The Journal of Supercomputing
https://doi.org/10.1007/s11227-023-05060-2

Time series analysis acceleration with advanced vectorization extensions

Ricardo Quislant[1] · Ivan Fernandez[1,2] · Eladio Gutierrez[1] · Oscar Plata[1]

Accepted: 14 January 2023
© The Author(s) 2023

Abstract
Time series analysis is an important research topic and a key step in monitoring and predicting events in many fields. Recently, the *Matrix Profile* method, and particularly two of its Euclidean-distance-based implementations—SCRIMP and SCAMP—have become the state-of-the-art approaches in this field. Those algorithms bring the possibility of obtaining exact motifs and discords from a time series, which can be used to infer events, predict outcomes, detect anomalies and more. While matrix profile is embarrassingly parallelizable, we find that auto-vectorization techniques fail to fully exploit the SIMD capabilities of modern CPU architectures. In this paper, we develop custom-vectorized SCRIMP and SCAMP implementations based on AVX2 and AVX-512 extensions, which we combine with multithreading techniques aimed at exploiting the potential of the underneath architectures. Our experimental evaluation, conducted using real data, shows a performance improvement of more than 4× with respect to the auto-vectorization.

Keywords Time series analysis · Matrix profile · Parallelism · Vectorization

✉ Ricardo Quislant
quislant@uma.es

Ivan Fernandez
ivanfv@uma.es

Eladio Gutierrez
eladio@uma.es

Oscar Plata
oplata@uma.es

[1] Department of Computer Architecture, University of Malaga, Bulevar Louis Pasteur, 35, 29071 Malaga, Andalucia, Spain

[2] Universitat Politècnica de Catalunya (UPC), C. Jordi Girona, 1-3, 08034 Barcelona, Catalonia, Spain

Published online: 02 February 2023

1 Introduction

A time series is a chronologically ordered set of samples of a real-valued variable that can contain millions of observations. Time series analysis (TSA) seeks extracting models in a large variety of domains [1] such as epidemiology, DNA analysis, economics, medicine, geophysics, and speech recognition. Particularly, *motif* [2] (similarity) and *discord* [3] (anomaly) discovery has become one of the most frequently used primitives in time series data mining [4–9]. TSA poses the problem of solving the all-pairs-similarity-search (also known as similarity join). Specifically, given a time series sliced in subsequences, retrieve the nearest motif and the furthest discord neighbor for every subsequence with respect to the rest ones.

The state-of-the-art method for motif and discord discovery is *Matrix Profile* [10] (MP). MP solves the similarity join problem and allows time-manageable computation of very large time series datasets. In this work, we focus on this technique, which provides full joins without the need to specify a similarity threshold, which is a very challenging task in this domain. When analyzing a time series, the matrix profile is another time series representing the minimum distance subsequence for each subsequence (motifs). Maximum values of the profile highlight the most dissimilar subsequences (discords). Figure 1 depicts a naive example of anomaly detection using *matrix profile*, where the sinusoidal signal has an anomaly between values 250 and 270. The *matrix profile* output of this time series shows low values for the periodic subsequences of it as they are very similar to the other subsequences, and higher values for the anomalies and their neighboring subsequences.

The two state-of-the-art implementations of matrix profile are SCRIMP [11] and SCAMP [12]. While being embarrassingly parallel, we find that auto-vectorization approaches (e.g., using OpenMP [13]) fail to fully exploit the SIMD capabilities of modern CPU architectures. Thus, this work is focused on how much benefit those matrix profile implementations can take from custom vectorizations based on AVX2 and AVX-512 extensions.

This work makes the following *contributions*:

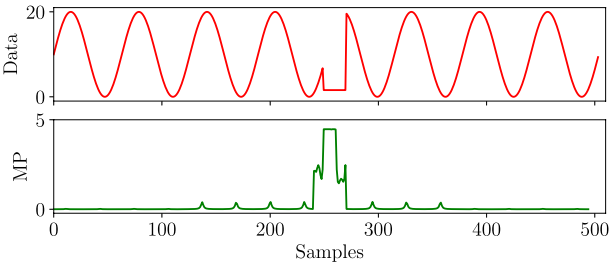

Fig. 1 A time series with an anomaly and its matrix profile. The anomaly appears as high values of Euclidean distance, which means low similarity with respect to the other subsequences of the series

- We analyze auto-vectorized implementations of the matrix profile, and find that this approach is not able to fully exploit the underneath SIMD capabilities of Intel CPUs.
- We develop `SCRIMPvect` and `SCAMPvect`, two custom-vectorized versions of SCRIMP and SCAMP, exploiting AVX2 and AVX-512 vector extensions.
- We detect a problem with masking vectorization instructions on AVX2, and propose a solution to tackle it by using AVX-512 instructions with 256-bit registers.
- We perform a thread affinity and load balance exploration.
- We analyze the roofline of the algorithms before and after vectorization.
- We perform an analysis of `SCRIMPvect` and `SCAMPvect` in terms of performance and find that our approach shows an improvement of more than 4× with respect to the auto-vectorization.

The paper is organized as follows. Section 2 outlines the motivation. In Sect. 3, we present some background related to the problem. Section 4 explains how the algorithms have been implemented to extract maximum performance from the platforms. In Sect. 5, we present the experimental evaluation and discuss the results. Section 6 provides some related work. Finally, Sect. 7 draws the conclusions.

2 Motivation

We evaluate the CPU/memory roofline for SCRIMP and SCAMP to determine the main limiting factors of the algorithms. Intel's Advisor [14] analyzes and provides these kind of roofline reports, showing the arithmetic intensity and memory traffic demands of the main loops in the application. Figure 2 shows these data for the most time consuming loops (the blue and green dots) in SCRIMP and SCAMP. The results show that both algorithms are bound by compute and memory roofs. The arithmetic intensity, that is, the number of arithmetic operations per byte, is not so high for SCAMP than for SCRIMP. Locality is not so prominent in general for both algorithms.

These results motivate the use of custom-vectorization approaches like those proposed in this paper, as the computation is performed with vectors of contiguous data. Consequently, caches will be better harnessed than in the scalar version of the algorithms. These approaches will make the dots in Fig. 2 go higher on the y-axes and, hopefully on the x-axes as well, with the outcome of a performance improvement.

3 Background

3.1 Time series analysis. The matrix profile

A given *time series* T is a sequence of n data values t_i, $1 \leq i \leq n$, collected over the course of time. We define $T_{i,m}$ as a subsequence of T, where i is the index of its first data value, T_i, and m is the number of data points in the subsequence with $1 \leq i$, and

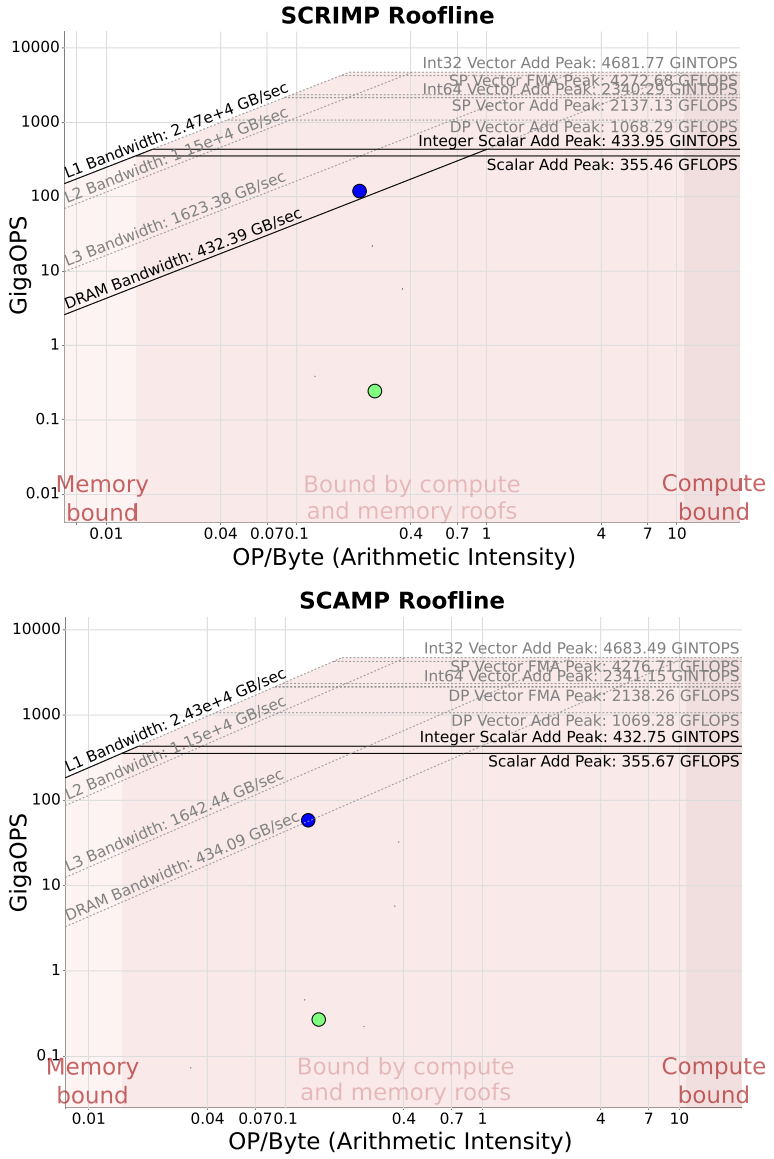

Fig. 2 SCRIMP and SCAMP rooflines (INT+FLOAT) for the Power time series with 180,000 samples, 1325 window size, and 128 threads. Green dots correspond to the loop computing the dot product in case of SCRIMP (line 6 in Fig. 6) and the covariance in case of SCAMP. Blue dots correspond to the loop computing the rest of the diagonal (line 13 in Fig. 6). See Sect. 3 to gain more insight into the algorithms

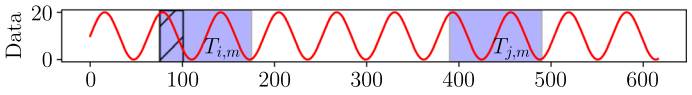

Fig. 3 Example of two subsequences, $T_{i,m}$ and $T_{j,m}$, of a given time series. When computing the matrix profile P, subsequences starting in the exclusion zone of $T_{i,m}$ are ignored for their high similarity

$m \leq n$. In the literature, we also find $T_{i,m}$ defined as a *window* of length m. Figure 3 presents an example of a time series with two subsequences highlighted, $T_{i,m}$ and $T_{j,m}$.

