GPU-APUMPEDI: A Parallel Algorithm for Computing Approximate Pan Matrix Profiles of Time Series

Zhang, Jing; Nikovski, Daniel; Nakamura, Takaaki

TR2023-091 July 15, 2023

Abstract

The Matrix Profile (MP) of a test time series with respect to a base time series has been shown to be a versatile primitive for many data mining tasks including time series anomaly detection. The MP records distances from all subsequences in the test time series to their respective nearest neighbors in the base time series. The Pan Ma- trix Profile (PMP) is a matrix with each row being an MP corresponding to a single subsequence length, and computing explicitly an exact PMP is slow. We propose a GPU-based approximation algorithm called GPU- APUMPEDI to compute the PMP under unnormalized Euclidean dis- tance by combining GPU-based MP algorithms and linear interpolation. We validate its efficiency and effectiveness through extensive numerical experiments on the UCR Anomaly Archive.

International conference on Time Series and Forecasting 2023

GPU-APUMPEDI: A Parallel Algorithm for Computing Approximate Pan Matrix Profiles of Time Series

Jing Zhang[1], Daniel Nikovski[1], and Takaaki Nakamura[2]

[1] Mitsubishi Electric Research Labs, Cambridge, MA, USA
{jingzhang,nikovski}@merl.com
[2] Mitsubishi Electric Corporation, Kamakura, Japan
nakamura.takaaki@dy.mitsubishielectric.co.jp

Abstract. The Matrix Profile (MP) of a test time series with respect to a base time series has been shown to be a versatile primitive for many data mining tasks including time series anomaly detection. The MP records distances from all subsequences in the test time series to their respective nearest neighbors in the base time series. The Pan Matrix Profile (PMP) is a matrix with each row being an MP corresponding to a single subsequence length, and computing explicitly an exact PMP is slow. We propose a GPU-based approximation algorithm called GPU-APUMPEDI to compute the PMP under unnormalized Euclidean distance by combining GPU-based MP algorithms and linear interpolation. We validate its efficiency and effectiveness through extensive numerical experiments on the UCR Anomaly Archive.

Keywords: Approximate Pan Matrix Profile, GPU, unnormalized Euclidean distance, interpolation, time series anomaly detection

1 Introduction

The Matrix Profile (MP) has been introduced as a useful tool for various data mining tasks in time series data [1]. The MP keeps track of the distances between the nearest neighbors of subsequences in a given time series. Algorithms based on the Fast Fourier Transform (FFT) or Dynamic Programming (DP) have been proposed to calculate the MP with the use of the normalized Euclidean distance metric [1,2]. Additionally, algorithms based on DP [3] or a special data structure such as a double-ended queue [4] have been developed to compute the MP with unnormalized Euclidean distance metric. The best time complexity for these algorithms is $O(n^2)$, with n being the length of the given time series, and is independent of the subsequence length.

Despite its usefulness, the MP requires the user to set the critical parameter of subsequence length for similarity search. To eliminate this need, the Pan Matrix Profile (PMP) was proposed [5]; it is a matrix with each row being a Matrix Profile for a different subsequence length. This makes the PMP a parameter-free version of the MP, with further potential for various time series data mining tasks. However, computing an exact PMP is slow, with a time complexity of $O(|M|n^2)$, where M is the list of all considered subsequence lengths, and $|M|$ is the number of subsequence lengths in M. To speed up the computation, the SKIMP approximation algorithm was proposed in [5], which optimizes the order of the candidate subsequence lengths to compute the MPs. However, as noted in [5], the normalized ℓ_2 distance is used in the computation, making it impossible to predict or produce an upper or lower bound for the MP for a given subsequence length.

Recently, other works [6,7] have attempted to eliminate the single parameter in MP computation, but specifically for the purpose of discord discovery (anomaly detection). The authors propose the MERLIN and MERLIN++ algorithms for finding arbitrary-length discords in massive time series archives. If we do not consider using GPU-based parallel computing, the MERLIN++ algorithm is the state-of-the-art in terms of speed, but it only focuses on finding discords as quickly as possible and avoids keeping track of the anomaly scores of most of the subsequences. On the other hand, the MP/PMP algorithms provide more information, including the anomaly scores of non-discords, which can be important for analysts in some situations. We also note that the output of the MERLIN++ algorithm is exactly the same as that of the MERLIN algorithm; the only advantage of the later version is the relatively faster speed. It is also worth pointing out that there still exist a lot of iterative computations in MERLIN++, making it difficult to be ported to a GPU-based version.

