Massively Parallel Neural Signal Processing on a Many-Core Platform

Although the ensemble empirical mode decomposition (EEMD) method and Hilbert-Huang transform (HHT) offer an unrivaled opportunity to understand neural signals, the EEMD algorithm’s complexity and neural signals’ massive size have hampered EEMD application. However, a new approach using a many-core platform has proven both efficient and effective for massively parallel neural signal processing.

The past decade has brought remarkable advances in the processing of neural signals—that is, neural activity signatures such as electroencephalogram (EEG), magnetoencephalography (MEG), electrocorticography (ECoG), magnetic resonance imaging (MRI), and functional MRI (fMRI). Neural signal analysis is vital in detecting, diagnosing, and treating brain disorders and related diseases.

Neural signals are naturally nonlinear and nonstationary. Prior to the introduction of nonlinear methods, linear algorithms—such as short-Fourier transform, Wigner-Ville distribution, and wavelet filtering—had been extensively used in spectral analysis of EEG recordings. A major shortfall of linear approaches is that neural signals’ temporal patterns, such as instantaneous amplitude and phase or frequency, can’t be accurately estimated. The recent advent of ensemble empirical mode decomposition (EEMD), in conjunction with Hilbert-Huang transform (HHT), has revolutionized the study of neural signals, giving neuroscientists an unrivaled opportunity to understand the true physical and neurophysiological meanings of neural signals. However, the EEMD algorithm’s complexity and neural signals’ routinely massive size hamper EEMD application in neural science research and practice.

Here, we present an approach to this problem that uses a many-core platform (a GPU for home entertainment) that enables massively parallel neural signal processing. By analyzing a multichannel EEG of absence seizures, we demonstrate our approach’s efficiency and effectiveness in identifying the seizure state.

Why EEMD?
EEMD breaks down a complicated nonlinear and nonstationary signal without a basis function—such as sine or wavelet functions—into a collection of oscillatory intrinsic mode functions (IMFs) that are embedded in the signal and reflect the signal’s true physical characteristics. To do this, EEMD adds white noise to the signal under investigation to form an ensemble of noise-added signals and takes full advantage of white noise’s statistical
characteristics to perturb the signal in its true solution neighborhood, and to cancel itself out after serving its purpose. EEMD eliminates mode mixing in all cases automatically and excels in resistance to noise, which makes EEMD far superior to its linear counterparts in processing neural signals.

On the other hand, as we describe later, the EEMD algorithm demands repetitively processing many trials of the noise-added signal in an ensemble, and the outputs’ precision depends on the number of trials, which should be large enough to neutralize the effect of added noises. Furthermore, as experimental techniques for recording neural activities have been advancing quickly—rapidly increasing the number of channels (electrodes) and sampling frequencies—neural signals’ density and the spatial scale have been increasing exponentially. Neural signal analysis with EEMD manifests a problem that’s not only data intensive, but also highly compute intensive. For example, it typically takes a few weeks to process one hour’s EEG data using a high-end desktop computer.

Neural signals have high temporal and spatial parallelism, while the EEMD algorithm has built-in parallelism. As researchers have noted, parallelism is the future of computing. The key to effectively improving a compute- and data-intensive application’s performance lies in appropriately using the potential parallelism in various forms embedded in the target application. It’s a natural choice to develop a parallel EEMD on a high-performance computing (HPC) architecture to fulfill the performance requirement by using the multidimensional parallelism in an application of neural signal analysis. For more information on this, see the sidebar, “Related Work in HPC Research and GPGPU Computing.”

Many-core computing platforms—GPUs—provide massively parallel computing platforms. We developed parallel EEMD approaches that successfully adopt general-purpose computing on the GPU (GPGPU), a state-of-the-art technique empowered by a many-core computing platform. Our approaches properly parallelize the EEMD application into numerous concurrent fine-grained sub-tasks, each of which handles a small piece of neural signal (time series). The proposed approaches have been developed using Nvidia’s CUDA, a general-purpose parallel computing architecture. The intensive computation involved in the whole problem space is then mapped onto numerous threads evenly executed by the many parallel GPU cores.