We find several ways to measure the similarity between two time series subsequences, $T_{i,m}$ and $T_{j,m}$. One possible approach is the use of the *z-normalized Euclidean distance* [11], $d_{i,j}$, which is calculated as follows:

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

where $Q_{i,j}$ is the dot product of subsequences $T_{i,m}$ and $T_{j,m}$. μ_x and σ_x are the mean and the standard deviation of the data points in $T_{x,m}$, respectively.

One of the state-of-the-art matrix profile's implementations, SCRIMP [11], exploits the fact that the dot product can be updated incrementally for subsequences in the diagonal of the distance matrix D (see Fig. 4) to minimize the computation. Consequently, the dot product can be expressed as follows:

$$Q_{i,j} = Q_{i-1,j-1} - T_{i-1}T_{j-1} + T_{i+m-1}T_{j+m-1} \tag{2}$$

Matrix profile algorithms can compute diagonals either in-order or in random order. If a partial solution is sufficient for the target application, using a random order for the diagonals allows stopping the execution of the algorithm before it finishes. In contrast, if the application needs the oracle (exact) matrix profile, we can use the in-order implementation which leads to a lower execution time thanks to data locality. These algorithms are designed to store only the matrix profile and the matrix profile index arrays, computing the minimum distances $d_{i,j}$ on the fly. Note that the neighboring subsequences of $T_{i,m}$ are highly similar to it (i.e., $d_{i,i+1} \approx 0$) due to the overlaps. Thus, we can exclude these subsequences from computing D to find similar subsequences other than the neighboring ones. This is done by defining an exclusion zone (diagonal stripped zone in Fig. 3) for each subsequence. In general, the exclusion zone for a subsequence $T_{i,m}$ of length m is $\frac{m}{4}$ [11].

We can use the distance measure to find the most similar subsequences out of all subsequences of a time series T. There are three steps to this procedure: (i) building a symmetric $(n - m + 1) \times (n - m + 1)$ matrix D, called *distance matrix*, with a window

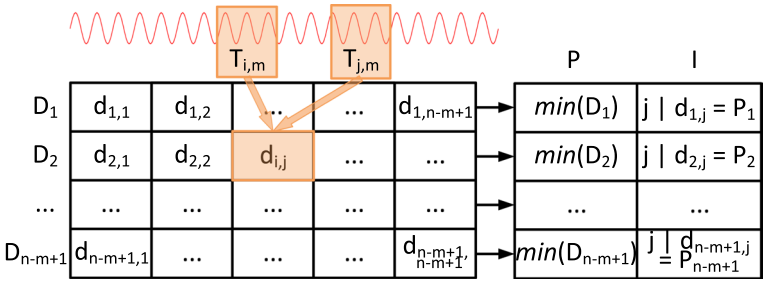

Fig. 4 Computation of the matrix profile P and the profile index I from the distance matrix D. P_i is the minimum distance of each row (or column) D_i. I_i is the index of the subsequence providing such minimum

size m and a time series of length n. Each D cell, $d_{i,j}$, stores the distance between two subsequences, $T_{i,m}$ and $T_{j,m}$. Thus, each row (or column) of D stores the distances between a subsequence of T and every subsequence of T; (ii) finding the two subsequences whose distance is minimum (i.e., most similar sequences) in each row (or column) of D. This can be computed by building the *matrix profile*, P, which is a vector of size $n - m + 1$. Each cell P_i in P stores the minimum value found in the i^{th} row (or column) of D; (iii) finding the indices of the most similar subsequences of the matrix profile. This requires building another vector I –called *matrix profile index*– of the same size as that of P, where $I_i = j$ if $d_{i,j} = P_i$. In this way, P contains the minimum distances between subsequences of T, while I is the vector of "pointers" to the location of these subsequences. Figure 4 depicts an example of the distance matrix (D), the matrix profile (P), and the matrix profile index (I) for the time series in Fig. 3.

SCAMP [12] follows a similar computation scheme than SCRIMP, but instead replaces the sliding dot product with a mean-centered-sum-of-products with the goal of reducing the number of operations required and the floating-point rounding errors. The following equations can be precomputed in $O(n - m + 1)$ time, having the length of the profile vector defined as $n - m + 1 = l$:

$$df_i = \frac{T_{i+m-1} - T_{i-1}}{2}, \quad 0 < i < l \tag{3}$$

$$dg_i = T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}, \quad 0 < i < l \tag{4}$$

$$ssq_i = \begin{cases} \sum_{k=0}^{m-1} \left(T_k - \mu_0\right)^2, & i = 0 \\ ssq_{i-1} + \left(T_{i+m-1} - \mu_i + T_{i-1} - \mu_{i-1}\right) & \\ \left(T_{i+m-1} - T_{i-1}\right), & 0 < i < l \end{cases} \tag{5}$$

$$\sigma_i = \sqrt{ssq_i}, \quad 0 \leq i < l \tag{6}$$

Equations (3) and (4) are terms used in the covariance update of Eq. (7), and the standard deviation (L2-norm of subsequence $T_{i,m} - \mu_i$) calculated in Eqs. (5) and (6) is used for the Pearson correlation coefficient depicted by Eq. (8). Note the exclusion zone in the limits of Eq. (7) given by $\frac{m}{4}$.

$$\sigma_{i,j} = \begin{cases} \sum_{k=0}^{m-1} \left(T_k - \mu_0\right)\left(T_{k+j} - \mu_j\right), & i = 0, \frac{m}{4} < j < l \\ \sigma_{i-1,j-1} + df_i dg_j + df_j dg_i, & i > 0, \frac{m+4}{4} < j < l \end{cases} \tag{7}$$

$$P_{i,j} = \frac{\sigma_{i,j}}{\sigma_i \sigma_j} \tag{8}$$

$$D_{i,j} = \sqrt{2m\left(1 - P_{i,j}\right)} \tag{9}$$

The matrix profile can be derived incrementally for each diagonal of the distance matrix, Eq. (7), from the calculation of the covariance of two subsequences of the first row (first piece in Eq. (7)). The Pearson correlation coefficient in Eq. (8) can be computed in fewer operations and it is more robust than the Euclidean Distance used by SCRIMP. Equation (9) calculates the distance from the Pearson coefficient in $O(1)$.

3.1.1 Parallelization

Matrix profile's diagonal independence scheme allows easy parallelization without need for synchronization. The goal is to avoid the use of mutual exclusion methods, looking for performance boosting. This is achieved by expanding [15] the matrix profile, creating a replica (row) per thread as depicted in Fig. 5.

This is computationally safe as long as the performed operation per iteration is commutative and associative, i.e., it is a reduction loop [16]. Although shared, each replica can be seen as a private storage during the calculation of the matrix profile, in such a way that each thread computes its private matrix profile, storing the computed partial results in it. The only imperative synchronization is a barrier to wait for all threads to finish its private computation. Once all threads have calculated their private matrix profiles, a final reduction operation (per column) is carried out to compute the final results. No synchronization is necessary in this stage because each thread is in charge of a different column, so it can be done fully in parallel. Note that an analogous approach is applicable to the matrix profile index, as shown in Fig. 5.

The baseline auto-vectorized implementation for our SCRIMP algorithm, (Fig. 6), is a vectorized-parallel version presented in [17]. It first precomputes the mean and standard deviation of each time series subsequence (line 1), and initializes the matrix profile array (line 3). Then, the distances between pairs of subsequences are calculated following the diagonals of the distance matrix (lines 4–26). The for loop is fully parallelized,

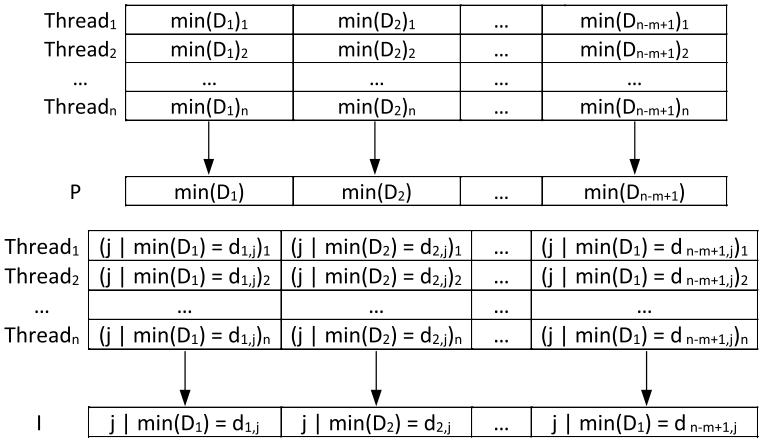

Fig. 5 Expanded structure for matrix profile distances (top) and indexes (bottom). Each thread computes a subset of diagonals of D using its private structures. A final reduction is needed

```
1:  μ, σ ← computeMeanDev(T, m);
2:  vectFact ← 8/sizeof(datatype);
3:  P ← ∞;
4:  for idx ← tid * nDiag to (tid + 1) * nDiag − 1 do
5:      i ← 0; j ← diag_idx;
6:      q ← dotProduct(T_{i,m}, T_{j,m});
7:      d ← dist(m, q, μ_i, σ_i, μ_j, σ_j);
8:      if d < P_i then
9:          P_i ← d; I_i ← j;
10:     if d < P_j then
11:         P_j ← d; I_j ← i;
12:     i ← i + 1;
13:     for j ← diag_idx to size(P) do
14:         for k ← 0 to vectFact − 1 do
15:             qs_k ← t_{i+m−1+k} t_{j+m−1+k} − t_{i−1+k} t_{j−1+k};
16:         qs_0 ← qs_0 + q
17:         for k ← 1 to vectFact − 1 do
18:             qs_k ← qs_k + qs_{k−1};
19:         q ← qs_{vectFact−1};
20:         for k ← 0 to vectFact − 1 do
21:             ds_k ← dist(m, qs_k, μ_{i+k}, σ_{i+k}, μ_{j+k}, σ_{j+k});
22:             if ds_k < P_{i+k} then
23:                 P_{i+k} ← ds_k; I_{i+k} ← j + k;
24:             if ds_k < P_{j+k} then
25:                 P_{j+k} ← ds_k; I_{j+k} ← i + k;
26:         i ← i + vectFact;
```

Fig. 6 Parallel and auto-vectorized SCRIMP algorithm

with each thread computing a random subset of diagonals provided by their indices in the *diag* array in line 5. For the first element of the diagonal, we need to compute the dot product of the first pair of subsequences (line 6) in parallel. The rest are updated following Eq. (2). For the proper vectorization of the dot product update, the algorithm separates the calculation of the diagonal in several steps: (i) the products in Eq. (2) are calculated in parallel for *vectFact* elements of the diagonal (lines 14–15); (ii) the previous dot product, q, is added to the element calculated in step (i) (line 16); (iii) the subsequent dot products are updated sequentially using the previous ones (lines 17–18) saving the last one in q for the next iteration of the diagonal (line 19); (iv) distances are calculated in parallel (lines 20–21); and (v) the profile is updated in parallel as well (lines 22–25). The vectorization factor, *vectFact*, is given by the datatype width with respect to that of double precision (line 2).

