2012 IEEE 12th International Conference on Data Mining

Efficient Pattern-Based Time Series Classification on GPU

Kai-Wei Chang[1], Biplab Deka[2], Wen-Mei W. Hwu[2], Dan Roth[1]
[1]*Dept. of Computer Science,* [2]*Dept. of Electrical and Computer Engineering*
University of Illinois at Urbana-Champaign, IL USA
{kchang10, deka2, w-hwu, danr}@illinois.edu

***Abstract*—Time series shapelet discovery algorithm finds subsequences from a set of time series for use as primitives for time series classification. This algorithm has drawn a lot of interest because of the interpretability of its results. However, computation requirements restrict the algorithm from dealing with large data sets and may limit its application in many domains. In this paper, we address this issue by redesigning the algorithm for implementation on highly parallel Graphics Process Units (GPUs). We investigate several concepts of GPU programming and propose a dynamic programming algorithm that is suitable for implementation on GPUs. Results show that the proposed GPU implementation significantly reduces the running time of the shapelet discovery algorithm. For example, on the largest sample dataset from the original authors, the running time is reduced from half a day to two minutes.**

***Keywords*-Time Series, GPU, Classification, Pattern-based Classification**

I. Introduction

Time series classification has several important real world applications [1] from intrusion detection to genomic research. Ye and Keogh [2] illustrate several examples including an example for classifying leaves of *Urtica dioica* and *Verbena urticifolia* (see Figure 1 in [2]). The approach described in [2] entails converting the problem into a time series classification problem and then using a new pattern-based methodology for classification. The pattern-based method, called shapelet discovery, aims at finding the subsections (called *pivot*) of the time series that are predictive of a class and useful for classification. It builds a decision tree-like classifier based on the shapelets and makes decision by examining if the distance between the pivot and any subsequence of the objects is less than a threshold.

Although the pattern-based method is shown to provide accurate, interpretable classification results for problems from a variety of domains, it requires a significant computational effort. In the training stage of the algorithm, the time complexity of a naive implementation to discover the pivots (the *shapelet discovery* algorithm) is $O(N^2L^4)$, where N is the number of time series and L is the length of the time series. A recent paper [3] proposed reuse of computations and reduces the time complexity to $O(N^2L^3)$. However, it is still expensive for several data sets of interest. Results presented in [3] suggest that for several problems of interest, the running time could be as high as 10^4 to 10^5 seconds.

In this work, we mitigate this limitation of the shapelet discovery algorithm by modifying the algorithm for execution on Graphics Processing Units (GPUs) and show that it is possible to achieve orders of magnitude of speedup with this approach. GPUs currently have good support for general purpose computation and come bundled with standard desktops making it an attractive option for speeding up applications. Recently, GPUs have been widely applied in several fields ranging from signal processing to computational finance [4]. GPUs have also been successfully applied in speeding up several data mining and machine learning models such as SVM [5], [6], unsupervised deep learning [7], and LDA [8]. However, it should be noted that achieving good speedups from a GPU implementation requires a good understanding of GPU architecture and programming model. We present a detailed discussion of these factors and show how our implementation exploits them for better performance.

Sart et al. [9] present an earlier attempt to accelerate time series classification by using GPUs for dynamic time warping (DTW) subsequence search. Their problem setting is different from ours as their goal is to find the best time series that matches a given query or a pivot. However, in the shapelet discovery problem that we consider, our aim is to find the most representative pivot.

The contributions of this work are:

- We develop a dynamic programming algorithm for finding shapelets on GPU. The proposed algorithm is simple and has potential to be extended for other pattern-based learning algorithm and applications.
- We show that the shapelet discovery algorithm can derive significant performance gains from a GPU based implementation. This makes the algorithm a viable option for larger datasets and allows it to perform more extensive search for a better shapelet.
- We also present different optimizations of the GPU implementation that make it possible to achieve significant speedups. This, we believe, would serve as a tutorial for other data mining researchers interested in implementing their algorithms on a GPU.

The rest of this paper is organized as follows. Section II discusses the details of the serial version of the pattern based time series classification approach as presented in [2] and [3]. We discuss the details of GPU devices in

Algorithm 1 Shapelet Discovery Algorithm in [2]

```
ShapeletDiscovery($\mathcal{D}$):
 1) Compute the initial entropy $E(\mathcal{D})$.
 2) For all pivots $\boldsymbol{T}_{i,s,l}, i \in [1,N], l \in [1,L], s \in [1, L-l]$.
    a) Compute $dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_j), \forall j \in [1,N]$.
    b) Sort $\mathcal{D}$ by the distances s.t. $\boldsymbol{T}_{\mathcal{L}(j)}, \forall j$ are in order.
    c) For $j = 1 \ldots L-1$, sequentially visit $\boldsymbol{T}_{\mathcal{L}(j)}$
       i) Compute information gain $I(\boldsymbol{T}_{i,s,l}, \theta_j)$ by (3),
          where $\theta_j$ is defined in (4).
       ii) $(\boldsymbol{T}_{i^*,s^*,l^*}, \theta^*) \leftarrow \text{argmax}(I(\boldsymbol{T}_{i,s,l}, \theta_j), I(\boldsymbol{T}_{i^*,s^*,l^*}, \theta^*))$.
 3) Return $(\boldsymbol{T}_{i^*,s^*,l^*}, \theta^*)$.
```

Section III. Sections IV and V present the details of our parallel implementation of the shapelet discovery algorithm suitable for execution on GPUs. Our experimental results are presented in Section VI. Finally, Section VII concludes the work. Our code is available at http://cogcomp.cs.illinois.edu.

II. Pattern-Based Time Series Classification

Various classification techniques have been applied for time series classification (e.g., [10], [11], [12]). Recently, [2], [3] proposed a shapelet based classification approach based on a decision tree-like algorithm [13] and showed that the algorithm is able to identify the informative patterns of time series that are predictive of a class label. They argue that for many situations, building such an interpretable model is more important. In this section we describe the shapelet discovery algorithm for time series data and analyze its complexity. Readers may refer to [2] for illustrative examples of how this approach is useful for several real world applications.

Given a data set $\mathcal{D} = \{\boldsymbol{T}_i, y_i\}_{i=1}^N$, where $\boldsymbol{T}_i = \langle T_{i,1}, \ldots, T_{i,L}\rangle$ is a real-valued sequence of length L, and $y_i \in \{1, \ldots, C\}$ is a multi-class label, the objective of the shapelet discovery algorithm is to find a contiguous subsequence belonging to one of the time series and a distance threshold that can be used to classify the data set. For simplicity, we assume all the time series in a data set have same lengths. One can easily extend the model to deal with time series in various lengths. The shapelet discovery algorithm first generates all candidate subsequences

$$\boldsymbol{T}_{i,s,l} = \langle T_{i,s}, \ldots, T_{i,s+l}\rangle, \forall i \in [1,N], l \in [1,L], s \in [1, L-l].$$

For each of these subsequences (called pivots), it computes the distance to each of the time series based on the following definition.

