Parallelization of an algorithm for automatic classification of medical data

Victor M. Garcia-Molla, Addisson Salazar, Gonzalo Safont, Antonio M. Vidal, Luis Vergara

Victor M. Garcia-Molla[1], Antonio M. Vidal: Department of Information Systems and Computing (DSIC), Universitat Politècnica de València, 46022 SPAIN.
Email: vmgarcia@dsic.upv.es, avidal@dsic.upv.es

Addisson Salazar, Gonzalo Safont, Luis Vergara: Institute of Telecommunications and Multimedia Applications (ITEAM), Universitat Politècnica de València, 46022 SPAIN.
Email: asalazar@dcom.upv.es, gonsaar@upvnet.upv.es, lvergara@dcom.upv.es

Abstract. In this paper, we present the optimization and parallelization of a state-of-the-art algorithm for automatic classification, in order to perform real-time classification of clinical data. The parallelization has been carried out so that the algorithm can be used in real time in standard computers, or in high performance computing servers. The fastest versions have been obtained carrying out most of the computations in Graphics Processing Units (GPUs). The algorithms obtained have been tested in a case of automatic classification of electroencephalographic signals from patients.

Keywords: High Performance Computing; Bioinformatics; Automatic Classification; ICA (independent component analysis); SICAMM; GPU Computing

1 Introduction

Modern clinical monitoring systems usually deliver large amounts of data, which in some cases must be processed in real time. Typical examples of such medical procedures are electroencephalography (EEG), functional magnetic resonance imaging (fMRI), etc.. Furthermore, sometimes the medical data must be heavily processed, for example, the processing of medical images or the spectral analysis of EEG data. Very often the amount of data cannot be handled appropriately (in a reasonable time) by standard computers and parallel computing becomes necessary for achieving acceptable computing times [1,2,3]. This is especially true when real time computing is needed.

[1] Corresponding author vmgarcia@dsic.upv.es

In this paper, we use the SICAMM (Sequential Independent Component Analysis Mixture Modeling) algorithm, a Bayesian classification procedure [4]. The case study tackled in this paper is the analysis of EEG signals in epileptic patients taken during memory and learning neuropsychological tests. Since this case study provides large amounts of output data, initial versions of the SICAMM algorithm (written in Matlab language [12]) would need large, unacceptable computing times.

This paper describes several parallelization techniques that are applied to the SICAMM algorithm. The parallel algorithms proposed have been implemented in different hardware platforms so that the system can be adapted to different requirements. The best results have been obtained using GPUs for the heaviest computations. Moreover, the GPU utilization has allowed an increase in the performance of SICAMM algorithm, making it possible its use in real-time applications.

The rest of the paper is structured as follows. Section2 describes the SICAMM algorithm. Section 3 discusses the different optimizations and parallelization applied to the SICAMM algorithm. Section 4 presents a case study of EEG signal processing where the different implementations were applied. Section 5 includes an analysis of the results in terms of efficiency. Conclusions and future work are presented in Section 6.

2 State of the art: Automatic classification, Sequential ICAMM

The automatic classification problem can be described as follows: let $x_i \in \mathbb{R}^M$, $i = 1,2, \ldots.$ *Nx* be the observed data, M the number of data sources and *Nx* the number of signal to be processed. Each data vector x_i may belong to one of K different classes. A procedure is sought that classifies the data, i.e. a procedure that indicates if the vector x_i belongs to one of the k existing classes. The output of the classification is given in a vector of class assignments, $Ce \in \mathbb{R}^{Nx}$, such that if the i-th data vector x_i is found to belong to the j-th class ($\text{j} = 1, \ldots, K$), then $Ce(i)$ is given the value j. There are many automatic classification methods (neural networks, support vector machines, k-nearest neighbors, etc. see for example [7]). Here, we only consider and optimize the SICAMM method [4].