Based on the monotonicity of the MPs with respect to subsequence lengths under unnormalized distances, [8] proposed an alternative algorithm, APUMPEDI, short for Approximating Pan Matrix Profile under Unnormalized Euclidean Distance by Interpolation, to save computation time of (approximate) PMPs under unnormalized distances. However, it does not readily use the parallel computational power of modern GPUs, thus still facing speed bottlenecks when dealing with longer time series (for example one with 1 million data points). Thus, it is desirable to devise a GPU-based algorithm that further speeds up APUMPEDI.

Our contributions in this work are three-fold: (i) We extend the APUMPEDI algorithm to enable it to deal with more general PMP computation situations where we could have a base time series X^* and a test time series X and the former is not necessarily the same as the latter. (ii) We port the extended APUMPEDI algorithm to a GPU-based version. (iii) We conduct extensive numerical experiments to show the efficiency and effectiveness of our GPU-APUMPEDI algorithm. In particular, we consider several comparison studies using the UCR Anomaly Archive, thus showing the speed-up factors of GPU-APUMPEDI with respect to CPU-based alternatives and demonstrating the excellent anomaly detection performance compared to UCR-MP (MP under normalized distance) and MERLIN algorithms.

The paper is structured as follows. Section 2 extends the definition of the MP/PMP of time series and the monotonicity of the PMP under unnormalized

Euclidean distances. Section 3 details the GPU-APUMPEDI algorithm. The results of numerical experiments are presented in Section 4. We provide conclusions in Section 5 and a brief implementation of the GPU-APUMPEDI algorithm in the Appendix.

Notation: A time series X is composed of real-valued samples, each denoted by x_t, taken at time indices ranging from 0 to $n-1$. The j-th subsequence of X can be defined as $X_{j,\ldots,j+m-1} = [x_j, x_{j+1}, \ldots, x_{j+m-1}]$ for any subsequence length m such that $1 \leq m \leq n-1$ and $0 \leq j \leq n-m$. The length of a list A is indicated by $|A|$, and the index of a value m in a list M is shown as $m@M$. A range of elements from $[L, U]$ with a step size of S $(0 < S < U - L)$ can be expressed as $\text{range}(L, U, S)$, which is a list of all values that can be calculated as $L + i \times S$, where i is an integer. We write "w.r.t." as a shorthand for "with respect to."

2 Preliminaries

We first extend the definitions of the Matrix Profile [4] and the Pan Matrix Profile [9], under unnormalized Euclidean distances, to the case where we could have a base time series X^* that is not the same as the investigated time series X itself. For simplicity, we henceforth use UMP (resp., PUMP) (the letter "U" indicates "unnormalized" and is pronounced /ˈʌ/) to denote the Matrix Profile (resp., Pan Matrix Profile) of a test time series X w.r.t. a base time series X^*, under unnormalized Euclidean distance (i.e., the ℓ_2 distance).

Definition 1 (UMP)
The UMP of time series $X = [x_0, x_1, \ldots, x_{n-1}]$ w.r.t. $X^* = [x^*_0, x^*_1, \ldots, x^*_{n^*-1}]$ is a new time series $Y = [y_0, y_1, \ldots, y_{n-m}]$, where for $0 \leq j \leq n-m$,

$$y_j = \min_{0 \leq j' \leq n^*-m} d(X_{j,\ldots,j+m-1}, X^*_{j',\ldots,j'+m-1}),$$

where $d(\cdot,\cdot)$ is the ℓ_2 distance, and $1 \leq m \leq n^* - 1$ is a predefined subsequence length (window size).

In other words, at time index j, the value of the UMP of X w.r.t. X^* is the unnormalized Euclidean distance between the j-th subsequence and its nearest-neighbor subsequence in X^*.

Definition 2 (PUMP)
Given a list of subsequence lengths $M = [m_0, m_1, \ldots, m_{|M|-1}]$, the PUMP of time series $X = [x_0, x_1, \ldots, x_{n-1}]$ w.r.t. $X^* = [x^*_0, x^*_1, \ldots, x^*_{n^*-1}]$ is an $|M| \times n$ matrix P with each row filled by an UMP of X w.r.t. X^*, which corresponds to a specific subsequence length; in particular,

$$P[i,j] = \min_{0 \leq j' \leq n-m_i} d(X_{j,\ldots,j+m_i-1}, X^*_{j',\ldots,j'+m_i-1}), \tag{1}$$

$\forall 0 \leq i \leq |M|-1, 0 \leq j \leq n-m_i$, where $d(\cdot,\cdot)$ is the ℓ_2 distance and m_i is the subsequence length corresponding to the i-th row that satisfies $1 \leq m_i \leq n^* - 1$.

