IEICE TRANS. INF. & SYST., VOL.E94–D, NO.12 DECEMBER 2011

PAPER Special Section on Parallel and Distributed Computing and Networking

Evaluation of GPU-Based Empirical Mode Decomposition for Off-Line Analysis

Pulung WASKITO[†a)], ***Nonmember***, **Shinobu MIWA**[†], ***Member***, **Yasue MITSUKURA**[†], ***Nonmember***, ***and*** **Hironori NAKAJO**[†b)], ***Member***

SUMMARY In off-line analysis, the demand for high precision signal processing has introduced a new method called Empirical Mode Decomposition (EMD), which is used for analyzing a complex set of data. Unfortunately, EMD is highly compute-intensive. In this paper, we show parallel implementation of Empirical Mode Decomposition on a GPU. We propose the use of "partial+total" switching method to increase performance while keeping the precision. We also focused on reducing the computation complexity in the above method from O(N) on a single CPU to O(N/P log (N)) on a GPU. Evaluation results show our single GPU implementation using Tesla C2050 (Fermi architecture) achieves a 29.9x speedup partially, and a 11.8x speedup totally when compared to a single Intel dual core CPU.

key words: *Empirical Mode Decomposition (EMD), Hilbert-Huang Transform (HHT), GPU, CUDA*

1. Introduction

In the field of signal processing, it is common to analyze complex sets of data, e.g: harmonic signals, stock price fluctuation, and yearly average temperature. These complex data are so called non-stationary data, where the frequency changes over time. Traditionally, non-stationary data is analyzed using Short-Time Fourier Transform (STFT). However, since STFT has its own drawback, i.e. there exists a trade-off between frequency and time resolution, an adequate analysis cannot be performed. In recent years, new methods have been introduced to overcome such disadvantage, giving higher precision in signal analysis [1], [2].

One of such methods is Empirical Mode Decomposition (EMD) in conjunction with a Hilbert Spectral Analysis (HSA), together called Hilbert-Huang Transform (HHT) [1]. HHT is designed as an adaptive, empirically based data-analysis method for non-stationary process in off-line analysis, i.e. analysis where the whole problem data is given. On the other hand, HHT needs a large amount of computation. As discussed later, such complexity comes from the algorithm having two nested loops structure with the inner loop having an $O(N)$ complexity. As a result, it takes hours to analyze even for a short 3 seconds harmonic signal.

The EMD algorithm, as will be described later, is the performance bottleneck of HHT, but it also shows a high degree of data parallelism. Therefore, parallelizing EMD accelerates the speedup of the program. However, many of the current research on EMD focuses on the application [3]–[6] and its extension [7]–[13]. Therefore we take advantage of CUDA to develop a parallel model of EMD. Although there exist a previous work concerning parallelization strategy of EMD using GPU [14], since it is aiming for on-line analysis, the focus differs greatly to the one described in this paper. This is explained in detail in Sect. 5.

The main contribution of our work is to provide a parallel EMD algorithm for off-line analysis. The algorithm is developed on a GPU. It consists of two parts: *partial method* and *total method*. By using two methods appropriately, the algorithm achieves high precision and high performance.

In this paper, we extend our previous research [15]. This paper aims to find adequate parameters for our parallel method. In addition, we show new results using NVIDIA Fermi with cache architecture. As a result, our implementation achieves a 29.9 times speedup partially for single result, and 11.8 times speedup for all results when compared to a typical general purpose CPU's execution time.

The rest of this paper is organized as follows. Section 2 summarizes the HHT and the EMD algorithm. Section 3 describes the details of our parallelization strategy for EMD using CUDA. Experimental results are analyzed in Sect. 4. In Sect. 5, we summarize previous work related to the GPGPU-aided EMD. Finally, Sect. 6 summarizes our work and concludes the paper.

2. The Hilbert-Huang Transform

As shown in Fig. 1, HHT consists of two different algorithms. One is called Hilbert Spectral Analysis (HSA), which computes the instantaneous frequency from the data. However, HSA cannot be used on data where multiple frequencies are mixed. Therefore, the other algorithm, Empirical Mode Decomposition (EMD) acting as a preprocessor is applied to the input data prior to HSA.

Although the details are omitted here, the workload of HSA is substantially smaller compared to the workload of EMD [14]. Additionally, the majority of HSA process consists of applying Fast Fourier Transform (FFT) and its inverse to the data over the frequency domain. The parallelization of FFT and its inverse have been studied extensively over the years, and there have been previous works concerning the FFT implementation in CUDA [16].

Manuscript received December 28, 2010.
Manuscript revised June 10, 2011.

[†]The authors are with the Department of Computer and Information Sciences, Tokyo University of Agriculture and Technology, Koganei-shi, 184–8588 Japan.
a) E-mail: pulungw@nj.cs.tuat.ac.jp
b) E-mail: nakajo@cc.tuat.ac.jp

DOI: 10.1587/transinf.E94.D.2328

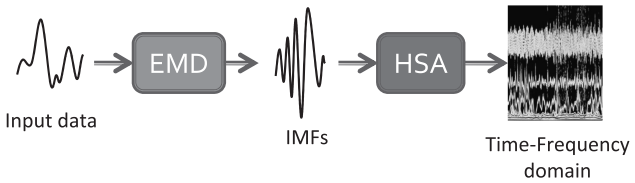


Fig. 1 Hilbert-Huang transform (HHT).

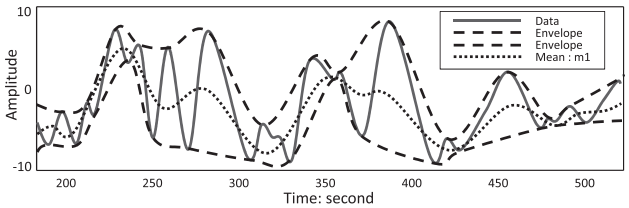


Fig. 2 Envelope and mean calculation.

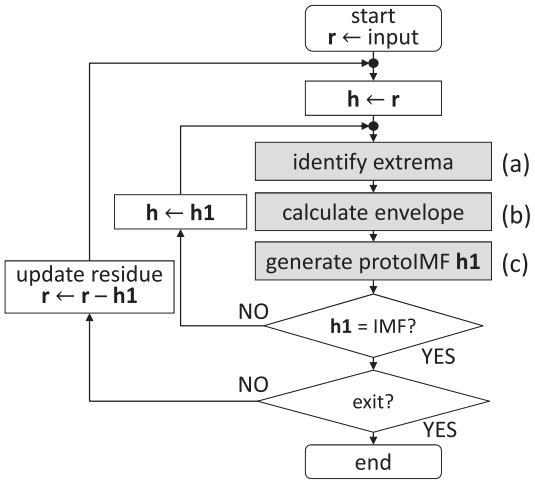


Fig. 3 EMD algorithm.