Definition 1
Distance from a time series $\boldsymbol{T}_j$ to a pivot $\boldsymbol{T}_{i,s,l}$ is defined as the minimum of the distance between $\boldsymbol{T}_{i,s,l}$ and any subsequence $\boldsymbol{T}_{j,u,l}, u = [1, L-l]$ derived from $\boldsymbol{T}_j$ whose length is equal to that of $\boldsymbol{T}_{i,s,l}$. That is,

$$dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_j) = \min\nolimits_{u \in [1,L-l]} dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_{j,u,l}). \quad (1)$$

Different measures of distance (example Euclidean distance) between $\boldsymbol{T}_{i,s,l}$ and $\boldsymbol{T}_{j,u,l}$ can be used for the shapelet discovery algorithm. In this paper, we follow [3] to use the normalized Euclidean distance[1]:

$$dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_{j,u,l}) = \sqrt{\frac{1}{l}\sum_{t=0}^{l-1}\left(\frac{T_{i,s+t} - \mu_{i,s,l}}{\sigma_{i,s,l}} - \frac{T_{j,u+t} - \mu_{j,u,l}}{\sigma_{j,u,l}}\right)^2}, \quad (2)$$

where $T_{i,s+t}$ is the $(s+t)$th element of $\boldsymbol{T}_i$ and $\mu_{i,s,l}$ and $\sigma_{i,s,l}$ are the mean and standard deviation of $\boldsymbol{T}_{i,s,l}$, respectively. Once the distances are found, the pivot splits the data set $\mathcal{D}$ into two groups $\mathcal{D}_L$ and $\mathcal{D}_R$ with a threshold θ, such that

$$\begin{aligned} dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_j) &\le \theta, \quad \forall \boldsymbol{T}_j \in \mathcal{D}_L, \\ dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_j) &> \theta, \quad \forall \boldsymbol{T}_j \in \mathcal{D}_R. \end{aligned}$$

The algorithm seeks the best pivot $\boldsymbol{T}_{i^*,s^*,l^*}$ and the best threshold θ^* to maximize the quality of the split. The quality of the split is measured by the information gain:

$$I(\boldsymbol{T}_{i,s,l}, \theta) = E(\mathcal{D}) - \frac{|\mathcal{D}_L|}{|\mathcal{D}|}E(\mathcal{D}_L) - \frac{|\mathcal{D}_R|}{|\mathcal{D}|}E(\mathcal{D}_R), \quad (3)$$

where $E(\cdot)$ is an entropy function defined by

$$E(\bar{\mathcal{D}}) = -\sum_{c=1}^{C} \frac{|\{\boldsymbol{T}_j \in \bar{\mathcal{D}} \mid y_j = c\}|}{|\bar{\mathcal{D}}|} \log\left(\frac{|\{\boldsymbol{T}_j \in \bar{\mathcal{D}} \mid y_j = c\}|}{|\bar{\mathcal{D}}|}\right).$$

Ye and Keogh [2] develop an algorithm to find the best θ given a pivot $\boldsymbol{T}_{i,s,l}$ using an order-line. First, the time series are sorted according to the distances, so $\boldsymbol{T}_{\mathcal{L}(j)}, \forall j$ is in order, where $\mathcal{L}(\cdot)$ is an index function. Although θ can be any real number, there are at most $L-1$ possible combinations to split $\mathcal{D}$ into $\mathcal{D}_L$ and $\mathcal{D}_R$ based on the distances. Therefore, $\boldsymbol{T}_{\mathcal{L}(1)}, \boldsymbol{T}_{\mathcal{L}(2)}, \ldots, \boldsymbol{T}_{\mathcal{L}(L-1)}$ are sequentially visited and for each $\boldsymbol{T}_{\mathcal{L}(j)}$, $I(\boldsymbol{T}_{i,s,l}, \theta_j)$ is computed with

$$\theta_j = \frac{dist\left(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_{\mathcal{L}(j)}\right) + dist\left(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_{\mathcal{L}(j+1)}\right)}{2}, \quad (4)$$

and

$$\theta^* = \theta_{j^*}, j^* = \arg\max\nolimits_j I(\boldsymbol{T}_{i,s,l}, \theta_j)$$

is calculated. Finally, a shapelet is the pivot corresponding to the highest information gain with the best split. To break a tie, if two pivots have the same information gain, the one with the larger separation gap is favored, where the separation gap is defined as

$$\frac{1}{|\mathcal{D}_R||}\sum_{\boldsymbol{T}_j \in \mathcal{D}_R} dist(\boldsymbol{T}_j, \boldsymbol{T}_{i,s,l}) - \frac{1}{|\mathcal{D}_L|}\sum_{\boldsymbol{T}_j \in \mathcal{D}_L} dist(\boldsymbol{T}_j, \boldsymbol{T}_{i,s,l}).$$

Algorithm 1 summarizes the shapelet discovery algorithm.

To analyze the time complexity of algorithm 1 we note that computing the distance from a pivot to a time series by

[1]In order to remove offset and difference in scale, *Mueen et al.* [3] first normalized the time series to zero mean and unit variance. Then, they computed a length-normalized distance.

Algorithm 2 Building Decision Tree

```
BinaryDecisionTree($\mathcal{D}$, nodeD):
  1) If $\mathcal{D}$ satisfies the stopping condition: return.
  2) $(\boldsymbol{T}^*, \theta) \leftarrow$ ShapeletDiscovery(D).
  3) Assign $(\boldsymbol{T}^*, \theta)$ to nodeD.
  4) $\mathcal{D}_L \leftarrow \{\boldsymbol{T}_j \in \mathcal{D} \mid dist(\boldsymbol{T}_j, \boldsymbol{T}^*) \leq \theta\}$.
  5) $\mathcal{D}_R \leftarrow \{\boldsymbol{T}_j \in \mathcal{D} \mid dist(\boldsymbol{T}_j, \boldsymbol{T}^*) > \theta\}$.
  6) BinaryDecisionTree($\mathcal{D}_L$, left child of the nodeD).
  7) BinaryDecisionTree($\mathcal{D}_R$, right child of the nodeD).
```

Eq. (1) is $O(L^2)$ as evaluating Eq. (2) is $O(L)$. There are total N time series. Therefore Step 2a in Algorithm 1 takes $O(NL^2)$. To sort the time series based on the distances (Step 2b) requires $O(NlogN)$. By tracing $|\{\boldsymbol{T}_j \in D_L | y_j = c\}|$ and $|\{\boldsymbol{T}_j \in D_R | y_j = c\}|$, the information gain can be computed in constant time, and the inner loop (Step 2c) costs only $O(N)$. Therefore, the computational complexity of each pivot is $O(NL^2)$, and the overall complexity is $O(N^2L^4)$ as there are $O(NL^2)$ pivots.

