GPU-Accelerated Signal Decomposition for Efficient EEG Processing: Methods and Applications

PhD Thesis Booklet

Zeyu Wang

Supervisor: Dr. Juhász Zoltán

Department of Electrical Engineering and Information Systems

Doctoral School of Information Science and Technology

University of Pannonia

Veszprém, Hungary

2025

1 Introduction

Electroencephalography (EEG) is a widely used, non-invasive method for monitoring brain activity in both clinical and research settings. It offers high temporal resolution and portability, but its nonlinear, non-stationary nature, combined with noise and artifacts, makes processing complex—especially for high-density, multi-channel datasets, where computational demands are high.

To address these challenges, this thesis explores advanced signal decomposition methods integrated with GPU-based parallel computing to enhance EEG analysis. Unlike CPUs, which are optimized for low latency, GPUs deliver high throughput through massively parallel execution, following the Single Program Multiple Data (SPMD) model. Kernel functions, executed across threads organized in blocks and grids, allow for simultaneous operations on large datasets.

Three GPU-accelerated EEG algorithms are introduced. First, the Improved Complete Ensemble Empirical Mode Decomposition with Adaptive Noise (CEEMDAN) reduces computational overhead in time-frequency analysis by efficiently handling noise-signal combinations. Second, the Multivariate Empirical Mode Decomposition (MEMD) leverages GPU parallelism to preserve inter-channel correlations in multi-channel EEG data, supporting large-scale connectivity analysis. Third, Independent Component Analysis (ICA) benefits from GPU tensor cores for accelerated matrix operations, significantly improving artifact removal and source localization in high-density recordings.

Together, these GPU-optimized techniques significantly enhance EEG processing efficiency, enabling rapid, scalable analysis for applications such as neurofeedback, seizure detection, cognitive assessment, and brain-computer interfaces. This work empowers large-scale neuroscience research with improved speed, accuracy, and scalability.

2 GPU Implementation of ICEEMDAN

I designed and implemented a massively parallel and performance-optimized GPU version of the Improved Complete Ensemble EMD with the Adaptive Noise (ICEEMDAN) algorithm that significantly reduced the execution time. The achieved speedup (up to 260x) enables researchers to rapidly execute data-driven time-frequency decomposition of non-stationary EEG signals. The resulting GPU program is the first known publicly available GPU implementation of this algorithm.

Empirical Mode Decomposition (EMD) is a signal decomposition algorithm [1] that can decompose the signal into a finite number of Intrinsic Mode Functions (IMFs) [2]. It can decompose the signal according to the time scale features of the data itself, without any pre-defined basis functions, which makes it fundamentally different from the Fourier or wavelet decomposition and suitable for the frequency decomposition of natural signals. However, the EMD method is not robust enough in the presence of noise, the intermittent noise in the signal can cause mode mixing in the decomposition result.

Ensemble Empirical Mode Decomposition (EEMD) [3], a noise-assisted decomposition method, is introduced to address the mode

mixing problem. By adding white noise to the raw signal, EEMD helps separate different frequency components more effectively, reducing the influence of intermittent noise and improving decomposition accuracy. The EEMD method replicates the signal by adding random white noise to it. Then, these signals are decomposed individually. The number of IMFs generated by each signal group may be inconsistent, leading to the inability to align each IMF during the averaging operation.

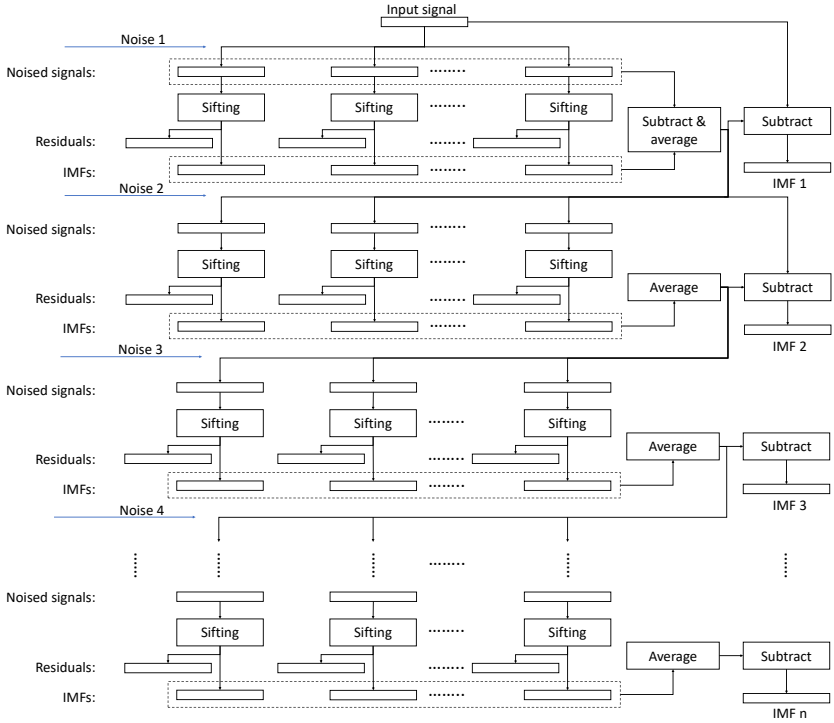

Figure 2-1: The execution flowchart of the ICEEMDAN algorithm.

The Improved Complete EEMD with Adaptive Noise (CEEMDAN) [4] [5] solves the mode alignment problem, provides an invertible decomposition and eliminates early noise components from the IMF set. Unlike in Ensemble Empirical Mode Decomposition (EEMD), where the IMFs are extracted for the different signal plus noise realizations independently and averaged at the end, in ICEEMDAN the extracted IMFs are averaged during the iterative process, and the average is used to compute the input signal for the next iteration. In addition, noise is added to the signal differently, to control the signal-to-noise ratio and match the frequency spectrum of the noise and the new input signals. For this, the EMD algorithm is executed on the Gaussian noise with zero mean and unit variance to extract noise IMFs, which will be added to the signal as the IMF extraction proceeds. The overall structure of the algorithm is depicted in Figure 2-1.