We will start by describing the ICAMM (Independent Component Analyzer Mixture Modeling) method. ICAMM is an automatic classification method where several classes are considered and each class is modeled using an independent component analyzer (ICA). ICAMM has been applied to a number of real-world applications, for instance, non-destructive testing, image processing, and change detection (see [8-10] and their references). The SICAMM algorithm is the extension of ICAMM to the case where data has sequential dependence i.e., when x_i depends in some way on the values of $x_j, j < i$.

Let us review the main concepts of the SICAMM algorithm. An ICA is formulated as a linear model of the observed data, the vectors $x_i \in \mathbb{R}^M$. These vectors are assumed to be a linear transformation of a vector of sources $s_i \in \mathbb{R}^M$ given by a mixing matrix $\in \mathbb{R}^{M \times M}$, as $x_i = A\, s_i$.

If the mixing matrix A is invertible (with $W = A^{-1} \in \mathbb{R}^{M \times M}$ being the demixing matrix), we can express the joint probability density $\mathrm{p}(x_i)$ in terms of the product of the marginal densities of the elements of $\mathrm{p}(s_i)$, as $\mathrm{p}(x_i) = |\det W|\ \mathrm{p}(s_i)$, where $s_i = W\, x_i$. The general expression of ICAMM requires some bias vectors to separate the components of the mixture of K classes or data groups. To do this, each class will have its own mixing matrix (A_k) and its own bias vector (b_k). Therefore, if x_i belongs to class k ($Ce(i) = k$), then $x_i = A_k\, s_{k,i} + b_k$, where it is assumed that x_i belongs to class k, denoted by $Ce(i)$. A_k and s_k are, respectively, the mixing matrix and the source vector of the ICA model of class k, and b_k is the corresponding bias vector.

SICAMM extended the mixture model to consider time dependencies. This is modelled through transition probabilities, π_{kj}, which give the probability that the i-th data vector belongs to class k given that the $i-1$-th data vector belongs to class j: $\pi_{kj} = P(Ce(i) = k|Ce(i-1) = j)$. An initial description of the SICAMM algorithm in its more general form (K classes) is shown as Algorithm 1.

```
1  Algorithm 1: SICAMM with $K$ classes:
2  Input: demixing matrices $W_1, W_2, \ldots W_K$; bias vectors $b_1, b_2, ,\ldots, b_K$ and signals $x_i$, $i = 1 \ldots Nx$
3  Output: vector $Ce$
4  /* Initialization; $i = 1$ , Classification of $x_1$ */
5  Calculate sources for each class, $s_k = W_k(x_1 - b_k)$, $k = 1 \ldots K$
6  Calculate posterior probabilities $P(Ce(1) = k|x_1)$ using an ICAMM algorithm (e.g., [11]):

       $$P(Ce(1) = k|x_1) = \frac{|\det W_k|\ p(s_k)\ P(Ce(1) = k)}{\sum_{j=1}^{K} |\det W_j|\ p(s_j)\ P(Ce(1) = j)}$$

7  Initial signal $x_1$ is assigned to the class with maximum posterior probability
8  /* classification of $x_i, i > 1$ */
9  for $i = 2 \ldots Nx$, with $Nx$ being the number of signals:
10     Select current signal, $x_i$, and build $X_i = [x_1, \ldots, x_i]$
11     Calculate sources for each class, $s_k = W_k(x_i - b_k)$, $k = 1 \ldots K$
12     Calculate conditional class probabilities using:

       $$P(Ce(i) = k|X_{i-1}) = \sum_{j=1}^{K} \pi_{kj}\ P(Ce(i-1) = j|X_{i-1})$$

13     Calculate posterior probabilities using:

       $$P(Ce(i) = k|X_i) = \frac{|\det W_k|\ p(s_k)\ P(Ce(i) = k|X_{i-1})}{\sum_{j=1}^{K} |\det W_j|\ p(s_j)\ P(Ce(i) = j|X_{i-1})}$$

14     Current signal $x_i$ is assigned to the class with maximum $P(Ce(i) = k|X_i)$
15 end for
```

3 Optimization and parallelization

In this section, we discuss the implementation of the SICAMM algorithm plus the different optimizations. We start with an initial implementation of the SICAMM algorithm, given in pseudo code.