3.2 Vectorization extensions

During the last decades, we have witnessed great advances with respect to the SIMD capabilities of commodity processors. Going back to 1997, Intel introduced

the MMX [18] instructions in the first Pentium generation. In 1998, AMD introduced 3DNow! [19] in the AMD K6-2 processor. In 1999, Intel introduced SSE [20] (Streaming SIMD Extensions) in the second Pentium generation, consisting of four incrementally improved subgenerations (SSE, SSE2, SSE3 and SSE4). More recently, Intel proposed Advanced Vector Extensions (AVX) [21] in 2008. These AVX have been increasingly upgraded to support new features and larger vectors, being AVX, AVX2 and AVX-512 introduced in chronological order. In this work, we focus our attention in the state-of-the-art SIMD extensions of modern processors (i.e., AVX and its successive subgenerations).

3.2.1 AVX

AVX (2011) is the first generation of Advanced Vector Extensions, introduced in Sandy Bridge by Intel and in Bulldozer by AMD. These extensions use sixteen 256-bit YMM registers to perform single arithmetic instructions (floating point) on multiple data. This way, each register is capable of handling at the same time either eight single-precision values or four double-precision values.

3.2.2 AVX2

AVX2 (2013) is the second generation of Advanced Vector Extensions, introduced in Haswell by Intel. The main improvements were the capability of using integers, instead of floating point numbers only, and the introduction of new instructions (e.g. `vpmaskmov` for conditional SIMD integer packed loads and stores [22]).

3.2.3 AVX-512

AVX-512 (2016) is the third generation of Advanced Vector Extensions, introduced in Knights Landing by Intel. AVX-512 increases the register width from 256-bit to 512-bit and introduces multiple subsets of extensions aimed to different purposes (e.g., byte manipulation, prefetching instructions, neural network instructions,...). CPUs do not necessarily implement every extension subset introduced in AVX-512.

4 `SCRIMPvect` and `SCAMPvect`

In this section, we present our proposed vectorization implementations of SCRIMP and SCAMP, discussing the changes and techniques added to them.

4.1 Vectorization approach

We already find an OpenMP auto-vectorization-based implementation in the literature [17]. Using such approach, the authors unroll the inner loops of SCRIMP (line 13 in Fig. 6) in a way that OpenMP SIMD pragma realizes where to enable and exploit the vectorization, thus improving performance. Figure 7 presents to what

extent that approach compares with the custom-vectorization scheme proposed in this work. The OpenMP approach vectorizes within one diagonal of the distance matrix (D), which imposes certain difficulties related to the fact that each element of the diagonal depends on the immediately previous element. This is partially solved by the aforementioned inner loop unrolling. Our custom vectorization, though, packs the elements of contiguous diagonals (a meta-diagonal) in the same vector, thus getting rid of such dependencies.

4.2 Implementations

While having similar implementations, SCRIMP and SCAMP differences mainly relies on the computation of the distance between two time series subsequences. With this in mind, we propose a very similar vectorization scheme for both algorithms. Whenever possible, we use FMA (Fused Multiply Add) instructions with the goal of speeding up the execution of the calculations.

We also notice that the intrinsics of AVX2 and AVX-512 are very similar and there is a clear bijection between them. In addition, we create *wrappers* for each intrinsic that allow us to use the same set of macros for AVX2 (Fig. 8) and AVX-512 (Fig. 9). In this way, we can specify whether we are going to use one instruction set extension or the other at compile time by specifying -DAVX2 or -DAVX-512. The types used for the matrix profile (DTYPE) and the profile index (ITYPE) are double and long long int, respectively, which are 64-bit types. This facilitates the vectorization as both P and I can be packed with the same number of elements. Furthermore, 64-bit integers allow for series longer than 4 billion elements.

First, in the outermost loop (line 4 in Fig. 6), each thread take iterations in VPACK sets (see Fig. 10), being VPACK the width of the vector (4 in case of AVX2 and 8 for AVX-512). Notice that the diagonals in the exclusion zone are elided (*exclusionzone* + 1).

Then, we compute the dot product (the covariance in the case of SCAMP) of the two subsequences involved in the first element of the diagonal so that we can

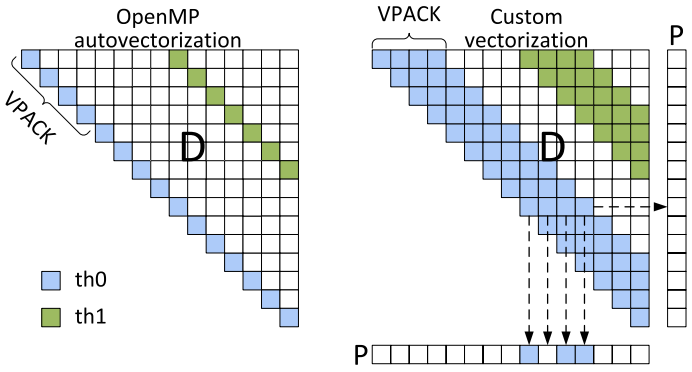

Fig. 7 OpenMP auto-vectorization (left) versus custom vectorization (right) with 2 threads. Vector width (VPACK) set to 4 distance values

```
#define ALIGN 32
#define VPACK 4
#define VDTYPE __m256d
#define VITYPE __m256i
#define VMTYPE VITYPE

#define SETZERO_PD() _mm256_setzero_pd()
#define SET1_PD(a) _mm256_set1_pd(a)
#define SET1_EPI(a) _mm256_set1_epi64x(a)
#define LOADU_PD(a) _mm256_loadu_pd(a)
#define LOADU_SI(a) _mm256_loadu_si256(a)
#define STORE_PD(a, b) _mm256_store_pd(a, b)
#define STOREU_PD(a, b) _mm256_storeu_pd(a, b)
#define STORE_SI(a, b) _mm256_store_si256((VITYPE *)a, b)
#define STOREU_SI(a, b) _mm256_storeu_si256((VITYPE *)a, b)
#define FMADD_PD(a, b, c) _mm256_fmadd_pd(a, b, c)
#define SUB_PD(a, b) _mm256_sub_pd(a, b)
#define ADD_PD(a, b) _mm256_add_pd(a, b)
#define MUL_PD(a, b) _mm256_mul_pd(a, b)
#define CMP_PD(a, b, c) (VMTYPE) _mm256_cmp_pd(a, b, c)
#define BLEND_EPI(a, b, c) _mm256_blendv_epi8(a, b, c)
#define BLEND_PD(a, b, c) _mm256_blendv_pd(a, b, (VDTYPE)c)
#define MASKSTOREU_PD(a, b, c) _mm256_maskstore_pd(a, b, c);
#define MASKSTOREU_EPI(a, b, c) _mm256_maskstore_epi64(a, b, c);
```

Fig. 8 Wrappers for the AVX2 version of the algorithms

obtain the distance afterwards (lines 6 and 7 in Fig. 6). For the OpenMP auto-vectorization, the dot product is easily auto-vectorized by the *simd pragma*, as only two subsequences are involved. However, the custom approach has to compute dot products of the first subsequence in the time series with other VPACK subsequences at a time. VPACK subsequences makes up a meta-diagonal, as shown in Fig. 7. As shown below (the code is from STAMPvect), we have to load unaligned (`LOADU_PD`) packs of elements from the time series and the precomputed statistics arrays, belonging to the VPACK subsequences that we want to compare with the first subsequence in terms of distance. The time series values of the first subsequence have to be replicated in the vector (`SET1_PD`) to do so:

Thus, in the case of SCAMP, we obtain the covariances packed in a vector, for each of the diagonals we have taken. From this data, we obtain the correlation and then we substitute the data of the matrix profile (P) if the correlation is higher than the previously calculated, as we illustrate in Fig. 12. Notice that the comparison (`CMP_PD`) of the obtained correlations with those already present in P are done in parallel when updating P vertically (see Fig. 7). However, when updating P horizontally we should compute the horizontal maximum of the vector containing the correlations (`correlation_v`) and then update the P value

```
#define ALIGN 64
#define VPACK 8
#define VDTYPE __m512d
#define VITYPE __m512i
#define VMTYPE __mmask8

#define SETZERO_PD() _mm512_setzero_pd()
#define SET1_PD(a) _mm512_set1_pd(a)
#define SET1_EPI(a) _mm512_set1_epi64(a)
#define LOADU_PD(a) _mm512_loadu_pd(a)
#define LOADU_SI(a) _mm512_loadu_si512(a)
#define STORE_PD(a, b) _mm512_store_pd(a, b)
#define STOREU_PD(a, b) _mm512_storeu_pd(a, b)
#define STORE_SI(a, b) _mm512_store_si512((VITYPE *)a, b)
#define STOREU_SI(a, b) _mm512_storeu_si512((VITYPE *)a, b)
#define FMADD_PD(a, b, c) _mm512_fmadd_pd(a, b, c)
#define SUB_PD(a, b) _mm512_sub_pd(a, b)
#define ADD_PD(a, b) _mm512_add_pd(a, b)
#define MUL_PD(a, b) _mm512_mul_pd(a, b)
#define CMP_PD(a, b, c) _mm512_cmp_pd_mask(a, b, c)
#define BLEND_EPI(a, b, c) _mm512_mask_blend_epi64(c, a, b)
#define BLEND_PD(a, b, c) _mm512_mask_blend_pd(c, a, b)
#define MASKSTOREU_PD(a, b, c) _mm512_mask_storeu_pd(a, b, c)
#define MASKSTOREU_EPI(a, b, c) _mm512_mask_storeu_epi64(a, b, c)
```

Fig. 9 Wrappers for the AVX-512 version of the algorithms

```
#pragma omp for schedule(dynamic)
for (ITYPE diag = exclusionZone + 1; diag < profileLength; diag +=
↪  VPACK)
```

Fig. 10 Loop partitioning in meta-diagonals of VPACK diagonals

if applicable. Conversely, we opted for doing this sequentially (see Sect. 4.3 for more information).

In the case of SCRIMP, instead of using the criterion of maximizing the correlation, we look for the pair of subsequences with the smallest Euclidean distance, as we explained in Sect. 3. To do so, we change the flag `CMPT_GT_OQ` to `CMPT_LT_OQ` in the comparison instruction `CMP_PD`.

