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

A Thesis Submitted for the Degree of Doctor of Philosophy in Computer Science

DOI:10.18136/PE.2025.944

Zeyu Wang

Supervisor:
Dr. Juhá szZoltán

Department of Electrical Engineering and Information Systems

Doctoral School of Information Science and Technology

University of Pannonia

Veszprém, Hungary

2025

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

The thesis was prepared for the award of a doctoral degree (PhD) within the framework of the Doctoral School of Information Science and Technology at University of Pannonia

in the discipline of Computer Sciences

written by: Zeyu Wang

Supervisor: Dr. Juhász Zoltán

I recommend the dissertation for acceptance: yes / no.

……………………….
supervisor

I recommend the dissertation for peer review.

……………………….
chair of the DDHC

The PhD-candidate has achieved …….. % at the public debate.

The composition of the Final Examination Committee:

chair:……..……………………………………………………………………….

reviewers:..……………………………………………………………………

members:……………………………………………………………………….

Veszprém,

……………………….
chair of the committee

Qualification of degree: …...............................

Veszprém,

……………………….
chair of the UDHC

Acknowledgment

First and foremost, I would like to express my most heartfelt gratitude to my supervisor, Dr. Juhász Zoltán. Every bit of my achievements is inseparable from his boundless support, guidance, and encouragement. In moments of despair, he is my source of optimism; in times of confusion, he is the lighthouse guiding me through the fog; and in my exhaustion, he is the serene harbor where I find solace. His solid theoretical foundation, extensive engineering experience, and rigorous scholarly attitude have been an invaluable source of inspiration for me. His philosophical reflections not only enlighten my research but also bring epiphanies into my life. I still remember the moment four years ago when he accepted me as his disciple and taught me to contribute new knowledge to mankind. As an idealist, I was deeply inspired by his words—how glorious and magnificent such a mission is! I am willing to be his eternal devotee.

My deep thanks to Dr. Hangos Katalin, Dr. Hartung Ferenc, Dr. Görbe Péter, and Dr. Dulai Tibor from the doctoral school for their support and assistance during every semester report. My gratitude also extends to my lab mates, Iffah Syafiqah, Dr. Mohamed Issa, and Szabó Patrícia, for bringing joy and care to my daily life. I am deeply thankful to my lecturers—Dr. Kozmann György, Dr. Bari Ferenc, Dr. Nagy Zoltán, and Dr. Kósa István—for imparting their knowledge to me. Lastly, I sincerely appreciate the staff of the Faculty of Information Technology, especially Ujvári Orsolya, Lényi Szilvia, Dr. Lányi Cecília, Dr. Fodor Attila, Dr. Süle Zoltán, Dr. Vassányi István, and Dr. Szücs Veronika, for providing me with invaluable academic and administrative support.

I would like to thank the Chinese and Hungarian governments for their collaboration in the field of education, especially the China Scholarship Council and the Hungarian Tempus Public Foundation, for providing financial support for my studies in Hungary over the past four years.

I want to express my deepest thanks to my mother, Yuanling Zhou, and my father, Xiqiang Wang, for their unconditional support, no matter what I choose to pursue. Most importantly, I am profoundly grateful to my lifelong partner, my beloved wife Nuoya Zhang. Finding a soulmate in life is rare, but I am fortunate to have met, known, and married her. Her love and guidance will forever be my driving force and beacon.

Last but not least, I would like to express my gratitude to my great motherland for allowing me to live in a peaceful country in a turbulent era. It is because of this peace that I have been able to sit at a calm desk to study, delve into academic pursuits, and explore the unknown, free from the sufferings of famine and war. I will remain true to my original aspiration, keep my mission firmly in mind, and work tirelessly to realize the Chinese Dream of national rejuvenation.

Zeyu Wang, 2025

Abstract

Electroencephalography (EEG) is an essential non-invasive technique for monitoring and analyzing brain activity, widely used in both clinical and research domains to investigate physiological and pathological states. It offers high temporal resolution, affordability, and portability, but its inherent nonlinear, non-stationary nature, coupled with susceptibility to noise and artifacts, makes signal processing highly complex, also for the high-density, multi-channel EEG dataset, the processing is often computationally expensive. The advent of advanced signal decomposition techniques and high-performance computing solutions, such as Graphics Processing Units (GPUs), has opened new avenues for efficient and accurate EEG signal processing. This thesis explores advanced signal decomposition methods combined with high-performance GPU-based parallel computing to enhance the efficiency and effectiveness of EEG analysis.

First, I present a GPU implementation of the Improved Complete Ensemble Empirical Mode Decomposition with Adaptive Noise (CEEMDAN) algorithm. The CEEMDAN algorithm retains the original EMD's strength in decomposing EEG signals without relying on a predefined function, while effectively mitigating mode mixing issues caused by intermittent noise through the use of adaptive noise. However, the inclusion of adaptive noise increases the computational complexity significantly. The GPU-optimized approach dramatically accelerates the time-frequency analysis of EEG signals using CEEMDAN by parallelizing the processing of noise-signal combinations and minimizing function startup overhead, thereby enabling more efficient extraction of oscillation patterns. The proposed method demonstrates significant performance improvements over CPU-based implementations, paving the way for efficient processing of large-scale, noise-containing EEG signals in neuroscience and clinical diagnostics.

Second, I focus on the Multivariate Empirical Mode Decomposition (MEMD) method, tailored for multi-channel EEG data. Multi-channel signal decomposition requires projection in a high-dimensional space, where the computational load increases with the number of direction vectors used for projection. Leveraging the parallel processing capabilities of GPUs, the MEMD algorithm executes the projection of multi-channel signals onto different direction vectors and their subsequent decomposition in parallel, significantly enhancing efficiency. The optimized GPU implementation not only achieves a substantial reduction in processing time but also preserves the inter-channel correlations critical for functional connectivity studies. This advancement facilitates efficient network analysis of large-scale multi-channel bioelectric signals.

Finally, I introduce an innovative application of GPU tensor cores to the Independent Component Analysis (ICA) algorithm, a foundational technique for artifact removal in EEG processing. The performance of ICA is often limited by the extensive matrix multiply-add operations it requires. Tensor cores are specifically designed to handle these operations with high efficiency, delivering enhanced computational performance and memory throughput of the algorithms. By leveraging tensor cores for these operations in ICA, the proposed GPU implementation substantially boosts its efficiency, making it feasible to process large-scale,

high-density EEG datasets. This contribution effectively addresses a critical bottleneck in EEG preprocessing pipelines, particularly for applications that demand rapid artifact removal and data cleaning.

By combining GPU parallelization with advanced signal decomposition methods, this thesis achieves a significant advancement in EEG processing efficiency, facilitating applications such as neurofeedback, seizure detection, cognitive workload monitoring, and brain-computer interfaces. The integration of scalable, high-performance computing empowers researchers to analyze large EEG datasets from multi-center studies or public repositories with unparalleled speed and accuracy.

Összefoglaló

Az elektroencephalográfia (EEG) az agyi aktivitás monitorozására és elemzésére szolgáló non-invazív méréstechnológia, amelyet széles körben alkalmaznak klinikai vizsgálatokban illetve az agykutatásban. Az EEG nagy időbeli felbontással rendelkezik, megfizethető és hordozható, azonban a mért jelek feldolgozása – a jel nemlineáris, nem stacionárius jellege, valamint a zajra és műtermékekre való érzékenysége miatt – komplex folyamat, ami sokcsatornás EEG-mérések esetén gyakran számításigényes is. A fejlett jel-dekompozíciós módszerek és a nagyteljesítményű számítástechnikai megoldások – például a grafikus feldolgozó egységek (GPU-k) – megjelenése új utakat nyitott a hatékony és pontos EEG-jelfeldolgozásban. Ez a dolgozat azt vizsgálja, miként lehet fejlett dekompozíciós módszereket kombinálni a nagyteljesítményű GPU-alapú párhuzamos számítástechnikával az EEG-elemzések hatékonyságának és eredményességének növelése érdekében.

A dolgozatban először bemutatom az Improved Complete Ensemble Empirical Mode Decomposition with Adaptive Noise (CEEMDAN) algoritmus GPU megvalósítását. A CEEMDAN algoritmus megtartja az eredeti Empirical Mode Decomposition (EMD) algoritmus lényegét az EEG-jelek felbontásában, azaz, hogy nem előre definiált bázisfüggvényekre épít, miközben hozzáadott adaptív zaj használatával hatékonyan enyhíti az időszakos zaj miatt az EMD módszerben jelentkező módkeveredési problémákat. Az adaptív zaj beépítése azonban jelentősen megnöveli a számítási komplexitást. A GPU-ra optimalizált implementáció a zaj-jel kombinációk feldolgozásának párhuzamosításával és a függvényhívási költségek minimalizálásával jelentősen felgyorsítja az EEG-jelek idő-frekvencia elemzését, ezáltal lehetővé téve az EEG jelet felépítő oszcillációs komponensek hatékonyabb kinyerését. A javasolt módszer jelentős teljesítményjavulást mutat a CPU-alapú megvalósításokhoz képest, megnyitva az utat a nagyméretű EEG mérési adathalmazok hatékony feldolgozásához.

Ezt követően a Multivariate Empirical Mode Decomposition (MEMD) módszerre összpontosítok, amit kimondottan többcsatornás adatok dekompozíciójára terveztek. A módszer lényege, hogy az egyes elektródák jeleit összehangolva bontja komponensekre, ezáltal biztosítva, hogy az a felbontás minden elektródán azonos számú és frekvenciájú komponens sorozatot eredményez. A többcsatornás jelbontáshoz az adatokat sokdimenziós térbe kell vetíteni, ami miatt a számítási költség a projekcióhoz használt irányvektorok számával arányosan nő. A MEMD algoritmus a GPU-k párhuzamos feldolgozási képességeit kihasználva párhuzamosan hajtja végre a többcsatornás jelek különböző irányvektorokra történő vetítését majd azok dekompozícióját, jelentősen növelve a számítási hatékonyságot. Az optimalizált GPU implementáció nem csak a feldolgozási idő jelentős csökkenését eredményezi, hanem megőrzi a funkcionális konnektivitási vizsgálatok számára kritikus csatornák közötti korrelációkat is. A létrejött párhuzamos program így megkönnyíti a nagyméretű többcsatornás bioelektromos jelek hatékony hálózati elemzését is.

Végezetül bemutatom a GPU tenzor magok innovatív alkalmazását az Independent Component Analysis (ICA) algoritmusban, ami az EEG-feldolgozás során a műtermékek eltávolításának alapvető technikája. Az ICA algoritmus teljesítményét gyakran korlátozza a

végrehajtása során szükséges nagyszámú mátrix szorzás-összeadás műveletek. A tenzor magokat kifejezetten arra tervezték, hogy nagy hatékonysággal hajtsák végre ezeket a műveleteket, így javítva az algoritmusok számítási teljesítményt és memória áteresztőképességét (throughput). A tenzor magok felhasználásával a javasolt GPU megvalósítás jelentősen megnöveli ezekhez az ICA műveletek hatékonyságát, lehetővé téve nagyméretű, sokcsatornás EEG adathalmazok gyors feldolgozását. Ez a hozzájárulás hatékonyan javítja az EEG előfeldolgozási feladatsor (pipeline) szűk keresztmetszetét jelentő ICA végrehajtását, különösen az olyan alkalmazások esetében, amelyek gyors műtermék-eltávolítást és adattisztítást igényelnek.

A GPU-alapú párhuzamosság és a fejlett jel-dekompozíciós módszerek kombinálásával a dolgozat jelentős előrelépést ért el az EEG feldolgozás hatékonyságának növelésében, megkönnyítve az olyan alkalmazások, mint a neurofeedback, az epilepszia rohamok észlelése, a kognitív terhelés monitorozása, vagy az agy-számítógép interfészek, hatékonyabb megvalósítását. A skálázható, nagy teljesítményű számítástechnika integrálása ezen felül lehetővé teszi a kutatók számára, hogy páratlan sebességgel és pontossággal elemezzék a különböző kutatóközpontokból származó, publikus Interneten elérhető adattárakból származó nagyméretű EEG adatkészleteket.

List of abbreviations

AI	Arithmetic Intensity
APU	Accelerated Processing Unit
BCI	Brain-Computer Interface
BSS	Blind Source Separation
CEEMDAN	Complete Ensemble Empirical Mode Decomposition with Adaptive Noise
CNN	Convolutional Neural Network
CSP	Common Spatial Patterns
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CWT	Continuous Wavelet Transform
DFT	Discrete Fourier transform
DTFT	Discrete Time Fourier Transform
DWT	Discrete Wavelet Transform
ECoG	Electrocorticography
EEG	Electroencephalography
EEMD	Ensemble Empirical Mode Decomposition
EMD	Empirical Mode Decomposition
ENIAC	Electronic Numerical Integrator and Computer
ERP	Event-Related Potential
ESD	Energy Spectrum Density
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
fMRI	Functional Magnetic Resonance Imaging
FMA	Fused Multiply-Add
GEMM	General Matrix Multiplication
GPC	Graphics Processing Cluster
GPU	Graphics Processing Unit
HHT	Hilbert-Huang Transform
HOS	Higher Order Statistics
HPC	High-Performance Computing
ICA	Independent Component Analysis
IMF	Intrinsic Mode Function
JADE	Joint Approximation Diagonalization of Eigenmatrices
LSTM	Long Short-Term Memory Network
MEMD	Multivariate Empirical Mode Decomposition
MI	Modulation Index
MIC	Many Integrated Core
MPP	Massively Parallel Processing

PET	Positron Emission Tomography
PCA	Principal Component Analysis
RF	Random Forest
RNN	Recurrent Neural Network
SD	Standard Deviation
SEEG	Stereo-electroencephalography
SFU	Special Function Unit
SM	Stream Multiprocessor
SOBI	Second Order Blind Identification
SP	Stream Processor
SPMD	Single Program Multiple Data
STFT	Short-Time Fourier Transform
SVM	Support Vector Machine
TPC	Texture Processing Cluster
WMMA	Warp Matrix Multiply and Accumulate
WT	Wavelet Transform

Table of Contents

Acknowledgment I
Abstract II
Összefoglaló IV
List of abbreviations VI
Table of Contents VIII
1 INTRODUCTION 1
1.1 EEG Signal Processing 2
1.2 GPU Parallelism 5
1.2.1 Development of HPC and GPU 5
1.2.2 GPU hardware hierarchy and CUDA programming 8
1.2.3 Necessity of GPU computing in EEG processing 12
1.3 Thesis Organization 13
2 SIGNAL DECOMPOSITION METHODS 15
2.1 Introduction to Signal Decomposition 15
2.2 Frequency Decomposition Methods 17
2.3 Spatial Decomposition Methods 25
3 GPU IMPLEMENTATION OF ICEEMDAN 32
3.1 Materials and Methods 32
3.1.1 Related works 32
3.1.2 Design of the parallel implementation 34
3.2 Details of Implementation 36
3.2.1 Data structure and initialization 36
3.2.2 The parallel sifting process 37
3.3 Results 38
3.3.1 Test hardware 38
3.3.2 Numerical validation 39
3.3.3 Computational performance and optimization 40
3.4 Summary 45
4 GPU IMPLEMENTATION OF MEMD 47
4.1 Materials and Methods 47
4.1.1 Related works 47
4.1.2 Design of the parallel implementation 50
4.2 Details of Implementation 51
4.2.1 Pre-processing 51
4.2.2 Signal projection to direction vectors 51
4.2.3 Extrema detection 52
4.2.4 Cubic spline interpolation 53
4.3 Results 56
4.3.1 Test hardware 56
4.3.2 Numerical validation 56
4.3.3 Performance results 60
4.3.4 Performance analysis 63

4.4 Summary 66
5 GPU IMPLEMENTATION OF ICA 67
5.1 Materials and Methods 67
5.1.1 Related works 67
5.1.2 Design of the Parallel implementation 69
5.2 Details of Implementation 70
5.2.1 Tensor core applications 70
5.2.2 Potential further parallelism from multiple data blocks 77
5.3 Results 78
5.3.1 Test environment 79
5.3.2 Numerical validation 79
5.3.3 Performance results and analysis 81
5.4 Summary 82
6 CONCLUSIONS 84
7 SUMMARY OF THE MAIN CONTRIBUTIONS 86
Thesis I: GPU implementation of ICEEMDAN 86
Thesis II: GPU implementation of MEMD 86
Thesis III: GPU implementation of ICA 87
LIST OF PUBLICATIONS 88
References 89
APPENDIX 1

1 INTRODUCTION

Electroencephalography (EEG) is a non-invasive technique for recording the electrical activity of the brain, widely used in clinical and research to study neural processes. The physiological basis of EEG lies in the synchronized activity of large populations of neurons, particularly pyramidal cells in the brain cortex. When these neurons charge/discharge, they generate post-synaptic potentials that sum to produce voltage fluctuations detectable on the scalp. These electrical signals, which contain rich physiological and pathological information, are recorded through electrodes placed on the scalp, capturing neural activity in real time.

In 1875, Richard Caton recorded the electrical activity of the cerebral cortex of animals through experiments [1]. In 1924, Hans Berger invented the term electroencephalogram and recorded electrical activity on the human scalp for the first time [2], laying the foundation for modern EEG technology. Over time, EEG has evolved significantly, with advances in electrode technology, signal processing, and computational techniques enabling more precise and detailed analyses of brain activity.

The brain is a complex and rapidly changing system, and its physiological state transitions may occur within milliseconds. Brain imaging technologies like functional Magnetic Resonance Imaging (fMRI) [3], or Positron Emission Tomography (PET) [4] can provide high spatial resolution but their temporal resolution is poor, which means difficulty in capturing the dynamic characteristics of the brain. In contrast, the high temporal resolution of EEG, on the order of milliseconds, makes it ideal for studying fast neural dynamics, such as Event-Related Potentials (ERPs) [5]. Additionally, EEG is cost-effective, portable, and non-invasive, making it accessible for a wide range of applications, from clinical diagnostics to cognitive neuroscience. However, EEG also has limitations, such as low spatial resolution and susceptibility to noise and artifacts, which increase the complexity of data processing.

EEG signals are typically weak, with amplitudes ranging from a few microvolts to tens of microvolts, and contain several key frequency bands: delta (0.5–4 Hz), theta (4–8 Hz), alpha (8–13 Hz), beta (13–30 Hz), and gamma (30–100 Hz), each associated with different neural states and cognitive processes [6], [7], [8], [9]. The brain's electrical activity arises from intricate neural interactions, including synaptic transmissions, neuronal charging/discharging, and network oscillations. These processes are governed by nonlinear dynamics, meaning that small changes in inputs can lead to disproportionately large or unpredictable changes in the outputs, making EEG signals inherently nonlinear [10], [11], [12]. EEG signals are also non-stationary [13], [14], meaning their statistical properties, such as mean, variance, and distribution, vary over time. This non-stationarity arises from the brain's constantly changing states, influenced by cognitive processes, sensory inputs, physiological conditions, and external stimulations. The nonlinear and non-stationary nature of EEG signals reflect the brain's dynamic adaptability and complexity, requiring advanced processing methods to unlock their full potential in research and clinical applications.

1.1 EEG Signal Processing

EEG signal processing involves extracting meaningful information from noisy, non-stationary, high-dimensional signals. From the 1940s, research on EEG signal processing gradually developed and began to be applied in the clinical field [15], and over the decades, EEG processing methods have evolved to address the challenges posed by the complexity and variability of EEG data. Initially, EEG signals were analyzed using a manual time-domain method, i.e., clinicians analyzed EEG waveforms with their naked eyes based on their clinical experience, which is intuitive but inefficient, and the subjectivity of the doctors may cause misjudgment. As one of the earliest attempts to apply computational methods to EEG processing, Dietsch first used the Fourier transform to analyze EEG signals in 1932 [16], which enabled the identification of dominant rhythms (e.g., alpha waves) and their association with various states of consciousness. Since the 1940s, with the development of computer and digital signal processing technologies, a series of classic time domain, frequency domain, time-frequency domain, and time-space domain analysis methods have emerged.

Time domain analysis is typically intuitive and has a clear physical meaning, and its main focus is extracting the waveform features of EEG signals in the time domain. Commonly used methods include correlation analysis [17], histogram analysis [18], waveform parameter analysis [19], variance analysis [20], waveform recognition [21], coherency analysis [22], etc. Additionally, statistical-based parametric models [23] can also be used to extract EEG time domain features which can then be used for classification and recognition. However, given the inherent complexity of EEG signal waveforms, waveform-based EEG analysis methods are not universally effective.

The purpose of frequency domain analysis is to separate the oscillations of different frequencies in EEG and extract their distribution feature. Given that EEG signals are stochastic, their energy is infinite over infinite time, and their average power is not zero. Therefore, stochastic signals must be power signals. On the contrary, if the energy is finite and the average power is zero, the signal is an energy signal. According to the Dirichlet condition, if a signal $f(t)$ has a Fourier transform, it should satisfy the following conditions:

(i) $f(t)$ must be absolutely integrable over a period.
(ii) $f(t)$ has a finite number of maxima and minima.
(iii) $f(t)$ has only a finite number of discontinuities in any finite.

Evidently, as a stochastic signal, the period of the EEG signal is infinite, so it cannot be directly Fourier Transformed. Therefore, one way to get the distribution of different frequencies of the EEG signal is to start from its power distribution, i.e. Power Spectral Density (PSD). Fortunately, the Wiener-Khinchin theorem proves that the PSD of a stochastic signal is equal to the Fourier transform of the autocorrelation function of this signal. Therefore, the process to obtain the PSD of the EEG signal is: first find the autocorrelation function of the given EEG signal, and then perform Fourier transform on the autocorrelation function to obtain the PSD. Another method is to take the length of the given EEG signal as its period, that is, to cutoff the stochastic signal, so that it becomes an energy signal. Fourier transform can be can be directly performed for energy signals and its square can be taken to

obtain the Energy Spectrum Density (ESD) which also reflects the distribution of different frequencies. These two methods are usually called the indirect method and the direct method. Spectrum features are important features of EEG signals, and many studies have been carried out based on the frequency domain analysis [24], [25], [26], [27]. Unfortunately, spectrum analysis only reflects the frequency distribution of the signal, but loses the temporal information of the signal.

Time-frequency domain analysis can profile the frequency distribution of a signal while retaining its time information, i.e., the time-varying frequency distribution, which is particularly valuable for studying the non-stationary nature of EEG signals, where oscillatory patterns and transient events vary over time. Commonly used time-frequency analysis methods in EEG signal processing include: Short-Time Fourier Transform (STFT) [28], [29], Wavelet Transform (WT) [28], [30], [31], Hilbert-Huang Transform (HHT) [32], [33], [34], etc.

STFT divides the signal into overlapping segments (windows) and computes the Fourier Transform for each segment. This approach offers a time-frequency representation of the signal and is computationally efficient, essentially a repetition of the FFT. However, due to the fixed window size, time and frequency resolution are conjugate to each other and the trade-off between them becomes difficult (Heisenberg uncertainty principle). WT uses wavelets—localized oscillatory functions of varying scales—to analyze the signal. Unlike STFT, WT adapts to both high and low-frequency components, offering multi-resolution analysis, which makes it well-suited for non-stationary signal. Nevertheless, WT still uses a matching mechanism between pre-set functions (wavelets) and signals to extract time-frequency features, and the choice of wavelet basis will affect the results. HHT decomposes the signal into Intrinsic Mode Functions (IMFs) using Empirical Mode Decomposition (EMD) and applies the Hilbert Transform to derive instantaneous frequency and amplitude. The time-frequency analysis method based on EMD and the Hilbert Transform not only removes the constraints of the uncertainty principle, but can also use the adaptability and data-driven nature of EMD to decompose the signal to narrowband components to describe the time-frequency characteristics of the non-stationary signals more accurately. However, compared with STFT and WT, HHT is computationally intensive and sensitive to intermittent noise.

Since EEG signals are multidimensional signals recorded by electrodes at different locations, in addition to time and frequency information, EEG also contains spatial information. For the third-dimensional feature of EEG signals, spatial domain analysis plays an important role in brain network construction and analysis [35], [36], EEG source localization [37], [38], [39], brain state recognition [40], [41], etc. Spatial decomposition methods based on the statistical features of multidimensional signals, such as Independent Component Analysis (ICA) [42], [43], [44], Principal Component Analysis (PCA) [45], [46], and Common Spatial Patterns (CSP) [47], [48], have been widely used in EEG signal processing.

The 1990s and early 2000s marked the integration of machine learning techniques into EEG analysis. Algorithms like Support Vector Machines (SVMs) Random Forest (RF), and k-Nearest Neighbors (k-NNs) were applied to classify EEG patterns [49], [50], [51], particularly in applications like seizure detection [52] and brain-computer interfaces (BCIs) [53]. Since

the 21st century, with the development of high-performance computing hardware and software, deep learning has emerged as a transformative force in EEG processing. Neural network architectures, such as Convolutional Neural Networks (CNNs) and Long Short-Term Memory (LSTM) networks, have been used for tasks ranging from emotion recognition to motor imagery classification [54], [55], [56].

Nowadays, EEG signal processing involves a multi-step pipeline, encompassing preprocessing, feature extraction, analysis, and interpretation. Pipelines combining time, frequency, statistics, and spatial information have become an effective way of EEG research.

Preprocessing is a critical step in enhancing the quality of the acquired raw EEG data by reducing noise and artifacts. This process employs several techniques to ensure clean and interpretable signals. Filtering, for instance, utilizes band-pass filters to isolate specific frequencies while eliminating unwanted components like low-frequency drifts and high-frequency noise. Artifact removal methods, such as ICA or regression-based methods, help eliminate disturbances caused by eye blinks, muscle activity, and environmental interference [57], [58], [59]. Additionally, re-referencing—choosing an appropriate reference electrode or applying an average reference—improves the clarity and interpretability of the data [60], [61]. Segmenting further organizes the EEG data by dividing it into epochs, often aligned with specific events or stimulation, to facilitate event-related analyses.

Once the EEG data is preprocessed, feature extraction focuses on identifying meaningful patterns or features in the EEG signals. These features are important for understanding neural activity and can be categorized into different types. Time-domain features, such as amplitude, peak latency, and event-related potentials (ERPs), offer insights into temporal dynamics [62]. Frequency-domain features analyze spectral power and activity within specific frequency bands (e.g., alpha, beta) to reveal oscillatory patterns [63]. Spatial features, enhanced through techniques like ICA, PCA, and CSP, improve the signal-to-noise ratio for targeted tasks or brain regions [64], [65]. Lastly, time-frequency features, derived from methods like STFT, WT, and HHT, provide a comprehensive view that combines temporal and spectral information [66].

The extracted features are subsequently analyzed to address specific research or clinical questions. Connectivity analysis methods, such as coherence, phase synchronization, and graph metrics, are used to examine functional interactions between brain regions [67], [68], [69]. Statistical analysis also plays an important role [70], [71], enabling hypothesis-driven studies to uncover relationships between neural activity and cognitive or behavioral variables. Additionally, classification algorithms are often employed in this stage, including SVMs, RF, k-NNs, or deep learning algorithms, to classify EEG patterns for applications like seizure detection or mental state identification [72], [73].

Visualization methods aid in result interpretation and play an important role in presenting findings effectively, ensuring clear illustrations of complex data. Topographic maps provide a spatial representation of neural activity across the scalp, while time-frequency plots illustrate the oscillation changes over time. Connectivity graphs, on the other hand, highlight interactions between different brain regions, offering a comprehensive view of the brain's functional dynamics [74]. Together, these visualization methods facilitate a deeper

understanding of the results and a better interpretation of their implications.

1.2 GPU Parallelism

1.2.1 Development of HPC and GPU

High-performance computing (HPC) is a technology that uses powerful processor clusters to process massive amounts of data and solve complex problems at extremely high speeds. Its related software and hardware research has become a hotspot in the field of computer science, offering potential solutions to the complex computing challenges encountered in the fields of natural and engineering sciences [75], [76], [77], [78], [79]. Since the birth of the first general-purpose Electronic Numerical Integrator And Computer (ENIAC) in 1945, high-performance computing and supercomputers have evolved significantly over the past half century. The CDC 6600, released in 1964 by Control Data Corporation, is considered the first true supercomputer, reaching 3 mega floating-point operations per second (MFlop/s). In the 1970s and 1980s, Cray Research dominated with machines like the Cray-1 (80 MFlop/s) and Cray-2 (1.9 GFlop/s). The 1990s saw the rise of massively parallel processing (MPP), exemplified by the ASCI Red, the first supercomputer to surpass 1 TFlop/s in 1997. In the 21st century, IBM's Roadrunner (2008) was the first to exceed 1 PFlop/s, and Fugaku, developed by RIKEN and Fujitsu in 2020, reached over 442 PFlop/s. As of November 2024, the peak performance of the El Capitan supercomputer (operated by Lawrence Livermore National Laboratory, U.S. Department of Energy), which ranks first in the TOP500 supercomputers list and is equipped with 11039616 CPU and GPU cores, has reached 2.7 exaFlop/s.

Due to their energy efficiency advantage, heterogeneous architectures consisting of general-purpose processors and accelerators have become the development trend of high-performance computing system architecture. As shown in Figure 1-1(a), among the TOP500 supercomputers, 209 are heterogeneous computing systems, and their accelerators are mainly provided by NVIDIA, AMD, and Intel. The computing power ratios provided by different heterogeneous systems are shown in Figure 1-1(b), which shows that the computing power provided by heterogeneous systems is almost four times that provided by homogeneous systems.

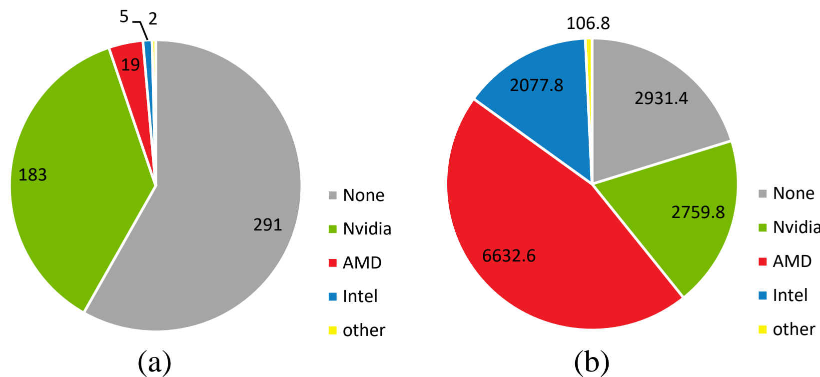

Figure 1-1: (a): The number of systems equipped with different accelerators in the TOP500 supercomputer ranking; (b): The ratio of computing power (PFlop/s) provided by systems equipped with different accelerators.

The mainstream accelerators include FPGA (Field-Programmable Gate Array) [80], GPU (Graphics Processing Unit), APU (Accelerated Processing Unit) [81], MIC (Many Integrated Core) [82], etc. These advanced accelerators not only provide development opportunities for large-scale scientific computing and engineering simulation, but also bring challenges to algorithm development and implementation. Among them, GPU, as the most commonly used accelerator in major supercomputers, has developed from a single-purpose graphics processing unit to a high-speed parallel computing unit. In addition, due to the excellent energy efficiency ratio and advantageous usability, the heterogeneous computing system of "CPU+GPU" has been widely used in engineering fields such as physical simulation [83], molecular dynamics [84], signal processing [85], etc.

Nvidia first proposed the concept of Graphics Processing Unit (GPU) in 1999 with the release of its GeForce256 based on Celsius architecture. Compared with the previous Fahrenheit architecture, Celsius architecture increased the number of image processing pipelines, specifically for accelerating graphics rendering. In the subsequent generations, Nvidia used fixed-point pixel shaders and vertex shaders to replace the image processing pipeline design, greatly improving the programmability of GPU and expanding its graphics rendering capabilities. In 2006, NVIDIA released the first architecture with a unified shader model, the Tesla architecture. The original shader units (pixel shaders, vertex shaders) for processing vectors were replaced with a homogeneous collection of universal scalar-based floating-point processors (Stream Processors, SPs) that can perform a more universal set of tasks. The following year, the Compute Unified Device Architecture (CUDA), a general-purpose computing programming model for GPUs, was released, which opened up a wide range of applications for GPUs in parallel computing.

Since the launch of CUDA, Nvidia has made a major shift in the design of GPUs regarding functionality and capability. It has begun to focus on improving the efficiency of GPUs for general-purpose parallel computing. Nvidia implemented dynamic parallelism and Hyper-Q features [86] in the Kepler architecture, further improving the GPU's computing efficiency and parallelization. In the Maxwell architecture, the GPU's computational efficiency in low-power mode was increased, solving the energy efficiency ratio problem of large multi-GPU systems. The NVLink and Unified Memory were introduced in the Pascal architecture, which improved the efficiency of CPU-GPU and GPU-GPU data transfer, providing solutions for data-intensive computing tasks while also improving the scalability of GPUs. With the development of artificial intelligence and neural networks, Nvidia introduced the tensor core in the Volta architecture, which is dedicated to matrix multiply-add operations that are most involved in machine learning algorithms. The Ray-tracing core, first introduced in the Turing architecture, greatly improves the GPU's game performance. The Hopper architecture, launched in 2022, can be fused with Nvidia's Grace CPU into a single die, called the Grace-Hopper Superchip, which further reduces the overhead of GPU and CPU data exchange and improves the scalability of heterogeneous computing systems.

Since the release of its first product, STG-2000, in 1995, Nvidia has released more than one thousand GPU products. The architecture and main features of each generation of products are shown in Table 1-1.

Architecture	Time	Products	Important features
Fahrenheit	1995	Riva TNT series	Integrates 3D computing and 2D display
Celsius	1999	GeForce2 series GeForce4 MX series GeForce4 Go series Quadro4 NVS series	DirectX 7.0 OpenGL 1.2 128MB VRAM **First GPU**
Kelvin	2001	GeForce3 series GeForce4 series Quadro4 series	DirectX 8.0 OpenGL 1.5 Shader Model 1.3 Vertex Shader 1.1 128 MB VRAM
Rankine	2003	GeForce FX (5xxx) series GeForce FX Go 5 series GeForce PCX series Quadro NVS series Quadro FX (x000) series	DirectX 9.0 OpenGL 2.1 Shader Model 2.0a Vertex Shader 2.0a 256MB VRAM
Curie	2004	GeForce 6 (6xxx) series GeForce 7 (6xxx) series Quadro FX (x500) series Quadro FX (x600) series	DirectX 9.0c OpenGL 2.1 Shader Model 3.0 256MB VRAM
Tesla	2006	GeForce 8 (8xxx) series GeForce 9 (8xxx) series GeForce 100 series GeForce 200 series GeForce 300 series GeForce 405 Quadro FX (x700) Quadro FX (x800) C870, D870, S870, C1060, S1070, S1075,	DirectX 10.0 OpenGL 3.3 Shader Model 4.0 Stream processors (SPs, CUDA cores) Streaming Multiprocessors (SMs) **CUDA support** Compute capability 1.0-2.0
Fermi	2010	GeForce 400 series GeForce 500 series GeForce 600 series Quadro (x000M) series C2050, M2050, C2070, C2075, M2070, M2090, S2050, S2070	40 nm process 16 SMs 32 SPs / SM Compute capability 2.0-2.1
Kepler	2012	GeForce 600 series GeForce 700 series Quadro (Kxxx) series K10, K20, K40, K80	28 nm process 15 SMs 192 FP32 SP / SM 64 FP64 SP / SM Compute capability 3.0-3.7
Maxwell	2014	GeForce 900 series Quadro (Mxxx) series M4, M6, M10, M40, M60	28 nm process 16 SMs 128 FP32 SP / SM 64 FP64 SP / SM Compute capability 5.0-5.2

Architecture	Year	Product families	Key features
Pascal	2016	GeForce 10 series	16 nm process 60 SMs 64 FP32 SP / SM 32 FP64 SP / SM **NVLink support** Compute capability 6.0-6.2
		Quadro (Pxxx) series	
		P4, P6, P40, P100	
Volta	2017	Nvidia TITAN V	12 nm process 80 SMs 64 FP32 SP / SM 32 FP64 SP / SM **8 Tensor cores / SM** Compute capability 7.0-7.5
		Quadro GV100	
		V100	
Turing	2018	GeForce 16 series RTX 20 series GeForce (MX 4xx) series	12 nm process 72 SMs 64 FP32 SP / SM 2 FP32 / SM 8 Tensor cores / SM **Ray-tracing cores support** Compute capability 7.5-8.6
		Quadro (Tx000) series	
		T4, T10, T40	
Ampere	2020	RTX 30 series GeForce RTX 4010	7 nm process 108 SMs 64 FP32 SP / SM 32 FP64 SP / SM 4 Tensor cores / SM Compute capability 8.0-8.6
		Quadro (Ax000) series	
		A2, A10, A30, A40, A100	
Hopper	2022	H100, H800	4 nm process 132 SMs 128 FP32 SP / SM 64 FP64 SP / SM 4 Tensor cores / SM **Superchip support** Compute capability 9.0
Ada Lovelace	2022	RTX 40 series	4N process 144 SMs 128 FP32 SP / SM 4 FP64 SP / SM 4 Tensor cores / SM Compute capability 8.9
		Quadro (x000A) series	
		L4, L20, L40	
Blackwell	2024	RTX 50 series	4NP process 192 SMs 128 FP32 SP / SM 4 Tensor cores / SM Compute capability 10.0
		B200	

Table 1-1: The GPU architectures released by Nvidia as of January 2025, their corresponding product families, and key features.