Using the shapelet discovery algorithm, a binary decision tree can be built by a recursive process. Algorithm 2 lists the procedure. The program starts with a root node and the entire training set. For each node, it finds a shapelet and a corresponding threshold using the shapelet discovery algorithm. The shapelet and the threshold are assigned to the current node as a decision rule, and are applied to separate the data set into two subsets. Then the process recursively generates the left and right children of the node over the two subsets. The procedure stops when a stopping condition is satisfied.

An Improved Algorithm from [3]. The bottleneck of Algorithm 1 is the distance calculation. *Mueen et al.* [3] proposed an algorithm to reduce the distance calculation of each pivot from $O(NL^2)$ to $O(NL)$. They rewrite Eq. (2) as

$$dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_{j,u,l}) = \sqrt{2\left(1 - \frac{\Omega_{i,j,l,s,u} - l\mu_{i,s,l}\mu_{j,u,l}}{l\sigma_{i,s,l}\sigma_{j,u,l}}\right)},$$

where

$$\Omega_{i,j,l,s,u} = \boldsymbol{T}_{i,s,l} \cdot \boldsymbol{T}_{j,u,l} = \sum_{t=0}^{l-1} T_{i,s+t}T_{j,u+t}. \qquad (5)$$

In [3], they proposed pre-computing some statistics of the data, such that $\mu_{i,s,l}, \mu_{j,u,l}, \sigma_{i,s,l}, \sigma_{j,u,l}$, and $\Omega_{i,j,l,s,u}$ can be evaluated in constant time, and the distance calculation can be speeded up. The pre-computation requires $O(NL^2)$ space to store temporary information if the pivots are examined sequentially. However, if the pivots are examined simultaneously, it requires $O(N^2L^2)$ space in total. In Section V-B, we illustrate why the algorithm proposed in [3] is not suitable for our GPU implementation. We also propose an alternative dynamic programming algorithm to reduce the

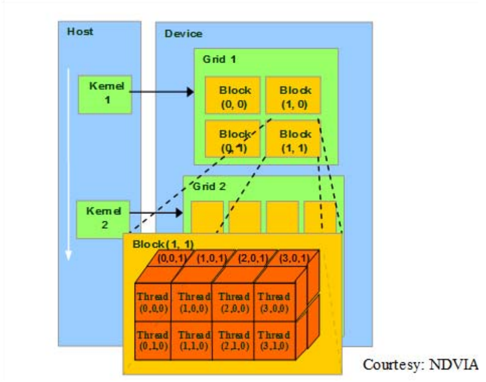

Figure 1: An illustration of the GPU programming model [14].

computational complexity of the distance calculation. *Mueen et al.* [3] also proposed a pruning technique by computing an upper-bound of the information gain of a pivot. We show in Sections VI-A and VI-C that the pruning technique is not useful in our implementation.

Logical Shapelets. In a decision tree model, the tree structure implies logical operations. The decisions made sequentially in a path from the root to a leaf are combined with an AND operation, while different decision paths that generate the same output label at leaves imply an OR operation. *Mueen et al.* [3] proposed an alternative way that makes use of logical shapelets to directly inject logic operations into the decision tree. The logical shapelet is defined recursively by

$$\begin{aligned}&\text{LogicShapelet} := \text{LogicShapelet} \wedge \text{LogicShapelet},\\ &\text{LogicShapelet} := \text{LogicShapelet} \vee \text{LogicShapelet},\\ &\text{LogicShapelet} := \text{Shapelet},\end{aligned}$$

where $\wedge$ and $\vee$ represent AND and OR logic operators, respectively. To discover logical shapelets, they further define the following distance functions

$$\begin{aligned}dist(\boldsymbol{T}_j, \boldsymbol{T}_{s_1} \wedge \boldsymbol{T}_{s_2}) &= \max(dist(\boldsymbol{T}_j, \boldsymbol{T}_{s_1}), dist(\boldsymbol{T}_j, \boldsymbol{T}_{s_2})),\\ dist(\boldsymbol{T}_j, \boldsymbol{T}_{s_1} \vee \boldsymbol{T}_{s_2}) &= \min(dist(\boldsymbol{T}_j, \boldsymbol{T}_{s_1}), dist(\boldsymbol{T}_j, \boldsymbol{T}_{s_2})),\end{aligned}$$

where $\boldsymbol{T}_{s_1}, \boldsymbol{T}_{s_2}$ are two logical shapelets, and $\boldsymbol{T}_j$ is a time series. Then, logical shapelets can be found by using a modified shapelet discovery algorithm. They show that for some data sets, logical shapelets outperform shapelets in terms of accuracy.

III. Graphics Processors

In this section we briefly discuss the details of the GPU architecture that are relevant to understanding the different considerations in our parallel shapelet algorithm discussed in Sections IV and V. In this work we use a NVIDIA GeForce GTX 480 GPU and we use the Compute Unified Device Architecture (CUDA) tools for programming. It should however be noted that the design considerations

Memory	Read/Write Speed	Size	Scope
register	∼1 cycle	32kB (per SM)	thread
shared	∼5 cycles	48kB (per SM)	block
global	∼500 cycles	1.5GB	grid
constant	∼5 cycles (w/caching)	64kB	grid

Table I: The access speed, size and the scope of different memories found on the GTX 480 GPU.

discussed in this work are also relevant for other graphics processors.

A. CUDA Programming Model

Figure 1 shows the thread architecture in the CUDA programming model. In the CUDA programming model the host (CPU) code launches kernels for execution on the GPU. These kernels consist of several threads organized into one-dimensional, two-dimensional or three-dimensional thread blocks. Each thread block is a batch of threads that synchronize and cooperate with each other through shared resources and are guaranteed to run simultaneously. The number of threads in a thread block is specified by the programmer, and is up to 1024 in GTX 480. As several thread blocks execute in parallel, the number of threads simultaneously running in the GPU can be very large. At a higher level, thread blocks are also organized into grids. Each grid contains many thread blocks that conceptually execute in parallel, while different grids run sequentially on the GPU.

GPUs contain several types of memories for different proposes. Understanding the details of them is essential for extracting good performance from any GPU implementation. Table I lists the access speed, size and the scope of different memories found on the GTX 480 GPU. All threads have access to a large global memory with long access latency. Each block can access its own shared memory which is smaller but has lower access latency and can be used to share data between the threads in the same block. Each thread has registers which are fast, but each thread can only access its own registers. There is also constant memory allowing programmers to store read-only values. With caching, accessing to constant memory is fast.

It should be noted that the latency of memory access can severely limit the throughput of a GPU implementation. Therefore, a careful choice of memory type when implementing an algorithm can significantly improve the performance. For example, we discuss in Section V-A how global memory accesses can be reduced by using shared memory and demonstrate the performance improvement for our implementation in Section VI-B.