We restrict ourselves to the case where two classes ($K = 2$) are considered. The source probabilities $p(s_1)$ and $p(s_2)$ ($ps1$ and $ps2$ in Algorithm 2) are computed using two sets of (correctly) pre-classified signals, $S_1 \in \mathbb{R}^{M \times Nf1}$ and $S_2 \in \mathbb{R}^{M \times Nf2}$. Therefore, the input data is the following: the two pre-classified sets $S_1$and S_2, the demixing matrices $W_1 \in \mathbb{R}^{M \times M}$, $W_2 \in \mathbb{R}^{M \times M}$; the bias vectors $b_1 \in \mathbb{R}^M$, $b_2 \in \mathbb{R}^M$; and the probability transition parameter $r \in \mathbb{R}$, where $r = \pi_{11} = \pi_{22}$. Accordingly, $\pi_{21} = \pi_{12} = 1 - r$. The signals to be classified are the vectors $x_i \in \mathbb{R}^M$; in a real situation, these vectors should be processed at a fast rate. A total of Nx signals are analyzed. The first version of this algorithm was developed in Matlab (R2016b), using the Statistics Toolbox function “ksdensity” to obtain the probability densities $ps1$ and $ps2$. Although

```
Algorithm 2: Optimized SICAMM for 2 classes:
Input: $W_1$, $W_2$, $b_1$, $b_2$, $S1$, $S2$, $Nx$, r , and signals $x_i$, i = 1..$Nx$
Output: vector $Ce$
/*Precomputation of sig1, sig2 */
for j=1:M
    med1=median(S1(j,:)); med2=median(S2(j,:))
    sig1[j]= median(abs(S1(j,:)-med1)/0.6745)
    sig2[j]= median(abs(S2(j,:)-med2)/0.6745)
end for
dW1=det(W1); dW2=det(W2)
/*Main Loop */
for i=1:Nx
    $s_1 = W_1 * (x_i - b_1)$; $s_2 = W_2 * (x_i - b_2)$
    ps1=density(S1, $s_1$,sig1); ps2=density(S2, $s_2$,sig2)
    p1=abs(dW1)*ps1;   p2=abs(dW2)*ps2
    if (i==1)
        p=p1+p2
        PC1=p1/p; PC2=p2/p
    else
        PC1X=r*PC1+(1-r)*PC2;   PC2X=(1-r)*PC1+r*PC2
        p=p1*PC1X+p2*PC2X
        PC1=(p1*PC1X)/p;    PC2=(p2*PC2X)/p
    end if
    if PC1>PC2
        Ce(i)=1
    else
        Ce(i)=2
    end if
end for
```

the results were correct, the code was quite inefficient due to (among other circumstances) repeated pre-computations of medians and typical deviations of the pre-classified signals, S_1 and S_2. A basic (but more efficient) version of the SICAMM algorithm is presented here as Algorithm 2, where these pre-computations are made only once, outside of the main loop.

For each incoming signal x_i, the output is the value of the vector $Ce(i)$, which will be 1 if the signal x_i belongs to class 1 and 2 if the signal x_i belongs to class 2. The part of algorithm with the larger computational cost is the computation of the source probabilities $ps1$ and $ps2$. The "density" routine (Algorithm 3) is a simplified and adapted version of the Matlab "ksdensity" function. In order to obtain reasonable execution times, this algorithm was coded in C language.

```
1 Algorithm 3: density
2 Input: S, s, sig; Output: ps
3 [M,Nf]=size(S)
4 weight=1.0/Nf
5 Pi_constant=1/sqrt(2*PI)
6 /* computation of the density for each component s(j)
7 for each s(j),  the density probability is out(j)*/
8 for j=1:M
9       u=sig(j) * pow(4/(3*Nf),(1.0/5.0))
10       aux=0
11      for  i=1:Nf
12              z=(S(j,i)-s(j))/u
13              aux=aux + exp(-0.5*z*z)*Pi_constant
14      end for
15      out(j)=aux*weight/u
16 end for
17 /* computation of the joint probability for all the vector s */
18 ps=1.0
19 for j=1:M
20      ps=ps*out(j)
21 end
```

