Journal of Computational Science 74 (2023) 102180

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Efficient GPU implementation of the multivariate empirical mode decomposition algorithm

Zeyu Wang [*], Zoltan Juhasz

Department of Electrical Engineering and Information Systems, University of Pannonia, Egyetem u. 10, Veszprem 8200 Hungary

ARTICLE INFO

Keywords:
Multivariate empirical mode decomposition
GPU
CUDA
EEG

ABSTRACT

An efficient GPU implementation of the Multivariate Empirical Mode Decomposition (MEMD) method is presented for speeding up the process of decomposing non-stationary multi-channel bioelectric signals into different oscillation modes. Each step of the MEMD algorithm is designed with performance in mind and implemented to remove all unnecessary overheads caused by CPU-GPU communication, data transfer operations and synchronisation. The implementation is validated with synthetic and real EEG signals of different lengths and channels (up to 128 channels) on different GPU cards, and compared to existing serial MEMD implementations. The final implementation achieved between 180x-430x speedup compared to MATLAB and a 10x improvement over the only known existing GPU implementation. The average decomposition error of our implementation is below 1.2 %. Our GPU program is the fastest known GPU implementation of the MEMD algorithm that reduces execution time from hours to seconds and as such makes it possible to perform MEMD time-frequency analysis of high-density EEG (MEG) or similar multi-channel signals in a fraction of time and opens the road towards its practical applicability.

1. Introduction

Multivariate Empirical Mode Decomposition (MEMD) is a recently introduced method for analysing the spatiotemporal dynamics of multivariate signals [1] based on Huang's Empirical Mode Decomposition (EMD) [2] proposed for time-frequency analysis of natural signals. The key difference between EMD-based and traditional time-frequency analysis approaches (such as FFT and Wavelet decomposition) is that EMD is a data-driven, adaptive method that does not rely on a set of predetermined basis functions, and its basis functions (Intrinsic Mode Functions, IMFs) are derived automatically from the data itself.

While EMD and MEMD have been used successfully in many application domains (e.g. geology, earthquake monitoring, astronomy, man-built structure monitoring, machine vibration analysis), it is especially suited to multi-sensor biosignal analysis. In this paper, we focus on its efficient use for analysing electroencephalography (EEG) data. EEG registers scalp potential variations generated by neural sources of the brain. Neuroscience research identified and confirmed that communication between different cortical areas is facilitated by oscillations [3], of whose amplitude and frequency may be modulated by other underlying processes or conditions. Consequently, accurate detection of ongoing oscillations is a key step in many EEG signal-processing analyses. Being the result of natural processes in the brain, EEG signals, however, do not satisfy the stationarity and periodicity conditions required for Fourier or Wavelet transform based time-frequency analysis. Moreover, these two transforms suffer from the time-frequency uncertainty principle, which means we cannot achieve high temporal and frequency resolution at the same time. In addition, these methods fail to uncover amplitude and frequency modulations of the extracted oscillatory basis functions.

EMD presents a special opportunity for the EEG community as it can decompose a wide-band EEG signal into several narrow band IMFs that carry frequency and amplitude modulation information and provide instantaneous frequency and phase information at every time step [4]. With EMD, we may be able to extract information that previously stayed hidden from us during analysis, hence it can become the tool helping researchers to understand the mesoscopic behaviour and dynamics of the brain.

Several studies used the original EMD algorithm or one of its variants for analysing EEG signals [5–9]. Despite promising results, there are still

* Corresponding author.
E-mail address: zeyu.wang@mik.uni-pannon.hu (Z. Wang).

https://doi.org/10.1016/j.jocs.2023.102180
Received 28 December 2022; Received in revised form 1 November 2023; Accepted 7 November 2023
Available online 10 November 2023

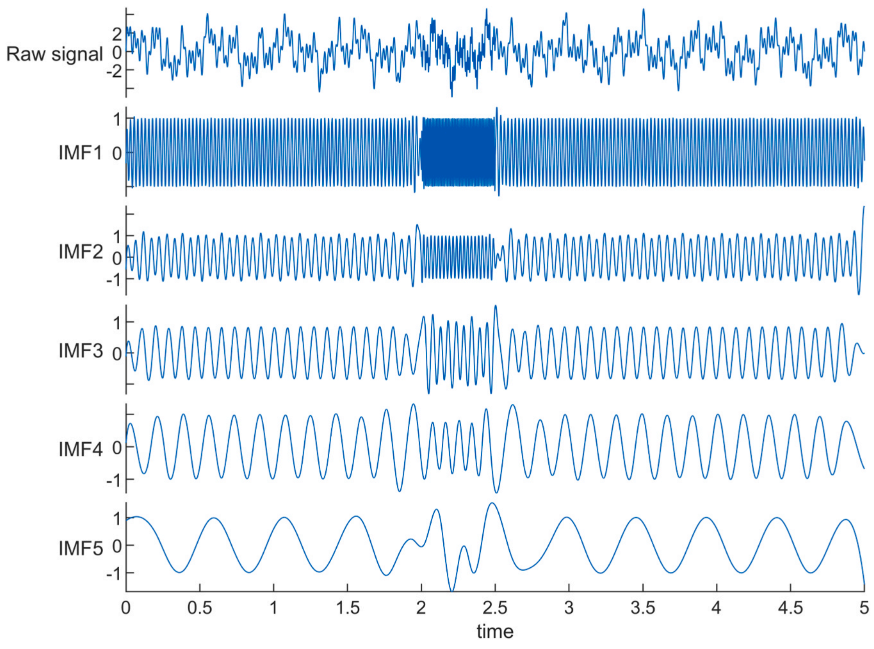

Fig. 1. The original synthetic signal and its decomposed IMFs. Mode mixing is evident around 2–2.5 s.

several methodological and technical obstacles that limit the widespread application of these techniques. The original EMD method is very sensitive to noise, hence noise-assisted extensions such as Ensemble Empirical Mode Decomposition (EEMD) [10] and Complete Ensemble Empirical Mode Decomposition with Assisted Noise (CEEMDAN) [11] were recommended as improvements. Unfortunately, both variants result in a two orders of magnitude increase in execution time. As a further problem, EMD and its variants are univariate methods, thus they decompose signals to IMFs in isolation (i.e. single electrode only). EEG, however, is a multivariate signal, recorded by several electrodes simultaneously. If a single channel EMD (or EEMD/CEEMDAN) is executed independently on the different signals of the multi-electrode EEG dataset, there is no guarantee that the same number of IMFs will be extracted from each signal and with identical central frequencies. Consequently, we will not be able to infer correct inter-electrode relationships that are mandatory for spatiotemporal spectral analysis and brain connectivity network calculation.

The multivariate extension (MEMD) of EMD provides the necessary mechanism to treat a multi-electrode EEG measurement as a multivariate signal and decompose each electrode into an identical number of IMFs with matching central frequencies. The main drawback of this method is its prohibitively large execution time. To achieve practical usability, 2–3 orders of magnitude speedup is required, which can only be achieved economically with GPU accelerators. State-of-the-art GPUs provide massively parallel execution capabilities with exceptional efficiency reaching computational peak performance up to 10–40 TFlop/s. We assume that GPU computing, GPU architecture and their fundamental programming concepts are familiar to the readers of this journal. For further technical details, we refer the readers to the research literature, and various other sources of programming and hardware documentation.

In this paper, we describe an efficient, high-performance parallel CUDA implementation of the MEMD algorithm that (i) can be executed on NVIDIA GPUs and (ii) reduce the execution time from hours to seconds. To the best of our knowledge, this is the second known MEMD GPU implementation, and the first one that provides publicly available source code[1] and demonstrates exceptional performance up to 128 EEG channels.

The structure of the paper is as follows. Section 2 introduces the Empirical Mode Decomposition, the Ensemble Empirical Mode Decomposition and the Multivariate Empirical Mode Decomposition methods, followed by an overview of existing works in the parallel implementations of EMD and its variants. Section 3 describes the parallel implementation strategy for the MEMD method and provides details of the CUDA GPU implementation focusing on performance-oriented design. Section 4 presents the results of our work in terms of implementation accuracy, execution time, speedup and performance analysis. The paper ends with the Conclusions.

2. Related work

The frequency range of interest of EEG measurements is usually between 0.1 and 100 Hz. Based on historical developments and physiological evidence, this frequency range is partitioned into distinct characteristic frequency bands, namely into the delta (1–4 Hz), theta (4–8 Hz), alpha (8–13 Hz), beta (13–30 Hz) and gamma (30–150 Hz) bands. While power changes of these bands might reveal important diagnostic information, to better understand the cortical processes underlying the resting state and task execution mechanisms of the human brain, time-frequency analysis is required.

Traditionally, the Short-Time Fourier Transform (STFFT) and the Continuous Wavelet Transform (CWT) using the Morlet wavelet family are the most widespread methods for time-frequency analysis [12]. Both methods assume pre-determined basis functions and rely on signal stationarity [13]. The EEG signal – as most natural signals – is non-periodic and non-stationary, violating our basic assumptions. In addition, the exact time-localization of cortical events is difficult due to the time-frequency uncertainty principle.

Several new techniques have been proposed over the past two decades attempting to overcome some or all of the above limitations, namely the synchrosqueezed Fourier [14–16], and synchrosqueezed wavelet transforms [17], Empirical Mode Decomposition [2], Variational Mode Decomposition[18], and Singular Spectrum Analysis [19]. Of these methods, Empirical Mode Decomposition and its variants are used most frequently in EEG research, therefore we focus here on this method only.

[1] https://github.com/EEGLab-Pannon/MEMD-GPU

2.1. Empirical mode decomposition

Empirical Mode Decomposition (EMD) is a data-driven signal decomposition algorithm [2] that can separate a signal into a finite number of so-called Intrinsic Mode Functions (IMFs) [4]. IMFs are narrow band signals that contain only one dominant oscillatory mode of the signal. The advantages of EMD over the Fourier of Wavelet transforms are that (i) the method can be used without a pre-determined set of basis functions, (ii) the extracted narrow band oscillatory modes (IMFs) carry amplitude and frequency modulation information and (iii) can be used to extract instantaneous frequency and phase information. The EMD algorithm has a filter-bank property [20] and as a result, the signal can be easily analysed in a multi-resolution fashion.

The EMD algorithm automatically extracts the intrinsic mode functions from the signal starting with the highest frequency components and progressing to the lower frequencies. The exact steps of the algorithm are listed in Algorithm 1. The first step of the decomposition process is the detection of the extrema (minima and maxima) of the input signal. The extreme points are used to generate the upper and lower envelopes of the signal by using cubic spline interpolation. Next, the mean envelope is calculated from the upper and lower envelopes and subtracted from the original signal, creating a residual signal. This residual is regarded as a potential IMF. A proper IMF should satisfy the following two conditions; (i) the number of extreme points and the number of zero-crossings must be equal or the difference should not exceed one, (ii) the mean value of the mean envelope should be approximately zero [2]. Since these two conditions are difficult to satisfy simultaneously, usually the standard deviation, *SD*, between two residues is used as the stopping criterion of the sifting process:

$$SD = \sum_{k=0}^{K} \frac{|R_{k-1}(t) - R_k(t)|^2}{(R_{k-1}(t))^2} < \varepsilon$$

where R_{k-1} and R_k are the final residual signal in the sifting iteration $k-1$ and k, respectively, and ε is the sifting iteration threshold. If residues of two subsequent iterations are identical within ε, R_k will be regarded as a proper IMF. This IMF is then subtracted from the input signal to create the new input signal for the next iteration of the algorithm that extracts the subsequent lower frequency IMF. The process stops when no further oscillatory IMFs can be extracted or the number of IMFs reaches a pre-set limit.

Algorithm 1. **EMD**: Empirical Mode Decomposition [2].

```
Input: x(k) – single-channel time series
Output: IMF(k) – N extracted Intrinsic Mode Function time series
  1.  set config
  2.  i = 0
  3.  while (IMF stopping criterion is not met)
  4.    create a working copy of the input signal: x'(k) = x(k)
  5.    while (sifting stopping criterion is not met)
  6.      find extrema locations of x'(k)
  7.      perform cubic spline interpolation on the extrema to obtain the upper and
                lower envelope of the working copy of the signal
  8.      compute the mean of the upper and lower envelopes: m(k) = (upper(k) + lower(k)) / 2
  9.      subtract the mean envelope from the working copy: s(k) = x'(k) – m(k)
  10.     x'(k) = s(k)
  11.   end
  12.   IMF[i](k) = s(k)
  13.   x(k) = x(k) - IMF[i] (k)
  14.   i = i + 1
  15. end
```

Once the decomposition is complete, the original signal can be represented as:

$$X(t) = \sum_{i=1}^{N} IMF_i(t) + R_N(t)$$

where $R_N(t)$ is the final residue that can be considered as the global trend signal.