Here, we present a case study using EEMD and HHT to analyze an EEG dataset generated by an epilepsy patient with absence seizures. We first decomposed the EEG dataset using EEMD into a number of IMFs. We then calculated the Hilbert-Huang Spectral Entropy (HHSE). These efforts aimed to explore the feasibility of distinguishing seizure states based on the theory of the HHT and EEMD approach. Experimental results show that HHSE is highly suited for detecting seizure states. After that, we carried out a series of performance studies on the GPGPU-enabled parallel approaches. For a one-hour EEG segment, the parallel approach can complete analysis in less than an hour; the original serial counterpart requires hundreds of hours to handle the same task.

**Neural Signal Structure**

The neural signals we investigated are a massive EEG dataset continuously captured—using multiple simultaneous electrodes—from an epilepsy patient over several hours. As a result, the obtained EEG dataset persist as a multichannel time series. The volume of the EEG dataset prior to analysis scales in three dimensions: the time ($T$), the number of channels ($NC$), and the sampling rate ($SR$). The total number of variables in the multichannel time series is $T \times NC \times SR$.

When processing the source EEG dataset with EEMD, we apply a sliding time window to each individual EEG time series, corresponding to one channel, to generate numerous short time series (epochs) covered by the windows. There’s an overlap of a fixed length between two consecutive epochs. We write the ratio of the overlap’s length against an epoch’s length as $\rho$. Clearly, the size of data in the form of epochs “swells” to $1/(1 - \rho)$ of the source EEG dataset. Obviously, the number of epochs that might be extracted from a source time series is proportional to $T$.

In the initial stage of an EEMD method, each epoch will be repeatedly added with white noise to generate an “ensemble” of trials, where each trial represents an instance of the original epoch mixed with noise. A trial is the elementary data chunk computed by the EEMD algorithm. Typically, the number of trials ($NT$) in an ensemble should be in the hundreds. After initialization,
Massively Parallel Computing Using GPGPU

A modern GPU possesses single-instruction multiple-data (SIMD) cores, high on-chip bandwidth, and explicit data transfers between fast local memories and external dynamics RAM. All of these establish GPU as the go-to solution when coping with data and compute-intensive problems.

GPGPU has recently boomed with the enhanced programmability of GPUs. Relatively high abstraction levels have emerged, and Nvidia’s Compute Unified Device Architecture (CUDA) is salient. The current CUDA fosters a software environment that allows programming on a GPU with a slightly extended version of C, so a CUDA-based application can execute on a GPU’s many cores in a massively parallel fashion.

A CUDA-based application defines C functions as “kernels” to be explicitly executed on the GPU, or device. Usually, the kernels are invoked from the CPU (“host”) application’s main executable, which will be executed for \(N\) times via \(N\) individual CUDA threads. CUDA threads are grouped into blocks, and threads in a block can cooperate together by efficiently sharing data through a fast shared memory and by synchronizing their execution to coordinate memory access. Threads, memories, and synchronization are exposed to developers for fine-grained data and thread parallelism—such as dealing with a single float-type variable per thread. A developer should partition the problem into a set of fine-grained subtasks, which can be mapped to CUDA threads accordingly.

### References


and immediately before being fed to the EEMD algorithm, the data’s size in the form of trials (see Figure 1a) explodes by hundreds of times over that of the source EEG dataset—that is, in terabytes.

As Figure 1b shows, the formation of the initialized EEG dataset can be viewed as a three-level hierarchy. Each EEG data channel (time series) can be singled out from the whole multichannel dataset at the top level. At the middle level, many epochs can be generated from an EEG time series by applying a sliding time window. At the bottom level, an epoch spawns numerous trials by adding white noise to itself.
Hilbert-Huang Transform and EEMD

As Figure 2 shows, for a nonstationary signal \( x(t) \), the EEMD method can decompose the signal into a series of intrinsic mode functions (IMFs), \( \text{imf}_i (i = 1, 2, \ldots, N) \), where \( N \) is the number of IMFs.\(^5\) The signal \( x(t) \) can be written

\[
x(t) = \sum_{i=1}^{n-1} \text{imf}_i (t) + r_n (t).
\]  

(1)

Applying the Hilbert transform and the concept of Shannon entropy, we can construct the Hilbert-Huang spectral entropy (HHSE), which denotes a more accurate and nearly continuous distribution of the signal's energy. IMFs and HHSE precisely characterize the true physical nature of signal \( x(t) \). More in-depth discussions on EEMD and HHT are available elsewhere.\(^2,5\)