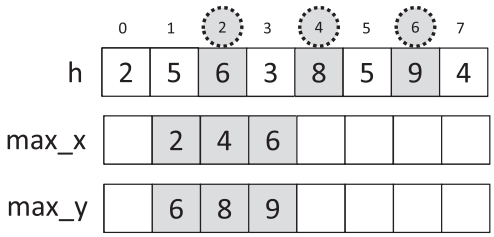


Fig. 4 Extrema padding.

Therefore, this paper focuses only on the parallelization and implementation of EMD in CUDA. The EMD procedure is explained in more details as follows.

2.1 EMD Overview

The basic idea of the Empirical Mode Decomposition is a simple assumption that any data consists of different simple intrinsic modes of oscillation [1]. At any given time, the data may have many different coexisting modes of oscillation one superimposing on the others. The result is the final complicated data. Each of these oscillatory modes is represented by an intrinsic mode function (IMF) with the following definition:

- in the whole dataset, the number of extrema and the number of zero-crossings must either equal or differ at most by one, and
- at any point, the mean value of the envelope (Fig. 2) defined by the local maxima and the local minima is zero.

To decompose the input signal $f(t)$ into IMFs, the EMD algorithm is shown as a flowchart in Fig. 3. Let the value of the current input signals be $s_i(t)$ with i as number of IMF. First, identify all the local extrema (step a), then connect all the local maxima by a cubic spline line to produce the upper envelope (step b). Repeat the procedure for the local minima to produce the lower envelope (step b). The upper and lower envelopes should cover all the data between them (see Fig. 2). Their mean is designated as $m_i(t)$, and the difference between the data and $m_i(t)$ is calculated as $h_i(t) = s_i(t) - m_i(t)$(step c).

Unfortunately, $h_i(t)$ does not yet satisfy the definition of an IMF. Such function is called a protoIMF. The same procedure is simply executed to $h_i(t)$ during the inner loop process until it reaches the definition of IMF.

After repeated sifting, $h_i(t)$ becomes an IMF. Overall, $h_i(t)$ should contain the finest scale of the signal. Then $h_i(t)$ can be separated from the rest of the data by $r_i(t) = s_i(t) - h_i(t)$. Since the residue $r_i(t)$ still contains longer frequency variations in the data, it is treated as the new data and subjected to the same sifting process as described above. This procedure can be repeated with all the subsequent $r_i(t)$, until the residue $r_i(t)$ becomes a monotonic function from which no more IMFs can be extracted. By definition, the relation between the input data $f(t)$ and IMFs $h_i(t)$ is as follows:

$$f(t) = \sum_{i=1}^{n} h_i(t) + r(t) \tag{1}$$

As described below, the complexity of each of the inner loop procedure is shown to be $O(N)$.

2.2 Extrema Identification

Each extrema point (maxima and minima) can be identified by comparing each value of input signal s_i with the previous and behind value in sequence $i = 0, \cdots, N-1$. Since this procedure sequentially compares each values in the data, the complexity can be expected to be $O(N)$. The x and y value of each maxima (minima) is then stored in new arrays. After this procedure, separate arrays of maxima and minima set are created which are used in the following envelope calculation, shown in Fig. 4. This figure shows a case for maxima identification, with the gray part representing the identified maxima.

2.3 Envelope Calculation

Cubic Spline Interpolation is used to interpolate the interval between extrema creating an envelope. The essential idea of cubic spline interpolation is as follows. Given a set of extrema (x_j, y_j), $j = 1, \cdots, n$, the cubic spline curve (e_i) connecting them can be calculated using the following equation,

$$e_i = Ay_j + By_{j+1} + Cy''_j + Dy''_{j+1} \tag{2}$$

where $x_j < i < x_{j+1}$, and y''_j as the second derivatives of each x_j. Each A, B, C and D can be calculated using x_j and x_{j+1} respectively

In practice, since the values of each y''_i are unknown, they need to be specified before calculating Eq. (2). The key idea of a cubic spline requires the first derivative be continuous across the boundary between two intervals (y'_j from interval x_{j-1} and x_j is the same as y'_j from interval x_j and x_{j+1}). After taking the derivatives of Eq. (2) and rearranging it, the following equation is obtained and can be used to calculate a unique set of y''_i values:

$$ay''_{j-1} + by''_j + cy''_{j+1} = r \tag{3}$$

where a, b, c and r are also calculated using x_j, x_{j-1} and x_{j+1} respectively.

The cubic splines are especially practical because the set of Eq. (3) can be arranged in a *tridiagonal* matrix just as shown below.

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{n-1} \\ 0 & & & a_n & b_n \end{pmatrix} \begin{pmatrix} y''_1 \\ y''_2 \\ y''_3 \\ \vdots \\ y''_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_n \end{pmatrix} \tag{4}$$

These equations can be solved in $O(n)$ operations by the Tridiagonal Matrix Algorithm (TDMA). TDMA is a simpler form of Gaussian elimination method, where if the equations can be represented as a tridiagonal matrix, the solution can be obtained by just one forward sweep and one backward elimination. Once y''_i are obtained, simply applying Eq. (2) brings the result of the envelope.

We point out that the envelope calculation does not require pivot operations. This is because matrices generated by Cubic Spline Interpolation are strictly diagonally dominant [17]. Gaussian elimination without pivoting is stable for diagonally dominant matrices. Thus, the Eq. (4) can be solved in a simple manner.

2.4 protoIMF Generation

After the upper envelope and lower envelope are calculated, finding the mean value h_i can be calculated in $O(N)$ operations. The C code snapshot of the operation is shown below.

```
for(i=0; i<N; i++){
    m[i]  = (upper[i] + lower[i])/2.0;
    h1[i] = h[i] - m[i];
}
```

3. Parallelization Strategy for EMD

In this section, the details on our proposed parallelization strategy for EMD are explained. We first describe a common CUDA implementation to achieve high performance, followed with problems that arises with naive implementation. Then we explain our approach to solve the problems which we name *Partial+Total* method.