1.2.2 GPU hardware hierarchy and CUDA programming

The heart of GPU chip architecture is the Streaming Multiprocessor (SM). As an example, Figure 1-2 shows the hardware composition of the AD102 chip (used in the RTX4090, RTX6000, and L40) based on the Ada Lovelace architecture. The full AD102 GPU includes 12 Graphics Processing Clusters (GPCs), 72 Texture Processing Clusters (TPCs), 144 Streaming Multiprocessors (SMs), and a 384-bit memory interface with 12 32-bit memory

controllers. Like prior GPUs, each SM in AD102 is divided into four processing blocks (or partitions), with each partition containing a 64 KB register file, an L0 instruction cache, one warp scheduler, one dispatch unit, 16 CUDA Cores that are dedicated for processing FP32 operations (up to 16 FP32 operations per clock), 16 CUDA Cores that can process FP32 or INT32 operations (16 FP32 operations per clock or 16 INT32 operations per clock), one Ada Fourth-Generation Tensor Core, four Load/Store units, and a Special Function Unit (SFU) which executes transcendental and graphics interpolation instructions.

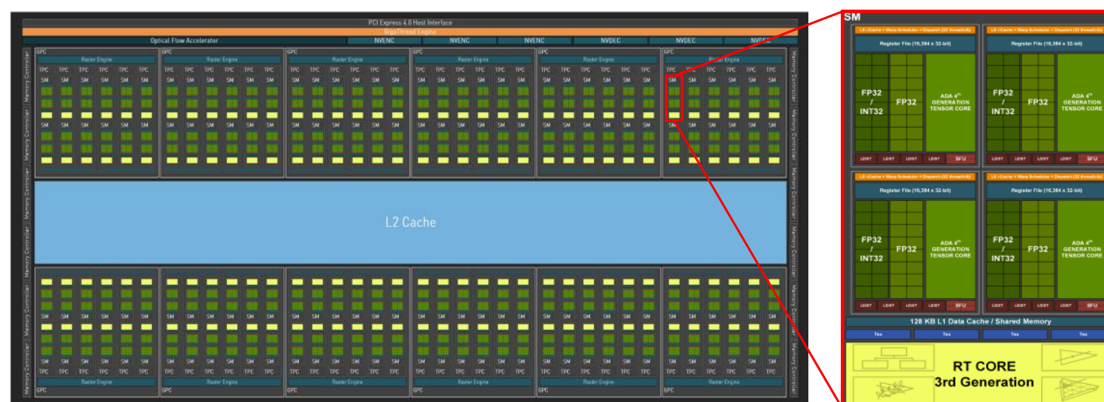

Figure 1-2: Block diagram of the AD102 chip and its interior SM [87].

Components	Units	Physical location	
		IN/OUT SM	ON/OFF CHIP
Execution	SP/CUDA core	IN	ON
	Double-Precision (DP) units	IN	ON
	SFU	IN	ON
	LD/ST	IN	ON
	Tensor cores	IN	ON
	RT cores	IN	ON
Memory	Registers	IN	ON
	L0 / Instruction cache	IN	ON
	L1 cache / Shared memory)	IN	ON
	L2 cache	OUT	ON
	Texture cache	IN	ON
	Global / Local / Constant memory	OUT	OFF
Control	Giga thread engine	OUT	ON
	Warp scheduler	IN	ON
	Instructions dispatcher	IN	ON

Table 1-2: The hardware units that make up the GPU architecture, and their physical locations in the GPU chip.

For the memory system, Each SM in the AD102 chip contains a 256 KB (64KB for each processing block) register file, and 128 KB of L1/Shared Memory, which can be configured for different memory sizes depending on the needs of the graphics or compute workload. Outside of SM but still on-chip, the AD102 chip has been outfitted with 98304 KB of L2 cache, and all data loaded from the off-chip memory are first cached by the L2 cache. Compared to the on-chip memory, off-chip memory has higher access latency and lower bandwidth, but larger size, GPU products based on the Ada architecture can be equipped with

up to 80 GB of off-chip memory. In general, the GPU hardware architecture is composed of three components: execution, memory, and control. Their detailed units and physical locations are outlined in Table 1-2.

Unlike CPUs optimized for low latency, GPUs are designed for high throughput, which means their programming and execution model is completely different from that of CPUs. As shown in Figure 1-3, in a heterogeneous computing system composed of CPUs and GPUs, CPUs and GPUs execute different parts of the code (blue and green boxes) respectively to accomplish the computation task jointly. The CPU side (host side) is responsible for program flow control, memory allocation and copying, and launching functions running on the GPU side (device side). The GPU side is responsible for high-throughput intensive computing tasks, and when the computation is completed, the results on the device side will be returned to the host side. Therefore, a typical execution pattern is: memory initialization (executed on host), memory copy (from host to device), execution (executed on device), and memory copy (from device to host).

The function running on the GPU is called the kernel function, and it is launched from the host side as follows:

```
dim3 gridDim(x, y, z);
dim3 blockDim(x, y, z);
functionName<<<gridDim, blockDim, sharedMemSize, streamIdx >>>(inputParam);
```

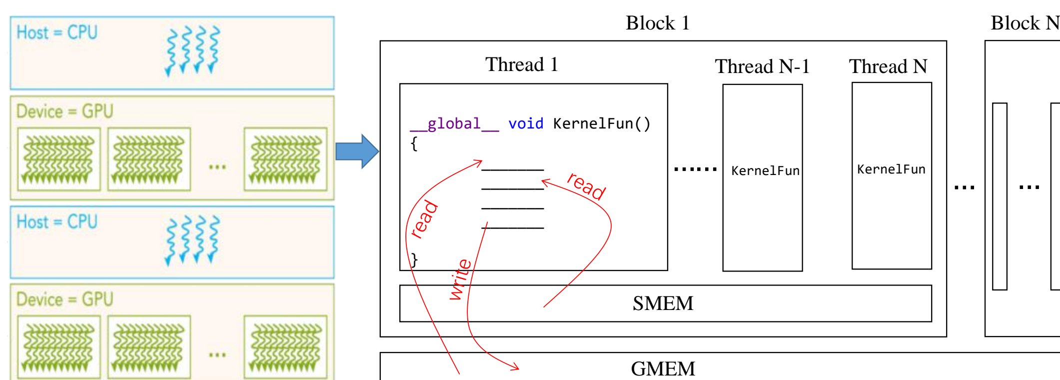

Figure 1-3: The programming pattern for the GPU in heterogeneous computing systems: the launch and execution of kernel function.

Unlike functions executed on the host side, launching a kernel function requires specifying the thread grid and block size, the amount of shared memory to allocate, and the execution stream. The essence of a kernel function is the definition of a single thread's behavior. When the host launches a kernel function, it is necessary to specify the number and shape of threads to be started, that is, thread blocks and thread grids. As illustrated in Figure 1-3, since all threads are defined by the same kernel function, their behaviors are the same, i.e., they perform the same calculations and flows, but they can index different addresses in memory (shared and global memory) based on their unique index numbers in the grid and block. This mode is called Single Program Multiple Data (SPMD). In addition, each thread has its own register space, which means that for a variable defined in a kernel function, depending on how many threads are launched, there will be that many variables allocated in the register file. For

thread blocks, they have their own shared memory, which means that only threads in the same thread block can access each other's shared memory.

Thanks to the numerous computing cores in the GPU, all launched threads can be executed in parallel. However, threads are not mapped to computing cores one-to-one, they are executed dynamically based on the software and hardware hierarchy. In CUDA programming, the corresponding relationship between the GPU hardware and software hierarchy is shown in Figure 1-4.

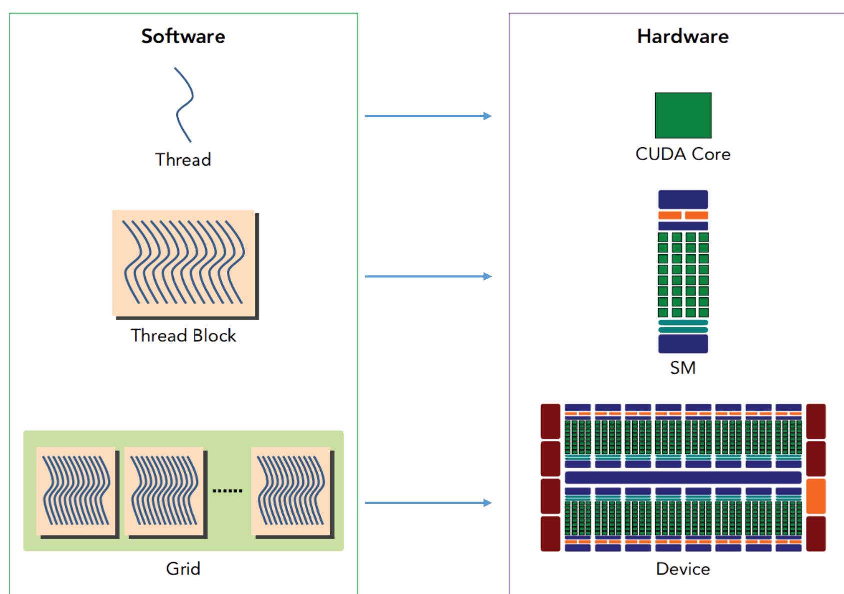

Figure 1-4: The hierarchy correspondence between software and hardware in CUDA programming [88].

The processing object of SM, the heart of GPU, is the thread block, and it should be noted that one SM can handle multiple blocks simultaneously instead of just one block. The exact number of blocks does not depend on how many SPs are in one SM or how many threads are in a block, but on how many threads can be processed by one SM at the same time, that is, it depends on the hardware resources (registers, shared memory, etc.) required by one thread and the total hardware resources that one SM can provide. An SM can run several concurrent blocks, and all threads in the blocks are organized into fundamental units, warps (32 threads per one), and handed off to SPs/CUDA cores. Further, the processing of blocks by SMs is not synchronous, that is, when an SM finishes processing its resident blocks, it will dynamically look for other unprocessed blocks to process. The processing object of SP is the single thread, which means one SP processes only one thread at a certain moment. In Practice, SPs are organized to process warp, and the 32 threads in a warp will strictly execute the same instruction simultaneously by SPs, but for different data. This mode is called Single Instruction Multi Data (SIMD).

The principle of execution mode in CUDA is that the hardware should not be idle. For example, when threads in a warp or block are engaged in memory operations (e.g., loading or writing) which are very time-consuming, SPs and SMs will switch to other resident warps or blocks to execute their computing operations instead of waiting for the end of the memory

operations. To keep the hardware units in the GPU as busy as possible (i.e., maximum utilization), the number of thread blocks launched for the kernel function should be much larger than the number of SMs in the GPU, and the total number of threads should significantly exceed the total number of SPs, and preferably by an integer multiple of 32.

1.2.3 Necessity of GPU computing in EEG processing

EEG signal processing is an important cornerstone of neuroscience, cognition, psychology and other brain-based research fields. The rich physiological and pathological information of brain activity contained in EEG is crucial for diagnosing neurological diseases, understanding brain function and developing brain-computer interfaces. However, due to the complexity, high density and real-time processing requirements of EEG signals, processing EEG signals effectively is computationally intensive. Graphics processing units (GPUs), with their parallel processing capabilities, have emerged as potential game changers in this field. Despite the bright prospects, the application of GPUs in EEG signal processing remains an open question, full of opportunities and challenges.

The unique characteristics of EEG data and the limitations of conventional processing methods are the motivation for implementing EEG processing using GPUs. EEG is usually measured at a high sampling rate (1024Hz, 2048Hz) and in the high-density EEG measurement, the number of electrodes can reach 256. In the practical EEG experiment, the minimum number of subjects is 20, and each subject needs to be measured from minutes to hours depending on the task. These factors can make the measurement easily accumulate several gigabytes of data, which requests robust computational resources for processing. On the other hand, the processing pipelines of EEG signals are complex and involve several computationally demanding steps. First, in the preprocessing stage, operations such as artifact removal, filtering, and normalization are required for massive EEG data. Secondly, in feature extraction, time-consuming algorithms such as Wavelet Transform (WT) [89], Independent Component Analysis (ICA) [90], Empirical Mode Decomposition (EMD) [91], and Modulation Index (MI) [92] are involved in extracting the time domain, frequency domain, spatial domain, and statistical features of EEG signal. Finally, classification and prediction operations based on machine learning algorithms [93] might be involved in interpreting the signal. GPUs, with their massively parallel architecture, excel in handling such complex and time-consuming tasks, making them an ideal choice for implementing EEG processing algorithms. Applications such as brain-computer interfaces (BCI) [93], neurofeedback [94], epileptic seizure detection [95], and measurement visualization [96] require real-time processing to ensure timely intervention or interaction. Traditional CPU-based computing systems designed for sequential processing often cannot meet the throughput requirements of these applications.

Modern neuroscience research increasingly relies on large-scale EEG datasets, such as those collected in multi-center studies [97] or publicly available repositories [98]. Processing such large-scale datasets using any algorithm is computationally expensive, and GPUs not only provide higher computing power but also support scalable processing, enabling researchers to analyze these datasets more efficiently. Extracting effective features or information from massive EEG data often requires advanced algorithms. For example, the rise of deep learning models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks

(RNNs) are increasingly used for tasks such as emotion recognition [99], mental workload estimation [100], and neurological disease diagnosis [101]. These models are computationally intensive and usually require GPU acceleration for effective training and deployment. By reducing computation time and cost, GPU-accelerated EEG analysis can enhance precision medicine approaches, tailoring interventions based on individual unique neural patterns. For example, personalized neurofeedback therapy for ADHD or epilepsy can become more accurate and effective [102], [103]. For researchers, GPUs can make the development of EEG processing algorithms more efficient, allowing them to validate ideas and get feedback faster, thereby facilitating efficient EEG research.

Despite their potential, GPU implementation of EEG signal processing algorithms is not without hurdles. These challenges underscore why this remains an open problem, ripe for exploration. First, many traditional EEG signal processing algorithms are not inherently parallelizable. Adapting these algorithms to take full advantage of GPU architecture requires extensive redesign, including redesigning workflows and optimizing memory usage. Although GPUs offer high throughput, high-performance EEG processing is still very difficult to implement due to the complex hardware-software hierarchy and the GPU programming model that is completely different from that for sequential devices. Especially on heterogeneous computing systems composed of GPUs and other processors, high-performance implementation requires a deep understanding of GPU hardware and software. Additionally, GPUs are resource-intensive and may not be suitable for all environments, especially in portable or wearable EEG systems where energy efficiency is important. Developing lightweight and power-efficient GPU-based solutions is a critical area of research [104]. Last but not the least, most of the EEG processing scripts used in the EEG community are implemented by MATLAB, and the use of GPUs requires expertise in parallel computing, which may not be common among EEG researchers. Bridging this knowledge gap is critical to advancing GPU implementation in the field of EEG processing.

The application of GPUs in EEG signal processing represents a frontier with immense potential to revolutionize neuroscience, cognition, psychology, and other brain-based research fields. The parallel processing capabilities of GPUs offer a path to overcoming the computational bottlenecks inherent in EEG analysis. However, algorithm adaptions, overhead optimization, energy efficiency, and knowledge gaps between different research communities highlight that this field remains an open problem, requiring concerted efforts from researchers and engineers. By developing GPU implementations for EEG processing algorithms, accelerating pipeline processing, and enhancing energy efficiency, the research community can unlock the full potential of GPUs in EEG signal processing.

1.3 Thesis Organization

This section describes the structure of the thesis. **Chapter 1** starts with the physiological background knowledge and fundamental signal characteristics of EEG. It then explores the development and methods of EEG signal processing, followed by an introduction to the principles of GPU parallelization. This includes a discussion on the development of GPU and high-performance computing, an overview of GPU hardware architecture and CUDA

programming model, and an explanation of the critical role of GPU computing in EEG signal processing.

Chapter 2 begins by proving the development history of signal decomposition, highlighting its applications across various fields, along with its advantages and the challenges it faces. It also provides an overview of the algorithms used in frequency decomposition and spatial decomposition.

In **Chapter 3**, I design and develop a massively parallel and performance-optimized GPU implementation of the Improved Complete Ensemble EMD with the Adaptive Noise (CEEMDAN) algorithm that significantly reduces the computational time (from hours to seconds) of time-frequency analysis and oscillation extraction of EEG signal.

An efficient GPU implementation of the Multivariate Empirical Mode Decomposition (MEMD) method is presented in **Chapter 4**, which is used for speeding up the process of decomposing non-stationary multi-channel bioelectric signals into different oscillation modes.

In **Chapter 5**, I introduced an innovative application of GPU tensor cores to the EEG spatial decomposition algorithm, Independent Component Analysis (ICA), the de facto standard for EEG artifact removal. This method, while highly effective, is computationally expensive. By using tensor cores, I significantly enhanced its computational efficiency.

Finally, in **Chapters 6** and **7**, I conclude and summarize the results of the thesis work and list my publications related to the presented work in the final chapter.

2 SIGNAL DECOMPOSITION METHODS

2.1 Introduction to Signal Decomposition

Signals are carriers of information. In reality, most of the signals measured are usually not carriers of a single piece of information but contain multiple pieces of information or even unwanted noise, that is, mixed composite signals. The mix and complexity of signals make it challenging to analyze signals and extract useful information. Signal decomposition is a fundamental concept in the field of signal processing, allowing mixed complex signals to be broken down into simpler components (e.g. sine/cosine waves). This methodology is pivotal across various domains, including engineering, medicine, economics, and environmental sciences [105], [106], [107], [108]. By transforming complex signals into manageable parts, researchers can uncover hidden patterns, enhance clarity, and improve overall system performance. Such capabilities make signal decomposition indispensable in the modern era of signal processing.

The concept of signal decomposition has its roots in mathematical theory and early developments in Fourier theory. In the 19th century, Jean-Baptiste Joseph Fourier proved that any periodic signal could be expressed as a sum of sinusoidal components. This insight set the stage for modern signal decomposition techniques and foundationally changed how signals were analyzed, providing the basis for frequency-domain representation. With the development of signal processing technology, Fourier's theory has been extended to accommodate more complex signals. Methods like the Short-Time Fourier Transform (STFT) and Wavelet Transform (WT) emerged to address time-varying signals, offering both time and frequency resolution. Hilbert-Huang Transform (HHT) [109] and Empirical Mode Decomposition (EMD) [110] later expanded the toolkit, enabling the effective decomposition of non-linear and non-stationary signals. These advances have broadened the applicability of signal decomposition, allowing it to address the challenges of increasing complexity of signals in various scientific and engineering fields.

The main purpose of signal decomposition is to extract useful information from complex signals by breaking them into simpler components. By simplifying complex signals, the underlying structure and patterns of the signal can be effectively revealed. It helps reduce noise, isolating and removing unwanted components to enhance the quality of the signal for further analysis [111], [112]. Signal decomposition also facilitates feature extraction, revealing hidden patterns or characteristics that may not be immediately visible in the raw data [113], [114]. Furthermore, it enables data compression by representing signals with their fundamental components, making storage and transmission more efficient [115], [116]. Finally, the simplified components often correspond to more intuitive physical or practical phenomena [117], improving the interpretability of results and making the theories more applicable to real-world scenarios.

A wide range of techniques is available for signal decomposition, each suited to different types of signals and analytical objectives. but they can basically be categorized into frequency-based and space-based decomposition methods. The Fourier Transform (FT)

converts time-domain signals into frequency-domain representations [118], making it ideal for identifying periodic components in stationary signals. Short-Time Fourier Transform (STFT) offers a localized frequency analysis by segmenting signals into overlapping time windows, providing a dynamic view of frequency variations [119]. The Wavelet Transform (WT) provides both time and frequency localization, enabling the analysis of non-stationary signals by capturing variations in both domains [120]. Empirical Mode Decomposition (EMD) is particularly effective for non-linear and non-stationary data, breaking signals into Intrinsic Mode Functions (IMFs) that reflect their unique oscillatory modes [121]. For multi-channel signals collected simultaneously by different channels at different locations, each channel collects a mixed signal from all signal sources, and the spatial decomposition method is used to unmix the multi-channel signal. Independent Component Analysis (ICA) is a statistical method used to decompose a multivariate signal into its underlying independent source components by assuming that these source signals are statistically independent and non-Gaussian in nature [122]. Each technique has its strengths and limitations, requiring careful selection based on the nature of the signal and the goals of the analysis.

Signal decomposition has diverse applications in various fields, enabling more effective analysis and interpretation of complex data. In biomedical signal processing, it is used to analyze electroencephalograms (EEG), electrocardiograms (ECG), and functional magnetic resonance imaging (fMRI) data by separating meaningful components like frequency bands or removing artifacts such as eye blinks and muscle noise [123], [124], [125]. In telecommunications, it facilitates efficient signal modulation, demodulation, compression, and error detection [126], [127]. In audio and speech processing, signal decomposition techniques improve noise reduction [128], enable speech recognition [129], and optimize audio compression formats like MP3 [130]. In seismology and geophysics, they help differentiate between seismic wave types, remove background noise, and detect microseismical events for earthquake monitoring and resource exploration [131], [132], [133]. In economics and finance, signal decomposition analyzes time-series data to uncover trends, cycles, and irregularities, aiding in forecasting and decision-making [134], [135]. Additionally, in image and video processing, it supports tasks like compression, feature detection, and motion analysis by breaking down visual data into spatial and temporal components [136], [137], [138]. These applications highlight the multi-functionality and significance of signal decomposition in extracting meaningful insights in diverse domains.

Despite its adaptability, signal decomposition is not without challenges. Selecting the appropriate method requires a deep understanding of the nature of the signal and the intended application. Computational complexity, particularly with advanced methods like EMD and ICA, can be unaffordable for large datasets. Interpreting decomposed components remains a challenge in certain domains, requiring domain-specific professional knowledge. Additionally, for the decomposition algorithm itself, ensuring the separation of meaningful signals and noise demands careful numerical validation and tuning of parameters.

The foreseeable future of signal decomposition lies in integrating emerging technologies to address current limitations. Powerful parallel computing capabilities provided by CPU-GPU based heterogeneous computing systems are increasingly being employed to improve the efficiency of signal decomposition, particularly for high-dimensional and complex signals

[139], [140], [141], [142], [143]. A comparison of several successful applications is presented in Table 2-1.

Papers	Application domains	Key techniques	Performance gain
[139]	EMG processing	Bayesian filtering	~100x
[140]	3D image processing	Convolution	~5x
[141]	Neural data processing	MEMD	6x-16x
[142]	Structural health monitoring	FDD	~100x
[143]	Moving object detection	Tensor decomposition	3.8x

Table 2-1: Applications of CPU-GPU systems in signal decomposition and their comparisons.

Decomposition methods based on machine learning algorithms, capable of dynamically adapting to signal characteristics, are also on the rise, promising greater flexibility and accuracy, which of course requires the support of high-performance computing device. Real-time signal decomposition is becoming feasible with lighter algorithms and low-power devices, enabling applications like wearable health monitoring. In summary, signal decomposition is increasingly becoming an indispensable tool in various disciplines, and its applications are far-reaching and impactful. As new techniques and technologies emerge, the potential of signal decomposition to address the challenges of increasingly data-driven scientific research will only grow, cementing its role as a cornerstone of signal processing.

2.2 Frequency Decomposition Methods

In 1822, Fourier proved that any function can be decomposed into several sinusoidal signals of different frequencies [144]. This decomposition method was named Fourier Transform (FT) and is widely used in signal frequency analysis. For a continuous signal $f(t) \in L^2(R)$, its Fourier transform is as follows:

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-j\omega t}dt \tag{2.1}$$

$$e^{-j\omega t} = cos(\omega t) - jsin(\omega t) \tag{2.2}$$

where ω is the continuous frequency. Usually, analog signals (i.e. continuous signals) need to be sampled before they can be stored and processed by computers, that is, the input signal will be discretized in the time domain. For a discrete sequence $x(n)$, its Discrete Time Fourier Transform (DTFT) is:

$$X(e^{j\omega}) = \sum_{-\infty}^{+\infty} x(n)e^{-j\omega n} \tag{2.3}$$

Where ω is still a continuous frequency, but if the frequency information of the input signal needs to be stored and processed in a computer, it should also be discretized. Therefore, for a discrete sequence $x(n)$ of length N, its Discrete Fourier transform (DFT) is:

$$X(k) = \sum_{n=0}^{N} x(n) e^{-j\frac{2\pi}{N}nk} \tag{2.4}$$

Where k is the index of the discrete frequency, and essentially, DFT is the accumulated correlation between the discrete input signal and discrete multi-frequency sinusoidal signals. The most commonly used fast Fourier transform (FFT) is a fast implementation of DFT, which can convert the time domain information of the input signal into the frequency domain. However, equation 2.4 shows that the accumulation scope of FFT is the entire time domain of the signal, so the result of FFT does not contain any time domain information. As shown in Figure 2-1, signals $s1$ and $s2$ are completely different in the time domain, but after performing FFT on them, their frequency domain information is the same, that is, FFT cannot provide us with information about how frequency changes over time.

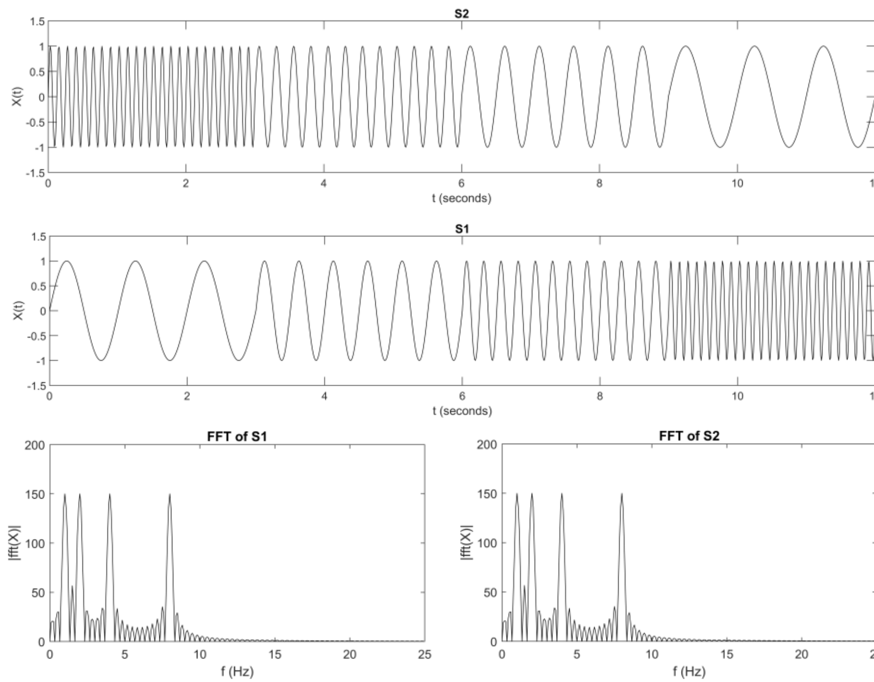

Figure 2-1: Two different signals and their Fourier transform result.

In order to solve the problem of the Fourier transform not providing information about frequency changes over time, Dennis Gabor proposed a technique that introduces time windows into the Fourier transform [145], the short-time Fourier transform (STFT). The basic principle of STFT is to introduce a window function $g(t)$ sliding along the time axis in the Fourier transform, and then perform the Fourier transform on the product of function $g(t - \tau)$ (τ is the midpoint of the time window) and input signal, that is, perform Fourier transform on the part of the input signal within the time window range. In this way, the frequency information in different sliding time windows can be obtained.

Although STFT can combine the time domain information and frequency domain information of the input signal to provide time-frequency information by sliding the time window and decomposing the signal within the time window, it is not adaptive due to the fixed time window size and the pre-defined sinusoid function. If the time window is too large, the temporal resolution will be low, and if the time window is too small, the frequency resolution

will be low, especially the low-frequency information may be lost. Therefore, as shown in Figure 2-2, STFT is not suitable for processing non-stationary signals whose frequency or distribution varies with time, such as EEG signals.

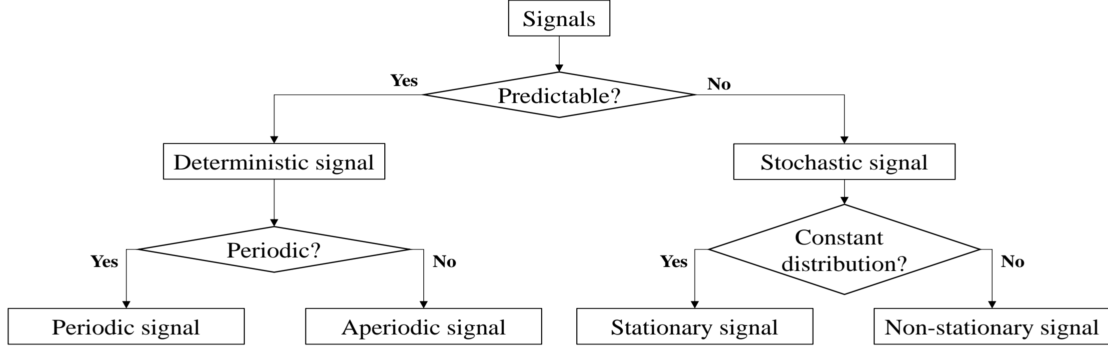

Figure 2-2: The signal classifications and properties.

In order to extract more accurate time-frequency information of non-stationary signals, Morlet proposed wavelet transform (WT) [146], which uses variable-sized sliding time windows and wavelets to analyze signals. As shown in Figure 2-3, different from the pre-defined sinusoid sinusoidal basis function of Fourier transform, WT uses wavelet basis functions with different scales to convolve the signal.

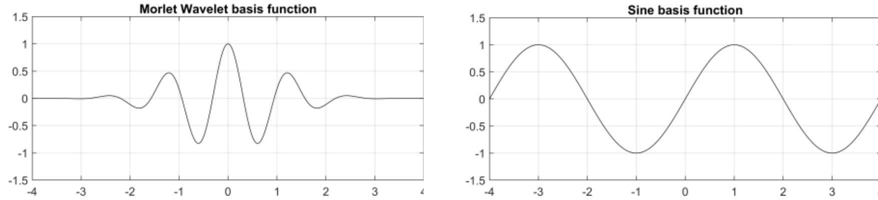

Figure 2-3: The wavelet basis function (Morlet wavelet) used in wavelet transform and the sine/cosine basis function used in Fourier transform.

Since the basis function is variable, WT can strike a balance between time resolution and frequency resolution, that is, long time windows and lower frequency wavelets are used to convolve the low-frequency components in the signal, while short time windows and higher frequency wavelets are used to convolve the high-frequency components. For a function $\Psi(x) \in L^2(R)$, if its Fourier transform satisfies:

$$C_{\Psi} = \int_{-\infty}^{+\infty} \frac{|\Psi(\omega)|^2}{|\omega|} d\omega < \infty \tag{2.5}$$

Then $\Psi(x)$ is a wavelet basis function or mother wavelet function. Furthermore, by shifting and stretching or shrinking function $\Psi(x)$, a set of wavelet sequences can be represented:

$$\Psi_{a,b}(x) = \frac{1}{\sqrt{|a|}} \Psi\left(\frac{x-b}{a}\right) \tag{2.6}$$

Where a is the scale factor a to represent the "stretching" or "shrinking", and b is to represent the "shifting", i.e., the position of the wavelet basis function. Therefore, for a continuous function $f(x) \in L^2(R)$, its wavelet transform is:

$$W(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} f(x)\, \Psi\left(\frac{x-b}{a}\right) dx \tag{2.7}$$

Where a and b are continuous scale factors, so the above equation is called Continuous Wavelet Transform (CWT) [147]. As shown in Figure 2-4, the CWT results of signal $s1$ and $s2$ can differentiate the distribution of different frequencies in the time domain.

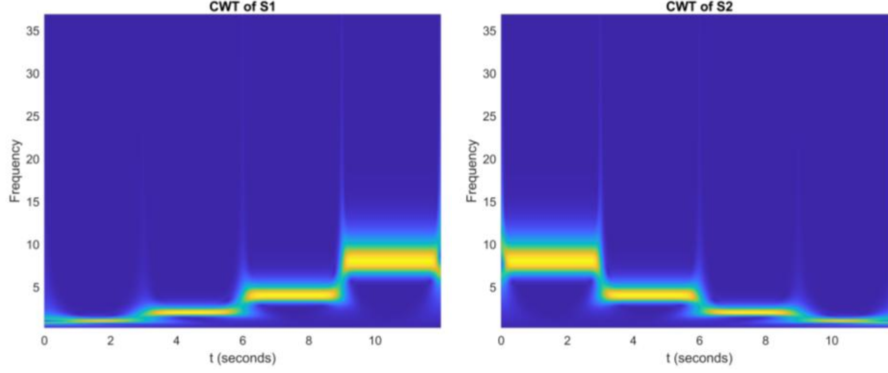

Figure 2-4: CWT results of signal $s1$ and $s2$ in Figure 2-2.

CWT can extract the time-frequency information of the non-stationary signal in a more accurate way, which is helpful in the comprehensive analysis and reconstruction of signals, but it also produces redundancy in results. For example, signals $s1$ and $s2$ contain only four distinct frequency components respectively, but CWT will continuously compute a wide frequency range, which may be problematic in signal decomposition and compressing, and also increase the computational cost. Therefore, the scale factors a and b can be discretized to solve these problems. Taking $a = {a_0}^j$, $b = k{a_0}^j b_0$, and $j, k \in Z$. Then the discretized wavelet sequence is:

$$\Psi_{j,k}(x) = {a_0}^{-\frac{j}{2}} \Psi\left({a_0}^{-j} x - k b_0\right) \tag{2.8}$$

Correspondingly, the Discrete Wavelet Transform (DWT) [148] of $f(x) \in L^2(R)$ is:

$$W(j, k) = {a_0}^{-\frac{j}{2}} \int_{-\infty}^{+\infty} f(x) \Psi\left({a_0}^{-j} x - k b_0\right) dx \tag{2.9}$$

Unfortunately, both the Fourier and the wavelet transform require pre-defined basis functions of fixed frequencies given by analytic formulae as templates, which is not a suitable approach for natural signals, such as EEG, where the signal shape, amplitude, and frequency can change arbitrarily. In fact, in a non-stationary, non-periodic signal, there are multiple oscillation modes at any time instance, which means that the signal at each time point contains multiple instantaneous frequencies [149] so, the Fourier-based or wavelet-based time-frequency analysis of non-stationary signals has only mathematical but no physical meaning. Empirical Mode Decomposition (EMD) is a signal decomposition algorithm proposed by Huang et al. [110] in 1998 that decomposes the signals into a finite number of narrow-band intrinsic mode functions (IMFs) [121]. EMD 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. The produced narrow-band IMFs can be analyzed using the Hilbert transform for the exact instantaneous frequency and phase information, which is significant in non-stationary signal analysis, such as EEG signals.

The exact process of the EMD computation is depicted in the flowchart of Figure 2-6. First,

the extreme points of the input signal are detected (as shown in Figure 2-5), then cubic spline interpolation is used to generate the upper and lower envelopes based on the maximum and minimum points, respectively.

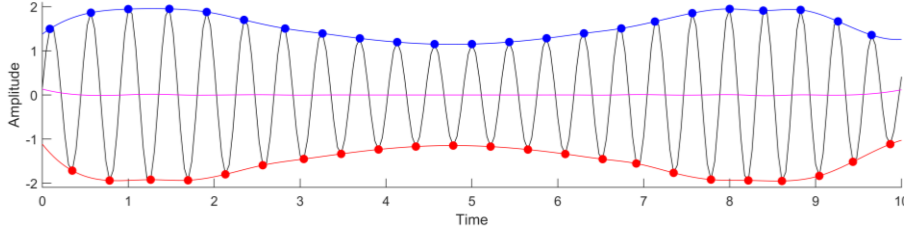

Figure 2-5: Extreme points detection and envelope generation of input signal in EMD algorithm.

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 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 of the mean envelope should be approximately zero.

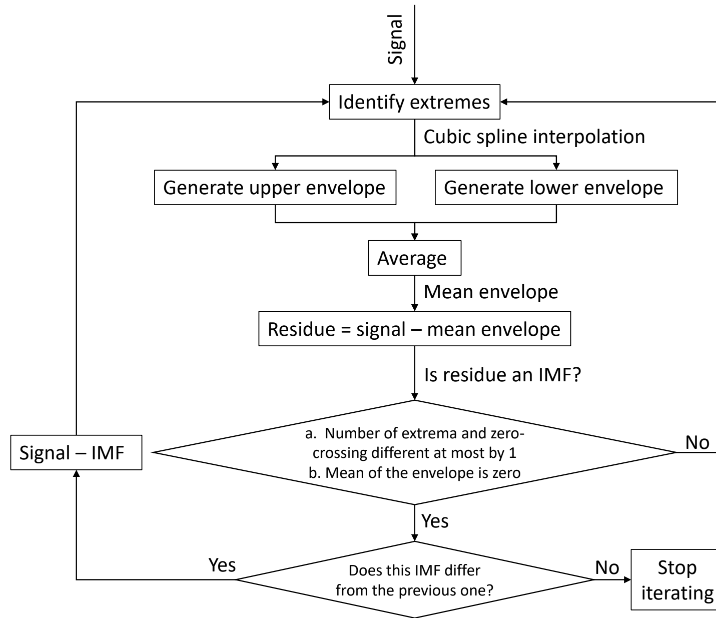

Figure 2-6: The flowchart of the EMD processing steps.