Given the size of a time window and the overlap of two consecutive windows, the whole procedure for processing an epoch of signal in a time window, \( x(t) \), occurs in several steps. We first need to decompose a signal to obtain its true IMFs prior to calculating its HHSE; we do this in eight steps:

- **Step 1.** Calculate the number of IMFs in \( x(t) \): \( NT = \log_2(\text{length}(x(t))) - 1 \); set the amplitude of white noise to be added to \( x(t) \); set the number of trials in the ensemble: \( NT \), and set \( k = 1 \).
- **Step 2.** In processing the \( k \)th trial, set \( x_k(t) = x(t) + n_k(t) \); \( n_k(t) \) denotes the white noise; set \( i = 1 \); and initialize the residual signal \( r_i(t) = x_k(t) \).
- **Step 3.** After initiating the extraction of the \( i \)th (in the \( k \)th trial) IMF, set \( j = 1 \) and \( b_{j-1} = r_i(t) \); calculate the local minima \( b_{max}(t) \) and maxima \( b_{min}(t) \) of \( b_{j-1}(t) \).
- **Step 4.** Interpolate \( b_{max}(t) \) and \( b_{min}(t) \) using cubic spline interpolation lines, thus to extract the upper and lower envelopes of \( b_{j-1}(t) \); eventually \( m_{j-1}(t) \)—the mean of the upper and lower envelopes—can be calculated as \( b_j(t) = b_{j-1}(t) - m_{j-1}(t) \).
- **Step 5.** Check if \( b_j(t) \) is an IMF; when the termination requirement is satisfied, Step 3 will be returned with \( j = j + 1 \). Otherwise, the \( i \)th IMF in the \( k \)th trial is obtained, \( \text{imf}_i[k](t) = b_j(t) \); then set \( r_{i+1}(t) = r_i(t) - \text{imf}_i[k](t) \) as the new residual signal for sifting the \( (i + 1) \)th possible IMF.
- **Step 6.** If \( r_{i+1}(t) \) still has at least two extrema, then Step 3 will be returned with \( i = i + 1 \); Otherwise, the decomposition procedure on the \( k \)th trial stops. Thus the noise-added signal \( x_k(t) \) can be decomposed as

\[
x_k(t) = \sum_{i=1}^{NT-1} \text{imf}_i[k](t) + r_{NT}(t),
\]

where \( r_{NT}(t) \) is the residual or trend of the \( x_k(t) \).
- **Step 7.** If \( k \) isn't greater than \( NTE \), Step 2 will be returned with \( k = k + 1 \); Otherwise, the preset trials have been completed; the \( i \)th IMF of \( x(t) \) is the average of the results in all the trials,

\[
\text{imf}_{i}[t] = \frac{\sum_{k=1}^{NT} \text{imf}_i[k](t)}{NT},
\]

where \( i = 1, 2, \ldots, NT \).
- **Step 8.** Based on \( \text{imf}_i(t), i = 1, 2, \ldots, NT \), we can eventually obtain the HHSE of \( x(t) \); Step 1 will then be returned to process the next epoch, if any.

We’ve measured the overhead of the serial implementation. The instructions for implementing the EEMD algorithm (Steps 2 through 6, known as an EEMD procedure) can consume approximately 99.8 percent of the overall execution time. Hence, we trust the bottleneck is with these EEMD procedures and focus on them here, tackling the bottleneck by parallelizing EEMD procedures.
Calculating HHSE on EEMD manifests a perfect paradigm of single instruction, multiple data (SIMD).

The performance improvement of parallel applications is determined by parallelism in various forms with the problem, the software environment, and the hardware platform (see Figure 3). We use the term degree of parallelism (DoP) to quantify the parallelisms with several fronts.

A problem’s DoP means the number of portions of it that can be concurrently solved and executed with the same results as those obtained in a serial manner. A problem’s parallelism might be spatial, temporal, or organizational, and might concern data or computations.

A software environment’s DoP denotes the number of concurrent tasks that the environment supports at the development stage (for example, you might set less than 500 threads in a thread pool over Windows Vista; see http://msdn.microsoft.com/en-us/magazine/cc163327.aspx).