To illustrate the operation of the EMD algorithm we show the decomposition of a synthetic signal X containing five sine waves (x_σ, x_θ, x_α, x_β, x_γ) of frequencies representing each EEG frequency band.

$$\begin{cases} x_\sigma(\mathrm{t}) = \sin(2\pi * 2.1 * t) \\ x_\theta(\mathrm{t}) = \sin(2\pi * 5.8 * t) \\ x_\alpha(\mathrm{t}) = \sin(2\pi * 11.2 * t) \\ x_\beta(\mathrm{t}) = \sin(2\pi * 18.9 * t) \\ x_\gamma(\mathrm{t}) = \sin(2\pi * 40.3 * t) \end{cases}$$

$$X(t) = x_\sigma(\mathrm{t}) + x_\theta(\mathrm{t}) + x_\alpha(\mathrm{t}) + x_\beta(\mathrm{t}) + x_\gamma(\mathrm{t})$$

The result of the decomposition process of a synthetic signal is shown in Fig. 1. The top row shows the input signal, while the other rows display the extracted IMFs in order of decreasing central frequency. Notice that IMFs 1–6 contain multiple frequencies (oscillation modes) within a given IMF. This phenomenon is called mode mixing caused by signal noise that changes the location of the signal extrema and disturb the decomposition process [21]. The need to solve the mode mixing problem has led to the development of several variants of the EMD algorithm that we describe briefly in the next subsection.

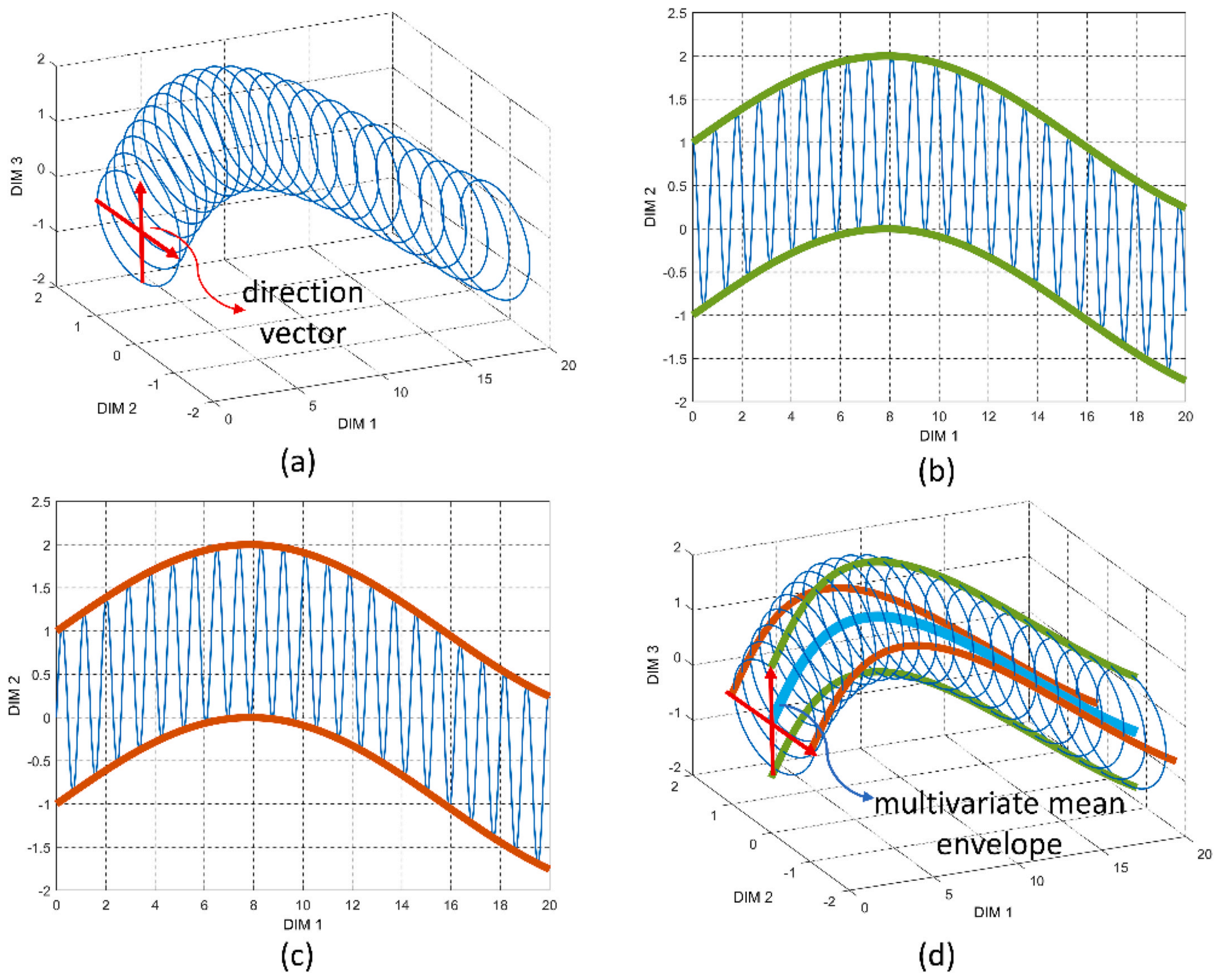

Fig. 2. The principle of extreme point detection in a multivariate signal. (a) Two direction vectors used for projecting the multivariate signal. (b, c) The projected signals and their upper and lower envelopes produced along the two direction vectors. (d) The mean multivariate envelope (blue line) produced by averaging all multivariate upper and lower envelopes.

Table 1
CUDA kernels used in different stages in the MEMD GPU implementation.

Steps in MEMD	Kernel function (s)	Description
Preprocess	generateHammSeq()	Generate Hammersley sequence from primes
	generateDirVec()	Generate direction vectors from Hammersley sequence
Signal projection	cublasSgemm()	Multiply the input signal and the direction vectors
Extrema detection	findExtremaShfl()	Detect the location of extreme points
	scanLargeDeviceArray()	Generate index of compact vector
	scanSmallDeviceArray()	Generate index of compact vector
	selectExtremaMax/Min()	Generate compact extrema vector
	setBoundary()	Set the boundary condition
Cubic spline interpolation	tridiagonalSetup()	Generate tridiagonal system
	cusparseSgtsv2()	Solve the tridiagonal system
	splineCoefficients()	Generate spline coefficients for gaps
	interpolate()	Generate upper and lower envelopes
Envelopes averaging	averageUppperLower()	Generate multivariate mean envelope for direction vectors
	averageDirection()	Generate the multivariate mean envelope of input signal
Signal updating	updateSignal()	Subtract the mean envelope to generate new input signal

2.2. Improvements of the EMD algorithm

To solve the mode mixing problem, Wu et al. [22] proposed a noise-assisted signal decomposition method called Ensemble Empirical Mode Decomposition (EEMD). This algorithm uses multiple copies (called realizations) of the input signal created by adding random Gaussian noise to the signal before the decomposition process. As a result, the distribution of extreme points of the signal will be more uniform in a statistical sense and become less sensitive to intermittent noise. The number of realizations in EEMD is a problem-dependent configuration parameter but, in general, it is in the order of few hundreds. The execution flow of the EEMD method is depicted in Algorithm 2. Compared to EMD, an additional loop is required for the realizations where each iteration starts with replicating the signal by adding to it some random white noise. Then, these signals will be decomposed individually as described in the EMD section. Once the IMFs for each noisy copy are extracted, they are averaged to generate the resultant final IMF set.

Algorithm 2. **EEMD**: Ensemble empirical mode decomposition.

```
Input: x(k) – single-channel time series
Output: IMF(k) – N extracted Intrinsic Mode Function time series
    1.  set configuration parameters
    2.  IMF = 0
    3.  R = number of realizations
    4.  for (r = 1 to R)
    5.      create a working copy of the input signal: x'(k) = x(k) + noise(k)
    6.      IMF[r] = EMD(x')
    7.  end
    8.  IMF = average(IMF)
```

One disadvantage of the EEMD method is that due to the added noise, the signal reconstructed from the IMFs is not identical to the original signal, slight noise will appear after the reconstruction. A solution to this problem is given by the improved Complete Ensemble Empirical Mode Decomposition with Assisted Noise (CEEMDAN) [11]. We skip the details of this algorithm as it is not the focus of this paper. The interested readers are referred to the reference for further details.

2.3. Multivariate empirical mode decomposition

The EEMD and CEEMDAN algorithms provide reliable decompositions for single channel signals. EEG, however (just as many other multi-sensor application datasets), is normally not a single channel – univariate – signal, but obtained using a large number of electrodes simultaneously. If we apply EMD, EEMD or CEEMDAN on such datasets in a channel-by-channel fashion, different channels may produce different number of IMFs with potentially different central frequencies. This is called the mode alignment problem that can present serious difficulties during group level, spatiotemporal or connectivity analysis. The Multivariate Empirical Mode Decomposition (MEMD) proposed by Rehman et al. [1] provides a solution to this problem.

In MEMD, the multi-channel signal is regarded as a multivariate signal in a high-dimensional space. Similarly to EMD, we still need to calculate the extreme points and mean envelope and perform the IMF sifting process, but for multivariate signals, the concept of extreme points is not well-defined, as the occurrence of extreme points depends on the projection of the signal to a particular dimension. Fig. 2 shows a simple example of the multivariate signal extreme point detection, in which we project the multivariate signal onto two different directions. The projected signals then are used in the extreme point detection step and result in two sets of upper and lower envelopes. These will be averaged to find the mean and subsequently the multivariate IMF.

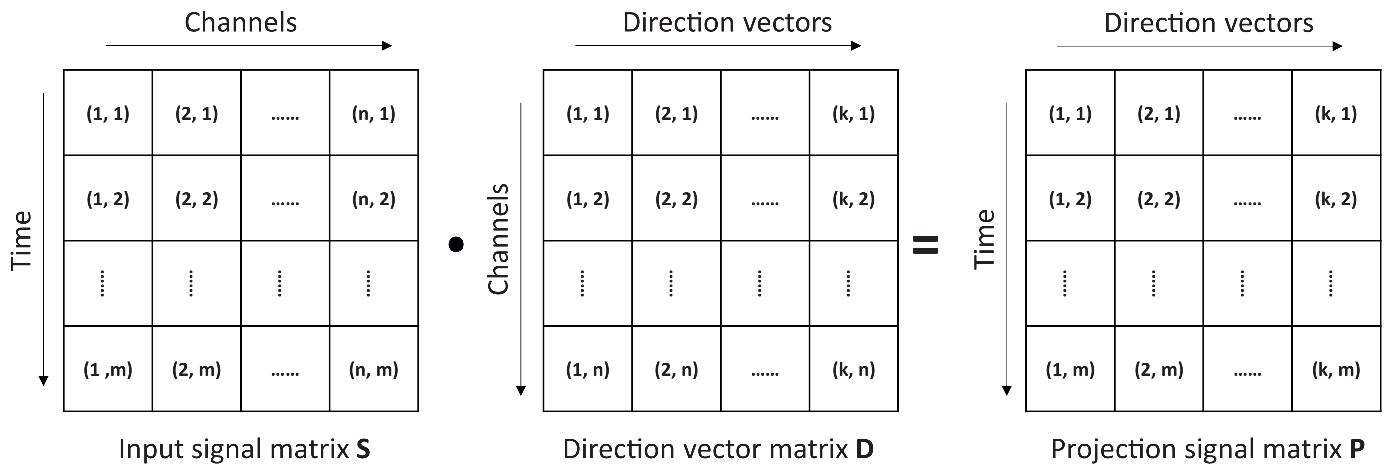
Channels
Time
(1, 1) (2, 1) ...... (n, 1)
(1, 2) (2, 2) ...... (n, 2)
(1 ,m) (2, m) ...... (n, m)
Input signal matrix S
Direction vectors
Channels
(1, 1) (2, 1) ...... (k, 1)
(1, 2) (2, 2) ...... (k, 2)
(1, n) (2, n) ...... (k, n)
Direction vector matrix D
Direction vectors
Time
(1, 1) (2, 1) ...... (k, 1)
(1, 2) (2, 2) ...... (k, 2)
(1, m) (2, m) ...... (k, m)
Projection signal matrix P

Fig. 3. Computing the projected signals by matrix-matrix multiplication.

The use of the direction vectors and signal projection are the main differences between MEMD and univariate EMD variants. To achieve precise projections, the number of direction vectors must be as large as possible but at least twice the number of channels [23]. The direction vectors should also be uniformly distributed on a hypersphere. The Hammersley sequence, a low-discrepancy sequence that provides uniform angle sampling—also used by Rehman et al. [1]—can provide the proper basis for generating direction vectors with the desired properties.

The details of the MEMD algorithm are outlined in Algorithm 3. First, the multivariate input signal will be projected onto the direction vectors and we will get the projected signal on each direction vector. Then, extreme point detection is performed on each projected signal, and each detected extreme point corresponds to a multivariate extreme point in the input signal, that is, a point in a high-dimensional space. The next step is to perform cubic spline interpolation in each dimension to create the upper and lower multivariate envelopes. By averaging these envelopes we generate the multivariate mean envelope of the input signal. Finally, by subtracting the mean envelope from the input, we can obtain a candidate IMF. If this IMF meets the stopping criterion, the iteration will stop, otherwise, the candidate IMF will be used as the input signal for the next iteration.