In fact, it is very difficult for the residue to satisfy the two conditions at the same time, so standard deviation, SD, between two residues is usually used as the criterion for stopping the sifting process:

$$SD = \sum_{i=0}^{T} \frac{\left|R_{(k-1)}(t) - R_k(t)\right|^2}{\left(R_{(k-1)}(t)\right)^2} \tag{2.10}$$

where R_{k-1} and R_k are the final residual signal in the sifting iteration $k-1$ and k, respectively. If the residue is an IMF, then it will be compared with the IMF generated by the previous iteration. If they are the same, the process will be stopped, if not, the IMF will be subtracted from the input signal creating a new input signal for the next iteration round. After

finishing the decomposition, the original signal can be represented as:

$$X(t) = \sum_{i=1}^{N} IMF_i(t) + R_N(t) \quad (2.11)$$

At any time point, each IMF represents a single instantaneous frequency value, i.e., each IMF describes a single oscillatory mode, consequently using the Hilbert transform analysis of the IMF, the time-dependent instantaneous frequency of the given IMF can be obtained.

Hilbert transform is a powerful tool for signal time-frequency analysis. The Hilbert spectrum can reflect the relationship between the instantaneous frequency of the signal and time, but due to the mixing of instantaneous information in non-stationary and nonlinear signals, the Hilbert transform cannot fully characterize its instantaneous phase and frequency features. The EMD method can stabilize the non-stationary signal by decomposing it into IMFs, therefore, the Hilbert transform (HT) of the IMFs can describe the time-frequency features of the signal more accurately:

$$HT(t) = \frac{1}{\pi} P \int_{-\infty}^{+\infty} \frac{IMF(t')}{t - t'} dt' \quad (2.12)$$

where P indicates the Cauchy principal value. Using this formula, the analytical signal $Z(t)$ of the IMF can be obtained as:

$$Z(t) = IMF(t) + iHT(t) = a(t)e^{-\vartheta(t)} \quad (2.13)$$

where $\vartheta(t)$ is the instantaneous phase function. Based on the rate of change of the phase, the instantaneous frequency is represented as:

$$f(t) = \frac{d\vartheta(t)}{dt} \quad (2.14)$$

This method of combining EMD and Hilbert transform is called Hilbert-Huang transform (HHT).

The EMD method, unfortunately, is not robust enough in the presence of noise. As shown in Figure 2-7, intermittent noise can result in mode mixing, i.e., one IMF contains oscillations of largely disparate scales. Mode mixing will not only lead to wrong time-frequency distribution but also may make IMF lose its physical meaning.

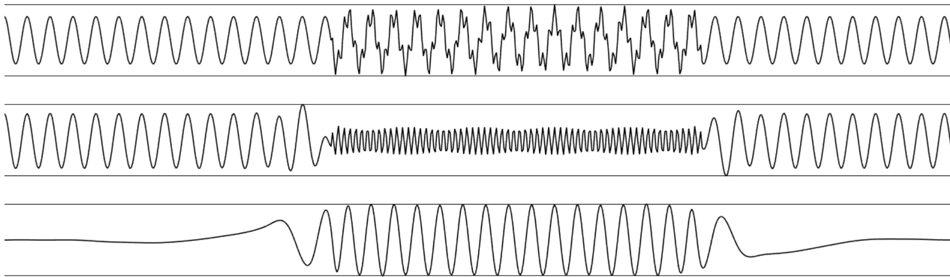

Figure 2-7: Mode mixing problem caused by intermittent noise in the input signal.

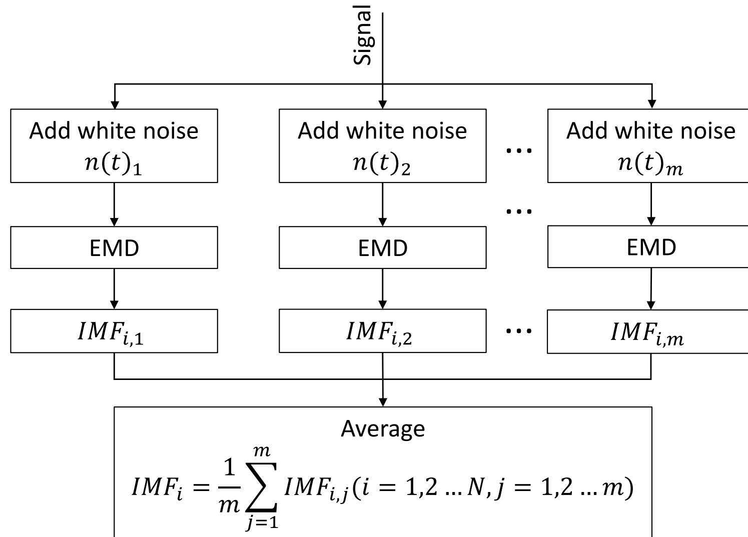

Figure 2-8: High-level structure of the EEMD calculation process.

In order to deal with the mode mixing problem, Wu et al. proposed a noise-assisted signal decomposition method in 2009 [150], the Ensemble Empirical Mode Decomposition (EEMD), shown in Figure 2-8. Based on the uniform distribution of white noise spectrum, the algorithm adds white noise to the raw signal before decomposition, so that the intermittent noise is submerged in the added noise. Therefore, the distribution of extreme points of the signal will be more uniform, which can effectively suppress the mode mixing caused by intermittent factors. The EEMD method replicates the signal by adding to it random white noise. Then, these signals will be decomposed individually. The number of IMFs generated by each signal group may be inconsistent, which will lead to the inability to align each IMF during the averaging operation.

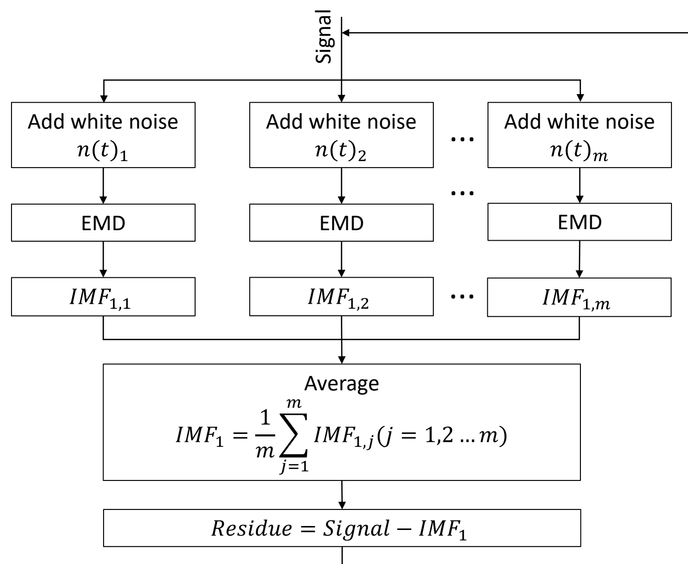

Figure 2-9: High-level structure of the CEEMDAN calculation process

In order to solve the IMF alignment problem and improve the decomposition efficiency, Torres *et al*. proposed the complete EEMD with adaptive noise (CEEMDAN) in 2011 [151]. Compared with the EEMD method, CEEMDAN first adds noise to the raw signal to create

the set of input signals, then calculates only the first IMF corresponding to each group, and averages these IMFs to obtain the first real IMF. Next, the real IMF will be subtracted from the raw signal and the next iteration is started. Figure 2-9 shows the flow of on iteration. Since the averaging operation is performed after the first IMF of signals with white noise is calculated in each iteration, CEEMDAN solves the IMF alignment problem, and by controlling the parameters of the white noise added in different iterations, faster iterations can be achieved. However, the CEEMDAN method is still a decomposition method for single-channel signals. Although it solves the alignment problem of multiple groups of IMFs obtained from single-channel signals with different noise, it cannot solve the alignment problem of IMFs obtained by multi-channel decomposition.

Rehman *et al*. proposed the multivariate empirical mode decomposition (MEMD) method in 2010 [152], [153] to solve the alignment problem of multiple IMFs obtained from multi-channel signal decomposition. As shown in Figure 2-10, the MEMD method regards the input *N*-channel signal as a curve in $(N + 1)$ dimensional space, where the extra dimension is the time, and then sets a uniformly distributed set of direction vectors in the *N*-dimensional space. Curves are projected onto planes composed of different direction vectors and time axes to generate upper and lower envelopes of the projected signal on each direction vector, and the mean envelope is calculated from the envelopes along all directions, next comes the process of iteration and sifting. Although MEMD solves the problem of IMF alignment of multi-channel signals, it still suffers from the problem of mode mixing that is present in the one-dimensional EMD method.

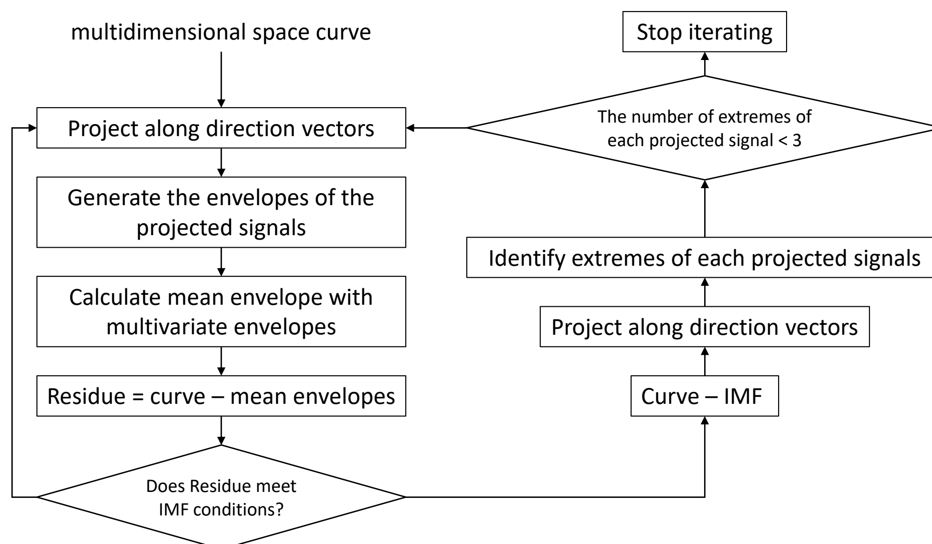

Figure 2-10: The flowchart of the MEDM processing steps.

As an improvement, Rehman *et al*. proposed the noise-assisted MEMD (NA-MEMD) method [154] to solve the mode mixing problem in MEMD. NA-MEMD does not add noise to the original multi-channel signal directly, but adds noise to the (*N*+1)-dimensional curve composed of the multi-channel signals, so it adds several noise channels and performs MEMD on the new multi-channel signals composed of the original multi-channel signals and the noise channels. Among all the IMFs obtained by decomposition, the IMFs corresponding

to signal channels can be obtained by eliminating the IMFs corresponding to the noise channels.

EEG signals are generally multi-channel signals measured by multiple electrodes. Also, the signal of each channel is non-stationary, non-periodic and nonlinear, with certain level of correlations among channels. Since the MEMD method solves the alignment problem of IMFs, it can also preserve the mutual information among channels on the basis of analyzing non-stationary and nonlinear signals.

Signal frequency decomposition methods represented by EMD and its variants are commonly used in the preprocessing or feature extraction stage of the EEG signal processing pipeline. Their adaptive and data-driven nature enables them to efficiently decompose non-stationary signals into IMFs, each containing only a single oscillatory mode. These methods provide higher resolution in capturing transient and localized EEG signal dynamics as compared to conventional fixed-base techniques (e.g., wavelet or Fourier transform). However, they also introduce a significant computational bottleneck in end-to-end EEG processing pipelines. The iterative sifting process inherent to EMD, which involves repeated extrema detection and envelope interpolation, is computationally intensive and challenging to parallelize. Furthermore, the time complexity grows with the number of EEG channels and signal length, making EMD methods less scalable for high-dimensional datasets. Their sensitivity to noise and lack of a universal stopping criterion often necessitate the use of ensemble averaging or post-processing, further increasing computational load. Therefore, without significant optimization, EMD is not suitable for large-scale applications in end-to-end EEG signal processing pipelines. These constraints highlight the need for high-performance EMD implementations that can retain the benefits of EMD while reducing latency and computational cost—which is important for accelerating end-to-end EEG signal processing pipelines.

2.3 Spatial Decomposition Methods

The focus of contemporary signal processing technologies is shifting from Gaussian, stationary, linear signals to non-Gaussian, non-stationary, nonlinear signals, that is, blind signals. The most challenging part of blind signal processing is how to separate the signals generated by different sources in the collected mixed signal, this is the Blind Signal Separation/Blind Source Separation (BSS) problem. The BSS problem originated from the cocktail party problem proposed by Cherry [155] in 1953. As shown in Figure 2-11, at a cocktail party, there are different sound sources (speaker #1, #2, #3) that exist simultaneously, and they are recorded by several microphones (#1, #2, #3) at the same time. Therefore, each microphone actually records the mixed signal generated from three speakers, that is, three mixed signals will be produced. In this case, how to decompose the recorded mixed signals to obtain the original signals of the three sound sources is an interesting topic. When people are interested in a certain speaker and pay attention to him, the brain can automatically extract the voice of the corresponding speaker and block out other voices. The spatial decomposition method is used to achieve a similar function, and the most effective method is the Independent Component Analysis.

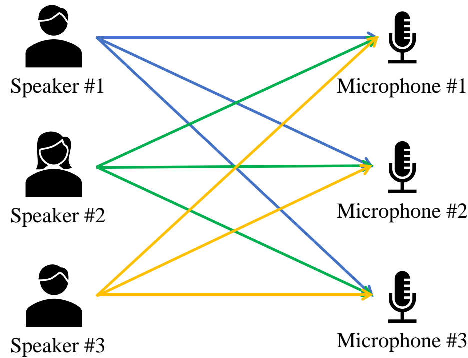

Figure 2-11: Illustration of the cocktail party problem, multiple sound sources are recorded by multiple microphones.

The idea of ICA originated from the work of Ans *et al.* [156] in neurophysiology and Barness *et al.* [157] in communications in the 1980s. Herault and Jutten proposed a neural network-based method [158] to solve the BSS problem. Although this method can solve the problem of separating mixed signals containing two sources, it does not point out the role that the higher-order statistics (HOS) information plays in blind signal separation. But the algorithm is already taking shape as the ICA algorithm. Comon noticed the importance of higher-order statistics information in the ICA problem and defined it using the information theory framework [159]. He also gave an iterative algorithm for the two-source mixed signal separation problem and expanded the algorithm to the multi-source mixed signal situation. In 1991, Comon, Jutten *et al.* described the BSS/ICA problem in detail and gave algorithms as well as its stability analysis in paper [160], [161], [162]. In 1994, Comon summarized the previous work [163], gave the mathematical model of ICA, and answered the following important questions: (i) Does the ICA problem have a solution? (ii) Is the solution unique? (iii) How to quantify the independence between source signals? (iv) How to approach the solution of ICA based on independence? (v) How to measure the effectiveness of the solution? This paper laid the theoretical foundation for ICA and facilitated the development of its variants and applications in various fields.

To address the blind identification problem in beamforming, Cardoso *et al.* proposed a variant of ICA: Joint Approximation Diagonalization of Eigenmatrices [164], [165] (JADE) algorithm, which uses directional vectors for estimation instead of restoring the hypothesized value of the original signal, and fourth-order cumulants are used as an indicator to quantify the statistical independence of the source signal. Moreover, Cardoso improved the calculation of gradient in ICA by introducing a relative gradient and proposed the equivariant adaptive source separation method [166]. Belouchrani *et al.* [167] proposed another ICA variant, the Second Order Blind Identification (SOBI) algorithm. This algorithm exploits the time coherence of the source signals and relies only on stationary second-order statistics based on a joint diagonalization of a set of covariance matrices.

Oja and Karhunen *et al.* conducted an in-depth study of nonlinear PCA in the 1980s and noted the connection between nonlinear Principal Component Analysis (PCA) and BSS/ICA in the 1990s [168], [169], [170], [171]. Based on these works, they proposed an efficient variant of ICA, Fast-ICA [172], [173], also called Fixed-point ICA, which is one of the most used

algorithms. Bell and Sejnowski *et al.* proposed the Infomax-ICA [174] algorithm in 1995 and successfully separated a mixed signal containing 10 speech signals. The algorithm introduces the activation function used in neural networks into ICA and constructs a loss function based on information entropy. The nonlinear activation function can effectively handle the HOS correlation between different independent components, and the introduction of information entropy allows people to understand ICA from the perspective of information theory. Similar to artificial neural networks, Infomax-ICA can efficiently separate the independent components of mixed signals through the convergence of the loss function by the backpropagation algorithm.

The purpose of Infomax-ICA performing iterations to converge the loss function is to update the unmixing matrix to remove the dependencies between components. Before removing the dependencies, it is necessary to remove the correlation, which is also called signal whitening, and the correlation between channels is expressed in terms of covariance. Given signals X and Y with n samples, their covariance is expressed as:

$$Cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}\big(x_i - E(X)\big)\big(y_i - E(Y)\big) \tag{2.15}$$

Where $E(X)$ and $E(Y)$ represent the average values of the input signals, and:

$$Cov(X,Y)\begin{cases} > 0 & Positive\ correlation \\ < 0 & Negative\ correlation \\ = 0 & No\ correlation \end{cases} \tag{2.16}$$

Therefore, for a multi-channel signal, denoted by matrix $\mathbf{A}$, the correlations between the channels can be expressed by its covariance matrix:

$$\begin{bmatrix} Var(CH1) & Cov(CH1,CH2) & Cov(CH1,CH3) & \cdots & Cov(CH1,CHn) \\ Cov(CH2,CH1) & Var(CH2) & Cov(CH2,CH3) & \cdots & Cov(CH2,CHn) \\ Cov(CH3,CH1) & Cov(CH3,CH2) & Var(CH3) & \cdots & Cov(CH3,CHn) \\ \vdots & \vdots & \vdots & & \vdots \\ Cov(CHn,CH1) & Cov(CHn,CH2) & Cov(CHn,CH3) & & Var(CHn) \end{bmatrix} \tag{2.17}$$

According to equation 2.15, if each channel subtracted from its own mean value in advance, the covariance matrix of the multi-channel signal can be easily obtained by $\mathbf{AA}^{\mathrm{T}}$. This process of removing the mean value is also called signal centralization.

If there is no correlation between the channels of the multi-channel signal, then its covariance matrix should be diagonal, but this is not the case. This means that some transformations should be done on the raw signal to diagonalize its covariance matrix, i.e., to remove the correlation between the channels, which requires singular value decomposition (SVD). For the matrix $\mathbf{A}$, its SVD is expressed as:

$$\mathbf{A} = \mathbf{U\Sigma V}^{\mathrm{T}} \tag{2.18}$$

Where Σ is a diagonal matrix, and the elements on the diagonal are the singular values of matrix $\mathbf{A}$. $\mathbf{U}$ and $\mathbf{V}^{\mathrm{T}}$are unitary matrices, which satisfies:

$$\mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{I}$$
$$\mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I} \tag{2.19}$$

By multiplying both sides of equation 2.18 by $\mathbf{A}^{\mathrm{T}}$, the following derivation can be obtained:

$$\begin{aligned}
\mathbf{A}\mathbf{A}^{\mathrm{T}} &= \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}} * \left(\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}}\right)^{\mathrm{T}} \\
\mathbf{A}\mathbf{A}^{\mathrm{T}} &= \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}} * \mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^{\mathrm{T}} \\
\mathbf{A}\mathbf{A}^{\mathrm{T}} &= \mathbf{U}\boldsymbol{\Sigma}^{2}\mathbf{U}^{\mathrm{T}} \\
\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{U} &= \mathbf{U}\boldsymbol{\Sigma}^{2} \\
\mathbf{U}^{\mathrm{T}}\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{U} &= \mathbf{U}^{\mathrm{T}}\mathbf{U}\boldsymbol{\Sigma}^{2}
\end{aligned} \tag{2.20}$$

Then let $\mathbf{Y} = \mathbf{U}^{\mathrm{T}}\mathbf{A}$, the following expression can be obtained:

$$\mathbf{Y}\mathbf{Y}^{\mathrm{T}} = \boldsymbol{\Sigma}^{2} \tag{2.21}$$

$\mathbf{Y}\mathbf{Y}^{\mathrm{T}}$ is consistent with the covariance matrix expression of the centralized multi-channel signal $\mathbf{A}$. More importantly, $\boldsymbol{\Sigma}$ is a diagonal matrix, which means that there is no correlation between the channels of the matrix $\mathbf{Y}$. Therefore, the decorrelated multi-channel signal $\mathbf{A}$ can be expressed as:

$$\mathbf{A}_{deco} = \sqrt{\mathbf{U}^{\mathrm{T}}\mathbf{A}} \tag{2.22}$$

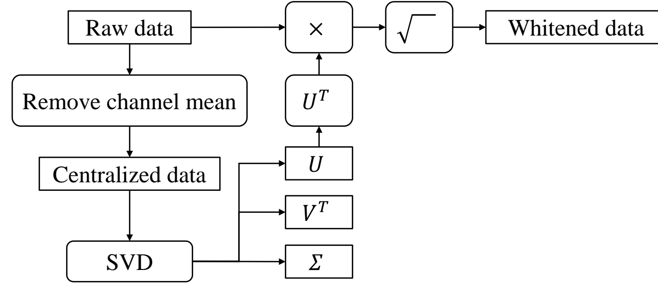

Figure 2-12: The flowchart of ICA's decorrelation preprocessing of raw signals.

Therefore, before the backpropagation algorithm is iterated to converge the loss function, the preprocessing flow of the raw data is shown in Figure 2-12. The raw data needs to be centralized and whitened first.

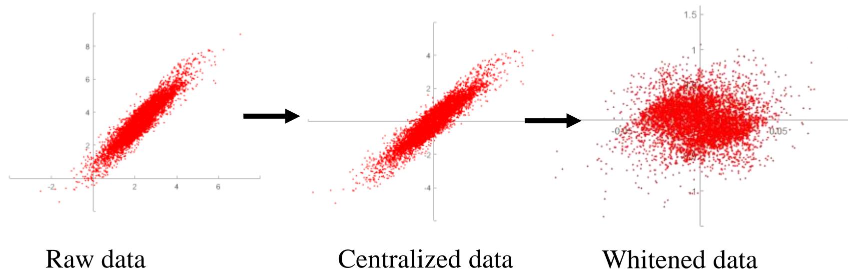

Figure 2-13: The shifting, rotation and stretching during raw signals whitening.

From a geometric point of view, as shown in Figure 2-13, for a 2-channel (2-dimensional) signal, data centralization actually shifts the data to the origin in the coordinate system, and data whitening rotates and stretches the data to decouple each dimension in terms of

correlation. Data whitening can reduce the redundancy of the input signal and improve the convergence of the ICA algorithm.

Figure 2-14 demonstrates the processing flow of preprocessed data. The whitened data $x_i(t)$ is computed by the unmixing matrix W and a nonlinear function $g(\cdot)$, then the independence between the components $y_i(t)$ will be evaluated by the independence criterion (loss function). If the result does not meet the requirements, the backpropagation algorithm is performed according to the learning rate and gradient to update the unmixing matrix W. Following a series of iterations, the dependencies between the components in the output data $s_i(t)$ will be eliminated.

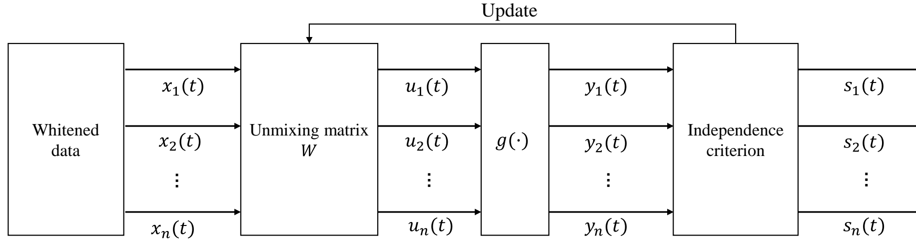

Figure 2-14: The processing flow of solving the unmixing matrix and separating independent components using whitened data.

The mathematical description of ICA is as follows. Assuming an unknown source signal vector $s(t)$ and an unknown mixing matrix $\mathbf{A}$, the signal $x(t)$ measured by the electrodes can be expressed as

$$x(t) = \mathbf{A}s(t) \tag{2.23}$$

The goal is to determine the original, unknown source signal $s(t)$ from $x(t)$ using the inverse of matrix $\mathbf{A}$, $\mathbf{W} = \mathbf{A}^{-1}$. The ICA algorithm is used to compute the unmixing matrix $\mathbf{W}$ to determine the estimate $y(t)$ of the source signal $s(t)$ as

$$y(t) = \mathbf{W}x(t) = \mathbf{WA}s(t) \tag{2.24}$$

To determine whether the calculated components are statistically independent, three commonly used criteria can be used based on: maximum non-Gaussianity, minimum mutual information, and maximum likelihood estimation. In this thesis, the minimum mutual information method (Infomax ICA) is used. Mutual information is defined as

$$I(X;Y) = H(X) - H(X|Y) \tag{2.25}$$

where $H(X|Y)$ represents the conditional entropy, defined as follows

$$H(X|Y) = H(X,Y) - H(Y) \tag{2.26}$$

where $H(X,Y)$ is the joint entropy of X and Y, and the calculation formula for the entropy of a given variable and the joint entropy of two variables is defined as follows:

$$H(X) = -\sum_{x} P(x)logP(x) \tag{2.27}$$

$$H(X,Y) = -\sum_{x,y} P(x,y)logP(x,y) \tag{2.28}$$

where $P(x)$ is the probability distribution function of variable X. The process of minimizing the mutual entropy of X and Y means that the uncertainty of X is continuously reduced after Y is observed. The ICA algorithm based on the minimum mutual information method is called Infomax ICA, which was proposed by Bell and Sejnowski [9]. The pseudocode of the computational algorithm of Infomax ICA is shown in Table 2-2

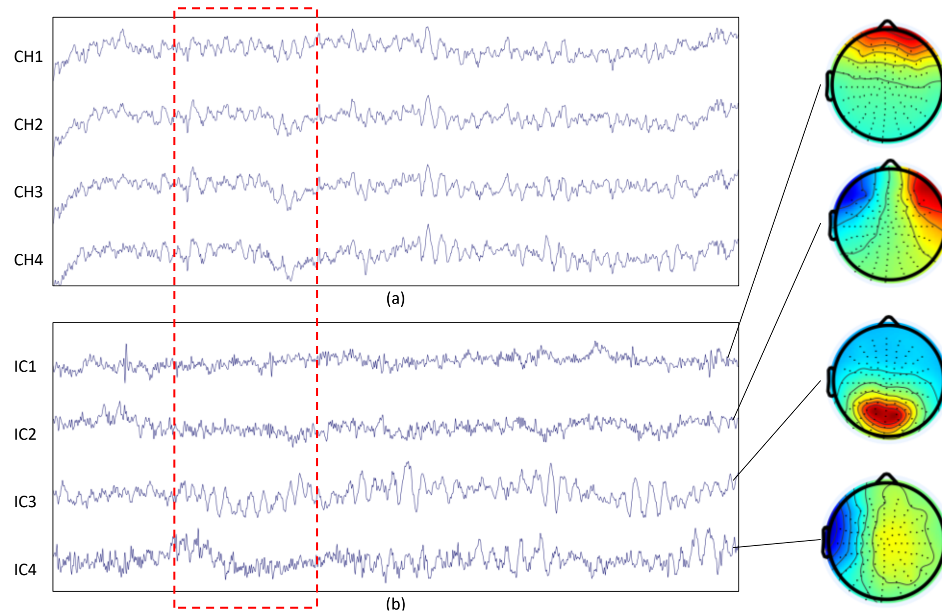

Figure 2-15: The decomposition of a 128-channel EEG recording. (a): The first four channels of input EEG signal. (b): The first four ICs from the decomposition results. The red box highlights the separation of the alpha wave component. The topographic view on the right illustrates the distribution of the ICs across the scalp.

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 2-2: 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 ones. Figure 2-15 presents an example of EEG signals decomposed into independent components (ICs). After applying ICA to the original signal, the mixed source signals are separated. As highlighted in the red box, the alpha waves distributed across

different channels are extracted into a distinct independent component, IC3. The topographic view on the right further illustrates that this component corresponds to alpha wave activity recorded by the occipital electrode.

ICA, as a representative spatial decomposition method, plays a critical role in EEG processing pipelines, primarily during the artifact removal and source separation stages. By separating multichannel EEG recordings into statistically independent components, ICA allows for the identification and removal of artifacts such as eye blinks, muscle noise, and cardiac signals, as well as the isolation of putative neural sources. While this improves signal quality and interpretability, ICA introduces notable computational and practical bottlenecks in end-to-end EEG processing pipelines. The algorithm's computational complexity increases with the number of channels and the length of recordings, and the batch-based backpropagation algorithm introduces a large number of iterations, which greatly increases the computational intensity. Moreover, ICA assumes stationarity and linear mixing, which may not hold for all EEG datasets, potentially reducing its reliability and necessitating manual inspection or further processing steps. Another major bottleneck is the lack of fully automated and generalizable approaches for identifying artifact components, making ICA-dependent pipelines semi-automated at best. These constraints are especially limiting in end-to-end EEG processing pipelines. As a result, while ICA is powerful for spatial decomposition and artifact removal, its integration into fully automated, low-latency, end-to-end EEG processing pipelines remains a key challenge, often requiring simplifications, approximations, or replacement with more computationally efficient alternatives.

3 GPU IMPLEMENTATION OF ICEEMDAN

This chapter describes a massively parallel GPU implementation of the Improved Complete Ensemble Empirical Mode Decomposition with Adaptive Noise (ICEEMDAN) algorithm, a crucial tool for precise time-frequency analysis of non-stationary EEG signals. Due to the data-driven nature, Empirical Mode Decomposition (EMD) and its variants are more suited to extract different oscillations from non-stationary signals, but are too time-consuming for practical use. The proposed implementation significantly reduces the computational time compared to the most widely used MATLAB implementation and enables researchers to perform EMD-based EEG analysis routinely, even for high-density EEG measurements.

3.1 Materials and Methods

3.1.1 Related works

The Improved CEEMDAN algorithm (referred to as ICEEMDAN in the rest of the paper) was developed by Colominas *et al.* [175]. This algorithm solves the mode mixing problem, provides an invertible decomposition and eliminates early noise IMF components from the IMF set. Unlike in 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 white Gaussian noise with zero mean and unit variance to extract noise IMFs, which will be added to the signal as the IMF extraction proceeds. [175]

Improved CEEMDAN algorithm
1. $x^{(i)} = x + \beta_0 E_1(w^{(i)})$ // Signals are added with noises to produce realizations
2. $r_1 = \frac{1}{K}\sum_{k=1}^{K} M\left(x + \beta_0 E_1(w^{(i)})\right) = \langle M(x^{(i)})\rangle$ // Calculated by EMD to obtain residue
3. **for** k^{th} IMF in K IMFs
3.1 **if** k = 1
3.1.1 $d_1 = x - r_1$ // At the first stage, calculate the first mode
3.1.2 $r_2 = \langle M\left(r_1 + \beta_1 E_2(w^{(i)})\right)\rangle$. // Estimate the second residue
3.1.3 $d_2 = r_1 - r_2$ // Define the second mode
end if
3.2 $r_k = \langle M\left(r_{k-1} + \beta_{k-1} E_k(w^{(i)})\right)\rangle$ // from k=3, estimate the residue
3.3 $d_k = r_{k-1} - r_k$ // Compute the kth mode
end for

Table 3-1: Pseudocode of Infomax Independent Component Analysis

The algorithm, described formally, is as follows: let $Ek(\cdot)$ represent the operator that returns the kth 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 pseudocode of the ICEEMDAN algorithm is shown in Table 3-1 [170].

Notice that when creating the signal plus noise realizations, $\beta_{k-1}E_k\left(w^{(i)}\right)$, i.e. the kth IMF of the Gaussian white noise is added to the signal instead of just white noise, such as $\beta_{k-1}w^{(i)}$. Also, the amplitude β_k of the noise mode changes from one iteration to the next as given by $\beta_k = \varepsilon_0 std(r_k)$, where $std()$ is the standard deviation of the signal.

The overall structure of the algorithm is depicted in Figure 3-1, whereas the flowchart of the sifting process for generating the noise IMFs is shown in Figure 3-2.

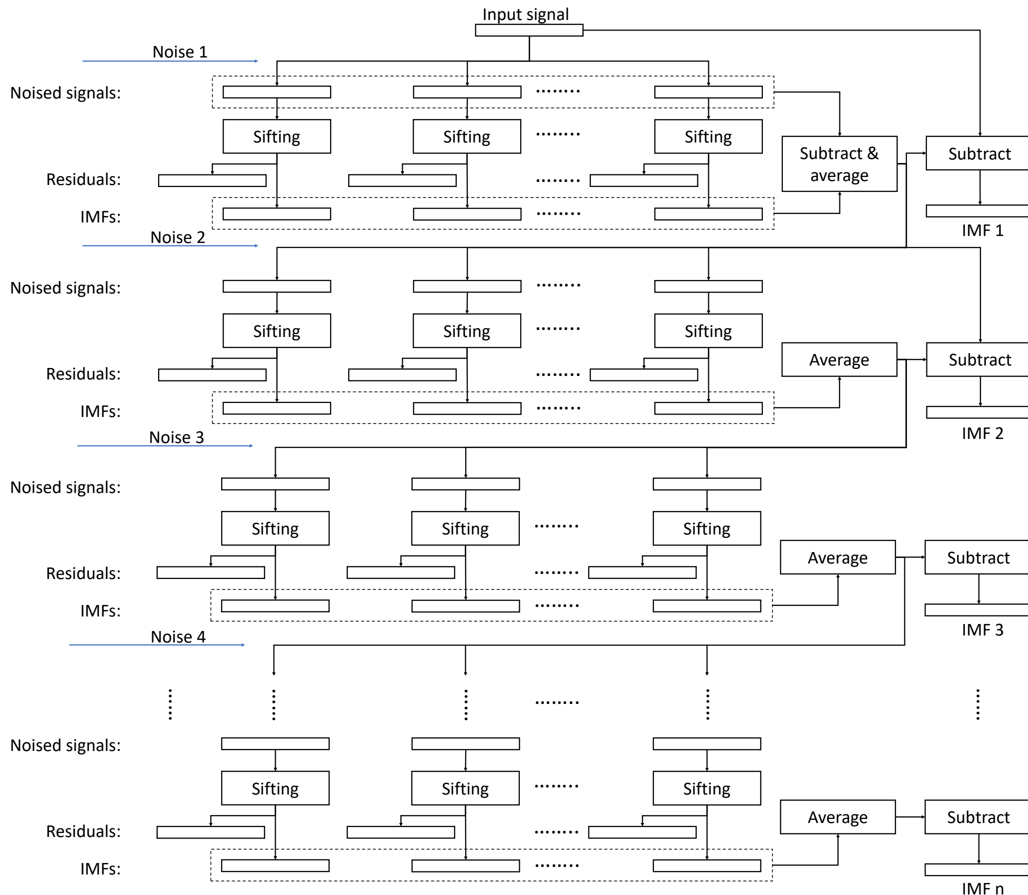

Figure 3-1: The execution flowchart of the ICEEMDAN algorithm. 'Sifting' is the basic processing step of the implementation and the noise inputs are the decomposition results (IMFs) from the Gaussian noise realizations. The sifting process of each realization can be executed in parallel.

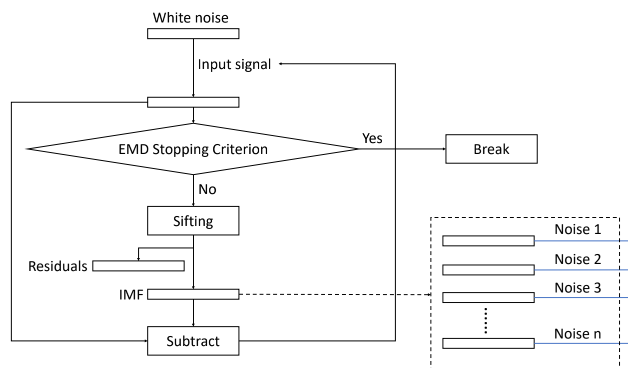

Figure 3-2: The flowchart of the EMD decomposition of the Gaussian noise. The resulting noise IMFs, Noise 1 – n are used as added noise in the algorithm as shown in Figure 3-1.

Several parallel implementations have been developed for the EMD algorithm. The library libeemd [176] 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 [177], [178]. Xie *et al* created a CUDA EMD version for seismic data processing that achieved 4x speedup on a GT240 GPU card [179]. Huang *et al* [180] reported 33.7x speedup on a C2050 GPU using overlapped piecewise cubic spline interpolation technique.

Since the Ensemble EMD 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 [181]. 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 [182]. Chen *et al*. developed a real-time CUDA EEMD implementation for anesthesia monitoring purposes [183]. They showed that it is possible to achieve real-time processing speed with a GTX295 GPU card (31.3x speedup, dual GPU card). A comparison of these parallel implementations of EMD and its variants is presented in Table 3-2.

Papers	Algorithms	Parallel techniques	Performance gain
[176]	EMD/EEMD/CEEMD	OpenMP	~10x
[177]	EMD	CUDA	~29x
[179]	EMD	CUDA	~4x
[180]	HHT	CUDA	33.7x
[181]	EEMD	CUDA	60.62x
[183]	EEMD	CUDA	31.3x

Table 3-2: A comparison of several reported parallel implementations of EMD and its variants.

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.