Note that, in Definition 2, we initialize the unfilled entries of P (i.e., $P[i,j], \forall 0 \leq i \leq |M|-1, n-m_i < j \leq n-1$) as 0. Next, we list a proposition about the monotonicity of the PUMP with respect to the subsequence lengths. The idea of its proof is the same as the one for [9, Theorem II.1].

Proposition 1 (Monotonicity of PUMP) *Assume for the list of subsequence lengths $M = [m_0, m_1, \ldots, m_{|M|-1}]$ we have $m_{i_1} < m_{i_2}, \forall 0 \leq i_1 < i_2 \leq |M|-1$. Then the PUMP of time series $X = [x_0, x_1, \ldots, x_{n-1}]$ w.r.t. $X^* = [x_0^*, x_1^*, \ldots, x_{n^*-1}^*]$ defined by* (1) *satisfy $P[i_1, j] \leq P[i_2, j], \forall 0 \leq i_1 < i_2 \leq |M|-1, 0 \leq j \leq n - m_{i_2}$.*

As pointed out in [9], Proposition 1 would not hold for the PMP under normalized Euclidean distances.

3 Algorithm

3.1 The Extended APUMPEDI Algorithm

In this section, we extend the APUMPEDI (stands for "Approximating Pan Matrix Profile under Unnormalized Euclidean Distance by Interpolation") algorithm to compute the PUMP for a test time series X w.r.t. a base time series X^*. The algorithm works as follows. Once we have computed the UMPs of X w.r.t. X^* for a set of selected subsequence lengths, we do interpolation for each and every missing subsequence length; e.g., if we have done computing the UMPs of X w.r.t. X^* for the following 5 subsequence lengths, $[10, 20, 30, 40, 50]$, then we do interpolation for the corresponding intervals $(10, 20)$, $(20, 30)$, $(30, 40)$, and $(40, 50)$, respectively, to obtain approximate UMPs of X w.r.t. X^* for subsequence lengths within these intervals. As pointed out in [8], the above idea is essentially different from that of the SKIMP algorithm [5]; the SKIMP algorithm orders the selected subsequence lengths and computes MPs under normalized distance for only a portion of them from the beginning.

We are now in a position to formalize the algorithm. Let L (resp., U) denote the minimum (resp., maximum) subsequence length such that $2 \leq L < U \leq n-1$, $S \in (0, U-L)$ the step size, and $\theta \in (0, 1]$ the completion rate for the PUMP computation. We wrap up the steps as Alg. 1, where Lines 10 through 18 describe the interpolation procedure. We apply a dynamic programming based algorithm [3] for the `UMP` subroutine (Line 8 in Alg. 1), which computes UMP of X w.r.t. X^*, for a given subsequence length m. For the `Interpolate` subroutine (Line 13), we use linear interpolation (see [8] for details) for simplicity of porting the algorithm to a GPU-based version. It is seen that, assuming $n^* \leq n$, the time complexity of this algorithm is $O(\theta|M|n^2)$, and the space complexity is $O(|M|n)$, where $|M| = \lceil (U-L)/S \rceil$ is the number of all predesignated candidate subsequence lengths.

Algorithm 1 Approximating Pan Matrix Profile under Unnormalized Euclidean Distance by Interpolation

```
1: procedure APUMPEDI($X^*, X, L, U, S, \theta$)
```

```
 2:     $n \leftarrow$ length of $X$
 3:     $M \leftarrow \text{range}\big(L, U, S\big)$
 4:     $P \leftarrow |M| \times n$ matrix of NaN's
 5:     $M' \leftarrow \text{range}\big(L, U, \lfloor\frac{1}{\theta}\rfloor \cdot S\big)$
 6:     for $m$ in $M'$ do
 7:         $i \leftarrow m@M$
 8:         $P[i,:] \leftarrow \texttt{UMP}(X^*, X, m)$
 9:     end for
10:     $x_{vec} \leftarrow M'$
11:     for $j$ in range$(0, n, 1)$ do
12:         $y_{vec} \leftarrow [P[m@M, j]$ for $m$ in $M']$
13:         $f(\cdot) \leftarrow \texttt{Interpolate}(x_{vec}, y_{vec})$
14:         for $m$ in $M \setminus M'$ do
15:             $i \leftarrow m@M$
16:             $P[i, j] \leftarrow f(m)$
17:         end for
18:     end for
19:     return $P$
20: end procedure
```

3.2 GPU-APUMPEDI

To port the above APUMPEDI algorithm to a GPU-based version, in Algorithm 1, we compute Lines 8, 11-18 in parallel by using GPU CUDA cores, respectively. For the convenience of interested readers, we provide a Python/Numba[3]/STUMP [10] based implementation at the Appendix.

