if *R* is set to 100%, which is used in the classic DTW, this method will be absolutely useless. So a hybrid structure of LB_pDTW and LB_Keogh/ reversed LB_Kegoh can do performance cost trade-off with the ratio of X/Y.

With lower bound technique, we can prune off many subsequences whose hybrid lower bound distance has exceeded the threshold. As shown in Figure 3, if we set the threshold to be 40, only a small portion of the subsequences still need the DTW calculation, as the red star line shown in Figure 4.

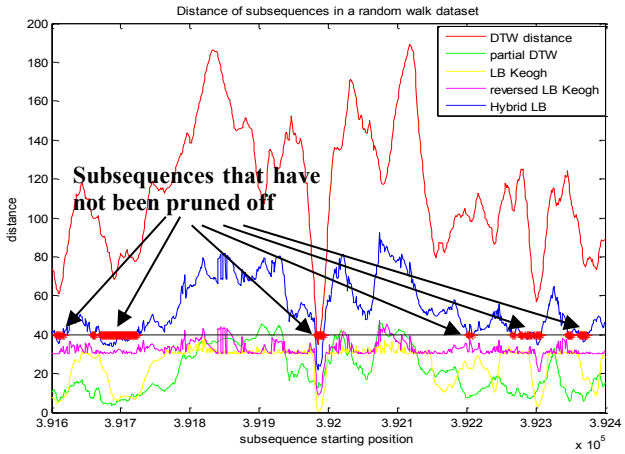



**Figure 4: DTW distance and different lower bound distance in a random walk dataset, where *M*= 128, *X*=10, *Y*=118, R=5%.**

## 3.3 DTW

Since candidate subsequences, which still need DTW calculation, is usually continuous, we can use a computation-reuse DTW algorithm, like SPRING [11], to calculate the DTW distance of multiple subsequences in a single search. In classic DTW algorithm, one *M*-by-*M* warping matrix is generated for every subsequence. Note that there are *N*-*M* subsequences of length *M* in the whole sequence of length *N*, so totally *N*-*M* matrixes are generated. In the neighboring two subsequences, for example, $S_{i,k}$ and $S_{i+1,k+1}$, only one tuple is different, which means that only one column is different in the two warping matrixes. Based on this data reusability, the SPRING algorithm modifies the boundary condition of the classic DTW, generates a single *M*-by-*N* matrix and finds the matching subsequences only in one search. N paths grow from all cells in the bottom line of the M-by-N matrix at the same time, instead of only from the left-bottom cell, with extra labels recording the starting positions of the paths. The less similar paths will be replaced by the more similar paths when the paths grow. The SPRING algorithm is defined as follows:

$$DTW(S_{s,e}, P) = D(e,M)$$

$$D(i,j) = dist(s_i, p_j) + \min \begin{cases} D(i-1,j) \\ D(i-1,j-1) \\ D(i,j-1) \end{cases} \quad (8)$$

$$Sp(i,j) = \begin{cases} Sp(i-1,j) & \text{if } D(i-1,j) \text{ is the minimum} \\ Sp(i-1,j-1) & \text{if } D(i-1,j-1) \text{ is the minimum} \\ Sp(i,j-1) & \text{if } D(i,j-1) \text{ is the minimum} \end{cases} \quad (9)$$

$$D(i,0) = 0;\ D(0,j) = \infty;$$

$$Sp(i,0) = i;\ Sp(0,j) = 0;\ \text{where } 1 \le i \le N, 1 \le j \le M;$$

Obviously, SPRING reduces the time complexity from O(*(M-N)*M*M*) to O(*M*N*), which is a quite high speedup. However, some researchers reject this algorithm because if we do normalization on every subsequence, all the tuples are different even in the neighboring subsequences, thus the data reusability does not exist. However, as the assumption 1 in section 3.1, we can do normalization on *C+1*continuous subsequences, because the offset can be seen as time-invariant in a small interval. Then we can apply SPRING to *C+1*subsequences, e.g. the red-star-line shown in Figure 4, to achieve a coarse-grained parallelism up to *C+1,* as they are normalized with the same coefficients.

We slightly modify the SPRING algorithm to allow constrains R: In formula (8), the former *D(u,v)* can be selected only if $u - R \le SP(u,v) + v \le u + R$, which means the current cell is in the constraint line. In boundary condition, *D(i,0)* can be set to *0* only if the current subsequence is *valid*, which means it has not been pruned off. Or *D(i,0)* is set to infinite, too.

