Title

Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins

ISBN

9781509054725

Authors

Zhu, Yan
Zimmerman, Zachary
Senobari, Nader Shakibay
et al.

Publication Date

2016-12-01

DOI

10.1109/icdm.2016.0085

Peer reviewed

Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins

Yan Zhu[1], Zachary Zimmerman[1], Nader Shakibay Senobari[2], Chin-Chia Michael Yeh[1], Gareth Funning[2], Abdullah Mueen[3], Philip Brisk[1] and Eamonn Keogh[1]

[1]Department of Computer Science and Engineering, University of California, Riverside
{yzhu015, zzimm001, myeh003}@ucr.edu, {philip, eamonn}@cs.ucr.edu
[2]Department of Earth Sciences, University of California, Riverside, {nshak006, gareth}@ucr.edu
[3]Department of Computer Science, University of New Mexico, mueen@cs.unm.edu

***Abstract*—Time series motifs have been in the literature for about fifteen years, but have only recently begun to receive significant attention in the research community. This is perhaps due to the growing realization that they implicitly offer solutions to a host of time series problems, including rule discovery, anomaly detection, density estimation, semantic segmentation, etc. Recent work has improved the scalability to the point where exact motifs can be computed on datasets with up to a million data points in tenable time. However, in some domains, for example seismology, there is an insatiable need to address even larger datasets. In this work we show that a combination of a novel algorithm and a high-performance GPU allows us to significantly improve the scalability of motif discovery. We demonstrate the scalability of our ideas by finding the full set of exact motifs on a dataset with one hundred million subsequences, by far the largest dataset ever mined for time series motifs. Furthermore, we demonstrate that our algorithm can produce actionable insights in seismology and other domains.**

Keywords—Time series; joins; motifs; GPUs.

I. Introduction

Time series motifs are approximately repeated subsequences found within a longer time series. While time series motifs have been in the literature for fifteen years [5], they recently have begun to receive significant attention *beyond* the data mining community. Recent years have seen them applied to problems as diverse as understanding the network of genes affecting the locomotion of the *C. elegans* nematode [4] and cataloging speech pathologies in humans [3].

While significant progress has been made in how we define, score, rank and visualize motifs, actually *discovering* them in large datasets remains a bottleneck. To date, we are not aware of any attempts to mine any dataset larger than one million data points [11]. In this work we show how we can significantly improve the scalability of exact motif discovery both by leveraging GPU hardware, and by modifying the recently introduced STAMP algorithm [23]. The STAMP algorithm computes time series subsequence *joins* with an efficient *anytime* algorithm [23]. Our key observations here are:

- The solution to the full exact 1NN *time series join* can be converted to the exact solution for any definition *of time series motif* [14], with only trivial extra effort.
- The *anytime* property of STAMP may be useful to some users, but as we explain below, in our motivating domain of seismology it is not required or helpful. As we will show, if we are willing to forego this property, we can compute motifs at least an order of magnitude faster than STAMP can. Moreover, forgoing the anytime property also makes it much easier to leverage GPU hardware.

In keeping with the STAMP name [23], we call our faster algorithm STOMP, Scalable Time series Ordered-search Matrix Profile, and its GPU-accelerated version GPU-STOMP.

In this work we show that GPU-STOMP allows us to significantly push the scalability envelope. We demonstrate the scalability of our ideas by extracting motifs from a dataset with one hundred million objects. Such a computation requires computing (or *admissibly* pruning) 499,999,999,500,000,000 pairwise Euclidean distances calculations. If each Euclidean distance calculation took 0.0000001 seconds, a brute force algorithm would require 1,585 years[1]. As we will show, we can compute this join in just twelve days. We recognize the twelve days seems like significant computational time, but consider that this data represents 58 days of continuous seismology recorded at 20Hz. Thus even at this massive scale we are much faster than real time. Fig. 1 previews a pair of repeating earthquake sequences (essentially, a *time series motif* [5]) discovered by our algorithm in a seismologic dataset.

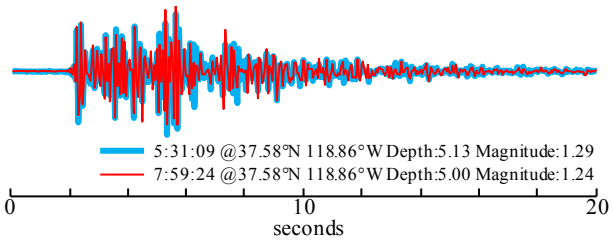

Fig. 1. A pair of repeating earthquake sequences (motifs) we discovered from seismic data recorded at a station near Mammoth Lakes on February 17th, 2016. One occurrence (fine/red) is overlaid on top of another occurrence (bold/blue) that happened hours earlier. (best viewed in color).

Here the two occurrences are very similar in spite of happening 148 minutes apart; however, the geophysics of earthquakes means that in principle we could see such similar

[1] 499,999,999,500,000,000 * 0.0000001 seconds = 1585.49 years

events millennia apart. Naturally we are limited to the few decades humans have been recording such data (see Fig. 7).

The rest of this paper is organized as follows. In Section II we introduce background and related work in data mining and (briefly) seismology. In Section III we introduce the necessary definitions and notations, which allow us to introduce our algorithms in Section IV. We conduct an extensive empirical comparison in Section V, including comparisons to the most obvious rival methods. Finally, in Section VI we offer conclusions and directions for related work.

II. Background and Related Work

A. Motif Discovery Background

Motif discovery for time series was introduced in 2003 [5] (although the classic paper of Agrawal, Faloutsos and Swami foreshadows motifs by computing all-pair similarity for time series [1]); since then, it has created a flurry of research activity. One major direction has been to apply motifs to solving problems in domains as diverse as bioinformatics [4], speech processing [3], robotics, neurology and entomology [14]. The other major research focus has been extensions and generalizations of the original work, especially attempts to improve scalability [11][14]. These attempts at improving the scalability of motif discovery fall into two broad classes; *approximate* and *exact* motif discovery [11][14].

Clearly approximate motifs can be faster to compute, and may be useful in some domains. However, there are domains in which the risk of false negatives is simply unacceptable. Consider our motivating domain of seismology [24]. This is a domain in which false negatives could affect public policy, change insurance rates for customers, and conceivably cost lives by allowing a dangerous site to be developed for dwellings. Given that the task at hand is to find *exact* motifs, all known methods based on *hashing* [24] and/or *data discretization* [5] can be dismissed from consideration.

Virtually every time series data mining technique has been applied to the motif discovery problem, including indexing [11][21], data discretization [5], triangular-inequality pruning [14], hashing [24], early abandoning etc. However, all these techniques rely on the assumption that the *intrinsic* dimensionality of time series is much lower than the *recorded* dimensionality [22]. This is generally true for data such as heartbeats and gestures, etc.; however, it is not true for seismograph data, which is intrinsically high dimensional. To see this, we performed a simple experiment.

We measured the *Tightness of Lower Bounds* (TLB) for three types of data, using the two most commonly used dimensionality reduction representations for time series, DFT and PAA (In addition, PAA is essentially equivalent to the Haar wavelets for this purpose [22]). The TLB is defined as:

$$TLB = LowerBoundDist(A,B) / TrueEuclideanDist(A,B)$$

It is well understood that the TLB is near perfectly (inversely) correlated with wall-clock time, CPU operations, number of disk access or any other performance metric for similarity search, all-pair-joins, motifs discovery, etc. [22].