4 Numerical Results

4.1 Data Set

We use the UCR Anomaly Archive [11] benchmark data set, which encompasses 250 distinct univariate time series from various fields like human medicine, biology, meteorology, and industry. The time series in the archive include both naturally occurring and artificially induced anomalies, with the majority being artificially generated. This offers a more in-depth analysis based on the type of anomaly introduced. The UCR Anomaly Archive was initially utilized in an anomaly detection contest prior to the 2021 ACM SIGKDD conference and was published by Wu and Keogh [12]. It serves as an alternative to commonly employed benchmark data sets, such as Yahoo S5 [13], Numenta [14], and NASA [15], which have been criticized for trivial anomalies, unrealistic anomaly densities, mislabeled ground truth, etc. Each time series in the UCR Anomaly Archive features a single anomaly, sometimes subtle, occurring after a particular time stamp, with the data prior to that stamp considered normal. The time series

[3] https://numba.pydata.org/

in the archive have lengths varying from 6,684 to 900,000 data points, with anomalies ranging from 1 to 1701 data points in length.

4.2 Computation Time

We conduct all the experiments on a PC with an AMD Ryzen 9 3950X CPU and an NVIDIA GeForce RTX 3090 GPU. We show the results (logarithmic computation time (seconds) vs. data index) of a set of comparison experiments on computation time in Fig. 1. The investigated algorithms are GPU-APUMPEDI, GPU-PUMP, and CPU-PUMP. We take $L = 20, U = 200, S = 1, \theta = 0.1$ for GPU-APUMPEDI, and take $L = 20, U = 200, S = 1$ for GPU-PUMP and CPU-PUMP. The indices of the time series we include are $20, 40, \ldots, 200$. The lengths of the corresponding base time series X^* and test time series X are listed in Table 1. Also shown in Table 1 are the speed-up factors, where "speed-up factor I" denotes the speed-up factor for GPU-APUMPEDI w.r.t. GPU-PUMP, and "speed-up factor II" denotes the speed-up factor for GPU-APUMPEDI w.r.t. CPU-PUMP. It is not surprising that all "speed-up factor I" values are close to 10, because we set $\theta = 0.1$ for GPU-APUMPEDI, and this would enable GPU-APUMPEDI to only cost about ten percent of the computation time compared to GPU-PUMP. Some of the "speed-up factor I" values are less than 10, due to the fact that the interpolation procedure still consumes some computational resources. Of particular interest are actually the "speed-up factor II" values, which range from 12 up to 434, depending on the lengths of X^* and X for each time series. During our experiments, we found that the larger these lengths, the larger the "speed-up factor II" values we would see. For the longest time series (No. 240 with length $240,030$ for X^* and length $659,970$ for X), the running time of GPU-APUMPEDI is about 20 minutes, while the running time of CPU-PUMP would be a few days.

Table 1: Specifications of selected time series and speed-up results.

time series #	20	40	60	80	100	120	140	160	180	200
length of X^* (k)	5	6	22	30	5	15	1	3.5	20	20
length of X (k)	7	24.066	43	160.05	25.001	15	6.321	7.808	35	108.001
speed-up factor I	8	9	9	10	8	8	9	8	10	10
speed-up factor II	14	48	102	**434**	52	32	12	15	87	265

4.3 Anomaly Detection Results

Metric We use the UCR score as the performance metric for our comparison study. Given any time series from the UCR Anomaly Archive, if its ground-truth anomaly starts at `begin` and ends at `end`, then the length of the anomaly is `len = end - begin + 1`. Letting the prediction of an algorithm be an integer `Q`,

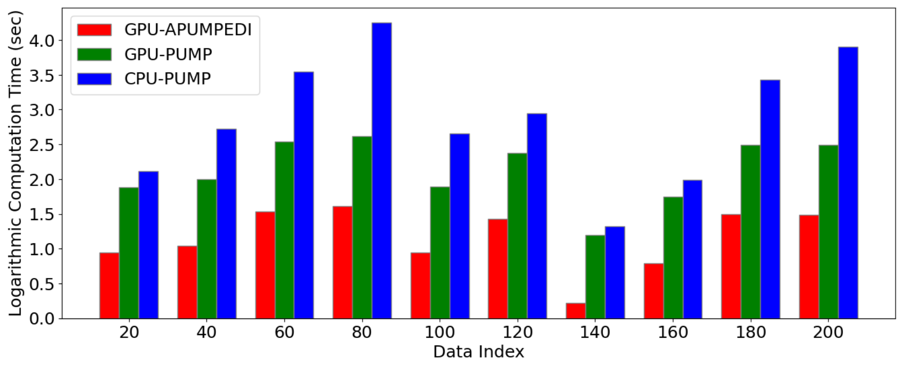


Fig. 1: The logarithmic computation time (seconds) vs. data index. We use logarithmic timing values instead of the raw values for the purpose of better visualization; note that some of the computation time values for GPU-APUMPEDI are too small (compared to the much larger timing values for CPU-PUMP), and they would be almost invisible if directly plotting them in the same plot.

then `Q` is labeled as correct if and only if `min(begin-len , begin-100) < Q < max(end+len, end+100)`.