2.1 Related work

Several parallel implementations have been developed for the EMD algorithm. The library libeemd [6] is written in the C programming language and provides sequential and OpenMP-based parallel CPU implementations for the EMD, EEMD and CEEMDAN algorithms. The implementation achieves around 10x speedup compared to MATLAB ones. The rapid rise of GPU technology in High-performance Computing gave rise to several parallel GPU-accelerated EMD implementations, too. 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 C1060 and C2050 NVIDIA Tesla cards, respectively [7], [8]. Xie et al. created a CUDA EMD version for seismic data processing that

achieved 4x speedup on a GT240 GPU card [9]. Huang et al [10] reported 33.7x speedup on a C2050 GPU using overlapped piecewise cubic spline interpolation technique.

Since the EEMD has significantly higher computation cost than EMD due to the large number of noise-assisted copies of the original signal, parallelism in this case is mandatory to achieve acceptable execution times. The implementation by Wang et al. was 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 [11]. In a follow-up paper, they compared 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 [12]. Chen et al. developed a real-time CUDA EEMD implementation for anesthesia monitoring purposes [13]. They showed that it is possible to achieve real-time processing speed with a GTX295 GPU card (31.3x speedup, dual GPU card).

The reviewed GPU implementations have the following characteristics in common: (i) they use early generation, by now outdated GPU processors and early versions of the CUDA programming language; (ii) the achieved speedup values are relatively modest; and (iii) source code is not publicly available. More importantly, a high-performance GPU implementation of the more time-consuming ICEEMDAN algorithm is still missing.

2.2 Methods

The algorithm, described formally, is as follows: let $Ek(\cdot)$ represent the operator that returns the k^{th} IMF (mode) using the EMD algorithm of its input signal. Let $M(\cdot)$ be the operator that returns the local mean of the upper and lower envelopes of the signal it is applied to. Let $w^{(i)}$ be a realization of white Gaussian noise with zero mean and unit variance, and let $\langle\cdot\rangle$ be the action of averaging across all realizations. With these operators, the parallel pseudocode of the ICEEMDAN algorithm is shown as follows:

Parallel IEEMDAN Algorithm:

for all channels **do in parallel**
  generate noise signal $w^{(i)}$ and its IMFs $E_k\left(w^{(i)}\right)$ by EMD
  for all realizations **do in parallel**
    add noise IMF $E_k\left(w^{(i)}\right)$ to signal x to obtain the current realization
    compute the local means $M\left(x^{(i)}\right)$ **in parallel**
  end for
  average $M\left(x^{(i)}\right)$ across realizations to get r_1 in parallel
  compute first mode d_1 as $d_1 = x - r_1$
  while no more IMFs can be extracted **do** {for k=2 and up}
    for all realizations **do in parallel**
      compute the local means $M\left(x^{(i)}\right)$ **in parallel**
    end for
    average $M\left(x^{(i)}\right)$ across realizations (r_k) **in parallel**
    compute mode d_k as $d_k = r_{k-1} - r_k$ **in parallel**
  end while
end for

Table 2-1: Pseudo-code of parallel ICEEMDAN algorithm.

The GPU implementation of ICEEMDAN is divided into two parts:

EMD processing of the Gaussian noise to generate noise IMFs, and EMD processing of noise-added input signal. Each numerical step in ICEEMDAN is implemented by one or several custom kernel functions or highly optimized CUDA library functions. The kernel is a function executing on the GPU device using multiple threads in a single-instruction-multiple-stream fashion. Each thread has a unique index in order to map a thread to a data element in memory. Kernels are launched on the host (CPU program) to be executed on the GPU with threads organized into blocks, and blocks into grids. One-, two- and three-dimensional indexing can be used to map threads onto 1D, 2D and, 3D data structures. NVIDIA GPUs contain a large number of compute (integer, FP32, FP64 and tensor) cores; the internal thread schedulers will assign instructions from threads to different cores based on the operand type for parallel execution.

In the initialization phase, memory is allocated for all variables and the noise IMFs are generated for ICEEMDAN. Unlike other noise-assisted EMD variants, the noise added in ICEEMDAN is the IMFs generated from Gaussian noise decomposition rather than Gaussian noise itself, so before the real decomposition of the input signal, the EMD processing needs to be performed on the Gaussian noise to generate noise IMFs which will be added as noises.

The core step of the ICEEMDAN algorithm is the parallel GPU implementation of the sifting algorithm that extracts one mode (IMF) from the input signal. This is used in extracting both the noise and signal IMFs. The main steps of the Sifting process are: extrema detection, envelope generation with cubic spline interpolation, local mean computation and signal residue calculation. These steps are

implemented by custom kernel functions and in some cases using highly optimized CUDA library functions. The functions used in the sifting process implementation are listed in Table 2-2.

Sifting operation	Kernel/library function
Find local signal maxima	select_extrema_max()
Find local signal minima	select_extrema_min()
Solve the tridiagonal system	*cusparseSgtsv2_nopivot()*
Coefficients collection	spline_coefficient()
Cubic spline interpolation	interpolate()
Compute mean envelope	averageUpperLower()
Signal update for next iteration	averageUpdateSignal()

Table 2-2: The major steps of the sifting process and their corresponding GPU kernel/library function, the Italic font represents the CUDA library function.

Each of these steps are executed in a massively parallel fashion. In the extrema detection step, one thread is launched for each signal/noise sample that compares the sample with the left and right neighbors to detect minima and maxima values and their locations. The extrema are used in the next step to generate the upper and lower envelopes of the signal. This is performed by cubic spline interpolation based on the extrema values, which requires the solution of many tri-diagonal systems of equations. The Parallel Cyclic Reduction solver implementation, `cusparseSgtsv2_nopivot()`, provided in the cuSPARSE library is used for this step. The solver provides us with a set of spline coefficients that we compute in function `spline_coefficient()`. These are inputs to the interpolation kernel `interpolate()` that returns the interpolated values of the upper and lower envelopes. The mean of these two envelopes is calculated in parallel in function `averageUpperLower()` as sample wise means.

Once the Sifting process is completed, the new first mode for each realization is obtained. These modes are subtracted from their corresponding realizations and the results are averaged across the realizations to produce the new residue r_{k+1}. These steps are executed by `produceResidue()` and `averageUpdateSignal()`.