Fig. 2 shows unambiguous results. There is *some* hope that we could avail current speed-up techniques when considering (relatively smooth and simple) ECGs, *little* hope that the noisy and more complex human activity would yield to such optimizations, and *no* hope that anything currently in the literature will help with seismological data. This claim is further borne out in our detailed experiments in Section V.

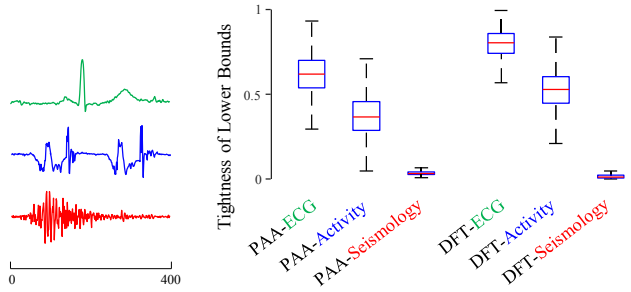

Fig. 2. *left*) Samples from three datasets, ECG, Human Activity and Seismology (available in [17]). *right*) The tightness of lower bounds, averaged over 10,000 random pairs, using PAA and DFT.

Even if we ignore this apparent death-knell for indexing/spatial access techniques, we could still dismiss them for other reasons, including memory considerations. As we shall see in Section IV, a critical property of our algorithm is that unlike all indexing/spatial access methods it does not need to explicitly *extract* the subsequences. For example, consider a time series of length 100 million, with eight bytes per value, requiring 0.8 GB. Our algorithm requires an overhead of seven other vectors of the same size (including the output), for an easily manageable total of 6.4 GB. However, any indexing algorithm that needs to extract the subsequences will increase memory requirements by *at least* O(d), where d is the reduced dimensionality used in the index [12]. Given that d may be 20 or greater, this means the memory requirements grow to at least 16 GB. With such a large memory footprint, we are almost certainly condemned to a random access disk-based algorithm, dashing any hope of any speedup.

A related advantage of our framework is that we can choose the subsequence length just prior to doing the motif discovery. In contrast, any index-based technique must commit to a subsequence length before *building* the index, perhaps hours/days before any actual searching could begin [20][22]. If such an index is built to support subsequences of say length 200, it cannot be used to join subsequences of length 190 or 205, etc. (See Section 1.2.3 of [20]). Thus if we change our mind about the length of patterns we are interested in, we are condemned to a costly rebuilding of the entire index.

In summary, while we obviously are unable to absolutely guarantee that there is no other scalable solution to our task-at-hand, we are very confident that there is no existing off-the-shelf technology that can be used or adapted to allow us to get within two orders of magnitude of the results we obtain on the largest datasets.

B. Seismological Background

While our algorithms are completely general and can be applied to any domain, seismological data is of particular

interest to us, due to its sheer scale and importance in human affairs.

In the early 1980s it was discovered that in telemetry of seismic data recorded by the same instrument from sources in given region there will be many similar seismograms [6]. Geller and Mueller [6] suggested that "*The physical basis of this clustering is that the earthquakes represent repeated stress release at the same asperity, or stress concentration, along the fault surface.*" These patterns are called "repeating earthquake sequence" in seismology, and exactly correspond to the more general term "*time series motifs*". Fig. 1 shows an example of a repeating earthquake sequence pair from seismic data.

A more recent paper notes that many fundamental problems in seismology can be solved by joining seismometer telemetry in search of these repeating earthquake sequence [24], including the discovery of foreshocks, aftershocks, triggered earthquakes, swarms, volcanic activity and induced seismicity. However, the paper further notes that an *exact* join with a query length of 200 on a data stream of length 604,781 requires 9.5 days. Their solution, a transformation of the data to allow LSH based techniques, does achieve significant speedup, but at the cost of false negatives and the need for careful parameter tuning. For example, [23] notes that they need to set the threshold to a very precise 0.818 to achieve good results. While we defer a full discussion of experimental results until Section V, the ideas introduced in this paper can reduce the quoted 9.5 days for exact motif discovery from a dataset of size 604,781, to less than one minute, without the need to tune any parameters and with a guarantee of no false negatives.

It is important to note that this kind of speed up really is game-changing in this domain. It allows seismologists to quickly identify or detect earthquakes that are identical or similar in location without the need for trilateration, and can also provide useful information on relative timing and relative location of such events [2][9][10].

Somewhat more controversially, some researchers have suggested that the slow slip on the fault accompanying non-volcanic tremors (a sequence of Low Frequency Earthquakes, many of which are repeated) may temporarily increase the probability of triggering a large earthquake. Therefore, detecting and locating these repeating LFEs allows more robust short-term earthquake forecasting [9].

Finally, we note that seismologists have been early adopters of GPU technology [13] and other high performance computing paradigms. However, their use of this technology has been limited to similarity search, not motif search.

III. Notation and Definitions

While we mostly follow the framework introduced in [23], for completeness we review all necessary definitions.

A. Definitions

We begin by defining the data type of interest, *time series*:

Definition 1: A *time series* T is a sequence of real-valued numbers t_i: $T = t_1, t_2, ..., t_n$ where n is the length of T.

We are interested not in *global*, but *local* properties of time series. A local region of time series is called a *subsequence*:

Definition 2: A *subsequence* $T_{i,m}$ of a time series T is a continuous subset of the values from T of length m starting from position i. Formally, $T_{i,m} = t_i, t_{i+1}, ..., t_{i+m-1}$, where $1 \leq i \leq n-m+1$.

We can take a subsequence and compute its distance to *all* subsequences in the same time series. We call this a *distance profile*:

Definition 3: A *distance profile* D_i of time series T is a vector of the Euclidean distances between a given query subsequence $T_{i,m}$ and each subsequence in time series T. Formally, $D_i = [d_{i,1}, d_{i,2}, ..., d_{i,n-m+1}]$, where $d_{i,j}$ ($1 \leq i, j \leq n-m+1$) is the distance between $T_{i,m}$ and $T_{j,m}$.

We assume that the distance is measured by Euclidean distance between z-normalized subsequences [22].

We are interested in finding the nearest neighbors of all subsequences in T, as the closest pairs of this are the classic definition of time series motifs [5][14]. Note that by definition, the i^{th} location of distance profile D_i is zero, and very close to zero just before and after this location. Such matches are called *trivial matches* in the literature [14]. We avoid such matches by ignoring an "exclusion zone" of length $m/4$ before and after the location of the query. In practice, we simply set $d_{i,j}$ to infinity ($i-m/4 \leq j \leq i+m/4$) while evaluating D_i.

We use a vector called *matrix profile* to represent the distances between all subsequences and their nearest neighbors:

Definition 4: A *matrix profile* P of time series T is a vector of the Euclidean distances between each subsequence $T_{i,m}$ and its nearest neighbor (closest match) in time series T. Formally, $P = [\min(D_1), \min(D_2), ..., \min(D_{n-m+1})]$, where D_i ($1 \leq i \leq n-m+1$) is the distance profile D_i of time series T.

We call this vector matrix profile because one (naïve and space-inefficient) way to compute it would be to compute the full distance matrix of all pairs of subsequences in time series T, and then evaluate the minimum value of each column. Fig. 3 illustrates both a *distance profile* and a *matrix profile* created on the same raw time series T.