# 4. IMPLEMENTATION

## 4.1 System Architecture

The following figure is our hardware framework:

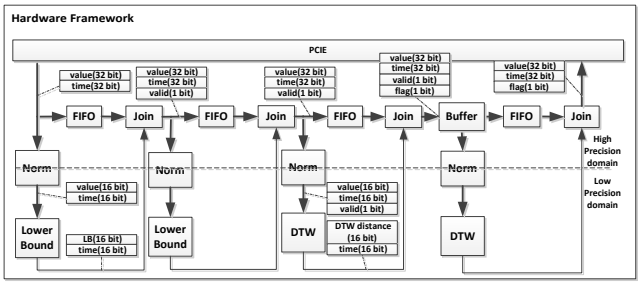



**Figure 5: Hardware framework of the whole system**

Compared with the algorithm framework in Figure 2, our hardware framework has two main differences: one is that we separate DTW and lower bound into two individual loops without direct connection; another one is that we place duplicate loops for both lower bound and DTW. In our system, *C* subsequences are seen as a group, where *C* is the constant in Assumption 1, and we set *C* to be *M*, where *M* is the length of the pattern.

Our system works as follows:

Firstly, the CPU sends the basic configuration information to the FPGA, for example, the normalized pattern, the threshold.

Secondly, the streaming time series will be transferred through PCIE to FPGAs. Each of the normalizer has a sliding window of length *M+M*, which is the total length of *M+1*continuous subsequences. The *normalizer* in the first loop picks out the first *M* continuous subsequences and normalizes them, while the second loop picks out the next *M* continuous subsequences. Then the two windows both slide in a step of *2*M* tuples to prepare for the next calculation, just like a Ping-Pong buffer. The normalized subsequences will be represented by a shorter width number, before they are sent to *Hybrid Lower Bound* modules. Then the lower bound result will be joined with the original sequence, based on the sample time tag. In the *Join* module, if the lower bound distance has exceeded the threshold, the starting tuple of this subsequence will be asserted *invalid* with a label, which means it has been pruned off.

Thirdly, if a tuple with *valid* tag high is founded in the first *M* tuples of the sliding window of the third loop, which means this subsequence still needs DTW calculation, the normalizer will pick out all the *2*M* continuous tuples in the window and do normalization. Then the *DTW* module will calculate the DTW distance for the subsequences which are asserted *valid*. All the first *M* tuples of the original sequence will be asserted "invalid" after they flow into the *DTW* module, indicating they do not need DTW calculation again. The DTW result will be joined with the original sequence, too. If the DTW distance is smaller than the threshold, the *Join* module will assert another tag *flag* high, indicating that this subsequence is what we want. If the DTW module is busy, the normalizer stops normalization until the DTW module is idle again, but still slides and updates the mean& standard, which means the original sequence flows without stop.

Fourthly, the whole sequence will be stored in a buffer and very few tuples are still asserted *valid*. The fourth loop works just as the third loop, but the first *M* tuples in the sliding window will be output to the last FIFO only if they are all asserted "invalid".

Finally, the sequence will be sent back to CPU tagged with label *flag* indicating which subsequences are the similar ones.

### *4.1.1 Four loops to guarantee streaming*

The main reason for the duplicate loops of lower bound and DTW is that we try to guarantee the streaming processing. As we describe before, the first loop picks out *M+M* tuples and executes normalization and lower bound calculation on them. Only *M* subsequences starting from the first *M* tuples have been processed. If we place a duplicate loop, the second *M* subsequences starting from the second *M* tuples will be processed by the second loop. In this way, all the tuples can just flow through the both loops for once without stop. The third loop also tries to guarantee the streaming processing of the original sequence. However, as the DTW module is not implemented in full-pipeline and needs many cycles to finish the processing, so the last loop must work in a stop-wait way. An extra buffer is used, and the stream may stop before the fourth loop. Since most subsequences have been pruned off by the lower bound, the burden of these two loops is not heavy and usually can process the sequence in streaming. This can be seen as another kind of coarse-grained parallelism.

The main reason for separating the DTW and lower bound into two loops, instead of directly connecting each other in one loop, is that not all the subsequences need to calculate DTW distance as they may have been pruned off.

### *4.1.2 Two-phase precision reduction*

The researchers using FPGA often encounter the precision problem. As floating point arithmetic or wide fixed point arithmetic does not scale well in FPGAs, precision reduction is often used to save resource cost. We try to maintain more precision in precision reduction. In our system, we separate the precision reduction into two phases. The first phase is executed by CPU, if the data are represented by floating number or very wide fixed point number, the CPU converts them into 32 bits fixed point number (shifted left x bits to maintain x bits fraction); and then send them into the *high precision domain*, which is used for data transfer, as shown in Figure 5. The second precision reduction phase is executed after normalization. With Assumption 2, the dynamic range of the values has been significantly reduced after normalization, so fixed point number of the same width can maintain more accuracy. For example, after z-normalization as formula (4), most of the tuples range from -1 to 1, and nearly no tuple's absolute value is larger than 2, no matter what the range of the original time series is. So we can represent the subsequence with 2 bit for integer, 14 bits for fraction, and 1 bit for sign. Then the subsequence will be sent into the *low precision domain*, which is the data processing domain. Finally, all the results will be converted into a label, before they are joined with the original high-precision sequence. In the next time the subsequences need to be processed, they are normalized and precision-reduced again to make sure that the precision error will not be accumulated.