B. GPU Hardware Details

The above CUDA programming model provides an abstract view of programming on massively parallel GPU devices. Programming under the abstract model allows the

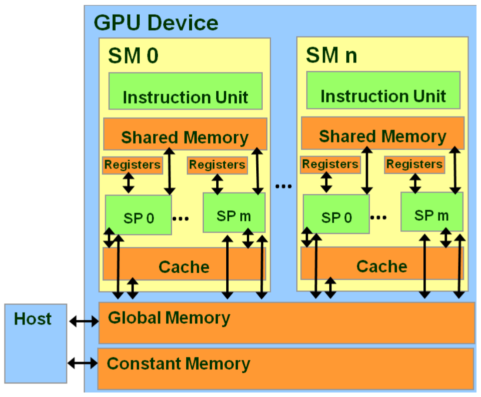

Figure 2: An illustration of a GPU device.

code to be executed on other GPU devices with the same compute compatibility. However, to fully utilize the device, we need to have a better understanding about the GPU hardware.

Figure 2 shows a schematic of a typical NVIDIA GPU. The GPU consists of several Multiprocessors (MPs) with each multiprocessor containing several Stream Processors (SPs). For example, GTX 480 has 15 SMs, each with an instruction unit, 8 SPs and its own physical shared memory, cache and register file. At most 8 blocks can be assigned to each SM if the device resources are sufficient. The actual number of blocks assigned to each SM, called occupancy, depends on the number of threads and the amount of registers and shared memory used by each thread.[2] Once a block is assigned to a SM, the threads in the block are further divided into 32-thread units called warps. A warp is a basic unit in the GPU thread scheduling and all the 32 threads in the same warp execute the same instruction on SPs. This style of parallel execution is called Single Instruction Multiple Data (SIMD) and it reduces the effort for fetching and decoding instructions. The warps residing in an SM time-share the computational resources of the SM. Although each SM only executes one warp at a time, time-sharing with other warps residing in the same SM help to amortize the delay of long latency operations. For example, when a warp needs to wait for accessing data from memory, the warp will be put into a waiting phase. Then, one of the other warps will be scheduled for execution.

IV. A Naive GPU implementation

In the following two sections, we describe our GPU implementation of the shapelet discovery algorithm. We begin by introducing a baseline implementation that implements a brute-force algorithm. Then, in Section V we will discuss how this algorithm can be improved by taking into account various details of execution on GPUs.

[2]The occupancy can be computed using the spreadsheet at NVIDIA's website.

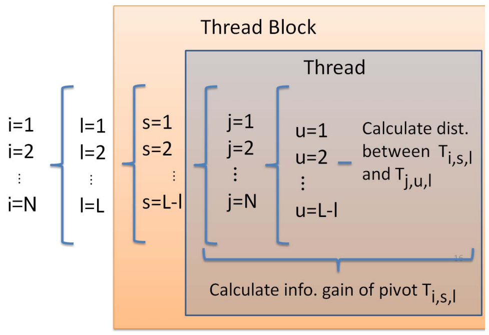
Thread Block
Thread
i=1
i=2
i=N
l=1
l=2
l=L
s=1
s=2
s=L-l
j=1
j=2
j=N
u=1
u=2
u=L-l
Calculate dist. between T_{i,s,l} and T_{j,u,l}
Calculate info. gain of pivot T_{i,s,l}

Figure 3: An illustration of how we distribute the workload among threads and blocks. The shapelet discovery algorithm requires to compute the distances of each pivot $\boldsymbol{T}_{i,s,l}$ to all N time series and to calculate the information gain associated with the pivot (see Algorithm 1 for details). In our implementation, each thread handles the computation corresponding to one pivot. Thread blocks consist of threads that deal with pivots of the same length and belonging to the same time series.

The shapelet discovery algorithm described in II can be summarized as follows. For each time series (of the given N time series of length L), all possible contiguous subsequences (called *pivots*) of length 1 to L (or from some parameters *minLength* to *maxLength* in practice) are generated. Now for each of these pivots, the distances of this pivot to all the N time series are computed and the time series are sorted according to these distances. Finally, these N time series are partitioned so that the corresponding information gain is maximized. Thus each pivot can be associated with a resulting *information gain*. The pivot with the highest associated information gain is chosen as the shapelet.

An illustration of how we parallelize the computation for GPUs can be found in Figure 3. For efficient implementation of this parallel computation on GPUs we need to address the following questions:

1) ***How do we distribute the work among individual threads?*** There is a tradeoff in the amount of work assigned to each thread. Increasing the work assigned to each thread results in increased hardware resources (registers for example) needed by each thread. Since the hardware resources on a SM are limited, this limits the number of threads that can be assigned to an SM. This limits the total number of threads that can be scheduled on the GPU at any time, thus reducing the parallelism of the implementation. On the other hand, decreasing the work assigned to each thread increases the number of threads and results in higher overhead of creating threads and thread blocks. Also, small amount of work in each thread limits the possible cooperation between threads in a thread block. *In our implementation, we assign the work associated with each pivot to a thread.* This includes calculating distances to all N time series, sorting and information gain calculation. In our naive implementation, we implement the $O(NL^2)$ brute-force algorithm to compute distances, and use a non-recursive[3] quick sort algorithm. Therefore, the computing complexity of each thread is $O(NL^2)$. The overall complexity is $O(N^2L^4)$ as there are $O(NL^2)$ threads in total.

2) ***How do we group these threads into thread blocks?*** As mentioned in Section III threads are grouped into thread blocks. 32 threads from the same block constitute a warp and all threads from the same warp are executed together under a SIMD model. Thus it is important to group threads with similar amount of work into a block. Otherwise, a thread with a long running time will result in high wait penalty for other threads. Also, since multiple threads (in a warp) execute the same instruction in lockstep, it can lead to inefficiencies when different threads take different control paths at a branch. This is called control divergence and is generally resolved by executing instructions from one control path for a subset of the threads and then re-executing along the other control path for the remaining threads. This can severely limit the throughput of an SM and hence similar threads need to be grouped into a block to reduce the chances of control divergence. Also, similarity of the work done in the threads also increases the chances of cooperation within the threads in the thread block. *In our implementation, all thread blocks consist of threads corresponding to pivots that are of the same length and are derived from the same time series.*

Note that in the baseline implementation, the threads do not cooperate with each other. Therefore, threads waste device resources and computation time in redundant work. In the next section, we exploit redundancy in memory accesses (Section V-A) and computations (Section V-B) to improve our implementation.

Implementing Logical Shapelet Discovery Algorithm on GPUs. Here we briefly describe how to implement the logical shapelet discovery algorithm on GPUs. Given the current best logical shapelet $\boldsymbol{T}_{L^*}$ and a logical operation $\oplus \in \{\wedge, \vee\}$, we can pre-compute $d_{L^*,j} = dist(\boldsymbol{T}_{L^*}, \boldsymbol{T}_j), \forall j \in [1, N]$ and store the distances in constant memory. Then, we can replace the distance calculation in Step 2a of Algorithm 1 as

2.a.1 Compute $d_{i,s,l,j} \leftarrow dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_j), \forall j$.
2.a.2 if $\oplus = \wedge$, $dist(\boldsymbol{T}_{i,s,l} \wedge \boldsymbol{T}_{L^*}, \boldsymbol{T}_j) = \max(d_{i,s,l,j}, d_{L^*,j})$.
2.a.3 if $\oplus = \vee$, $dist(\boldsymbol{T}_{i,s,l} \vee \boldsymbol{T}_{L^*}, \boldsymbol{T}_j) = \min(d_{i,s,l,j}, d_{L^*,j})$.