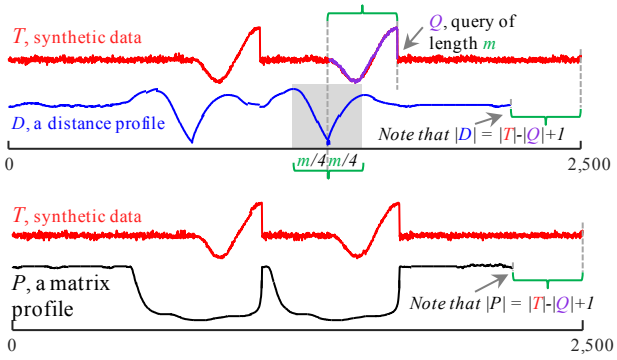

Fig. 3. *top*) One distance profile (Definition 3) created from a random subsequence Q of T. If we created distance profiles for all possible subsequences of T, the element-wise minimum of this set would be the matrix profile (Definition 4) shown at (*bottom*). Note that the two lowest values in P are at the location of the 1st motif [5][14].

One important fact to note is that the full distance matrix is symmetric: D_i is both the i^{th} row and the i^{th} column of the full distance matrix. Fig. 4 shows this more concretely.

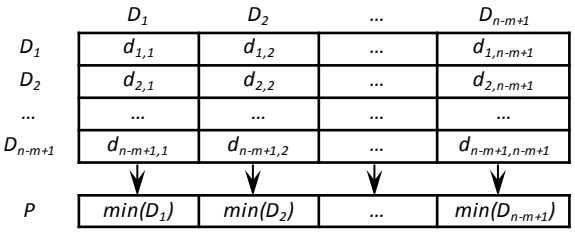

Fig. 4. An illustration of the relationship between the distance profile, the matrix profile and the full distance matrix. For clarity, we note we do *not* actually create the full distance matrix, as this would have untenable memory requirements.

The i^{th} element in the matrix profile P tells us the Euclidean distance from subsequence $T_{i,m}$ to its nearest neighbor in time series T. However, it does not tell us *where* that neighbor is located. This information is recorded in a companion data structure called the *matrix profile index*.

Definition 5: A *matrix profile index I* of time series T is a vector of integers: $I=[I_1, I_2, \ldots I_{n-m+1}]$, where $I_i=j$ if $d_{i,j} = \min(D_i)$.

By storing the neighboring information this way, we can efficiently retrieve the nearest neighbor of query $T_{i,m}$ by accessing the i^{th} element in the matrix profile index.

To briefly summarize this entire section: we can create two meta time series, the *matrix profile* and the *matrix profile index*, to annotate a time series T with the distance and location of all its subsequences' nearest neighbors within itself. As the reader may already have realized, the smallest pair of values in the *matrix profile* correspond to the best motif pair under the classic definition [11][14][5], and the corresponding values in the *matrix profile index* tell us where the motifs are located. Moreover, as both [23][14] argue, the *top-k* motifs, *range* motifs, and any other reasonable variant of motifs can trivially be computed given all the information in the *matrix profile*, the focus of the rest of this paper.

B. A Brief Review of the STAMP Algorithm

The recently-introduced STAMP algorithm can efficiently compute the *full* and *exact* matrix profile and matrix profile index of a given time series [23]. The STAMP algorithm essentially evaluates the distance profile D_i of query subsequence $T_{i,m}$ by exploiting FFT to calculate the dot product between $T_{i,m}$ and all subsequences of time series T. The overall time complexity of the algorithm is $O(n^2 \log n)$ and space complexity is $O(n)$, where n is the length of time series T. The STAMP algorithm can process a time series with up to a million data points in tenable time. However, to tackle the problems in our motivating domain seismology, there is an unquenchable need to process even larger datasets. It would take STAMP more than 20 years to analyze a seismology time series sampled at 20Hz for about 2 months, which is of length 100 million (see TABLE IV). In the next section, we will show a new and fast algorithm which, when built on top of a GPU, can finish processing the same time series in just 12 days.

IV. Algorithms

In this section we begin by showing that we can improve upon the STAMP algorithm [23] to create the much faster STOMP algorithm. We then further demonstrate that the architecture of STOMP lends itself to porting to GPUs.

A. The STOMP Algorithm

As we shall explain below, STOMP is similar to STAMP [23] in that it can be seen as highly optimized nested loop searches, with the repeated calculation of distance profiles as the inner loop. However, while STAMP must evaluate the distance profiles in random order (to allow its anytime behavior), STOMP performs an *ordered* search. It is by exploiting the locality of these searches, that we can reduce the time complexity by a factor of $O(\log n)$.

Before we explain the details of the algorithm, we first introduce a formula to calculate the z-normalized Euclidean distance $d_{i,j}$ of two time series subsequences $T_{i,\mathrm{m}}$ and $T_{j,m}$ using their dot product, $QT_{i,j}$:

$$d_{i,j} = \sqrt{2m\left(1 - \frac{QT_{i,j} - m\mu_i\mu_j}{m\sigma_i\sigma_j}\right)} \quad (1)$$

Here m is the subsequence length, μ_i is the mean of $T_{i,m}$, μ_j is the mean of $T_{j,\mathrm{m}}$, σ_i is the standard deviation of $T_{i,m}$, and σ_j is the standard deviation of $T_{j,\mathrm{m}}$.

The technique introduced in [20] allows us to obtain the means and standard deviations with $O(1)$ time complexity; thus, the time required to compute $d_{i,j}$ depends only on the time required to compute $QT_{i,j}$. Here we claim that $QT_{i,j}$ can also be computed in $O(1)$ time, once $QT_{i-1,j-1}$ is known.

Note that $QT_{i-1,j-1}$ can be decomposed as:

$$QT_{i-1,j-1} = \sum_{k=0}^{m-1} T_{i-1+k} T_{j-1+k} \quad (2)$$

and $QT_{i,j}$ can be decomposed as:

$$QT_{i,j} = \sum_{k=0}^{m-1} T_{i+k} T_{j+k} \quad (3)$$

Thus we have:

$$QT_{i,j} = QT_{i-1,j-1} - t_{i-1}t_{j-1} + t_{i+m-1}t_{j+m-1} \quad (4)$$

Our claim is thereby proved.

The relationship between $QT_{i,j}$ and $QT_{i-1,j-1}$ indicates that once we have the distance profile D_{i-1} of time series T with regard to $T_{i-1,\mathrm{m}}$, we can obtain the distance profile D_i with regard to $T_{i,\mathrm{m}}$ in just $O(n)$ time.

However, we cannot benefit from the relationship between $QT_{i,j}$ and $QT_{i-1,j-1}$ in the special case when $i=1$ or $j=1$. This problem is easy to solve: we can pre-compute the dot product values in these two special cases with FFT, as shown in

TABLE I. Concretely, we use *SlidingDotProduct*($T_{1,m}$, T) to calculate the first dot product vector $QT_1 = [QT_{1,1}, QT_{1,2}, \ldots, QT_{1,n-m+1}] = [QT_{1,1}, QT_{2,1}, \ldots, QT_{n-m+1,1}]$. The dot product vector is stored in memory and used as needed.