Next, we iterate and compare with all remaining subsequences by incrementing j in the innermost loop (as in line 13 of Fig. 6). Once all iterations are computed, we combine the work of the threads, which have been working with a private profile, in a final reduction (not shown in Fig. 6 but actually performed, as the privatization is the key for the algorithm to be embarrassingly parallel). The code in Fig. 13 is our vectorization for the final reduction in SCAMPvect. That code snippet goes through

all the values of each thread's private profile and takes the pair of subsequences whose correlation is higher in the case of SCAMP, and those of smaller Euclidean distance in the case of SCRIMP. This way, it composes the final result for the matrix profile of the time series.

4.3 Challenges

In this subsection, we discuss some challenges we found while developing our implementations. Specifically, the fact that taking the data in groups of *VPACK* may lead to obtain inaccurate results, particularly in the case of SCRIMP.

When we allocate memory for the time series and its statistics (mean and deviation) the arrays are initialized to 0, and also the extra VPACK elements allocated at the end of them. If these elements are used by a thread computing on the edge of the distance matrix (D), it could read data beyond the end of the time series. These 0 elements distort the distance calculation (e.g., the SCRIMPS's Euclidean distance $= 2 * (windowSize - (dotProd - windowSize * means[j] * means[i])/(devs[j] * devs[i]))$, if the means and deviations are 0 then the distance is minus infinite). SCRIMP is an algorithm that searches for the minimum distance among subsequences. Therefore, that minus infinity distance will be a minimum that will be stored into the matrix profile, when this should not be the case.

To solve this problem, we initialize all *VPACK* values to Not-a-Number, so that the result of operating with them will be *NaN* as well. This way, any comparison with a *NaN* value will be false and will not be taken into account. To this end, we used the flag `_CMP_LT_OQ` for the comparisons. O, for Ordered, causes false to be returned if either element in the comparison is *NaN*. And Q, for Quiet, causes no signal to be thrown if there is a division by 0 or any error in the comparison.

Another challenge we faced is that of the horizontal maximum mentioned in Sect. 4.2. To update the matrix profile horizontally, as depicted in Fig. 7, we have to implement the vector horizontal maximum of the correlation vector (`correlation_v` in Fig. 12). Whereas this problem can be solved by means of maxing the vector with a shuffled version of itself until we obtain the maximum value replicated throughout the whole vector, the problem is that we have to obtain the index of the subsequence giving such a maximum. That is not a trivial task, so we decided to do it sequentially.

5 Experimental evaluation

5.1 Experimental setup and methodology

We conduct our experiments using a server which is equipped with four Intel Xeon Gold 5218 processors [23], featuring AVX2 and AVX-512 vectorization extensions. This system has 755GB of RAM, and we will refer to it as *xeongold* from now on. The setup of xeongold is arranged in four sockets with 16 cores in each processor. Each core implements hyperthreading so that 2 threads can execute simultaneously

in a core. Consequently, the total number of hardware threads available amounts to 128. Each core owns 32KB of L1I and 32KB of L1D caches, along with a private 1KB L2 cache. The L3 cache is shared among cores and its size is 22MB. The TLB is split in its first level with up to 128 entries for instructions and 64 entries for data, for 4KB pages. There is a second TLB level shared among cores with 1536 entries [24].

The codes are compiled using the GNU g++ compiler, version 7.5.0, with -O2 optimization flag, so that auto-vectorization is not enabled for the scalar code. For the AVX2 and AVX-512 codes we use `-mavx2` and `-mav512f` flags, respectively, and `-mfma` to enable fused-multiply-add instructions. Furthermore, we have experimented with AVX-512 instructions using 256-bit YMM registers, for which we use the `-mav512vl` flag. All experiments are executed 10 times and the time results are averaged. Thread affinity is left in charge of the operating system unless otherwise mentioned. The operating system is Ubuntu with Linux kernel 4.15. A study on the effect of thread affinity can be found in Sect. 5.4.

We use a comprehensive set of series for the experimental evaluation, whose parameters are detailed in Table 1. We show the window size (subsequence length, m) and the number of samples (n) of the time series, as well as the maximum and minimum values. The name of each series is descriptive of the type of content they contain. Note that all time series used in this work come from real data [25]. Table 2 shows the series we use to evaluate the proposals with large data sets.

5.2 Results

We present the results for the SIMD optimizations in SCRIMP and SCAMP in Figs. 14 and 15, respectively. We show how the speedup curves change with the number of threads, from 1 to 128, with respect to the scalar with one OpenMP thread for the various implementations: (i) scalar (No-vect), (ii) AVX2 vectorization (AVX2), (iii) AVX-512 instructions using 256-bit registers (AVX-512vl), (iv) AVX-512 vectorization (AVX-512) and (v) OpenMP auto-vectorization (OMP-Vect) (SCRIMP only).

In general, we observe that the version of the algorithm using AVX-512 is far superior to the rest in almost all cases, but AVX2 is not always clearly better than the non-vectorized one.

We observe a particularly noticeable improvement obtained by 512-bit vectorization and a high number of threads in the *ECG*, *Power*, *Seismology* and *Penguin* series. These are series with much more samples than those of *Audio* and *Human*, so there is more work to be done by every thread. Notice that AVX-512 divides the distance matrix in meta-diagonals of 8 diagonals each (see Fig. 7), and these meta-diagonals are the work unit for each thread. In *Audio* and, more noticeably in *Human*, which only has about 7000 samples (see Table 1), this fact leaves less than 1000 meta-diagonals for the threads to work with, which makes load balancing difficult to manage, see Sect. 5.5 to gain insight into the load imbalance problem. On top of that, those meta-diagonals are very different in size and, consequently, in number of computations. The meta-diagonals near the main diagonal are much larger than

Table 1 Time series workloads

Time series	n	m	Max	Min
Audio	20.234	200 (2 s)	6.69	−56.48
ECG	180.000	500 (2 s)	2.6	0.325
Power	180.000	1.325 (8 h)	140	0
Seismology	180.000	50 (2.5 s)	696.1	−185.7
Human Act	6.997	120 (12 s)	2.51	−2.9
Peng. Behav	109.842	800	0.52	−0.21

Table 2 Large time series

Time series	n	m	Max	Min
ECG	1.800.000	500 (2 s)	3.39	−1635
Power	1.859.587	1.325 (8 h)	140	0
Seismology	1.728.000	50 (2.5 s)	2329.3	−2334.6

those towards the end of the distance matrix. The 64 and 128 thread configurations are specially harmful for *Audio* and *Human* because of this imbalance.

We can notice a slowdown of AVX2 over the No-Vect implementation for *ECG*, *Power*, and *Penguin* up to 16 threads. The other implementations do not go below the scalar version, but do not show much of a speedup up to 8 threads either. This behavior coincides with the series having a large window size, from 500 to 1325 samples. A large window makes the loop in Fig. 11 perform many unaligned loads that hinders the vectorization improvement. Moreover, the L1D cache is stressed by the larger window, and the data TLB is stressed even more, as shown in Table 3. The table gathers results of the data TLB store misses for a series with large window size, Power, and a series with small window size, Seismology, and the same length. This problem vanishes with 32 threads (see speedups in Figs. 14 and 15). The amount of aggregated TLBs and the smallest work to be done by each thread seems to be enough to tackle the TLB problem. With larger series, the problem can persist even with 32 threads and two sockets (see Sect. 5.6).

Moreover, there is an additional problem to AVX2 related with the MASKSTORE and BLEND instructions (see Figs. 12 and 13) which introduce high overhead, in contrast to AVX-512 where the performance is improved via opmask predicated instructions (see Sect. 5.3). From 16 threads upwards, different sockets are being used, so the cache and TLB capacity increases, which seems to hide the slowdown of the aforementioned overhead. Short-window time series, though, benefit greatly from vectorization approaches, as the overhead caused by these problems is not so predominant.

In the case of SCRIMP (Fig. 14), we also include an implementation of the algorithm that uses the automatic vectorization by means of the OpenMP library [17]. The results of this version are presented as OMP-Vect in the plots. In general, we find that OMP-Vect is clearly worse for all series than AVX-512, and also worse

```
VDTYPE covariance_v = SETZERO_PD();
for (ITYPE i = 0; i < windowSize; i++) {
  VDTYPE tSeriesWinDiag_v = LOADU_PD(&tSeries[diag + i]);
  VDTYPE meansWinDiag_v = LOADU_PD(&means[diag]);
  VDTYPE tSeriesWin0_v = SET1_PD(tSeries[i]);
  VDTYPE meansWin0_v = SET1_PD(means[0]);
  covariance_v = FMADD_PD(SUB_PD(tSeriesWinDiag_v, meansWinDiag_v),
↪  SUB_PD(tSeriesWin0_v, meansWin0_v), covariance_v);
}
```

Fig. 11 SCAMPvect's covariance computation of the first subsequences in a meta-diagonal. The rest are computed incrementally from these values

than AVX2 except for those series with large window size and few threads. Actually, the OMP-Vect implementation barely outperforms the No-Vect one, except for the *Human* series.

5.3 Masking overhead

Seeking the reason behind the dramatic lack of performance of AVX2 we break down the time of the covariance computation (code snippet in Fig. 11) and the conditional profile update (code snippet in Fig. 12) to find that `MASKSTOREU` is a source of high overhead. We use `_mm256_maskstore` instrinsics for the AVX2 implementation (see Fig. 8) and `_mm256_mask_storeu` intrinsics for the AVX-512 (see Fig. 9), which in turn compile to `vmaskmov`/`vpmaskmov` and `vmovup mask` instructions, respectively.

Intel AVX-512 introduces opmask registers and predicate instructions which encode an operand from the set of opmask registers, so that the user has per-element control on the computational operation and memory update depending on the mask [26]. Conversely, AVX2 provides differentiated instructions for masking and blending (e.g., `vmaskmov` and `vblendv`) which can degrade the performance in certain cases. The Intel 64 and IA-32 Architectures Optimization Reference Manual [26] recommends using `vmaskmov` only in cases where `vmovup` cannot be used. Furthermore, it states the following: "32-byte AVX store instructions that span two pages require an assist that costs roughly 150 cycles". Assists are microcode invoked by hardware to help handle certain events. We use `perf` to retrieve the number of these assists triggered by our implementations to get the data in Table 4. We use `other_assists.any` which gets the "Number of times a microcode assist is invoked by hardware other than FP-assist. Examples include AD (page Access Dirty) and AVX* related assists", as stated by `perf list`. We can see that the data is highly correlated with that in Table 3. Although the assists are not broken down, we might say by the figures in the table that AVX2's `vmaskmov` implementation might trigger assists even with 0-masked addresses, which may set A and D bits and might update the TLB. The Intel's ISA Reference Manual [22] states for `vmaskmov` that, in cases where

mask bits indicate data should not be loaded or stored, paging A and D bits will be set in an implementation dependent way.