3.1.2 Design of the parallel implementation

As shown in Figure 3-1, the ICEEMDAN algorithm has more internal dependencies as EEMD. EEMD for a multi-channel EEG dataset can be trivially parallel as the pseudo-code in

Table 3-3 illustrates. All channels and all realizations can be processed simultaneously as they are independent entities, only a final reduction step is required to average the IMFs across the realizations for each channel. The degree of parallelism in the double nested loop is in the order of 10^4.

```
Parallel EEMD Algorithm:
for all channels do in parallel
        for all realizations of the current channel do in parallel
            compute IMFs in parallel
        end for
        average IMFs across realizations in parallel
end for
```

Table 3-3: Pseudo-code of parallel EEMD algorithm.

On traditional CPU-based parallel systems, this provides enough parallelism so that the IMF computation (EMD) is sufficient to be executed in a serial manner. On GPUs, however, where the number of cores is very close to 10^4 and the number of parallel threads must be at least two orders of magnitude higher than the core count, the IMF computation should be performed in parallel, too.

The ICEEMDAN algorithm requires synchronization among realizations after each IMF extraction to compute the IMF means, to add new random noise, and to calculate the input signal for the next iteration. Hence, the high-level structure of the parallel algorithm changes to the following one:

```
Parallel ICEEMDAN Algorithm:
for all channels do in parallel
    generate noise signal w^(i) and its IMFs E_k(w^(i)) by EMD
    for all realizations do in parallel
        add noise IMF E_k(w^(i)) to signal x to obtain the current realization
        compute the local means M(x^(i))
    end for
    residue r_1 by averaging M(x^(i)) across realizations 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(x^(i))
        end for
        residue r_k by averaging M(x^(i)) across realizations in parallel
        compute the mode d_k as d_k = r_{k-1} - r_k
    end while
end for
```

Table 3-4: 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.

3.2 Details of the Implementation

3.2.1 Data structure and initialization

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, I need to perform EMD processing on the gaussian noise to generate noise IMFs which will be added as noises. The `curandGenerateNormal()` function from the cuRand library is used to generate several gaussian noises with mean 0 and variance 1, which are stored in GPU memory. Then, these noise signals will be used as input signals for EMD processing, as shown in Figure 3-2, after several siftings, each noise signal will generate a set of IMFs, therefore, the noise IMFs in the memory is a three-dimensional vector (as shown in Figure 3-3). For ICEEMDAN, each set of IMFs is a realization, and the IMFs in the realization will be added to the corresponding input signals to start the next sifting process (as shown in Figure 3-1). In the initialization section of the implementation, all potential memory spaces are allocated and initialized, necessary host-device memory copies are also performed to avoid host-device memory swaps during iterations.

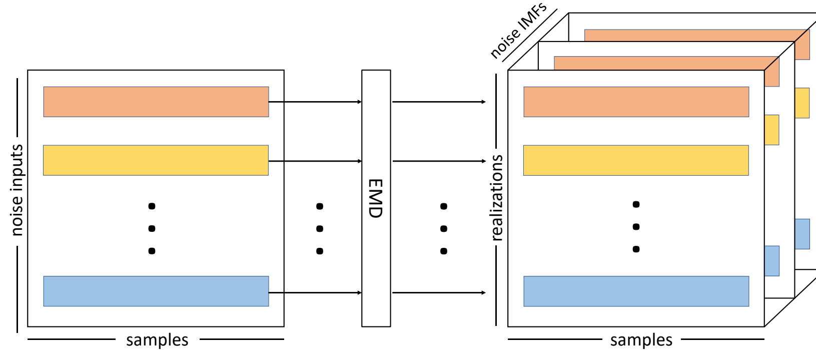

Figure 3-3: 3-D layout in memory of noise IMFs resulting from the decompositions of the different noise vectors. The same color represents the IMFs decomposed from the same noise vector.

The ICEEMDAN algorithm is executed for C signal channels, each containing N samples. During the decomposition of the signal, K modes (IMFs) are extracted from I signal plus noise realizations. The most important data structures and their layout for processing a single channel are described in the following, and the complete list of data structures and their sizes and functions in which they are used are listed in Table S1 in the appendix part.

The input signal x is stored in an N element FP32 vector. The K modes of the Gaussian noise

realizations $w^{(i)}$ are extracted at the beginning of the program and stored in a $K \times I \times N$ element three-dimensional FP32 data matrix. Modes from this matrix will be added to the input signal during the iterative process of IMF extraction. Before executing the Sifting process, the noise realizations are generated and stored in an $I \times N$ FP32 matrix. The output of the Sifting step is I residue signals and I modes, both stored in $I \times N$ FP32 matrices. To minimize memory usage, the data structures of the input signal, the residues and IMFs of the realizations are reused and overwritten in each iteration of the IMF extraction loop. The final result, the extracted averaged IMFs are stored in a $K \times N$ FP32 matrix.

Only the original input signal and the final IMFs are stored in the host (CPU) memory. All other data structures are allocated in the global GPU memory. The input signal is copied to GPU memory before the algorithm starts and the final IMFs are copied back to host at the end. There are no host-device memory copy operations during the execution of the GPU code. The zero mean and unit variance noise is generated by using the `curandGenerateNormal()` function in the cuRAND CUDA library.

3.2.2 The parallel sifting process

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 3-5.

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()
Collect coefficients for interpolation	spline_coefficient()
Cubic spline interpolation	interpolate()
Compute mean envelope	averageUpperLower()
Signal update for next iteration	averageUpdateSignal()

Table 3-5: The major steps of the sifting process and their corresponding GPU kernel/library function.

Each of these steps is 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 generates a set of solutions based on the extreme values of the input signal. These solutions are then passed to the kernel function `spline_coefficient()`, which launches the same number of threads as the interpolation splines. Each thread computes the four coefficients of a cubic polynomial and stores the resulting polynomial coefficients in GPU memory. The coefficients of the cubic polynomial serve as inputs to the interpolation function

`interpolate()`, which launches the same number of threads as the signal samples. Each thread retrieves the four coefficients of the corresponding cubic polynomial from GPU memory and computes the interpolated values of the upper and lower envelopes. The kernel function `averageUpperLower()` computes the average of the two envelopes, launching the same number of threads as the signal samples. Each thread loads the corresponding values from the upper and lower envelopes, calculates their average, and stores the result back in GPU memory.

Once the Sifting process completes, 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 the kernel functions `produceResidue()` and `averageUpdateSignal()`, both of which launch a 2D thread grid composed of 1D thread blocks, ensuring coverage of every sample across all realizations. The number of thread blocks is optimized based on the signal length and the number of realizations, maximizing the utilization of all streaming multiprocessors (SMs). Each thread block consists of 256 threads, efficiently leveraging all streaming processors (SPs) within each SM.

3.3 Results

This section demonstrates the numerical accuracy, speedup achieved, and efficiency of the ICEEMDAN GPU implementation. First, the details of the hardware used during the validation and performance measurements are provided. This is followed by numerical validation results and computational performance measures; all execution time measures were calculated after 5 warm-up cycles with 10 repetitions averaged. Finally, I discuss various performance optimization steps used to improve execution efficiency and device utilization.

3.3.1 Test hardware

Tests and measurements were conducted on three NVIDIA GPUs including gaming and compute-only cards. Each GPU represented different GPU architecture families. Details of the GPUs used in the study are provided in Table 3-6. Specifically, Titan Xp (Pascal) and RTX 3070 (Ampere) gaming cards were used during development and testing, a Tesla V100 (Volta) accelerator card (provided by GPULAB) and a Tesla A100 card (provided by Komondor supercomputer) were used for additional performance measurements. Since these GPUs have different internal architecture, CUDA core counts and theoretical peak performance, they enabled us to explore performance differences attributable to varying hardware parameters. For CPU tests, an Intel i7-9700K 8-core CPU based computer with a Windows 10 operating system and MATLAB 2019a is used.

	Titan Xp	**Tesla V100**	**RTX 3070 mobile**	**Tesla A100**
Architecture	Pascal	Volta	Ampere	Ampere
CUDA cores	3840	5120	5120	6912
Clock frequency (GHz)	1.48	1.46	1.62	1.41
Memory (GB)	12	16	8	40
FP32 performance (TFlop/s)	11.36	14.03	16.59	19.5
CUDA version	10.2	11.3	11.4	12.0

Table 3-6: Architecture parameters of the GPU platforms used for measurements.

3.3.2 Numerical validation

The numerical correctness of the implementation was validated with a synthetic signal [175] and a real EEG dataset provided as a sample data file in the EEGLAB [184] software distribution. The GPU implementation was compared with a MATLAB implementation considered as the golden standard [185] . To quantify the accuracy of the decomposition results obtained with different implementations, I introduce the Similarity Index metric ρ given as

$$\rho_i\big(x_i(t), y_i(t)\big) = \frac{cov\big(x_i(t), y_i(t)\big)}{\sqrt{var\big(x_i(t)\big)}\sqrt{var\big(y_i(t)\big)}} \tag{3.1}$$

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.

The synthetic signal contains two frequency components and features intermittent noise. The length of the entire signal is 1000 samples, one component s_1 is a sinusoid signal with non-zero values from sample 500 to 750 with a frequency of 255 Hz. The other component s_2 is also a sinusoidal signal but spans the entire signal duration, from sample 0 to 1000 with a frequency of 65 Hz. The composite signal $s = s_1 + s_2$, expressed as follows:

$$s_1 = \begin{cases} 0 & if\ 1 \le n \le 500 \\ \sin\big(2\pi 0.255(n - 501)\big) & if\ 501 \le n \le 750 \\ 0 & if\ 751 \le n \le 1000 \end{cases} \tag{3.2}$$

$$s_2 = \sin\left(2\pi 0.065(n-1)\right)$$

The synthetic signal s and its two constituent components s_1 and s_2 are shown in Figure 3-4 (a). The interval from sample 501 to 750 in signal s is a period of intermittent noise, which makes the signal well suited for testing mode mixing.

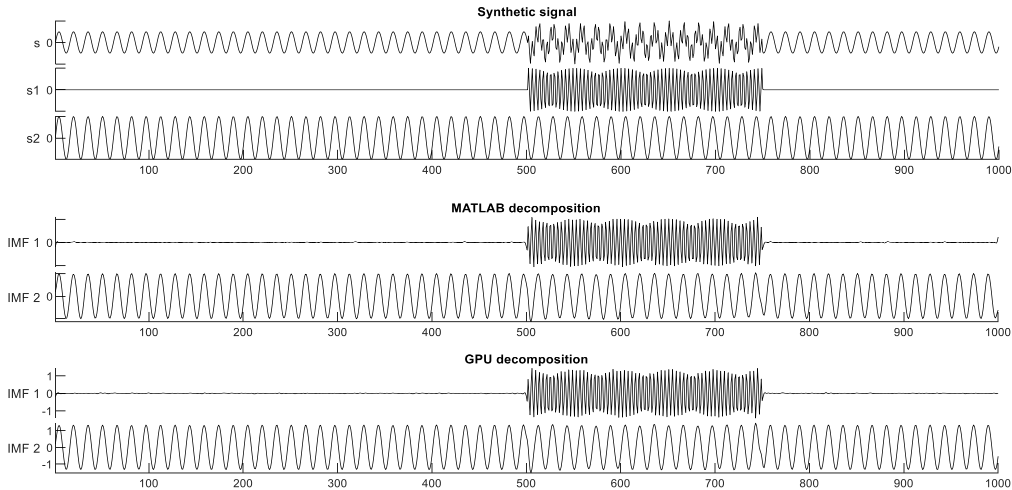

Figure 3-4: The synthetic dual-frequency signal (top), and the decomposition results from the MATLAB (middle) and CUDA (bottom) implementations.

Figure 3-4 (b) and (c) show the decomposition results of this dual-frequency synthetic signal

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 IMF 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 $\rho_{s2}^{CUDA} = 99.91\%$.

Next, I show the decomposition results obtained from a real EEG dataset. The selected signal is Channel 4 of the sample data file "eeglab_data.set" distributed with the EEGLAB Toolbox containing 30504 data samples (sampling frequency is 512 Hz, signal length: 1 minute). Figure 3-5 shows the extracted IMFs and the resulting Similarity Index values. Since there is no ground truth in this case, or in the case of any real EEG measurements, the Similarity Index is computed from the MATLAB and CUDA implementation results, treating the MATLAB result as the 'ground truth'.

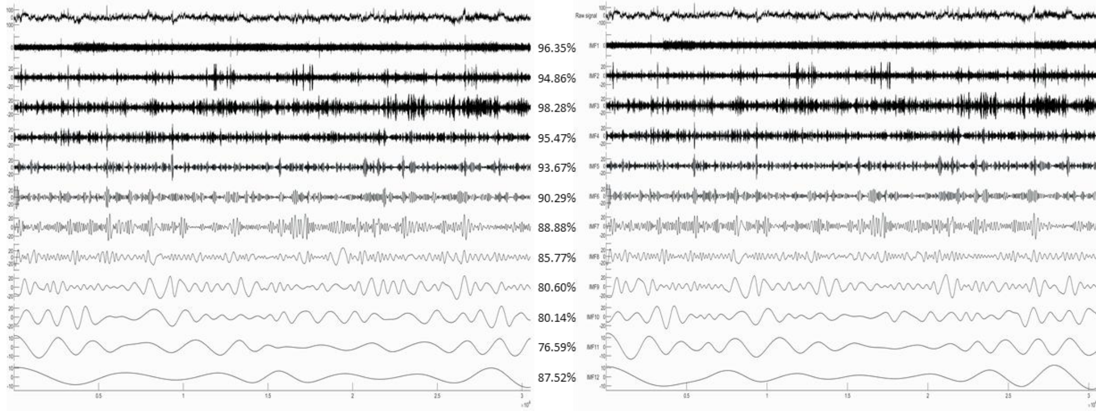

Figure 3-5: The decomposition results (only IMFs 1-12 are shown) of Channel 4 from the EEGLAB sample dataset produced by the MATLAB (a) and CUDA (b) implementations with the corresponding Similarity Index values.

Higher frequency IMFs show very good agreement of the two implementations. Lower frequency IMFs show somewhat reduced level of similarity, which is likely to be caused by the different random number generator in the two implementations, and different the boundary conditions during extrema detection and spline interpolation.

3.3.3 Computational performance and optimization

I start the performance results section by showing the execution times of the baseline MATLAB and libeemd Improved CEEMDAN implementations (Figure 3-6). Three input parameters (signal length N, number of iterations in the sifting process S, and number of realizations I) were varied during the tests. It should be noted that since the implementation provided by libeemd uses a completely different iteration stop criterion, a fixed number of iterations is used for a fair comparison. For the sample size N=102401 that represents 50 seconds of data at $f_s = 2048$ Hz or 6.6 minutes at $f_s = 256$ Hz, the MATLAB execution time varied between 6 and 53 minutes depending on the number of realizations (I=100, 200, …, 500) and sifting iterations (S=10, 20, 50). Execution times from the libeemd implementations for the same input parameters varied between 10 seconds and 4 minutes.

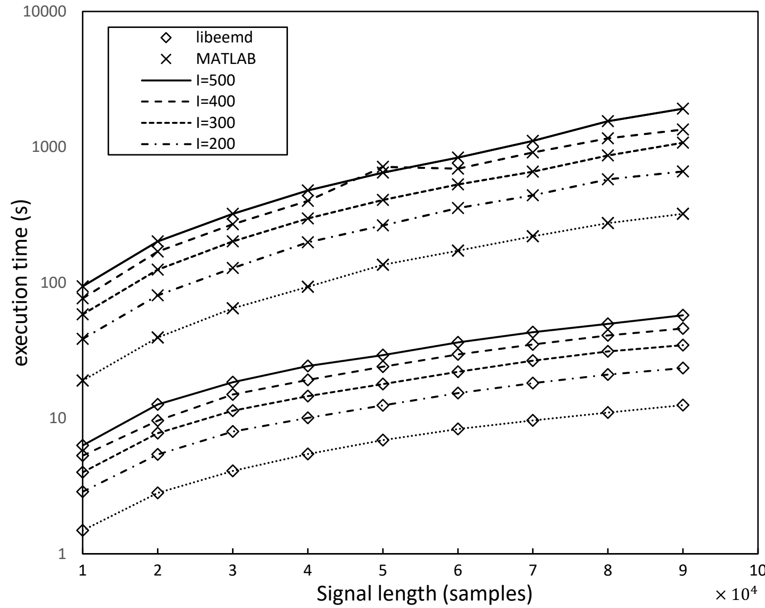

Figure 3-6: Execution time of the MATLAB and libeemd versions of the Improved CEEMDAN algorithm in function of signal length N and varying number of realizations I. The number of sifting operations is fixed, $S = 10$.

Before showing the execution time of the final GPU implementation, I illustrate an important performance optimization strategy. The execution profiling of the first implementation of the algorithm revealed that the cuSPARSE tridiagonal solver executes many small kernels, which – due to the large number of signal realizations – results in a significant performance overhead. Figure 3-7 shows the execution timeline of the tridiagonal solver on many realizations. It can be seen that the GPU is not fully utilized during the execution of kernels; there are idle time gaps between the kernels.

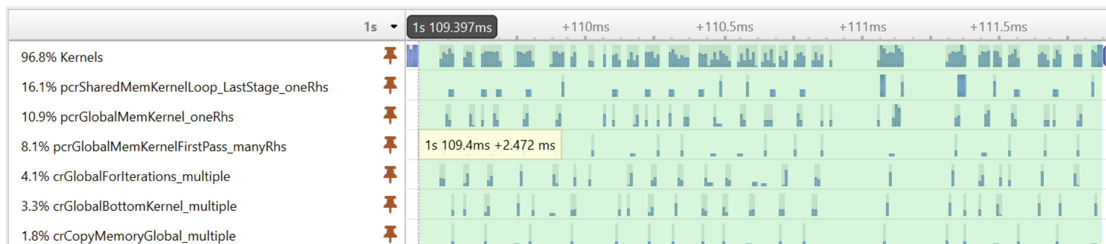

Figure 3-7: The execution timeline of the kernel functions used by the tridiagonal solver.

Fortunately, the CUDA programming model provides an elegant solution to the problem of launching many small kernels; the CUDA Graph execution model. With CUDA graphs, one can create a Directed Acyclic Graph from a set of kernels, and later the complete graph can be launched with a single launch call. Graphs can be created programmatically or captured at runtime during program execution. In the final version, the latter approach is used. The first execution of the graph is performed by launching the kernels individually to capture the graph. From the second execution, only the captured graph is launched. Figure 3-8 shows the result of the optimization achieved with CUDA graphs. The same tridiagonal solver is executed as before, but the kernels are now executed much more compactly, without large gaps reducing the execution time from 2.47ms to 0.65ms.

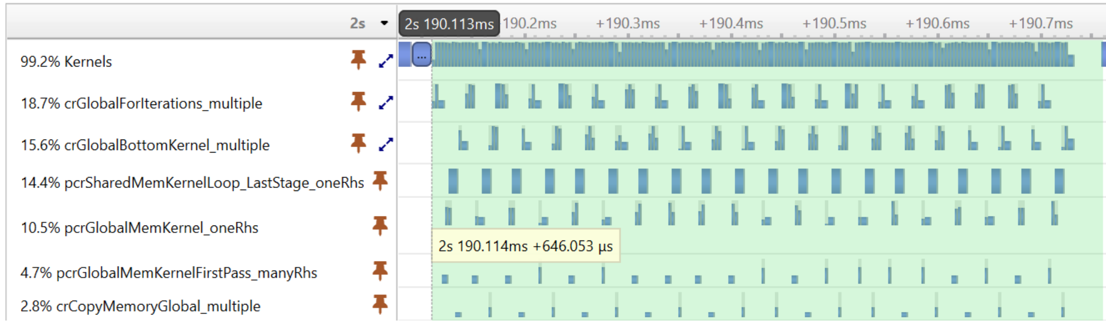

Figure 3-8: The execution timeline of the kernel functions of the tridiagonal solver using the CUDA graph optimization.

The execution time of the optimized GPU implementation is shown in Figure 3-9 . Runtime is in the range of 1-10 seconds for the N = 102401 sample size. I measured the execution time up to N = 358401 samples (representing about 3 minutes of measurements at f_s = 2048 Hz). From these values I calculated the speedup values compared to MATLAB, which is shown for different number of sifting iterations in Figure 3-10. It is important to note that the speedup increases with sample size and in a super linear fashion. That is, the more samples are processed, the faster the GPU version becomes compared to the MATLAB version. The exact speedup values for the full set of realization values are given in Table S2 in in the appendix part. The highest speedup was attained at N=102401, S=10 and I=500.

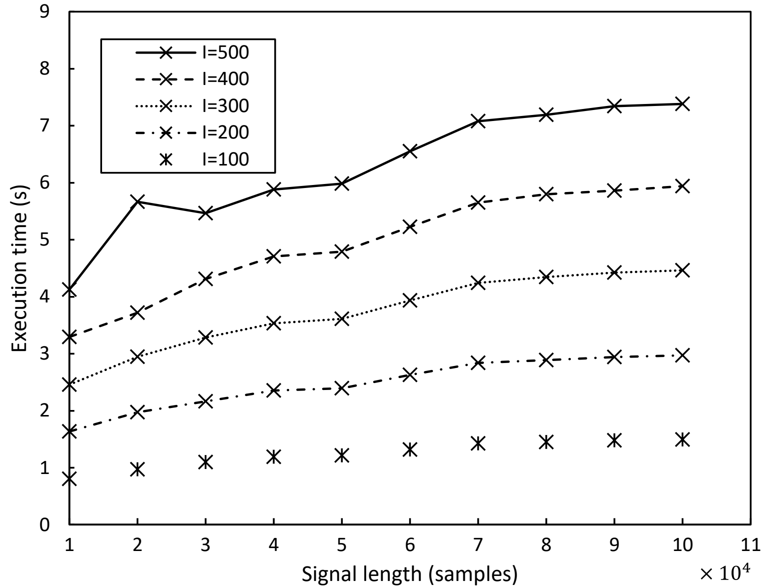

Figure 3-9: Execution time of the GPU ICEEMDAN algorithm (measured on a A100 GPU) in function of signal length N and varying number of realizations I. The number of sifting operations is fixed, S=10.

Next, I show the efficiency of the implementation by analysing the program execution time and profiling the arithmetic efficiency of kernel functions. Table 3-7 shows the relative weight of the GPU kernels during the execution of the ICEEMDAN algorithm in function of signal length on the RTX 3070 mobile card. The number of iterations in the decomposition process is fixed at 10, and the number of realizations is 500. Each column indicates the relative contribution of each kernel to the overall execution time. Green marks kernels provided by

NVIDIA libraries, while blue marks customized self-develop kernels. The last two rows of the table show the total contribution of the CUDA library functions and the customized kernels to the overall execution time.

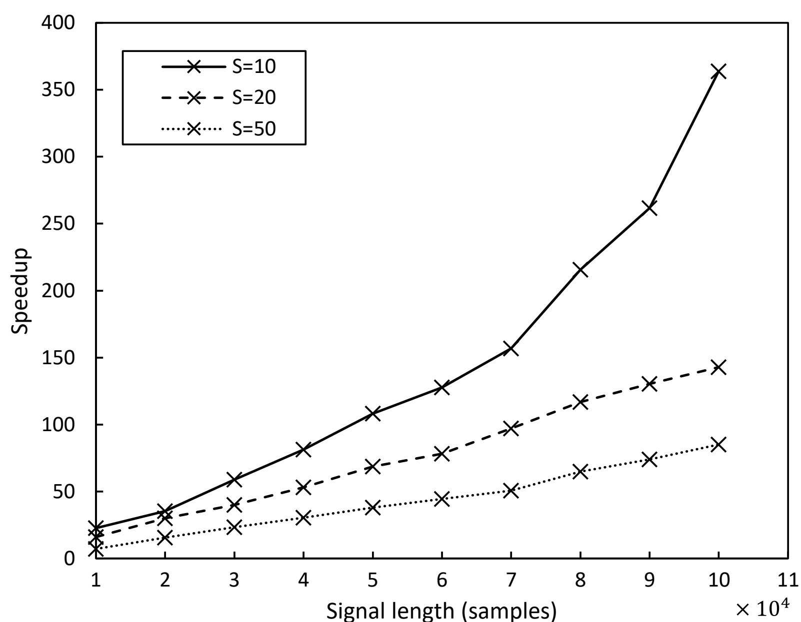

Figure 3-10: Speedup of the GPU implementation (executed on A100) over the MATLAB version in function of signal length N and varying number of sifting iterations S. The number of realizations is fixed, I=500.

	sample size						
kernels	**4097**	**8193**	**16385**	**32769**	**43009**	**79873**	**102401**
crGlobalForIterations_multiple	10.2%	9.2%	14.4%	17.8%	16.4%	17.1%	16.1%
crGlobalBottomKernel_multiple	8.6%	7.3%	13.1%	15.4%	13.8%	14.0%	13.3%
crCopyMemoryGlobal_multiple	4.5%	3.9%	3.5%	2.7%	2.5%	1.8%	1.7%
pcrGlobalMemKernel_oneRhs	17.7%	23.0%	13.5%	10.6%	14.2%	10.6%	9.6%
pcrSharedMemKernelLoop_LastStage_oneRhs	22.8%	19.8%	17.9%	14.0%	11.9%	8.9%	8.1%
pcrGlobalMemKernelFirstPass_manyRhs	7.8%	6.8%	5.9%	4.6%	4.1%	3.0%	2.7%
interpolate	0.8%	1.9%	2.7%	4.2%	5.2%	7.4%	8.9%
DeviceScanKernel	16.5%	15.0%	13.8%	12.0%	11.3%	11.3%	10.9%
preSetTridiagonalMatrix	0.6%	1.1%	2.0%	3.1%	3.6%	5.0%	5.8%
tridiagonal_setup	0.4%	0.6%	0.7%	0.9%	1.1%	1.5%	1.7%
select_extrema_min	0.3%	0.7%	0.9%	1.2%	1.5%	2.1%	2.3%
select_extrema_max	0.3%	0.7%	0.9%	1.2%	1.5%	2.0%	2.3%
spline_coefficients	0.3%	0.5%	0.6%	0.7%	0.9%	1.2%	1.3%
mean	0.2%	0.3%	0.6%	1.0%	1.1%	1.5%	1.7%
averageUpperLower	0.2%	0.5%	0.9%	1.3%	1.4%	1.9%	2.2%
updateRealizations	0.3%	0.5%	0.7%	1.2%	1.4%	1.9%	2.2%
produceSX	0.2%	0.5%	0.8%	1.3%	1.4%	2.0%	2.2%
DeviceScanInitKernel	7.5%	6.4%	5.5%	4.4%	3.9%	2.9%	2.6%
find_extrema_shfl_max	0.1%	0.3%	0.5%	0.7%	0.8%	1.1%	1.3%
find_extrema_shfl_min	0.2%	0.3%	0.5%	0.7%	0.8%	1.1%	1.3%
thresholdJudge	0.1%	0.3%	0.6%	0.9%	1.0%	1.3%	1.5%
tridiagonal solver kernels	71.6%	70.0%	68.3%	65.1%	62.9%	55.4%	51.5%
custom kernels	28.0%	29.6%	31.7%	34.8%	36.9%	44.2%	48.2%

Table 3-7: The relative contribution of individual kernels to the overall execution time in function of signal length.

The profiling shows a trend that with increasing signal length, the customized self-develop

kernels accounts for an increasing proportion of the overall execution time with the kernel function `interpolate()` becoming the dominating factor. The NVIDIA library functions are limiting performance for smaller input data sizes (72% vs 28%) but as the data size increases, their effect becomes smaller (52% vs 48%).

I also performed a Roofline performance analysis [186] of the performance critical kernels of the implementation at two different signal lengths (4k, 100k). The Roofline Model is a visual performance model used to analyze the efficiency of computational workloads on modern computer architectures. It helps identify whether a given computation is compute-bound (limited by floating-point operations per second, Flop/s) or memory-bound (limited by memory bandwidth). The model consists of a performance roof (attainable Flop/s), a memory-bound region, a compute-bound region, and arithmetic intensity (AI), which indicates where a computation falls on the spectrum of memory versus compute limitations. The performance P of a given workload is determined by:

$$P = \min\left(P_{max}, \frac{I}{B}\right) \tag{3.3}$$

Where P_{max} is the peak computational performance (in Flop/s), I is the AI (in Flop/byte), and B is the memory bandwidth (in bytes/s). AI is defined as:

$$I = \frac{Flop}{Bytes\ Transferred} \tag{3.4}$$

Where $Flop$ is the number of floating-point operations performed, and Bytes Transferred represents the amount of data moved between memory and the processor. Low AI values (e.g., <1 Flop/byte) indicate memory-bound workloads, whereas high AI values (e.g., >10 Flop/byte) suggest compute-bound workloads.

As can be seen in Figure 3-11, all kernel functions are memory-bound based on their AI values, 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 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 suggest that my implementation becomes more efficient as signal size increases.

In order to explore the effect of GPU hardware architecture on the execution performance of the implementation, the execution times on three different GPUs (see Table 3-6 for details) are measured and compared. Figure 3-12 shows the execution time values obtained on the different GPUs (S = 100, I = 200). The effect of hardware evolution and introduction of new architectural features is evident. The best results were obtained with an Ampere GPU, followed by Volta. The oldest architecture, Pascal (Titan Xp), produced the longest execution times.

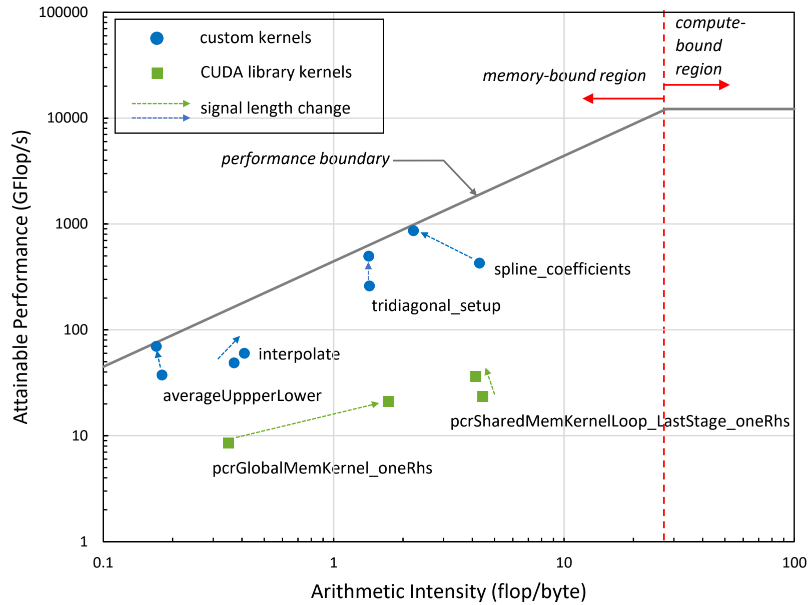

Figure 3-11: The Roofline performance results of the kernels executed on the RTX 3070 mobile GPU showing the performance positions of the main kernels of the implementation. Arrows indicate performance change as signal length is increased from 4k to 100k samples.

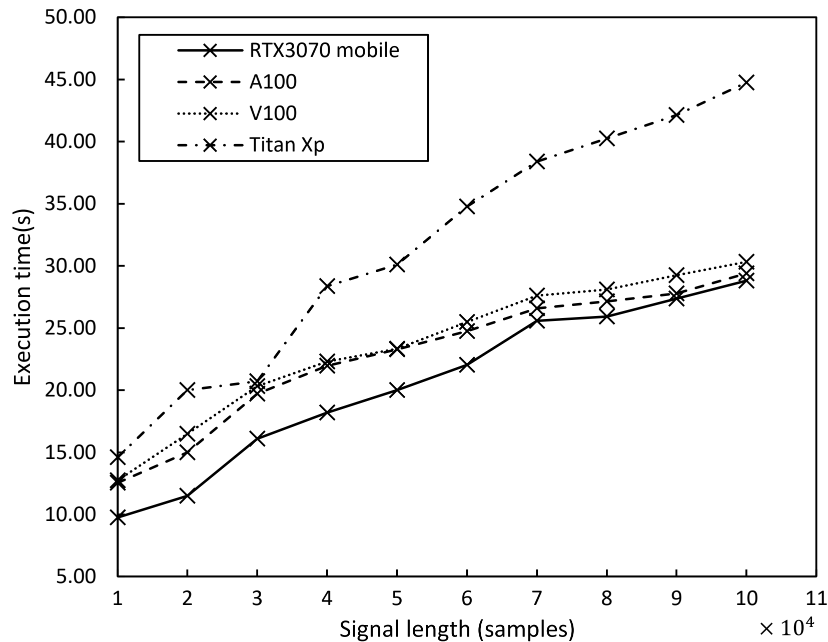

Figure 3-12: The execution time of the GPU algorithm on three different GPU architectures. The results demonstrate that each newer architecture generation (Pascal -> Volta -> Ampere) provides increased performance for the same program.

3.4 Summary

In this chapter, I described a massively parallel GPU implementation of the Improved CEEMDAN algorithm. The ICEEMDAN method is a crucial tool for the precise time–frequency analysis of nonstationary EEG signals. It can be used in various stages of EEG

processing, from preprocessing through time–frequency to connectivity analysis, and to calculate instantaneous frequency, power, and phase information in a very short amount of time, enabling researchers to uncover the dynamic properties of brain processes underlying perception and task execution.

Despite some known limitations, to my knowledge, this is the first GPU implementation of the Improved CEEMDAN algorithm. Here, I present evidence of the efficiency of my implementation reaching potentially a four-orders-of-magnitude increase in computing speed over the most frequently used MATLAB implementation. The source code of the implementation is publicly available under the MIT License at the Github page https://github.com/EEGLab-Pannon/CEEMDAN-GPU (accessed on 20 October 2023) of our group. The released implementation allows users and researchers to perform the decomposition of nonstationary natural signals into oscillatory components almost instantly, opening up new opportunities in research and in applications.

4 GPU IMPLEMENTATION OF MEMD

In this chapter, I present an efficient GPU implementation of the Multivariate Empirical Mode Decomposition method 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 synchronization. The implementation is validated with synthetic and real EEG signals of different lengths and channels on different GPU cards, and compared to existing serial MEMD implementations.

4.1 Materials and Methods

4.1.1 Related works

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 EMD, EEMD or CEEMDAN are applied on such datasets in a channel-by-channel fashion, as shown in Figure 4-1, 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. In EEG signal processing, the channels are not independent, their connectivity characteristics are essential for describing brain functions and characterizing physiological states. Extracting oscillations of the same frequency from different channels plays a vital role in multi-channel joint analyses, such as brain network construction and EEG source localization. While EMD is well-suited for non-stationary signal analysis due to its data-driven decomposition advantage, the mode alignment problem poses challenges when analysing multi-channel signals, such as EEG signals.

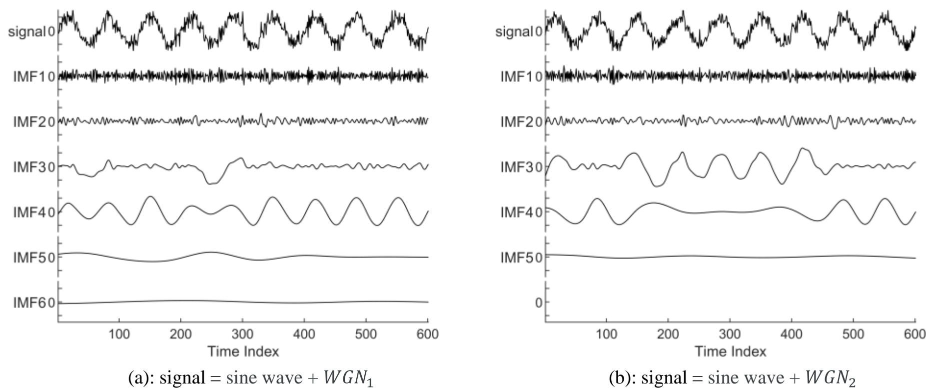

Figure 4-1: Signals with similar statistics often produce IMFs that are different in number (6 IMFs from (a) and 5 IMFs from (b)) and frequency (note the mode-mixing in (b)).

The Multivariate Empirical Mode Decomposition (MEMD) proposed by Rehman *et al*. [152] 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, the extreme points and mean envelope are still need to be calculated to 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. Figure 4-2 shows a simple example of the multivariate signal extreme point detection, in which the multivariate signal is projected 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.

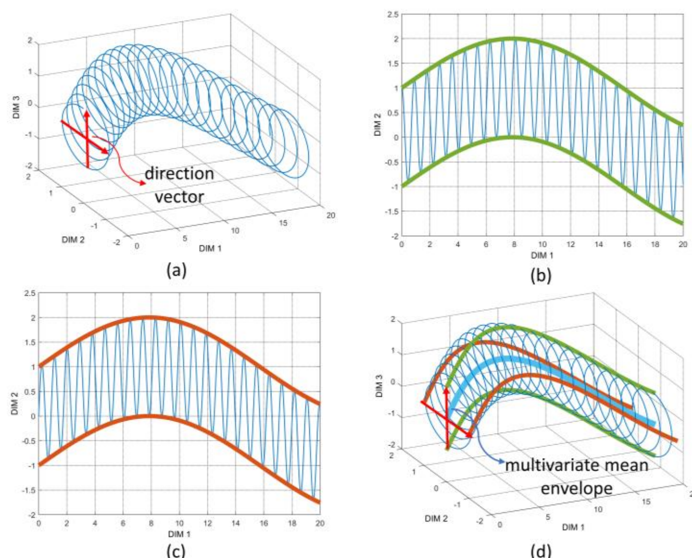

Figure 4-2: The principle of extreme point detection in a multivariate signal. (a) Two direction vectors used for projecting the multivariate signal. (b, c) The projected signals and their upper and lower envelopes produced along the two direction vectors. (d) The mean multivariate envelope (blue line) produced by averaging all multivariate upper and lower envelopes.