### *4.1.3 Support for multi FPGAs*

A common solution to the scalability problem is to do performance-resource trade-off, i.e. support the large scale problem at the cost of performance degradation. This is a good feature, and is also what our system has. But what we can do if we don't want to slow down the performance even in the largest scale problem? The answer is to use more FPGAs (or other more devices like GPU). With this motivation, we design a loose coupling data path in our system. In fact, the data transfer between different loops only includes the original time series, tagged with two bits labels. The loose coupling makes it easy to use multi FPGA to get a further speedup. This is not a trivial or useless idea. It is very common that people would like to search for a very long pattern, for example, DNA chains. We can simply map the four loops into four boards. In fact, we can place more than one loops as same as the third loop to reduce the burden of the last loop and thus improve the throughput of the whole system.

## 4.2 Normalizer

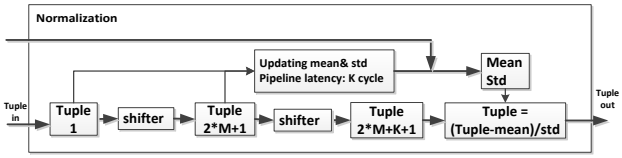



**Figure 6: Architecture of the normalizer**

As shown in Figure 6, all the tuples of the sequence will flow into the normalizer one by one. A RAM-based shifter is used to delay the sequence for *2*M* cycles, and we use the input tuple and output tuple of the shifter to update the normalization coefficients (mean & standard deviation) in every cycle, based on Equation (2,3). As the updating of coefficients needs another *K* cycles, we use another shifter to delay the sequence for *K* cycles. Then at the $(2*M+K)^{th}$ cycle, if we assert the *enable* signal high for one cycle, the coefficients of the first 2*M tuples will be stored in the register *Mean Std*, and the normalizer will start to output the normalized *2*M* tuples. In the same way, if we assert "enable" high at the $(4*M+K)^{th}$ cycle, the normalizer will output the following normalized 2*M tuples.

## 4.3 Hybrid Lower Bound

Our hybrid lower bound consists of LB_pDTW, LB_Keogh and reversed-LB_Keogh, as shown in Figure 7.

Firstly, the subsequences will flow into the LB_pDTW module. In this module, we place a processing elements (PE) matrix, as shown in Figure 8, corresponding to the partial warping matrix generated by the first *X* tuples of every subsequence and the pattern, to calculate the partial DTW distance. Each PE only calculates the accumulated distance in one cell, based on Equation (1), for example, the PE (1,1) only calculate the distance between the first tuple of every subsequence and the first tuple of the pattern. In this full-pipeline way, this module can process one subsequence per cycle. We also use a max/min filter to generate the envelopes of LB_Keogh, as Equation (5).

Secondly, the following *Y* tuples of every subsequence are sent to the LB_Keogh module, while the envelope of these *Y* tuples will be sent to the reversed LB_Keogh module. The main structure of LB_Keogh/reversed LB_Keogh module consists of a multiplexer array and an adder tree: the multiplexer array is used to select the distance of the corresponding tuples as Equation (6), and the adder tree is used to accumulate the distance as Equation (7). Note that if $X+Y>M$, the redundant element will be masked; if $X+Y<M$, we actually use a partial LB_Keogh to replace the LB_Keogh.

Finally, we use *LB_pDTW +max{LB_Keogh, reversed LB_Keogh}* as the final lower bound.