2.3 Results

This section demonstrates the numerical accuracy, speedup achieved, and efficiency of the ICEEMDAN GPU implementation. The numerical correctness of the implementation was validated with a synthetic signal and a real EEG dataset (in the full thesis) provided as a sample data file in the EEGLAB software distribution. The GPU implementation was compared with a MATLAB implementation considered as the golden standard. To quantify the accuracy of the decomposition results obtained with different implementations, I introduce the Similarity Index metric ρ given as:

$$\rho_i\big(x_i(t), y_i(t)\big) = \frac{cov\big(x_i(t), y_i(t)\big)}{\sqrt{var\big(x_i(t)\big)}\sqrt{var\big(y_i(t)\big)}}$$

where $cov()$ represents covariance of the two input IMF signals $x_i(t)$ and $y_i(t)$ produced by the GPU and MATLAB implementations, respectively, and $var()$ represents the variance of the input signal. The index ρ varies between 0 and 1, $\rho = 1$ representing that $x_i(t)$ and $y_i(t)$ are identical.

Figure 2-2 (b) and (c), show the decomposition results of a dual-frequency synthetic signal s, performed with the reference MATLAB and the CUDA implementations, respectively. The Similarity Index in

this case is computed between one constituent component of signal s (the ground truth) and the IMF produced by either implementation; i.e. I measure how accurately the IMFs reproduce the original components of the raw synthetic signal. The MATLAB implementation gives similarity index values $\rho_{s1}^{MATLAB} = 99.63\%$ and $\rho_{s2}^{MATLAB} = 99.95\%$. The CUDA implementation produced nearly identical results as the MATLAB one, $\rho_{s1}^{CUDA} = 99.62\%$ and $\rho_{s2}^{CUDA} = 99.91\%$.

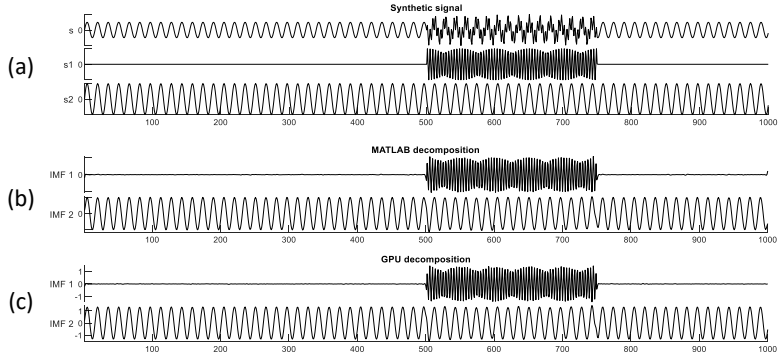

Figure 2-2: The synthetic dual-frequency signal (top), and the decomposition results from the MATLAB (middle) and CUDA (bottom) implementations.

The achieved speedup of the optimized GPU implementation is calculated compared to MATLAB, which is shown for different number of sifting iterations in Figure 2-3. It is important to note that the speedup increases with sample size and in a super-linear fashion. That is, the more samples we process, the faster the GPU version becomes compared to the MATLAB version.

A Roofline Model analysis is performed on performance critical kernels of the GPU implementation at two different signal lengths (4k, 100k) to illustrate the efficiency.

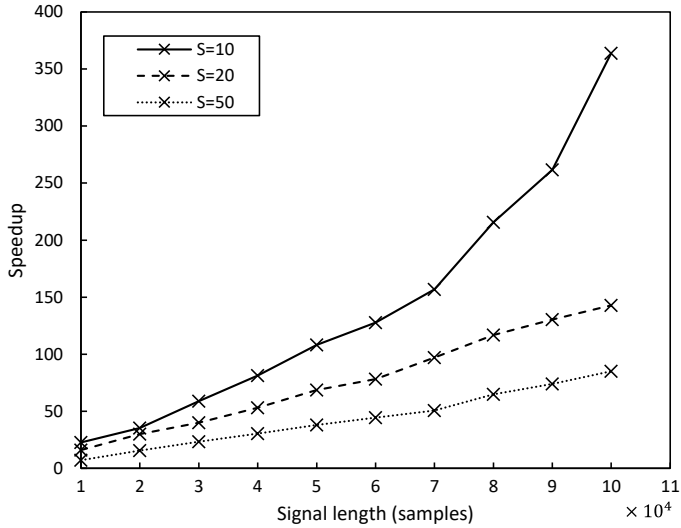

Figure 2-3: Speedup of the GPU implementation (executed on A100) over the MATLAB version in function of signal length N and varying number of sifting iterations S. The number of realizations is fixed, I=500.

As can be seen in Figure 2-4, all kernel functions are memory-bound based on their Arithmetic Intensity (AI), that is, the performance is limited by the memory bandwidth not the computational performance of the GPU. The green boxes represent the kernels part of the NVIDIA library, while the blue dots represent the customized self-develop kernels. The closer the dots are to the performance boundary, the more efficient the kernels are. Kernels significantly below the line vertically indicate performance problems, typically latency issues. The results indicate that kernels are closer to the theoretical performance limit (performance attainable at a given arithmetic intensity value) than the NVIDIA kernels. The arrows in the figure indicate the performance change of the kernels when increasing the signal length from 4k to

100k. The subsequent change in the kernels' position in the Roofline diagram suggests that my implementation becomes more efficient as signal size increases.

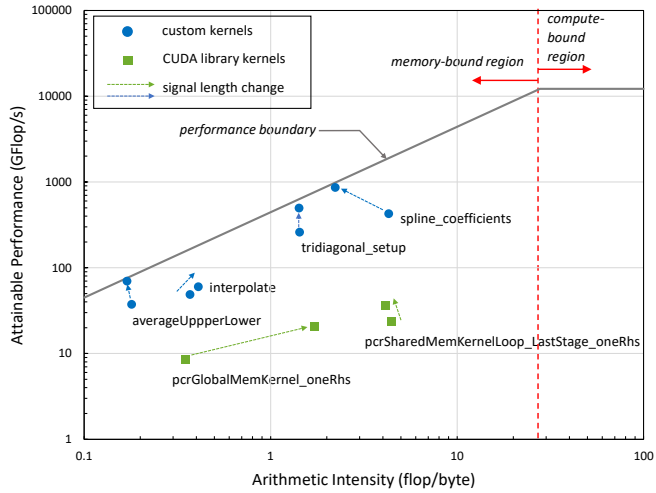

Figure 2-4: Speedup of the GPU implementation (executed on A100) over the MATLAB version in function of signal length N and varying number of sifting iterations S. The number of realizations is fixed, I=500.