A hardware platform’s DoP equals the number of processing units, which operate in parallel in a collaborative manner (for example, the number of CPUs/cores in a parallel computer, the number of processing nodes in a distributed system, and the number of cores in a GPU).

The grain of a problem determines the type of parallel/distributed systems suitable for solving the problem. For example, the Internet-based Grid computing platform typically handles coarse-grained problems, while high-end computer clusters are more suited to fine-grained ones. Multicore computers and many-core platforms (such as GPUs) often apply to the problems of extremely fine grains.

**Multilevel Parallelism**

The EEMD application’s parallelism can be characterized in at least two levels (see Figures 4a and 4b).

At the epoch level, the EEMD procedure for an epoch of time series is treated as a whole. The data in an epoch are input to the same EEMD procedure individually, and the outputs from any EEMD procedure instance won’t be consumed by another. The DoP equals the number of epochs, which increases linearly with the EEG dataset’s size.

At the trial level, a trial (a noise-added epoch) is treated as a whole. Given a number of trials per EEMD instance, each trial’s decomposition is always performed independently from the others. An original epoch’s IMFs (see Figure 2) are inferred only after computing the ensemble of trials. The DoP equals the number of trials per ensemble \( \times \) the number of epochs. A task at the trial level handles only a short time series (such as approximately 1,000 points). The instructions in steps 2 through 7 remain the same when processing any trials.

Even at the trial level, we might be able to further divide the computation into finer subtasks to exhaustively utilize a many-core platform, such as a GPU, with extremely fine-grained parallelism.

**Massively Parallel EEMD on GPGPU**

We developed two alternatives of massively parallel EEMD on top of CUDA by adapting to the trial-level parallelism.

**GPGPU-Aided EEMD**

We first considered using SG-EEMD, a single GPU to support a parallel EEMD. Figure 5a shows the execution task graph as a directed acyclic graph (DAG). The job of processing the
original signal (or any portions of the signal) can be mapped to a number (denoted by \( NW \) in the figure) of concurrent epoch-level tasks.

Moreover, each epoch-level task can be further decomposed and mapped to a number (denoted by \( NT \) in the figure) of trial-level subtasks, which are executed by an ensemble of CUDA threads. Figure 5b shows how an individual epoch-level task can be handled by SG-EEMD. Given that an epoch (\( \text{Epoch}[i] \)) consists of 1,024 data points, we produced a scheme that maps this task to CUDA threads in the following steps:

- **Step 1.** After initialization, the host invokes 1,024 CUDA threads evenly grouped as 8 blocks to calculate the standard deviation (\( \text{STD} \)) of the 1,024 points of the original epoch. The EEMD procedure then makes \( NT \) (no less than 100) duplicates of \( \text{Epoch}[i]/\text{STD} \) and adds white noise to each duplicate to generate an individual trial \( \text{Trial}[j] (j = 1, 2, \ldots, NT) \) in an ensemble.

- **Step 2.** To deal with \( NT \) trials, the host initializes \( NT \) thread blocks with 128 threads operating in each block. Each thread calculates only the extrema of 8 data points. Obviously, we exploit a parallelism here that's even finer than the trial level.

- **Step 3.** The host invokes \( NT \) blocks with only one thread per block to handle the calculation of envelope lines, which is compute intensive. This means \( NT \) cores on the GPU are assigned to these subtasks, each core computing one trial's envelope lines individually.

- **Step 4.** The host once again initializes \( NT \) thread blocks with 128 threads operating in each block. There are \( 128 \times NT \) CUDA threads working together to extract an IMF of \( NT \) trials. Steps 2 through 4 loop 10 times until all IMFs of the ensemble’s trials are completely extracted.

We have an adaptive CUDA thread setting. The cubic interpolation method (Steps 3 and 4) used to calculate envelope lines is complex and hard to parallelize, and is encapsulated in a single CUDA thread. Our design minimizes the thread number in a thread block,\(^9\) while it creates thread blocks—as many as possible—when executing threads of this type. Hence, these threads can make the most of CUDA cores to deal with intensive computations and occupy as much fast shared memory (manipulated by each block) as possible to buffer the intermediate data generated during their executions. On the contrary, we maximize the number of threads in each block for simple operations.
Computing in Science & Engineering

SG-EEMD adopts an execution mode of “single CPU thread, many CUDA (GPU) threads”—that is, a single CPU thread performs the executions from the point at which the neural data are fetched from storage to the point at which the HHSE is calculated. However, the overhead incurred by fetching data from storage is significant. Also, we want to explore the new EEMD approach’s scalability and the potential for using multiple GPU devices.