In the following sections we present results for only the original shapelet discovery algorithm. However, as shown above it can easily be extended to the logical shapelet algorithm.

[3]In current CUDA C++, it is not allowed to implement recursive function in the device.

V. Performance Considerations and Implementation Details

In this section, we describe various factors that affect the performance of the parallel implementation on GPUs. In Section VI-B, we show that significant speedups can be achieved by carefully considering these factors.

We first discuss memory access issues which are usually the most common concerns for GPU implementations. Second, we discuss why the original distance computation algorithm is not a good candidate for GPU implementation. We present a more efficient dynamic programming algorithm that reduces the computational complexity of the distance computation from $O(NL^2)$ to $O(NL)$. Third, we discuss how insufficient hardware resources might limit us from using the full computation capabilities of the GPU and limit the size of problems we can handle. We present a technique that allows us to overcome this limitation for our GPU implementation.

A. Memory Considerations

We discuss two aspects of memory accesses in this section. First, the design of GPUs is specialized for high throughput rather than low latency and so, global memory access are often not able to sustain the GPU's computing throughput. Time spent on global memory accesses often becomes the bottleneck in GPU implementations. Second, because of the hardware design of GPUs, an inefficient memory access pattern may *sequentialize* memory accesses made by the parallel threads in a warp resulting in significantly higher execution times. Thus, it is crucial to avoid both these situations to achieve good performance.

Amortizing Global Memory Access via Shared Memory. In our baseline implementation, each thread loads the time series from global memory independently. Therefore, to load the values of a time series, each thread requires $O(L)$ global memory access. As mentioned in Section III, data accesses from global memory cost about 500 GPU cycles. Even with caching, the global memory access still slows down the program execution (see experiments in Section VI-B).

To amortize the global memory accesses, a common technique called tiling fetches the data and stores it in shared memory. Algorithm 3 lists the procedure for our application. To load a time series, each thread is responsible to load an element from global memory to shared memory.[4] Then, the threads are synchronized using a barrier synchronization function __syncthreads(). This reduces the number of global memory accesses in each thread to $O(1)$. Notice that we simplify the algorithm by assuming the number of threads is equal to the length of the time series. If the number of elements in a time series is too large, we can divide the time series into tiles. Then, all threads in a block collaborate to load a tile of the time series at a time. In some applications, if threads have only a small number of instructions between memory accesses, tiling might not be sufficient to amortize the global memory access. In such situations, one might consider applying pre-fetching technique. An example can be found in [15].

[4]Note that thread ID and array index start at 0 while the indices of elements in each time series start at 1.

Algorithm 3 A GPU implementation using shared memory.

```
1) Thread $s$ loads $T_{i,s+1}$ to shared memory Ti[s].
2) __syncthreads().
3) Perform calculation using array Ti.
4) For all time series $\boldsymbol{T}_j, j = 1 \dots L$
   a) Thread $s$ loads $T_{j,s+1}$ to shared memory Tj[s].
   b) __syncthreads().
   c) Perform calculation using array Tj.
```

Global Memory Coalescing. As mentioned in Section III, all threads in a warp execute the same instruction at the same time. Therefore, by design, if threads in a warp execute a load instruction to access data in aligned, consecutive global memory locations, the hardware can gather all these accesses to a consolidated access to the DRAMs and reduce the access overhead. Consider the case when a thread block loads a time series from global memory to shared memory. Assume that the values of time series are stored as double precision floating point numbers (8 Bytes), and starting memory address of the time series is M. If M is a multiple of 128 Bytes, and thread i accesses memory location $M + i \times 8$, all the accesses will be coalesced. Such access pattern allows the DRAMs to deliver data at a high rate, and reduces the time to access memory. In our application, all elements of time series are stored in consecutive locations and they are accessed by consecutive threads. Therefore, the global memory accesses are coalesced.

Bank Conflicts. In many GPU devices, shared memory is divided into memory banks and threads can access only one address from each bank at a time. Therefore, if threads in a warp try to access memory in the same bank, it results in a *bank conflict* and the accesses are executed *sequentially*. In general, bank conflicts can be avoided by careful data layout in shared memory (padding extra bytes is a common strategy).

B. An Efficient Algorithm for Distance Computation

As mentioned in Section II, *Mueen et al.* [3] proposed an algorithm to reduce the total computational time of distance computation from $O(N^2L^4)$ to $O(N^2L^3)$ by pre-computing and storing temporary information. However, the algorithm requires $O(N^2L)$ space to store the pre-computed information if the pivots are evaluated simultaneously. A typical problem that we consider has a value of around 1,000 for N and L. The size of memory required to store the temporary information in this case is beyond the memory capacity of typical GPUs. Moreover, data access from global memory is slow. Therefore, pre-computing and storing statistics of time series in global memory is not desirable. In the following, we propose a dynamic programming algorithm to reduce the computational complexity of each thread from $O(NL^2)$ to $O(NL)$ by using shared memory. Then, the overall computational complexity becomes $O(N^2L^3)$ as there are $O(NL^2)$ threads in total. The complexity of our GPU implementation is thus the same as that of the algorithm in [3].

We follow [3] to compute $dist(\boldsymbol{T}_{i,s,l}, \boldsymbol{T}_{j,u,l})$ via Eq. (II). If we directly evaluate $\Omega_{i,j,l,s,u}$, the time complexity of Eq. (5) is $O(L)$. However, Eq. (5) has the following recursive relationship:

$$\begin{aligned}\Omega_{i,j,l,s+1,u+1} &= \sum_{t=0}^{l-1} T_{i,s+t+1}T_{j,u+t+1} \\ &= \sum_{t=1}^{l} T_{i,s+t}T_{j,u+t} \\ &= \Omega_{i,j,l,s,u} + T_{i,s+l}T_{j,u+l} - T_{i,s}T_{j,u}.\end{aligned} \tag{6}$$