3 GPU Implementation of MEMD

I designed and implemented an efficient GPU version of the Multivariate Empirical Mode Decomposition (MEMD) method for speeding up the process of decomposing non-stationary multi-channel bioelectric signals into different oscillation modes. The final GPU program is the fastest known GPU implementation, compared to MATLAB, it achieves up to 430x speedup depending on the size of the

input signal. This opens the way for researchers to perform empirical mode-based time-frequency analysis correctly, ensuring synchronized frequency band representations of the channels in timely manner.

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) provides a solution to this problem.

The details of the MEMD algorithm are outlined in Table 3-1. First, the multivariate input signal will be projected onto the direction vectors and the projected signal on each direction vector will be obtained. 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.

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**

Table 3-1: Pseudo-code of MEMD algorithm.

3.1 Related work

In addition to the GPU implementations of single-channel EMD variants such as EMD, EEMD, and CEEMD described in the previous section, efforts have also been made to extend these methods to

multichannel signals. Chen et al. developed an implementation of the multidimensional EEMD algorithm [14], while Mujahid et al. reported the first and only multivariate EMD (MEMD) GPU implementation [15]. Their study compared a GPU-optimized MEMD implementation with a MATLAB implementation on six-variable and sixteen-variable datasets, achieving a 6-16x speedup over the MATLAB version.

Although they leveraged multiple levels of parallelism and employed various GPU optimization techniques, their work was limited by the use of early-generation GPUs and an outdated version of the CUDA programming language. More importantly, their performance improvements were relatively modest, they did not test on larger EEG datasets, and their source code was not publicly available, limiting further advancements and practical adoption.

3.2 Methods

In my 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.

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. In the parallel implementation of MEMD, I allocate memory for all variables in GPU memory before the computation starts, and the memory exchange between CPU and GPU is limited to 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). The variables required by each GPU kernel function, along with their size and memory layout, as well as additional technical details of the implementation—including preprocessing, signal projection onto the direction vector, extreme value detection, and cubic spline interpolation—are detailed in the complete thesis

3.3 Results

In this section, I demonstrate the numerical accuracy, speedup, and efficiency of the proposed GPU implementation of the MEMD algorithm. A hexavariate dataset is generated 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 x_5 in Channels 1, 3-4 and 6.

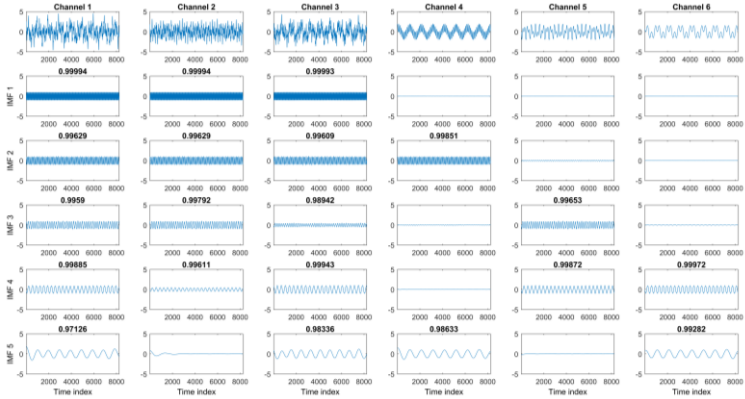

Figure 3-1: Decomposition results of the synthetic hexavariate signal with Similarity Index values shown in bold for each component.

Figure 3-1 shows the result of the decomposition of this dataset including the Similarity Index values above each IMF component. The correct mode alignment is evident, and the lowest Similarity Index value is 0.971, while the majority of similarity index values are above 0.99. Additional results on test data, including synthetic signals with noise and real EEG data, are available in the full thesis.

Next, the MATLAB MEMD execution time was compared with the proposed GPU implementation executed on various GPU cards (RTX 3070 mobile, Titan XP, Tesla V100). The implementations were performed by varying the number of input channels, the number of samples per channel, and the number of direction vectors.

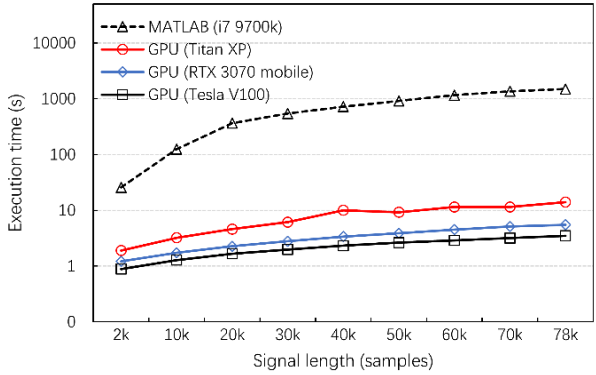

Figure 3-2: 32-channel 64 direction vectors, up to 78k samples, MATLAB vs GPU execution times.

The execution time and speedup results of a 32 channel EEG dataset are shown in Figure 3-2 and Figure 3-3. 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 i7-9700K CPU. More performance test cases are available in the full thesis.

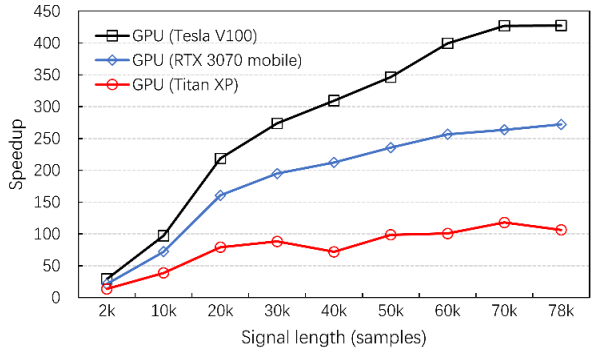

Figure 3-3: Speedup for 32-channel 64 direction vectors, up to 78k samples, compared to MATLAB.