We therefore developed another GPGPU-aided EEMD—MG-EEMD—that has a “multiple CPU threads, many CUDA threads” design to hide the data-access overhead and exploit multiple GPU devices when possible. To achieve this, we used CPU-based multithreading and hybrid parallelization as follows.

First, to achieve CPU-based multithreading, \( \text{GPU}\_\text{Num} + 1 \) concurrent CPU threads are created at the host end, where \( \text{GPU}\_\text{Num} \) denotes the number of GPUs embedded in the host. An individual CPU thread is dedicated to loading data from the file system to the host’s main memory and initializing the epochs to be consumed by CUDA threads.

To achieve hybrid parallelization, the other \( \text{GPU}\_\text{Num} \) CPU threads manage the same number of GPUs (if there's more than one). Each of these CPU threads corresponds to an individual GPU device and assigns an exclusive channel of EEG data to the designated GPU device for processing. In other words, we enable parallelization at the data-channel level. In the context of each CPU thread, the corresponding GPU still processes the EEG time series at the trial level. Such a hybrid parallelization scheme maps the most coarse-grained tasks to GPU devices and the finest subtasks to CUDA threads.

**Experiments and Discussion**

We implemented the GPGPU-aided EEMDs using CUDA 2.3 and analyzed an EEG dataset with absence seizures (see Figure 6). Our testbed is a desktop computer equipped with an Nvidia GeForce GTX 295 video card, a last-generation card designed for gaming that has dual GPU chips with 240 CUDA cores and 896 Mbytes of Graphics Double Data Rate, version 3 (GDDR3) per GPU (for more information, see www.nvidia.com/object/product_geforce_gtx_295_us.html).

**Analysis Using G-EEMD and HHSE**

The EEG dataset was obtained from a patient with absence epilepsy. The EEG signals were recorded from scalp surface at F3, F4, C3, C4, O1, and O2 in the International 10–20 System using six electrodes. The onset and offset of absence
Seizures were determined by the start and end time of spike-wave discharges; the SWDs were defined as large-amplitude rhythmic 3–4 Hz discharges with typical SW morphology lasting less than a second. Our study investigated absence seizure by analyzing the EEG dataset via calculation of HHSE. Figure 7 shows a segment of six-channel 80-second-long EEG recordings covering the period of an absence seizure (the onset is at 63 seconds).

The parameter settings for the EEMD algorithm are epoch = 1,000, overlap = 750, white noise amplitude for an EEMD trial = 0.1 × standard deviation of an epoch, and 100 trials per ensemble. Figure 7b shows the Hilbert-Huang spectrum of the first channel, emphasizing two EEG segments: a seizure-free segment (from 0–4 s) and a seizure segment (from 64–68 s). We checked the results against those obtained using alternative EEMD implementations, and the differences are negligible.

The EEG recordings demonstrate a high-amplitude firing pattern during the seizure state. Figure 8a and 8b show the corresponding Hilbert-Huang spectrum and the marginal spectrum. The difference in spectrum distributions of the two EEG segments is apparent. In the absence seizure state, a significant increase of 3–4 Hz in activity can be observed.

The low-frequency band of 0–2 Hz and the frequency band over 30 Hz were removed to filter out artifacts to highlight the middle- and low-frequency brain activities, which make strong contributions during seizures. We then calculated HHSE within the band of 2–30 Hz. Figure 9 plots the HHSE time courses for all six channels of the EEG recordings. Clearly, the entropy values decrease significantly during the seizure state.
We began the performance test by analyzing the first 2,628.65 seconds of one channel of EEG data. Table 1 shows the execution times to process one- and six-channel EEG data segments and then the whole EEG dataset (24-hour recordings) using the GPGPU-aided EEMDs.