Algorithm 3. **MEMD**: Multivariate Empirical Mode Decomposition.

```
Input: X – a multivariate, M-channel time series as an M x K matrix
Output: IMF – extracted multivariate Intrinsic Mode Function time series as an N x M x K matrix
 1.  set configuration parameters
 2.  i = 0;
 3.  generate a number of direction vectors based on Hammersley sequence D
 4.  while (IMF stopping criterion is not met)
 5.    compute the projection of the signal to the direction vectors: P = X · D
 6.    while (sifting stopping criterion is not met)
 7.      find extrema locations of the projected signal: p = extrema(P)
 8.      perform cubic spline interpolation on the input signal values indexed by p to
                 obtain the multivariate upper and lower envelopes U and L
 9.      compute the mean of the upper and lower envelopes for each direction:
                 M = mean(U, L)
10.      subtract the mean envelope from the working copy: S = X – M
11.      X = S
12.    end
13.    IMF[i] = S
14.    X = X - IMF[i]
15.    i = i + 1
16.  end
```

2.4. Sequential and GPU implementations

Increasing interest in the use of EMD-based methods over the past two decades have given rise to numerous implementations of Empirical Mode Decomposition and its variants using different programming languages and hardware architectures. Flandrin et al. released MATLAB implementation of the EMD, EEMD algorithms in 2007 and for the CEEMDAN algorithm in 2012 [11]. To help the work of the EEG signal analysis community, Al-Subari et al. developed the EMDLAB toolbox [24] as an extension plug-in for the EEGLAB MATLAB framework [25] widely used by the neuroscience community. Starting with version R2018a, MATLAB supports EMD calculation with the built-in emd() function. While MATLAB implementations are very suitable for EMD research and for quick integration into existing data-processing pipelines, these implementations are sequential in nature and their typical execution time (up to several hours or days for high-density EEG datasets[2]) is not acceptable for routine, production use. libeemd [26] is a library written in the C programming language that provides sequential and OpenMP-based parallel implementation for EMD, EEMD and CEEMDAN that achieves around 10x speedup compared to MATLAB.

The rapid rise of GPU technology in High Performance Computing gave rise to several parallel GPU-accelerated *EMD* implementations. Waskito et al. reported the first single-precision CUDA EMD implementation for audio signal processing achieving 29x and 29.9x speedups compared to sequential C versions on a C1060 and C2050 NVIDIA Tesla card, respectively [27,28]. Xie et al. created a CUDA EMD version for seismic data processing that achieved 4x speedup on a GT240 GPU card [29]. Huang et al. [30] reported 33.7x speedup on a C2050 GPU using overlapped piecewise cubic spline interpolation technique.

Since *Ensemble EMD* has significantly higher computation cost due to the large number of noise assisted copies of the original signal, parallelism in this case is mandatory to achieve acceptable execution times. Wang et al.'s implementation is developed for offline spectrum discrimination of hyperspectral remote sensing images and achieved 60.62x speedup over a sequential C implementation running on an NVIDIA C1060 Tesla GPU card [31]. In a follow-up paper, they compare serial MATLAB, sequential and multi-core C as well as their CUDA implementation (C1060 GPU) and found that sequential C is 5 times faster than MATLAB, a quad-core C version is 15 times, while the CUDA version is 60 times faster than the MATLAB implementation [32]. Chen

[2] The execution times of the MATLAB MEMD implementation on an 8-core CPU (Intel Core i7–9700 K, 3.60 GHz) system were 105.7 and 399 min for the 64-channel 128 direction vector and 128-channel 256 direction vector cases (signal length: 79,872 samples), respectively. For the typical group size of 25 subjects, these runtimes result in an overall execution time of 44 h (64 channels) and 6.9 days (128 channels).

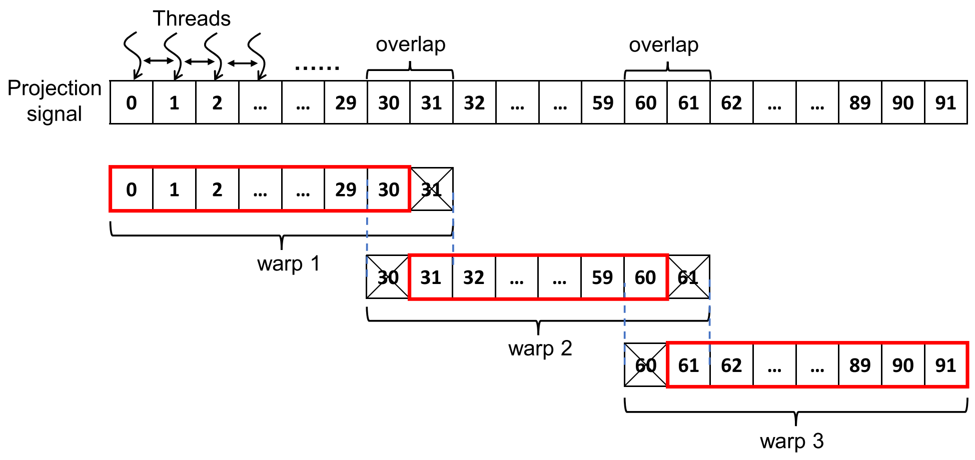

Fig. 4. The principle of extreme point detection using warp shuffle operations. Red colour marks threads performing the extrema detection steps.

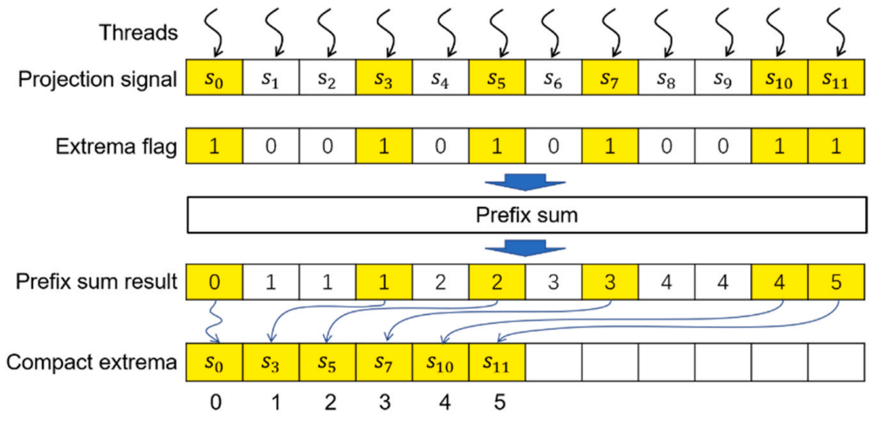

Fig. 5. The principle of generating compact extreme point vectors using the prefix sum operation.

et al. developed a real-time CUDA EEMD implementation [33] for anaesthesia monitoring purposes. They showed that it is possible to achieve real-time processing speed with a GTX295 GPU card (31.3x speedup, dual GPU card). EMD and EEMD are designed for processing single sensor channel data but extensions exist for multidimensional cases. Chang et al. developed implementation for the *Multi-Dimensional EEMD* algorithm [34] and Mujahid et al. reported the first *Multivariate EMD* GPU implementation [23] achieving 6–16x speedup over the MATLAB implementation.

The common characteristics of the reviewed GPU implementations are that (i) they all exploit multiple levels of parallelism and use several GPU optimisation techniques to achieve acceptable performance gain, and (ii) use early generation, now outdated GPU processors and early versions of the CUDA programming language; (iii) the achieved speedup values are relatively modest, and (iv) source code is not publicly available.

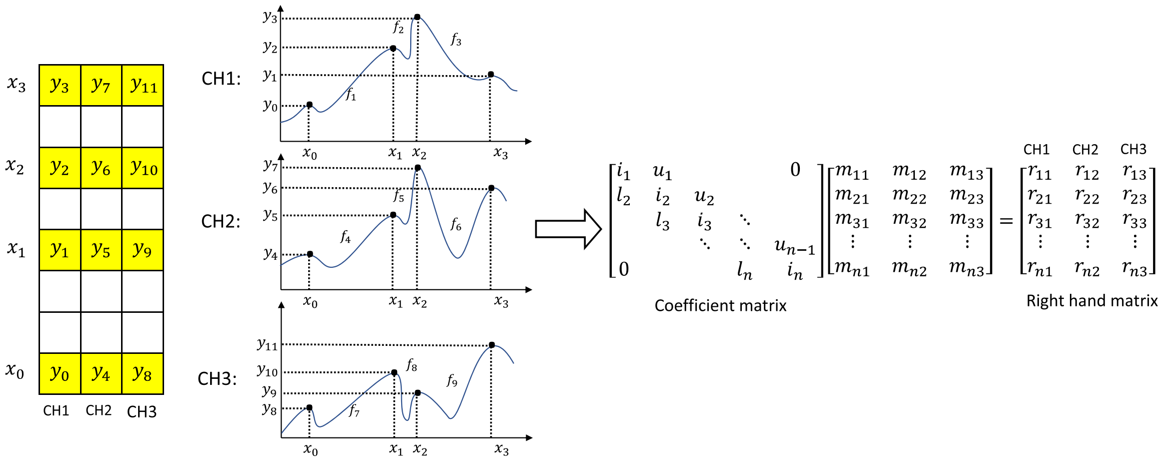

Fig. 6. Batch Multi-right-handed tridiagonal matrix equation generated by multivariate extreme points.

Table 2
Device variables used in the GPU implementations, their size and kernels in which they are referenced.

Variable name	Dimensions	Used in kernel function
d_current	SignalLength $\times$ SignalDim	cublasSgemm()
		selectExtrema()
		averageDirection()
d_directionVectors	SignalDim $\times$ NumDirVector	cublasSgemm()
d_projectSignals	SignalLength $\times$ NumDirVector	cublasSgemm()
		findExtremaShfl()
d_sparseFlag	SignalLength $\times$ NumDirVector	findExtremaShfl()
		scanLargeDeviceArray()
		scanSmallDeviceArray()
		selectExtrema()
d_ScanResult	SignalLength $\times$ NumDirVector	scanLargeDeviceArray()
		scanSmallDeviceArray()
		selectExtrema()
d_compactValue	SignalLength $\times$ SignalDim $\times$ NumDirVector	selectExtremaM()
d_compactIndex/		splineCoefficients()
		interpolate()
d_upperDia	SignalLength $\times$ SignalDim $\times$ NumDirVector	tridiagonalSetup()
d_middleDia		cusparseSgtsv2()
d_lowerDia		
d_right		
d_solutionGtsv	SignalLength $\times$ SignalDim $\times$ NumDirVector	cusparseSgtsv2()
		splineCoefficients()
d_b (d_upperDia)		splineCoefficients()
d_c (d_middleDia)		interpolate()
d_d (d_lowerDia)		
d_envelopeVaule		interpolate()
		averageUppperLower()
d_meanEnvelope		averageUppperLower()
		averageDirection()
d_running	SignalLength $\times$ SignalDim	
d_IMFs	SignalLength $\times$ SignalDim $\times$ NumIMFs	

3. Methods

Now we turn to the description of our GPU-based parallel MEMD implementation. As explained in Section 2.3 and show in Algorithm 3, the main steps of the iterative MEMD algorithm are (i) signal projection onto the multidimensional direction vectors, (ii) extrema detection in the projected signals, (iii) cubic spline interpolation to generate upper and lower envelope of the projected signal (iv) computation of the mean envelope, and (v) generating the IMF and updating the input signal for the next iteration. While this series of steps is inherently sequential in execution, there are many opportunities for parallelism in the execution in each step. The strategy and implementation details of our parallel solution is described in the rest of this section.

3.1. *Parallel design*

In our parallel implementation, each step of MEMD is implemented by one or more kernel functions executed on the GPU, or by highly optimized CUDA library functions. Essentially, the kernel function is a description of the thread behaviour, so by designing the kernel function, we can arrange independent tasks to different threads. Since the GPU has a large number of computing cores, threads can be executed in parallel. When we launch the kernel function on the CPU side, the threads are organized into blocks with different sizes, and the thread blocks are organized into grids with different sizes. The calculation steps of MEMD and its corresponding kernel functions are shown in Table 1. Threads in kernel functions need to read data from memory for computation, so the layout of data in memory also plays an important role in the parallel implementation of MEMD. The following parts are the details about the implementation in each step and memory layout.

3.2. *Pre-processing*

In the pre-processing stage, we perform all GPU memory allocations and initialisations, and generate the direction vectors used later during the signal projection step. As shown in Fig. 2, direction vectors are required in MEMD to generate multivariate signal envelopes. These vectors are unit vectors of a hypersphere represented by their angles. The more angles we have, the more uniform the distribution of the vectors can be provided we use suitable sampling algorithm. The Hammersley sequence – a low-discrepancy sequence that provides uniform angle sampling – was used by Rehman et al. [1] as the basis for generating direction vectors. We also adopted this method in our implementation. First, a set of prime numbers are generated on the CPU and copied into the GPU global memory. Next, the CUDA kernel function generateHammSeq() is executed with each thread performing a radical inverse operation [35] on one prime number from the set. This is followed by the kernel function generateDirVec() that generates the corresponding direction vector on the hypersphere. Performing the GPU memory allocation, initialisation and generating the direction vectors is

Table 3
Architecture parameters of the GPU platforms used for measurements.