TABLE I. CALCULATE SLIDING DOT PRODUCT WITH FFT

	Procedure *SlidingDotProduct*(*Q, T*) Input: A query *Q, and* a user provided time series *T* Output: The dot product between *Q* and all subsequences in *T*
1	*n* ← *Length(T), m* ← *Length(Q)*
2	T_a ← *Append T with n zeros*
3	Q_r ← *Reverse(Q)*
4	Q_{ra} ← *Append* Q_r *with 2n-m zeros*
5	Q_{raf} ← *FFT*(Q_{ra}), T_{af} ← *FFT*(T_a)
6	*QT* ← *InverseFFT(ElementwiseMultiplication*(Q_{raf}, T_{af}))
7	*return QT[m:n]*

We are now in a position to introduce our STOMP algorithm in TABLE II.

TABLE II. STOMP ALGORITHM

	Procedure *STOMP*(*T, m*) Input: A time series *T* and a subsequence length *m* Output: Matrix profile *P* and the associated matrix profile index *I of T*
1	*n* ← *Length(T), l* ← *n-m+1*
2	*μ, σ* ← *ComputeMeanStd(T, m)* // *see* [20]
3	*QT* ← *SlidingDotProduct(T[1:m], T), QT_first* ← *QT*
4	*D* ← *CalculateDistanceProfile(QT, μ, σ)* // *see* (1)
5	*P*← *D, I*← *ones* // *initialization*
6	***for*** *i = 2* ***to*** *l* // *in-order evaluation*
7	***for*** *j= l* ***downto*** *2* // *update dot product, see* (4)
8	*QT[j]* ← *QT[j-1]-T[j-1]×T[i-1]+T[j+m-1]×T[i+m-1]*
9	***end for***
10	*QT[1]* ← *QT_first[i]*
11	*D* ← *CalculateDistanceProfile(QT, μ, σ, i)* // *see* (1)
12	*P, I* ← *ElementWiseMin(P, I, D, i)*
13	***end for***
14	***return*** *P, I*

The algorithm begins in line 1 by computing the matrix profile length *l*. In line 2 we precalculate the mean and standard deviation of every subsequence in *T*. Line 3 calculates the first dot product vector *QT* with the algorithm in TABLE I. In line 5 we initialize the matrix profile *P* and matrix profile index *I*. The loop in lines 6-13 calculates the distance profile of every subsequence of *T* in sequential order, with lines 7-9 updating *QT* according to (4). We update *QT*[*1*] in line 10 with the pre-computed *QT_first* in line 3. Line 11 calculates distance profile *D* according to (1). Finally, line 12 compares every element of *P* with *D*: if $D[j] < P[j]$, then $P[j] = D[j]$, $I[j] = i$.

The time complexity of STOMP is $O(n^2)$; thus, we have achieved a O(log*n*) factor speedup over STAMP [23]. Moreover, it is clear that $O(n^2)$ is optimal for any exact motif algorithm in the general case. The O(log*n*) speedup clearly makes little difference for small datasets, those with just a few tens of thousands of datapoints [5]. However, as we consider the datasets with millions of datapoints, this O(log*n*) factor begins to produce a very useful order-of-magnitude speedup.

To better understand the efficiency of STOMP, it is important to clarify the time complexity of the classic brute force algorithm is $O(n^2m)$. The value of *m* is domain dependent, but in Section V.F we consider real world problems where it is 2,000. Most techniques in the literature gain speedup by shaving a little off the n^2 factor; however, we gain speedup by reducing the *m* factor to O(*1*). Moreover, it is important to remember that the techniques in the literature can only reduce this n^2 factor if the data cooperates by having a low intrinsic dimensionality (recall Fig. 2), *and* the domain requires a short subsequence length. In contrast, the speedup for STOMP is completely independent of *both* the structure of the data and the subsequence length.

In spite of this dramatic improvement, it still takes STOMP approximately 5-6 hours to process a time series of length one million. Can we further reduce the time?

Note that the STOMP algorithm is extremely amenable to parallel computing frameworks. This is not a coincidence; the algorithm was conceived with a view to eventual hardware acceleration. Recall that the space requirement for the algorithm is only O(*n*); there is no data dependency in the main inner loop of the algorithm (lines 7-9 of TABLE II), so we can update all entries of *QT* in parallel. The evaluation of each entry in vectors *D*, *P* and *I* in lines 11 and 12 are also independent of each other. In the next section, we will introduce a GPU-based version of STOMP, taking advantage of these observations to further speed up the evaluation of matrix profile and thus motif discovery.

B. Porting STOMP to a GPU Framework

The Graphic Processor Unit, or GPU, is "*especially well-suited to address problems that can be expressed as data-parallel computations*" [15]. It has its own memory, and can launch multiple threads in parallel. Here we use the ubiquitous Single Instruction Multiple Data (SIMD) NVIDIA CUDA architecture, where we can assign multiple threads to process the same set of instructions on multiple data.

The threads on the GPU are managed in thread blocks. Threads in a thread block run simultaneously, and can cooperate with each other through shared local resources. A CUDA function is called a *kernel*. When we launch a kernel, we can specify the number of blocks, and the number of threads in each block to run on GPU. For example, the NVIDIA Tesla K80 allows launching at most 1024 threads within a block and as many as 2^{63} blocks (a total of 2^{73} threads), which is much more than enough to process a time series of length 100 million.

The GPU implementation of the STOMP algorithm in TABLE II can be decomposed into four steps:

- CPU copies the time series to GPU global memory.
- CPU launches GPU kernels to evaluate *μ*, *σ*, initial *QT*, *D*, *P* and *I*.
- CPU iteratively launches GPU kernels to update *QT*, *D*, *P* and *I*.
- CPU copies final output (*P* and *I*) from GPU.

In the first step, the CPU copies time series *T* (input vector of TABLE II) to the global memory of GPU. The time used to copy a time series of length 100 million takes less than a second. Note that in order to run the STOMP algorithm, we need to allocate space to store eight vectors in the GPU global memory: *T*, *μ*, *σ*, *QT*, *QT_first*, *D*, *P* and *I*. A double-precision

time series of length 100 million is approximately 0.8GB, so the algorithm consumes approximately 6.4GB global memory space. This is feasible for NVIDIA Tesla K40 and K80 cards; however, if the device used has less memory space available, we can simply split the time series into small chunks and evaluate one chunk at a time with the GPU.

In the second step, the CPU launches GPU kernels to evaluate the vectors in parallel. The mean and standard deviation vectors in line 2 of TABLE II can be efficiently evaluated by CUDA Thrust [15]. The first QT vector in line 3 can be evaluated in parallel by applying cuFFT, the NVIDIA CUDA Fast Fourier Transform [16] to the *SlidingDotProduct* function in TABLE I. We assign a total of n-m+1 threads to evaluate QT_first, D, P and I in lines 3-5 in parallel, with the j^{th} thread processing the j^{th} entry of these vectors one by one.

Now that we have initialized QT, D, P and I, we can start to update them iteratively. In the third step, the CPU runs the outer loop in lines 6-13 of TABLE II iteratively. In each iteration the CPU launches a GPU kernel with n-m+1 threads, parallelizing the evaluation of QT, D, P and I. As shown in Fig. 5, the first thread reads $QT[1]$ from the precomputed QT_first vector, while the second to the last threads evaluate their corresponding entry of QT using (4).