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 [141]. The direction vectors should also be uniformly distributed on a hypersphere. As shown in Figure 4-3, the Hammersley sequence, a low-discrepancy sequence that provides uniform angle sampling—also used by Rehman *et al*. [152]—can provide the proper basis for generating direction vectors with the desired properties.

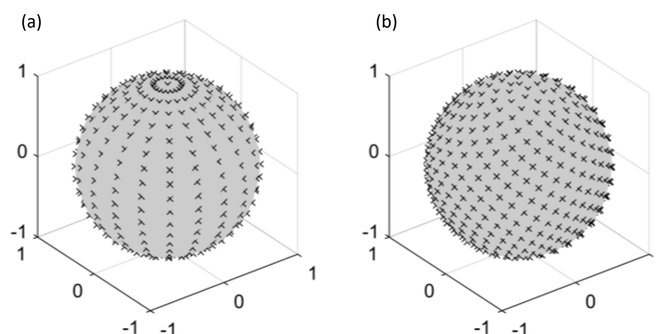

Figure 4-3: Direction vectors for taking projections of trivariate signals on a two-sphere generated by using (a) spherical coordinate system and (b) a low-discrepancy Hammersley sequence.

The details of the MEMD algorithm are outlined in Table 4-1. First, the multivariate input

signal will be projected onto the direction vectors and the projected signal on each direction vector will be produced. 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, the multivariate mean envelope of the input signal is generated. Finally, by subtracting the mean envelope from the input, a candidate IMF can be obtained. 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 4-1: Pseudo-code of MEMD algorithm.

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 [175] . To help the work of the EEG signal analysis community, Al-Subari et al. developed the EMDLAB toolbox [187] as an extension plug-in for the EEGLAB MATLAB framework [188] 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) is not acceptable for routine, production use. libeemd [176] 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 advancement of GPU technology in High-Performance Computing has led to the development of various parallel GPU-accelerated EMD implementations. Waskito *et al.* introduced the first single-precision CUDA EMD implementation for audio signal processing, achieving speedups of 29x and 29.9x compared to sequential C versions on NVIDIA Tesla C1060 and C2050 cards, respectively [177], [178] . Similarly, Xie *et al.* developed a CUDA-based EMD implementation for seismic data processing, delivering a 4x speedup on a GT240 GPU card [179] . Huang *et al.* further reported a 33.7x speedup on a C2050 GPU by utilizing an overlapped piecewise cubic spline interpolation technique [180].

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.

4.1.2 Design of the parallel implementation

In the 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 behavior, so by designing the kernel function, independent tasks can be arranged to different threads. Since the GPU has a large number of computing cores, threads can be executed in parallel. When the kernel function is launched 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 S3 in the appendix part.

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 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 x16 bus can only provide 32GB/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 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 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 S4 in the appendix part lists the memory requirement of each variable and their use in the GPU kernels.

The GPU implementation of MEMD primarily consists of four components: pre-processing, signal projection, extrema detection, and cubic spline interpolation. The main task of the preprocessing stage is to generate direction vectors for projection based on the Hammersley

sequence. Unlike MATLAB's on-the-fly generation strategy, I generate all direction vectors at once and allocate fixed memory for them, thereby eliminating the need for repeated global memory access during computation, which significantly reduces memory access overhead. Signal projection is essentially a matrix multiplication operation; therefore, I employed the highly optimized cuBLAS library functions to ensure performance. For extrema detection, I introduced an innovative use of the warp overlap mechanism to handle boundary extrema detection, avoiding the traditional sliding window approach and significantly improving parallel efficiency for longer signals. Cubic spline interpolation involves solving tridiagonal systems. To handle the interpolation of multivariate extrema across different dimensions, I introduced a novel approach using a multi-right-hand tridiagonal matrix solver, which enables parallel interpolation across dimensions.

4.2 Details of the Implementation

4.2.1 Pre-processing

In the pre-processing stage, I perform all GPU memory allocations and initialisations, and generate the direction vectors used later during the signal projection step. As shown in Figure 4-2 , 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 are used, the more uniform the distribution of the vectors can be provided, if a suitable sampling algorithm is used. The Hammersley sequence – a low-discrepancy sequence that provides uniform angle sampling – was used by Rehman *et al.* [152] as the basis for generating direction vectors. I also adopted this method in my 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 [189] 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 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.

4.2.2 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 Figure 4-4, the projection signal matrix **P** (SignalLength × NumDirVector) can be obtained after the input signal matrix **S** (SignalLength × SignalDim) and the direction vector matrix **D** (SignalDim × NumDirVector) are multiplied.

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.

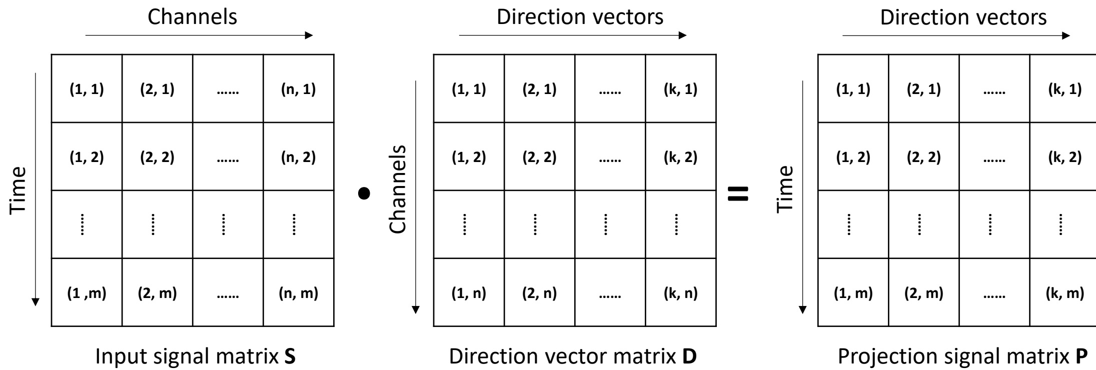

Figure 4-4: Computing the projected signals by matrix-matrix multiplication.

4.2.3 Extrema detection

The way to detect extrema is to compare a signal value with the values of its two adjacent neighbors 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 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, three load instructions are still needed per thread, and performance degrading shared memory bank conflicts may also occur.

In my parallel implementation, I 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 an intrinsic hardware accelerated warp-level instruction that provides direct access to register variables of the other threads within a warp. (A 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 Figure 4-5 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, the positions and values of all extreme points in the projection signal are obtained. These positions are the actual sample index values stored in a sparse vector. To make this vector suitable for cubic spline interpolation, the position vector needs to be compacted to a dense continuous sequence of location indexes.

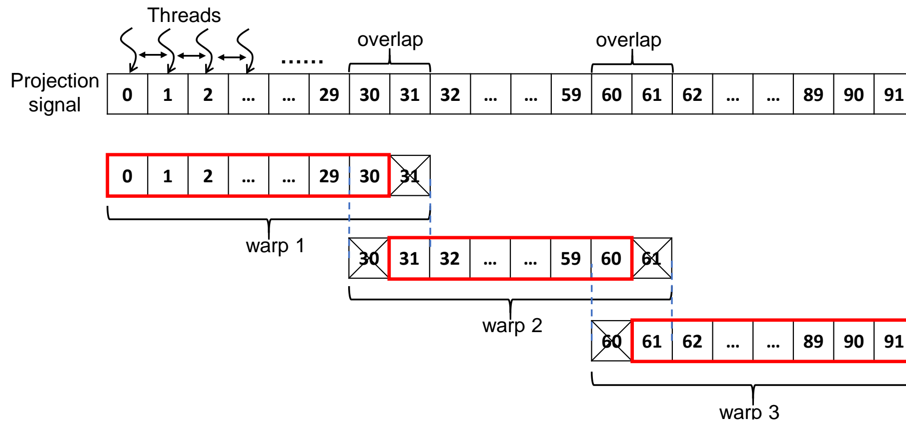

Figure 4-5: The principle of extreme point detection using warp shuffle operations. Red boxes mark threads performing the extrema detection steps.

The vector compaction step is illustrated in Figure 4-6. During the extreme point detection step, an auxiliary flag vector is generated 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.

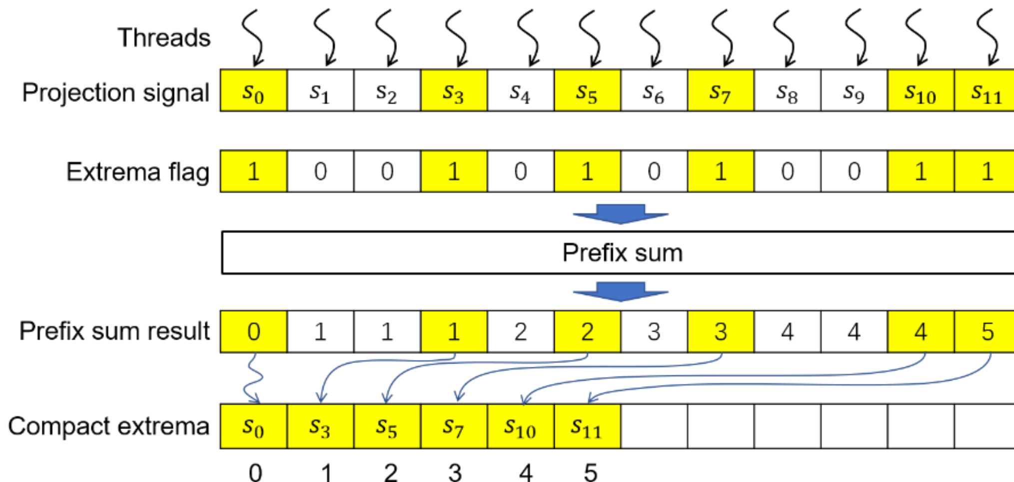

Figure 4-6: The principle of generating compact extreme point vectors using the prefix sum operation.

Since the first and last points of the signal do not have left or right neighbors, 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.

4.2.4 Cubic spline interpolation

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

$$S_i(x) = a_i + b_i x + c_i x^2 + d_i x^3 \tag{4.1}$$

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 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, $2n$ equations are obtained. 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 there are $4n-2$ equations, and there are two equations away from solving the unknowns. The last two missing equations are given by the boundary conditions. In the proposed implementation, a natural spline is chosen 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:

$$\begin{bmatrix} 1 & 0 & 0 & & \cdots & 0 \\ h_0 & 2(h_0+h_1) & h_1 & 0 & & \\ 0 & h_1 & 2(h_1+h_2) & h_2 & 0 & \\ 0 & 0 & h_2 & 2(h_2+h_3) & h_3 & \vdots \\ \vdots & & & & & \\ & & 0 & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \\ 0 & \cdots & & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_n \end{bmatrix} = 6 * \begin{bmatrix} 0 \\ \frac{y_2 - y_1}{h_1} - \frac{y_1 - y_0}{h_0} \\ \frac{y_3 - y_2}{h_2} - \frac{y_2 - y_1}{h_1} \\ \frac{y_4 - y_3}{h_3} - \frac{y_3 - y_2}{h_2} \\ \vdots \\ \frac{y_n - y_{n-1}}{h_{n-1}} - \frac{y_{n-1} - y_{n-2}}{h_{n-2}} \\ 0 \end{bmatrix} \tag{4.2}$$

After the m is solved, the coefficients of each spline function can be expressed as:

$$a_i = y_i \tag{4.3}$$

$$b_i = \frac{y_i - y_i}{h_i} - \frac{h_i}{2} m_i - \frac{h_i}{6}(m_{i+1} - m_i) \tag{4.4}$$

$$c_i = \frac{m_i}{2} \tag{4.5}$$

$$d_i = \frac{m_{i+1} - m_i}{6h_i} \tag{4.6}$$

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 back-projection with the location of the extreme points in the projected signal. Figure 4-7 shows a simple example for explanation. There are three input channels and and certain number of projected signals. Four extreme point locations are detected in one of the projected signals, $x0, x1, x2, x3$, which are projected back to each input signal to obtain signal values $y0 - y11$.

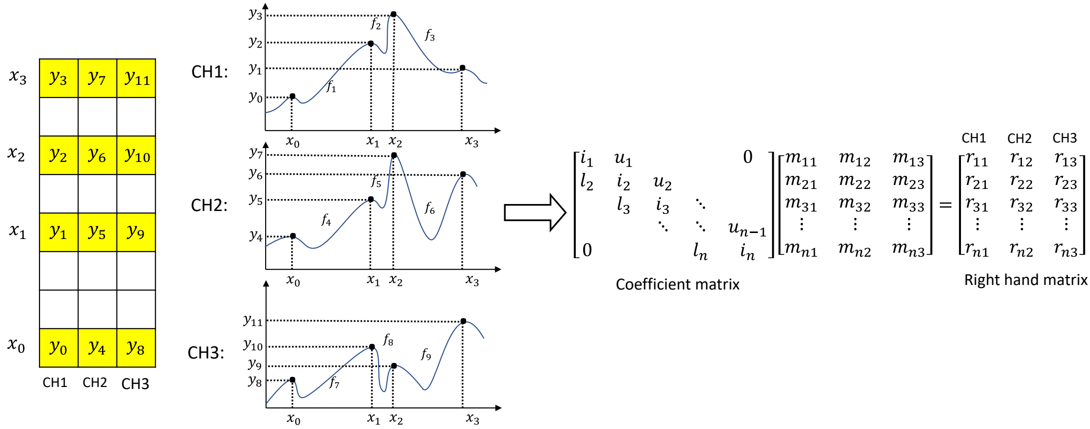

Figure 4-7: Batch Multi-right-handed tridiagonal matrix equation generated by multivariate extreme points.

Each multivariate extreme point is composed of three values obtained for the given 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 hands side vectors represent the different channels. For this, the multi-right-hand tridiagonal system solver function cusparseSgtsv2 of the cuSparse CUDA library is used. This function uses the Parallel Cyclic Reduction [190] 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. My 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, a binary search is used 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, the mean multivariate envelope is calculated 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, these mean multivariate envelopes need to be averaged 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.

4.3 Results

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

4.3.1 Test hardware

Different types of NVIDIA GPU cards were used in the testing process to explore performance variation across architecture families and card models. Different NVIDIA gaming and compute cards are selected, and their details are listed in Table 4-2 . The now outdated GTX 980 was used only to compare performance with the only known MEMD GPU implementation [141]. The Titan Xp and RTX 3070 were used both for development and testing. The V100 card (provided by GPULAB) 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, an i7-9700K CPU (8 cores, 3.60 GHz base frequency, 4.90 GHz turbo frequency) and MATLAB r2019a are used. During the tests, all 8 physical cores of the CPU were fully utilized to ensure maximum performance for the MATLAB implementation.

	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
FP32 (TFlop/s)	4.98	11.36	14.03	16.59
CUDA version	10.2	10.2	11.3	11.4

Table 4-2: Architecture parameters of the GPU platforms used for measurements.

4.3.2 Numerical validation

I used three multi-channel datasets to verify the numerical correctness of the GPU implementation of MEMD. The numerical verification was conducted not only to assess the consistency of the decomposition results across different implementations, but also to ensure that the number of IMFs generated across channels was consistent and that the oscillatory modes represented by corresponding IMFs were properly aligned.

4.3.2.1 Synthetic dataset 1

The first synthetic dataset used in the validation process was described in [152] (also

available online[1]) 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 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 [141], it is primarily used for performance comparison but for visual inspection, the result of the MEMD decomposition of this dataset obtained with the proposed GPU implementation is shown in Figure 4-8. The top row contains the individual input signals of the hexavariate dataset (Channels 1-6). The subsequent rows, from IMF 1 to 7, show the extracted oscillatory modes for each channel, starting with the noise (IMF1-3) then the signal components.

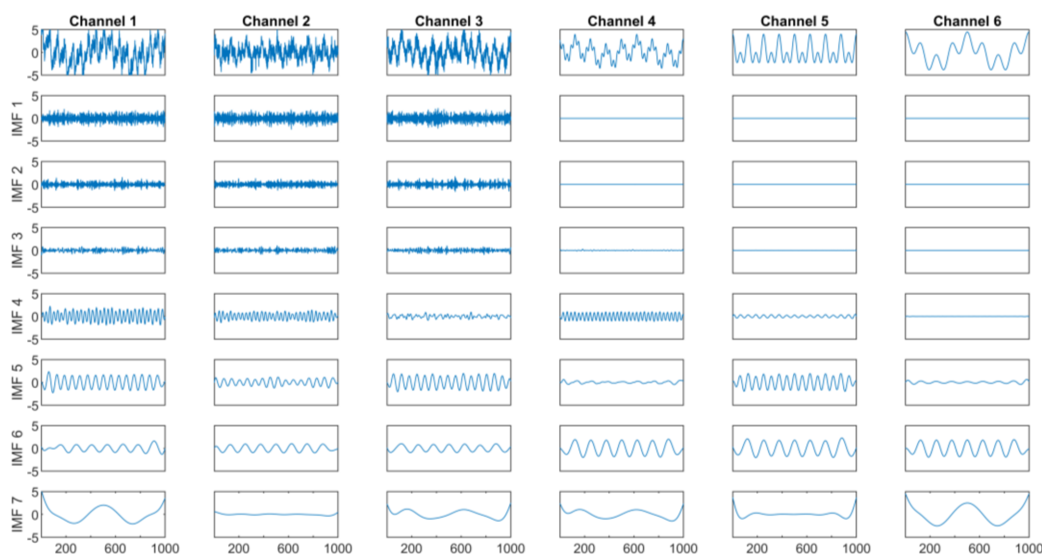

Figure 4-8: Result of the MEMD decomposition of the hexavariate dataset. Note that the oscillation modes with same frequencies from different channels are aligned in the same IMF.

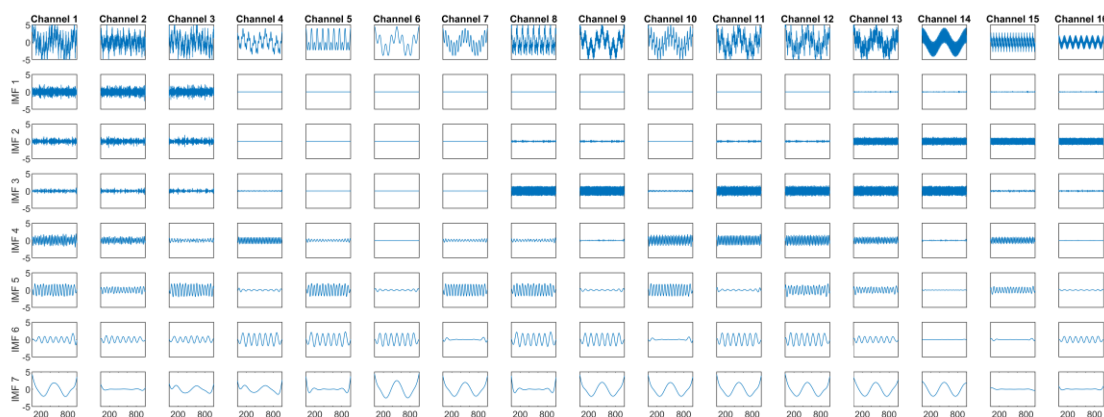

Figure 4-9: Result of the MEMD decomposition of the 16-channel dataset.

Figure 4-9 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. The proposed implementation correctly

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

identified the noise and sine wave components common in multiple channels in all three datasets and matched the results reported in [141].

4.3.2.2 Synthetic dataset 2

The second synthetic dataset was created for testing the numerical accuracy of the proposed GPU decomposition implementation. I 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.

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

$$\rho_i\big(x_i(t), IMF_i(t)\big) = \frac{cov\big(x_i(t), IMF_i(t)\big)}{\sqrt{var\big(x_i(t)\big)}\sqrt{var\big(IMF_i(t)\big)}} \tag{4.7}$$

where $cov()$ represents covariance of the input signal and the corresponding IMF, $var()$ 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. Figure 4-10 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.

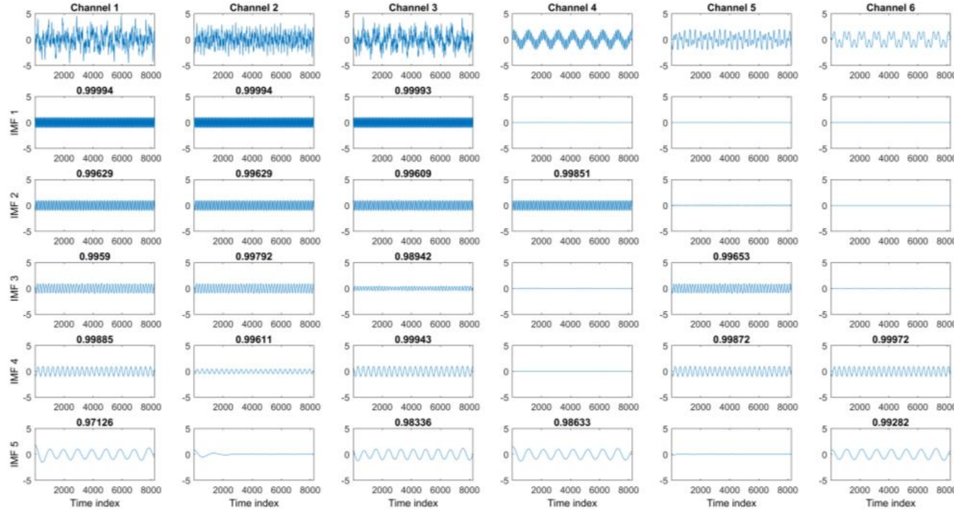

Figure 4-10: Decomposition results of the synthetic hexavariate signal with Similarity Index values shown in bold for each component.

4.3.2.3 EEG dataset

To compare the GPU decomposition results with the MEMD MATLAB implementation, I selected a real EEG dataset from the samples provided with the EEGLAB Toolbox [184]. The selected dataset consists of 32 channels each with 30504 samples (sampling frequency is 512 Hz). In , I show the result of the signal decomposition of Channel 4 of the dataset from t = 1 to 5 seconds. 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 [187]. The resulting similarity index values of the 32 channels for each IMF are plotted in Figure 4-12. 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.

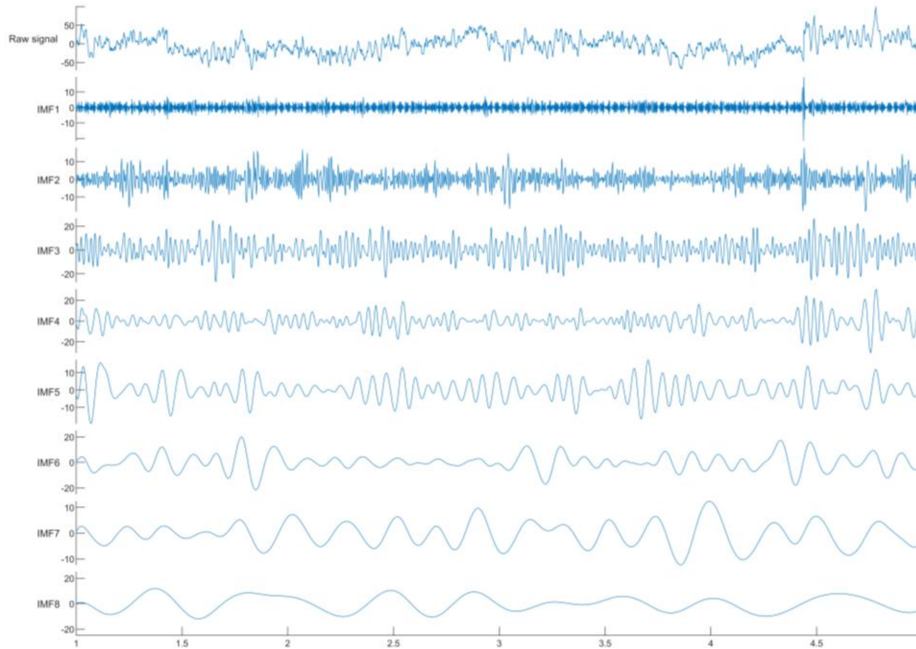

Figure 4-11: A section of the EEG signal (Channel 4) of the EEG sample dataset and the resulting IMFs (IMF1-8) of the GPU MEMD implementation.

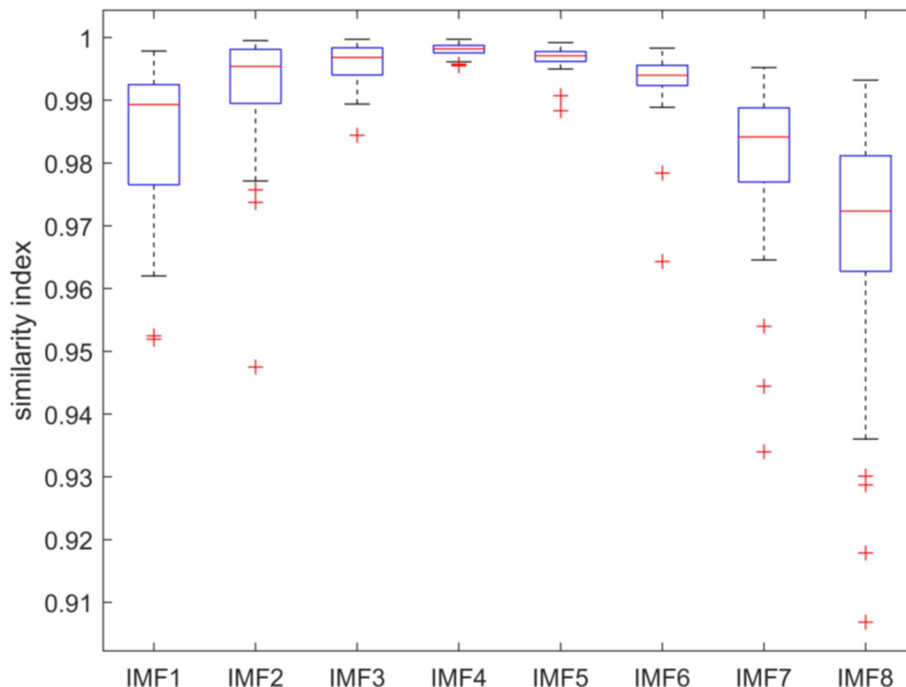

Figure 4-12: Boxplot of Similarity Index values representing the difference between the MATLAB and GPU implementations of the MEMD algorithm. Each IMF category represents the distribution of the Similarity Index values of all channels for that IMF. Red lines represent median values.

4.3.3 Performance results

In this section, I 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. All performance measurements were averaged from 10 repetitions after 5 warm-up runs.

I measured and compared the execution time of the GPU implementation on different GPU cards and compared it with MATLAB runtimes. Also, the speedup results indicating the speed advantage of the GPU implementation over the MATLAB one are provided. 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, I compared the performance of my implementation to the only MEMD GPU implementation found in the literature [141]. 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 the proposed implementation objectively, I had to execute the program on the very same type of card (GTX 980) that was used in [141]. The execution time obtained this way with the proposed version on two datasets (6 and 16 channels, signal length = 1000 samples) is shown in Figure 4-13. On average, the proposed GPU implementation achieves a 1.69x speedup in execution time.

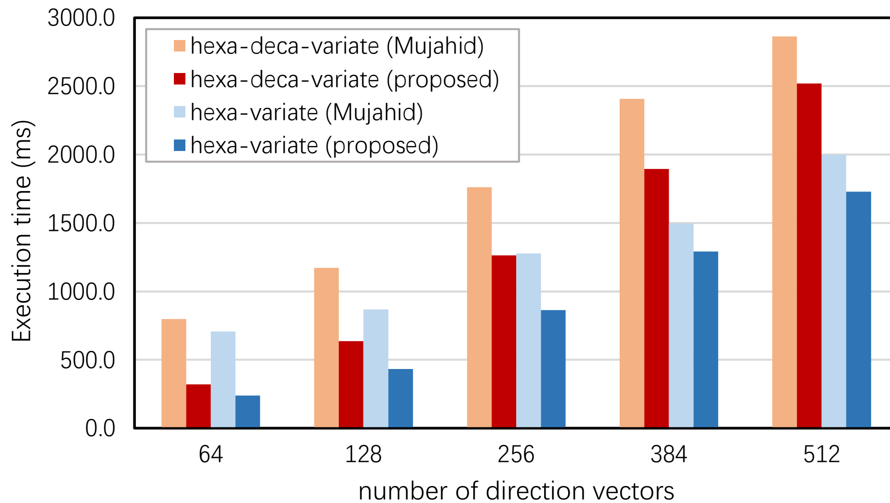

Figure 4-13: Execution time of the proposed GPU implementation compared to [141] on an NVIDIA GTX 980 GPU card. Two (a 6 and a 16-channel) datasets with 1000 samples per channel were used in both implementations using varying number of direction vectors.

Next, the MATLAB MEMD execution time was compared with the 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 4-2. Since the memory size was different on the different GPU cards, slightly different problem sizes have to be used on each system. Table 4-3 shows the maximum number of input samples each GPU can process for different channel count and projection vector size parameter combinations.

	RTX 3070 mobile			Titan XP			Tesla V100		
number of channels	# 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

Table 4-3: The maximum number of data samples per channel the proposed implementation can process on the different test GPU cards.

The execution time and speedup results of the different test cases (32, 64 and 128 EEG channels) are shown in the following figures (Figure 4-14-Figure 4-19). 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.

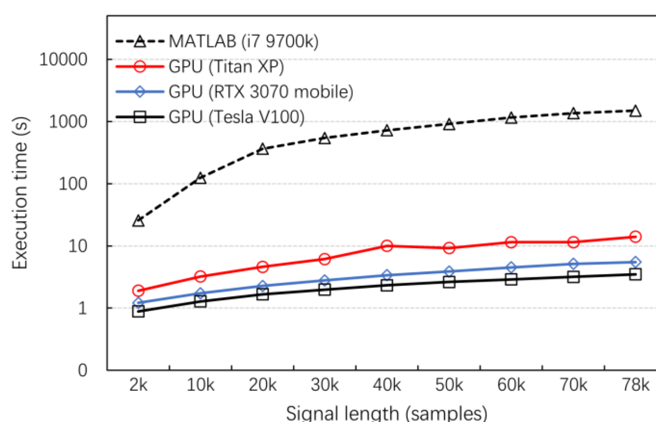

Figure 4-14: 32-channel 64 direction vectors, up to 78k samples, MATLAB vs GPU execution times.

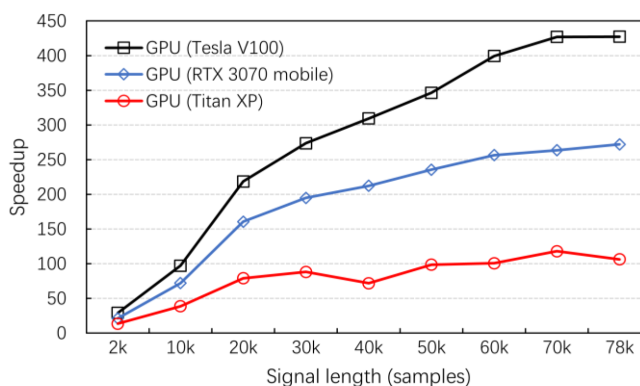

Figure 4-15: Speedup for 32-channel 64 direction vectors, up to 78k samples, compared to MATLAB.

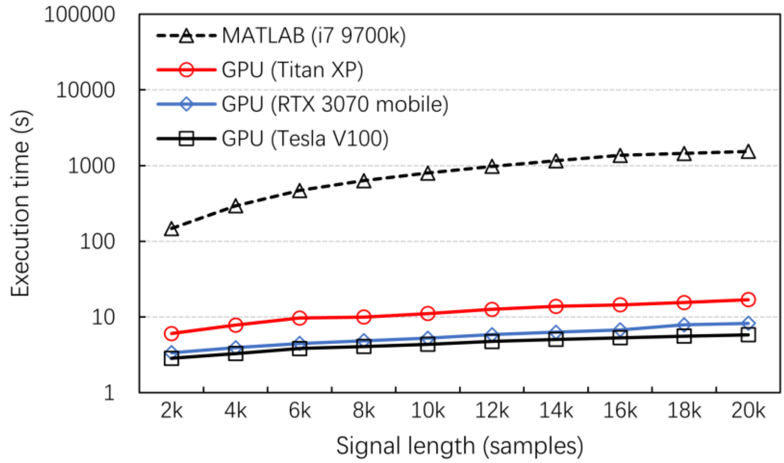

Figure 4-16: 64-channel 128 direction vectors, up to 20k samples, MATLAB and GPU execution times.

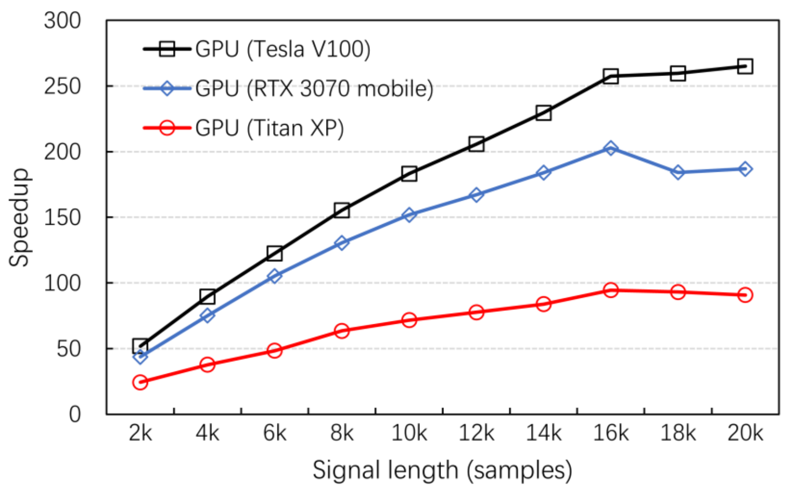

Figure 4-17: Speedup for 64-channel 128 direction vectors, up to 20k samples compared to MATLAB.

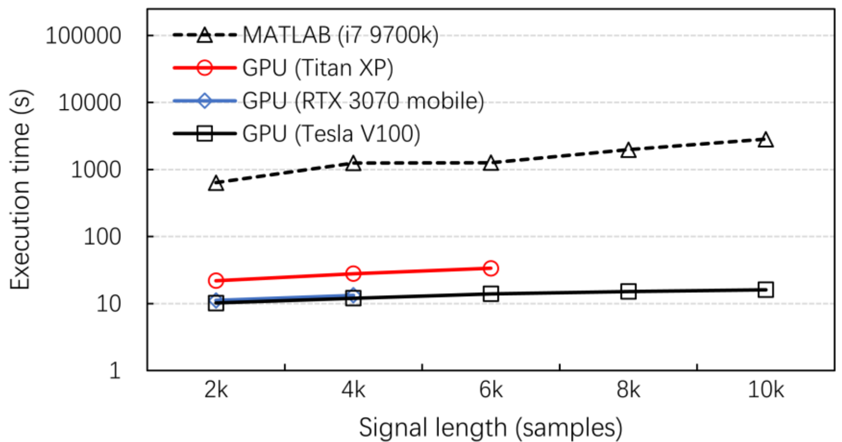

Figure 4-18: 128-channel 256 direction vectors, up to 10k samples, MATLAB and GPU execution times.

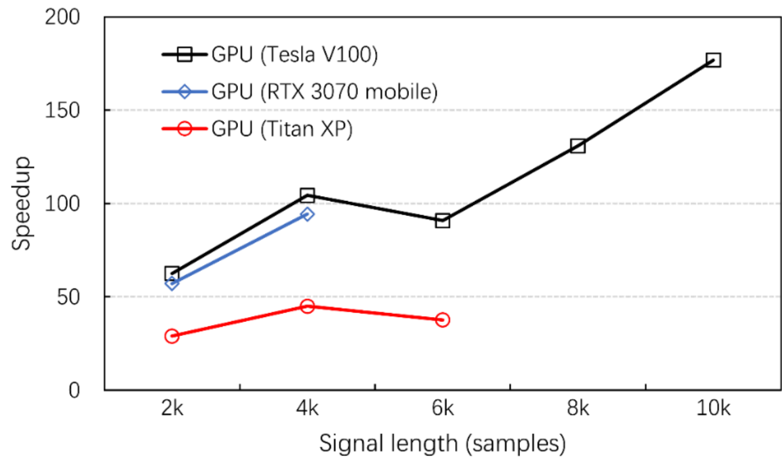

Figure 4-19: Speedup for 128-channel 256 direction vectors, up to 10k samples compared to MATLAB.

The execution time results show that for all channel number cases the GPU execution time was below or around 10 seconds 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.

4.3.4 Performance analysis

Here I provide a summary of the performance evaluation of the proposed MEMD implementation. I profiled the code to identify potential performance bottlenecks and see the relative weight of each GPU kernel during program execution. Table 4-4 shows the results I obtained with the 32-channel EEG dataset (64 direction vectors) on the V100 card while varying the input signal length.

	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 4-4: Relative contributions of the kernels to the overall execution time for different signal lengths.

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 customized self-develop kennels. To help visualise performance trends, I used green and blue colour bars to show the change of the relative contributions of the cuSparse and self-develop kennels, respectively. The last two rows of the table show the total relative contribution of the cuSparse library and the self-develop kernels.

Table 4-5 shows the profiling results obtained with a 128-channel EEG dataset (256 direction vectors), also executed on a V100 GPU. Both Table 4-4 and Table 4-5 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 the self-develop spline interpolation kernel 'interpolate'. At around 80k sample size, the time spent in the solver and in other kernels becomes equal, after which the self-develop 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.