3.1 Shared Memory Usage and Its Problem

A common implementation in CUDA is to split the tasks to group of thread blocks, where each block is executed in parallel by each Streaming Multiprocessor (SM) in GPU. Each SM has a small on-chip shared memory that can be accessed very quickly by every thread in the same block compared to global memory access. Maximizing the use of this shared memory is the focus in our parallelization strategy.

To use the shared memory fully, the tasks are divided to regions such that each region fits in each SM's shared memory. Ideally, each block applies the inner loop procedure as described on the previous section only to the data inside the region.

However, as explained in Sect. 2.3, during the envelope calculation, the coordinate of every extrema points need to be considered to calculate the second derivative y''_j in Eq. (4). Since each block is isolated and is practically executing each region independently from each other, considering all the extrema from the input data is practically impossible. Each block cannot access the extrema data outside its own shared memory. Upon calculating the envelope with the limited data, the envelope from each block will not be smooth and continuous across the boundary between regions. As a result, the resulting IMF may contain error.

Therefore, the data is overlapped for some interval around the boundary during the splitting of the input [15]. This behavior is shown in Fig. 5. Each block has an extra part of the data around the boundary allowing it to calculate extra part of the envelope. After the inner loop procedure terminates, each block discards the extra part and outputs the result, resulting in one continuous IMF.

However, there is a limitation in this *Partial* method. If the number of extrema found in the region is too small for interpolating a continuous envelope, a wrong envelope may be produced. A certain number of extrema in each thread block is needed to maintain one continuous envelope across regions. To facilitate this, we propose a switching method between shared and global memory explained in the next section [15].

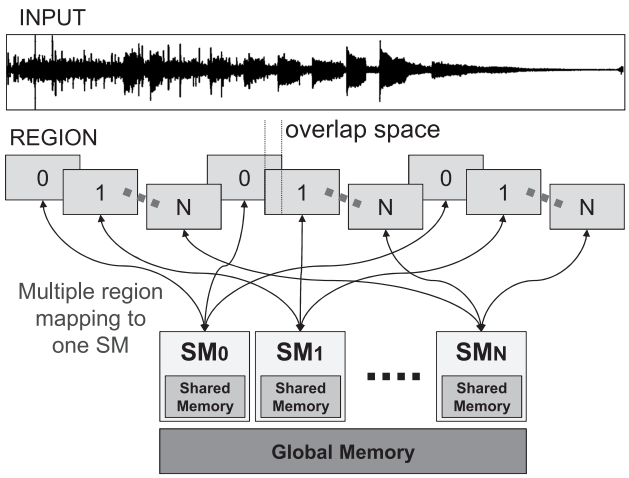


Fig. 5 Proposed EMD partial method.

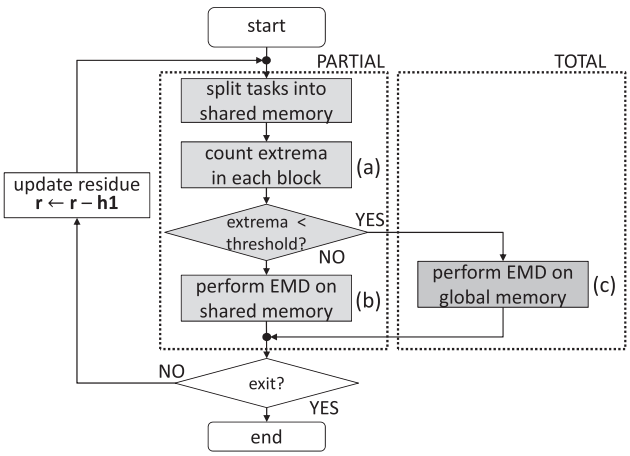


Fig. 6 EMD partial+total method.

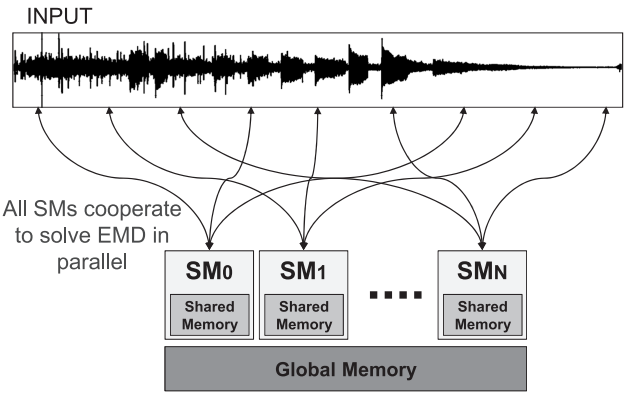


Fig. 7 Proposed EMD total method.

3.2 Coordination between Partial and Total Methods

Our proposed coordination method is shown in Fig. 6. When the kernel is launched, the partial method is first used to determine the number of extrema in each region (step a). If the number is larger than a threshold, the partial method is continued (step b). However if the number in one region is lower than a threshold, the EMD method is switched to the *total* method (step c), where EMD is calculated this time using whole input data in the global memory where all SMs cooperate to solve (see Fig. 7).

Checking of other regions which contain less extrema (step a) requires inter-block synchronization, which increases the total execution time. To reduce this side effect, checking is done only at the beginning of calculating IMF_i. This is because the number of extrema in each region is stable during the inner loop of the EMD process.

In our proposed strategy, the switch to total method is executed inside the kernel, so to facilitate inter-block communication, a GPU lock-based synchronization is applied [19]. The basic idea is to use a global mutex variable combined with atomicAdd function to count the number of thread blocks which reach the synchronization point. It acts as a barrier function stalling each thread block until every block reaches the same point [18].

Since in a CUDA programming model, the execution of a thread block is non-preemptive, i.e. its execution cannot be interrupted until it is finished, care must be taken when attempting to coordinate a group of thread blocks to avoid deadlocks. Therefore, a one-to-one mapping between thread blocks and SMs is applied. For a GPU with N number of SMs, no more than N blocks can be used in the kernel. This results in multiple region mapped to one SM as shown in Fig. 5. After completing the first IMF calculation, a new input data is calculated and the procedure described above is repeated.

Next the parallelization strategy for each of the inner loop procedure is described with focus on reducing the complexity of each procedure.

3.3 Extrema Identification

In this procedure, each thread decides whether or not one point is an extrema. Since comparison with the point before and behind that point can be done for each point independently, extrema identification is simple to perform on a CUDA implementation. Each thread is mapped to a single point in the data to perform the comparison in parallel.

However, problem arises when trying to store the extrema coordinate into separate arrays. Just as described in Sect. 2.2, the subsequent envelope calculation needs separate arrays for x and y values as shown in Fig. 4.