Note that in contrast to the CPU STOMP algorithm, which uses only one vector QT to store both QT_{i-1} and QT_i, here we need to use two vectors to separate them. The reason is that as *the* threads evaluate entries in QT in parallel, we need to avoid any entry to be written *before* it is read. A simple and efficient way to do this is to create two vectors, QT_odd and QT_even. When the outer loop variable i in line 6 is even, the threads read from QT_odd and write to QT_even; when i is odd, the threads read data from QT_even and write to QT_odd. After this, the threads evaluate D with (1), and the j^{th} thread updates P and I if $D[j] < P[j]$.

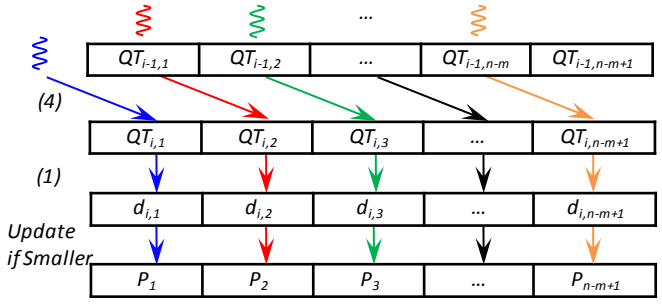

Fig. 5. Division of work among threads in the third step of GPU STOMP.

When all iterations complete, we have reached the last step of GPU STOMP, where the CPU copies P and I back to the system memory.

C. Further Parallelizing STOMP with multiple GPUs

The above parallelization scheme is suitable if we only have one single GPU device. Can we further reduce the processing time if there are two or more GPUs available?

Thus far, we have been using CPU to iteratively control the outer loop of the STOMP algorithm in TABLE II. We start by computing the first distance profile (the first row) in Fig. 4 and its corresponding QT vector. Then in each iteration we compute a new row of the distance matrix in Fig. 4, and maintain the minimum-so-far values of each column in vector P. When the iteration is completed, P becomes the exact matrix profile.

This outer loop computation can be further parallelized. Assume we have k independent GPU devices, and we also have $(n\text{-}m+1)/k = q$. We can then divide the distance matrix in Fig. 4 into k sections: device *1* evaluates the 1^{th} to the q^{th} rows, device *2* evaluates the $(q+1)^{th}$ to the $(2q)^{th}$ rows, etc. Essentially, device k uses the parallelized version of *SlidingDotProduct* function in TABLE I to calculate $QT_{q(k-1)+1}$ and $D_{q(k-1)+1}$, then evaluates the following q-*1* rows iteratively. The k devices can run in parallel, and when the evaluation completes, we can simply find the minimum among all the k matrix profile outputs. In short, we can achieve a k-times speed up by using k identical GPU devices.

By porting all the introduced techniques to NVIDIA Tesla K80, which contains two GPU devices on the same unit, we are able to obtain the matrix profile and matrix profile index of a seismology time series of length 100 million within 19 days. Are there any further optimizations left?

D. A Technique to Further Accelerate GPU STOMP

Fig. 5 showed the process to compute the i^{th} row of the distance matrix in Fig. 4 by n-m+*1* parallel threads. Recall that the distance matrix is symmetric; half of the distance computations can be saved if we instead only evaluate the i^{th} to the last columns. We show this strategy in Fig. 6.*top*.

However, note that we would like to maintain the $O(n)$ space complexity of our algorithm; if we simply move on to the $(i+1)^{th}$ row in Fig. 4 without further processing, then $P_i = min(d_{1,i}, d_{2,i}, \ldots, d_{i,i})$, and would not be updated anymore. To fix this, we need to launch another kernel after Fig. 6.*top* is completed. The new kernel is shown in Fig. 6.*bottom*.

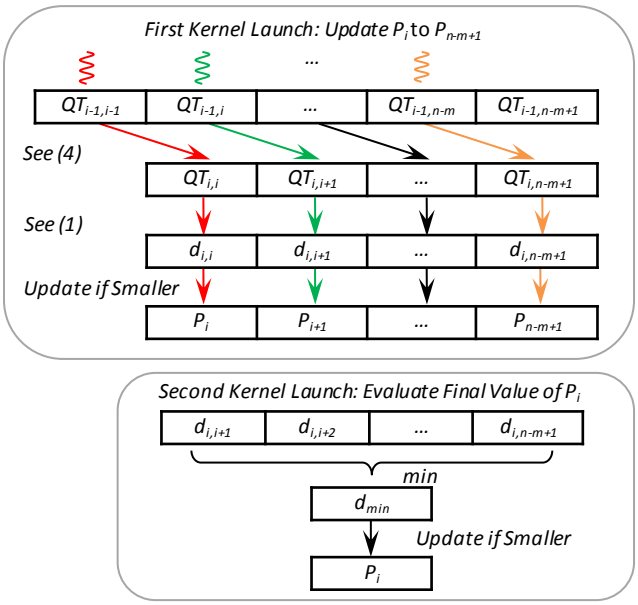

Fig. 6. Modifying the third step of GPU-STOMP. *top)* Launch only n-m-i+*2* threads (instead of the n-m+*1* threads in Fig. 5) this time at the i^{th} iteration. *bottom)* Launch another kernel to evaluate the final value of P_i.

Essentially, we have used an analogous reduction technique as in [7] to obtain $d_{min} = min(d_{i,i+1}, d_{i,i+2}, \ldots, d_{i,m+n-1})$, which also equals $min(d_{i+1,i}, d_{i+2,i}, \ldots, d_{n-m+1,i})$ because of symmetry. If $d_{min} < P_i$, we set $P_i = d_{min}$, so that $P_i = min(D_i)$. Though we need to launch one more kernel to process each row, which takes some extra time, the cost is much less than what we save here when the time series length is large. For example, the new technique reduced the time to process a time series of length 100 million from 19 days to about 12 days on NVIDIA Tesla K80.

Note that we are launching fewer and fewer threads in each iteration. To apply this new technique to multiple GPUs, we need to ensure that each GPU is loaded with similar amount of work, so that they will finish in similar time. Here, for NVIDIA Tesla K80, we computed the first $(n-m+1)(1-1/\sqrt{2})$ distance profiles with the first GPU, and the last $(n-m+1)/\sqrt{2}$ distance profiles with the second GPU.

V. EMPIRICAL EVALUATION

We have designed all our experiments such that they can be easily reproducible (although some require access to a GPU). To allow this, we have built a webpage [17] which contains all datasets and code used in this work. We begin by a careful comparison to STAMP, which is obviously the closest competitor; we consider more general rival methods later.

Unless otherwise noted, we used an Intel i7@4GHz PC with 4 cores to evaluate all the CPU-based algorithms; we used a server with two Intel Xeon E5-2620@2.4GHz cores and an NVIDIA Tesla K80 GPU to evaluate GPU-STOMP.

A. STAMP vs STOMP

We begin by demonstrating that STOMP is faster than STAMP, and that this difference grows as we consider increasingly large datasets. We further measure the gains made possible by using GPU-STOMP. In TABLE III we measure the performance of the three algorithms on increasingly long random walk time series with a fixed subsequence length 256.