	GTX 980	Titan Xp	Tesla V100	RTX 3070 mobile
Architecture	Maxwell	Pascal	Volta	Ampere
CUDA cores	2048	3840	5120	5120
Clock frequency (GHz)	1.126	1.48	1.46	1.62
Memory (GB)	4	12	16	8
Peak FP32 performance (TFlop/s)	4.98	11.36	14.03	16.59
CUDA version	10.2	10.2	11.3	11.4

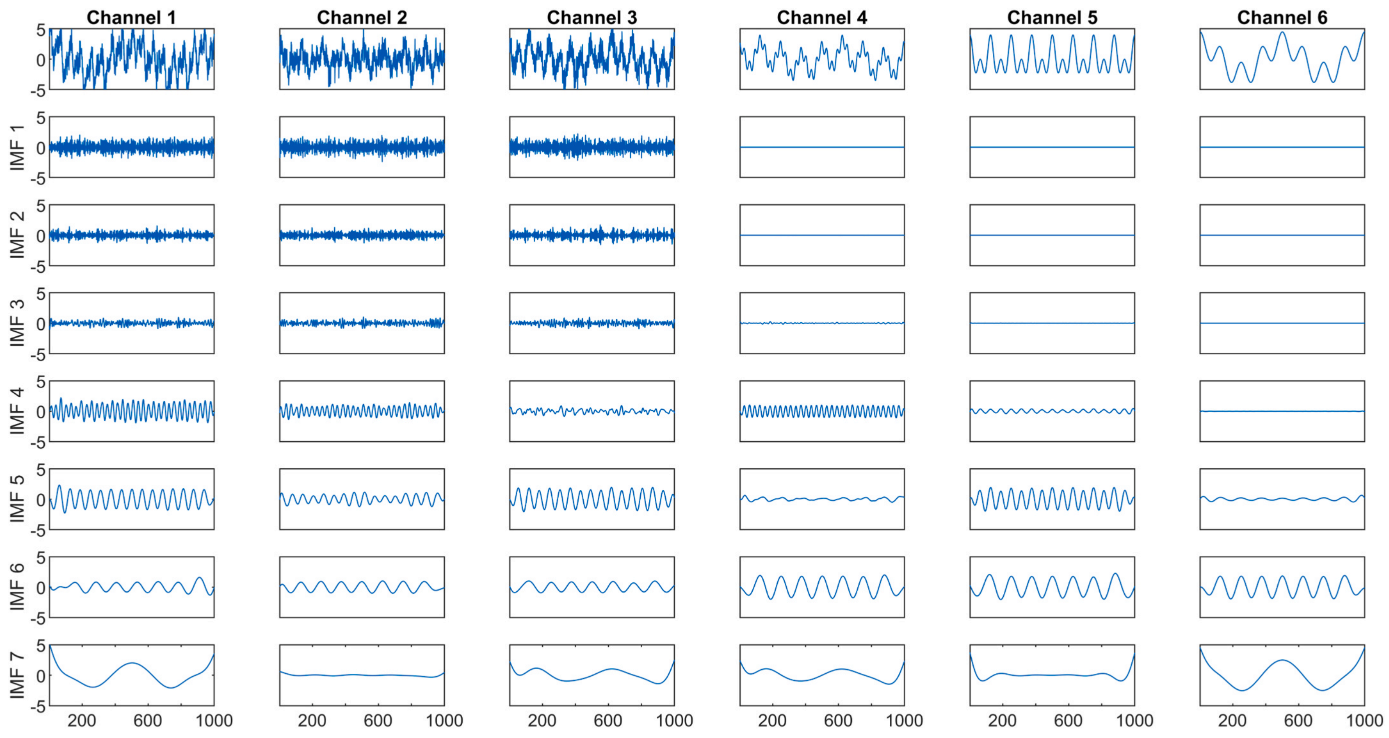

Fig. 7. Result of the MEMD decomposition of the hexavariate dataset.

a one-time execution step, not part of the iterative process of MEMD, therefore it has negligible impact on the performance of the overall algorithm.

3.3. Signal projection to direction vectors

Once the direction vectors are obtained, the multivariate input signal must be projected onto each direction vector. The projection of one signal vector onto a direction vector can be represented as the dot product of the two vectors; consequently, the projection of a multivariate input signal onto multiple direction vectors can be represented as a matrix-matrix multiplication operation. As shown in Fig. 3, the projection signal matrix **P** (SignalLength $\times$ NumDirVector) can be obtained after we multiply the input signal matrix **S** (SignalLength $\times$ SignalDim) and the direction vector matrix **D** (SignalDim $\times$ NumDirVector).

The multivariate input signal projection operation (matrix-matrix multiplication) is performed by the cuBLAS library function cublasSgemm. The cuBLAS linear algebra library is highly optimized for NVIDIA GPUs and provides convenient and high-performance program interfaces for vector and matrix algebra operations. The resulting projected signal matrix **P** is input to the subsequent extrema detection step, during which each projected signal (a column of **P**) will be used as an independent input signal.

3.4. Extrema detection

The way to detect extrema is to compare a signal value with the values of its two adjacent neighbours and decide whether the value is an extrema (maximum or minimum) or not. This operation can be performed for each value in parallel by a GPU thread. This requires one

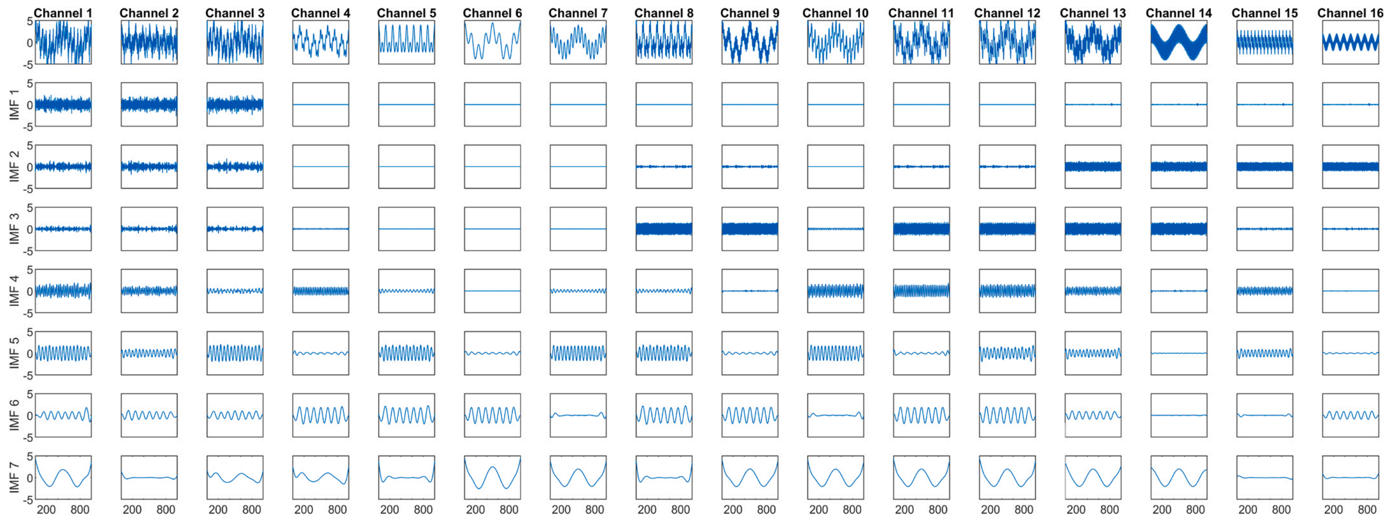

Fig. 8. Result of the MEMD decomposition of the 16-channel dataset.

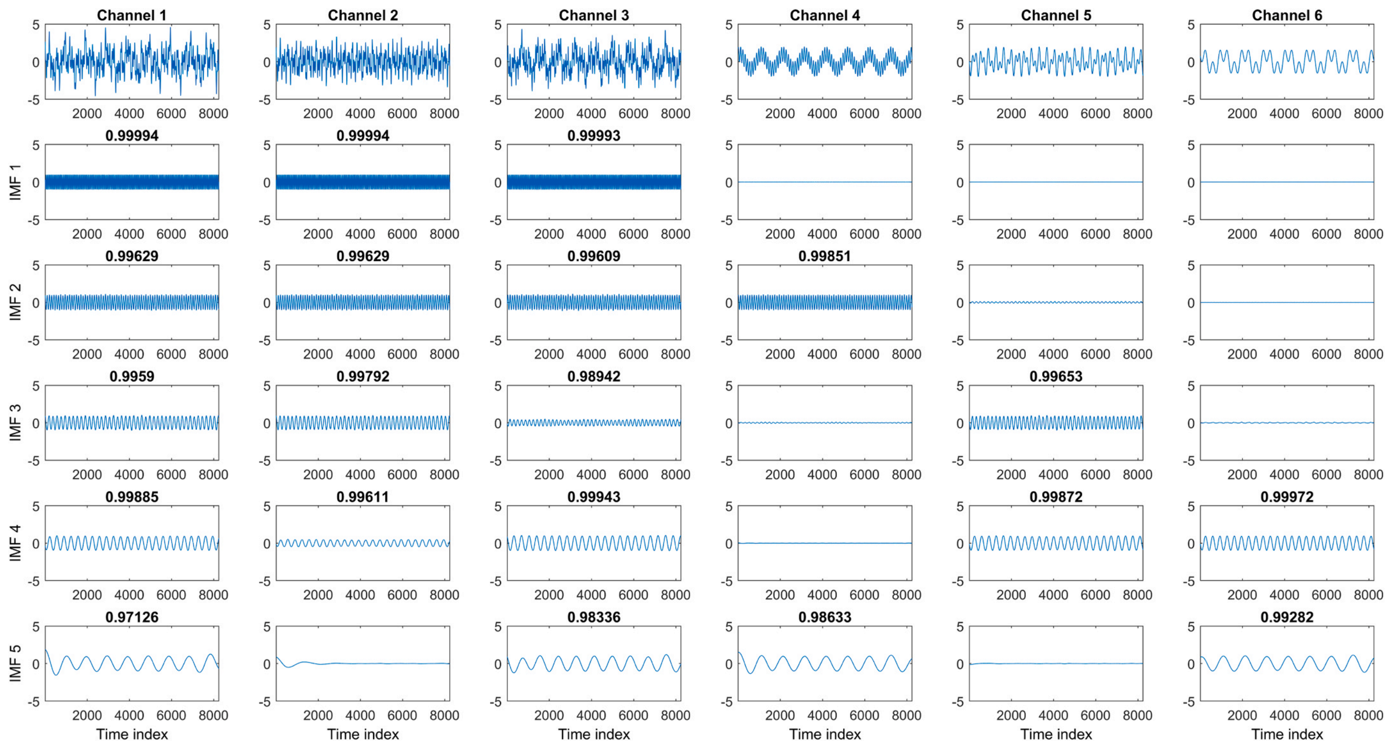

Fig. 9. Decomposition results of Dataset B hexavariate signal with Similarity Index values shown in bold for each component.