Here prefix sum method is used for the padding procedure to obtain the max_x and max_y arrays. Figure 8 shows the prefix sum behavior on an array with 8 elements. First the element of each of identified maxima is set to 1, otherwise set to 0. Then apply the prefix sum as shown in the figure, resulting each element to be the sum of all the elements in the array up to its index. Finally using this value as a new array index, the value from the data are copied to the max_x and max_y arrays shown in Fig. 4.

The prefix sum algorithm performs $O(Nlog(N))$ addi-

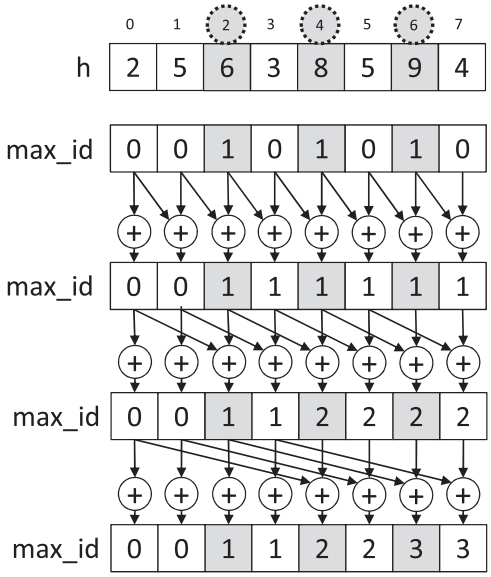


Fig. 8 Prefix sum behavior.

tion operations. Thus the complexity of extrema identification becomes $O(\frac{N}{P}log(N))$ with the number of processors as P. Although the prefix sum algorithm performs more operations than the $O(N)$ sequential scan (Sect. 2.2), in practice with large P number, the prefix sum algorithm performs better. The GPU implementation uses TESLA C2050 with 448 Stream Processors (SP), thus prefix sum is chosen for the padding procedure.

3.4 Envelope Calculation

Just as described in Sect. 2.3, during the envelope calculation, the solution for tridiagonal matrix equations is calculated. There are a couple of variants to solve tridiagonal linear systems in parallel, namely cyclic reduction (CR) and parallel cyclic reduction (PCR) [20]. In this paper, the PCR method is chosen because less synchronization is needed compared to CR method.

CR consists of two phases, the forward reduction and backward substitution. From step complexity, CR has $O(N)$ step complexity, while PCR has $O(log(N))$ step complexity. Although the step complexity is lower in PCR, each step in PCR consists of N operations. In comparison, CR has more step complexity but less operations for each step [20]. In this paper, PCR is chosen as the tridiagonal system solvers. The reason for this is in this implementation, especially in global method, synchronization point between all blocks is needed for every step. Since the cost of each synchronization is high, keeping the synchronization number lower is preferred. With PCR, the number of synchronization is lower than CR, therefore PCR is better suited for this implementation.

In a PCR method, the Gaussian elimination procedure is performed in parallel, eliminating all of the terms for each row until a matrix which has only one diagonal non zero entries is obtained. The cyclic reduction procedure is explained below.

First, the first three rows of matrix Eq. (4) can be transformed into general form shown in equations below.

$$b_1y_1''+c_1y_2'' = r_1 \quad (5)$$
$$a_2y_1''+b_2y_2''+c_2y_3'' = r_2 \quad (6)$$
$$a_3y_2''+b_3y_3''+c_3y_4'' = r_3 \quad (7)$$

Focusing on Eq. (6) on the 2nd row, the a_2y_1'' term can be eliminated by substituting $y_1'' = (r_1 - c_1y_2'')/b_1$ from the Eq. (5) on the 1st row. In a similar fashion, the c_2y_3'' term can also be eliminated by substituting $y_3'' = (r_3 - a_3y_2'' - c_3y_4'')/b_3$ from the Eq. (7) on the 3rd row. The key idea of cyclic reduction is to eliminate the a_jy_{j-1}''term and c_jy_{j+1}''term for each row j using the equations above and below it. Because the elimination for each row can be performed independently, cyclic reduction can easily be calculated in parallel.

After the first step, the a_jy_{j-1}'' and c_jy_{j+1}''term for each row j are eliminated and replaced with $a_j'y_{j-2}''$ and $c_j'y_{j+2}''$term. The Eq. (5), (6), (7) become,

$$b_1'y_1''+c_1'y_3'' = r_1 \quad (8)$$
$$b_2'y_2''+c_2'y_4'' = r_2 \quad (9)$$
$$a_3'y_1''+b_3'y_3''+c_3'y_5'' = r_3 \quad (10)$$

which can be written back into the matrix form as,

$$\begin{pmatrix} b_1' & 0 & c_1' & & 0 \\ 0 & b_2' & 0 & c_2' & \\ a_3' & 0 & b_3' & \ddots & \\ & \ddots & \ddots & \ddots & 0 \\ 0 & & a_n' & 0 & b_n' \end{pmatrix} \begin{pmatrix} y_1'' \\ y_2'' \\ y_3'' \\ \vdots \\ y_n'' \end{pmatrix} = \begin{pmatrix} r_1' \\ r_2' \\ r_3' \\ \vdots \\ r_n' \end{pmatrix} \quad (11)$$

Notice on the left side of the equations, each variable a_j and c_j changes position after each step with a_j moving to the left and c_j to the right respectively, while variable b_j stays on the same position. This procedure is then repeated to the new equations, this time substituting the $a_j'y_{j-2}''$ and $c_j'y_{j+2}''$term from each row j with row $j-2$ and $j+2$. After the second step, the equations become,

$$\begin{pmatrix} b_1'' & 0 & 0 & c_1'' & & 0 \\ 0 & b_2'' & 0 & 0 & c_2'' & \\ 0 & 0 & b_3'' & 0 & \ddots & \\ a_4'' & 0 & 0 & b_4'' & \ddots & \\ & \ddots & \ddots & \ddots & \ddots & \\ 0 & & a_n'' & 0 & 0 & b_n'' \end{pmatrix} \begin{pmatrix} y_1'' \\ y_2'' \\ y_3'' \\ y_4'' \\ \vdots \\ y_n'' \end{pmatrix} = \begin{pmatrix} r_1'' \\ r_2'' \\ r_3'' \\ r_4'' \\ \vdots \\ r_n'' \end{pmatrix} \quad (12)$$

with the variable a_j and c_j moving farther and eliminated completely from the equations. The subsequent procedure is done using row $j-4$ and $j+4$ respectively. Finally, after $log(n)$ operations, a matrix which is having one diagonal non zero entries is obtained just as shown in Eq. (13). Looking at

these matrix equations, it is naturally clear that the solution is trivial.

$$\begin{pmatrix} b_1''' & & & & 0 \\ & b_2''' & & & \\ & & b_3''' & & \\ & & & \ddots & \\ 0 & & & & b_n''' \end{pmatrix} \begin{pmatrix} y_1'' \\ y_2'' \\ y_3'' \\ \vdots \\ y_n'' \end{pmatrix} = \begin{pmatrix} r_1''' \\ r_2''' \\ r_3''' \\ \vdots \\ r_n''' \end{pmatrix} \quad (13)$$

Thus, the complexity of envelope calculation can be reduced to $O(\frac{N}{P}log(n))$.