The SG-EEMD needs only 583 seconds to process one-channel EEG data of 2,628.65 seconds. When dealing with six-channel EEG data of the same length using one GPU, it can barely support the requirement of real-time processing. The MG-EEMD further reduces the execution time to 42 percent of the SG-EEMD’s. The result is reasonable because MG-EEMD can exploit two GPUs, and the MG-EEMD’s design takes advantage of the minimal synchronization overhead among multiple GPUs on the same host—in contrast to traditional parallel and distributed computing platforms. EEG data loading also incurs significant overhead in SG-EEMD application. MG-EEMD uses an additional CPU thread to load data, which operates simultaneously with the CUDA threads; it therefore effectively hides the overhead of accessing massive EEG data.

Our results also show that the performance of alternative EEMD implementations remain almost the same when the neural data under investigation scale up. In the third test, we compared the performance of SG-EEMD and MG-EEMD with the whole EEG dataset, and the results were exactly as expected.

The EEMD application is data intensive and has profound parallelism at multiple levels. The many-core computing platform provides massive computing capability for data- and compute-intensive applications.

The GPGPU-aided approaches exhibit excellent scalability when handling massive data from one channel to multiple channels. This is enabled by

- a proper mapping of the application’s data and computation parallelism to the parallelism with CUDA’s and GPUs, and
- the many-core architecture’s extensive scalability.

We can envision further performance improvements to EEMD applications using a host with more GPUs or a platform with more GPUs linked via a high-speed network such as Infiniband. We anticipate that, in the near future, a hybrid computing technology combining GPGPU and cutting-edge parallel and distributed computing will prevail in analyzing massive neural signals with rapidly increasing density and channels.

Table 1. Results of performance evaluation on GPGPU-aided EEMDs.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SG-EEMD*</td>
</tr>
<tr>
<td></td>
<td>MG-EEMD**</td>
</tr>
<tr>
<td>One-channel segment</td>
<td>583 seconds</td>
</tr>
<tr>
<td></td>
<td>—</td>
</tr>
<tr>
<td>Six-channel segment</td>
<td>3,473 seconds</td>
</tr>
<tr>
<td></td>
<td>1,488 seconds</td>
</tr>
<tr>
<td>Entire six-channel dataset</td>
<td>31.81 hours</td>
</tr>
<tr>
<td>(24-hour recordings)</td>
<td>13.62 hours</td>
</tr>
</tbody>
</table>

* An SG-EEMD is a single GPU that supports a parallel EEMD.
** An MG-EEMD uses multiple GPUs to support a parallel EEMD.
Talent of Creative Research Excellence Programme (Hebei, China), the Programme of High-Resolution Earth Observing System (China), and the Fundamental Research Funds for the Central Universities (CUG110608, CUG, Wuhan). Dan Chen gratefully acknowledges support from the Birmingham-Warwick Science City Research Alliance.

References

Dan Chen is a professor in the School of Computer Science, China University of Geosciences, Wuhan. His research interests include computer-based modeling and simulation, high-performance computing, and neuroinformatics. Chen has a PhD in computer engineering from Nanyang Technological University, Singapore. Contact him at chendan@pmail.ntu.edu.sg.

Lizhe Wang is a principal research engineer at Indiana University’s Pervasive Technology Institute. His research interests include high-performance computing, Grid computing and e-Science, and cloud computing. Wang has a Doctor of Engineering degree from the University of Karlsruhe, Germany. Contact him at lizhe.wang@gmail.com.

Gaoxiang Ouyang is a postdoctoral research fellow in the Institute of Electrical Engineering at Yanshan University, Qinhuangdao, China. His research interests include EEG analysis, neural models, and dynamic systems. Ouyang has a PhD in manufacturing engineering from the City University of Hong Kong. Contact him at ouyang@ysu.edu.cn.

Xiaoli Li is a professor and head of the Department of Automation at the Institute of Electrical Engineering, Yanshan University, Qinhuangdao, China. His research interests include computational intelligence, signal processing and data analysis, monitoring, bio-signal analysis, and manufacturing systems. Li has a PhD in mechanical engineering from Harbin Institute of Technology, China. Contact him at xlli@ysu.edu.cn.

Selected articles and columns from IEEE Computer Society publications are also available for free at http://ComputingNow.computer.org.

**SC11**
International Conference for High Performance Computing, Networking, Storage and Analysis

12-18 November 2011
Seattle, Washington, USA