Results We compare three algorithms for time series anomaly detection: (i) MERL_UMP, (ii) UCR_MP, and (iii) MERLIN. The essential idea of MERL_UMP and UCR_MP is the same (i.e., we compute the matrix profile of the test time series X w.r.t. the base time series X^*, identify the maximum value in the matrix profile, find the corresponding time index, and assign this time index as the detected anomaly location), but the former uses unnormalized Euclidean distance, whereas the latter uses normalized Euclidean distance. Similar to [16], the reason why we include MERLIN is because it is the built-in method provided with the UCR Anomaly Archive, which is the benchmark data set in our study. We also note that both MERL_UMP and UCR_MP can be computed in parallel using a GPU (or multiple GPUs), while MERLIN (and MERLIN++) involves a lot of iterative steps which brings difficulty for parallel computation. In other words, although MERLIN++ is the current state-of-the-art in terms of CPU-based computation, it is definitely not the fastest when comparing to other algorithms that can make use of GPUs. As a matter of fact, during our experiments, the MERLIN/MERLIN++ algorithms fell behind MERL_UMP and UCR_MP in terms of speeds, because the latter two can make use of the computational power of the GPU. We point out that our GPU-APUMPEDI algorithm here could play a big role (in terms of saving computation time) for MERL_UMP when θ is set relatively small (say 0.1).

In Figure 2 we plot the UCR scores of the three investigated algorithms for subsequence lengths ranging from 5 to 200. For MERLIN, a few subsequence lengths less than 60 led to divergent results and we skip those. It is seen that

MERL_UMP consistently performs the best when the subsequence length is less than 110, while UCR_MP and MERLIN have very close scores and their scores are slightly higher than MERL_UMP's when subsequence lengths exceed 110.

To see more clearly the effectiveness of our GPU-APUMPEDI algorithm, in Figures 3 through 7, we plot 5 time series (X^* in green, X in blue, and the ground-truth anomaly in red) from the UCR Anomaly Archive, and their respective approximate PUMP's heatmap. Note that these approximate PUMPs are computed using the GPU-APUMPEDI algorithm with $L = 20, U = 200, S = 1, \theta = 0.1$. It is seen that the heatmaps derived from No. 100, 140, and 160 time series directly indicate the correct anomaly; the lightest area corresponds to the largest UMP values, thus helping us locate the time index of the anomaly.

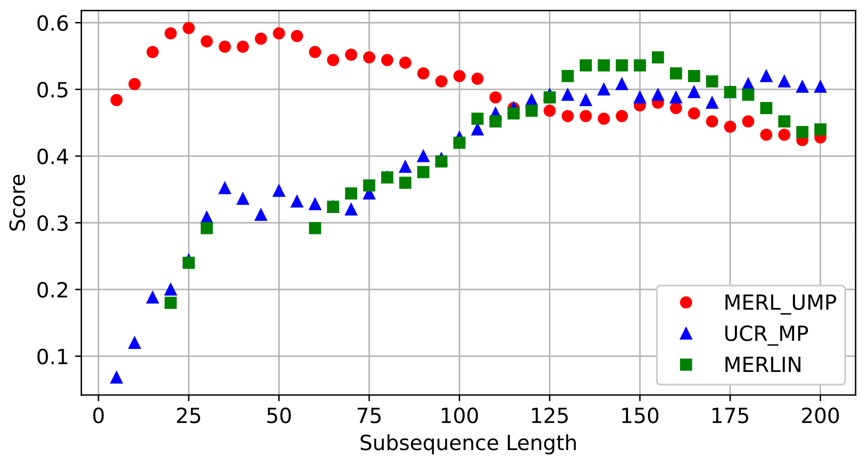


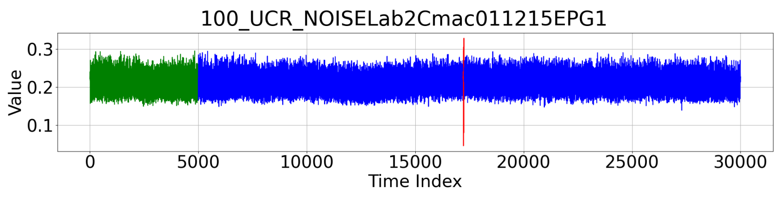
Fig. 2: The UCR scores on the UCR Anomaly Archive of the three investigated anomaly detection algorithms for subsequence lengths ranging from 5 to 200.