TABLE III. TIME REQUIRED FOR MOTIF DISCOVERY WITH $m = 256$, VARYING n, FOR THE THREE ALGORITHMS UNDER CONSIDERATION

Algorithm \ Value of n	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
STAMP	15.1 min	1.17 hours	5.4 hours	24.4 hours	4.2 days
STOMP	4.21 min	0.3 hours	1.26 hours	5.22 hours	0.87 days
GPU-STOMP	10 sec	18 sec	46 sec	2.5 min	9.25 min

Note that we choose m's length as a power-of-two only to offer the best case for (the FFT-based) STAMP; our algorithm is agnostic to such issues.

A recent paper on finding motifs in (only) seismograph datasets also considers a dataset of about 2^{19} in length and reports taking 1.6 hours, about the same as STOMP [24]. However, their method is probabilistic and allows false negatives (twelve of which were actually observed, after checking against the results of a 9.5 day brute-force search [24]). Moreover, it requires careful tuning of several parameters, and does not lend itself to GPU implementation.

We wish to consider the scalability of even larger datasets with GPU-STOMP. However, in order to do so we must estimate the time for the two other algorithms. Fortunately, both the other algorithms allow an approximate prediction of the time needed, given the data length n. To obtain the estimated time, we evaluated only the first 100 distance profiles of both STAMP and STOMP, and multiplied the time used by $(n-m+1)/100$. In TABLE IV we consider much larger datasets, one of which reflects the data used in a case study in Section V.C.

TABLE IV. TIME REQUIRED FOR MOTIF DISCOVERY WITH VARIOUS m AND VARIOUS n, FOR THE THREE ALGORITHMS UNDER CONSIDERATION

Algorithm \ m | n	2000 | 17,279,800	400 | 100,000,000
STAMP (*estimated*)	*36.5 weeks*	*25.5 years*
STOMP (*estimated*)	*8.4 weeks*	*5.4 years*
GPU-STOMP (actual)	9.27 hours	12.13 days

Note that the 100-million-length dataset is one hundred times larger than the largest motif search in the literature [11].

All three algorithms under consideration have the very desirable property that the time required is independent of the subsequence length m. To see this, in TABLE V we measure the time required with n fixed to 2^{17}, for increasing m.

TABLE V. TIME REQUIRED FOR MOTIF DISCOVERY WITH $n = 2^{17}$, VARYING m, FOR THE THREE ALGORITHMS UNDER CONSIDERATION

Algorithm \ Value of m	64	128	256	512	1,024
STAMP	15.1 min	15.1 min	15.1 min	15.0 min	14.5 min
STOMP	4.23 min	4.33 min	4.21 min	4.23 min	2.92 min
GPU-STOMP	10 sec	10 sec	10 sec	10 sec	10 sec

Note that the time required for the longer subsequences is actually slightly shorter. This unintuitive fact is because the number of pairs that must be considered for a time series join [23] is $(n-m+1)^2$, so as m becomes larger, the number of comparisons becomes slightly smaller.

B. STOMP vs State-of-the-Art Motif Discovery Algorithms

Beyond independence of the subsequence length demonstrated in TABLE V, all three matrix profile-based algorithms also have the very desirable property that the time required is independent of *data* under consideration. To see this, we will compare to the recently introduced Quick-Motif framework [11], and the more widely known MK algorithm [14]. The Quick-Motif method was the first technique to do exact motif search on one million subsequences.

To level the playing field, we do not avail of GPU acceleration, but use the identical hardware (a PC with Intel i7-2600@3.40GHz) and programming language for all algorithms. Note that for a fair comparison with STAMP [23], which is written in MATLAB, in Section V.A we measured the performance of STOMP based on its MATLAB implementation. However, because the two rival methods in this section (Quick-Motif and MK) are written in C/C++, here we measure the runtime of (the CPU version of) STOMP based on its C++ implementation.

We use the original author's executables [18] to evaluate the runtime of both MK and Quick-Motif. The reader may wonder why the experiments here are less ambitious than in the previous sections. The reason is that beyond *time* considerations, the rival methods have severe *memory*

requirements. For example, for a seismology data with m = 200, $n = 2^{18}$, we measured the Quick-Motif memory footprint as large as 1.42 GB. In contrast, STOMP requires only 14MB memory for the same data, which is less than 1/100 of this. If this ratio linearly interpolates, Quick-Motif would need more than ½ terabyte of main memory to tackle the one-hundred-million benchmark, which is simply infeasible. Moreover, for Quick-Motif it is possible that a different dataset of the exact same size could require a larger or smaller footprint. In contrast, the space required for STOMP is independent of both the structure of data and the subsequence length.

This severe memory requirement makes it impossible to compare the STOMP algorithm with Quick-Motif on the seismology data, since Quick-Motif often crashed with an *out-of-memory* error as we varied the value of m. However, we noticed that the memory footprint for Quick-Motif tends to be much smaller with smooth data. Therefore, instead of comparing performance of the algorithms on seismology data, in TABLE VI we experimented on the much smoother ECG dataset (used in [20]), which is an ideal dataset for both MK and Quick-Motif to achieve their *best* performance.

TABLE VI. TIME REQUIRED FOR MOTIF DISCOVERY WITH $n = 2^{18}$, VARYING m, FOR VARIOUS ALGORITHMS

Algorithm \ m	512	1,024	2,048	4,096
STOMP	501s (14MB)	506s (14MB)	490s (14MB)	490s (14MB)
Quick-Motif	27s (65MB)	151s (90MB)	630s (295MB)	695s (101MB)
MK	2040s (1.1GB)	N/A (>2GB)	N/A (>2GB)	N/A (>2GB)

As we can see, both the runtime and memory requirement for STOMP are independent of the subsequence length. In contrast, Quick-Motif and MK both scale poorly in subsequence length in both runtime and (especially) memory usage. Note that the memory requirement of Quick-Motif is not monotonic in m, as reducing m from 4,096 to 2,048 requires three times as much memory. This is not a flaw in implementation (we used the author's own code) but a property of the algorithm itself.

In retrospect, this poor showing of both Quick-Motif and MK are unsurprising given the observations in Fig. 2. Both algorithms *can* be fast in ideal situations, with smooth data, short subsequence lengths, and "tight" motifs in the data. But both can, and *do*, require very large memory space and degenerate to brute-force search in less ideal situations. Moreover, as we will show in the next two sections, STOMP is actually computing much more useful information than the two rival methods.

C. Case Studies in Seismology: Infrequent Earthquake Case

To allow confirmation of the correctness and utility of STOMP, we begin by considering a dataset for which we know the answer from external sources. On April 30th, 1996, there was an earthquake of magnitude 2.12 in Sonoma County, California[2]. Then, on December 29th, 2009, about 13.6 years later, there was another earthquake with a similar magnitude. We concatenated the two full days in question to create a single time series of length 17,279,800 (see TABLE IV for timing results) and examined the top motifs with m = 2,000 (twenty seconds). As Fig. 7.*top* shows, the top motif here is *not* an earthquake, but an unusual sensor artifact [8].