I performed a Roofline 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. Figure 3-4 illustrates the performance of the most critical kernels of the proposed 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 I developed for this implementation. The figure shows that my kernels reach close to theoretical performance (lie on or close to the global memory performance boundary line), while the NVIDIA kernels perform relatively poorly.

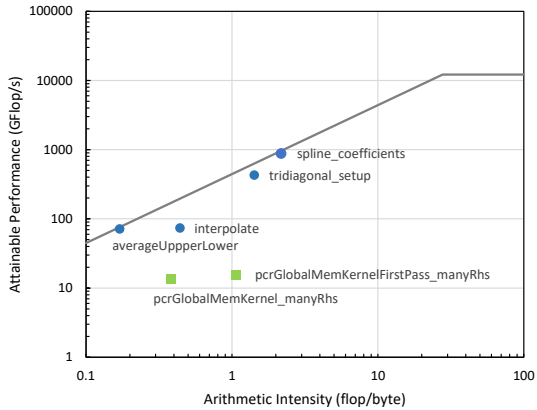

Figure 3-4: The Roofline performance model of the RTX 3070 mobile GPU showing the performance positions of the main kernels of the implementation.

4 GPU Implementation of ICA

I examined the use of GPU tensor cores for the implementation of the Independent Component Analysis (ICA) algorithm. I demonstrated that tensor cores can be used efficiently in low-level CUDA kernels, providing more flexibility and optimization opportunities that are achievable with high-level numerical library functions. Thisis the first known tensor core-based ICA implementation and the fastest to date. Compared to the widely used MATLAB implementation, it achieves up to 43x speedup depending on the size of the input signal. The new implementation significantly reduces the execution time of artifact removal and preprocessing of large-scale, high-density EEG datasets.

The de facto standard method for removing artifacts in EEG measurement is based on Independent Component Analysis (ICA),

which is able to decompose the measured contaminated (mixed) signal into statistically independent source components. EEG signals are a linear mixture of spontaneous neural signals and artifact signals, and studies have shown that ICA can separate neural signals from artifacts. There are several variants of ICA, such as FastICA, Infomax ICA, SOBI, JADE, and AMICA, but for human EEG signals, Infomax ICA is considered as the best choice. The pseudocode of the computational algorithm of Infomax ICA is shown in Table 4-1. However, the ICA algorithm is computational intensity, driven by the matrix multiply-add operations in the algorithm, brings challenges, particularly when working with large datasets. For large studies, execution running for days is not uncommon.

Infomax Independent Component Analysis

1. Initialize $W(0)$
2. **while** (not converged and t < limit)
    2.1 **Y** = permutate(**X**)
    2.2 **for** each block **B** in signal **Y**
        2.2.1 $\mathbf{U} = \mathbf{WY} + bias$ // Step 1
        2.2.2 $\mathbf{Y} = \tanh(\mathbf{U})$ // Step 1
        2.2.3 $\mathbf{W}^* = \mathbf{W} + \eta(\mathbf{I} - \mathbf{YU}^{\mathrm{T}} - \mathbf{UU}^{\mathrm{T}})\mathbf{W}$ // Steps 2, 3
        2.2.4 update bias // Step 4
        end for
end while

Table 4-1: Pseudocode of Infomax Independent Component Analysis.

In the pseudocode above, **W** is the weight matrix and η represents the learning rate. A high learning rate may accelerate the iterative process, but it may cause non-convergence. A lower learning rate helps convergence but requires more iterations and spend more time. **I** and **Y** represent the identity matrix and the observed signal, respectively. The

non-linear function $\tanh(\mathbf{Y})$ is used for signals with super-Gaussian distributions, while $\mathbf{Y} - \tanh(\mathbf{Y})$ is used for sub-Gaussian one.

4.1 Related work

The only known implementation of Infomax ICA on GPU is CUDAICA, proposed by Raimondo et al. [16], which includes both a CUDA library function version and a GPU kernel function version. They tested their implementation on NVIDIA GTX560 and Tesla C2070 and developed different CPU versions based on ATLAS (a portable self-optimizing BLAS) and Intel's Math Kernel Library (MKL). Additionally, a Windows-compatible version of CUDAICA was later provided by a Chinese researcher.

Overall, CUDAICA achieved a 4.5-20x speedup compared to CPU implementations. However, they did not benchmark their implementation against MATLAB ICA, which is the most widely used in the EEG processing community. Given the high-performance tensor cores available in modern GPUs, there is significant potential for further accelerating ICA computations on GPUs.

4.2 Methods

As described in The only known implementation of Infomax ICA on GPU is CUDAICA, proposed by Raimondo et al. [16], which includes both a CUDA library function version and a GPU kernel function version. They tested their implementation on NVIDIA GTX560 and Tesla C2070 and developed different CPU versions based on ATLAS (a portable self-optimizing BLAS) and Intel's Math Kernel Library (MKL). Additionally, a Windows-compatible version of CUDAICA was later provided by a Chinese researcher.

Overall, CUDAICA achieved a 4.5-20x speedup compared to CPU implementations. However, they did not benchmark their implementation against MATLAB ICA, which is the most widely used in the EEG processing community. Given the high-performance tensor cores available in modern GPUs, there is significant potential for further accelerating ICA computations on GPUs.

The ICA algorithm contains a large number of iterations (iterations on data blocks and gradient convergence), and each iteration involves a large number of matrix multiply-add operations, which is one of the main performance bottlenecks of the ICA algorithm. Therefore, optimizing matrix multiply-add operations is crucial to accelerating the ICA algorithm. Fortunately, the tensor core is designed for matrix multiply-add operations.

Similarly to the well-known CUDA core, the tensor core is also a computing unit in the Streaming Multiprocessor (SM). Still, the difference is that the processing object of the tensor core is a set of matrices rather than a single value processed by the CUDA core. Each tensor core provides a 4×4×4 matrix processing array that performs the operation D = A * B + C, where A, B, C, and D are 4×4 matrices, which means that each tensor core can perform 64 floating-point Fused-Multiply-Add (FMA) operations per clock cycle, this is 64 times faster than the CUDA core. Therefore, for implementing algorithms involving many matrix multiplication and addition operations, tensor cores can provide multiple times the performance of the CUDA cores.

Each step in ICA is implemented by a kernel function, which essentially describes the behaviour of a single thread running on GPU.