3.5 protoIMF Generation

There exists a data parallelism in the code shown in Sect. 2.4. Therefore, the protoIMF generation can easily be parallelized on the GPU. Each thread is mapped to each point where they first calculate the mean and then proceed to generate the protoIMF.

3.6 Implementation on Fermi Architecture

Fermi architecture is the newest GPU architecture for CUDA released by NVIDIA. One of the addition compared to previous GT200 series is programmer-managed 64 KB scratch-pad memory† in each SM. The 64 KB shared memory can be configured as either 48 KB of scratch-pad memory with 16 KB of L1 cache (s48k/c16k), or 16 KB of scratch-pad memory with 48 KB of L1 cache (s16k/c48k). When configured with 48 KB of scratch-pad memory, programs that make extensive use of scratch-pad memory can perform up to three times faster. For programs whose memory accesses are not known beforehand, the 48 KB L1 cache configuration offers greatly improved performance over direct access to DRAM.

In our partial+total method, the total method is significantly slower than the partial method due to extensive access to global memory [15]. It is therefore preferable to configure the shared memory as s16k/c48k to reduce the global memory latency for total method. Partial method can also take advantage of this large cache. When designing program in parallel machine with cache architecture, we consider the spatial and temporal locality by tiling the data to fit inside the cache memory.

Figure 9 shows the tiled data structure for partial method. SM_DATA is a long data array where one SM_DATA array is allocated for each SM. In a case with 14 SMs, then 14 SM_DATA arrays are allocated. These data arrays lies on global memory instead on scratch-pad memory. Inside the SM_DATA array, the data needed for the calculation are explicitly tiled. All data are allocated in sequence on the memory. Therefore, when the size of SM_DATA is less than the cache size, all the data inside SM_DATA are guaranteed to be loaded on to the cache.

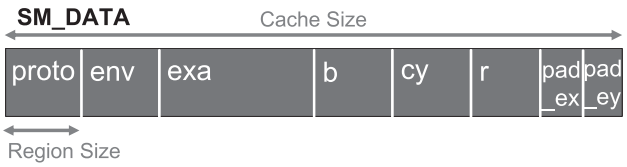


Fig. 9 Data tiling in partial method to utilize Fermi cache architecture.

These are the minimum amount of data needed for inner loop procedure. *proto* is an array to store the calculated protoIMF data. The size of this array is set equal to the region size. *env* is used to store the maxima envelope and mean envelope. *exa* is used for temporary arrays in the padding procedure (prefix sum). *pad_ex* and *pad_ey* are used to store the padding results.

4. Evaluation Results

First, the evaluation to find effective overlap space and switching threshold is shown. Then, the evaluation results of our *partial+total* EMD method are presented.

4.1 Evaluation Environment

Evaluation environment is shown in Table 1. We use a single Intel Core 2 Duo 2.53 GHz for CPU, TESLA C1060 (GT200) with 240 SP (Stream Processor), and TESLA C2050 (Fermi) with 448 SP. For the CPU program, the HHT program included in MIST (Media Integration Standard Toolkit) is used. MIST is a standard library for media and signal processing developed by Nagoya University [21]. The CPU program is a sequential program utilizing only 1 core, therefore in Table 1 only 1 core is shown.

For the input data, the harmonic signal Hotel California.wav sampled at 44.1 kHz is chosen. For our partial EMD method evaluation, the region size is set to 256 points on GT200 and 1024 points on Fermi, due to different size of the scratch-pad memory. For the partial+total EMD method, the thread block is set to 30 for GT200 and 14 for Fermi, respectively. These number are based on the number of SMs in each GPU. The size of shared memory limits the size of region that our implementation can process. However as long the total data size fits on the global memory, our implementation can efficiently process the data. The number of threads per block that are responsible for the EMD calculation of each region is 256 in GT200, and 512 in Fermi. Therefore each thread in a block is in charge for 2 points in each region.

Additionally, the check to decide whether the protoIMF

Table 1 Evaluation environment.

	CPU	GT200	Fermi
Proc.	Core 2 Duo	TESLA C1060	TESLA C2050
Cores Num.	1	240 (30 SM)	448 (14 SM)
Clock Sp.	2.53 GHz	1.296 GHz	1.15 GHz
Dev. Env.	GCC 4.1.2 gcc -O3	CUDA SDK 2.3 nvcc	CUDA SDK 3.1 nvcc

†NVIDIA refers to programmer-managed memory as shared memory. In this paper, we refer to such memory as scratch-pad memory, which is a more general term.

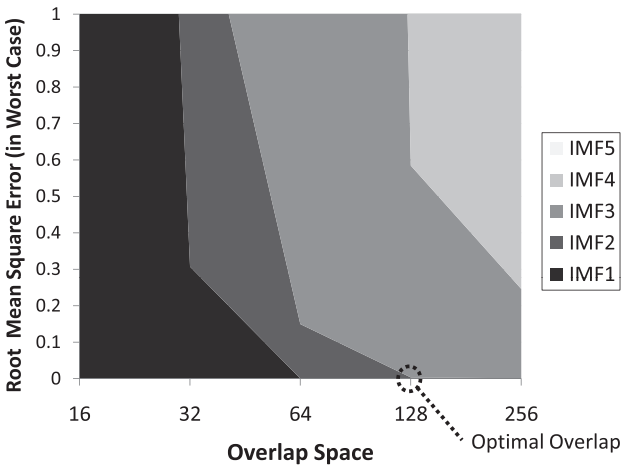


Fig. 10 Error corresponding to overlap space in partial method.