Some interesting aspects of Algorithms 2 and 3 are commented on below:

- The initial iteration of the algorithm (line 20, for the first signal) is slightly different from the rest of the iterations in order to correctly initialize the probabilities $PC1$ and $PC2$.
- For each incoming signal, the most computationally expensive part is the execution of the "density" function (Algorithm 3). The computational cost of Algorithm 3 for each signal (discarding lower order terms) can be established by examining lines 12 and 13 in Algorithm 3. These two lines are executed $M \cdot Nf$ times, and there are six floating point operations and one call to the exponential function in these lines.

Therefore, the cost per signal is $6 \cdot M \cdot Nf$ flops and $M \cdot Nf$ evaluations of the exponential function. Algorithm 3 is called twice with each signal, one with $S1 \in \mathbb{R}^{M \times Nf1}$ and the other with $S2 \in \mathbb{R}^{M \times Nf2}$. Therefore, the theoretical cost of the whole "density" algorithm for a single entry signal x_i can be established as $6 \cdot M \cdot (Nf1 + Nf2)$ flops and $M \cdot (Nf1 + Nf2)$ exponentials.

- Algorithm 3 is easily parallelizable in different ways. The inner loop ("For i=1:Nf", line 11) can be parallelized using, for example, standard OpenMP directives [13]. The outer loop ("For j=1:M", line 8) can also be executed trivially in parallel. In such situations, usually the best strategy is to parallelize the outer code using OpenMP (so that each "j" index is processed by a single core), and to use compiler directives to force the compiler to "vectorize" the inner loop. This means that, in each core, the computation is accelerated by using vector registers and vector instructions, such as the AVX instruction set, which can carry out several operations in a single clock cycle [14]. The parallel version tested in Section 5 was parallelized in this way.

3.1 Block version.

Algorithm 2 must process incoming signals sequentially because of the dependence of the probabilities $PC1X$ and $PC2X$ on the accumulated probabilities $PC1$ and $PC2$, which come from former iterations. However, the computational cost of the code where these probabilities are used (lines 16 to 28) is minimal; the largest computational cost is in the calls to the "density" function and, to a lesser extent, in the matrix-vector products in line 13. On the other hand, the computations carried out in "density" for an incoming signal, and the matrix-vector products, are completely independent from the computations for other signals. Therefore, it is possible to group several incoming signals in a block and perform several calls to "density" (possibly in parallel), and later on, process the output of the "density" function for all of the signals of the block, obtaining the corresponding values of the Ce vector.

A block version of Algorithms 2 and 3 was developed where the signals are processed in blocks of B_{size} signals. The use of blocks improves the memory traffic, and allows the matrix-vector products in line 13 to be embodied as matrix-matrix products, which are significantly more efficient [14]. They are carried out using calls to BLAS routines [15].

This version obtained good performance, but it cannot be presented here due to lack of space. Nevertheless, the best results were obtained with GPU-accelerated versions, which are described in the next section.

3.2 GPU version

Nowadays GPUs are used in many science fields where heavy computations are required. These devices, which were originally designed to handle the graphic interface with the computer user, have had strong development driven by the videogame market.

Therefore, GPUs have actually become small supercomputers. The main feature of GPUs (considered as computing devices) is that they have a large number of small, relatively slow cores (slow compared with CPU cores). The number of cores in a NVIDIA GPU ranges today from less than 100 in cheap GeForce cards to around 4000 cores in a Tesla K40.