5 Conclusion

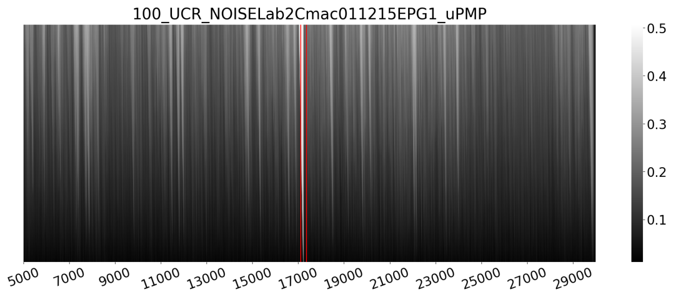
In this study, we extended the APUMPEDI algorithm to enable it to deal with more general PMP computation situations where we could have a base time series and a test time series and the former could be different from the latter. We ported the extended APUMPEDI algorithm to a GPU-based version. Finally, through extensive numerical experiments, we showed the efficiency and effectiveness of our algorithm.

References

1. Yeh, C.C.M., Zhu, Y., Ulanova, L., Begum, N., Ding, Y., Dau, H.A., Silva, D.F., Mueen, A., Keogh, E.: Matrix profile i: all pairs similarity joins for time series:

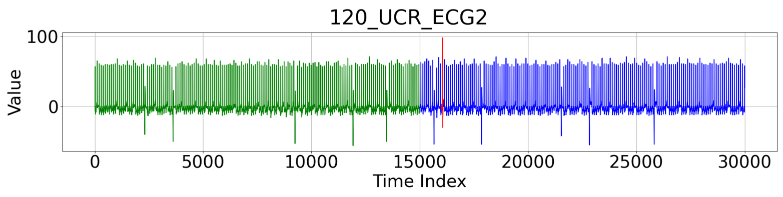


(a) The time series No. 100.

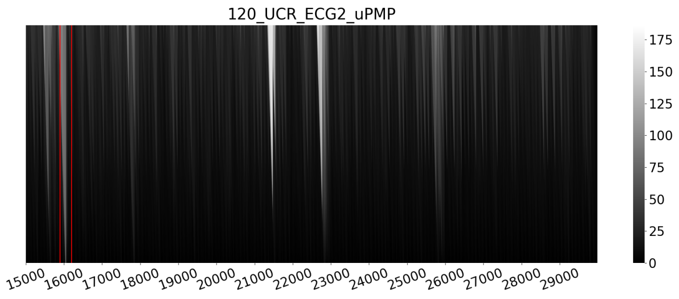


(b) The heatmap of the corresponding PUMP.

Fig. 3: Time series No. 100 from the UCR Anomaly Archive and the heatmap of its PUMP derived using the GPU-APUMPEDI algorithm with $L = 20, U = 200, S = 1, \theta = 0.1$. The lightest area correctly indicates an anomaly.

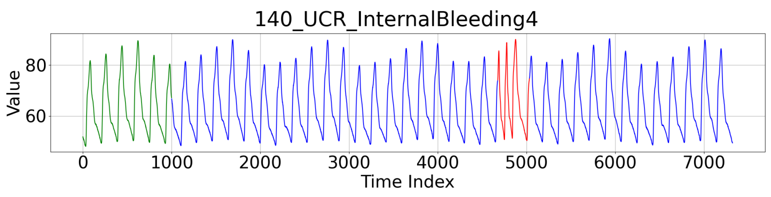


(a) The time series No. 120.

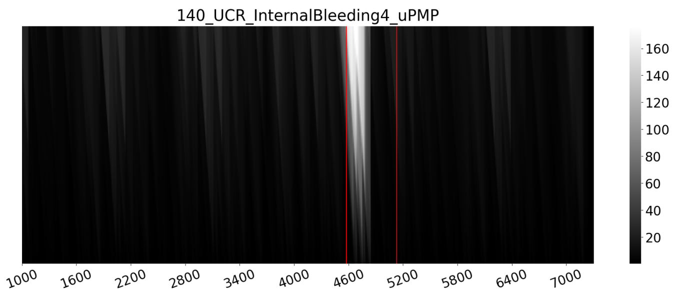


(b) The heatmap of the corresponding PUMP.