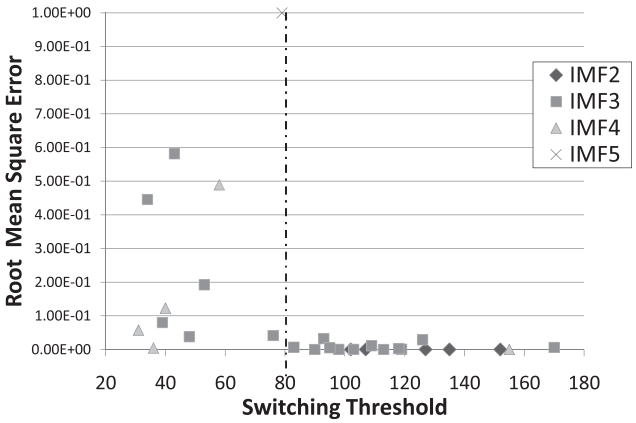


Fig. 11 Error corresponding to the minimum number of extrema at the beginning of calculating $IMF_i (i = 2, 3, \cdots, 5)$.

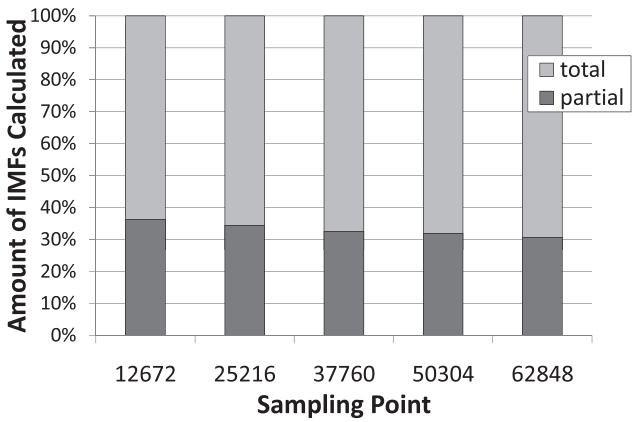


Fig. 12 Breakdown of partial and total method.

satisfy the IMF condition is not performed, but instead stopping the inner loop procedure after 1,000 loops. The number of iteration of the outer loop is set to 8. These simplification is not practical, however, it enables to compare the outputs of proposed algorithm with those of original algorithm at any steps. Therefore, we exploit the simplification for the evaluation of the precision. We also evaluate the proposed algorithm without the simplification later.

4.2 Precision Results

First, we evaluate how long overlap space is optimal. We use 20 sets of datum as test datum, which are randomly selected from the input data. Each test data contains 64 K sampling points and the head data (which consists of 12672 points) is used for the evaluation. For these datum, we calculate IMFs with the partial method.

Figure 10 shows the root mean square error (RMSE) of $IMF_i (i = 1, 2, \cdots, 5)$ with partial method compared to the total method. RMSE is accumulated during IMF calculation and cumulative RMSE in the worst case is shown in the figure. The x-axis is the evaluated overlap space, and the y-axis is the calculated RMSE. Lower RMSE means the precision does not deteriorate. The result shows that overlap space larger than 128 gives high enough precision for IMF_2. That is, if the space is larger than 128, the partial method ensures precise results for IMF_1 and IMF_2. Therefore we set the overlap point parameter to 128.

Second, we investigate optimal switching threshold. As described in Sect. 3.2, at the beginning of IMF_i calculation, the total method starts if the number of extrema in a region is less than the threshold. In other words, the minimum number of extrema at the beginning of IMF_i calculation determines whether the partial method can calculate precisely or not. If precise IMF_i can not be calculated, the threshold should be larger than the minimum number.

Figure 11 shows the relationship between RMSE of IMF_i and the minimum number of extrema at the beginning of the calculation. The x-axis is the minimum number, and the y-axis is the RMSE. For the evaluation, we use above 20 test datum to get plotted pairs of the number and RMSE. A singular point exists at 79, while all RMSEs are significantly small in the area larger than 79. Thus, we set the switching threshold to 80 for the following evaluation.

4.3 Performance Results

In Fig. 12, we show the breakdown of IMFs calculated with partial and total method. The number of iteration of the inner loop is fixed in this experiment, so the amount of IMFs calculated by a method represents how many times critical operations (which consists of three steps mentioned in Sect. 2.2) are executed by its method. The x-axis is the sampling point evaluated, and the y-axis is the average amount of IMFs calculated by each method. The amount of IMFs calculated by partial method and total method is shown in dark and light area, respectively.

For large input size, the amount of IMFs calculated by the partial method is reduced. This is because a region which consists of less extrema than the threshold has a higher chance of appearing as the datum increases. If such region exists in the datum, the method is switched to the

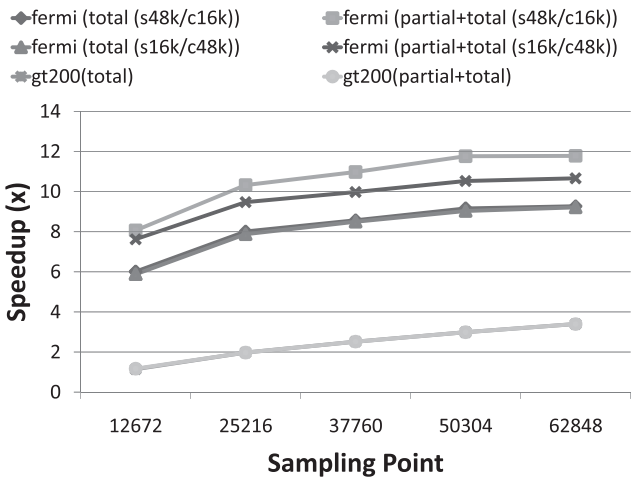


Fig. 13 Total speedup of proposed method compared to CPU.

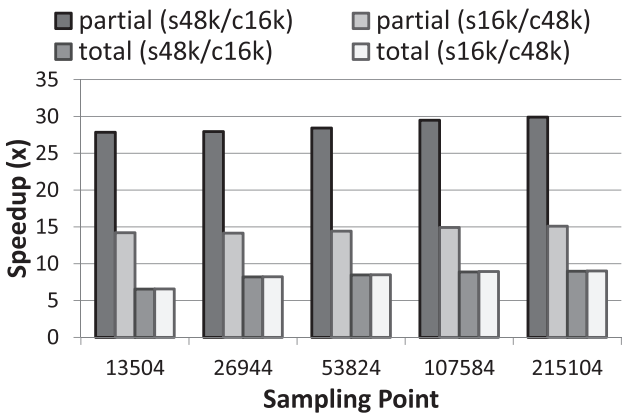


Fig. 14 Speedup of proposed method using different shared memory configuration for single IMF.