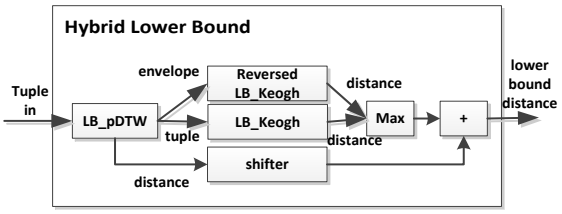



**Figure 7: Architecture of the hybrid lower bound**

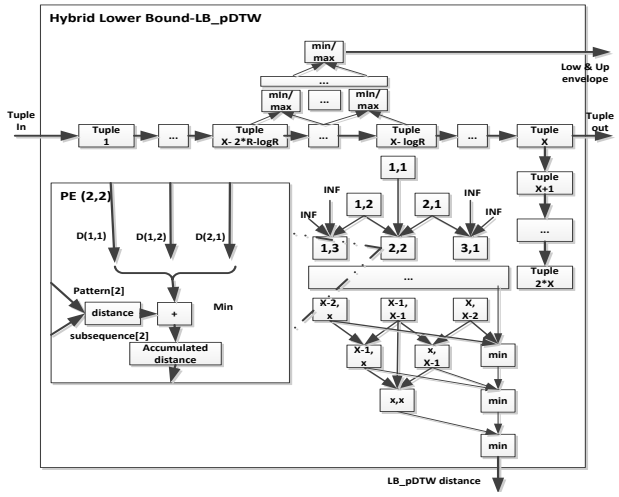



**Figure 8: Architecture of LB_pDTW. To make the figure clear, the connecting line between tuple delayed registers for tuple and the PE-matrix are not exhibited in the figure.**

## 4.4 PE-ring for DTW

Note that the SPRING algorithm only extends the warping matrix to *2M-by-M*, aiming to get multiple results with one search, but it makes no difference to the path search or data dependency. Based on Equation (1), it is easy to find that only the cells on the diagonals of the warping matrix, as the green cells in Figure 10, have no data dependency and can be calculated in parallel, with parallelism up to *M*. A trivial way to exploit this parallelism is to place a PE-array on the diagonals of the matrix. This way is used in [12], though it is generated by a C-to-VHDL tool. The PE-array structure exploits the full parallelism, but suffers from lacking flexibility and scalability. The parallelism degree (PD) changes in the starting phase, e.g. PD=1 at the first cycle and PD =2 at the second cycle, so the PEs in the array need to be frequently switched. The max parallelism degree must be equal to the pattern length and cannot be tuned, so we cannot do performance-cost trade-off and it cannot support long pattern limited by the logic resource. If a new pattern of different length is wanted, the parameter of the PE array's length must be modified, and the compilation of several hours may be unacceptable in real-time systems.

We propose a simple but effective PE-ring structure, as shown in Figure 9. All the PEs are connected one by one, and the temporal results are only transferred between the neighboring PEs. A multiplexer is used to send the pattern and the boundary condition into the ring in the starting phase, and then it is switched to the FIFO. The FIFO is used to buffer the temporal result of the last PE when all the PEs are busy. When the first PE is idle again, the FIFO sends out the temporal results to the first PE, acting as a delayed PE. A tuple router is used to route every tuple of the subsequence to the next idle PE one by one. A result router is used to route the final results of all PEs to the output port.

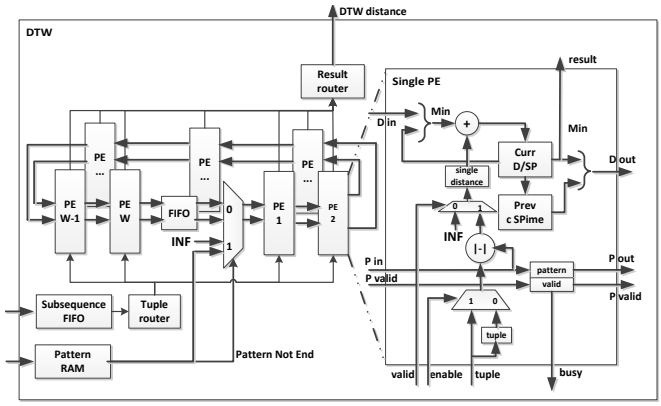



**Figure 9: PE-ring structures for DTW**

A single PE is only used to calculate one column of the warping matrix, as shown in Figure 10. With SPRING algorithm, the PE needs to record an extra label *SP*, indicating the starting position of the path, as the number in bracket in Figure 10. To finish the calculation of one column, every PE only needs the left and down boundary condition, as Equation (8,9), as well as the pattern. The left boundary condition can be obtained from the input port *D in* from the former PE, and the down boundary condition can be obtained from the delayed registers *Prev D/SP* of the previous result. The pattern only needs to be sent into the ring at the starting phase for once, and then all the tuples of the pattern flow around the ring without stop. In this way, all the PEs in the ring can surely obtain all the tuples of the pattern from the former PE.