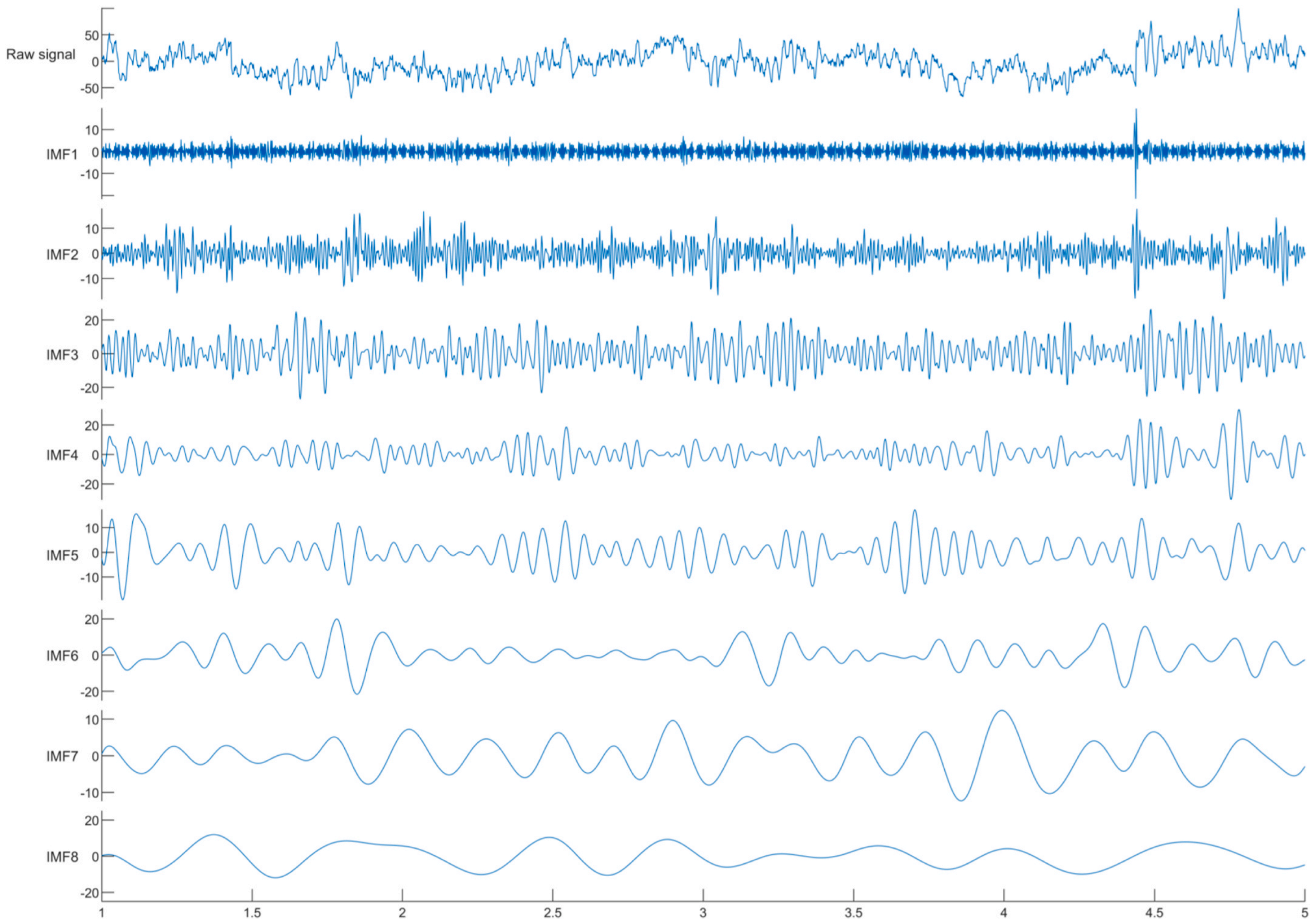

Fig. 10. A section of the EEG signal (Channel 4) of the EEG sample dataset and the resulting IMFs (IMF1–8) of the GPU MEMD implementation.

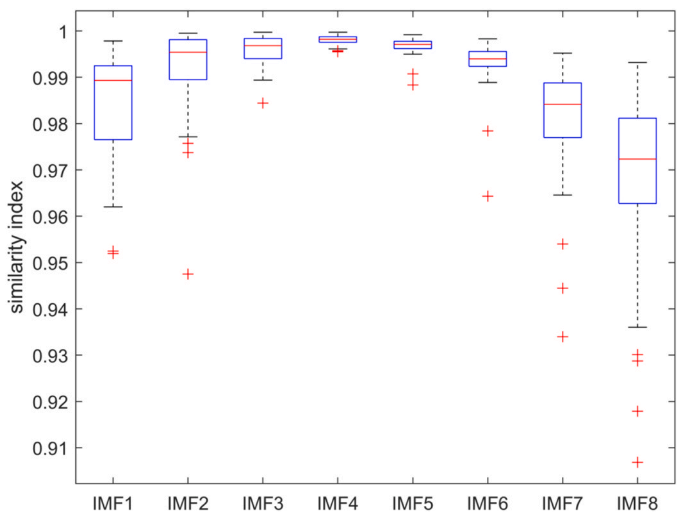

Fig. 11. Boxplot of Similarity Index values representing the difference between the MATLAB and GPU implementations of the MEMD algorithm. Each IMF category represents the distribution of the Similarity Index values of all channels for that IMF. Red lines represent median values.

thread to read three values then to perform the comparison. Reading the three values from global memory in each thread is an inefficient strategy for several reasons; (i) data needs to be loaded from slow memory, (ii) each thread needs 3 load instructions for 1 comparison, and (iii) each value is read multiple times as threads cover the entire input vector. A typical approach in CUDA programming in these situations is to use the on-chip shared memory that can store several input values in fast memory that is accessible to all threads in a thread block. This reduces the number of necessary load instructions and increases access speed as well. However, we still need three load instructions per thread, and performance degrading shared memory bank conflicts may also occur.

In our parallel implementation, we used the CUDA warp level shuffle instructions _shfl_up_sync and _shfl_down_sync in the extrema detection kernel function findExtremaShfl. The shuffle instruction is a intrinsic hardware accelerated warp-level instruction that provides direct access to register variables of the other threads within a warp. (Warp is a 32-thread unit of execution in NVIDIA GPU devices.) The shuffle instruction removes the need for loading neighbour values from memory; each thread reads a single value only into a local register that will be accessible to the neighbour threads via the shuffle operation. As shown in Fig. 4, there are 32 threads in each warp, and there is an overlap of two threads between every two warps. Except for the first and last warps, each warp is designed to ignore the comparison operation for the first and last thread, as these valued are examined by the second last thread in the previous warp and the second thread in the following warp, respectively. Using this warp-level overlap, the detection of extreme points can be completed without any omission and at very high efficiency.

After the detection of extreme points, we obtain the positions and values of all extreme points in the projection signal. These positions are the actual sample index values stored in a sparse vector. To make this vector suitable for cubic spline interpolation, we need to compact this position vector to a dense continuous sequence of location indexes.

The vector compaction step is illustrated in Fig. 5. During the extreme point detection step, we generate an auxiliary flag vector to identify the location of extreme points with a logical 1 value. Subsequently, the kernel function scanArray will perform a prefix sum operation on the flag vector and return the prefix sum result, i.e. the corresponding positions of the extreme points in a compact vector. Using this strategy, all parallel threads in the selectExtrema kernel function can work independently, in parallel, using the extrema flag vector as the predicate, the projection signal vector and the prefix sum result vector to generate the compacted extreme point vector.

Since the first and last points of the signal do not have left or right neighbours, respectively, it is not possible to determine whether these points are extrema or not. These two boundary points are handled separately by the kernel function setBoundary, which – in the current implementation – uses the slope extension method as the boundary condition.

3.5. Cubic spline interpolation

To calculate the upper and lower envelopes, we need to perform cubic spline interpolation based on the detected extreme points using the cubic spline function polynomial

$$S_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$$

With $n+1$extreme points (x_i, y_i), there will be n gaps corresponding to n cubic splines, each with four unknown coefficients (a, b, c, d), resulting in $4n$ unknowns that requires $4n$ equations.

The detected extreme points (spline control points) satisfy the spline

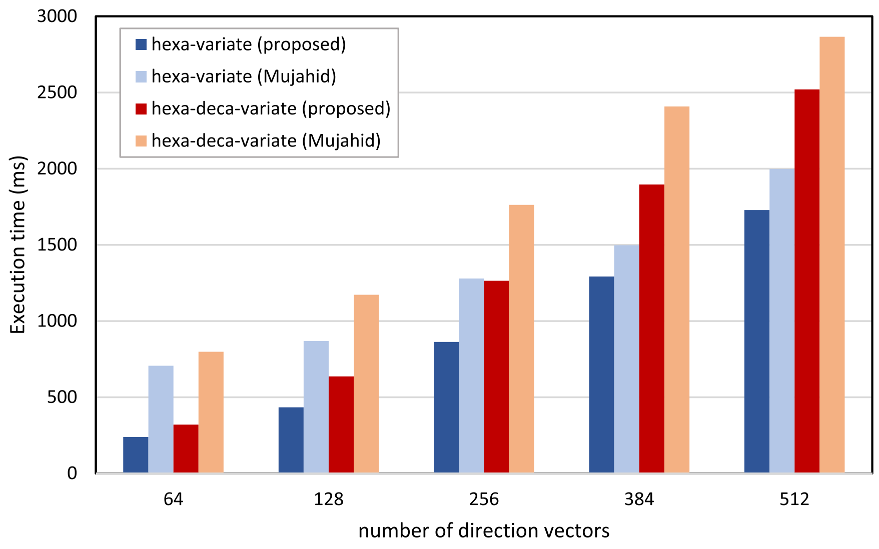

Fig. 12. Execution time of our GPU implementation compared to [23] on an NVIDIA GTX 980 GPU card. Two (a 6 and a 16-channel) datasets with 1000 samples per channel were used in both implementations using varying number of direction vectors.

Table 4
The maximum number of data samples per channel the proposed implementation can process on the different test GPU cards.

Number of channels	RTX 3070 mobile			Titan XP			Tesla V100		
	# Direction vectors			# Direction vectors			# Direction vectors		
	64	128	256	64	128	256	64	128	256
32	78k	40k	20k	118k	60k	30k	164k	84k	42k
64	40k	20k	10k	60k	30k	14k	84k	42k	20k
128	20k	10k	4k	30k	14k	6k	42k	20k	10k

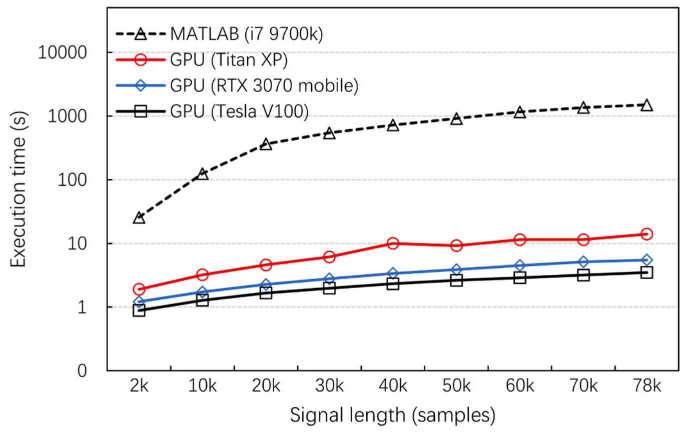

Fig. 13. 32-channel 64 direction vectors, up to 78k samples, MATLAB vs GPU execution times.

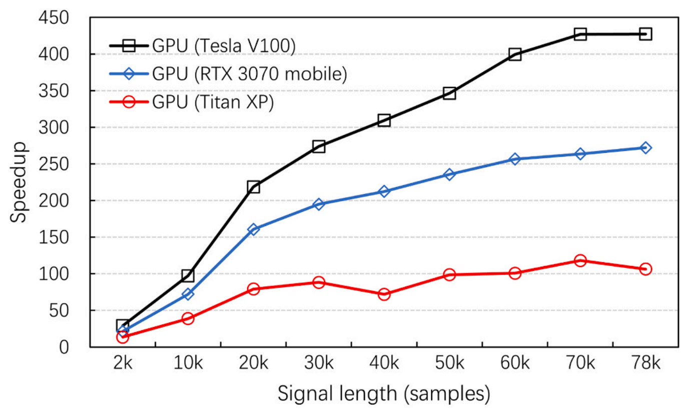

Fig. 14. Speedup for 32-channel 64 direction vectors, up to 78k samples, compared to MATLAB.

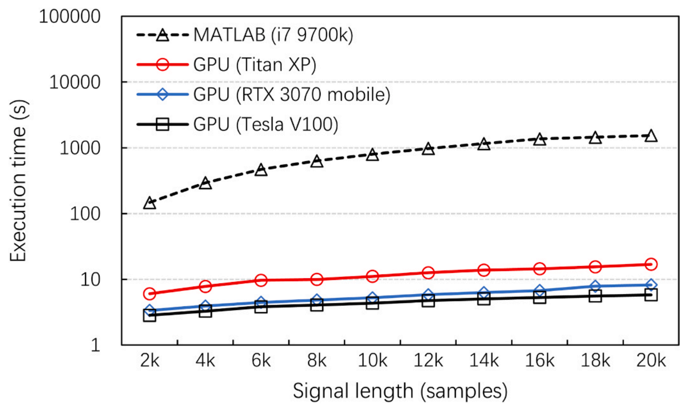

Fig. 15. 64-channel 128 direction vectors, up to 20k samples, MATLAB and GPU execution times.

function. Each extreme point must satisfy the two spline equations to its left and right. The first and last points only satisfy the right and left hand spline, respectively. As a result of these conditions, we obtain $2n$ equations. The next condition is continuity. The spline should be first and second derivative continuous at each extreme point, which condition will give us another $2(n-1)$ equations. Now we have $4n-2$ equations, and we are two equations away from solving the unknowns. The last two missing equations are given by the boundary conditions. In our implementation, we have chosen a natural spline to interpolate on the first and last points, having the second derivative at the first and last extreme points set to 0. This gives us the last two equations and the system of equations is now solvable.