	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%**

Table 4-5: Relative contributions of the kernels to the overall execution time. Channel count is 128, number of direction vectors are 256.

The GPU utilisation of the 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 GIop/s. The execution timeline of one iteration of the MEMD program is shown in Figure 4-20. 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.

I performed a Roofline [186] 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 is typically between 14 and 30, hence only kernels having an Arithmetic Intensity value of 14 or higher can achieve close to peak compute performance. The arithmetic intensity of the kernels in the proposed implementation are in the range of 0.1 to 2.2 and thus are limited by the memory bandwidth of the GPU cards. Figure 4-21 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. Unfortunately, the implementation is proprietary; hence, I did not have an opportunity to perform further performance optimisation on them.

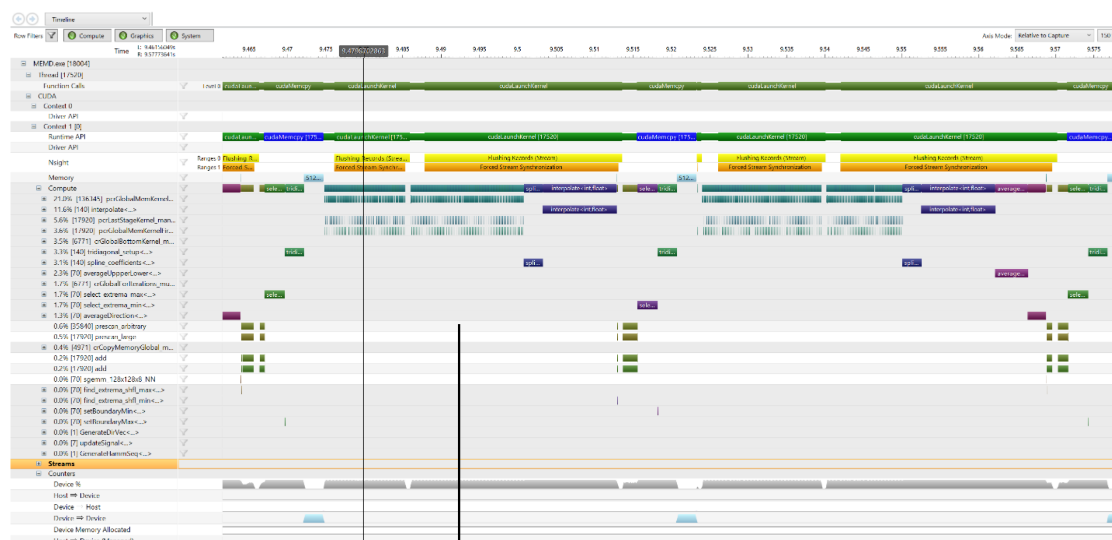

Figure 4-20: Execution timeline of one iteration of the IMF computation look of the MEMD algorithm.

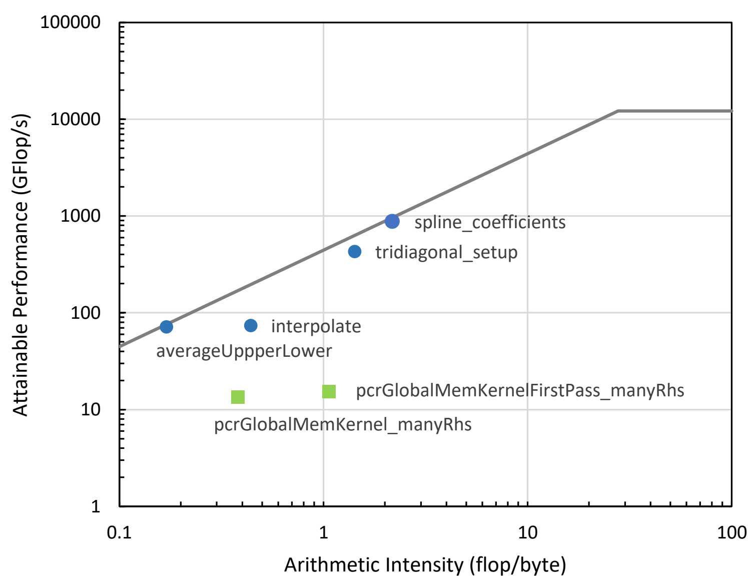

Figure 4-21: The Roofline performance model of the RTX 3070 mobile GPU showing the performance positions of the main kernels of the implementation.

4.4 Summary

In this chapter, I described an efficient GPU implementation of the MEDM algorithm implemented in CUDA. While my focus has been EEG signal decomposition and time-frequency analysis, MEMD and the proposed implementation can be used effectively in other multi-sensor natural data series analysis areas, such as earthquake and vibration monitoring, power optimization, image processing and so on.

The proposed 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 increases can be expected from the Ampere and Hopper architectures and from multi-GPU extensions that are among the future plans. The source code of the proposed implementation is available under the MIT Open-Source license for interested readers for use in applications or further improvements.

5 GPU IMPLEMENTATION OF ICA

In this chapter, I propose an Independent Component Analysis prototype implemented by tensor cores for speeding up the EEG spatial decomposition. ICA has become the de facto standard for artifact removal in EEG signal processing. However, its computational intensity, driven by the matrix multiply-add operations in the algorithm, brings challenges, particularly when working with large datasets. The proposed implementation uses the tensor cores in GPUs to efficiently perform matrix multiply-add operations, significantly accelerating the execution of ICA. Compared to the MATLAB version widely adopted in the EEG processing community, the proposed approach not only delivers significant speedup but also achieves improved GPU memory throughput.

5.1 Materials and Methods

5.1.1 Related works

Electroencephalography (EEG) allows us to measure the bioelectric activity of the brain using non-invasive scalp electrodes at millisecond time resolution. This property made it a fundamental tool in neuroscience research. There are several methods of acquiring EEG signals, such as the Electrocorticogram (ECoG) method, which acquires signals directly from the brain cortex, and the stereo-electroencephalography (SEEG) method, which acquires signals from the white matter of the brain, etc., and both of them are invasive methods that are only used in clinical surgery with very high cost. The most widely used method is the non-invasive scalp EEG, which is acquired by several electrodes placed on the scalp; however, due to the volume conductor effect produced by the scalp, skull, and cranial cavity, as shown in Figure 5-1, the signal acquired by each electrode $x(t)$ is actually a mixture of the true brain source signals $s(t)$. Meanwhile, the EEG signal has very small amplitude and is frequently contaminated with noise and unwanted artifacts, such as power line noise, eye movement or blinks, muscle noise or heart/pulse artifacts. These artifacts must be removed before further analysis to obtain reliable results.

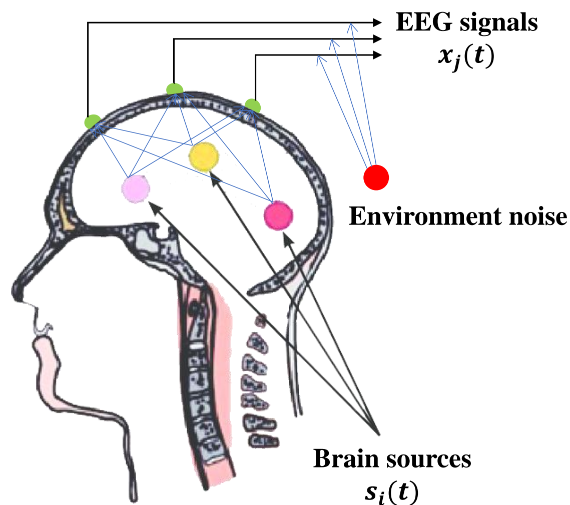

Figure 5-1: Acquired scalp EEG signals: a mixture of different true brain sources and noises.

The de facto standard method for artifact removal 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 [191].

One major disadvantage of the ICA algorithms is their high computational cost. Performing an ICA step for a high-density EEG recording can easily take an hour or more. For large studies, execution running for days is not uncommon. Parallel computing can help in reducing the execution time, and several ICA implementations have been developed for CPU [192], [193] and GPU systems [194], [195], [196], [197], [198]. Interestingly, only one GPU implementation is known for the Infomax ICA algorithm by Raimondo [198], developed more than a decade ago. Since then, GPU hardware and software technology have developed at an unprecedented pace, especially after the introduction of tensor core as a new computing hardware unit in the Nvidia's Volta architecture, the performance of GPU has achieved another leap.

As introduced in section 2.3, 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.

Precision	Peak performance (TFLOPS)
FP64	9.7
FP32	19.5
FP16	78
BF16	39
FP64 Tensor core	19.5
FP32 Tensor core	156
FP16 Tensor core	312
BF16 Tensor core	312

Table 5-1: Peak performance of NVIDIA A100 GPU for tensor cores and CUDA cores at different precision.

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. As shown in Table 5-1, for different precision types, the peak performance provided by the tensor core is much higher than that of the CUDA core, especially for the single precision commonly used in EEG signal processing, the performance of the tensor core is almost 8 times that of the CUDA core.

As the only known implementation of Infomax ICA on GPU, Raimondo *et al.* [198] tested their CUDAICA on NVIDIA's GTX560 and Tesla C2070, and built different CPU versions based on ATLAS (a portable self-optimizing BLAS) and Math Kernel Library (Intel MKL). Overall, their CUDAICA achieved a 4.5-20x speedup compared to the CPU implementation. However, they did not test the MATLAB implementation most commonly used by the EEG processing community, and it is clear that the introduction of new hardware features such as tensor cores can further accelerate the calculation of ICA on GPUs.

5.1.2 Design of the parallel implementation

In the parallel implementation, as shown in Figure 5-2 (a) and Table 2-2 in section 2.3, ICA is divided into four steps, each implemented by a kernel function executed on the GPU. The input data is divided into different data blocks, and each data block will be used to update the weight value in the unmixing matrix. After all the data blocks are fed, they will be re-permuted for the next feeding round. After several rounds of iterations, the final unmixing matrix used to separate the independent components in the input data can be produced.

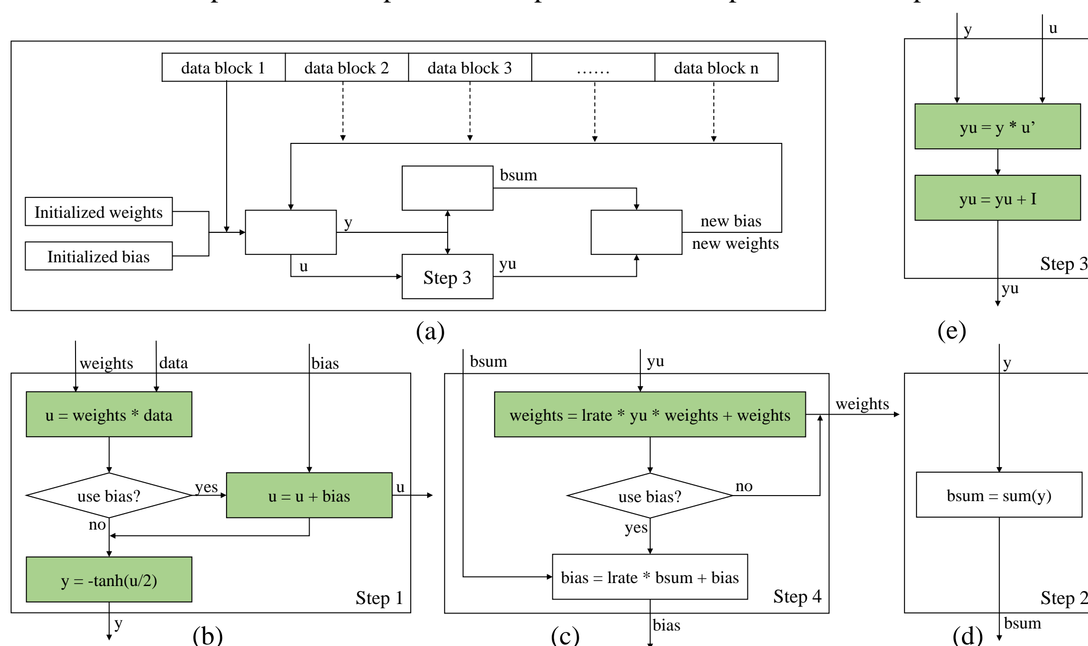

Figure 5-2: (a): The algorithm flow of ICA in one iteration; (b)-(e): the details of numerical operations in each step, and the highlighted parts involve matrix multiplication and addition.

The steps in ICA involve numerous numerical operations, with matrix multiplication and addition being key operations that could benefit from optimization through tensor cores. Specifically, any matrix operation of the form D = A * B + C should be a primary focus, as such fused multiply-add (FMA) operations can be executed efficiently by the tensor core. As illustrated in Figure 5-2 (b)-(e), the matrix multiplication and addition operations at each step

are highlighted for clarity.

Another important thing is the analysis of variables and memory allocation. The ICA algorithm involves numerous intermediate variables, which are passed between different steps. Some steps modify their input variables, while others do not, and the sizes of different variables are also different. Therefore, it is necessary to analyze the size, transfer, and modifications of each variable at every step when optimizing the ICA algorithm using tensor cores. As depicted in Figure 5-3, all variables are described by three dimensions: the number of channels (ch), number of samples (s), and block length (b). To avoid unnecessary overhead from memory swapping, all variables are allocated prior to iterative execution.

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.

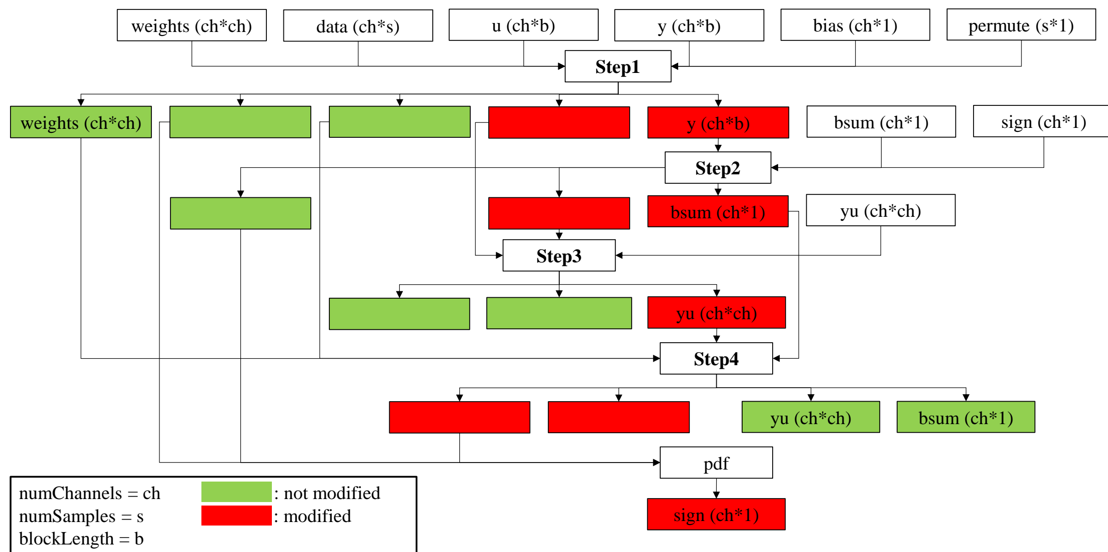

Figure 5-3: The variables flow in ICA implementation, including size, modifications, and pass path.

5.2 Details of the Implementation

5.2.1 Tensor core applications

The tensor core is similar to the CUDA core in that they can both be called in kernel functions to perform numerical computations, except that the CUDA core computes on a single value, while the tensor core computes on a series of matrices (multiply-add operations) as shown in Figure 5-4.

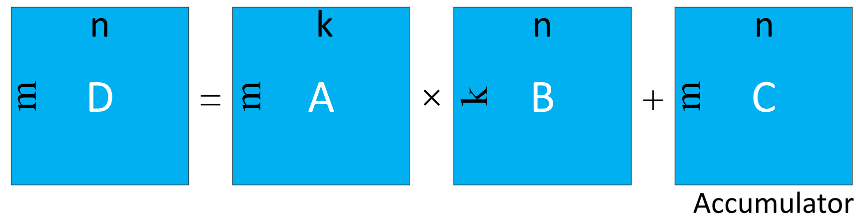

Figure 5-4: The processing object of tensor cores: matrix A, B, and an accumulator matrix.

CUDA library functions provide optimized support for tensor cores, enabling highly efficient matrix multiplications through APIs such as cuBLAS and cuDNN. These functions automatically leverage tensor cores when supported hardware and appropriate data types (such as FP16, BF16, or TF32) are used, significantly accelerating General Matrix Multiplication (GEMM) operations. However, relying solely on library functions like cuBLAS may not fully exploit tensor core capabilities in all scenarios, especially when operations involve custom data layouts, fusion with other computations, or fine-grained optimizations within kernels. In such cases, directly utilizing tensor cores in custom kernels remains crucial. NVIDIA is actively developing APIs that enable kernel functions to leverage tensor cores, such as cuBLASDx (still in beta version) that provides APIs that allow developers to use tensor cores explicitly within device code, enabling more flexible implementations of matrix multiplications, convolutions, or fused operations. By integrating tensor core intrinsic functions within kernel functions, developers can achieve better performance and efficiency for complex workloads beyond what prebuilt cuBLAS functions offer.

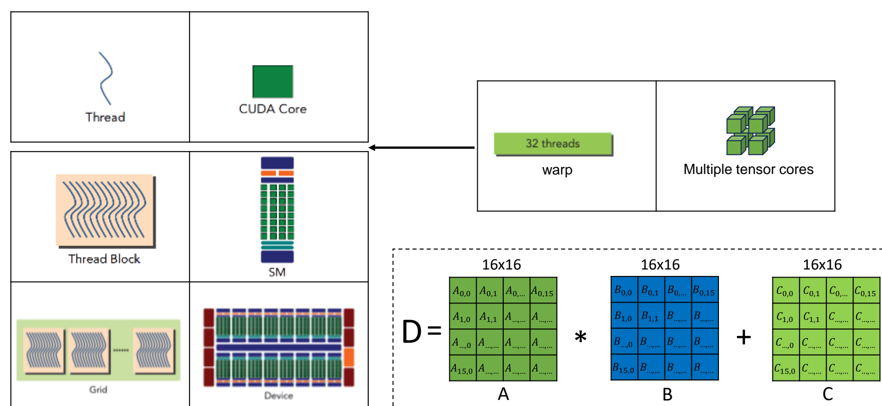

Figure 5-5: The hardware-software hierarchical correspondence of tensor cores, and the processing unit of a warp.

The programming model of the tensor core also differs from that of the CUDA core due to the different processing objects. As shown in Figure 5-5, the introduction of tensor cores causes a change in the hardware-software hierarchical correspondence in the original CUDA GPU programming: 32 threads will be treated as a warp, which will be processed by multiple tensor cores together, and this warp will provide a 16×16×16 matrix (single precision) processing array that performs the operation D = A * B + C, where A, B, C, and D are 16×16 matrices. Therefore, the operation of a single tensor core is opaque to users, and the interface between users and tensor cores is the warps consisting of 32 threads each, and 16×16 tile matrices (single precision). The instructions in the kernel function are executed by the CUDA

core by default, but for the tensor core, the Warp Matrix Multiply and Accumulate (WMMA) APIs need to be called in the kernel function to fragment, load, fill, compute, and store target matrices.

In fact, the computation of the tensor core is mixed-precision, i.e., the precision of the input and output matrices can be different. Therefore, for different precision, the matrix size that can be computed by a warp once is also different. The possible combinations are shown in Table 5-2. For the single-precision type most commonly used in EEG signal processing, a single warp can provide a 16×16×8 matrix (single precision) processing array.

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
_half	_half	float	16×16×16
_half	_half	float	32×8×16
_half	_half	float	8×32×16
tf323	tf32	float	16×16×8
double	double	double	8×8×4

Table 5-2: The combinations of different matrix sizes and precision types that a single warp can process.

Using tensor cores to perform the matrix multiply-accumulate operations in ICA is the key to optimizing ICA implementation. As highlighted in Figure 5-2, all FMA operations in ICA have been located, and they need to be further quantified into the smallest processing unit (tile) suitable for warps/tensor cores computation. As illustrated in Figure 5-6, a naive implementation is to start a two-dimensional thread grid based on block matrix multiplication, and each warp in the grid is responsible for a tile in the result matrix. One collective computation of the thread grid will complete the multiplication of the strips of matrix A and matrix B on their M and N dimensions (as shown in the green box in Figure 5-6), and the result matrix can be obtained after iterating on the K dimension.

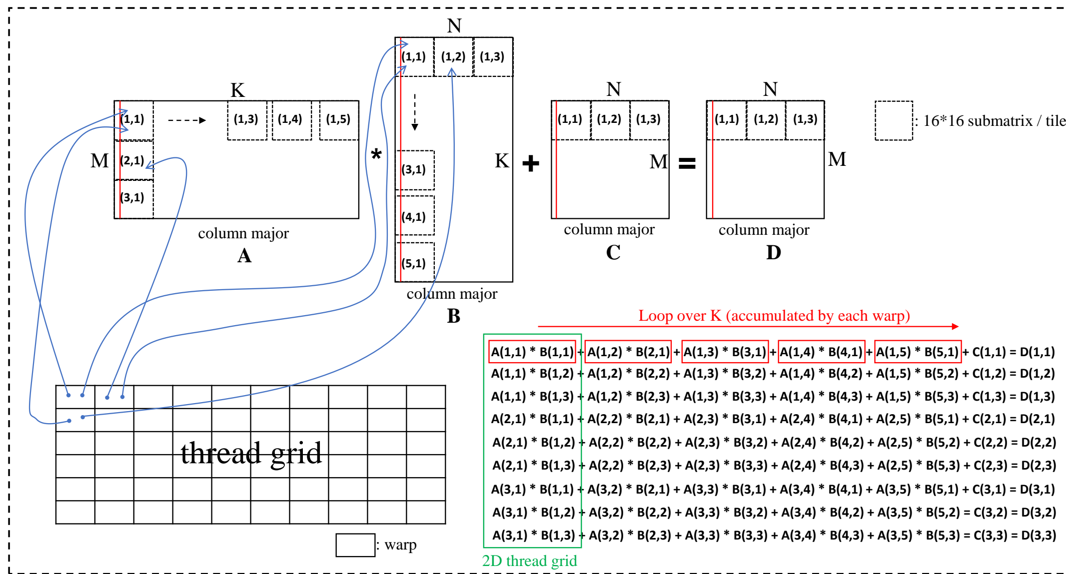

Figure 5-6: A naive strategy of matrix multiply-accumulate operation implemented by tensor cores. Each warp in the thread grid is responsible for a tile in the result matrix.

When calling tensor cores in a kernel function to perform matrix multiply-accumulate operations, there are several steps to follow: First, the matrices to be computed need to be fragmented into tiles, and the threads in the same warp need to know what kind of tiles they will process together, including the size of the tiles, the precision type, the row/column major, and the location (in matrix A, B, C or the accumulator). The implementation of the above functions in the kernel function requires the following statements:

```
wmma::fragment<wmma::matrix_a, m, n, k, half, wmma::column_major>a_frag;
wmma::fragment<wmma::matrix_b, m, n, k, half, wmma:: column_major>b_frag;
wmma::fragment<wmma::matrix_accumulator, m, n, k, float>acc_frag;
wmma::fragment<wmma::matrix_accumulator, m, n, k, float>c_frag;
```

Where a_frag, b_frag, c_frag, and acc_frag are essentially addresses pointing to the register file in the SM, each representing a tile-sized piece of memory. Then, before starting iterations over K dimension, the accumulator needs to be emptied by:

```
wmma::fill_fragment(acc_frag, 0.0f);
```

During the iterations over the K dimension, tiles in matrix A and B should be loaded from global memory to register files by:

```
wmma::load_matrix_sync(a_frag, gMemA, leadingDimA);
wmma::load_matrix_sync(b_frag, gMemB, leadingDimB);
```

Then, they will be multiplied in the register file and accumulated in the accumulator by the statement:

```
wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
```

After the iteration over the K dimension, the tiles in matrix C will be loaded to the register file and added to the result in the accumulator:

```
wmma::load_matrix_sync(c_frag, gMemC, leadingDimC, wmma::mem_col_major);
c_frag.x[idx] = alpha * acc_frag.x[idx] + beta * c_frag.x[idx];
```

Finally, the result tiles stored in the register file will be loaded back into the global memory by each warp, i.e., the D matrix:

```
wmma::store_matrix_sync(gMemD, c_frag, leadingDimC, wmma::mem_col_major);
```

In the naive implementation, data exchange only occurs between the off-chip global memory and the register files in the SM, and when warps perform matrix multiply-add operations, threads need to access the global memory multiple times. In the GPU memory hierarchy, the overhead (bandwidth + latency) of accessing global memory is the largest. Therefore, although the computation performance is improved by using tensor cores in the naive implementation, the memory performance is still constrained by global memory. Since the input data is stored in the global memory, access to the global memory is inevitable. However, by introducing the shared memory, also located in the SM, with less overhead and allowing data exchanges to occur between the shared memory and the register file as much as possible, the overhead due to frequent access to the global memory can be reduced.

As shown in Figure 5-7, a higher-performance implementation using shared memory as a buffer between global memory and the register file is proposed. 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. All thread blocks in the grid will index their corresponding submatrix in the result matrix from the top left to the bottom right. But if the number of thread blocks is not sufficient to cover the result matrix, then the thread grid will slide on the result matrix, also from the top left to the bottom right, to ensure that no elements are left out.

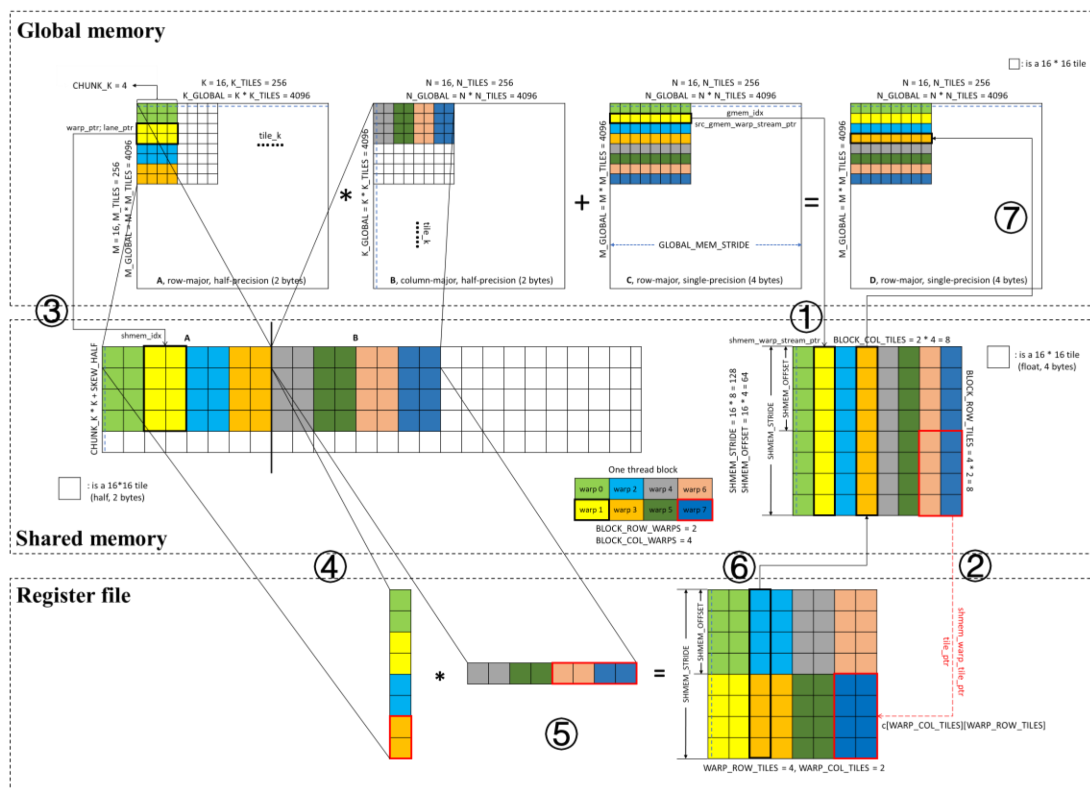

Figure 5-7: High-performance implementation of matrix multiply-accumulate operation powered by tensor cures and shared memory. Elements of different colors in matrices are handled by their same-colored warps.

As illustrated in Figure 5-5, warp is the smallest unit in the software hierarchy that needs to be handled when calling tensor cores (just like thread to CUDA core). However, based on the official CUDA Programming Guide [88], there is no definitive answer to the question of how threads in 2D blocks and 2D grids are grouped into warps of 32 threads. Practical experience [199] tells us that the threads in a 2D or 3D thread block are always linearized in row-major order first, and then every 32 threads are regarded as a warp, that is, for the launched 2D or 3D thread block, the layout of warps is also linear. Admittedly, any execution order within or between warps should not be assumed, but warps are still indexable like threads. In the high-performance implementation, for better indexing, the 8 warps in a thread block are treated as a 2×4 combination (BLOCK_ROW_WARPS=2; BLOCK_COL_WARP=4), and each warp is marked with the same color as the corresponding data in the matrix it processes.

Due to the introduction of the three-level memory hierarchy (global memory, shared memory,