To gain more insight into the masking problem we take advantage of the possibility of using the AVX-512 instruction set with 256-bit registers, with the option AVX-512vl. The only differences between our AVX2 and AVX-512vl implementations are the replacement of `vmaskmov` and `vblend` by `vmov mask` instructions. The performance improvement is noticeable by the numbers in Tables 3 and 4, and the performance gains over the AVX2 implementation in Figs. 14 and 15, especially for large window series up to 16 threads.

5.4 Thread affinity exploration

Results shown so far leaves thread affinity in charge of the operating system, which, as we will see here, is the best solution. In this section we study the effect of pinning threads to cores using the thread affinity features offered by OpenMP.

Figures 16 and 17 show the speedup results for SCRIMP (SCAMP yields similar results which are omitted in this section) for AVX-512 and AVX2, respectively, such that no thread affinity (nobind), close thread affinity (close) and spread thread affinity (spread) are enforced. For the close and spread thread affinity policies, we set the following environmental variable: `OMP_PLACES="0,1:64:2"`, so that 64 places are defined with 2 threads each. This way, each place corresponds to a physical core with 2 hyperthreads.

The close processor binding assigns the threads to places beginning from the first place and following next places in increasing order, with a final wrap-around when we run out of places. So we can assume that, from 1 to 16 threads, only one socket is used and no hyperthreading. With 32 threads, 2 sockets are used and no hyperthreading. with 64 threads, 4 sockets are used and no hyperthreading. And finally, with 128 threads, hyperthreading is used. The spread processor binding assign threads to places far away, thus using different sockets from 2 threads onward.

Figures show the same behavior for the close and nobind policies up to 32 threads spawned. However, the spread policy yields worse results in most cases due to the use of several sockets, which reduces locality harnessing and hinders thread synchronization (barrier at the end of `#pragma omp for` loops). This effect is more noticeable with shorter series (*Audio* and *Human*). For 64 threads, the nobind policy is able to yield slightly better results than the close policy, as the operating system have the ability to migrate threads to idle cores, just in case there is a load from system processes. For 128 threads the results are obviously the same whatever the policy.

The AVX2 results in Fig. 17 give the same results for large window size series up to 32 threads, no matter the policy, which hints thread affinity is not the culprit of the AVX2 slowdown with respect to No-Vect.

5.5 Load balance exploration

The main loop of the algorithms computes the diagonals of the distance matrix. The thread's work unit is the computation of a diagonal or a meta-diagonal (a group of VPACK diagonals), and the assignment of these (meta)-diagonals to threads is done dynamically by means of the `schedule(dynamic)` OpenMP clause.

As diagonals are of different lengths (shorter as they get the right end of the distance matrix), load balancing these algorithms can be a problem. Furthermore, the computation of the dot product (SCRIMP) or covariance (SCAMP) of the first element in the diagonal (see Fig. 11) is heavier than that of the remaining elements, which are computed incrementally from the value of the first element. This first element computation problem exacerbates as window size increase, and with vectorization due to unaligned accesses.

Figure 18 shows the SCAMP results (SCRIMP figures are very similar and we omit them for the sake of conciseness) of the various implementations when executed with `schedule(static)` compared with the executions with `schedule(dynamic)`. Lower is worse for the static versions. We can observe that a static partition of (meta-)diagonals is not a good idea for these algorithms. The worst outcome is yielded by the vectorized algorithms, as expected. No-Vect gets affected as well, but to a lower extent overall. This is because thread's work unit is a diagonal, which is finer grain than a meta-diagonal.

Note the deviation from 100% for Audio and Human with one thread. These are the shortest series of the set, which yields execution times under 1 s for one thread thus showing high variability even when averaging the 10 runs per experiment (see Sect. 5.1).

As mentioned in Sect. 5.2, AVX2 gets poor results when the window size is large, due to instruction set and TLB problems. We can see in Fig. 18 how static AVX2 yields the worse results due to the addition of load imbalance, specially with 32 threads, where 2 sockets are used.

Table 3 SCAMP data TLB store misses for the given implementations and series

Impl.	Series	n (K)	m	# dTLB store misses		
				#th = 2	#th = 16	#th = 32
AVX2				9.320.502.463	5.209.871.973	24.371.203
AVX-512vl	Power	180	1.325	3.903.164.142	1.949.028.376	9.241.465
AVX-512				1.874.597.618	744.614.596	1.868.692
AVX2				144.912.286	60.832.193	13.134.275
AVX-512vl	Seismo	180	50	57.375.614	44.190.128	6.439.342
AVX-512				40.623.017	31.620.232	2.220.071

```
alignas(ALIGN) DTYPE correlation[VPACK];
STORE_PD(correlation, correlation_v);
for (int ii = 0; ii < VPACK; ii++) {
  if (correlation[ii] > profile_tmp[i + my_offset]) {
    profile_tmp[i + my_offset] = correlation[ii];
    profileIndex_tmp[i + my_offset] = j + ii;
    }
}
VDTYPE profilej_v = LOADU_PD(&profile_tmp[j + my_offset]);
VMTYPE mask = CMP_PD(correlation_v, profilej_v, _CMP_GT_OQ);
MASKSTOREU_PD(&profile_tmp[j + my_offset], mask, correlation_v);
MASKSTOREU_EPI(&profileIndex_tmp[j + my_offset], mask, SET1_EPI(i));
```

Fig. 12 Comparison of obtained correlations with the already present in P, and update if applicable

```
#pragma omp for schedule(static)
for (ITYPE colum = 0; colum < profileLength; colum += VPACK) {
  VDTYPE max_corr_v = SET1_PD(-numeric_limits<DTYPE>::infinity());
  VITYPE max_indices_v;
  for (ITYPE th = 0; th < numThreads; th++) {
    VDTYPE profile_tmp_v = LOADU_PD(&profile_tmp[colum + (th *
↪   (profileLength + VPACK))]);
    VITYPE profileIndex_tmp_v = LOADU_SI((VITYPE
↪   *)&profileIndex_tmp[colum + (th * (profileLength + VPACK))]);
    VMTYPE mask = CMP_PD(profile_tmp_v, max_corr_v, _CMP_GT_OQ);
    max_indices_v = BLEND_EPI(max_indices_v, profileIndex_tmp_v, mask);
    max_corr_v = BLEND_PD(max_corr_v, profile_tmp_v, mask);
  }
  STORE_PD(&profile[colum], max_corr_v);
  STORE_SI(&profileIndex[colum], max_indices_v);
}
```

Fig. 13 Final reduction vectorization

5.6 Large series results

In this section, we explore the behavior of the different implementations of the algorithms when stressed over large series inputs. The series tested are those in Table 2, which are close to 2 million elements (n).

Figure 19 shows the results obtained for SCRIMP (on the left) and SCAMP (on the right). Thread affinity is left to the operating system, and iterations are dynamically scheduled by the OpenMP runtime. We can see that the larger series stress the caches and TLB even when adding more sockets and threads to the computation. The performance gains for large window size series (Power and ECG) are not so acute with 32 threads than those of smaller series. AVX2 yields poor results even with 64 and 128 threads. However, as more sockets and threads are used, the AVX-512 and

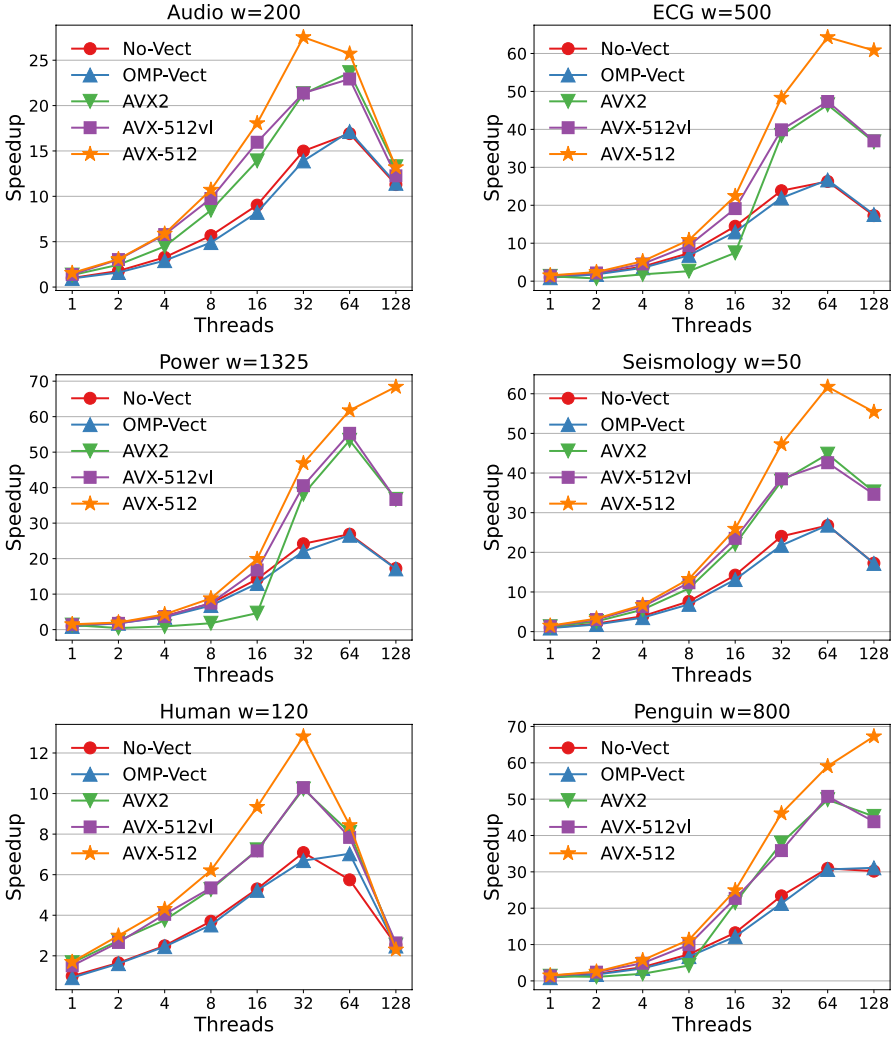

Fig. 14 SCRIMP results. 128 threads implies hyperthreading

Table 4 SCAMP other assists for the given implementations and series

Impl.	Series	n (K)	m	# Other assists		
				#th = 2	#th = 16	#th = 32
AVX2				2.284.759.397	2.299.158.617	10.163.170
AVX-512vl	Power	180	1.325	2.065.544.112	1.835.219.679	7.677.854
AVX-512				1.059.781.425	638.604.656	25.799
AVX2				54.138.705	31.766.931	2.284.644
AVX-512vl	Seismo	180	50	35.562.400	33.353.984	2.127.670
AVX-512				21.002.933	26.417.715	25.973