Using a step size $h_i = x_{i+1} - x_i$, and the continuity condition $m_i = S_i''(x_i)$, then under the boundary conditions of the natural spline, the entire system of equations can be expressed as the following tridiagonal matrix equation:

$$\begin{bmatrix} 1 & 0 & 0 & & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & 0 & & \\ 0 & h_1 & 2(h_1+h_2) & h_2 & 0 & \\ 0 & 0 & h_2 & 2(h_2+h_3) & h_3 & \vdots \\ \vdots & & & & & \\ & & 0 & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_n \end{bmatrix}$$

$$= 6 * \begin{bmatrix} 0 \\ \frac{y_2-y_1}{h_1} - \frac{y_1-y_0}{h_0} \\ \frac{y_3-y_2}{h_2} - \frac{y_2-y_1}{h_1} \\ \frac{y_4-y_3}{h_3} - \frac{y_3-y_2}{h_2} \\ \vdots \\ \frac{y_n-y_{n-1}}{h_{n-1}} - \frac{y_{n-1}-y_{n-2}}{h_{n-2}} \\ 0 \end{bmatrix}$$

After we solve for m, the coefficients of each spline function can be expressed as:

$$a_i = y_i$$

$$b_i = \frac{y_i - y_i}{h_i} - \frac{h_i}{2}m_i - \frac{h_i}{6}(m_{i+1} - m_i)$$

$$c_i = \frac{m_i}{2}$$

$$d_i = \frac{m_{i+1} - m_i}{6h_i}$$

Envelope interpolation in MEMD is a more complicated process than in the traditional univariate EMD. The extreme points in the projected signal will correspond to multivariate extreme points in the input signal using points generated by backprojection with the location of the extreme points in the projected signal. Fig. 6 shows a simple example for explanation. There are three input channels and and certain number of projected signals. We detected four four extreme point locations in one of the projected signals, x_0, x_1, x_2, x_3, which are projected back to each

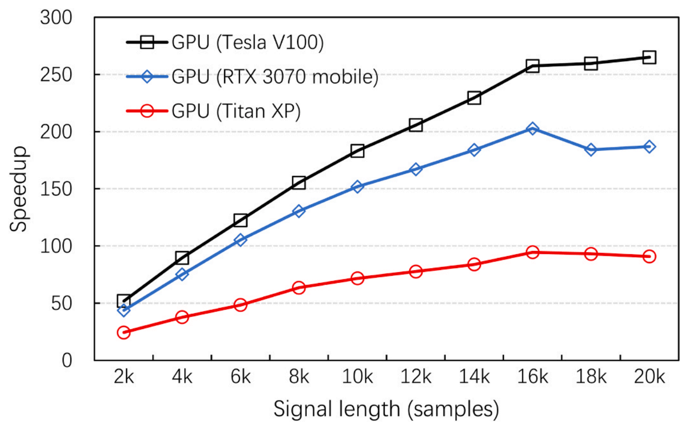

Fig. 16. Speedup for 64-channel 128 direction vectors, up to 20k samples compared to MATLAB.

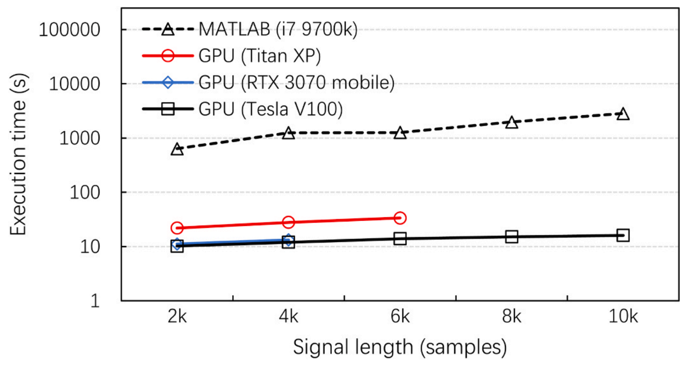

Fig. 17. 128-channel 256 direction vectors, up to 10k samples, MATLAB and GPU execution times.

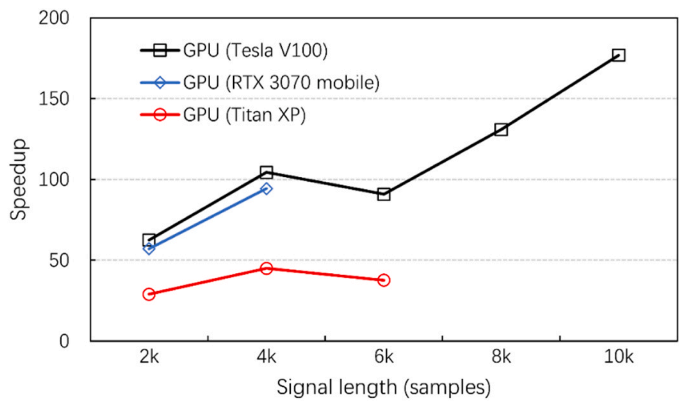

Fig. 18. Speedup for 128-channel 256 direction vectors, up to 10k samples compared to MATLAB.

input signal to obtain signal values y_0-y_{11}.

Each multivariate extreme point is composed of three values obtained for the give location with a lookup operation from the three input channels (e.g. x_0 –> y_0, y_4, y_8). Next, cubic spline interpolation, outline above, will be performed on each input channel. Note that each channel will generate one tridiagonal matrix equation using the same coefficient matrix as the other channels. This allows us to use a batch tridiagonal solver in which many right hand side vectors represent the different channels. For this we used the multi-right-hand tridiagonal system solver function cusparseSgtsv2 of the cuSparse CUDA library. This function uses the Parallel Cyclic Reduction [36] algorithm to solve the equations in parallel instead of the commonly used sequential Thomas algorithm.

Before solving the equations, the tridiagonal system of equation must be generated for each direction vector. Our CUDA kernel tridiagonalSetup generates each row of the tridiagonal coefficient matrix based on the extrema locations in the projected signal and one value of the right-hand matrix.

Once the system of equations is solved for m, the kernel function splineCoefficients will launch a number of threads equal to the number of interpolating splines to compute the corresponding polynomial coefficients (a, b, c, d) and store them in the GPU memory.

The actual interpolation operation based on the computed polynomial coefficients (a, b, c, d) is handled by the CUDA kernel interpolate. Since the number of extreme points will decrease with each IMF iteration, the number of values between extreme points that need to be interpolated will also vary with each iteration. Therefore, in the kernel function, we use a binary search to select the coefficients of the spline function corresponding to each gap, and each thread is scheduled to compute a single point in the interpolated spline.

After interpolating the multivariate upper and lower envelopes, the mean multivariate envelope of the input signal is calculated, which after subtracting it from the input signal will give us a potential IMF. This process consists of two parts. First, for each direction vector, we calculate the mean multivariate envelope from the multivariate upper and lower envelopes. This step is handled by the kernel function averageUpperLower. Second, since each direction vector will have its own corresponding mean multivariate envelope, we need to average these mean multivariate envelopes for all direction vectors to obtain the final mean multivariate envelope of the input signal. This step is performed by the kernel function averageDirection. After this step, one candidate IMF is generated by subtracting the multivariate average envelope from the input signal, and a new iteration of MEMD algorithm will begin.

3.6. *Data layout in memory*

In heterogeneous accelerated computing systems, the memory of the CPU and GPU are two independent subsystems that communicate with each other via the PCI-e bus. Even the current state-of-the-art PCI-e 4.0 × 16 bus can only provide 32 GB/s bandwidth, which is still far below the bandwidth of the GPU global memory (several hundred GB/s). Consequently, data exchange between CPU and GPU memory is a time-consuming operation which should be minimised to achieve high efficiency. The ideal situation is to allocate GPU memory only once, store all data in GPU memory and all operations are performed in GPU memory without CPU interaction. In our parallel implementation of MEMD, we allocate memory for all variables in GPU memory before the computation starts, and the memory exchange between CPU and GPU is limited to the copying the input signals and direction vectors to the GPU and retrieving the final decomposition results. The allocation of memory is mainly controlled by three size variables, namely the length of the multivariate signal (SignalLength), the dimension of the multivariate signal (SignalDim), and the number of direction vectors (NumDirVector). Table 2 lists the memory requirement of each variable and their use in the GPU kernels.

4. Results

In this section, we first list the architectural details of GPU systems we used for testing our implementation, then present the numerical validation results using synthetic datasets. this is followed by performance results using real high-density EEG datasets.

4.1. *Test hardware*

Different types of NVIDIA GPU cards were used in the testing process to explore performance variation across architecture families and card models. We selected different NVIDIA gaming and compute cards,

Table 5
Relative contributions of the kernels to the overall execution time. Channel count is 32, number of direction vectors are 64.

	sample size						
kernel	**4097**	**8193**	**16385**	**32769**	**43009**	**79873**	**104449**
pcrGlobalMemKernel_manyRhs	50.0%	47.6%	42.3%	35.5%	31.4%	23.0%	20.2%
pcrLastStageKernel_manyRhs	14.9%	12.1%	9.5%	7.2%	6.1%	4.4%	3.5%
pcrGlobalMemKernelFirstPass_manyRhs	12.5%	10.2%	7.9%	5.9%	5.0%	3.3%	2.8%
crGlobalForIterations_multiple	0.0%	0.0%	3.6%	4.5%	5.4%	5.9%	5.6%
crGlobalBottomKernel_multiple	0.0%	2.2%	3.3%	3.8%	6.8%	8.9%	9.1%
prescan_arbitrary	5.3%	4.5%	3.7%	2.9%	2.5%	1.8%	1.6%
prescan_large	3.8%	3.2%	2.7%	2.2%	1.9%	1.5%	1.3%
interpolate	2.8%	4.9%	8.1%	12.7%	14.5%	19.4%	21.7%
add	4.9%	4.1%	3.3%	2.5%	2.2%	1.6%	1.4%
averageUppperLower	0.8%	1.3%	2.0%	2.9%	3.3%	4.3%	4.7%
tridiagonal_setup	0.8%	1.2%	1.8%	2.7%	3.2%	4.1%	4.4%
spline_coefficients	0.7%	1.1%	1.8%	2.6%	3.1%	3.9%	4.3%
select_extrema_max	0.5%	0.8%	1.3%	2.3%	2.6%	3.4%	3.7%
select_extrema_min	0.5%	0.8%	1.3%	2.3%	2.6%	3.4%	3.7%
tridiagonal solver kernels	**77.3%**	**72.0%**	**66.6%**	**56.8%**	**54.7%**	**45.5%**	**41.2%**
custom kernels	**20.1%**	**21.9%**	**26.0%**	**33.1%**	**35.9%**	**43.3%**	**46.7%**

Table 6
Relative contributions of the kernels to the overall execution time. Channel count is 128, number of direction vectors are 256.

	sample size					
kernel	**2049**	**4097**	**6145**	**8193**	**10241**	**12289**
pcrGlobalMemKernel_manyRhs	48.5%	50.6%	48.4%	46.8%	45.4%	44.5%
pcrLastStageKernel_manyRhs	22.0%	18.4%	15.5%	14.4%	13.4%	12.0%
pcrGlobalMemKernelFirstPass_manyRhs	16.5%	14.7%	12.4%	11.6%	11.0%	10.0%
crGlobalForIterations_multiple	0.0%	0.0%	1.9%	2.1%	2.1%	2.8%
crGlobalBottomKernel_multiple	0.0%	0.0%	2.2%	2.5%	2.6%	3.5%
prescan_arbitrary	2.2%	1.9%	1.7%	1.6%	1.5%	1.0%
prescan_large	1.6%	1.4%	1.3%	1.2%	1.1%	1.5%
interpolate	2.2%	4.9%	5.2%	6.6%	7.9%	8.7%
add	2.1%	1.8%	1.5%	1.4%	1.3%	1.2%
averageUppperLower	0.6%	1.1%	2.0%	1.7%	2.0%	2.2%
tridiagonal_setup	0.6%	1.2%	1.3%	1.6%	1.9%	2.0%
spline_coefficients	0.6%	1.0%	1.3%	1.5%	1.8%	2.0%
select_extrema_max	0.4%	0.7%	1.0%	1.2%	1.4%	1.6%
select_extrema_min	0.4%	0.7%	1.0%	1.2%	1.4%	1.6%
tridiagonal solver kernels	**87.0%**	**83.7%**	**80.5%**	**77.4%**	**74.5%**	**72.7%**
custom kernels	**10.7%**	**14.5%**	**16.1%**	**18.0%**	**20.4%**	**21.7%**

whose details are listed in Table 3. The now outdated GTX 980 was used only to compare performance with the only known MEMD GPU implementation [23]. The Titan Xp and RTX 3070 were used both for development and testing. The V100 card was only used for performance measurement. The core count of the GPUs ranges between 2048 and 5120, while the performance varies from 4.98 to 16.69 TFlop/s (single precision). For the MATLAB tests, we used an i7–9700 K CPU (8 cores, 3.60 GHz base frequency, 4.90 GHz turbo frequency) and MATLAB r2019a.

4.2. Numerical validation

We used several synthetic datasets and real EEG measurements for validating the correctness of our implementation. Some datasets were used in order to compare our results with literature data while others were used for quantitative tests and comparison with the reference MATLAB implementations.