The use of GPUs for general-purpose computation received an enormous push with the launch of CUDA, a parallel programming environment for GPUs created by NVIDIA. CUDA allows for relatively easy programming of the GPUs by using extensions of the C language (it is also possible to program in other languages). CUDA programs look like standard C programs, but they have special routines to send/receive data from the GPU, and allow special pieces of code called "kernels", which contain the code to be executed in the GPU. A detailed description of CUDA can be found in many recent papers; we will assume a basic knowledge of CUDA from the reader in order to avoid another lengthy description of the basics of CUDA programming [6].

3.2.1 Description of the main GPU Algorithm

The idea behind our GPU implementation is: 1) to process the signals in blocks, processing them as matrices with B_{size} columns, 2) to leave the sequential part of the main loop of Algorithm 2 (lines 16 to 28) untouched, and 3) to use the GPUs for the bulk of the computation (lines 13 to 14).

The main GPU algorithm is Algorithm 4. The first change is that the data must be sent to the GPU; the data that must remain in the GPU all of the time (variables $W1$,$W2$,$b1$,$b2$,$S1, S2, sig1, sig2$) is sent to the GPU before the start of the main loop (line 9). Then the main loop starts like Algorithm 2, but the signals are grouped in blocks of B_{size} columns which are sent to the GPU. The matrix-matrix products are carried out using the CUBLAS "cublas_dgemm" routine [17]. The calls to "density" are rewritten as CUDA kernels that process a block with B_{size} signals in each call. A value of $B_{size} = 100$ has shown to be adequate to obtain good performance with this routine. These CUDA kernels are described in the next section. The goal is the same, to compute the probabilities $ps1$ and $ps2$. Once these probabilities are computed, they are sent back to the CPU, where the rest of the algorithm is identical to Algorithm 2.

```
Algorithm 4: GPU SICAMM for 2 classes.
Input: $W1, W2, b1, b2, S1, S2, x, Nx, B_{size}$
Output: vector Ce
/*Precomputation of sig1, sig2 like in Algorithm 2 */
Initial_iteration(PC1,PC2, $x_1$)
indce=2
/*Main Loop */
Send Data to GPU: $W1, W2, b1, b2, S1, S2, sig1, sig2$
for i=2:$B_{size}$: $Nx$
    $\mathrm{xB_i} = x(:, i: i + B_{size} - 1)$        /*select B_size signals */
    Send $\mathrm{xB_i}$ to GPU
    compute in GPU $sB_1 = W_1 * (\mathrm{xB_i} - \mathrm{repmat}(b_1, 1, B_{size}))$ /*CUBLAS call*/
    compute in GPU $sB_2 = W_2 * (\mathrm{xB_i} - \mathrm{repmat}(b_2, 1, B_{size}))$ /*CUBLAS call*/
    out1=density_block_GPU(S1, $sB_1$,sig1)  /*kernel with with M*$B_{size}$ blocks */
    out2=density_block_GPU(S2, $sB_2$,sig2)  /*kernel with M*$B_{size}$ blocks */
    ps1=product_reduction_GPU(out1)        /*kernel with $B_{size}$ blocks */
    ps2=product_reduction_GPU(out2)        /*kernel with $B_{size}$ blocks */
    Send ps1,ps2 to CPU.
    for j=1:B_size
        p1=abs(det(W1))*ps1(j);  p2=abs(det(W2))*ps2(j)
        PC1X=M(1,1)*PC1+M(2,1)*PC2
        PC2X=M(1,2)*PC1+M(2,2)*PC2
        p=p1*PC1X+p2*PC2X
        PC1=(p1*PC1X)/p; PC2=(p2*PC2X)/p
        if PC1>PC2
            Ce(indce)=1
        else
            Ce(indce)=2
        end if
        indce=indce+1
    end for
end for
```