Therefore, computing $\Omega_{i,j,l,s+1,u+1}$ from $\Omega_{i,j,l,s,u}$ can be done in constant time. For a fixed i, j, l, the recursion requires the initial values $\Omega_{i,j,l,1,u}, \forall u$ and $\Omega_{i,j,l,s,1}, \forall s$. They can be pre-computed parallelly by Eq. (5) using all the available threads. A similar relationship can be found to compute $\mu_{j,u,l}$ and $\sigma_{j,u,l}$:

$$\begin{aligned}&\mu_{j,u,l} = S_{j,u,l}/l, \quad \sigma_{j,u,l} = \sqrt{S2_{j,u,l} - \mu_{j,u,l}^2}, \\ &S_{j,u,l} = \sum_{t=0}^{l-1} T_{j,u+t} = S_{j,u-1,l} + T_{j,u+l-1} - T_{j,u-1}, \\ &S2_{j,u,l} = \sum_{t=0}^{l-1} T_{j,u+t}^2 = S2_{j,u-1,l} + T_{j,u+l-1}^2 - T_{j,u-1}^2.\end{aligned} \tag{7}$$

Algorithm 4 lists the details. For simplicity, we omit the __syncthreads() operation in the presentation. In our algorithm design, thread s in the thread block (i, l) deals with the pivot $T_{i,s+1,l}$. At the outer loop j (Step 2), Thread s computes the initial value $\Omega_{i,j,l,s+1,1}$ and $\Omega_{i,j,l,1,u+1}$ using (5) and stores the result in shared memory buf[s] and buf_s0[s], respectively. Then, in iterations of u (Step 2e), Thread 0 loads $\Omega_{i,j,l,1,u}$ from buf_s0[u], while Thread $s, s > 0$ fetches $\Omega_{i,j,l,s,u-1}$ from the buffer, and computes $\Omega_{i,j,l,s+1,u}$ by Eq. (6). The time complexity of all the operations in the inner loop (Step 2e) is $O(1)$, therefore, the total complexity is $O(NL)$. Section VI-B presents the speedup seen in distance computation by using this efficient algorithm.

Algorithm 4 An efficient algorithm for computing $dist(\boldsymbol{T}_{i,s+1,l}, \boldsymbol{T}_j), \forall j$ by Thread s on Block (i, l).

```
1) Compute $S_{i,s+1,l}, S2_{i,s+1,l}$ of $\boldsymbol{T}_i$.
2) For all time series $\boldsymbol{T}_j, j = 1 \ldots N$
   a) Compute $S_{j,1,l}, S2_{j,1,l}$ of $\boldsymbol{T}_j$.
   b) Compute $\Omega_{i,j,l,s+1,1}$ and store the result in buf[s].
   c) Compute $\Omega_{i,j,l,1,s+1}$ and store the result in buf_s0[s].
   d) Compute $dist(\boldsymbol{T}_{i,s+1,l}, \boldsymbol{T}_{j,1,l})$ by $S_{i,s+1,l}$, $S2_{i,s+1,l}$, $S_{j,1,l}$, $S2_{j,1,l}$, $\Omega_{i,j,l,1,s+1}$.
   e) For every starting position $u = 2 \ldots L$ on $\boldsymbol{T}_j$
      i) Compute $S_{j,u,l}$ using $S_{j,u-1,l}$ by (7).
      ii) Compute $S2_{j,u,l}$ using $S2_{j,u-1,l}$ by (7).
      iii) If $s$ = 0: load $\Omega_{i,j,l,1,u}$ using buf_s0[u],
      iv) Else: compute $\Omega_{i,j,l,u,s+1}$ using buf[s-1] by (6) and store result in buf[s].
      v) Compute $dist(\boldsymbol{T}_{i,s+1,l}, \boldsymbol{T}_{j,u,l})$ by $S_{i,s+1,l}$, $S2_{i,s+1,l}$, $S_{j,u,l}$, $S2_{j,u,l}$, $\Omega_{i,j,l,u,s+1}$.
   f) $dist(\boldsymbol{T}_{i,s+1,l}, \boldsymbol{T}_j) \leftarrow \min_u dist(\boldsymbol{T}_{i,s+1,l}, \boldsymbol{T}_{j,u,l})$.
```

C. Memory Shortage Problem When Dealing with Large Data Sets

In our implementation, each thread computes the information gain corresponding to a pivot. This computation requires sorting the time series according to their distances from the pivot. Therefore, each thread needs to compute and to store the distances in an array. As each block has $O(L)$ threads, to store all the distances takes $O(NL)$ memory. This exceeds the capacity of shared memory on GPUs for many problems of interest. Therefore, we store these distance in global memory. But, if we assume all thread blocks execute at the same time, the global memory is also not sufficient as there are $O(NL)$ thread blocks and $O(N^2L^2)$ space in total will be required. To avoid this memory shortage problem, one possible solution is to split the thread blocks into several grids and to execute grids in sequence. However, this solution induces extra penalty as the end time of a grid is determined by the end time of the last thread block, and a grid can execute only if the previous grids finish execution. In the rest of this section, we describe a solution that launches the kernel only once.

Although the CUDA programming model assumes that all thread blocks execute in parallel, the number of current thread blocks that actually execute is limited by hardware resources. For example, in GTX 480, there are only 15 SMs, and up to 8 thread blocks can be assigned to each SM. Therefore, there are at most 90 thread blocks active in the SMs and they are executed simultaneously. Inspired by the fact, we assume that there are at most $|B|$ thread blocks active in SMs at a time. Then, we equally divide the original thread blocks to $|B|$ groups, such that each group contains $O(NL/|B|)$ blocks. Then, we only need to generate $|B|$ thread blocks, and each of them sequentially executes one group of the original blocks. Since each thread block can release the distance values after it completes execution, the total memory usage is reduced to $|B|NL$. The parameter $|B|$ is specified by user according to the resource usages and the device parameters. In our experiments, the performance is insensitive to the choice of $|B|$ to a reasonable extent.

Data	N	C	L	CPU		GPU		
				from [3]	ours	BL	+SM	+All
Cricket	9	2	308	44.7	11.4	2.7	1.1	0.4
*BirdSong	10	2	995	40.7	11.5	0.0	0.0	0.0
Diatom	16	4	345	201.2	52.2	16.2	7.0	1.6
Motes	20	2	84	3.5	1.0	0.1	0.1	0.0
Sony	20	2	70	1.9	0.5	0.1	0.0	0.0
ECGFiveD	23	2	136	22.5	5.9	1.0	0.5	0.2
FaceFour	24	4	350	474.2	125.0	38.0	16.4	3.8
Symbols	25	6	398	768.5	202.2	64.0	34.7	8.3
Coffee	28	2	286	346.0	90.6	19.6	11.1	3.0
OliveOil	30	4	570	3,308.9	867.6	308.8	168.5	31.2
CBF	30	3	128	100.3	24.1	3.3	1.8	0.4
Arrowhead	36	3	100	1,444.5	391.5	62.7	48.6	17.9
Beef	30	5	470	1,836.9	471.7	160.8	85.9	18.1
Gun_Point	50	2	150	145.4	38.6	6.0	3.5	1.2
Car	60	4	577	13,769.2	3,622.0	1,392.5	702.7	127.7
*Lighting2	60	2	637	1,870.8	490.7	188.2	99.6	17.0
ItalyPower Demand	67	2	24	0.4	0.1	0.0	0.0	0.0
Haptics	69	2	364	4,418.3	1,179.4	333.6	187.4	48.8
Trace	100	4	275	3,920.6	1,060.2	226.8	123.3	34.2
ECG200	100	2	96	209.7	62.9	7.5	3.2	1.6
Plane	105	7	144	566.1	152.9	24.1	13.4	4.9
*OSULeaf	200	6	427	6,366.2	1,711.0	526.1	282.7	65.9
*Chlorine	467	3	166	1,970.1	553.7	87.6	49.5	17.2
*Wafer	1,000	2	152	6,791.7	1,900.5	262.9	150.6	54.6
*TwoPatterns	1,000	4	128	4,028.2	1,160.9	136.7	89.8	33.1