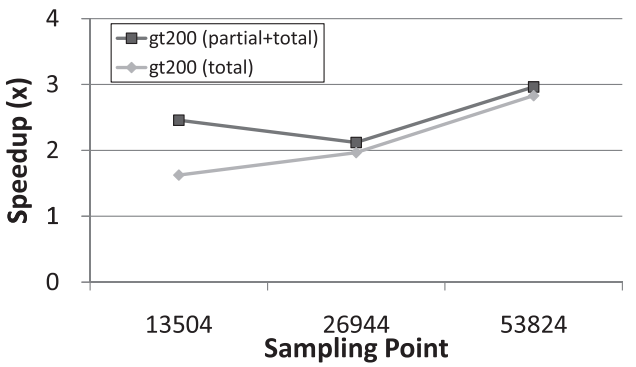


Fig. 15 Speedup of the proposed method using SD calculation compared to CPU.

total method earlier, reducing the ratio of partial method.

Figure 13 shows 6 lines corresponding to each configuration, of which the bottom 2 correspond to the speedup of total method on GT200. The rest represent Fermi's result. The x-axis is the evaluated sampling point, and the y-axis is the speedup normalized by the CPU runtime. In addition to the evaluation of partial+total method and total method, we also test different shared memory configuration, set as s48k/c16k and s16k/c48k respectively.

On GT200, partial+total method achieves 3.4x speedup (circle point) in best case. Meanwhile, on Fermi, partial+total method achieves 11.8x speedup (square point) in best case.

The number of cores on Fermi is about 2 times greater than that on GT200, but the speedup ratio on Fermi is about 3 times greater than that on GT200. This mismatch is caused by the difference in region sizes. As described in Sect. 4.1, the size on Fermi is quadruple compared to the size on GT200. The region size is a significant factor related to the amount of IMFs calculated by the partial method. Thus, parallelizing on Fermi greatly outperforms rather than parallelizing on GT200.

In Fig. 12 and Fig. 13, the speedup ratio of partial+total method on Fermi is improved as the size of test datum increases, while the ratio of IMFs calculated by the partial method is decreased. This is because the speedup by the total method is slightly improved as increasing the datum. The effect of the speedup by the partial method is reduced, however, the effect of the speedup by the total method causes the improvement of the total performance.

Overall, partial+total method is faster than total method, 11.8x speedup (square point) compared to 9.3x (diamond point) in best case respectively. For different shared memory configuration, the result shows that s48k/c16k achieves 11.8x speedup in best case (square point), while s16k/c48k configuration achieves 10.7x speedup in best case (cross point) for partial+total method. Total method using both configuration show the same performance (triangle and diamond point).

This means that large cache configuration on Fermi has little effect to the performance. To further examine the effect of shared memory configuration, we show the speedup derived from calculating a first IMF (Fig. 14). The x-axis is the evaluated sampling point, and the y-axis is the speedup normalized by the CPU runtime. The result shows that partial method using shared memory as a scratchpad achieves more than double the performance (maximum 29.9x speedup) compared to using it as a cache (maximum 15.1x speedup). Total method shows no advantage when using large cache, contrary to our assumption in previous section. We believe that this is because total method only has spatial locality, thus reducing the effectiveness of cache.

For comparison, Fig. 15 shows the speedup of the proposed method under GT200 when implementing SD calculation with inter-block synchronization on every iteration. Figure 15 shows the speedup is reduced to a maximum of 3.0x. We believe that this is caused by using inter-block synchronization which further introduces communication overhead. Therefore new approach is needed to avoid the synchronization in each iteration. In our future works, we will investigate approaches such as synchronizing after some hundred cycles.

Additionally, on input size 26944 points the speedup of shared-global method is reduced (see Fig. 15). This is

caused by the difference in loop count when partial method is active. Although partial method produces IMF with small error compared to CPU implementation, such error is enough to produce different loop count for the subsequent total method. In some cases, total loop count of the partial+total method is around the total loop count when using only global method. This results in similar speedup shown in Fig. 15, lowering the effect of partial method. Further analysis on the relationship between the loop count and shared method will also be our focus for future works.

5. Related Works

Chen et al. shows a GPGPU-aided Ensemble EMD for on-line analysis [14]. Ensemble EMD is a kind of EMD which is designed to overcome EMD's sensitivity to noise [7]. Their approach is very similar to ours, but there are two different points: 1) their target is EMD for on-line analysis, and 2) their implementation is not taking account of the GPU architecture.

In signal analysis, on-line analysis is different from off-line analysis. In on-line analysis, recent signals are subtracted from the historical input data as an input data. Then the input is processed by analytical algorithm. These two steps are alternated indefinitely. The signal is fed to the analytical system, so the system must analyze the input in real-time. Hence the analytical speed weighs heavier than the analytical resolution. For this purpose, input signals of EMD are preferred to be shorter, therefore they divide subtracted signals into small pieces beyond the analytical limitation.

In contrast, off-line analysis aims to obtain precise analyzed data by treating the whole problem data as an input. Therefore, the data fed to our system is longer than the one fed to theirs. We also divide the data into small pieces to improve performance. However, for the above purpose, we do not create extremely small pieces which causes deterioration in precision.

Since the EMD process for subtracted signals in on-line analysis is identical with the EMD process in off-line analysis, it seems that their parallelizing method could be exploited for off-line EMD. However, they do not take the GPU architecture into account sufficiently to develop a parallel EMD. For example, they use only one thread block, which means that the rest of SMs are idle. Moreover, they do not exploit shared memory at all. Thus our approach and theirs differ greatly in focus and implementation.

6. Conclusions

This paper presents an evaluation of our GPU-based EMD implementation for CUDA-enabled devices. A brief introduction to EMD has been provided prior to explaining our parallelization strategy. The results show that our proposed parallelization strategy reduces the complexity to $O(\frac{N}{P}log(N))$ when used on a GPU. The partial method achieves a maximum speedup of 29.9 times for calculation of a single IMF compared to sequential implementation on a CPU. Meanwhile the partial+total method achieves a maximum speedup of 11.8 times for calculation of all IMFs. Additionally, setting up an overlap interval to achieve the same result to reduce error is shown to be effective.

As for future works, we will further investigate a new approach to increase the partial method ratio. Second, we will implement our own parallel Hilbert Spectral Analysis and perform exhaustive comparison on the final result.