4.2.1. Synthetic Dataset 1

The first synthetic dataset used in the validation process was described in [1] (also available online[3]) and contains 6, 12 and 16-channel multivariate data series. The hexavariate dataset was generated by adding four different frequency sine waves (x_1: $f_1 = 2$ Hz, x_2: $f_2 = 8$ Hz, x_3: $f_3 = 16$ Hz and x_4: $f_4 = 32$ Hz) and noise to different subsets of

[3] https://www.commsp.ee.ic.ac.uk/~mandic/research/memd/MEMD_Supplement.zip

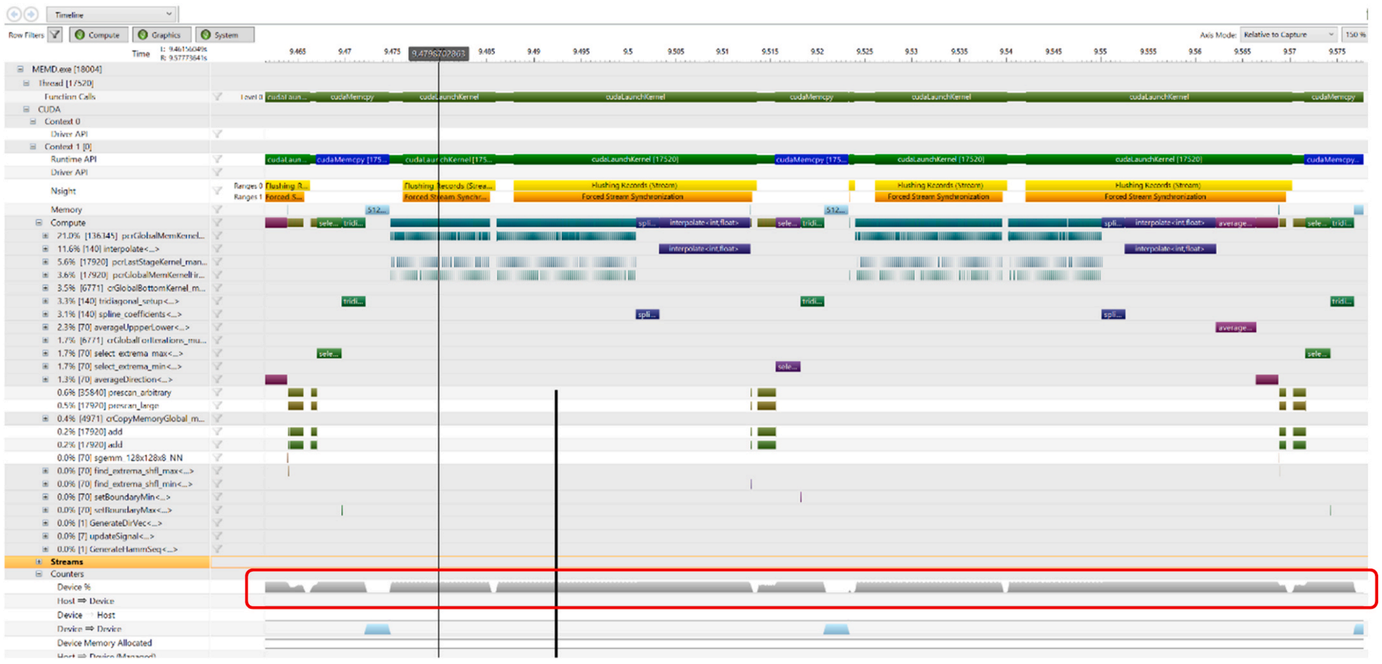

Fig. 19. Execution timeline of one iteration of the IMF computation look of the MEMD algorithm.

channels so that x_1 appears in Channels 1,2 and 4, x_2 in Channels 1–3 and 5, x_3 in all channels, x_4 in Channels 1, 3–4 and 6, and noise in Channels 1–3, making it suitable for mode alignment test. Since this dataset was used as test data in [23], we will use primarily for performance comparison but for visual inspection, the result of the MEMD decomposition of this dataset obtained with our GPU implementation is shown in Fig. 7. The top row contains the individual input signals of the hexavariate dataset (Channels 1–6). The subsequent rows, from IMF 1–7, show the extracted oscillatory modes for each channel, starting with the noise (IMF1–3) then the signal components.

Fig. 8 illustrates the decomposition result of the 16-channel synthetic signal. The correct mode alignment (some channels showing a noise or specific oscillatory mode while others showing none or negligible components) is evident. Our implementation correctly identified the noise and sine wave components common in multiple channels in all three datasets and matched the results reported in [23].

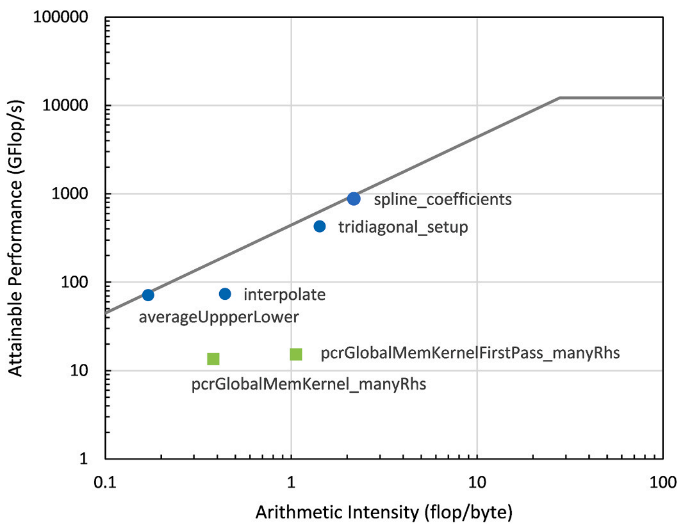

Fig. 20. The Roofline performance model of the RTX 3070 mobile GPU showing the performance positions of the main kernels of the implementation.

4.2.2. *Synthetic Dataset 2*

The second synthetic dataset was created for testing the numerical accuracy of our decomposition implementation. We generated a hexavariate dataset without added noise by adding five pure sine waves of different frequencies (x_1: $f_1 = 2$ Hz, x_2: $f_2 = 6$ Hz, x_3: $f_3 = 11$ Hz and x_4: $f_4 = 19$ Hz, x_5: $f_5 = 40$ Hz) to different subsets of channels so that x_1 appears in Channels 1–3, x_2 in Channels 1–4, x_3 in Channels 1–2 and 5, x_4 in Channels 1–3 and 5–6, while x5 in Channels 1, 3–4 and 6.

4.2.2.1. *Similarity index.*

To quantify the accuracy of the decomposition, we compared the original and decomposed signal components using the Similarity Index metric ρ given as

$$\rho_i(x_i(t), IMF_i(t)) = \frac{\mathrm{cov}(x_i(t), IMF_i(t))}{\sqrt{\mathrm{var}(x_i(t))}\sqrt{\mathrm{var}(IMF_i(t))}}$$

where $\mathrm{cov}()$ represents covariance of the input signal and the corresponding IMF, $\mathrm{var}()$ represents the variance of the input signal and the IMF, respectively. A $\rho = 1$ value represents identical input signal component and IMF, i.e. perfect decomposition. Fig. 9 shows the result of the decomposition of this dataset including the Similarity Index values above each IMF component. The lowest value of ρ is 0.971, while the majority similarity index values are above 0.99.

4.2.3. *EEG dataset*

To compare our decomposition results with the MEMD MATLAB implementation, we selected a real EEG dataset from the samples provided with the EEGLAB Toolbox [37]. The selected dataset consists of 32 channels each with 30,504 samples (sampling frequency is 512 Hz). In Fig. 10, we show the result of the signal decomposition of Channel 4 of the dataset from $t = 1$–5 s. The different oscillation frequencies, amplitude and frequency modulations are clearly visible in the different IMFs. The results were compared with the EMDLAB MEMD MATLAB implementation [24]. The resulting similarity index values of the 32 channels for each IMF are plotted in Fig. 11. The difference, which is due to differences in floating point arithmetic instruction implementations on the CPU and GPU and slight differences in extrema detection boundary conditions, is less than 1.8 % on average.

4.3. Performance results

In this section, we present performance results using several metrics that characterise the performance of GPU implementations. For end users, wall clock time is the most important measure along with time reduction level (speedup) when compared to the implementation they hope to replace. For developers, additional measures, such as hardware efficiency, achieved peak compute performance, arithmetic intensity are important, as they characterise the quality of the implementation and the extent to how much the GPU card is utilised during the program execution.

We measured and compared the execution time of our GPU implementation on different GPU cards and compared it with MATLAB runtimes. We also provide speedup results indicating the speed advantage of the GPU implementation over the MATLAB one. Note, however, that GPU speedup ($S = T_{CPU}/T_{GPU}$) can easily become a misleading metric as it is often based on inefficient CPU implementations, might ignore programming language and multi-core CPU execution effects.

First, we compared the performance of our implementation to the only MEMD GPU implementation found in the literature [23]. Since several new GPU architecture generations have appeared since the date of publication of Mujahid et al.'s paper, to compare the implementation efficiency of our implementation objectively, we had to execute our program on the very same type of card (GTX 980) that was used in [23]. The execution time obtained this way with our version on two datasets (6 and 16 channels, signal length = 1000 samples) from Dataset A is shown in Fig. 12. On average, our implementation achieves a 1.69x speedup in execution time.

Next, the MATLAB MEMD execution time was compared with our proposed GPU implementation executed on various GPU cards. The implementations were performed by varying the number of input channels, the number of samples per channel, and the number of direction vectors. The hardware details of the test systems are listed in Table 3. Since the memory size was different on the different GPU cards, we had to use slightly different problem sizes on each system. Table 4 shows the maximum number of input samples each GPU can process for different channel count and projection vector size parameter combinations.

The execution time and speedup results of the different test cases (32, 64 and 128 EEG channels) are shown in Figs. 13–18. The execution time figures show the MATLAB and three GPU execution time curves as a function of sample size. The Speedup figures show the speedup values calculated from the GPU runtime values and the MATLAB MEMD execution time. The MATLAB script was executed on an 8-core Intel i7–9700 K CPU.

The execution time results show that for all channel number cases the GPU execution time was below or around 10 s for all signal length. The speedup values show a positive correlation with signal length that implies that the more data is fed into the GPU the more efficient it becomes during execution due to the increased number of threads ready for execution. This demonstrably helps in hiding memory data transfer latencies. The V100 speedup values – in the range of 180–430x – are an order of magnitude higher than previously reported ones.

4.4. Performance analysis

Here we provide a summary of the performance evaluation of our MEMD implementation. We profiled the code to identify potential performance bottlenecks and see the relative weight of each GPU kernel during program execution. Table 5 shows the results we obtained with the 32-channel EEG dataset (64 direction vectors) on the V100 card while varying the input signal length. Each column shows the relative contribution of the kernels to the execution time for the signal length of the column. Kernels with names set in italic are part of the tridiagonal solver implementation cusparseSgtsv2_nopivot() of the NVIDIA cuSparse library. The other kernels were developed by us. To help visualise performance trends, we used green and blue colour bars to show the change of the relative contributions of the cuSparse kernels and ours, respectively. The last two rows of the table show the total relative contribution of the cuSparse library and our custom kernels.

Table 6 shows the profiling results obtained with a 128-channel EEG dataset (256 direction vectors), also executed on a V100 GPU. Both Table 5 and Table 6 show that the tridiagonal solver dominates program execution time. In the 32-channel dataset, the relative weight of the solver decreases with increasing signal lengths, largely due to the increasing execution time of our spline interpolation kernel 'interpolate'. At around 80k sample size, the time spent in the solver and in our other kernels becomes equal, after which our custom kernels tend to be more dominant in the execution time profile. In the 128-channel case, when the number of direction vectors are quadrupled, the memory limits the execution for up to 12k samples per channel, only. In this input size range, the cuSparse solver becomes an even more important performance limiting factor, with larger than 72 % share of the execution time. The execution time of the interpolation kernel increases here as well but its relative weight is much less than in the 32-channel case.

The GPU utilisation of our most critical kernel 'interpolate' reaches 54 % compute and 45 % memory utilisation. Its floating point performance is relatively modest (64 GFlop/s) as most operations involve integer arithmetic, but the integer performance is over 1600 GOp/s. The execution timeline of one iteration of the MEMD program is shown in Fig. 19. The timeline data was obtained on a Titan Xp card. At the bottom of the picture, the 'Device %' row illustrates the utilisation of the GPU is nearly 100 % throughout the program with very little idle time periods (gaps) in the timeline.

We performed a Roofline [38] performance analysis that showed that the kernels (thus the entire program) in general are memory bound as they perform relatively few floating point arithmetic instructions compared to the data movement operations. The instruction-to-byte ratio of modern GPUs are typically between 14 and 30, hence only kernels having an Arithmetic Intensity value of 14 of higher can achieve close to peak compute performance. The arithmetic intensity of the kernels in our implementation are in the range of 0.1–2.2 and thus are limited by the memory bandwidth of the GPU cards. Fig. 20 illustrates the performance of the most critical kernels of our implementation on the roofline model of the RTX 3070 mobile GPU. Kernels marked with green boxes designate the kernels implementing the cuSparse tridiagonal solver. Blue circles mark custom kernels we developed for this implementation. The figure shows that our kernels reach close to theoretical performance (lie on or close to the global memory performance boundary line), while the NVIDIA kernels perform relatively poorly. Unfortunately, the implementation is proprietary; hence, we did not have an opportunity to perform further performance optimisation on them.