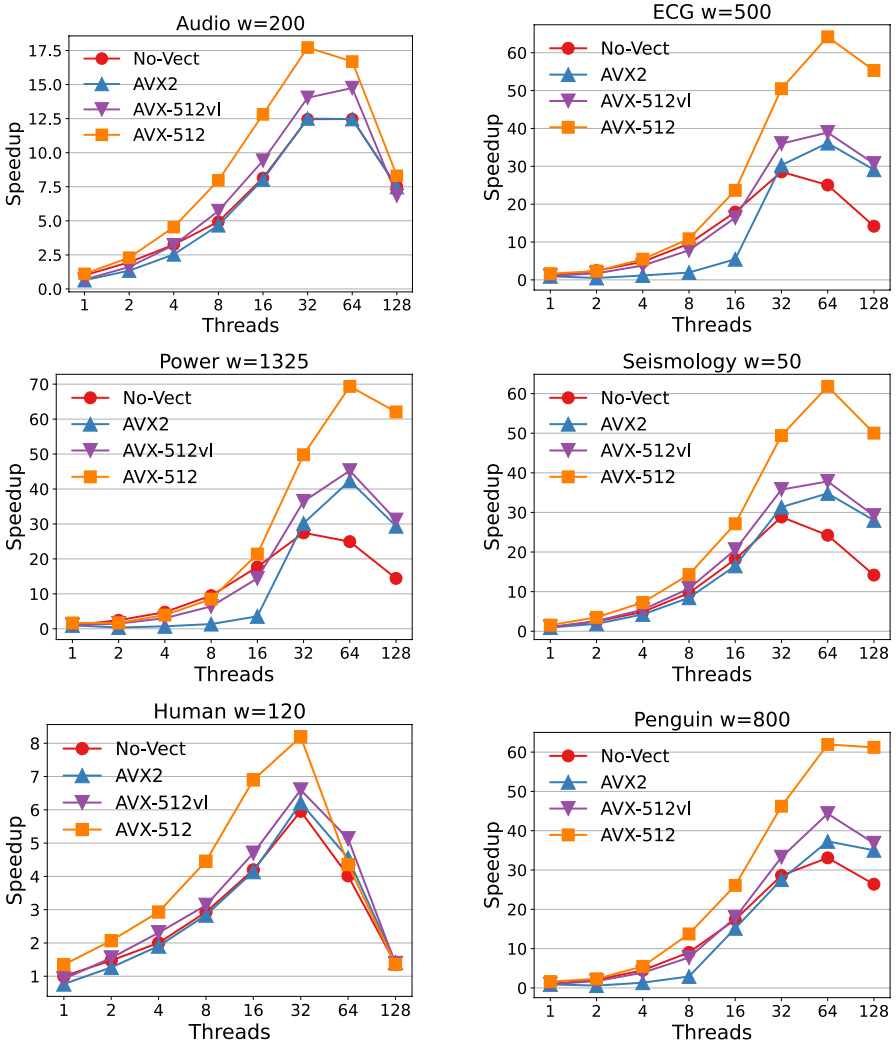

Fig. 15 SCAMP results. 128 threads implies hyperthreading

AVX-512vl implementations (which lacks the masking problem) scale very good, with speedups peaking up to 100× the serial. The problem of load imbalance seems to disappear with larger series as there are many more (meta)-diagonals for threads to work with, so the OpenMP on-demand dynamic scheduler makes a good job balancing the load.

It is worth to be noted the speedup increase of SCAMP over SCRIMP, as the latter gets less improvement by vectorization on the dot product loop (see next section). Also, SCRIMP needs for the division instruction, which poses a strain on the computation with a latency of 23 cycles, over 4 cycles for the multiplication on Skylake.

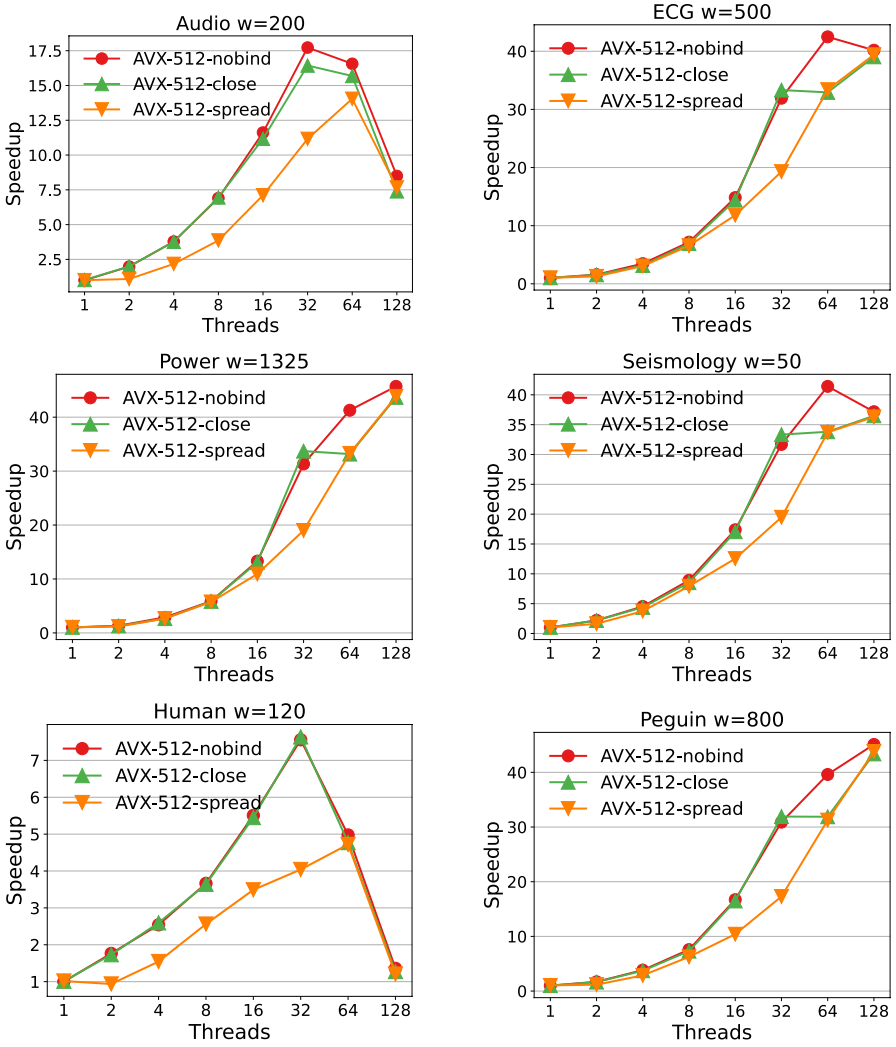

Fig. 16 SCRIMP thread affinity results using AVX-512

5.7 Roofline after vectorization

In Sect. 2, we show the roofline for the scalar versions of SCRIMP and SCAMP as a motivation to this work. In this section, we want to check the roofline after vectorization. For that purpose, Fig. 20 shows the results of the AVX-512 implementations of the algorithms. The green dots correspond to the loop which computes the dot product in SCRIMP (line 6 in Fig. 6) and the covariance in SCAMP (Fig. 11), and the blue dots correspond to the loop computing the rest of the diagonal (line 13 in Fig. 6).

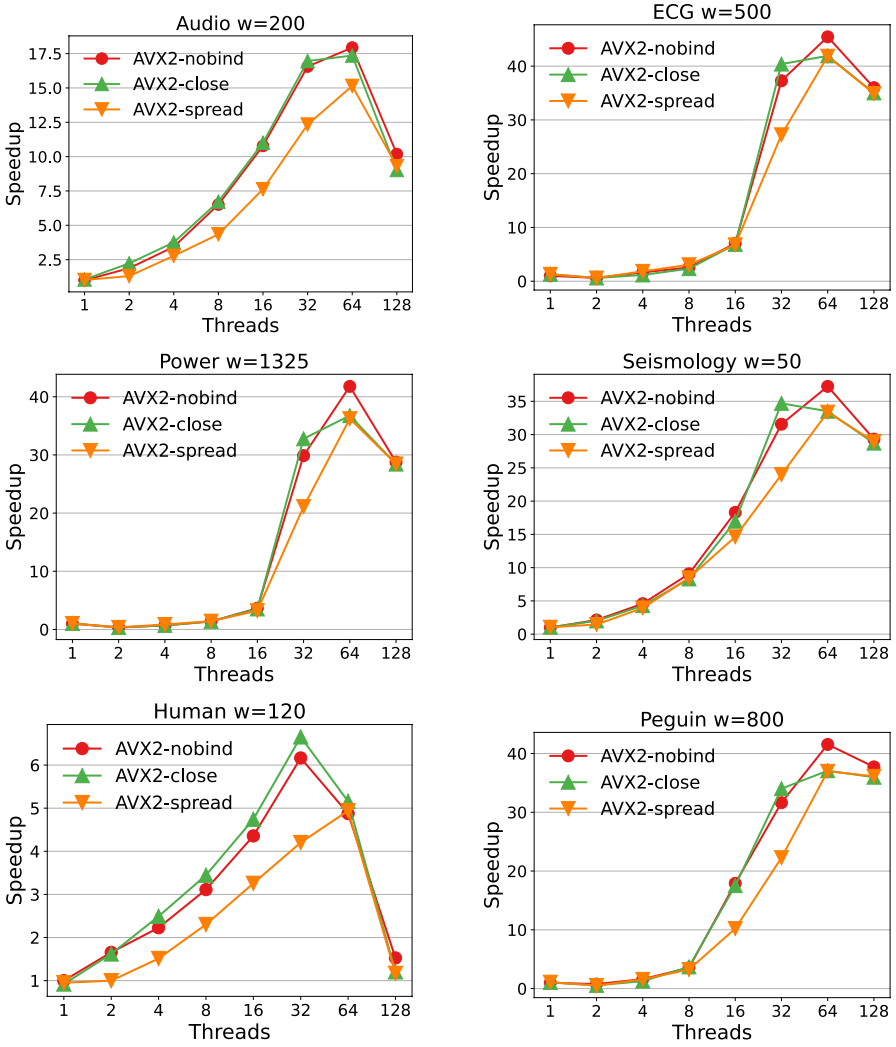

Fig. 17 SCRIMP thread affinity results using AVX2

As the diagonals have been packed into meta-diagonals (the thread unit of work) whose data is contiguous in memory, caches are better harnessed than the scalar counterpart which can assign a diagonal to a thread and the next diagonal to another thread. Furthermore, in the scalar implementation, the dot product/covariance computation (green dots) of two subsequences uses each element of the subsequences once. The vectorization makes that the reuse of those elements increase up to VPACK times, thus increasing the number of operations per second. SCAMP gets an improved behavior of vectorization on that loop since more arithmetic operations are performed over the same data: the subtraction of `means[0]` (see Fig. 11) is