When launching the kernel function, the number and shape of the threads should be specified, and each thread indexes the data in memory based on its "thread index" and subsequently writes the result back to memory. Typically, at a certain moment, one thread is executed by one CUDA core, but for tensor cores, 32 threads will be regarded as a warp, and one warp is processed by multiple tensor cores. A warp can provide the processing of a set of 16×16×16 matrix multiplication. Therefore, when utilizing tensor cores, the 16×16 tile matrix serves as the smallest operational unit.

High-performance matrix multiply-accumulate operation by tensor cores

for each **grid** in matrix **C**
    1. Stream submatrices of **C** from global memory to shared memory
    2. Load submatrices from shared memory to register file
    for each **block** in **A** and **B**
        for each **chunk** in **block**
            3. Stream chunks of **A** and **B** to shared memory
            for each **slice** in **chunk**
                4. Load slices of **A** and **B** to register file
                5. Perform the WMMA operation on two slices
            end for
        end for
    end for
    6. Load submatrices of **D** from register file to shared memory
    7. Stream submatrices from shared memory to global memory
end for

Table 4-2: Pseudocode of the high-performance matrix multiply-accumulate operation implemented with tensor cores and three-level memory hierarchy.

As shown in Figure 4-1, a higher-performance implementation of matrix multiply-add operation using tensor cores and three-level

memory hierarchy (global memory, shared memory, register file) is proposed to execute all the matrix multiply-add operations in ICA. In this implementation, one thread block is responsible for a 128×128 submatrix in the result matrix (C and D), so each thread block will traverse all corresponding strip submatrices in the input matrix (A and B), and perform multiply-accumulate operations on them. The pseudocode of the computation process is shown in Table 4-2.

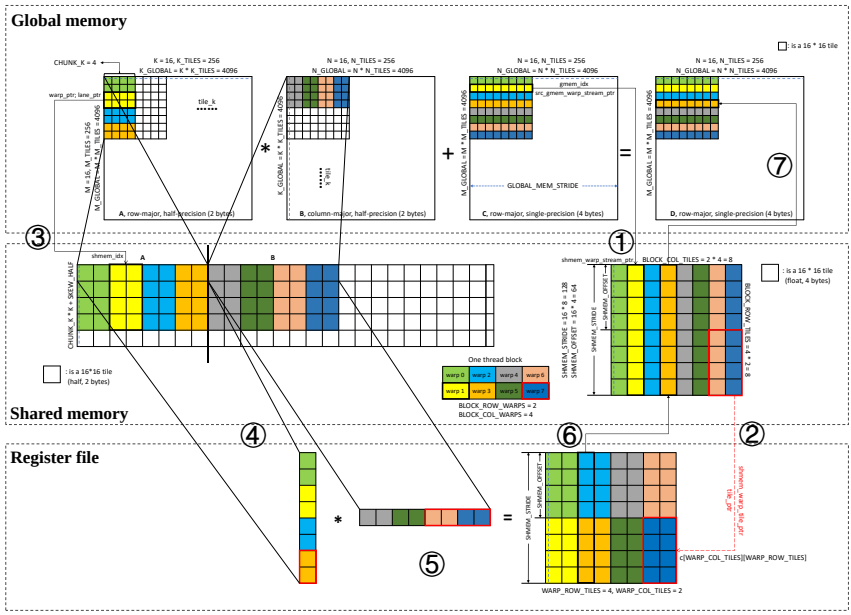

Figure 4-1: High-performance implementation of matrix multiply-accumulate operation powered by tensor cures and shared memory. Elements of different colors in matrices are handled by their same-colored warps.

4.3 Results

This section presents the test results of a prototype ICA implemented

by tensor cores. First, I show the numerical validation results of the tensor core version of the ICA, using the MATLAB ICA as benchmark and comparison. Then, I show the speedup under different numbers of channels and signal lengths. Finally, the kernel functions profiling results of the proposed ICA implementation are presented.

A 128×2 million (128 channels, 2 million samples per channel) synthetic input signal and a 128×128 unmixing matrix were generated to feed the MATLAB and tensor core versions of ICA implementations respectively. In the test, in order to verify the results more efficiently, all the elements in the input signal and unmixing matrix were normalized to 0.01. and the input signal was sliced into 4096 128×512 data blocks, each of which was used to update the unmixing matrix.

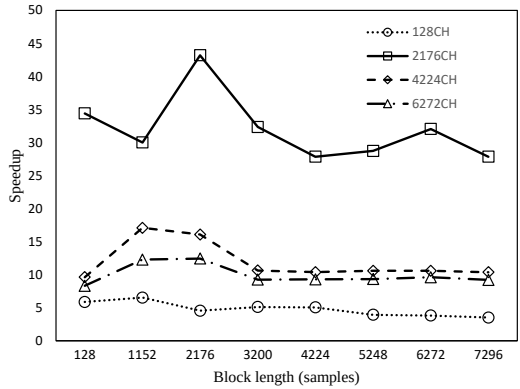

Figure 4-2: Speedup of ICA implemented by the tensor core for different number of channels and signal lengths, compared to MATLAB.

In the performance evaluation, the number of data blocks is fixed at 16, and input signals with varying block lengths ranging from 128 to 7296 and channel numbers ranging from 128 to 6272 are utilized to test the

performance of ICA implemented by tensor cores. As illustrated Figure 4-2, the speedup of the tensor core version of ICA compared to the MATLAB version varies with different data sizes. Since both use a sequential strategy for traversing data blocks, increasing the input data length does not result in a higher speedup, while the increase in the number of channels brings a 3-43x speedup for the tensor core version of ICA compared to the MATLAB version.

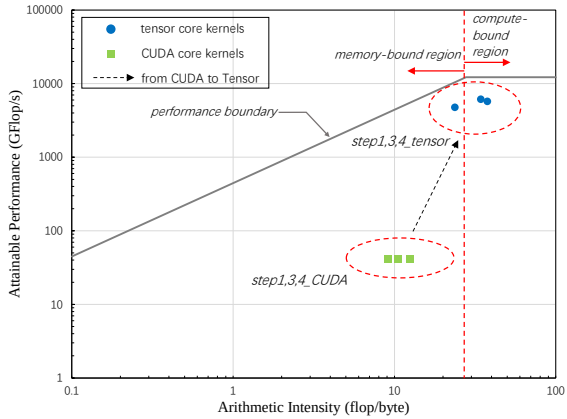

Figure 4-3: The Roofline performance results of the kernels from CUDA core and tensor core versions of ICA, executed on the RTX 3070 mobile GPU.