Fig. 4: Time series No. 120 from the UCR Anomaly Archive and the heatmap of its PUMP derived using the GPU-APUMPEDI algorithm with $L = 20, U = 200, S = 1, \theta = 0.1$. The lightest area introduces a false anomaly.

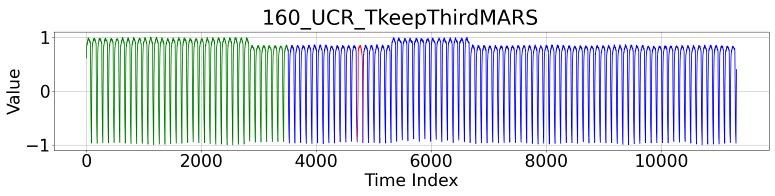


(a) The time series No. 140.

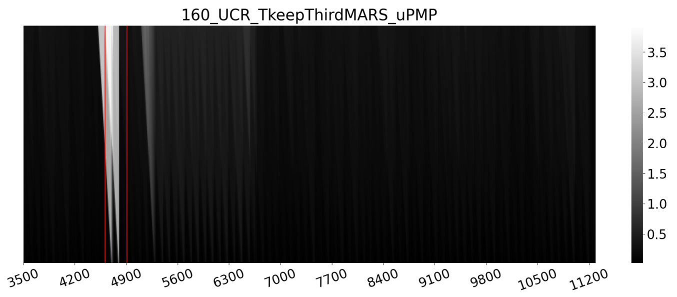


(b) The heatmap of the corresponding PUMP.

Fig. 5: Time series No. 140 from the UCR Anomaly Archive and the heatmap of its PUMP derived using the GPU-APUMPEDI algorithm with $L = 20, U = 200, S = 1, \theta = 0.1$. The lightest area correctly indicates an anomaly.

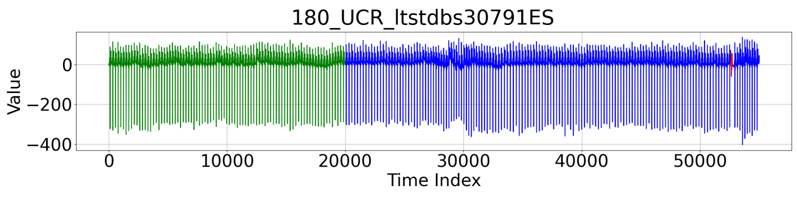


(a) The time series No. 160.

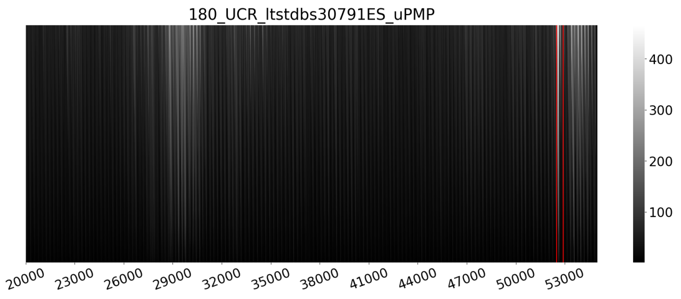


(b) The heatmap of the corresponding PUMP.

Fig. 6: Time series No. 160 from the UCR Anomaly Archive and the heatmap of its PUMP derived using the GPU-APUMPEDI algorithm with $L = 20, U = 200, S = 1, \theta = 0.1$. The lightest area correctly indicates an anomaly.



(a) The time series No. 180.



(b) The heatmap of the corresponding PUMP.

Fig. 7: Time series No. 160 from the UCR Anomaly Archive and the heatmap of its PUMP derived using the GPU-APUMPEDI algorithm with $L = 20, U = 200, S = 1, \theta = 0.1$. The lightest area could arguably correctly indicate an anomaly, but with lower confidence than the cases for No. 100, 140, and 160.

a unifying view that includes motifs, discords and shapelets. In: 2016 IEEE 16th international conference on data mining (ICDM). (2016) 1317–1322