Table II: Data statistics (N is number of time series, C is number of classes and L is the length of the time series) and running time of distance calculation. Time is in seconds. Running time for CPUs is provided both for the original implementation and our optimized version. The running time of the GPU implementation is provided for the baseline, the baseline with shared memory usage and the baseline with shared memory usage and a more efficient algorithm. For some data sets, CPU implementations are not fast enough to consider all possible pivots. Therefore, [3] restricts the search space to the pivots with length that are multiples of 10. We follow the setting in [3] and mark such data sets with an asterisk symbol (*). More discussion about the search space can be found in Section VI-C.

VI. Experiments

In this section we present the results of our experiments to analyze the performance of our GPU implementation. We first compare the performance of the CPU and GPU implementations of distance calculation and show the improvements in the GPU implementation by considering the various factors described in Section V. We then present the overall speedup of shapelet discovery algorithm and show that we achieve orders of magnitude speedup over the CPU implementation.

A. Experimental Settings

We implemented algorithms based on the source code from [3].[5] The experiments are conducted on a 64-bit Linux machine with Intel Xeon X5650 CPU @ 2.67GHz and NVDIA GTX480 graphics card. The CPU code is implemented in C/C++ and the GPU code is implemented in CUDA C/C++. All implementations use double precision floating point.

We generate results for all data sets used in [3] with the same parameters reported in their supplementary materials. The datasets are available at [16]. Table II lists the statistics (number of time series, their lengths and the number of classes) of the time series data sets we used.[6] To check the correctness of our implementation, we checked the distance between each pivot and each time series and confirmed that in most cases, the results of our GPU implementation are consistent with that of CPU implementation. We use a similar method to check the information gain corresponding to each pivot computed on the GPU. In almost all the cases, the GPU implementation discovers the same shapelets as the CPU implementation, although in some rare situations they find different shapelets because of numerical inaccuracy.

We also noticed that the CPU implementation from [3] is not optimized. It calls functions to compute means and standard deviations of time series and pivots in an inner loop. Because this calculation is done in constant time by using the pre-computing technique, the overhead of the function calls become a bottleneck for the program. By replacing the function call with in-line calculations, the running time is significantly reduced. Table II shows the results for both the original CPU implementation from [3] and our optimized CPU implementation. We confirm that, in our machine, the running time of the CPU implementation from [3] is similar to or less than that reported in [3][7].

B. Speedup of the Distance Computation

In this section, we investigate the time to do the distance computation. We compare the following implementations:

- The CPU implementation from [3].
- An optimized CPU implementation (see Section VI-A).
- GPU (BL): the baseline implementation (described in Section IV).
- GPU (+SM): the GPU implementation using shared memory (described in Section V-A).
- GPU (+All): the GPU implementation using shared memory and a dynamic programming algorithm (described in Section V-B).

Table II shows the running time of distance calculation for the shapelet discovery algorithm. For GPU implementations, the running time includes the time to transfer data between host memory and device memory and the time to launch

[5]Supporting materials of [3] including code and data is at http://www.cs.ucr.edu/~mueen/LogicalShapelet/

[6]The time series objects in BirdSong and Arrowhead data sets have different lengths. To simplify, we pad the short time series with zeros such that all the time series are of the same length.

[7]The only exception is Car data set. The implementation from [3] takes 17,024 seconds (see Figure 4). However, the overall running time reported in [3] is 1,940 seconds. Based on the size of the data, we suspect there is an error in reporting the parameters. Nevertheless, we use the provided parameters in our comparisons.

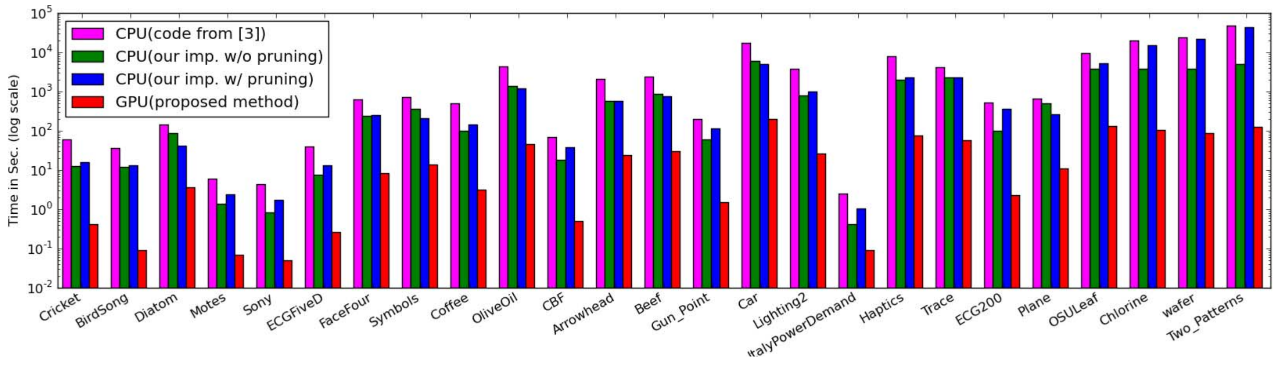

Figure 4: Comparison of total running time of various CPU and GPU implementations. **Time is in log scale.** Results show that the GPU implementation is faster than CPU implementations by 1-2 orders of magnitude.

the kernel. We find that our optimized CPU implementation (with function call overheads removed) is about four times faster than the original implementation from [3]. Although the naive baseline implementation on GPUs has a high computational complexity, we find that it significantly outperforms the optimized CPU implementation. This shows the benefits of the GPU's massively parallel computing power. Using shared memory to amortize global memory access improves the performance of the GPU baseline implementation by about two times on average. This shows the importance of carefully considering memory usage factors in a GPU implementation. Applying the dynamic programming algorithm described in Section V-B, GPU (+All) further improves the performance of the GPU implementation by 3.75x on an average. The average speedup of GPU (+All) over the CPU implementation from [3] is 118x and over the optimized CPU implementation is 32x. The best speedup of GPU (+All) over the CPU implementation from [3] is 271x and over the optimized CPU implementation is 65x. These results are consistent with the results seen in implementing other computational intensive algorithms on GPUs. Usually, a well-designed GPU implementation is found to be faster than a CPU implementation by 1 or 2 orders of magnitude.