We illustrate the process intuitively in Figure 10, where we have a ring of 7 PEs to search a pattern of length 7 in a sequence of length 14. At the 1st cycle, the first tuple of the subsequence, $s_1=8$, is sent into PE *1* from the router, along with the first tuple of pattern, $p_1=0$, from the *Pattern RAM* and the left boundary, *INF*. Then PE *1* records the starting position, $sp=1$, and starts to do calculation of cell *(1,1)* in the matrix. At the 2nd cycle, the second tuple, $s_2=1$, is sent into PE *2* from the router, along with the first tuple of pattern, $p_1=0$, from PE *1*, and the left boundary, $D(1,1)=8$, from PE 1. At the same time, PE *1* calculates cell *(1,2)* based on the second tuple of pattern, $p_2=5$ from the *Pattern RAM* and the left boundary, *INF* from the multiplexer. In this way, all the PEs in the ring start one by one at different time. At the 8th cycle, the PE 1 has finished all the calculation of the first column, and is idle again. Then PE 1 start to calculate cell *(8,1)* based on the 8th tuple of the subsequence, $s_8=0$, from the router. At the same time, the *Pattern Not End* signal is asserted low to switch the multiplexer from the pattern RAM to the FIFO, as the whole pattern has flowed into the ring. So the temporary results of PE *7* are used as new boundary condition to be sent into PE *1*, and PE 1 acts exactly as a virtual PE *8*. In this way, all the PEs finish the calculation, just like a ring rolling over the matrix. Note that here

we just want to give out an intuitive working process of our PE-ring and ignore the latency of the real FIFO.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P7=0 | INF | 26(1) | 19(1) | 23(2) | 16(2) | 12(2) | 14(2) | 12(2) | 6(2) | 14(2) | 17(8) | 11(8) | 12(8) | 14(8) | 12(8) | |
| P6=5 | INF | 18(1) | 19(1) | 19(2) | 7(2) | 5(2) | 9(2) | 5(2) | 11(2) | 10(8) | 8(8) | 5(8) | 7(8) | 9(8) | 11(8) | |
| P5=9 | INF | 15(1) | 22(1) | 18(2) | 3(2) | 5(2) | 5(2) | 8(2) | 7(2) | 7(8) | 4(8) | 7(8) | 9(8) | 11(8) | 17(8) | |
| P4=10 | INF | 14(1) | 21(1) | 13(2) | 3(2) | 5(2) | 5(2) | 8(2) | 7(2) | 7(8) | 4(8) | 7(8) | 9(8) | 11(8) | 17(11) | |
| P3=9 | INF | 12(1) | 13(2) | 7(2) | 2(2) | 4(2) | 4(2) | 7(2) | 14(8) | 4(8) | 5(8) | 6(8) | 8(8) | 10(11) | 11(14) | |
| P2=5 | INF | 11(1) | 5(2) | 2(2) | 6(2) | 8(2) | 11(5) | 7(7) | 5(8) | 3(8) | 7(8) | 7(8) | 8(11) | 9(13) | 5(14) | |
| P1=0 | INF | 8(1) | 1(2) | 4(3) | 9(4) | 7(5) | 9(6) | 6(7) | 0(8) | 8(9) | 9(10) | 6(11) | 7(12) | 7(13) | 3(14) | |
| value | | 8 | 1 | 4 | 9 | 7 | 9 | 6 | 0 | 8 | 9 | 6 | 7 | 7 | 3 | |
| time | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | |
| PE | | PE1 | PE2 | PE3 | PE4 | PE5 | PE6 | PE7 | PE1 | PE2 | PE3 | PE4 | PE5 | PE6 | PE7 | |

**Figure 10: Illustration of SPRING algorithm. The arrows imply the data dependency of DTW, and we can calculate the green cells in parallel; the red subsequence $S_{2,8}$ is the final result, as its final accumulated distance *6* is the minimum. The number in the cell is the accumulated distance, while the number in the bracket is the corresponding starting position.**

### *4.4.1 Why only FPGA can exploit the parallelism*

In our design, each column of the matrix is calculated serially in different PEs, and the parallelism is obtained by multiple running PEs. One of the most important things supports us to do this is that we can exactly control the behavior of PEs in a cycle-accurate way. As PEs start one cycle by one cycle, they operate exactly on the diagonals cells of the matrix all the time. For example, at the 13th cycle, PE *6* is calculating the cell(13,1), PE *5* is calculating the cell(12,2)…PE *7* is calculating the cell(7,7). In the next cycle, PE *6* needs the temporal result *D out* from PE *5* to continue its calculation, and PE *5* needs the temporal result *D out* from PE *4*…Benefited from the hard-wired synchronization of FPGA, the former PE can just send the result to the output port, and the temporal result can be received by the latter PE in the next cycle. However, if GPU is used in this algorithm, different threads must be placed on the diagonals of the matrix to exploit the fine-grained parallelism, but the temporal result must be stored into the shared memory and be read out to the latter PE. The latency of memory and the extra protocol to prevent the memory coherence problem slow down the next thread to continue its calculation. The only solution for GPU is to transfer all the temporal results of the whole column to the next thread through the shared memory, but this means the next thread must wait until the whole calculation of the former thread is finished, which becomes actually serial processing. Besides, the huge memory access becomes another new burden.