2. Zhu, Y., Zimmerman, Z., Senobari, N.S., Yeh, C.C.M., Funning, G., Mueen, A., Brisk, P., Keogh, E.: Matrix profile ii: Exploiting a novel algorithm and gpus to break the one hundred million barrier for time series motifs and joins. In: 2016 IEEE 16th international conference on data mining (ICDM), IEEE (2016) 739–748
3. Akbarinia, R., Cloez, B.: Efficient matrix profile computation using different distance functions. arXiv preprint arXiv:1901.05708 (2019)
4. Zhang, J., Nikovski, D.: Algorithms for fast computation of matrix profiles of time series under unnormalized Euclidean distances. In: International Conference on Applied Statistics and Data Analytics, available at `https://www.merl.com/publications/docs/TR2022-040.pdf`. (2022)
5. Madrid, F., Imani, S., Mercer, R., Zimmerman, Z., Shakibay, N., Keogh, E.: Matrix profile xx: Finding and visualizing time series motifs of all lengths using the matrix profile. In: 2019 IEEE International Conference on Big Knowledge (ICBK), IEEE (2019) 175–182
6. Nakamura, T., Imamura, M., Mercer, R., Keogh, E.: Merlin: Parameter-free discovery of arbitrary length anomalies in massive time series archives. In: 2020 IEEE International Conference on Data Mining (ICDM). (2020) 1190–1195
7. Nakamura, T., Mercer, R., Imamura, M., Keogh, E.: Merlin++: parameter-free discovery of time series anomalies. Data Mining and Knowledge Discovery (2023) 1–40

8. Zhang, J., Nikovski, D.: APUMPEDI: Approximating pan matrix profiles of time series under unnormalized Euclidean distances by interpolation. In: The 8th International conference on Time Series and Forecasting, available at https://www.merl.com/publications/docs/TR2022-088.pdf. (2022)
9. Zhang, J., Nikovski, D.: Algorithms for fast computation of pan matrix profiles of time series under unnormalized Euclidean distances. In: International Conference on Applied Statistics and Data Analytics, available at https://www.merl.com/publications/docs/TR2022-041.pdf. (2022)
10. Law, S.M.: STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining. The Journal of Open Source Software **4**(39) (2019) 1504
11. Keogh, E., Taposh, D.R., Naik, U., Agrawal, A.: Multi-dataset time-series anomaly detection competition. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. available at https://compete.hexagon-ml.com/practice/competition/39/. (2021)
12. Wu, R., Keogh, E.: Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress. IEEE Transactions on Knowledge and Data Engineering (2021)
13. Laptev, N., Amizadeh, S., Billawala, Y.: S5-a labeled anomaly detection dataset, version 1.0 (16M). In: available at https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70. (2015)
14. Lavin, A., Ahmad, S.: Evaluating real-time anomaly detection algorithms–the numenta anomaly benchmark. In: 2015 IEEE 14th international conference on machine learning and applications (ICMLA), IEEE (2015) 38–44
15. Hundman, K., Constantinou, V., Laporte, C., Colwell, I., Soderstrom, T.: Detecting spacecraft anomalies using LSTMs and nonparametric dynamic thresholding. In: Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining. (2018) 387–395
16. Rewicki, F., Denzler, J., Niebling, J.: Is it worth it? An experimental comparison of six deep-and classical machine learning methods for unsupervised anomaly detection in time series. arXiv preprint arXiv:2212.11080 (2022)

Appendix

We note that in the following implementation, we do not explicitly introduce the completion rate parameter θ; instead, one could implicitly tune θ by changing the value of `STEP_SIZE`.

```
@cuda.jit
def gpu_apumpedi(an_array):
    i, j = cuda.grid(2)
    if i < an_array.shape[0] and j < an_array.shape[1]:
        m = j + LOW_BOUND
        num_steps = int((m - LOW_BOUND)/STEP_SIZE)
        m_L = LOW_BOUND + num_steps*STEP_SIZE
        m_H = m_L + STEP_SIZE
        u_L = an_array[i, m_L-LOW_BOUND]
        u_H = an_array[i, m_H-LOW_BOUND]
        an_array[i, m-LOW_BOUND] = u_L + \
        (m - m_L)*(u_H - u_L)/STEP_SIZE

milestone_win_sizes = range(LOW_BOUND, UP_BOUND+1, STEP_SIZE)
x = sorted(milestone_win_sizes)
ary = np.zeros((len(ts[tp:])-x[0]+1, x[-1]-x[0]+1))

for m in x:
    ary[:len(ts[tp:])-m+1, m-x[0]] = \
    stumpy.gpu_stump(np.array(ts[tp:]), m, \
             T_B=np.array(ts[:tp]), \
             normalize=False, ignore_trivial = False)[:, 0]
for j in x:
    for i in range(len(ts[tp:])-j+1, ary.shape[0]):
        ary[i, j-x[0]] = ary[i, j-x[0]-(x[1]-x[0])]

d_ary = cuda.to_device(ary)

threadsperblock = (32, 32)
d0, d1 = d_ary.shape[0], d_ary.shape[1]
blockspergrid_x = math.ceil(d0 / threadsperblock[0])
blockspergrid_y = math.ceil(d1 / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

gpu_apumpedi[blockspergrid, threadsperblock](d_ary)

h_ary = d_ary.copy_to_host()
```