C. Overall Speedups

We compare the overall running time of the shapelet discovery algorithm for the following implementations:

- The CPU implementation from [3].
- The optimized CPU implementation (see Section VI-A) without pruning technique.
- The optimized CPU implementation with pruning technique (discussed below).
- The proposed GPU implementation.

Mueen et al. [3] proposed a pruning technique, in which they find an upper bound on the information gain corresponding to a pivot even before all the distance calculations for the pivot are completed. If the upper bound is less than the current best information gain, the pivot is discarded. The running time is significantly reduced if the time to calculate the upper bounds is much lesser than the time to calculate distances. However, as we speedup the distance calculation on the CPU, it is not clear whether pruning is still useful. Therefore, we report the results of our CPU implementation both with and without pruning. We do not compare with the implementation from [2], as [3] reports that their algorithm is significantly faster than the former.

Figure 4 shows the overall running time of each implementation. For a fair comparison, we report the execution time of the complete application including the time spent on the CPU. Results show that our GPU implementation is significantly faster than all the CPU implementations. It achieves a maximum 460x speedup over the CPU implementation from [3] and 149x speedup over the optimized CPU implementation without pruning. On average, the GPU implementation is faster than the optimized CPU implementation by 35 times. Notice that the computational complexities of the above implementations are the same. The speedup of our GPU implementation comes from the massive parallelism and from a careful design that fully utilizes the resources on the GPU.

For most data sets, especially the larger ones, pruning technique does not work well. This is because of the fact that after the running time of distance calculation is reduced, the overhead to compute the bound becomes significant. For example, in ECG200 data set, it costs 274 seconds to compute the bound for pruning. However, the total running time of our optimized CPU implementation without pruning is only 98 seconds. Therefore, we cannot take advantage of pruning the search space.

The performance advantage of our GPU implementation allows us to deal with larger data sets or to perform a more extensive search for a shapelet. For example, on one large data set, OSULeaf, the original CPU code requires about 4.5 hours to search the shapelet from pivots with length that are multiples of 10. The running time of our optimized CPU implementation is reduced to 1.7 hours, while our GPU implementation takes only 2 minutes. This allows us to further consider all pivots with all possible lengths and enlarge the search space by ten times. Our GPU implementation takes only about 20.2 minutes to perform this task. This allows us to look for a better shapelet than previous implementations and thereby improve classification accuracy from 69.01% to 72.31%. However, we note that considering more pivots does not always guarantee better performance. On Wafer, considering all possible pivots obtains the same test accuracy (99.89%) as searching over a subset of pivots with length that are multiples of 10.

VII. Discussion and Conclusion

In summary, in this work we presented a GPU implementation of a shapelet discovery algorithm for time series classification. We also discussed several hardware and software perspectives that affect the performance applications implemented on GPUs and show the performance improvements that can be achieved by carefully considering these factors. From the hardware point of view, we show the importance of using the appropriate kind of memory from the several types of memories found in GPUs. From the software aspect, we redesigned the shapelet discovery algorithm to make it suitable for parallel implementation on GPUs. Overall, our implementation is faster than the CPU implementation from [3] by several orders of magnitude.

It should be noted that the proposed algorithms and implementation considerations have potential to be applied in other applications that use pattern based features. Pattern based classification models generate features to indicate the existence of a specific pattern in an object, and has been attracting great interest because of their robustness and interpretability ([17], [18], [19], [20]). However, the mining time to discover a pattern usually grows with the number of training objects and often restricts the search space for these patterns. We believe that the computing power of the massively parallel GPUs can be a key to solve this scalability problem.

Acknowledgments. The authors would like to thank Prof. Rakesh Kumar for his valuable feedback and the authors of [3], [16] for sharing code and data. This research is supported by the Defense Advanced Research Projects Agency (DARPA) Machine Reading Program under Air Force Research Laboratory (AFRL) prime contract no. FA8750-09-C-0181. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the view of the DARPA, AFRL, or the US government.

References

[1] Z. Xing, J. Pei, and E. Keogh, "A brief survey on sequence classification," *SIGKDD Explorations*, vol. 12, pp. 40–48, 2010.

[2] L. Ye and E. Keogh, "Time Series Shapelets: A New Primitive for Data Mining," in *KDD*, 2009.

[3] A. Mueen, E. Keogh, and N. Young, "Logical-Shapelets: An Expressive Primitive for Time Series Classification," in *KDD*, 2011, pp. 1154–1162.

[4] W.-M. W. Hwu, Ed., *GPU Computing Gems*, Emerald & Jade ed. Morgan Kaufmann, 2011.

[5] B. Catanzaro, N. Sundaram, and K. Keutzer, "Fast support vector machine training and classification on graphics processors," in *ICML*. ACM Press, 2008, pp. 104–111.

[6] A. Cotter, N. Srebro, and J. Keshet, "A GPU-tailored approach for training kernelized SVMs," in *KDD*, 2011.

[7] R. Raina, A. Madhavan, and A. Ng, "Large-scale deep unsupervised learning using graphics processors," in *ICML*. ACM, 2009, pp. 873–880.

[8] F. Yan, N. Xu, and Y. A. Qi, "Parallel inference for latent dirichlet allocation on graphics processing units," in *NIPS*, 2009.

[9] D. Sart, A. Mueen, W. Najjar, V. Niennattrakul, and E. Keogh, "Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs," in *ICDM*, 2010.

[10] E. J. Keogh and M. J. Pazzani, "Scaling up dynamic time warping for datamining applications," in *KDD*, 2000, pp. 285–289.

[11] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," *PVLDB*, vol. 1, pp. 1542–1552, 2008. [Online]. Available: http://dx.doi.org/10.1145/1454159.1454226

[12] G. E. A. P. A. Batista, X. Wang, and E. J. Keogh, "A complexity-invariant distance measure for time series," in *SDM*, 2011.

[13] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, 1984.

[14] NVIDIA.

[15] G. Wang, X. Yang, Y. Zhang, T. Tang, and X. Fan, "Program Optimization of Stencil Based Application on the GPU-Accelerated System," in *ISPA*, 2009, pp. 219–225.

[16] E. Keogh, Q. Zhu, B. Hu, H. Y., X. Xi, L. Wei, and C. A. Ratanamahatana, "The UCR time series classification/clustering," http://www.cs.ucr.edu/~eamonn/time_series_data/.

[17] H. Cheng, X. Yan, J. Han, and C.-W. Hsu, "Discriminative frequent pattern analysis for effective classification," 2007.

[18] H. Cheng, X. Yan, J. Han, and P. S. Yu, "Direct discriminative pattern mining for effective classification," in *ICDE*, 2008.

[19] H. Saigo, N. Krämer, and K. Tsuda, "Partial least squares regression for graph mining," in *KDD*, 2008.

[20] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection: A discriminative pattern mining approach," in *KDD*, 2009.