### *4.4.2 Support for patterns of arbitrary lengths*

In the previous FPGA implementation [12], the fabric of the PE array is fixed and the number of PEs in the array must be equal to the length of the pattern. If a new pattern of different length is wanted, we must modify the parameter of the PE number, re-compile the design and re-configure the FPGAs, which may cost several hours and be unacceptable in an on-line system. However, in our PE-ring, there are no constraints about the PE number or the pattern length. If the PE number is larger than the pattern length, the redundant PE will be idle without changing the structure. If the PE number is smaller the pattern length, the temporal result can be stored into the FIFO and wait until the next PE is idle, which slows down the performance but will not lead to function errors. In a single PE, the structure is also independent of the pattern length, as the PE only needs to know whether a tuple of pattern is *valid*, which can be easily realized by tagging the tuple with a label. So when a pattern of different length is wanted, we only need to write the new pattern into the *Pattern RAM*. Limited by the on chip logic resource, the previous PE-array cannot support very long pattern, while our PE-ring supports almost infinite long patterns if the external memory used as FIFO is large enough.

### *4.4.3 Support for flexible parallelism degree*

The ring topology hides the specificity of each isomorphic PE. PE *1* will become virtual PE *W+1*, and PE *2* will become virtual PE *W+2*, if there are *W* PEs in the ring. Any PE in the ring can be removed without causing functional errors of the whole system, and the saved resource can be allocated to other modules, at the cost of linear performance degradation of the PE-ring. If there is abundant resource, a new PE can be simply inserted into the ring to improve the performance of the ring. The flexible performance-resource trade-off can be used to tune the bottleneck of the whole system.

### *4.4.4 Performance analysis of PE-ring*

If we place *W* PEs in the ring, and we need to find a pattern of length *M* in a sequence of length *N*, then we can approximately estimate the worst throughput of our system, assuming that there is no lower bound technique pruning any subsequences.

As we pick out *M* subsequences of length *2*M* tuples from the whole sequence to generate a *2*M*-by-*M* matrix, it will need *M* cycles for one PE to finish the calculation of one column. If *W*>*M*, all the tuples can be sent to the PE-ring without stopping, as every PE only need *M* cycles to process one column of the matrix. Considering that only *M* subsequences are processed once, the throughput is *M/(2*M)* tuple/cycle, and **1 tuple/cycle** for the whole two loops. If *W*<*M*, we can only send *W* tuples before all the PEs are busy, and the next tuple must be sent at the $M^{th}$ cycle, when the first PE is idle again. This means the input rate is *W/M* tuples/cycle. So we averagely need *2*M/(W/M)* cycles to send all the *2*M* tuples into the PE-ring. Extra *M* cycles are needed to wait for the calculation of the last column of the warping matrix, because we are not sure whether the first PE is idle now and we use the worst case to do estimation. Then we need totally *2*M/(W/M)+M* cycles to get the result of the *M* subsequences, achieving a throughput up to *M/(2*M/(W/M)+M) = W/(2*M+W)*. As we have two DTW modules, the total throughput will be ***2*W/(2*M+W)*** **tuples/cycle**. In our system, there are two rings, each consisting of 512 PEs. Our system runs at 150MHz, and we use 64 bits to store one tuple. So without the lower bound's pruning effect, the worst throughput is 1228.80/(*2*M*+512) GB/s if *M*>512, and 1.2 GB/s if *M*<512.

## 5. EXPERIMENTAL RESULT

### 5.1 Experiment Setup

Our system is implemented on TERASIC Company's Altera DE 4 Board with a Stratix IV GX EP4SGX530 FPGAs[15]. The board has an X8 PCI Express interface and is housed in an ASUS workstation PC with a 2.8GHz Intel i7-930 CPU, 16GB 1333MHz DDR3, and running Windows 7. The software program is also executed on this PC.

The resource cost of our system is shown as follows, with parameter *X*=8, *Y* =512 in lower bound, *W*= 512 in DTW, and data precision is 16 bits. The following experiments are all conducted under this setting.

| | | |
|---|---|---|
| Combinational ALUTs | 362,568/424,960 | (85%) |
| Dedicated logic registers | 230,160/424,960 | (54%) |
| Memory bits | 1,902,512/21,233,664 | (9%) |

The "TimeQuest Timing Analysis" in Quartus II 11.0 reports the frequency max is 167.8MHz, and we run it on 150MHz quite well.

## 5.2 Comparison with Software

T. Rakthanmanon et al. exploit almost all the possible software speedup techniques and got a high performance [10]. Their fast speed helps them to win the SIGKDD 2012 best paper award. So we choose their method as the best software implementation to be compared with. We download their open source program, and run it with different datasets. Unlike the parallel hardware solution in section 5.3, their software constrains the path with the parameter R, as in section 2.2. Although some other researchers insist that there should be no (or larger) constrain to improve the fault tolerance, they claim that the constraint should be as small as about 5% to prevent pathological warping. In our opinion, the constraint *R* is an application-dependent parameter. Though we test their program in cases that *R* is set to be a large one in some dataset, we only show the result as a comparison of computation power in extreme cases, not standing for that the larger constraint can improve the high level accuracy in these applications.