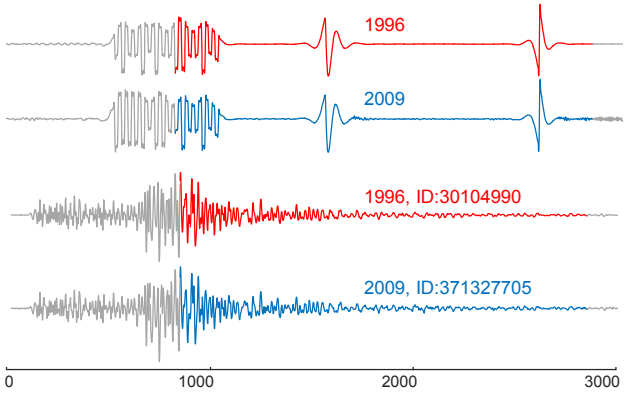

Fig. 7. Motifs (colored) shown in context (gray). *top*) The top motif discovered in the Sonoma County dataset is a sensor artifact, as are the next three motifs (not shown). *bottom*) The fifth motif is two true occurrences of an earthquake that happen 4,992 days apart.

There are a handful of other such artifacts, however, as shown in Fig. 7.*bottom*, the fifth best motif *is* the two occurrences of the earthquake. These misleading sensors artifacts are common, but could be easily filtered out in several ways [8]. For example, they have a zero crossing rate that is an order of magnitude lower than true earthquakes.

This example allows us to demonstrate yet another advantage of STOMP over rival methods. All the existing rival techniques can be expanded from top-1 motif discovery to top-k motif discovery; however, increasing k by even a modest amount will significantly degrade their speed.

Furthermore, consider again the example in Fig. 7. There is simply no way we could have known the "magic" value of $k = 5$ beforehand. If k was set to a large value to "be on the safe side", say k = 10, then all existing techniques would severely slow down because the best-so-far lower bound to prune unnecessary computations would be much looser. If we set k as a more conservative value, say k = 3, then we would miss the most valuable information in this seismology dataset. You might imagine that the rival methods could slowly increase from k to $k+1$ based on the user's lack of satisfaction with the k motifs she has examined thus far; however, each adjustment of k will require all existing techniques to perform significant extra computation, *even* if they have cached the results of every calculation they have performed.

In contrast, the time needed for STOMP is totally independent of k. We only need to run STOMP once; as the matrix profile obtained already contains all necessary information, it takes trivial extra effort to find the top k motif, no matter how large k is.

D. Parameter Settings

As we previously noted, STOMP (together with STAMP) is unique among motif discovery algorithms in being completely parameter-free. In contrast, Random Projection [5] has four parameters, Quick-Motif [11] has three parameters, Tree-Motif has four parameters [21], MK [14] has one parameter, and FAST has three parameters [24].

[2] A small earthquake of that magnitude would only be felt by attentive humans in the immediate vicinity of the epicenter.

That being said, the reader may wonder about the only input value besides the time series of interest: the subsequence length *m*. Note that this is a required input for all the other existing techniques as well. We do not consider *m* a true parameter, as it is a *user choice,* reflecting her prior knowledge of the domain. Nevertheless, it is interesting to ask how sensitive motif discovery is to this choice, at least in the seismology domain that motivates us.

To test this, we edited the data above such that the two earthquakes in Fig. 7.*bottom* happen exactly 13 minutes 20 seconds apart. We reran motif discovery with *m*=2,000 (twenty seconds), with double that length (*m*=4,000), and with half that length (*m*=1,000). Fig. 8 shows the result.

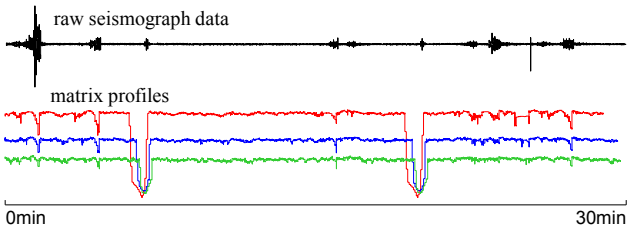

Fig. 8. *top*) Thirty minutes of seismograph data that has the two earthquakes from Fig. 7.*bottom* occur at 6min-40s and 20min. *bottom*) The matrix profile computed if we use the suggested subsequence length 2,000 (blue), or if we use twice the length (red), or half that length (green)

The results are very reassuring. At least for earthquakes, motif discovery is not sensitive to the user input. Even a poor guess as to the best value for *m* will likely give good results.

E. Case Studies in Seismology: Earthquake Swarm Case

In the previous section we discovered a repeating earthquake source that has a frequency of about once per 13.6 years. Here we consider earthquakes that are literally tens of millions of times more frequent.

Forecasting volcanic eruptions is of critical importance in many parts of the world [19]. For example, on May 18th, 1980 Mount St. Helens had a paroxysmal eruption that killed 57 people [10]. It is conjectured that explosive eruptions are commonly preceded by elevated or accelerated gas emissions and seismicity, thus seismology is a major tool for both monitoring and predicting such events.

In Fig. 9 we show a short section of the matrix profile of a seismograph recording at Mount St Helens. It is important to restate that this is *not* the raw seismograph data, but the matrix profile that STOMP computed from it.

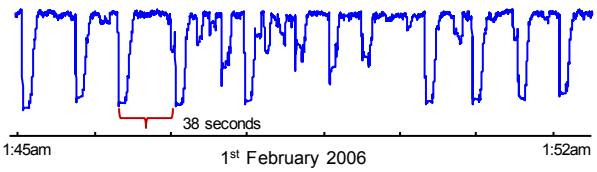

Fig. 9. The matrix profile of a seven-minute snippet from a seismograph recording at Mount St Helens.

The image shows a stunning regularity. Repeated earthquakes are occurring approximately once every thirty-eight seconds. This is consistent with the findings of a team from the US Geological Survey that reported that the earthquakes, which accompanied a dome-building eruption, appeared "*... so regularly that we dubbed them 'drumbeats'. The period between successive drumbeats shifted slowly with time, but was 30–300 seconds*" [10].

This example shows a significant advantage of our approach, that we share with STAMP but no other motif discovery algorithm. Instead of computing just O(*1*) distance values for the top *k* motifs, STOMP is computing *all* O(*n*) distances from *every* subsequence to their nearest neighbors. By plotting the entire matrix profile we can gain unexpected insights by seeing the motifs *in context*. For example, in the above we can see both the surprising periodicity of the earthquakes, and by comparing the smallest values in the matrix profile with the mean or maximum values, we can get a sense of how well the motifs are conserved, relative to "chance" occurrences. It could also potentially tell us whether there were changes to the earthquake source, reflecting changes in eruptive behavior over time.

A recent paper performed a similar analysis on the Mount Rainier volcano, making the interesting and unexpected discovery that the frequency of earthquakes is correlated with snowfall [2]. However, the paper bemoans at the number of ad-hoc "hacks" that needed to make such an exploration tenable. For example, "*In order to save on computing time, we cut out detections that are unlikely to contain a repeating earthquake event by excluding events with a signal width,"* and "*To save on computing time, we define that in order to be detected…*" etc. [2]. However, the results in TABLE IV tell us that we could simply bypass these issues by spending a few hours computing the *full* exact answers. This would avoid the risk that some speedup "trick" makes us miss an interesting and unexpected pattern.

F. A Case Study in Animal Behavior

While seismology is the primary motivator for this work, nothing about our algorithm assumes anything about the data's structure, or precludes us from considering other datasets. In this section we briefly consider telemetry collected from Magellanic penguins (*Spheniscus magellanicus*). The data was collected by attaching a small multi-channel data-logging device to the bird. The device recorded tri-axial acceleration, tri-axial magnetometry, pressure, etc. As shown in Fig. 10, for simplicity here we consider only Y-axis magnetometry.