register file), the implementation strategy of the high-performance version is not as straight-forward as the naive version, which involves more memory swapping and element indexing. The pseudocode of the computation process is shown in Table 5-3.

```
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 from global memory to shared memory
            for each slice in chunk
                4. Load slices of A and B from shared memory 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 5-3: Pseudocode of the high-performance matrix multiply-accumulate operation implemented with tensor cores and three-level memory hierarchy.

In step 1, the submatrices (128×128×4 bytes for each) of the C matrix stored in the global memory, handled by different thread blocks, are copied to the shared memory, that is, for a single thread block, one of the warps is responsible for copying 8 tiles (8×16×16×4 bytes). During the copying process, one thread will copy 16 bytes (int4 type), that is, one warp will copy 32×16 bytes at once, so each warp needs to iterate 16 times to copy all the elements of the 8 tiles from the global memory to the shared memory.

In step 2, the submatrices that have been copied to the shared memory are further copied to the register file. Still, in this process, a warp is responsible for copying 8 tiles, but it will be done by the tensor cores instead of the CUDA cores, so a warp can do the complete copy of a tile at once, so each warp needs to iterate 8 times to copy all the elements from the shared memory to the register file. It should be noted that the layout of the copied tiles in the register file is different from that in the shared memory (from 8×1 to 4×2), which requires additional indexing.

In step 3, thread blocks copy the chunks in matrices A and B from global memory to shared memory, and this process is performed by the CUDA cores. What is stored in the shared memory at this point is the submatrices of C, but since they have been copied to the register file in step 2, the shared memory is available for overriding. The first 4 warps in each thread block are responsible for copying chunks in matrix A, and the last 4 warps are responsible for copying chunks in matrix B. As shown in Figure 5-7, a warp is still responsible for copying 8 tiles (8×16×16×2 bytes), and each thread in the warp is responsible for copying 16 bytes, that is, a warp will copy 16×32 bytes at once, and thus each warp needs to iterate 8 times to complete the copying of 8 tiles. Be noted that, the layout of the chunks from matrices A and B changes from 2×4 in global memory to 4×2 in shared memory, and the order in which the elements of matrices A and B are stored in memory will determine the indexing strategy of

the threads in warps.

In step 4, the chunks from matrices A and B are further chopped into slices, each consisting of 8 tiles, and copied from shared memory to the register file. This process is performed by the tensor cores, so the 8 warps in thread blocks can copy the required slices from shared memory to register files at once without iterations.

In step 5, the WMMA operation is performed on one slice (8 tiles) from matrix A and one slice (8 tiles) from matrix B, which can produce a submatrix of 8 tiles by 8 tiles. In this process, each warp is responsible for 8 tiles in the submatrix, and the layout of these 8 tiles is 4×2, which means that each warp not only needs to iterate 8 times but also needs additional indexes. Depending on the width of the chunks stored in the shared memory, i.e., how many slices, steps 4 and 5 will be iterated that many times to complete the multiply-accumulate computation of the chunks.

In fact, the operations performed by steps 4 and 5 are similar to the naive implementation. The K dimension in the naive implementation is equivalent to the chunk width in steps 4 and 5, both parameters determine the number of iterations that each thread block needs to perform. In the naive implementation, one warp is responsible for one tile in the result submatrix, but in the high-performance implementation, one warp is responsible for eight. Besides the difference in the layout of the elements in memory, the most important difference is that the data exchange performed by tensor cores occurs between global memory and register files in the naive implementation, while occurs between shared memory and the register file in the high-performance implementation. Thanks to the high bandwidth and low latency of shared memory, this will greatly reduce the additional overhead caused by frequent data exchange.

Steps 6 and 7 are the reverse processes of steps 1 and 2. When the register files in different SMs are full of tiles of the result matrix D produced by warps, all tiles will first be copied to the shared memory of the SMs where they are located (performed by tensor cores), so that the shared memory storing the chunks of matrices A and B will be overridden by the submatrices of matrix D. Then, the submatrices of matrix D will be copied from shared memory to global memory (performed by CUDA cores) to piece together the full result matrix D.

Steps	Parameters
step 1, step 7	gmem_idx src_gmem_warp_stream_ptr shmem_warp_stream_ptr
step 2, step 6	shmem_warp_tile_ptr tile_ptr
step 3	warp_ptr lane_ptr shmem_idx

Table 5-4: Indexing parameters and their steps involved in the tensor core high-performance implementation of matrix multiply-accumulate operation.

The high-performance implementation with the three-level memory hierarchy performed by the tensor core improves the performance of matrix multiply-accumulate operation, but also increases the complexity of data indexing. As shown in Figure 5-7, the indexing strategy of the high-performance implementation involves many parameters (pointers and indexes), and

the objects they guide can be thread blocks, warps, or threads. Also, the memory they point to can be global memory, shared memory, or register files. Therefore, it is necessary to clarify all the parameters involved in data indexing, and Table 5-4 gives the key parameters used in the kernel function and the steps involved.

When data exchange occurs between global memory and shared memory, gmem_idx, src_gmem_warp_stream_ptr, and shmem_warp_stream_ptr are used. The first two will jointly specify the location in global memory for each warp in different thread blocks, and each warp will be assigned a location in shared memory by shmem_warp_stream_ptr. In this way, each thread block can perform the exchange of the corresponding submatrix between global memory and shared memory. For data exchange between shared memory and register files, shmem_warp_tile_ptr is used to specify the starting address of the 8 tiles stored in shared memory for each warp to copy, and tile_ptr specifies the starting address of each tile for each warp. In this way, the tensor core can complete the tile-to-tile data exchange between shared memory and register files.

Since the chunks of matrices A and B have different layouts in global and shared memory, and their precision type is different from the result matrix, transferring the chunks of matrices A and B requires additional indexes. Each thread block contains two sets of warps (the first four indexes A, the last four indexes B), and the starting address of each warp in global memory is specified by warp_ptr, and lane_ptr further specifies the starting address of each thread in each warp. The 32 threads in each warp are arranged in a 4×8 layout, that is, 8 threads per group, a total of four groups, shmem_idx will specify the element index in shared memory for each "thread group" containing 8 threads. Based on the above three parameters, thread blocks can copy the corresponding chunks in matrices A and B from global memory to shared memory.

5.2.2 Potential further parallelism from multiple data blocks

The tensor core accelerates the matrix multiply-accumulate operations in the ICA algorithm and improves the parallelization within each step. However, as shown in Figure 5-8, the existing ICA implementation structure still contains two levels of serialization: the traversal of the data block and the repetition of the entire input data. The input data containing multi-channel signals is divided into n data blocks of a specific length, and each data block is used to update the weight matrix in turn, that is, ICA will be executed n times in sequence. When all data blocks are traversed, the whole process will be repeated on the input signal until the specified number of iterations is reached.

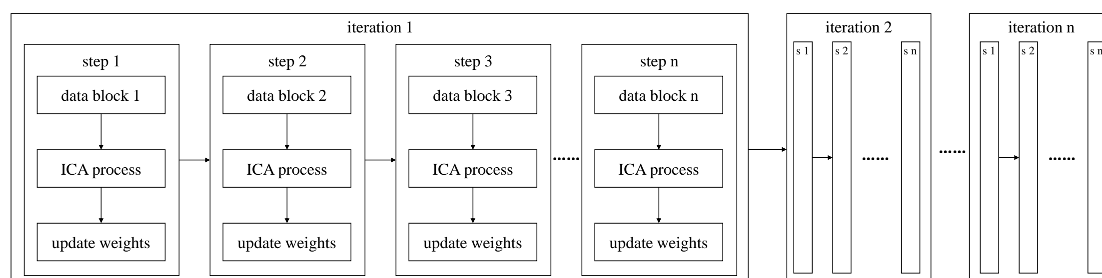

Figure 5-8: Data and processing flow of the existing ICA implementation that contains two levels of serialization.

ICA is essentially a machine learning algorithm based on the back-propagation algorithm. In the existing ICA implementation, one frame of input data is equivalent to a training sample, a data block is equivalent to a batch containing multiple training samples, the number of blocks is equivalent to the number of iterations over the training set, and the number of ICA iterations over the entire input data is equivalent to the number of epochs when training the model. For the independence criterion function as the loss function, in the process of finding its global optimal solution, the weight update operation will be performed after each data block is processed, that is, each data block needs to be processed sequentially to satisfy the dependency of their contribution to the weight.

Another strategy for updating the weights is illustrated in Figure 5-9. The contribution to the weights obtained from each data block is not used directly for updating but is stored temporarily. After all data blocks have been processed, the weights update operation is performed once. In this way, each data block is independent from each other, so their processing can be done in parallel, and after synchronization, the contributions from all the data blocks will be combined and used to update the weights. The implementation of ICA in Figure 5-9 eliminates serialization at the data block level and greatly reduces the total number of iterations, allowing GPU resources to be more fully utilized. In the case of multi-GPU systems, due to the independence between data blocks, each data block can even be processed by different GPUs to further improve performance.

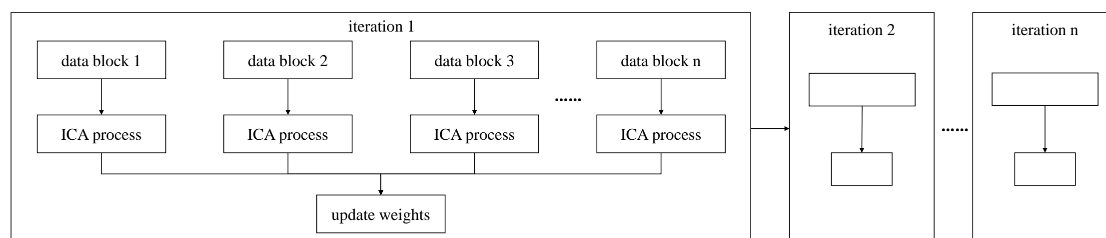

Figure 5-9: Data and processing flow of the block-parallel ICA implementation, data blocks can be processed parallelly.

The parallelism of the data blocks changes the weights updating strategy, which leads to a change in the approach path to the global optimum of the loss function (independence criterion function). Unlike other machine learning algorithms, which evaluate the model based on the accuracy of sample classification, ICA has high requirements on numerical accuracy, which is sensitive to the convergence path. Therefore, it is difficult to verify the numerical correctness of the ICA implementation based on data block parallelization. Meanwhile, in the field of machine learning, there are many studies on weight update strategies, which can be used as references for improving ICA implementation.

5.3 Results

This section presents the test results of a prototype ICA implemented by tensor cores. First, I introduce the software and hardware environment used for the tests. Then I show the numerical validation results of the tensor core version of the ICA, using the MATLAB ICA as a benchmark and the CUDA core version as a comparison. Finally, I show the kernel function profiling results of the proposed ICA implementation and the speedup under different

numbers of channels and signal lengths.

5.3.1 Test environment

MATLAB, the most widely used platform in the EEG processing community, was used as a benchmark for performance comparison. The MATLAB version of the ICA implementation was executed by MATLAB 2024a and tested on an Intel Core i7-11800H CPU, a processor from Intel's 11th-generation Tiger Lake architecture. This CPU features 8 physical cores and 16 threads, operates at a base frequency of 2.3GHz and can turbo boost up to 4.6GHz. The ICA implementation based on the tensor core and CUDA core was compiled in the CUDA 12.0 environment and tested on the NVIDIA RTX3070 mobile GPU. This GPU, built on NVIDIA's Ampere architecture, features 8GB of GDDR6 global memory and 5120 CUDA cores, with a base frequency of 1215 MHz that boosts up to 1720 MHz. Also, it is equipped with 160 third-generation tensor cores, providing powerful matrix computing performance. More details on RTX3070 mobile are shown in Table 5-5.

	Items	Values
Processors	SM Count	40
	CUDA Cores	5120
	Tensor Cores	160
	Ray Tracing Cores	40
Memory	Memory Size	8GB (DDR6)
	Memory Bus Interface	PCIe 4.0×16
	Bandwidth	448.0 GB/s
	L1 Cache	128KB (per SM)
	L2 Cache	4MB
Performance	FP32	15.97 TFLOPS
	FP64	249.6 GFLOPS
	CUDA Compute Capability	8.6
	Process Size	8 nm

Table 5-5: Specifications of the RTX 3070 mobile GPU used to implement the tensor core and CUDA core ICA.

5.3.2 Numerical validation

The main purpose of numerical verification is to ascertain the impact of the cumulative error caused by the matrix multiply-accumulate operation on the decomposition results of different ICA implementations. The computation of the tensor core is mixed-precision, i.e., the precision of the input and output matrices can be different, which makes it fundamentally different from the computations performed by MATLAB and CPU. Therefore, 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.

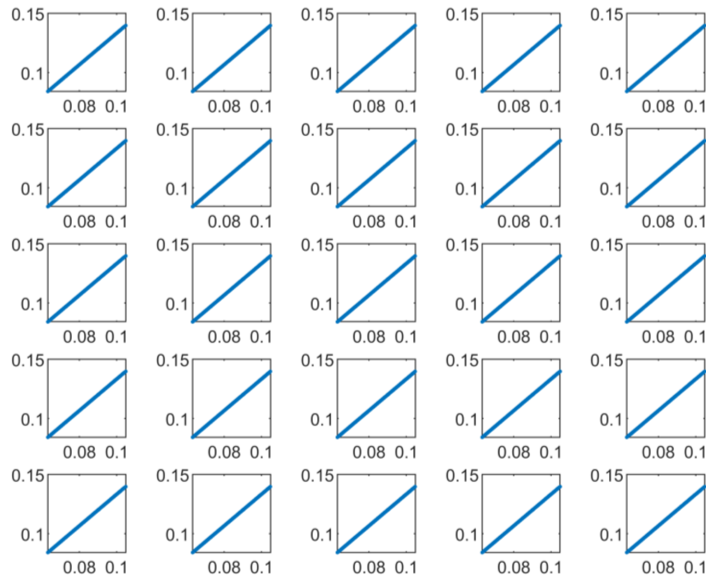

Figure 5-10: Consistency comparison results of IC pairs produced by MATLAB and tensor core versions of ICA, the average. The average similarity index between all IC pairs is higher than 0.99.

Following 512 iterations of ICA on the same input signal, the MATLAB and tensor core versions can obtain their own unmixing matrices. These matrices can then be multiplied with the original input signal to produce independent components (ICs). Then, their corresponding ICs are then plotted together for consistency comparison. If the plotted IC pairs are diagonal lines, it means that the ICs generated by different implementations are identical. Figure 5-10 shows the first 32 IC pairs produced by the MATLAB and tensor core versions of ICA, which have identical decomposition results. Using the same input and configuration, the ICs produced by the CUDA core version of ICA are also used for comparison. As shown in Figure 5-11, the diagonal of the ICs pairs shows that the CUDA core and tensor core versions of ICA also have identical decomposition results.

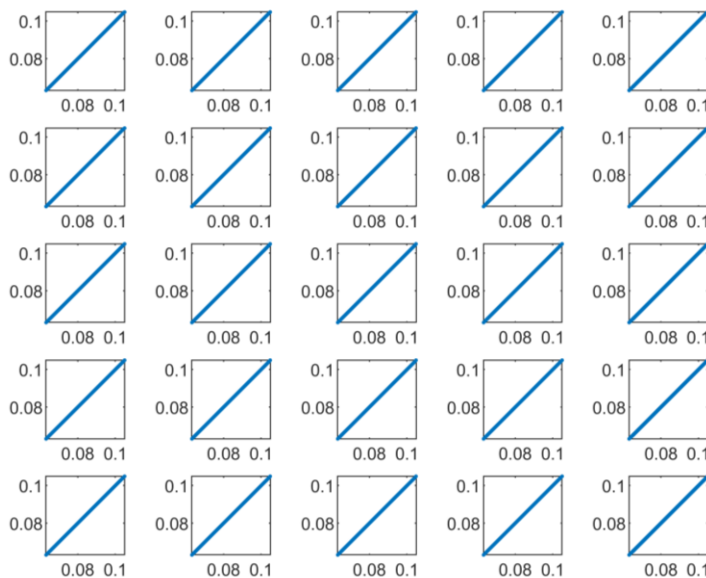

Figure 5-11: Consistency comparison results of IC pairs produced by CUDA core and tensor core versions of ICA. The average similarity index between all IC pairs is higher than 0.99.

5.3.3 Performance results and analysis

The performance of the ICA implementation based on the tensor core is demonstrated from two aspects: the speedup compared to the most widely used MATLAB version and the efficiency improvement compared to the CUDA core version. The execution time of ICA is determined by the size of the input data, which is influenced by two dimensions: the number of channels and the signal length. Given that the object processed by each iteration of ICA is a data block, the signal length is determined by the block length and the number of blocks. 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 in Figure 5-12, 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. Compared to the GPU implementations of CEEMDAN and MEMD, the acceleration of ICA based on tensor cores is somewhat less effective. This is mainly because ICA implementations still involve a significant amount of sequential execution, such as data block iterations and convergence iterations based on the back propagation algorithm. In contrast, in GPU-based EMD implementations, sequential operations only occur between IMFs, while the input signal is processed in parallel as a whole. The longer the data, the better the hardware can be utilized, leading to higher acceleration.

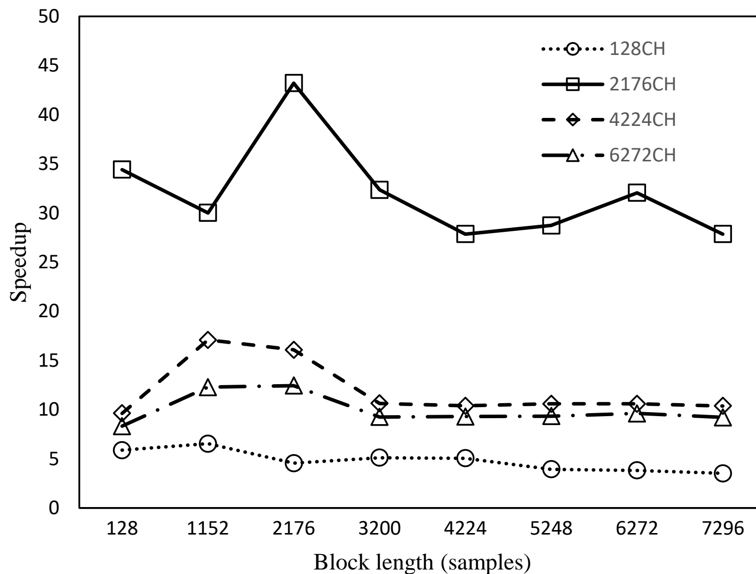

Figure 5-12: Speedup of ICA implemented by the tensor core for different number of channels and signal lengths, compared to MATLAB.

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 [186]. The performance bottleneck of the kernel function can be categorized into memory-bound and compute-bound. If a kernel function requests fewer arithmetic operations per memory access on average, it is memory-bound, otherwise it is compute-bound. Practically, the overhead of memory accesses is much greater than that of

performing computations: 10-20 clock cycles of latency for arithmetic operations, while 400-800 clock cycles of latency for global memory accesses. Therefore, the design of kernel functions should aim to perform as many arithmetic operations as possible in a limited number of memory accesses. The arithmetic intensity (flops/byte) is the measure of how many operations are done per bytes loaded or stored from memory, and the roofline model is the maximum performance that a given hardware can achieve at different arithmetic intensities. Therefore, the program or functions running on the given hardware can be profiled on the Roofline model to analyze performance bottlenecks and hardware usage efficiency.

As shown in Figure 5-13, 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.

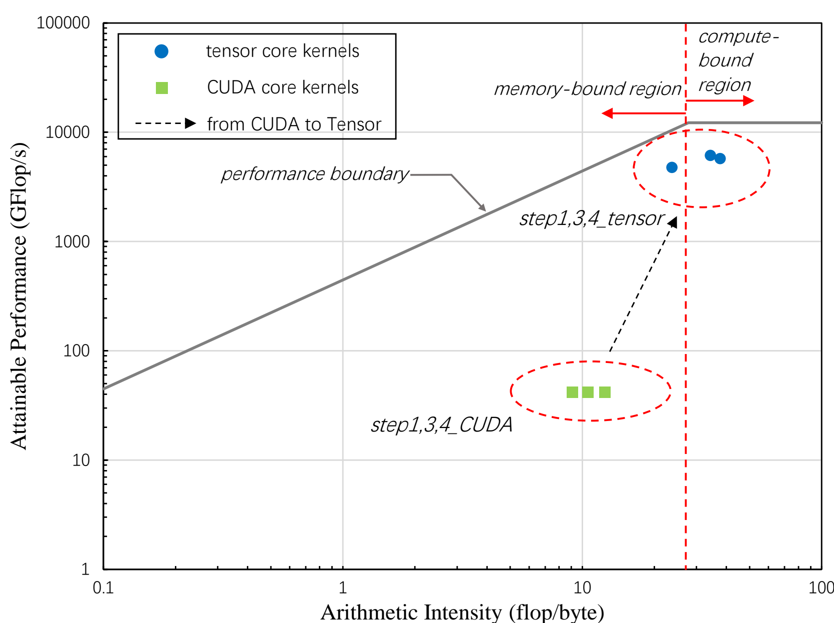

Figure 5-13: The Roofline performance results of the kernels from CUDA core and tensor core versions of ICA, executed on the RTX 3070 mobile GPU.

5.4 Summary

In this chapter, I described an ICA prototype implemented by the high-performance matrix multiply-accumulate operation based on tensor cores. As a spatial signal decomposition method, ICA is not only the de facto standard method for artifact removal in EGG signal processing, but also an important tool for restoring real brain source signals. Due to its blind source separation property, ICA also has important applications in audio processing, image processing, financial data analysis, etc., but the use of ICA in massive data processing is always problematic as the algorithm is computationally expensive.

The proposed implementation makes full use of the high efficiency of tensor cores in matrix multiply-accumulate operations and reduces the overhead of memory exchange in the ICA

implementation by introducing the three-level memory hierarchy. Numerical validations and performance analyses were performed on the proposed ICA implementation. In terms of decomposition results, the tensor core ICA implementation is numerically consistent with other implementations, which reveals that the mixed precision properties of the tensor core and the cumulative error of the computations have negligible impact. Compared to the MATLAB version, which is the most commonly used in the EEG processing community, the tensor core version of ICA achieves 3-43x speedup depending on the size of the input signal, and its memory throughput and hardware utilization are greatly improved compared to the CUDA core version running on the same GPU. This is still a work in progress, and there is plenty of room for optimization of ICA implementation. In the future, I will work on developing a complete descendant ICA implementation with further performance improvements brought by data block parallelization.

6 CONCLUSIONS

This thesis has 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 has 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.

A key contribution of this work is the design, development and optimization of GPU-accelerated implementations for EEG signal decomposition. The thesis systematically examined various signal decomposition methods, including Fourier Transform (FT), Wavelet Transform (WT), and Empirical Mode Decomposition (EMD), highlighting their strengths and limitations. Then, the study focused on advanced decomposition methods. For the time-frequency analysis of EEG signals, while the EMD algorithm effectively decomposes intrinsic oscillation patterns without relying on a preset function, it is highly sensitive to intermittent noise. The Improved Complete Ensemble EMD with Adaptive Noise (ICEEMDAN) enhances robustness against noise but is computationally intensive. The proposed GPU implementation of ICEEMDAN significantly accelerates computation while preserving adaptability to the dynamic characteristics of EEG signals and robustness to noise, achieving a maximum 260x speedup compared to the MATLAB implementation.

Traditional EMD is designed for single-channel signals, whereas EEG consists of multi-channel recordings from multiple electrodes. Multivariate Empirical Mode Decomposition (MEMD), an extension of EMD for multi-channel EEG, can extract intrinsic oscillation patterns while maintaining inter-channel correlations essential for functional connectivity research. However, its multi-channel nature makes MEMD computationally demanding. By parallelizing multi-dimensional signal processing, the proposed GPU-based MEMD implementation achieves a maximum acceleration of 430 times compared to MATLAB.

Additionally, a novel implementation leveraging tensor cores to accelerate Independent Component Analysis (ICA), a signal spatial decomposition method, was introduced. Analysis of the ICA algorithm revealed that extensive matrix multiply-add operations present a major performance bottleneck. By utilizing GPU tensor cores—optimized for efficient matrix operations—these computations, previously handled by CUDA cores, were offloaded to tensor cores. This optimization resulted in a max 43x speedup compared to the MATLAB ICA implementation.

The findings of this thesis emphasize the importance of parallel computing in EEG signal processing. For time-frequency analysis, GPU parallelization significantly enhances the extraction of oscillation patterns, improving both time-frequency resolution and computational efficiency. In spatial decomposition, GPUs accelerate processing by leveraging specialized hardware units optimized for numerical operations, transforming large-scale EEG processing from an impractical challenge into a feasible solution. Beyond performance improvements, this study also addresses practical considerations for implementing GPU-accelerated EEG processing. In CPU-GPU heterogeneous computing systems, optimizing memory allocation, efficiently scheduling threads in kernel functions, and leveraging new

hardware units are key factors in maximizing GPU performance for EEG signal decomposition.

The human brain consists of nearly 100 billion neurons, each forming thousands of synaptic connections and continuously generating vast amounts of information essential for understanding brain function and treating neurological disorders. However, in the post-Moore era, as data volumes grow exponentially, the tension between computational efficiency and massive data processing becomes increasingly evident. The era of relying solely on hardware advancements for performance gains has ended. Instead, unlocking the full potential of computing devices through algorithm parallelization presents new opportunities for EEG processing. That said, challenges remain. Algorithm parallelization is complex, requiring a deep understanding of both signal characteristics and computing hardware. Ensuring the robustness and scalability of parallelized signal decomposition remains a critical hurdle. Moreover, achieving real-time EEG processing on wearable, low-power portable devices is an ongoing challenge yet to be fully resolved.

In conclusion, this thesis has demonstrated that GPU acceleration offers a powerful solution for overcoming the computational limitations of EEG signal decomposition. By developing efficient GPU implementations of CEEMDAN, MEMD, and ICA, this research has contributed to the field of EEG processing, enabling faster, more precise, and scalable analysis of neural signals. These advancements hold significant potential for clinical applications, cognitive research, and next-generation brain-computer interfaces. Future work should continue to refine these techniques. One promising direction is to integrate the developed GPU-based signal decomposition algorithm into an end-to-end EEG processing pipeline to improve system-wide efficiency. Moreover, exploring the use of advanced hardware accelerators, such as tensor cores, in more robust ICA variants or other signal processing methods could help overcome current limitations and expand the practical applicability of EEG signal analysis.

7 SUMMARY OF THE MAIN CONTRIBUTIONS

Thesis I: GPU implementation of ICEEMDAN

I developed a GPU implementation of the Improved Complete Ensemble Empirical Mode Decomposition with Adaptive Noise (ICEEMDAN), a crucial algorithm for time-frequency analysis of non-stationary EEG signals. 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. Publications related to this thesis are as follows:

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.

Thesis II: GPU implementation of MEMD

I extended the scope of GPU-accelerated EEG analysis by implementing the Multivariate Empirical Mode Decomposition (MEMD) algorithm, which is designed for multi-channel bioelectric signals. 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. Publications related to this thesis are as follows:

1. **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.
2. **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.

Thesis III: GPU implementation of ICA

I introduced a novel application of GPU tensor cores to Independent Component Analysis (ICA), a fundamental technique for artifact removal in EEG processing. 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 this thesis are as follows:

1. **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.

LIST OF PUBLICATIONS

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.
6. **Z. Wang**, Z. Juhasz, "Implementation strategies for EEG processing on multi-GPU computing systems", XXXVI. Neumann Colloquium, Veszprem, Hungary, Dec 1 - 2, 2023, ISBN: 978-963-396-272-5.
7. **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.
8. H. Ma, **Z. Wang**, C. Li, J. Chen, Y. Wang, "Phase–Amplitude Coupling and Epileptogenic Zone Localization of Frontal Epilepsy Based on Intracranial EEG", Frontiers in Neurology, vol. 12, p. 718683, Sept 2021, doi: 10.3389/fneur.2021.718683
9. C. Li, S. Liu, **Z. Wang**, "Classifying Interictal Epileptiform Activities in Intracranial EEG Using Complex-Valued Convolutional Neural Network", International Journal of Psychophysiology, vol. 168, p. 104-105, Oct 2021, doi: 10.1016/j.ijpsycho.2021.07.314
10. C. Li, S. Liu, **Z. Wang**, G. Yuan, "Classifying epileptic phase-amplitude coupling in SEEG using complex-valued convolutional neural network", Frontiers in Physiology, vol. 13, p. 1085530, Jan. 2023, doi: 10.3389/fphys.2022.1085530

References

[1] L. G. . Kiloh and J. W. . Osselton, *Clinical electroencephalography [by] L.G. Kiloh and J.W. Osselton*. Butterworths, 1961. Accessed: Oct. 15, 2024. [Online]. Available: http://www.sciencedirect.com:5070/book/9781483167688/clinical-electroencephalography

[2] H. Berger, "Über das Elektrenkephalogramm des Menschen," *Arch Psychiatr Nervenkr*, vol. 87, no. 1, pp. 527–570, Dec. 1929, doi: 10.1007/BF01797193/METRICS.

[3] N. K. Logothetis, "What we can do and what we cannot do with fMRI," *Nature 2008 453:7197*, vol. 453, no. 7197, pp. 869–878, Jun. 2008, doi: 10.1038/nature06976.

[4] S. Basu, T. C. Kwee, S. Surti, E. A. Akin, D. Yoo, and A. Alavi, "Fundamentals of PET and PET/CT imaging," *Ann N Y Acad Sci*, vol. 1228, no. 1, pp. 1–18, Jun. 2011, doi: 10.1111/J.1749-6632.2011.06077.X.

[5] S. J. Luck, "Event-related potentials.," *APA handbook of research methods in psychology, Vol 1: Foundations, planning, measures, and psychometrics.*, pp. 523–546, 2012, doi: 10.1037/13619-028.

[6] G. Assenza and V. Di Lazzaro, "A useful electroencephalography (EEG) marker of brain plasticity: Delta waves," *Neural Regen Res*, vol. 10, no. 8, pp. 1216–1217, Aug. 2015, doi: 10.4103/1673-5374.162698.

[7] W. Klimesch, "EEG alpha and theta oscillations reflect cognitive and memory performance: a review and analysis," *Brain Res Rev*, vol. 29, no. 2–3, pp. 169–195, Apr. 1999, doi: 10.1016/S0165-0173(98)00056-3.

[8] M. L. Perlis, H. Merica, M. T. Smith, and D. E. Giles, "Beta EEG activity and insomnia," *Sleep Med Rev*, vol. 5, no. 5, pp. 365–376, Oct. 2001, doi: 10.1053/SMRV.2001.0151.

[9] C. S. Herrmann and T. Demiralp, "Human EEG gamma oscillations in neuropsychiatric disorders," *Clinical Neurophysiology*, vol. 116, no. 12, pp. 2719–2733, Dec. 2005, doi: 10.1016/J.CLINPH.2005.07.007.

[10] M. Palus, "Nonlinearity in normal human EEG: cycles, temporal asymmetry, nonstationarity and randomness, not chaos.," *Biol Cybern*, vol. 75, no. 5, pp. 389–396, 1996, doi: 10.1007/S004220050304/METRICS.

[11] C. J. Stam, B. Jelles, H. A. M. Achtereekte, S. A. R. B. Rombouts, J. P. J. Slaets, and R. W. M. Keunen, "Investigation of EEG non-linearity in dementia and Parkinson's disease," *Electroencephalogr Clin Neurophysiol*, vol. 95, no. 5, pp. 309–317, Nov. 1995, doi: 10.1016/0013-4694(95)00147-Q.

[12] Y. J. Lee, Y. S. Zhu, Y. H. Xu, M. F. Shen, H. X. Zhang, and N. V. Thakor, "Detection of non-linearity in the EEG of schizophrenic patients," *Clinical Neurophysiology*, vol. 112, no. 7, pp. 1288–1294, Jul. 2001, doi: 10.1016/S1388-2457(01)00544-2.

[13] W. Klonowski, "Everything you wanted to ask about EEG but were afraid to get the right answer," *Nonlinear Biomed Phys*, vol. 3, p. 2, May 2009, doi: 10.1186/1753-4631-3-2.

[14] S. Qin and Z. Ji, "Extraction of features in EEG signals with the non-stationary signal analysis technology," *Annual International Conference of the IEEE Engineering in*

Medicine and Biology - Proceedings, vol. 26 I, pp. 349–352, 2004, doi: 10.1109/IEMBS.2004.1403164.

[15] H. G. Schwartz and A. S. Kerr, “ELECTRICAL ACTIVITY OF THE EXPOSED HUMAN BRAIN: DESCRIPTION OF TECHNIC AND REPORT OF OBSERVATIONS,” *Journal of Nervous and Mental Disease*, vol. 43, no. 3, pp. 547–559, 1940, doi: 10.1001/ARCHNEURPSYC.1940.02280030121010.

[16] G. Dietsch, “Fourier-Analyse von Elektrencephalogrammen des Menschen,” *Pflugers Arch Gesamte Physiol Menschen Tiere*, vol. 230, no. 1, pp. 106–112, Dec. 1932, doi: 10.1007/BF01751972/METRICS.

[17] J. C. Shaw, “Correlation and coherence analysis of the EEG: A selective tutorial review,” *International Journal of Psychophysiology*, vol. 1, no. 3, pp. 255–266, Mar. 1984, doi: 10.1016/0167-8760(84)90045-X.

[18] B. Fujimori, T. Yokota, Y. Ishibashi, and T. Takei, “Analysis of the electroencephalogram of children by histogram method,” *Electroencephalogr Clin Neurophysiol*, vol. 10, no. 2, pp. 241–252, May 1958, doi: 10.1016/0013-4694(58)90031-2.

[19] D. Parameshwaran, N. P. Subramaniyam, and T. C. Thiagarajan, “Waveform complexity: A new metric for EEG analysis,” *J Neurosci Methods*, vol. 325, p. 108313, Sep. 2019, doi: 10.1016/J.JNEUMETH.2019.108313.

[20] G. H. B. Yford, “Signal variance and its application to continuous measurements of e.e.g. activity,” *Proc R Soc Lond B Biol Sci*, vol. 161, no. 984, pp. 421–437, Jan. 1965, doi: 10.1098/RSPB.1965.0013.

[21] S. Uchida, M. Matsuura, S. Ogata, T. Yamamoto, and N. Aikawa, “Computerization of Fujimori’s method of waveform recognition A review and methodological considerations for its application to all-night sleep EEG,” *J Neurosci Methods*, vol. 64, no. 1, pp. 1–12, Jan. 1996, doi: 10.1016/0165-0270(95)00115-8.

[22] P. L. Nunez *et al.*, “EEG coherency II: experimental comparisons of multiple measures,” *Clinical Neurophysiology*, vol. 110, no. 3, pp. 469–486, Mar. 1999, doi: 10.1016/S1388-2457(98)00043-1.

[23] A. Isaksson, A. Wennberg, and L. H. Zetterberg, “Computer Analysis of EEG Signals with Parametric Models,” *Proceedings of the IEEE*, vol. 69, no. 4, pp. 451–461, 1981, doi: 10.1109/PROC.1981.11988.

[24] X. W. Wang, D. Nie, and B. L. Lu, “EEG-Based Emotion Recognition Using Frequency Domain Features and Support Vector Machines,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7062 LNCS, no. PART 1, pp. 734–743, 2011, doi: 10.1007/978-3-642-24955-6_87.

[25] M. M. Siddiqui, G. Srivastava, and S. H. Saeed, “Diagnosis of insomnia sleep disorder using short time frequency analysis of PSD approach applied on EEG signal using channel ROC-LOC,” *Sleep Science*, vol. 9, no. 3, pp. 186–191, Jul. 2016, doi: 10.1016/J.SLSCI.2016.07.002.

[26] M. N. Alam, M. I. Ibrahimy, and S. M. A. Motakabber, “Feature Extraction of EEG Signal by Power Spectral Density for Motor Imagery Based BCI,” *Proceedings of the 8th International Conference on Computer and Communication Engineering, ICCCE*

2021, pp. 234–237, Jun. 2021, doi: 10.1109/ICCCE50029.2021.9467141.

[27] O. A. Petroff, D. D. Spencer, I. I. Goncharova, and H. P. Zaveri, "A comparison of the power spectral density of scalp EEG and subjacent electrocorticograms," *Clinical Neurophysiology*, vol. 127, no. 2, pp. 1108–1112, Feb. 2016, doi: 10.1016/J.CLINPH.2015.08.004.

[28] M. K. Kiymik, I. Güler, A. Dizibüyük, and M. Akin, "Comparison of STFT and wavelet transform methods in determining epileptic seizure activity in EEG signals for real-time application," *Comput Biol Med*, vol. 35, no. 7, pp. 603–616, Oct. 2005, doi: 10.1016/J.COMPBIOMED.2004.05.001.

[29] P. Peng, Y. Song, L. Yang, and H. Wei, "Seizure Prediction in EEG Signals Using STFT and Domain Adaptation," *Front Neurosci*, vol. 15, p. 825434, Jan. 2022, doi: 10.3389/FNINS.2021.825434/BIBTEX.

[30] H. Adeli, Z. Zhou, and N. Dadmehr, "Analysis of EEG records in an epileptic patient using wavelet transform," *J Neurosci Methods*, vol. 123, no. 1, pp. 69–87, Feb. 2003, doi: 10.1016/S0165-0270(02)00340-0.

[31] N. Hazarika, J. Z. Chen, A. C. Tsoi, and A. Sergejew, "Classification of EEG signals using the wavelet transform," *Signal Processing*, vol. 59, no. 1, pp. 61–72, May 1997, doi: 10.1016/S0165-1684(97)00038-8.

[32] H. Sharabaty, B. Jammes, and D. Esteve, "EEG analysis using HHT: One step toward automatic drowsiness scoring," *Proceedings - International Conference on Advanced Information Networking and Applications, AINA*, pp. 826–831, 2008, doi: 10.1109/WAINA.2008.271.

[33] S. Yang and F. Deravi, "Novel HHT-based features for biometric identification using EEG signals," *Proceedings - International Conference on Pattern Recognition*, pp. 1922–1927, Dec. 2014, doi: 10.1109/ICPR.2014.336.

[34] K. Fu, J. Qu, Y. Chai, and Y. Dong, "Classification of seizure based on the time-frequency image of EEG signals using HHT and SVM," *Biomed Signal Process Control*, vol. 13, no. 1, pp. 15–22, Sep. 2014, doi: 10.1016/J.BSPC.2014.03.007.

[35] A. Iraji *et al.*, "Spatial dynamics within and between brain functional domains: A hierarchical approach to study time-varying brain function," *Hum Brain Mapp*, vol. 40, no. 6, p. 1969, Apr. 2018, doi: 10.1002/HBM.24505.

[36] S. E. Petersen and O. Sporns, "Brain Networks and Cognitive Architectures," *Neuron*, vol. 88, no. 1, pp. 207–219, Oct. 2015, doi: 10.1016/J.NEURON.2015.09.027/ASSET/9AC35310-3DF1-4755-A54A-26D6DCC9114B/MAIN.ASSETS/GR7.JPG.

[37] M. A. Jatoi, N. Kamel, A. S. Malik, I. Faye, and T. Begum, "A survey of methods used for source localization using EEG signals," *Biomed Signal Process Control*, vol. 11, no. 1, pp. 42–52, May 2014, doi: 10.1016/J.BSPC.2014.01.009.

[38] J. Song *et al.*, "EEG source localization: Sensor density and head surface coverage," *J Neurosci Methods*, vol. 256, pp. 9–21, Dec. 2015, doi: 10.1016/J.JNEUMETH.2015.08.015.

[39] C. M. Michel and B. He, "EEG source localization," *Handb Clin Neurol*, vol. 160, pp. 85–101, Jan. 2019, doi: 10.1016/B978-0-444-64032-1.00006-0.

[40] Y. Li, W. Zheng, L. Wang, Y. Zong, and Z. Cui, "From Regional to Global Brain: A

Novel Hierarchical Spatial-Temporal Neural Network Model for EEG Emotion Recognition,” *IEEE Trans Affect Comput*, vol. 13, no. 2, pp. 568–578, 2022, doi: 10.1109/TAFFC.2019.2922912.