### *5.2.1 Dataset 1: random walk*

Random walks are often used to model many applications. To avoid the difference of the random number generator and the seeds used, we use the existing sequence attached with their code [10] and search for the same pattern, which are both attached with their code. This dataset has 1 million points, and the pattern (query) length is 128. We try to modify the parameter *R*, which means we try to find patterns of different warping degrees, and show the result in logarithmic coordinates in Figure 11. Benefited from the great power of PE-ring structure in DTW calculation, our performance does not depend on the parameter R in the case that pattern length is shorter than 512, while their performance significantly decreases as *R* increases. We achieve two orders of magnitude (1.924s/0.008s=240) speedup in the case that *R*=5%, and four orders of magnitude (223.7s/0.008s =27962) speedup in the case that *R*=100%, which is the classic unconstrained DTW.

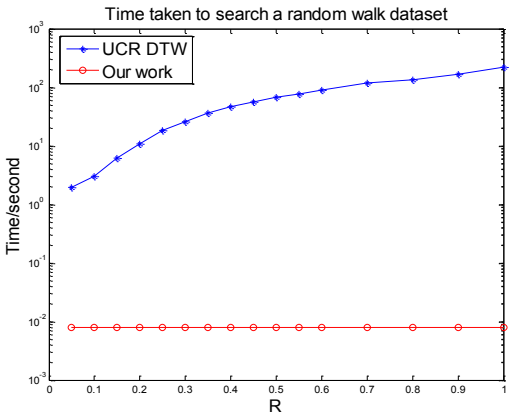



**Figure 11: Performance comparisons on a random walk data set, pattern length =128 (logarithmic coordinates).**

### *5.2.2 Dataset 2: medical data*

This is a traditional domain in which subsequence similarity search is greatly needed, for example, temperature monitoring, electrocardiogram, motion monitoring, gesture recognition, etc. We download a dataset of electrocardiogram (ECG) from the web site of [10]. This dataset has about 8G points, and we need to find a pattern of length 421, as shown in Figure 3(d), with *R* = 5%. We get a 19.28x faster performance compared with their work.

**Table 1: Time taken to search one year of ECG data**

| | UCR_DTW[10] | Our work | Speedup |
|---|---|---|---|
| ECG | 18.0 minutes | 56 seconds | 19.28 |

### *5.2.3 Dataset 3: speech recognition*

Speech recognition is another domain where needs DTW as a kernel subroutine. We download the CMU_ARCTIC speech synthesis databases from [16], and construct a speech of 1 minute by splicing together the first 21 utterances of all the 1132 utterances. Then we get a time series of about 1 million points, and we randomly cut out patterns from this speech. Some of the software cases are too time-consuming that we don't finish the test in more than one day, so we do not list them. Note that in traditional speech recognition, DTW distance is usually defined upon some features, instead of time series, e.g. MFCC, which are extracted from the frames of the speech. As we only want to evaluate the subsequence similarity search, we directly search the pattern on the speech time series, without any feature extraction or other operations. We achieve two orders of magnitude (0.827s/0.008s =103) speedup in the case that the pattern length =128, *R*=5%, and three orders of magnitude (634s/0.6s =1057) speedup in the case that pattern length = 16384, *R*=5%. Larger speedup can be achieved in the cases that *R* is larger than 5%, for example, in the case that pattern length = 16384, *R*=20%, the speedup is up to 63432(31716s/0.5s).

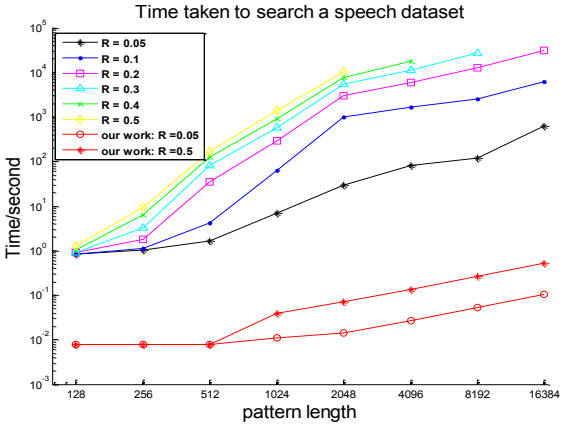



**Figure 12: Performance comparisons on a Speech Recognition dataset (logarithmic coordinates). Only the up bound (R=0.5) and low bound(R=0.05) of our work is shown to make the figure clear.**