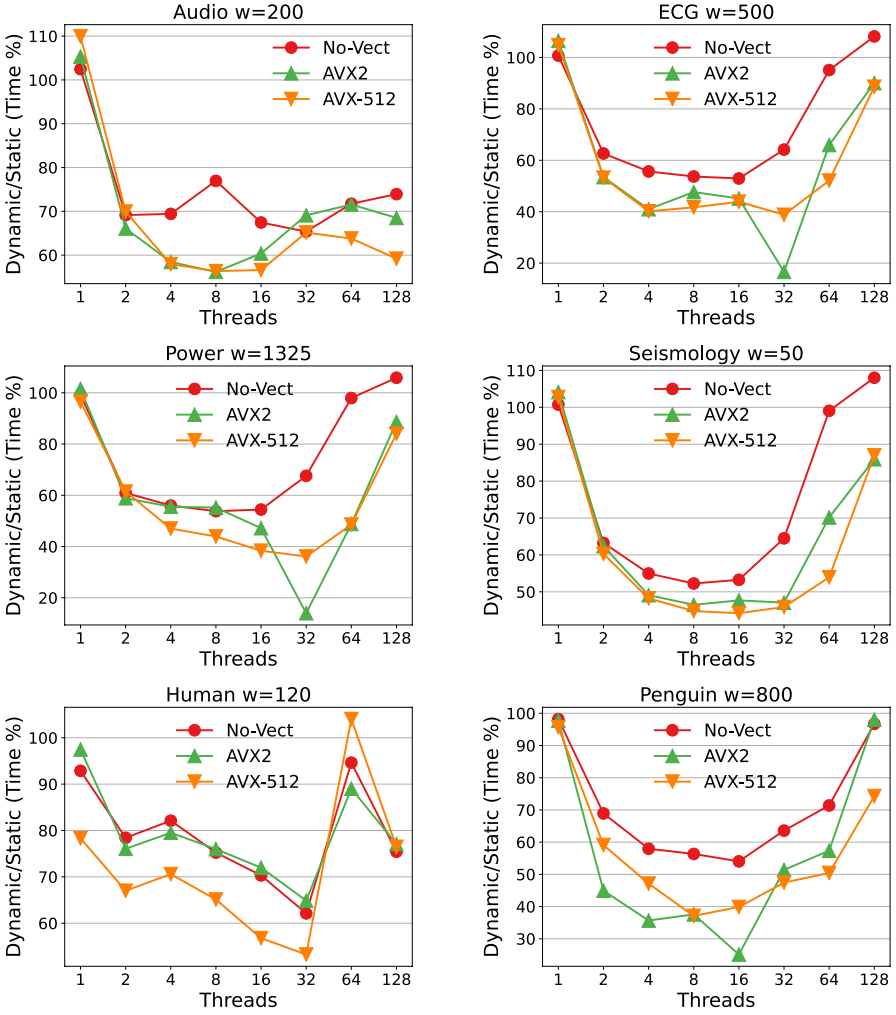

Fig. 18 SCAMP results with static loop partitioning with respect to dynamic

now done VPACK times, whereas SCRIMP does not need such a subtraction for the dot product computation.

6 Related work

We can find multiple techniques for time series data mining in the literature. Specifically, Chandola et al. wrote a survey on different techniques for anomaly detection on time series [27] up to 2009. The most recent techniques are based on deep learning [28], machine learning [29], neural networks [30] and motif discovery [10].

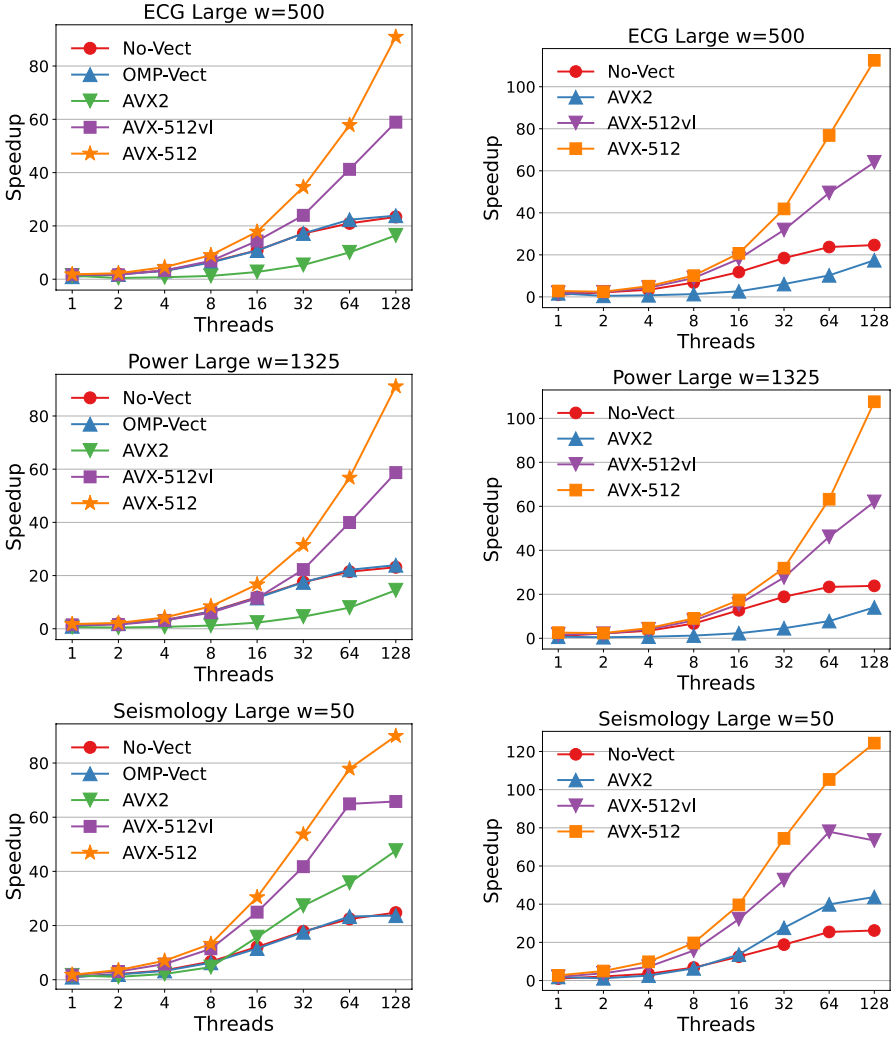

Fig. 19 Long series results

According to most recent works, there are two major approaches for time series motif discovery: approximate algorithms [31] and exact algorithms [10], such as matrix profile. Approximate algorithms usually take less execution time than exact ones. However, for large time series, an approximate algorithm can provide inaccurate results as they are based on probabilistic assumptions. As a result, the user has to set several (not intuitive) parameters, trying to get results accordingly to the expected ones, a fact which is not always possible.

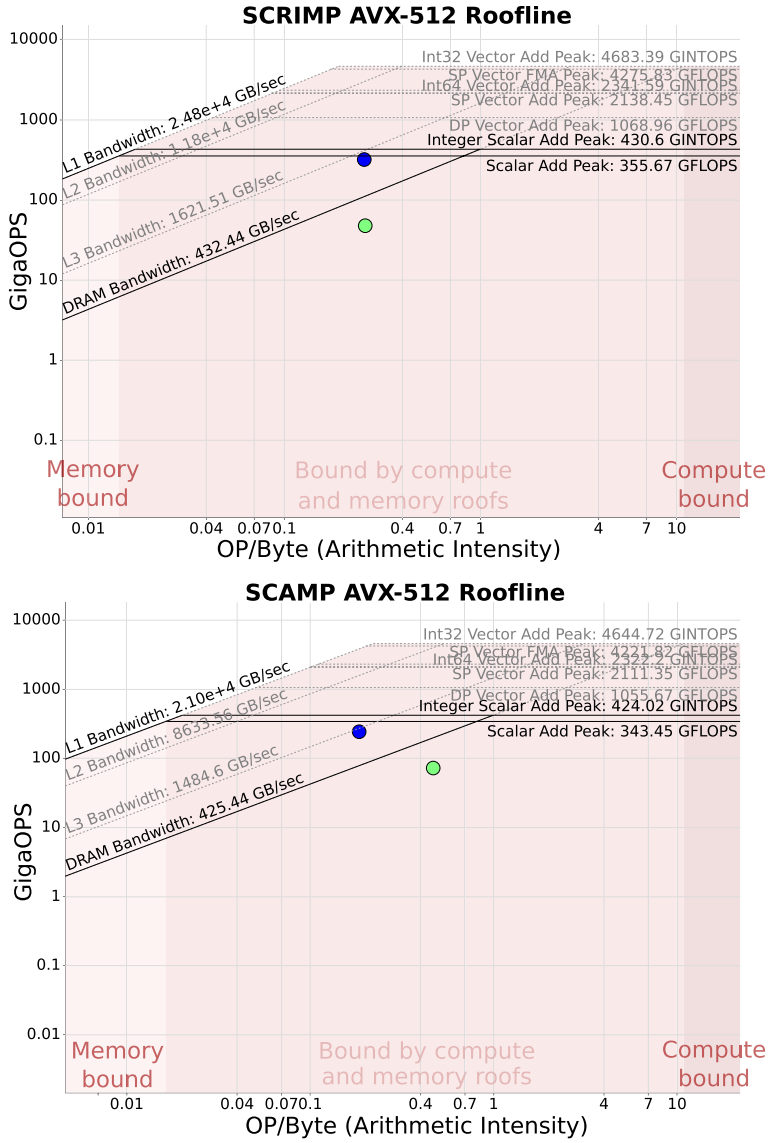

Fig. 20 SCRIMPvect and SCAMPvect AVX-512 rooflines (INT+FLOAT) for the Power time series with 180,000 samples, 1325 window size, and 128 threads

In this work, we focus our attention on the matrix profile technique and, in particular, the SCRIMP [11] and SCAMP [12] algorithms. Recently presented in literature, there are three major different implementations of the matrix profile: STAMP [10], STOMP [32] and SCRIMP [11]. The SCRIMP algorithm combines the best properties of STAMP, which computes the diagonals of the matrix in random order, with STOMP, which is faster than STAMP but computes the diagonals

in sequential order. By the time of writing this paper, SCAMP is the algorithm that offers better results.