In addition to the speedup in execution time compared to the MATLAB implementation, I also analyzed the performance of the kernel functions in the tensor core version of the ICA using the Roofline Model. As shown in Figure 4-3, the kernel functions in the ICA implementations based on the tensor core and CUDA core are profiled on the Roofline model of the RTX3070 mobile GPU. The kernel functions based on the CUDA core are all memory-bound and far from the performance boundary of the hardware, while the kernel

functions of the tensor core version have more advantages in memory throughput, improve the arithmetic density, and are closer to the theoretical performance of the hardware.

5 Conclusions and contributions

This dissertation explored the application and analysis of signal decomposition methods in EEG signal processing, with a particular emphasis on leveraging GPU parallelism to enhance computational efficiency. The research addressed the challenges posed by the nonlinear and non-stationary nature of EEG signals, the need for advanced decomposition techniques, and the computational demands associated with high-resolution EEG data analysis. The major contributions of this thesis include:

Thesis I. *GPU implementation of ICEEMDAN [1], [2], [3]*

I designed and implemented a massively parallel and performance-optimized GPU version of the Improved Complete Ensemble EMD with the Adaptive Noise (ICEEMDAN) algorithm that significantly reduced the execution time. The achieved speedup (up to 260x) enables researchers to rapidly execute data-driven time-frequency decomposition of non-stationary EEG signals. The resulting GPU program is the first known publicly available GPU implementation of this algorithm.

ICEEMDAN improves upon traditional Empirical Mode Decomposition (EMD) by addressing mode mixing issues and enhancing signal reconstruction accuracy. However, its computational complexity makes it impractical for large EEG datasets. This thesis

introduces a massively parallel CUDA-based implementation that achieves an astonishing two-orders-of-magnitude speedup, reducing processing times from hours to mere seconds. The optimization involves restructuring key computational steps, such as intrinsic mode function (IMF) extraction and noise addition, to maximize parallel execution across GPU cores. Extensive performance benchmarking across multiple GPU architectures (Pascal, Volta, and Ampere) confirms significant efficiency gains. Validation using both synthetic and real EEG signals demonstrates that the GPU implementation maintains numerical accuracy while dramatically accelerating computation. Moreover, a comprehensive performance analysis using the Roofline model shows that the optimized implementation approaches the hardware's theoretical performance limits. To support further research in EEG processing, the open-source code for the GPU-based ICEEMDAN is made publicly available.

Thesis II. *GPU implementation of MEMD [4], [5]*

I designed and implemented an efficient GPU version of the Multivariate Empirical Mode Decomposition (MEMD) method for speeding up the process of decomposing non-stationary multi-channel bioelectric signals into different oscillation modes. The final GPU program is the fastest known GPU implementation, compared to MATLAB, it achieves up to 430x speedup depending on the size of the input signal. This opens the way for researchers to perform empirical mode-based time-frequency analysis correctly, ensuring synchronized frequency band representations of the channels in timely manner.

Unlike conventional EMD, MEMD preserves inter-channel correlations while decomposing non-stationary signals. This thesis

presents a novel GPU-optimized implementation that minimizes CPU-GPU communication overhead and incorporates efficient tridiagonal solvers and spline interpolation kernels to maximize throughput. The results demonstrate a remarkable 180x to 430x speedup over traditional CPU implementations, significantly reducing processing times from days to minutes. Extensive benchmarking confirms scalability across different EEG dataset sizes and GPU architectures. Notably, the performance gains extend beyond EEG processing, making the GPU-accelerated MEMD applicable to other domains such as seismic signal analysis, power optimization, and biomedical signal processing. This broad applicability enhances the impact of MEMD as a tool for analyzing high-dimensional, nonlinear, and non-stationary signals.

Thesis III. *GPU implementation of ICA [6]*

I examined the use of GPU tensor cores for the implementation of the Independent Component Analysis (ICA) algorithm. I demonstrated that tensor cores can be used efficiently in low-level CUDA kernels, providing more flexibility and optimization opportunities that are achievable with high-level numerical library functions. This is the first known tensor core-based ICA implementation and the fastest to date. Compared to the widely used MATLAB implementation, it achieves up to 43x speedup depending on the size of the input signal. The new implementation significantly reduces the execution time of artifact removal and preprocessing of large-scale, high-density EEG datasets.

ICA is widely used for separating neural signals from artifacts such as eye blinks and muscle activity, but its computational demands often

make it infeasible for high-density measurement or large datasets. This thesis proposes the first ICA implementation that leverages tensor cores, specialized GPU units designed for highly efficient matrix multiply-add operations. The optimized implementation achieves up to a 43x speedup over MATLAB-based ICA, enabling efficient spatial decomposition and artifact removal for massive EEG datasets. A three-level memory hierarchy is introduced to reduce memory exchange overhead, while a potential block-parallel ICA processing approach eliminates serialization at the data block level, ensuring seamless parallel execution of multiple EEG channels. Validation experiments confirm that the tensor-core-based ICA produces results consistent with standard CPU implementations while maintaining high numerical accuracy. Additionally, this thesis highlights potential future optimizations, such as multi-GPU parallelization and hybrid execution strategies, which could further enhance ICA efficiency. By drastically reducing processing times from hours to seconds, this research paves the way for real-time EEG preprocessing, benefiting applications such as brain-computer interfaces, neurofeedback, and clinical diagnostics.

Publications related to thesis

(https://m2.mtmt.hu/gui2/?type=authors&mode=browse&sel=10081341)

[1] Z. Wang, Z. Nagy, Z. Juhasz, “On the Benefits of Empirical Mode Decomposition in Spatio-temporal EEG Analysis”, 45nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, May 23 - 27, 2022, ISSN: 1847-3946.