[41] R. Fu, H. Wang, T. Bao, and M. Han, “EEG intentions recognition in dynamic complex object control task by functional brain networks and regularized discriminant analysis,” *Biomed Signal Process Control*, vol. 61, p. 101998, Aug. 2020, doi: 10.1016/J.BSPC.2020.101998.

[42] G. Naik and D. Kumar, “An Overview of Independent Component Analysis and Its Applications,” *Informatica*, 2011, doi: 10.31449/INF.V35I1.334.

[43] A. Hyvärinen, “Independent component analysis: recent advances,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1984, Feb. 2013, doi: 10.1098/RSTA.2011.0534.

[44] J. Onton, M. Westerfield, J. Townsend, and S. Makeig, “Imaging human EEG dynamics using independent component analysis,” *Neurosci Biobehav Rev*, vol. 30, no. 6, pp. 808–822, Jan. 2006, doi: 10.1016/J.NEUBIOREV.2006.06.007.

[45] M. Greenacre, P. J. F. Groenen, T. Hastie, A. I. D’Enza, A. Markos, and E. Tuzhilina, “Principal component analysis,” *Nature Reviews Methods Primers 2022 2:1*, vol. 2, no. 1, pp. 1–21, Dec. 2022, doi: 10.1038/s43586-022-00184-w.

[46] U. Rajendra Acharya, S. Vinitha Sree, A. P. C. Alvin, and J. S. Suri, “Use of principal component analysis for automatic classification of epileptic EEG activities in wavelet framework,” *Expert Syst Appl*, vol. 39, no. 10, pp. 9072–9078, Aug. 2012, doi: 10.1016/J.ESWA.2012.02.040.

[47] H. Wang and W. Zheng, “Local temporal common spatial patterns for robust single-trial EEG classification,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 16, no. 2, pp. 131–139, Apr. 2008, doi: 10.1109/TNSRE.2007.914468.

[48] F. Lotte and C. Guan, “Regularizing common spatial patterns to improve BCI designs: Unified theory and new algorithms,” *IEEE Trans Biomed Eng*, vol. 58, no. 2, pp. 355–362, Feb. 2011, doi: 10.1109/TBME.2010.2082539.

[49] I. Güler and E. D. Übeyli, “Multiclass support vector machines for EEG-signals classification,” *IEEE Transactions on Information Technology in Biomedicine*, vol. 11, no. 2, pp. 117–126, Mar. 2007, doi: 10.1109/TITB.2006.879600.

[50] D. R. Edla, K. Mangalorekar, G. Dhavalikar, and S. Dodia, “Classification of EEG data for human mental state analysis using Random Forest Classifier,” *Procedia Comput Sci*, vol. 132, pp. 1523–1532, Jan. 2018, doi: 10.1016/J.PROCS.2018.05.116.

[51] M. N. A. H. Sha’abani, N. Fuad, N. Jamal, and M. F. Ismail, “kNN and SVM Classification for EEG: A Review,” *Lecture Notes in Electrical Engineering*, vol. 632, pp. 555–565, 2020, doi: 10.1007/978-981-15-2317-5_47.

[52] Y. Liu, W. Zhou, Q. Yuan, and S. Chen, “Automatic seizure detection using wavelet transform and SVM in long-term intracranial EEG,” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 20, no. 6, pp. 749–755, 2012, doi: 10.1109/TNSRE.2012.2206054.

[53] A. Rakotomamonjy and V. Guigue, “BCI competition III: Dataset II- ensemble of SVMs for BCI P300 speller,” *IEEE Trans Biomed Eng*, vol. 55, no. 3, pp. 1147–1154,

Mar. 2008, doi: 10.1109/TBME.2008.915728.

[54] H. Li, M. Ding, R. Zhang, and C. Xiu, "Motor imagery EEG classification algorithm based on CNN-LSTM feature fusion network," *Biomed Signal Process Control*, vol. 72, p. 103342, Feb. 2022, doi: 10.1016/J.BSPC.2021.103342.

[55] N. Mammone, C. Ieracitano, and F. C. Morabito, "A deep CNN approach to decode motor preparation of upper limbs from time–frequency maps of EEG signals at source level," *Neural Networks*, vol. 124, pp. 357–372, Apr. 2020, doi: 10.1016/J.NEUNET.2020.01.027.

[56] X. Du *et al.*, "An Efficient LSTM Network for Emotion Recognition From Multichannel EEG Signals," *IEEE Trans Affect Comput*, vol. 13, no. 3, pp. 1528–1540, 2022, doi: 10.1109/TAFFC.2020.3013711.

[57] W. Mumtaz, S. Rasheed, and A. Irfan, "Review of challenges associated with the EEG artifact removal methods," *Biomed Signal Process Control*, vol. 68, p. 102741, Jul. 2021, doi: 10.1016/J.BSPC.2021.102741.

[58] W. Mumtaz, S. Rasheed, and A. Irfan, "Review of challenges associated with the EEG artifact removal methods," *Biomed Signal Process Control*, vol. 68, p. 102741, Jul. 2021, doi: 10.1016/J.BSPC.2021.102741.

[59] C. Haddix *et al.*, "Removal of movement-induced EEG artifacts: current state of the art and guidelines," *J Neural Eng*, vol. 19, no. 1, p. 011004, Feb. 2022, doi: 10.1088/1741-2552/AC542C.

[60] D. Yao, Y. Qin, S. Hu, L. Dong, M. L. Bringas Vega, and P. A. Valdés Sosa, "Which Reference Should We Use for EEG and ERP practice?," *Brain Topogr*, vol. 32, no. 4, pp. 530–549, Jul. 2019, doi: 10.1007/S10548-019-00707-X/FIGURES/12.

[61] F. Chella, V. Pizzella, F. Zappasodi, and L. Marzetti, "Impact of the reference choice on scalp EEG connectivity estimation," *J Neural Eng*, vol. 13, no. 3, p. 036016, May 2016, doi: 10.1088/1741-2560/13/3/036016.

[62] C. M. Michel and T. Koenig, "EEG microstates as a tool for studying the temporal dynamics of whole-brain neuronal networks: A review," *Neuroimage*, vol. 180, pp. 577–593, Oct. 2018, doi: 10.1016/J.NEUROIMAGE.2017.11.062.

[63] B. S. Oken and K. H. Chiappa, "Short-term variability in EEG frequency analysis," *Electroencephalogr Clin Neurophysiol*, vol. 69, no. 3, pp. 191–198, Mar. 1988, doi: 10.1016/0013-4694(88)90128-9.

[64] N. Jrad and M. Congedo, "Identification of spatial and temporal features of EEG," *Neurocomputing*, vol. 90, pp. 66–71, Aug. 2012, doi: 10.1016/J.NEUCOM.2012.02.032.

[65] Y. Li, W. Zheng, L. Wang, Y. Zong, and Z. Cui, "From Regional to Global Brain: A Novel Hierarchical Spatial-Temporal Neural Network Model for EEG Emotion Recognition," *IEEE Trans Affect Comput*, vol. 13, no. 2, pp. 568–578, 2022, doi: 10.1109/TAFFC.2019.2922912.

[66] S. Morales and M. E. Bowers, "Time-frequency analysis methods and their application in developmental EEG data," *Dev Cogn Neurosci*, vol. 54, p. 101067, Apr. 2022, doi: 10.1016/J.DCN.2022.101067.

[67] M. A. Guevara and M. Corsi-Cabrera, "EEG coherence or EEG correlation?," *International Journal of Psychophysiology*, vol. 23, no. 3, pp. 145–153, Oct. 1996, doi:

10.1016/S0167-8760(96)00038-4.

[68] J. Fell and N. Axmacher, "The role of phase synchronization in memory processes," *Nature Reviews Neuroscience 2011 12:2*, vol. 12, no. 2, pp. 105–118, Jan. 2011, doi: 10.1038/nrn2979.

[69] C. A. Stefano Filho, R. Attux, and G. Castellano, "Can graph metrics be used for EEG-BCIs based on hand motor imagery?," *Biomed Signal Process Control*, vol. 40, pp. 359–365, Feb. 2018, doi: 10.1016/J.BSPC.2017.09.026.

[70] E. Derya Übeyli, "Statistics over features: EEG signals analysis," *Comput Biol Med*, vol. 39, no. 8, pp. 733–741, Aug. 2009, doi: 10.1016/J.COMPBIOMED.2009.06.001.

[71] A. Mensen and R. Khatami, "Advanced EEG analysis using threshold-free cluster-enhancement and non-parametric statistics," *Neuroimage*, vol. 67, pp. 111–118, Feb. 2013, doi: 10.1016/J.NEUROIMAGE.2012.10.027.

[72] M. K. Siddiqui, R. Morales-Menendez, X. Huang, and N. Hussain, "A review of epileptic seizure detection using machine learning classifiers," *Brain Inform*, vol. 7, no. 1, pp. 1–18, Dec. 2020, doi: 10.1186/S40708-020-00105-1/TABLES/3.

[73] X. W. Wang, D. Nie, and B. L. Lu, "Emotional state classification from EEG data using machine learning approach," *Neurocomputing*, vol. 129, pp. 94–106, Apr. 2014, doi: 10.1016/J.NEUCOM.2013.06.046.

[74] T. Mullen *et al.*, "Real-time modeling and 3D visualization of source dynamics and connectivity using wearable EEG," *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBS*, pp. 2184–2187, 2013, doi: 10.1109/EMBC.2013.6609968.

[75] T. Liu *et al.*, "Applying high-performance computing in drug discovery and molecular simulation," *Natl Sci Rev*, vol. 3, no. 1, pp. 49–63, Mar. 2016, doi: 10.1093/NSR/NWW003.

[76] Z. Guo *et al.*, "Extending the limit of molecular dynamics with ab initio accuracy to 10 billion atoms," *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pp. 205–218, Apr. 2022, doi: 10.1145/3503221.3508425.

[77] Z. Wu, P. Ma, X. Zhang, and G. Ye, "Efficient Management and Processing of Massive InSAR Images Using an HPC-Based Cloud Platform," *IEEE J Sel Top Appl Earth Obs Remote Sens*, vol. 17, pp. 2866–2876, 2024, doi: 10.1109/JSTARS.2023.3349214.

[78] B. D. Wirth, K. D. Hammond, S. I. Krasheninnikov, and D. Maroudas, "Challenges and opportunities of modeling plasma–surface interactions in tungsten using high-performance computing," *Journal of Nuclear Materials*, vol. 463, pp. 30–38, Aug. 2015, doi: 10.1016/J.JNUCMAT.2014.11.072.

[79] A. Plaza, J. Plaza, and H. Vegas, "Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware-based systems," *J Signal Process Syst*, vol. 61, no. 3, pp. 293–315, Dec. 2010, doi: 10.1007/S11265-010-0453-1/METRICS.

[80] E. Monmasson and M. N. Cirstea, "FPGA design methodology for industrial control systems - A review," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 4, pp. 1824–1842, Aug. 2007, doi: 10.1109/TIE.2007.898281.

[81] A. Branover, D. Foley, and M. Steinman, "AMD fusion APU: Llano," *IEEE Micro*, vol.

32, no. 2, pp. 28–37, Mar. 2012, doi: 10.1109/MM.2012.2.

[82] A. Duran and M. Klemm, "The Intel® many integrated core architecture," *Proceedings of the 2012 International Conference on High Performance Computing and Simulation, HPCS 2012*, pp. 365–366, 2012, doi: 10.1109/HPCSIM.2012.6266938.

[83] Z. Juhasz, J. Ďurian, A. Derzsi, Š. Matejčík, Z. Donkó, and P. Hartmann, "Efficient GPU implementation of the Particle-in-Cell/Monte-Carlo collisions method for 1D simulation of low-pressure capacitively coupled plasmas," *Comput Phys Commun*, vol. 263, p. 107913, Jun. 2021, doi: 10.1016/J.CPC.2021.107913.

[84] J. Yang, Y. Wang, and Y. Chen, "GPU accelerated molecular dynamics simulation of thermal conductivities," *J Comput Phys*, vol. 221, no. 2, pp. 799–804, Feb. 2007, doi: 10.1016/J.JCP.2006.06.039.

[85] O. Artiles and F. Saeed, "GPU-SFFT: A GPU based parallel algorithm for computing the Sparse Fast Fourier Transform (SFFT) of k-sparse signals," *Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019*, pp. 3303–3311, Dec. 2019, doi: 10.1109/BIGDATA47090.2019.9006579.

[86] R. S. Luley and Q. Qiu, "Effective utilization of CUDA hyper-Q for improved power and performance efficiency," *Proceedings - 2016 IEEE 30th International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 1160–1169, Jul. 2016, doi: 10.1109/IPDPSW.2016.154.

[87] NVIDIA, "NVIDIA ADA GPU ARCHITECTURE." Accessed: Feb. 11, 2025. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[88] John. Cheng, Max. Grossman, and Ty. KcKercher, "Professional CUDA C programming," 2014.

[89] H. Adeli, Z. Zhou, and N. Dadmehr, "Analysis of EEG records in an epileptic patient using wavelet transform," *J Neurosci Methods*, vol. 123, no. 1, pp. 69–87, Feb. 2003, doi: 10.1016/S0165-0270(02)00340-0.

[90] M. . Ullsperger and Stefan. Debener, "Simultaneous EEG and fMRI : recording, analysis, and application," p. 315, 2010, Accessed: Feb. 11, 2025. [Online]. Available: https://books.google.com/books/about/Simultaneous_EEG_and_FMRI.html?id=yo4UDAAAQBAJ

[91] F. Riaz, A. Hassan, S. Rehman, I. K. Niazi, and K. Dremstrup, "EMD-based temporal and spectral features for the classification of EEG signals using supervised learning," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 24, no. 1, pp. 28–35, Jan. 2016, doi: 10.1109/TNSRE.2015.2441835.

[92] A. B. L. Tort, R. Komorowski, H. Eichenbaum, and N. Kopell, "Measuring phase-amplitude coupling between neuronal oscillations of different frequencies," *J Neurophysiol*, vol. 104, no. 2, pp. 1195–1210, 2010, doi: 10.1152/JN.00106.2010.

[93] M. P. Hosseini, A. Hosseini, and K. Ahi, "A Review on Machine Learning for EEG Signal Processing in Bioengineering," *IEEE Rev Biomed Eng*, vol. 14, pp. 204–218, 2021, doi: 10.1109/RBME.2020.2969915.

[94] H. Marzbani, H. R. Marateb, and M. Mansourian, "Neurofeedback: A Comprehensive Review on System Design, Methodology and Clinical Applications," *Basic Clin*

Neurosci, vol. 7, no. 2, p. 143, Mar. 2016, doi: 10.15412/J.BCN.03070208.

[95] T. Zhang, J. Zhou, P. R. Carney, and H. Jiang, "Towards real-time detection of seizures in awake rats with GPU-accelerated diffuse optical tomography," *J Neurosci Methods*, vol. 240, pp. 28–36, Jan. 2015, doi: 10.1016/J.JNEUMETH.2014.10.018.

[96] Z. Juhasz and G. Kozmann, "A GPU-based simultaneous real-time EEG processing and visualization system for brain imaging applications," *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015 - Proceedings*, pp. 299–304, Jul. 2015, doi: 10.1109/MIPRO.2015.7160283.

[97] D. Lehmann *et al.*, "EEG microstate duration and syntax in acute, medication-naïve, first-episode schizophrenia: a multi-center study," *Psychiatry Res Neuroimaging*, vol. 138, no. 2, pp. 141–156, Feb. 2005, doi: 10.1016/J.PSCYCHRESNS.2004.05.007.

[98] S. Wong *et al.*, "EEG datasets for seizure detection and prediction— A review," *Epilepsia Open*, vol. 8, no. 2, pp. 252–267, Jun. 2023, doi: 10.1002/EPI4.12704.

[99] T. Zhang, W. Zheng, Z. Cui, Y. Zong, and Y. Li, "Spatial-Temporal Recurrent Neural Network for Emotion Recognition," *IEEE Trans Cybern*, vol. 49, no. 3, pp. 939–947, Mar. 2019, doi: 10.1109/TCYB.2017.2788081.

[100] M. R. Safari, R. Shalbaf, S. Bagherzadeh, and A. Shalbaf, "Classification of mental workload with EEG analysis by using effective connectivity and a hybrid model of CNN and LSTM," *Comput Methods Biomech Biomed Engin*, 2024, doi: 10.1080/10255842.2024.2386325.

[101] G. Xu, T. Ren, Y. Chen, and W. Che, "A One-Dimensional CNN-LSTM Model for Epileptic Seizure Recognition Using EEG Signal Analysis," *Front Neurosci*, vol. 14, p. 578126, Dec. 2020, doi: 10.3389/FNINS.2020.578126/BIBTEX.

[102] A. Kumar Singh, S. Soni, R. Singh Rana -, D. Sinha, K. Sasirekha, and K. Thangavel, "GPU based epileptic seizure detection using deep autoencoder with particle swarm optimization," *J Phys Conf Ser*, vol. 2318, no. 1, p. 012010, Aug. 2022, doi: 10.1088/1742-6596/2318/1/012010.

[103] Y. He *et al.*, "Classification of attention deficit/hyperactivity disorder based on EEG signals using a EEG-Transformer model∗," *J Neural Eng*, vol. 20, no. 5, Oct. 2023, doi: 10.1088/1741-2552/ACF7F5.

[104] M. Khatwani, M. Hosseini, H. Paneliya, T. Mohsenin, W. D. Hairston, and N. Waytowich, "Energy Efficient Convolutional Neural Networks for EEG Artifact Detection," *2018 IEEE Biomedical Circuits and Systems Conference, BioCAS 2018 - Proceedings*, Dec. 2018, doi: 10.1109/BIOCAS.2018.8584791.

[105] D. W. Stashuk, "Decomposition and quantitative analysis of clinical electromyographic signals," *Med Eng Phys*, vol. 21, no. 6–7, pp. 389–404, Jul. 1999, doi: 10.1016/S1350-4533(99)00064-8.

[106] K. Morita, T. Mizuno, and H. Kusuhara, "Decomposition profile data analysis of multiple drug effects identifies endoplasmic reticulum stress-inducing ability as an unrecognized factor," *Sci Rep*, vol. 10, no. 1, Dec. 2020, doi: 10.1038/S41598-020-70140-9.

[107] B. Zhu, S. Ma, R. Xie, J. Chevallier, and Y. M. Wei, "Hilbert Spectra and Empirical Mode Decomposition: A Multiscale Event Analysis Method to Detect the Impact of

Economic Crises on the European Carbon Market," *Comput Econ*, vol. 52, no. 1, pp. 105–121, Jun. 2018, doi: 10.1007/S10614-017-9664-X/METRICS.

[108] Y. Bai, M. Peng, and M. Wang, "A River Water Quality Prediction Method Based on Dual Signal Decomposition and Deep Learning," *Water (Switzerland)*, vol. 16, no. 21, p. 3099, Nov. 2024, doi: 10.3390/W16213099/S1.

[109] N. E. Huang and Z. Wu, "A review on Hilbert-Huang transform: Method and its applications to geophysical studies," *Reviews of Geophysics*, vol. 46, no. 2, Jun. 2008, doi: 10.1029/2007RG000228.

[110] 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.

[111] R. P. Kleihorst, R. L. Lagendijk, and J. Biemond, "Noise Reduction of Image Sequences Using Motion Compensation and Signal Decomposition," *IEEE Transactions on Image Processing*, vol. 4, no. 3, pp. 274–284, 1995, doi: 10.1109/83.366476.

[112] M. Srivastava, C. L. Anderson, and J. H. Freed, "A New Wavelet Denoising Method for Selecting Decomposition Levels and Noise Thresholds," *IEEE Access*, vol. 4, pp. 3862–3877, 2016, doi: 10.1109/ACCESS.2016.2587581.

[113] S. Krishnan and Y. Athavale, "Trends in biomedical signal feature extraction," *Biomed Signal Process Control*, vol. 43, pp. 41–63, May 2018, doi: 10.1016/J.BSPC.2018.02.008.

[114] K. Wang, C. H. Lee, and B. H. Juang, "Selective feature extraction via signal decomposition," *IEEE Signal Process Lett*, vol. 4, no. 1, pp. 8–11, 1997, doi: 10.1109/97.551687.

[115] J. Ma, T. Zhang, and M. Dong, "A novel ECG data compression method using adaptive fourier decomposition with security guarantee in e-health applications," *IEEE J Biomed Health Inform*, vol. 19, no. 3, pp. 986–994, May 2015, doi: 10.1109/JBHI.2014.2357841.

[116] J. J. Wei, C. J. Chang, N. K. Chou, and G. J. Jan, "ECG data compression using truncated singular value decomposition," *IEEE Transactions on Information Technology in Biomedicine*, vol. 5, no. 4, pp. 290–299, Dec. 2001, doi: 10.1109/4233.966104.

[117] K. S. Chan, J. Li, W. Eichinger, and E. Bai, "A new physics-based method for detecting weak nuclear signals via spectral decomposition," *Nucl Instrum Methods Phys Res A*, vol. 667, pp. 16–25, Mar. 2012, doi: 10.1016/J.NIMA.2011.11.067.

[118] M. A. Delsuc and P. O'Connor, "The Fourier transform in analytical science," *Nature Reviews Methods Primers 2024 4:1*, vol. 4, no. 1, pp. 1–16, Jul. 2024, doi: 10.1038/s43586-024-00326-2.

[119] H. K. Kwok and D. L. Jones, "Improved instantaneous frequency estimation using an adaptive short-time Fourier transform," *IEEE Transactions on Signal Processing*, vol. 48, no. 10, pp. 2964–2972, Oct. 2000, doi: 10.1109/78.869059.

[120] V. Pukhova, E. Gorelova, G. Ferrini, and S. Burnasheva, "Time-frequency representation of signals by wavelet transform," *Proceedings of the 2017 IEEE Russia*

Section Young Researchers in Electrical and Electronic Engineering Conference, ElConRus 2017, pp. 715–718, Apr. 2017, doi: 10.1109/EICONRUS.2017.7910658.

[121] 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.

[122] L. Zhukov, D. Weinsfein, and C. Johnson, "Independent component analysis for EEG source localization an algorithm that reduces the complexity of localizing multiple neural sources," *IEEE Engineering in Medicine and Biology Magazine*, vol. 19, no. 3, pp. 87–96, 2000, doi: 10.1109/51.844386.

[123] C. Q. Lai, H. Ibrahim, M. Z. Abdullah, J. M. Abdullah, S. A. Suandi, and A. Azman, "Artifacts and noise removal for electroencephalogram (EEG): A literature review," *ISCAIE 2018 - 2018 IEEE Symposium on Computer Applications and Industrial Electronics*, pp. 326–332, Jul. 2018, doi: 10.1109/ISCAIE.2018.8405493.

[124] M. A. Kabir and C. Shahnaz, "Denoising of ECG signals based on noise reduction algorithms in EMD and wavelet domains," *Biomed Signal Process Control*, vol. 7, no. 5, pp. 481–489, Sep. 2012, doi: 10.1016/J.BSPC.2011.11.003.

[125] C. G. Thomas, R. A. Harshman, and R. S. Menon, "Noise Reduction in BOLD-Based fMRI Using Component Analysis," *Neuroimage*, vol. 17, no. 3, pp. 1521–1537, Nov. 2002, doi: 10.1006/NIMG.2002.1200.

[126] A. N. Akansu, P. Duhamel, X. Lin, and M. De Courville, "Orthogonal transmultiplexers in communication," *IEEE Transactions on Signal Processing*, vol. 46, no. 4, pp. 979–995, Apr. 1998, doi: 10.1109/78.668551.

[127] "Wavelet, Subband and Block Transforms in Communications and Multimedia," *Wavelet, Subband and Block Transforms in Communications and Multimedia*, 2002, doi: 10.1007/B117716.

[128] D. Wu, W. P. Zhu, and M. N. S. Swamy, "A compressive sensing method for noise reduction of speech and audio signals," *Midwest Symposium on Circuits and Systems*, 2011, doi: 10.1109/MWSCAS.2011.6026662.

[129] J. F. Gemmeke, H. Van Hamme, B. Cranen, and L. Boves, "Compressive sensing for missing data imputation in noise robust speech recognition," *IEEE Journal on Selected Topics in Signal Processing*, vol. 4, no. 2, pp. 272–287, Apr. 2010, doi: 10.1109/JSTSP.2009.2039171.

[130] M. Qiao, A. H. Sung, and Q. Liu, "MP3 audio steganalysis," *Inf Sci (N Y)*, vol. 231, pp. 123–134, May 2013, doi: 10.1016/J.INS.2012.10.013.

[131] F. Li, B. Zhang, S. Verma, and K. J. Marfurt, "Seismic signal denoising using thresholded variational mode decomposition," *Exploration Geophysics*, vol. 49, no. 4, pp. 450–461, 2018, doi: 10.1071/EG17004.

[132] J. Tian *et al.*, "A novel identification method of microseismic events based on empirical mode decomposition and artificial neural network features," *J Appl Geophy*, vol. 222, p. 105329, Mar. 2024, doi: 10.1016/J.JAPPGEO.2024.105329.

[133] S. Maranò, C. Reller, H. A. Loeliger, and D. Fäh, "Seismic waves estimation and wavefield decomposition: Application to ambient vibrations," *Geophys J Int*, vol. 191, no. 1, pp. 175–188, Oct. 2012, doi: 10.1111/J.1365-246X.2012.05593.X/3/191-1-175-FIG014.JPEG.

[134] L. Zhang, D. Zhang, and W. Li, "Stock index trend analysis based on signal decomposition," *IEICE Trans Inf Syst*, vol. E97-D, no. 8, pp. 2187–2190, 2014, doi: 10.1587/TRANSINF.E97.D.2187.

[135] D. O. Afanasyev and E. A. Fedorova, "The long-term trends on the electricity markets: Comparison of empirical mode and wavelet decompositions," *Energy Econ*, vol. 56, pp. 432–442, May 2016, doi: 10.1016/J.ENECO.2016.04.009.

[136] B. E. Usevitch, "A tutorial on modern lossy wavelet image compression: Foundations of JPEG 2000," *IEEE Signal Process Mag*, vol. 18, no. 5, pp. 22–35, 2001, doi: 10.1109/79.952803.

[137] S. Zafar, Y. Q. Zhang, and B. Jabbari, "Multiscale Video Representation Using Multiresolution Motion Compensation and Wavelet Decomposition," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 1, pp. 24–35, 1993, doi: 10.1109/49.210541.

[138] Z. Černeková, C. Kotropoulos, and I. Pitas, "Video shot segmentation using singular value decomposition," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 3, pp. 181–184, 2003, doi: 10.1109/ICASSP.2003.1199137.

[139] T. Yu, K. Akhmadeev, E. Le Carpentier, Y. Aoustin, and D. Farina, "On-line recursive decomposition of intramuscular emg signals using gpu-implemented bayesian filtering," *IEEE Trans Biomed Eng*, vol. 67, no. 6, pp. 1806–1818, Jun. 2020, doi: 10.1109/TBME.2019.2948397.

[140] P. Karas and D. Svoboda, "Convolution of large 3D images on GPU and its decomposition," *EURASIP Journal on Advances in Signal Processing 2011 2011:1*, vol. 2011, no. 1, pp. 1–12, Nov. 2011, doi: 10.1186/1687-6180-2011-120.

[141] 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.

[142] M. Torbol, "Real-Time Frequency-Domain Decomposition for Structural Health Monitoring Using General-Purpose Graphic Processing Unit," *Computer-Aided Civil and Infrastructure Engineering*, vol. 29, no. 9, pp. 689–702, Oct. 2014, doi: 10.1111/MICE.12097.

[143] J. Bin, M. Kang, and Z. Liu, "GPU-Accelerated Tensor Decomposition for Moving Object Detection from Multimodal Imaging," *Proceedings of IEEE Sensors*, vol. 2020-October, Oct. 2020, doi: 10.1109/SENSORS47125.2020.9278924.

[144] J. B. J. Fourier, "The analytical theory of heat," *The Analytical Theory of Heat*, pp. 1–466, Jan. 2009, doi: 10.1017/CBO9780511693205.

[145] D. Gabor, "Theory of communication. Part 1: The analysis of information," *Journal of the Institution of Electrical Engineers - Part III: Radio and Communication Engineering*, vol. 93, no. 26, pp. 429–441, Nov. 1946, doi: 10.1049/JI-3-2.1946.0074.

[146] P. Goupillaud, A. Grossmann, and J. Morlet, "Cycle-octave and related transforms in seismic signal analysis," *Geoexploration*, vol. 23, no. 1, pp. 85–102, Oct. 1984, doi: 10.1016/0016-7142(84)90025-5.

[147] A. Muñoz, R. Ertlé, and M. Unser, "Continuous wavelet transform with arbitrary scales and O(N) complexity," *Signal Processing*, vol. 82, no. 5, pp. 749–757, May

2002, doi: 10.1016/S0165-1684(02)00140-8.

[148] S. Mallat, "A Wavelet Tour of Signal Processing: The Sparse Way," *A Wavelet Tour of Signal Processing: The Sparse Way*, pp. 1–805, Dec. 2008, doi: 10.1016/B978-0-12-374370-1.X0001-8.

[149] N. E. Huang, Z. Wu, S. R. Long, K. C. Arnold, X. Chen, and K. Blank, "On Instantaneous Frequency," *Advances in Data Science and Adaptive Analysis*, vol. 1, no. 2, pp. 177–229, Apr. 2009, doi: 10.1142/S1793536909000096.

[150] 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.

[151] 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.

[152] N. Rehman and D. P. Mandic, "Multivariate empirical mode decomposition," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 466, no. 2117, pp. 1291–1302, May 2010, doi: 10.1098/RSPA.2009.0502.

[153] D. P. Mandic, N. Ur Rehman, Z. Wu, and N. E. Huang, "Empirical mode decomposition-based time-frequency analysis of multivariate signals: The power of adaptive data analysis," *IEEE Signal Process Mag*, vol. 30, no. 6, pp. 74–86, 2013, doi: 10.1109/MSP.2013.2267931.

[154] N. Ur Rehman and D. P. Mandic, "Filter bank property of multivariate empirical mode decomposition," *IEEE Transactions on Signal Processing*, vol. 59, no. 5, pp. 2421–2426, May 2011, doi: 10.1109/TSP.2011.2106779.

[155] E. C. Cherry, "Some Experiments on the Recognition of Speech, with One and with Two Ears," *J Acoust Soc Am*, vol. 25, no. 5, pp. 975–979, Sep. 1953, doi: 10.1121/1.1907229.

[156] Bernard. Ans, Jeanny. Herault, and Christian. Jutten, "Adaptive neural architectures: detection of primitives," in *In Proc. of COGNITIVA'85*, Paris, 1985, pp. 803–851.

[157] Y. Bar-Ness, J. Carlin, and M. Steinberger, "Bootstrapping adaptive interference cancelers - Some practical limitations," 1982.

[158] J. Herault and C. Jutten, "Space or time adaptive signal processing by neural network models," *AIP Conf Proc*, vol. 151, no. 1, pp. 206–211, Aug. 1986, doi: 10.1063/1.36258.

[159] P. Comon, "Separation Of Stochastic Processes," *Workshop on Higher-Order Spectral Analysis*, pp. 174–179, 1989, doi: 10.1109/HOSA.1989.735291.

[160] C. Jutten and J. Herault, "Blind separation of sources, part I: An adaptive algorithm based on neuromimetic architecture," *Signal Processing*, vol. 24, no. 1, pp. 1–10, Jul. 1991, doi: 10.1016/0165-1684(91)90079-X.

[161] P. Comon, C. Jutten, and J. Herault, "Blind separation of sources, part II: Problems statement," *Signal Processing*, vol. 24, no. 1, pp. 11–20, Jul. 1991, doi: 10.1016/0165-1684(91)90080-3.

[162] E. Sorouchyari, "Blind separation of sources, part III: Stability analysis," *Signal Processing*, vol. 24, no. 1, pp. 21–29, Jul. 1991, doi: 10.1016/0165-1684(91)90081-S.

[163] P. Comon, “Independent component analysis, A new concept?,” *Signal Processing*, vol. 36, no. 3, pp. 287–314, Apr. 1994, doi: 10.1016/0165-1684(94)90029-9.

[164] J. F. Cardoso and A. Souloumiac, “Blind beamforming for non-gaussian signals,” *IEE Proceedings, Part F: Radar and Signal Processing*, vol. 140, no. 6, pp. 362–370, 1993, doi: 10.1049/IP-F-2.1993.0054.

[165] J. F. Cardoso and A. Souloumiac, “Jacobi Angles for Simultaneous Diagonalization,” *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 1, pp. 161–164, Jan. 1996, doi: 10.1137/S0895479893259546.

[166] J. F. Cardoso and B. H. Laheld, “Equivariant adaptive source separation,” *IEEE Transactions on Signal Processing*, vol. 44, no. 12, pp. 3017–3030, 1996, doi: 10.1109/78.553476.

[167] A. Belouchrani, K. Abed-Meraim, J. F. Cardoso, and E. Moulines, “A blind source separation technique using second-order statistics,” *IEEE Transactions on Signal Processing*, vol. 45, no. 2, pp. 434–444, 1997, doi: 10.1109/78.554307.

[168] J. Karhunen and J. Joutsensalo, “Representation and separation of signals using nonlinear PCA type learning,” *Neural Networks*, vol. 7, no. 1, pp. 113–127, Jan. 1994, doi: 10.1016/0893-6080(94)90060-4.

[169] J. Karhunen and J. Joutsensalo, “Generalizations of principal component analysis, optimization problems, and neural networks,” *Neural Networks*, vol. 8, no. 4, pp. 549–562, Jan. 1995, doi: 10.1016/0893-6080(94)00098-7.

[170] J. Karhunen, E. Oja, L. Wang, R. Vigário, and J. Joutsensalo, “A class of neural networks for independent component analysis,” *IEEE Trans Neural Netw*, vol. 8, no. 3, pp. 486–504, 1997, doi: 10.1109/72.572090.

[171] J. Karhunen, “Neural approaches to independent component analysis and source separation,” *The European Symposium on Artificial Neural Networks*, 1996.

[172] A. Hyvärinen and E. Oja, “A Fast Fixed-Point Algorithm for Independent Component Analysis,” *Neural Comput*, vol. 9, no. 7, pp. 1483–1492, Oct. 1997, doi: 10.1162/NECO.1997.9.7.1483.

[173] A. Hyvärinen, “Fast and robust fixed-point algorithms for independent component analysis,” *IEEE Trans Neural Netw*, vol. 10, no. 3, pp. 626–634, 1999, doi: 10.1109/72.761722.

[174] A. J. Bell and T. J. Sejnowski, “An information-maximization approach to blind separation and blind deconvolution.,” *Neural Comput*, vol. 7, no. 6, pp. 1129–1159, 1995, doi: 10.1162/NECO.1995.7.6.1129.

[175] 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.

[176] 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.

[177] 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.

[178] 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.

[179] 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.

[180] 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.

[181] 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.

[182] 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.

[183] 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.

[184] A. Delorme and S. Makeig, "EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics including independent component analysis," *J Neurosci Methods*, vol. 134, no. 1, pp. 9–21, Mar. 2004, doi: 10.1016/J.JNEUMETH.2003.10.009.

[185] Patrick Flandrin, "Empirical Mode Decomposition MATLAB Implementations." Accessed: Oct. 22, 2024. [Online]. Available: https://perso.ens-lyon.fr/patrick.flandrin/emd.html

[186] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, doi: 10.1145/1498765.1498785/SUPPL_FILE/CACM_ROOFLINE_APPENDIX_A.PDF.

[187] K. Al-Subari, S. Al-Baddai, A. M. Tomé, M. Goldhacker, R. Faltermeier, and E. W. Lang, "EMDLAB: A toolbox for analysis of single-trial EEG dynamics using empirical mode decomposition," *J Neurosci Methods*, vol. 253, pp. 193–205, Sep. 2015, doi: 10.1016/J.JNEUMETH.2015.06.020.

[188] A. Delorme and S. Makeig, "EEGLAB: An open source toolbox for analysis of single-trial EEG dynamics including independent component analysis," *J Neurosci Methods*, vol. 134, no. 1, pp. 9–21, Mar. 2004, doi: 10.1016/j.jneumeth.2003.10.009.

[189] J. H. Halton and G. B. Smith, "Algorithm 247: Radical-inverse quasi-random point sequence," *Commun ACM*, vol. 7, no. 12, pp. 701–702, Dec. 1964, doi: 10.1145/355588.365104.

[190] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 127–136, Jan. 2010, doi:

10.1145/1837853.1693472.
[191] A. Delorme, T. Sejnowski, and S. Makeig, "Enhanced detection of artifacts in EEG data using higher-order statistics and independent component analysis," *Neuroimage*, vol. 34, no. 4, pp. 1443–1449, Feb. 2007, doi: 10.1016/J.NEUROIMAGE.2006.11.004.
[192] D. B. Keith, C. C. Hoge, R. M. Frank, and A. D. Malony, "Parallel ICA methods for EEG neuroimaging," *20th International Parallel and Distributed Processing Symposium, IPDPS 2006*, vol. 2006, 2006, doi: 10.1109/IPDPS.2006.1639299.
[193] H. Du, H. Qi, and X. Wang, "A parallel independent component analysis algorithm," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, vol. 1, pp. 151–158, 2006, doi: 10.1109/ICPADS.2006.17.
[194] M. Plauth, F. Feinbube, P. Troger, and A. Polze, "Fast ICA on Modern GPU Architectures," *International Conference on Parallel and Distributed Computing: Applications and Technologies*, vol. 2015-July, pp. 69–75, Jul. 2014, doi: 10.1109/PDCAT.2014.19.
[195] T. N. Kumara, H. Gamaarachchi, G. Prathap, and R. Ragel, "Generalized and hybrid fast-ICA implementation using GPU," *16th International Conference on Advances in ICT for Emerging Regions, ICTer 2016 - Conference Proceedings*, pp. 13–20, Jan. 2017, doi: 10.1109/ICTER.2016.7829893.
[196] G. Benko and Z. Juhasz, "GPU implementation of the FastICA algorithm," *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019 - Proceedings*, pp. 196–199, May 2019, doi: 10.23919/MIPRO.2019.8757036.
[197] R. Ramalho, P. Tomás, and L. Sousa, "Efficient independent component analysis on a GPU," *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, pp. 1128–1133, 2010, doi: 10.1109/CIT.2010.205.
[198] F. Raimondo, J. E. Kamienkowski, M. Sigman, and D. Fernandez Slezak, "CUDAICA: GPU optimization of Infomax-ICA EEG analysis," *Comput Intell Neurosci*, vol. 2012, 2012, doi: 10.1155/2012/206972.
[199] NVIDIA GPU Teaching Kit, "Module 5.1: Thread Execusion Efficiency." Accessed: Feb. 11, 2025. [Online]. Available: https://engineering.purdue.edu/~smidkiff/ece563/NVidiaGPUTeachingToolkit/Mod5/Lecture-5-1-warps-simd.pdf

APPENDIX

Variable name	Dimensions	Used in kernel function
d_current	SignalLength × NumNoise	findExtremaShfl(), selectExtrema(), updateRealizations()
d_noisedSignal		produceFirstIMF() addNoise()
d_whiteNoise		curandGenerateNormal(), addNoise()
d_whiteNoiseModes	SignalLength × NumNoise × NumIMFs	addNoise(), updateRealizations(), updateSignal()
d_channelMeans	NumNoise	mean(), multiply()
d_channelVariance	NumNoise	variance(), multiply(), addNoise()
d_sparseFlag	SignalLength × NumNoise	findExtremaShfl(), DeviceScanInitKernel(), DeviceScanKernel(), selectExtrema()
d_noisedSignalIndex	SignalLength × NumNoise	findExtremaShfl(),selectExtrema(), interpolate()
d_ScanResult	SignalLength × NumNoise	scanLargeDeviceArray(), scanSmallDeviceArray(), selectExtrema()
d_compactValue d_compactIndex/	SignalLength × NumNoise	selectExtrema(), setBoundary(), tridiagonalSetup(), splineCoefficients(), interpolate()
d_num_extrema_max/ d_num_extrema_min	NumNoise	selectExtrema(), tridiagonalSetup(), splineCoefficients(), interpolate(), averageUppperLower(), siftingCriterion()
d_upperDia d_middleDia d_lowerDia d_right	SignalLength × NumNoise	preSetTridiagonalMatrix(), tridiagonalSetup() cusparseSgtsv2(), splineCoefficients(), interpolate()
d_meanEnvelope		averageUppperLower(), updateRealizations()
d_envelopeVauleMax/ d_envelopeVauleMin		averageUppperLower(), interpolate(), produceSX()
d_residue		produceResidue(), averageUpdateSignal()
d_forNext	SignalLength	averageUpdateSignal(), produceFirstIMF(), addNoise()
d_running	SignalLength	produceFirstIMF()
d_IMFs	SignalLength × NumIMFs	averageUpdateSignal(), produceFirstIMF()

Table S1: Device variables used in the GPU implementation, their size and kernels in which they are referenced.

Number of Sifting Iterations	Signal Length	Number of realizations				
		500	400	300	200	100
10	10241	22.9	18.4	18.7	18.5	18.1
	20481	40.4	33.7	32.9	31.8	31.0
	30721	46.5	48.7	48.4	46.0	46.2
	40961	62.4	65.2	64.4	64.4	60.3
	51201	82.2	113.5	86.2	83.4	84.7
	61441	96.8	100.5	102.1	102.6	98.6
	71681	118.7	122.0	117.6	117.7	116.8
	81921	162.4	151.1	151.0	150.7	142.8
	92161	194.6	169.8	181.1	166.9	161.3
	102401	265.2	215.7	190.5	207.7	196.1
20	10241	15.9	11.9	11.9	11.9	11.1
	20481	27.5	19.9	19.5	19.9	18.5
	30721	31.3	29.2	28.9	29.1	27.9
	40961	41.2	39.1	36.9	37.3	35.6
	51201	53.3	50.3	49.3	47.8	48.1
	61441	62.4	58.6	57.4	56.6	53.8
	71681	77.5	65.7	66.7	67.4	63.5
	81921	97.9	90.8	81.5	81.3	77.5
	92161	113.5	92.8	92.9	102.7	90.0
	102401	129.2	111.2	124.7	106.8	105.4
50	10241	7.3	6.6	6.4	6.5	7.0
	20481	15.0	11.5	11.1	11.0	12.3
	30721	17.9	17.7	17.8	17.2	16.4
	40961	23.4	22.4	22.9	21.0	22.2
	51201	31.2	30.0	30.7	30.4	28.5
	61441	33.6	32.4	34.7	32.6	31.5
	71681	34.5	37.8	36.4	35.3	37.0
	81921	49.3	46.0	46.1	44.2	43.3
	92161	54.9	53.8	47.4	51.1	49.4
	102401	63.6	66.2	64.6	54.8	53.4

Table S2: Speedup values for different test configurations (V100 vs MATLAB).

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

	averageDirection()	direction vectors Generate the multivariate mean envelope of input signal
Signal updating	updateSignal()	Subtract the mean envelope to generate new input signal

Table S3: CUDA kernels used in different stages in the MEMD GPU implementation.

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

Table S4: Device variables used in the GPU implementations, their size and kernels in which they are referenced