In addition to the matrix profile, we can find other approaches based on time series motif discovery, such as a probabilistic solution [33], spatio-temporal models [4], an indexing solution [34] and a symbolic representation [35].

Not many works can be found accelerating time series processing algorithms using SIMD approaches. For instance, in [36] the Xeon Phi KNC [37] (Knights Corner - a previous version of the Xeon Phi architecture) is used to speed up the Dynamic Time Warping (DTW) technique to measure the similarity between time series. However, their approach is limited by a master-worker scheme, and this target architecture, being older, does not include ultra high bandwidth 3D-stacked memory modules. A SCRIMP version tuned for a many-core CPU (Intel Xeon Phi KNL) using auto-vectorization can be found in [17]. We find multiple efforts to increase time series analysis efficiency and performance based on commodity CPU architectures, high-performance GPUs, heterogeneous systems and even Processing-Near-Memory [38] in [12, 17, 32, 39, 40], but they overlook the use of advanced vectorization extensions.

This work is an extension of [41] with the following contributions: (i) we detect a problem with AVX2 masking instructions and propose a solution by using AVX-512 instructions with 256-bit registers; (ii) we perform and discuss a thread affinity and load balance exploration; (iii) we analyze CPU/memory rooflines for the algorithms before and after vectorization; and, (iv) we carry out a thorough experimental analysis highlighting the results obtained for large series.

7 Conclusions and future work

In this work we take advantage of various SIMD optimization techniques for modern high performance processors, with the aim of improving the performance of time series analysis algorithms, specifically, the state-of-the-art matrix profile ones.

As a base, we use the OpenMP framework to use the multiple available hardware threads. Then, we further optimize our implementations via custom data level parallelism (SIMD) with various vector sizes. Specifically, we use Advanced Vector Extensions 2 (AVX2) and AVX-512. We also highlight the problem of masking with AVX2, which has been dramatically improved by predicate AVX-512 instructions with opmask registers. We explore thread-affinity and load balance to find that the OS is aware of NUMA and performs the best thread-affinity policy, and that the algorithms are highly imbalanced, thus taking advantage of the dynamic scheduling policy of OpenMP. Vectorization packs diagonals into meta-diagonals hindering the load balance even more. But this problem disappears with large series.

The algorithms involved get rather significant speedups out of our SIMD proposals, specially when it comes to AVX-512, with up to 4× speedup over the scalar implementation. The AVX2 implementation is in general as good as or better than the code automatically vectorized by the OpenMP framework, as long as the window size is not large so that the TLB is not stressed. The use of AVX-512

instructions with 256-bit registers yields better results among AVX-512 and AVX2, since the masking problem is addressed.

Acknowledgements This work has been supported by the Government of Spain under project PID2019-105396RB-I00, and Junta de Andalucía under projects P18-FR-3433, and UMA18-FEDERJA-197.

Author contributions All authors have contributed equally to this work.

Funding Funding for open access publishing: Universidad Málaga/CBUA. Not applicable.

Data availability Data sets used can be accessed from [25].

Declarations

Conflict of interest Not applicable.

Ethical approval Not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Shumway RH, Stoffer DS (2017) Time series analysis and its applications, 4th edn. Springer, Berlin, Heidelberg
2. Chiu B, Keogh E, Lonardi S (2003) Probabilistic discovery of time series motifs. In: KDD
3. Keogh E, Lin J, Lee S-H, Van Herle H (2006) Finding the most unusual time series subsequence: algorithms and applications. Knowl Inf Syst 11:1–27
4. McGovern A, Rosendahl DH, Brown RA, Droegemeier KK (2011) Identifying predictive multi-dimensional time series motifs: an application to severe weather prediction. Data Min Knowl Discov 22:232–258
5. Cassisi C, Aliotta M, Cannata A, Montalto P, Patanè D, Pulvirenti A, Spampinato L (2013) Motif discovery on seismic amplitude time series: the case study of mt etna 2011 eruptive activity. Pure Appl Geophys 170:529–545
6. Szigeti B, Deogade A, Webb B (2015) Searching for motifs in the behaviour of larval Drosophila melanogaster and Caenorhabditis elegans reveals continuity between behavioural states. J R Soc Interface 12:20150899
7. Garrard P, Nemes V, Nikolic D, Barney A (2017) Motif discovery in speech: application to monitoring Alzheimer's disease. Curr Alzheimer Res 14:951–959
8. Torkamani S, Lohweg V (2017) Survey on time series motif discovery. Wiley Interdis Rev Data Min Knowl Discov 7:e1199
9. Cartwright E, Crane M, Ruskin HJ (2019) Financial time series: motif discovery and analysis using VALMOD. In: ICCS
10. Yeh C-CM, Zhu Y, Ulanova L, Begum N, Ding Y, Dau HA, Silva DF, Mueen A, Keogh E (2016) Matrix Profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In: KDD

11. Zhu Y, Yeh C-CM, Zimmerman Z, Kamgar K, Keogh E (2018) Matrix profile XI: SCRIMP++: time series motif discovery at interactive speeds. In: KDD
12. Zimmerman Z, Kamgar K, Senobari NS, Crites B, Funning G, Brisk P, Keogh E (2019) Matrix profile XIV: scaling time series motif discovery with GPUs to break a quintillion pairwise comparisons a day and beyond. In: SoCC
13. Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R (2001) Parallel programming in openMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
14. Intel®Advisor User Guide. Analyze CPU roofline. https://www.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/analyze-cpu-roofline.html. Accessed July (2022)
15. Feautrier P (1988) Array expansion. In: 2nd ACM International Conference on Supercomputing (ICS), pp 429–441
16. Gutiérrez E, Plata O, Zapata EL (2000) A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In: 14th International Conference on Supercomputing (ICS), pp 78–87
17. Fernandez I, Villegas A, Gutierrez E, Plata O (2019) Accelerating time series motif discovery in the Intel Xeon Phi KNL processor. J Supercomput 75:7053–7075
18. Peleg A, Wilkie S, Weiser U (1997) Intel MMX for multimedia PCs. Commun ACM 40(1):24–38
19. Oberman S, Favor G, Weber F (1999) AMD 3DNow! technology: architecture and implementations. IEEE Micro 19(2):37–48
20. Ma W-C, Yang C-L (2002) Using Intel streaming SIMD extensions for 3D geometry processing. In: Pacific-Rim Conference on Multimedia, pp 1080–1087. Springer
21. Lomont C (2011) Introduction to intel advanced vector extensions. Intel White Paper 23
22. Intel®64 and IA-32 Architectures Software Developer's Manual Volume 2C: Instruction Set Reference, V-Z. https://cdrdv2.intel.com/v1/dl/getContent/671072. Accessed July (2022)
23. Intel Xeon Gold 5218. https://ark.intel.com/content/www/es/es/ark/products/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz.html. Accessed July (2022)
24. Cascade Lake - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake. Accessed July (2022)
25. The UCR Matrix Profile Page. https://www.cs.ucr.edu/~eamonn/MatrixProfile.html. Accessed July (2022)
26. Intel®64 and IA-32 Architectures Optimization Reference Manual. https://cdrdv2.intel.com/v1/dl/getContent/671488. Accessed July (2022)
27. Chandola V, Banerjee A, Kumar V (2009) Anomaly detection: a survey. ACM Comput Surv 41(3):15
28. Kanarachos S, Christopoulos S-RG, Chroneos A, Fitzpatrick ME (2017) Detecting anomalies in time series data via a deep learning algorithm combining wavelets, neural networks and Hilbert transforms. Expert Syst Appl 85:292–304
29. Lee TJ, Gottschlich J, Tatbul N, Metcalf E, Zdonik S (2018) Greenhouse: a zero-positive machine learning system for time-series anomaly detection. arXiv preprint arXiv:1801.03168
30. Wu J, Zeng W, Yan F (2018) Hierarchical temporal memory method for time-series-based anomaly detection. Neurocomputing 273:535–546
31. Torkamani S, Lohweg V (2017) Survey on time series motif discovery. Wiley Interdis Rev Data Min Knowl Discov 7(2):1199
32. Zhu Y, Zimmerman Z, Senobari NS, Yeh C-CM, Funning G, Mueen A, Brisk P, Keogh E (2016) Matrix Profile II: exploiting a novel algorithm and GPUs to break the one hundred million barrier for time series motifs and joins. In: International Conference on Data Mining (ICDM'16), pp 739–748
33. Chiu B, Keogh E, Lonardi S (2003) Probabilistic discovery of time series motifs. In: 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 493–498
34. Yi B-K, Faloutsos C (2000) Fast time sequence indexing for arbitrary Lp norms. In: 26th International Conference on Very Large Data Bases (VLDB), pp 385–394
35. Lin J, Keogh E, Wei L, Lonardi S (2007) Experiencing SAX: a novel symbolic representation of time series. In: KDD
36. Miniakhmetov R, Movchan A, Zymbler M (2015) Accelerating time series subsequence matching on the Intel Xeon Phi many-core coprocessor. In: International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp 1399–1404
37. Chrysos G (2012) Intel Xeon Phi coprocessor (codename Knights Corner). In: Hot Chips 24 Symposium (HCS'12), pp 1–31
38. Mutlu O, Ghose S, Gómez-Luna J, Ausavarungnirun R (2020) A modern primer on processing in memory. arXiv:2012.03112 [cs.AR]

39. Romero JC, Vilches A, Rodríguez A, Navarro A, Asenjo R (2020) ScrimpCo: scalable matrix profile on commodity heterogeneous processors. J Supercomput 76:9189–9210
40. Fernandez I, Quislant R, Gutiérrez E, Plata O, Giannoula C, Alser M, Gómez-Luna J, Mutlu O (2020) NATSA: a near-data processing accelerator for time series analysis. In: International Conference on Computer Design (ICCD)
41. Quislant R, Fernandez I, Serralvo E, Gutierrez E, Plata O (2022) Exploiting vector extensions to accelerate time series analysis. In: Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp 55–62

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.