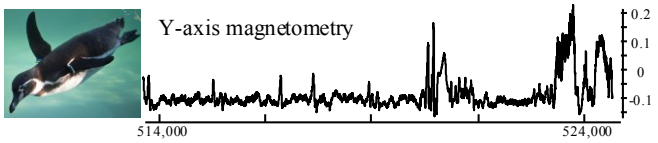

Fig. 10. *left*) The Magellanic penguin is a strong swimmer. *right*) A four-minute snippet of the full dataset reveals high levels of noise and no obvious structure.

The data is labeled by an observer with binoculars; thus we have a coarse ground truth for the animal's behavior. The full data consists of 1,048,575 data points recorded at 40 Hz (about 7.5 hours). We ran GPU-STOMP on this dataset, using a subsequence length of 2,000. This took our algorithm just 2.5 minutes. As shown in Fig. 11, the top motif is a surprisingly well conserved "shark fin" like pattern.

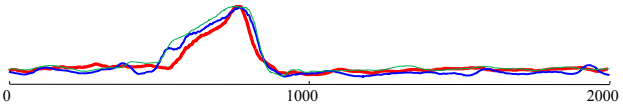

Fig. 11. The top motif of length 2,000 discovered in the penguin dataset. Only three examples are shown for visual clarity, there are eight such patterns.

What (if anything) does this pattern mean? Suggestively, we observed this pattern does not happen during any of the regions annotated as *nesting*, *walking*, *washing*, etc., but only during regions labeled *foraging*. Could this motif be related to a diving (for food) behavior?

Fortunately, diving is the one behavior we can unambiguously determine from the data, as the *pressure* increases by orders of magnitude when the penguin is under water. We discovered that the motif occurs moments before each dive, and nowhere else.

More generally, we see this example as being typical of the sort of interaction that motif discovery supports. In most cases motif discovery is not the *end* of analyses, but only the *beginning*. By correlating the observed motifs with other (internal or external) data we can form hypotheses and open avenues for further research. Recall the previous section; this is rather like how the team studying Mount Rainier's seismology discovered that its earthquakes are correlated with snowfall [2]. We believe that the STOMP algorithm may enable many such unexpected discoveries in a vast array of domains.

VI. Conclusions

We introduced STOMP, a new algorithm for time series motif discovery, and showed that it is theoretically and empirically faster than its strongest rivals in the literature, STAMP [23], Quick-Motif [11] and MK [14]. In the limited domain of seismology, we showed that STOMP is at least as fast as the recently introduced FAST algorithm [24], but STOMP does not allow false negatives and does not need careful parameter tuning. Moreover, for datasets and subsequences lengths encountered in the real world, STOMP requires one to three orders of magnitude less memory than rival methods. This is not a gap that is likely to be closed by a new implementation of these algorithms. STOMP is unique among motif discovery algorithms in *not* exacting subsequences, but doing all the computations in-situ.

We further showed optimizations that allow STOMP to take advantage of GPU architecture, opening an even greater performance gap and allowing the first exact motif search in a time series of length one-hundred-million.

In future work we plan to investigate multidimensional and incremental versions of our algorithms. The latter may have implications for real-time earthquake warning systems, reducing the probability of false alarms by ultra-fast lookup of dictionaries of previous confirmed events [24].

Acknowledgment

This research was funded by NSF IIS-1161997 II and NSF IIS 1510741. We gratefully acknowledge all the donors of the datasets.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami: Efficient Similarity Search In Sequence Databases. Springer Berlin Heidelberg. 1993: 69-84

[2] K. Allstadt, and S. D. Malone. Swarms of repeating stick-slip icequakes triggered by snow loading at Mount Rainier volcano. Journal of Geophysical Research: Earth Surface, 119.5 (2014): 1180-1203.

[3] A. Balasubramanian, J. Wang, P. Balakrishnan (In press). Discovering multidimensional motifs in physiological signals for personalized healthcare. IEEE Journal of Selected Topics in Signal Processing.

[4] A. E. X. Brown, E. I. Yemini, L. J. Grundy, T. Jucikas, and W. R. Schafer. A dictionary of behavioral motifs reveals clusters of genes affecting caenorhabditis elegans locomotion. Proceedings of the National Academy of Sciences, 110.2 (2013): 791–796.

[5] B. Chiu, E. Keogh, and S. Lonardi: Probabilistic discovery of time series motifs. KDD 2003: 493-498

[6] R. J. Geller, and C. S. Mueller. Four similar earthquakes in central California. Geophys. Res. Lett., 7.10 (1980): 821–824.

[7] M. Harris. Optimizing Parallel Reduction in CUDA. NVIDIA Developer Technology 2.4 (2007).

[8] J. Havskov, and G. Alguacil (2004). Instrumentation in Earthquake Seismology. Vol. 358. New York: Springer.

[9] T. Igarashi, T. Matsuzawa, A. Hasegawa (2003). Repeating earthquakes and interplate aseismic slip in the northeastern Japan subduction zone. Journal of Geophysical Research: Solid Earth, 108 (B5).

[10] R. M. Iverson, et al. Dynamics of seismogenic volcanic extrusion at Mount St. Helens in 2004–05. Nature 444.7118 (2006): 439–443.

[11] Y. Li, L. H. U, M. L. Yiu, Z. Gong. Quick-motif: An efficient and scalable framework for exact motif discovery. ICDE 2015: 579-590

[12] W. Luo, H. Tan, H. Mao, and L. M. Ni, 2012. Efficient Similarity Joins on Massive High-dimensional Datasets Using Mapreduce. In MDM'12, IEEE, pp. 1-10.

[13] X. Meng, X. Yu, Z. Peng, and B. Hong, Detecting earthquakes around salton sea following the 2010 mw7.2 El Mayor-Cucapah earthquake using GPU parallel computing, Procedia CS, vol. 9, pp. 937-946, 2012.

[14] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and M. B. Westover. Exact Discovery of Time Series Motifs. SDM 2009, vol. 9, pp. 473-484.

[15] NVIDIA CUDA C Programming Guide. Version 7.5. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[16] NVIDIA CUFFT Library User's Guide. Version 7.5. http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

[17] Project Website: http://www.cs.ucr.edu/~eamonn/MatrixProfile.html

[18] "Quick Motif", http://degroup.cis.umac.mo/quickmotifs/

[19] R. S. J. Sparks, Forecasting volcanic eruptions. Earth and Planetary Science Letters, 210.1 (2003): 1–15.

[20] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. Batista, M. B. Westover, Q. Zhu, J. Zakaria, E. J. Keogh: Addressing Big Data Time Series: Mining Trillions of Time Series Subsequences Under Dynamic Time Warping. TKDD 7.3 (2013): 10.

[21] L. Wang, E. S. Chng, and H. Li: A tree-construction search approach for multivariate time series motifs discovery. Pattern Recognition Letters 31,9 (2010): 869-875.

[22] X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, E. Keogh: Experimental comparison of representation methods and distance measures for time series data. Data Min. Knowl. Discov. 26.2 (2013): 275-309.

[23] C. C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, E. Keogh. Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets. IEEE ICDM 2016.

[24] C. E. Yoon, O. O'Reilly, K. J. Bergen, and G. C. Beroza. Earthquake Detection through Computationally Efficient Similarity Search. Sci. Adv. 1.11 (2015): e1501057.