Acknowledgements

This research is partially supported by Japanese MEXT Fund for Promoting Research on Symbiotic Information Technology.

References

[1] N.E. Huang, Z. Shen, S.R. Long, M.C. Wu, H.H. Shih, Q. Zheng, N.C. Yen, C.C. Tung, and H.H. Liu, "The empirical mode decomposition and the hilbert spectrum for nonlinear and non-stationary time series analysis," Proc. Royal Society A: Mathematical, Physical and Engineering Sciences, vol.454, pp.903–995, 1998.

[2] E. Rao and A. Bopardikar, Wavelet transforms, introduction to theory and applications, Addison Wesley Longman, 1998.

[3] N. Rehman and D.P. Mandic, "Filter bank property of multivariate empirical mode decomposition," IEEE Trans. Signal Process., vol.59, no.5, pp.2421–2426, 2011.

[4] N. Rehman and D.P. Mandic, "Empirical mode decomposition for trivariate signals," IEEE Trans. Signal Process., vol.58, no.3, pp.1059–1068, 2010.

[5] T. Rutkowski, D.P. Mandic, A. Cichocki, and A. Przybyszewski, "EMD approach to multichannel EEG data analysis - the amplitude and phase components clustering," J. Circuits, Systems, and Computers, vol.19, no.1, pp.215–229, 2010.

[6] D. Looney and D.P. Mandic, "Multi-scale image fusion using complex extensions of EMD," IEEE Trans. Signal Process., vol.57, no.4, pp.1626–1630, 2009.

[7] Z. Wu and N.E. Huang, "Ensemble empirical mode decomposition: A noise-assisted data analysis method," Advances in Adaptive Data Analysis, vol.1, no.1, pp.1–41, 2009.

[8] N. Rehman and D.P. Mandic, "Multivariate empirical mode decomposition," Proc. Royal Society A, vol.466, no.2117, pp.1291–1302, 2010.

[9] N.U. Rehman and D.P. Mandic, "Quadrivariate empirical mode decomposition," Proc. IJCNN'10, pp.2265–2270, 2009.

[10] N.U. Rehman and D.P. Mandic, "Qualitative analysis of rotational modes within three-dimensional empirical mode decomposition," Proc. ICASSP'09, pp.3449–3452, 2009.

[11] T. Tanaka and D.P. Mandic, "Complex empirical mode decomposition," IEEE Signal Process. Lett., vol.14, no.2, pp.101–104, 2007.

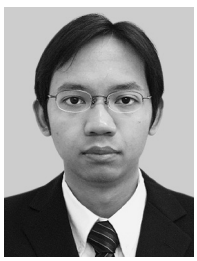
[12] M.U.B. Altaf, T. Gautama, T. Tanaka, and D.P. Mandic, "Rotation invariant complex empirical mode decomposition," Proc. ICASSP'07, pp.1009–1012, 2007.

[13] Y. Washizawa, T. Tanaka, D.P. Mandic, and A. Cichocki, "A flexible method for envelope estimation in empirical mode decomposition," Proc. 10th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems, KES-06, pp.1248–1255, 2006.

[14] D. Chen, D. Li, M. Xiong, H. Bao, and X. Li, "GPGPU-aided ensemble empirical-mode decomposition for EEG analysis during anesthesia," IEEE Trans. Inf. Technol. Biomed., vol.14, no.6, pp.1417–1427, 2010.

[15] P. Waskito, S. Miwa, Y. Mitsukura, and H. Nakajo, "Parallelizing hilbert-huang transform on GPU," Proc. 2nd Workshop on Ultra

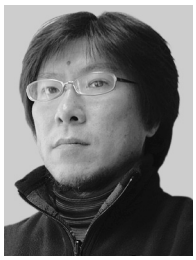
Performance and Dependable Acceleration Systems (UPDAS'10), pp.184–190, 2010.
[16] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," Proc. 2008 ACM/IEEE conference on Supercomputing, SC '08, pp.5:1–5:11, 2008.
[17] G. Dahlquist, Numerical Methods in Scientific Computing: Volume 1, Society for Industrial Mathematics, 2008.
[18] S. Xiao and W. chun Feng, "Inter-block GPU communication via fast barrier synchronization," Proc. 2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), pp.1–12, 2010.
[19] W. chun Feng and S. Xiao, "To GPU synchronize or not GPU synchronize?," Proc. 2010 IEEE International Symposium on Circuits and Systems, pp.3801–3804, 2010.
[20] Y. Zhang, J. Cohen, and J.D. Owens, "Fast tridiagonal solvers on the gpu," Proc. 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '10, pp.127–136, 2010.
[21] T. Takahashi, "Mist: Media integration standard toolkit," 2009.



Pulung Waskito was born in 1986 and received the B.E. degree from Tokyo University and Agriculture and Technology in 2009. He is currently working toward the M.E. degree. His research interests are general-purpose computation on GPUs, parallel processing and high performance computing. He received the 2nd place award under Free Category in GPU Challenge 2010. He is a student member of IPSJ.

Shinobu Miwa was born in 1977 and received the Doctor of Informatics degree from Kyoto University in 2007. He worked at Tokyo University of Agriculture and Technology as an Assistant Professor. He is now an Assistant Professor at the University of Tokyo from 2011. His research interests are computer architecture, high performance computing and embedded systems. He received the best paper award in 2010 Embedded System Symposium. He is a member of IPSJ and JSAI.

Yasue Mitsukura received the M.E. degree from Okayama Prefectural University in 1999 and D.E. degree from the University of Tokushima in 2001. She worked at the University of Tokushima and Okayama University as an Assistant Professor and a Lecturer, respectively. Since 2005, she is an Associate Professor at Tokyo University of Agriculture and Technology. Her current research interests are image processing, EEG analysis and signal processing. She is a member of IEEE, SICE and RISP.



Hironori Nakajo was born in 1961 and received the B.E. and M.E. degree in Electrical Engineering from Kobe University in 1985 and 1987, respectively. He was a Research Associate at Kobe University in 1989. He worked at Center for Supercomputing Research and Development (CSRD) of the University of Illinois at Urbana-Champaign as a Visiting Research Assistant Professor from 1998 to 1999. He is an Associate Professor at Institute of Engineering, Graduate School Tokyo University of Agriculture and Technology since 1999. His research interests are computer architecture, parallel processing, cluster computing and reconfigurable computing. He is a member of IPSJ, IEEE and ACM. (Doctor of Engineering).