[2] Z. Wang, Z. Juhasz, “Analysis of the Effects of Input Parameter Settings on the Quality of Electrophysiological Signal Decomposition in Empirical Mode Decomposition”, XXXV. Neumann Colloquium, Szeged, Hungary, Nov 24 - 25, 2022, ISBN: 978-615-5036-22-4.

[3] Z. Wang and Z. Juhasz, “GPU Implementation of the Improved CEEMDAN Algorithm for Fast and Efficient EEG Time-Frequency Analysis”, MDPI Sensors (Basel), vol. 23, no. 20, p. 8654, Oct. 2023, doi: 10.3390/S23208654/S1.

[4] Z. Wang and Z. Juhasz, “Efficient GPU implementation of the multivariate empirical mode decomposition algorithm”, Journal of Computational Science, vol. 74, p. 102180, Dec. 2023, doi: 10.1016/J.JOCS.2023.102180.

[5] Z. Wang and Z. Juhasz, “Massively Parallel EEG Algorithms for Pre-exascale Architectures”, European Conference on Parallel Processing (Euro-Par 2023), Limassol, Cyprus, Aug 28 - Sept 1, 2023, doi: 10.1007/978-3-031-48803-0_34

[6] Z. Wang, Gy. Kozmann, Z. Juhasz, “Towards a High-Performance Independent Component Analysis Implementation on GPUs”, XXXIV. Neumann Colloquium, Veszprem, Hungary, Dec 3 - 4, 2021, ISBN: 978-963-396-217-6.

References

[1] N. E. Huang *et al.*, “The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis,” *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 454, no. 1971, pp. 903–995, 1998, doi: 10.1098/RSPA.1998.0193.

[2] G. Wang, X. Y. Chen, F. L. Qiao, Z. Wu, and N. E. Huang, “On Intrinsic Mode Function,” *Advances in Data Science and Adaptive Analysis*, vol. 2, no. 3, pp. 277–293, Jul. 2010, doi: 10.1142/S1793536910000549.

[3] Z. Wu and N. E. Huang, “Ensemble Empirical Mode Decomposition: a Noise-Assisted Data Analysis Method,” *Advances in Data Science and Adaptive Analysis*, vol. 1, no. 1, pp. 1–41, Jan. 2009, doi: 10.1142/S1793536909000047.

[4] M. E. Torres, M. A. Colominas, G. Schlotthauer, and P. Flandrin, “A complete ensemble empirical mode decomposition with adaptive noise,” *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 4144–4147, 2011, doi: 10.1109/ICASSP.2011.5947265.

[5] M. A. Colominas, G. Schlotthauer, and M. E. Torres, “Improved complete ensemble EMD: A suitable tool for biomedical signal processing,” *Biomed Signal Process Control*, vol. 14, no. 1, pp. 19–29, Nov. 2014, doi: 10.1016/J.BSPC.2014.06.009.

[6] P. J. J. Luukko, J. Helske, and E. Räsänen, “Introducing libeemd: a program package for performing the ensemble empirical mode decomposition,” *Comput Stat*, vol. 31, no. 2, pp. 545–557, Jun. 2016, doi: 10.1007/S00180-015-0603-9/METRICS.

[7] P. Waskito, S. Miwa, Y. Mitsukura, and H. Nakajo, “Evaluation of GPU-Based Empirical Mode Decomposition for Off-Line Analysis,” *IEICE Trans. Inf. Syst.*, vol. E94-D, no. 12, pp. 2328–2337, 2011, doi: 10.1587/TRANSINF.E94.D.2328.

[8] P. Waskito, S. Miwa, Y. Mitsukura, and H. Nakajo, “Parallelizing Hilbert-Huang transform on a GPU,” *Proceedings - 2010 1st International Conference on Networking and Computing, ICNC 2010*, pp. 184–190, 2010, doi: 10.1109/IC-NC.2010.44.

[9] W. M. Yu, K. Xie, H. Q. Yu, P. Wu, T. Li, and M. F. Peng, “Hilbert-Huang Transformation of Large Seismic Data Based on GPU,” *International Symposium on Industrial Electronics*, pp. 249–252,

2011, doi: 10.1109/ISIE.2011.35.

[10] K. P. Y. Huang, C. H. P. Wen, and H. Chiueh, “Flexible Parallelized Empirical Mode Decomposition in CUDA for Hilbert Huang Transform,” *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pp. 1125–1133, Mar. 2014, doi: 10.1109/HPCC.2014.166.

[11] Y. L. Wang, H. Ren, M. Y. Huang, and Y. L. Chang, “GPU-based Ensemble Empirical Mode Decomposition approach to spectrum discrimination,” *Workshop on Hyperspectral Image and Signal Processing, Evolution in Remote Sensing*, 2012, doi: 10.1109/WHISPERS.2012.6874288.

[12] H. Ren, Y. L. Wang, M. Y. Huang, Y. L. Chang, and H. M. Kao, “Ensemble Empirical Mode Decomposition Parameters Optimization for Spectral Distance Measurement in Hyperspectral Remote Sensing Data,” *Remote Sensing 2014, Vol. 6, Pages 2069-2083*, vol. 6, no. 3, pp. 2069–2083, Mar. 2014, doi: 10.3390/RS6032069.

[13] D. Chen, D. Li, M. Xiong, H. Bao, and X. Li, “GPGPU-aided ensemble empirical-mode decomposition for EEG analysis during anesthesia,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 6, pp. 1417–1427, Nov. 2010, doi: 10.1109/TITB.2010.2072963.

[14] Z. Wu, N. E. Huang, and X. Chen, “The multi-dimensional ensemble empirical mode decomposition method,” *Adv Adapt Data Anal*, vol. 1, no. 3, pp. 339–372, Jul. 2009, doi: 10.1142/S1793536909000187.

[15] T. Mujahid, A. U. Rahman, and M. M. Khan, “GPU-Accelerated Multivariate Empirical Mode Decomposition for Massive Neural Data Processing,” *IEEE Access*, vol. 5, pp. 8691–8701, 2017, doi: 10.1109/ACCESS.2017.2705136.

[16] F. Raimondo, J. E. Kamienkowski, M. Sigman, and D. Fernandez Slezak, “CUDAICA: GPU Optimization of Infomax-ICA EEG Analysis,” *Comput Intell Neurosci*, vol. 2012, Jan. 2012, doi: 10.1155/2012/206972.