In [10], they do best-match-search while we do local-best-match search in our design. When their work encounters a similar subsequence at an early position, the threshold will be significantly reduced to the distance between the current subsequence and the pattern, and then lower bound technique can prune off more subsequences in the following search. Therefore, the search speed is highly dependent on where the first similar subsequence is, and this explains their fast search speed in ECG data set, which consists of repeated heartbeats pattern. As we mentioned before, lower bound aims to prune off subsequences in a cheap way, so the larger constraint makes lower bound less-effective and leads to more calling times of DTW. Besides, the DTW complexity increases with square of pattern length, while the lower bound complexity increases linearly with the pattern length. They both explain the higher speed-up in the cases that the pattern length is larger or the constraint *R* is larger.

## 5.3 Comparison with Hardware

D. Sart et al. [12] claim themselves to be the first using FPGAs to accelerate DTW based subsequence similarity search, though their FPGA design is generated by a C-To-VHDL tool. In their experiment, a data set records a fluctuating voltage occurring when the insect's stylet penetrates the plant, called the Electrical Penetration Graph (EPG) signal. This data set has 1,499,000 points, and the pattern length is 360. They need 80.39s with GPU, and 2.24 seconds with FPGAs, for the whole search, while we only need 0.011 second, achieving two orders of magnitude speedup to their FPGA implementation, and three orders of magnitude speedup to their GPU implementation.

**Table 2: Time taken to search EPG data set**

| EPG data set | D. Sart [12] | Our work | speedup |
|---|---|---|---|
| GPU | 80.39s | 0.011s | 7398 |
| FPGA | 2.24s | | 203 |

The main reason for the huge speedup is that GPU can't exploit the fine-grained parallelism of DTW, as stated in Section 4.4; although their FPGAs implementation exploits the fine-grained parallelism, the lack of computation-reuse feature makes them lose the speedup from the coarse-grained parallelism, which is also up to *C+1 by SPRING*. Both of their GPU and FPGA designs do not use lower bound pruning technique, which also slows down their performance. Their FPGA implementation cannot support pattern longer than 1024, while we can support nearly unlimited long pattern, benefited from our flexible and scalable PE-ring. Besides, the parallelism degree of their work cannot be tuned, as discussed in section 4.4.2.

Y. Zhang et al. [4] propose another GPU implementation, which parallelize the generation of the warping matrix but still process the path search serially. We cannot access to their dataset. Besides, in their implementation, DTW based subsequence similarity search is embedded in feature extraction of the speech recognition application. So we do not have a direct comparison, but we try to evaluate our work in an indirect approach: they claimed their work is 2 times faster than the GPU implementation of [12], so our work should be about 3,699 times faster than theirs. The reason for the speedup is similar. Although the parallel generation of the distance matrix makes it a little faster than the former GPU implementation, the serial path search limits the further speedup.

# 6. CONCLUSION

In this paper, we propose a novel stream-oriented framework for subsequence similarity search, and a novel PE-ring structure for DTW calculation. The framework guarantees accuracy with two-phase precision-reduction while reducing resource cost. It uses lower bound technique to prune off most of the subsequence, and utilizes the computation reusability to exploit the coarse-grained parallelism of subsequences similarity search. The PE-ring exploits the full fine-grained parallelism of DTW while supports on-line updating pattern of arbitrary lengths. Besides, flexible parallelism degree is achieved to do performance-cost trade-off. We also analyze why only FPGA can exploit such parallelism. The experimental results show that we can achieve several orders of magnitude speedup in accelerating subsequence similarity search compared with the best software and current GPU/FPGA implementations.

# 7. ACKNOWLEDGMENTS

This work was supported by IBM Research China University Relationship Program, National Science and Technology Major Project (2011ZX03003-003-01), National Natural Science Foundation of China (No.61271269, 61028006), and Tsinghua University Initiative Scientific Research Program.

# 8. REFERENCES

[1] Adams, N., Marquez, D., and Wakefield, G. 2005. Iterative deepening for melody alignment and retrieval. *ISMIR*.

[2] Alon, J., Athitsos, V., Yuan, Q., and Sclaroff, S. 2009. A unified framework for gesture recognition and spatiotemporal gesture segmentation. *IEEE PAMI* 31, 9, 1685-1699.

[3] Chadwick, N., McMeekin, D., and Tan, T. 2011. Classifying eye and head movement artifacts in EEG Signals. *DEST*, 285-291.

[4] 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 measures. *PVLDB* 1, 2, 1542-52.

[5] 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

[6] 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*

[7] 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 measures. *VLDB J.* 18, 3, 611-630.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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

[13] 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.

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

[15] www.terasic.com.tw

[16] www.festvox.org/cmu_arctic/

[17] 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.