Our implementation proved to be correct and efficient, executing the MEMD algorithm within 1.2–17 s for up to 128 channels. The maximum sample length on a V100 GPU for 128 channels is 10k that may represent 5–40 s of measurement depending on the selected sampling frequency (256–2048 Hz). This is typically sufficient for epoch processing in Event Related Potential studies or for analysing windowed data. When using lower sampling frequencies (256 or 512 Hz), our processing time is shorter than the length of the data window which offers the possibility of using it in real-time and/or clinical applications.

5. Conclusions

In this paper, we described an efficient GPU implementation of the MEMD algorithm implemented in CUDA. Multivariate Empirical Mode Decomposition enables the accurate extraction of AM/FM modulated oscillatory modes of multi-channel high-density EEG/MEG datasets without the presence of the mode-alignment problem. This can open new opportunities in the deeper understanding of the brain processes underlying the execution of cognitive tasks or finding biomarkers for the early diagnosis of neurodegenerative diseases.

Our implementation was validated and tested for performance and correctness on different hardware platforms using a varying set of parameter values of channel count, direction vectors and signal sample size. The final version achieved a 180x to 430x speedup dependent on the length of the input signal and the GPU used with high decomposition accuracy. This level of performance can reduce data analysis execution times from days to minutes or from hours to seconds. Further performance increase can be expected from the Ampere and Hopper architectures and from multi-GPU extensions that are among our future plans. The source code of our implementation is available under the MIT Open Source license for the interested readers for use in applications or further improvements at the following URL: https://github.com/EEGLab-Pannon/MEMD-GPU.

CRediT authorship contribution statement

Zeyu Wang: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft, Visualization. **Zoltan Juhasz:** Conceptualization, Methodology, Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

This research was partially supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund (Project no. 2019-2.1.11-TÉT-2019-00069), financed under the 2019-2.1.11-TÉT funding scheme and the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No 101008468 (SLICES-SC). The support of the NVIDIA GPU Grant Program is also gratefully acknowledged.

References

[1] N. Rehman, D.P. Mandic, Multivariate empirical mode decomposition, Proc. R. Soc. A Math. Phys. Eng. Sci. 466 (2010) 1291–1302, https://doi.org/10.1098/rspa.2009.0502.
[2] N.E. Huang, Z. Shen, S.R. Long, M.C. Wu, H.H. Snin, Q. Zheng, N.C. Yen, C.C. Tung, H.H. Liu, The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis, Proc. R. Soc. A Math. Phys. Eng. Sci. 454 (1998) 903–995, https://doi.org/10.1098/rspa.1998.0193.
[3] G. Buzsaki, A. Draguhn, Neuronal oscillations in cortical networks, Science 304 (2004) 1926–1929, https://doi.org/10.1126/science.1099745.
[4] G. Wang, X.Y. Chen, F.L. Qiao, Z. Wu, N.E. Huang, On intrinsic mode function, Adv. Adapt. Data Anal. 2 (2010) 277–293, https://doi.org/10.1142/S1793536910000549.
[5] A.R. Hassan, A. Subasi, Automatic identification of epileptic seizures from EEG signals using linear programming boosting, Comput. Methods Prog. Biomed. 136 (2016) 65–77, https://doi.org/10.1016/j.cmpb.2016.08.013.
[6] P. Shahsavari Baboukani, G. Azemi, B. Boashash, P. Colditz, A. Omidvarnia, A novel multivariate phase synchrony measure: application to multichannel newborn EEG analysis, Digit. Signal Process. A Rev. J. 84 (2019) 59–68, https://doi.org/10.1016/j.dsp.2018.08.019.
[7] H. Liang, S.L. Bressler, E.A. Buffalo, R. Desimone, P. Fries, Empirical mode decomposition of field potentials from macaque V4 in visual spatial attention, Biol. Cybern. 92 (2005) 380–392, https://doi.org/10.1007/s00422-005-0566-y.
[8] L. Wang, G. Xu, S. Yang, W. Yan, Application of Hilbert-Huang Transform for the Study of Motor Imagery Tasks, (2008) 3848–3851.
[9] S. Aviyente, A. Tootell, E.M. Bernat, Time-frequency phase-synchrony approaches with ERPs, Int. J. Psychophysiol. 111 (2017) 88–97, https://doi.org/10.1016/j.ijpsycho.2016.11.006.
[10] W. Zhaohua, N.E. Huang, Ensemble empirical mode decomposition: a noise-assisted data analysis method, Biomed. Tech. 55 (2010) 193–201.
[11] M.A. Colominas, G. Schlotthauer, M.E. Torres, Improved complete ensemble EMD: a suitable tool for biomedical signal processing, Biomed. Signal. Process. Control. 14 (2014) 19–29, https://doi.org/10.1016/j.bspc.2014.06.009.
[12] A. Bruns, Fourier-, Hilbert- and wavelet-based signal analysis: are they really different approaches? J. Neurosci. Methods 137 (2004) 321–332, https://doi.org/10.1016/j.jneumeth.2004.03.002.
[13] O.A. Rosso, L. Romanelli, S. Blanco, L. Romanelli, R.Q. Quiroga, S. Blanco, R. Q. Quiroga, H. Garcia, O.A. Rosso, Stationarity of the EEG Series, IEEE Eng. Med. Biol. Mag. 14 (1995) 395–399, https://doi.org/10.1109/51.395321.
[14] A. Ahrabian, D. Looney, L. Stanković, D.P. Mandic, Synchrosqueezing-based time-frequency analysis of multivariate data, Signal Process. 106 (2015) 331–341, https://doi.org/10.1016/j.sigpro.2014.08.010.
[15] C. Li, M. Liang, A generalized synchrosqueezing transform for enhancing signal time–frequency representation, Signal. Process. 92 (2012) 2264–2274, https://doi.org/10.1016/J.SIGPRO.2012.02.019.
[16] F. Auger, P. Flandrin, Y.T. Lin, S. McLaughlin, S. Meignen, T. Oberlin, H.T. Wu, Time-frequency reassignment and synchrosqueezing: an overview, IEEE Signal. Process. Mag. 30 (2013) 32–41, https://doi.org/10.1109/MSP.2013.2265316.
[17] I. Daubechies, J. Lu, H.T. Wu, Synchrosqueezed wavelet transforms: an empirical mode decomposition-like tool, Appl. Comput. Harmon. Anal. 30 (2011) 243–261, https://doi.org/10.1016/j.acha.2010.08.002.
[18] K. Dragomiretskiy, D. Zosso, Variational mode decomposition, IEEE Trans. Signal. Process. 62 (2014) 531–544, https://doi.org/10.1109/TSP.2013.2288675.
[19] J. Harmouche, D. Fourer, F. Auger, P. Borgnat, P. Flandrin, The sliding singular spectrum analysis: a data-driven nonstationary signal decomposition tool, IEEE Trans. Signal. Process. 66 (2018) 251–263, https://doi.org/10.1109/TSP.2017.2752720.
[20] Z. Wu, N.E. Huang, On the filtering properties of the empirical mode decomposition, Adv. Adapt. Data Anal. 2 (2010) 397–414, https://doi.org/10.1142/S1793536910000604.
[21] D.P. Mandic, N. Ur Rehman, Z. Wu, N.E. Huang, Empirical mode decomposition-based time-frequency analysis of multivariate signals: The power of adaptive data analysis, IEEE Signal. Process. Mag. 30 (2013) 74–86, https://doi.org/10.1109/MSP.2013.2267931.
[22] Z. Wu, N.E. Huang, Ensemble empirical mode decomposition: a noise-assisted data analysis method, Adv. Adapt. Data Anal. 1 (2009) 1–41, https://doi.org/10.1142/S1793536909000047.
[23] T. Mujahid, A.U. Rahman, M.M. Khan, GPU-accelerated multivariate empirical mode decomposition for massive neural data processing, IEEE Access 5 (2017) 8691–8701, https://doi.org/10.1109/ACCESS.2017.2705136.
[24] K. Al-Subari, S. Al-Baddai, A.M. Tomé, M. Goldhacker, R. Faltermeier, E.W. Lang, EMDLAB: a toolbox for analysis of single-trial EEG dynamics using empirical mode decomposition, J. Neurosci. Methods 253 (2015) 193–205, https://doi.org/10.1016/j.jneumeth.2015.06.020.
[25] A. Delorme, S. Makeig, EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics including independent component analysis, J. Neurosci. Methods 134 (2004) 9–21, https://doi.org/10.1016/j.jneumeth.2003.10.009.
[26] P.J.J. Luukko, J. Helske, E. Räsänen, Introducing libeemd: a program package for performing the ensemble empirical mode decomposition, Comput. Stat. 31 (2016) 545–557, https://doi.org/10.1007/s00180-015-0603-9.
[27] P. Waskito, S. Miwa, Y. Mitsukura, H. Nakajo, Parallelizing Hilbert-Huang transform on a GPU, in: Proceedings of the 2010 First Int. Conf. Netw. Comput. ICNC 2010. (2010) 184–190. 〈https://doi.org/10.1109/IC-NC.2010.44〉.
[28] P. Waskito, S. Miwa, Y. Mitsukura, H. Nakajo, Evaluation of GPU-based empirical mode decomposition for off-line analysis, IEICE Trans. Inf. Syst. E94-D (2011) 2328–2337, https://doi.org/10.1587/transinf.E94.D.2328.
[29] J.D. Bonita, L.C.C. Ambolode, B.M. Rosenberg, C.J. Cellucci, T.A.A. Watanabe, P. E. Rapp, A.M. Albano, Time domain measures of inter-channel EEG correlations: A comparison of linear, nonparametric and nonlinear measures, Cogn. Neurodyn 8 (2014) 1–15, https://doi.org/10.1007/s11571-013-9267-8.
[30] K.P.Y. Huang, C.H.P. Wen, H. Chiueh, Flexible parallelized empirical mode decomposition in CUDA for hilbert huang transform, in: Proceedings of the Sixteenth IEEE Int. Conf. High Perform. Comput. Commun. HPCC 2014, Eleventh IEEE Int. Conf. Embed. Softw. Syst. ICESS 2014 Sixth Int. Symp. Cybersp. Saf. Secur. (2014) 1125–1133. 〈https://doi.org/10.1109/HPCC.2014.166〉.
[31] Y. Wang, H. Ren, M. Huang, Y. Chang, GPU-based Ensemble Empirical Mode Decomposition Approach to Spectrum Discrimination, Department of Computer Science and Information Engineering, National Central University, Taiwan Center for Space and Remote Sensing Research, National Central Universit, (2012) 3–6.
[32] H. Ren, Y.L. Wang, M.Y. Huang, Y.L. Chang, H.M. Kao, Ensemble empirical mode decomposition parameters optimization for spectral distance measurement in hyperspectral remote sensing data, Remote Sens. 6 (2014) 2069–2083, https://doi.org/10.3390/rs6032069.
[33] D. Chen, D. Li, M. Xiong, H. Bao, X. Li, GPGPU-aided ensemble empirical-mode decomposition for EEG analysis during anesthesia, IEEE Trans. Inf. Technol. Biomed. 14 (2010) 1417–1427, https://doi.org/10.1109/TITB.2010.2072963.
[34] Z. Wu, N.E. Huang, X. Chen, The multi-dimensional ensemble empirical mode decomposition method, Adv. Adapt. Data Anal. 1 (2009) 339–372, https://doi.org/10.1142/S1793536909000187.
[35] J.H. Halton, Algorithm 247: radical-inverse quasi-random point sequence, Commun. ACM 7 (1964) 701–702, https://doi.org/10.1145/355588.365104.

[36] Y. Zhang, J. Cohen, J.D. Owens, Fast tridiagonal solvers on the GPU, ACM Sigplan Not. 45 (2010) 127, https://doi.org/10.1145/1837853.1693472.

[37] A. Delorme, S. Makeig, EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics including independent component analysis, J. Neurosci. Methods 134 (2004) 9–21, https://doi.org/10.1016/j.jneumeth.2003.10.009.

[38] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (2009) 65, https://doi.org/10.1145/1498765.1498785.

Zeyu Wang (BEng in Elect. Eng. 2018, MEng in Biomedical Eng. 2021, Shenyang University of Technology, P.R. China) is currently a PhD student in the Faculty of Information Technology of the University of Pannonia, Hungary. His main research interests include GPU computing, EEG signal processing and data analysis, signal decomposition methods and machine learning algorithms. In his PhD research he studies EEG signal decomposition methods and their efficient, high-performance GPU implementations.

Zoltan Juhasz (MEng in Elect. Eng. 1989, Ph.D. in Comp. Sci. 1996, Budapest University of Technology) is an Associate Professor at the University of Pannonia in Hungary. He also worked at the Queen's University of Belfast and the University of Exeter. He teaches the Java Programming, Parallel Programming and Cloud Programming at BSc and MSc levels. His research interests include parallel and distributed computations from instruction level parallelism to large scale grid and cloud-HPC systems, GPU-based scientific computing, medical signal processing, human-computer interaction, and visualization. He was the principal investigator of several national research projects, participated in several international projects, and received equipment grants from Sun Microsystems and NVIDIA Inc. He also participated in several industrial R&D projects. His results are published in over 100 refereed research papers. He has served on the program committee of numerous conferences and works as a reviewer for several scientific journals.