3.2.2 Description of the “density” algorithm for GPU

The “density” algorithm must be rewritten as a CUDA kernel. The work unit in the GPU is the thread. The cores of the GPU can execute these threads concurrently so there can be thousands of threads running in a GPU at the same time. GPU threads can be organized in thread blocks that can be one-, two-, or three-dimensional. The thread blocks can be visualized as “teams” of blocks working together with some shared information. The kernels are executed in parallel by all of the threads in all of the blocks, but they execute their tasks on different data. The blocks can also be organized in grids, which can also be one-, two-, or three-dimensional. Clearly, there is plenty of flexibility.

We have chosen the following scheme to obtain a kernel called “density_block_GPU” equivalent to “density”, but computes the probabilities of B_{size} signals in a single call to the kernel. We have chosen one-dimensional blocks, with 512 threads each. Each block of threads is responsible for the computation of an $out(j,k)$ value. In other words, each block performs computations analogous to the inner loop (“For i=1:Nf”) in “density”. This loop is a reduction, where all the “aux” values must be added to obtain the final $out(j,k)$ value. This reduction has been programmed using a simplified version of the algorithm for reductions in CUDA proposed by Mark Harris [18]. All of the threads of the block cooperate using shared memory.

```
Algorithm 5: density_block_GPU (CUDA kernel)
Input: S, sB, sig, $B_{size}$; Output: $out$
[M,Nf]=size(S)
weight=1.0/Nf
Pi_constant=1/sqrt(2*PI)
j=BlockIdx.x;  /* BlockIdx.x is a CUDA built-in function giving the Block row in the grid of blocks (of threads)*/
k=BlockIdx.y /* BlockIdx.y is a CUDA built-in function giving the Block column in the grid of blocks (of threads)*/
u=sig(j) * pow(4/(3*Nf),(1.0/5.0))
aux=0
/*All threads of block (j,k) cooperate in the “parallel for” to compute aux  */
parallel_for  i=1:Nf
        z=(S(j,i)-sB(j,k))/u
        aux=aux + exp(-0.5*z*z)*Pi_constant
end_parallel_For
out(j,k)=aux*weight/u
```

Then we use a two-dimensional grid of blocks of threads of size $M \cdot B_{size}$. The number of rows of the grid is M, and each row (the blocks of threads of that row) performs computations that are analogous to the “for j=1:M“ loop (line 8, Algorithm 3) . Finally, the number of columns of the grid is B_{Size}, and each column (the blocks of threads of that column) of the grid deals with a column of the B_{size} blocks of signals being processed. Then, the block of the grid with index (j,k) computes the value $out\ (j,k)$. A simplified version of the kernel is shown in Algorithm 5.

Lines 18 to 21 of the “density” function involve a further reduction, which is needed to compute the final ps values . In the GPU version, this computation must be carried out for B_{size} vectors. For the sake of simplicity, it was more convenient to carry out this reduction in a different kernel. Thus, the “density_Block_GPU” returns the matrix “out”, of size $M \cdot B_{Size}$, and a different kernel (Algorithm 6) carries out the final reduction through two kernel calls (lines 16 and 17 of Algorithm 4).

```
Algorithm 6: product_reduction_GPU (CUDA kernel)
Input: out; Output: ps
[M,B_size]=size(out)
k=BlockIdx.x
ps(k)=1.0
/*All threads of block k cooperate in the “parallel for” to compute ps(k)*/
parallel_for  j=1:M
        ps(k)=ps(k)*out(j,k)
end_parallel_For
```

4 Case study

As mentioned above, we considered the analysis of EEG signals taken from six epileptic patients that were performing a memory and learning neuropsychological test. This test was performed as part of the clinical evaluation of the patients.

The signals were recorded using an EEG device of 19 channels with a sampling frequency of 500 Hz positioned according to the $10-20$ system. The signals were filtered and split into epochs of 0.25 second length (i.e., window size of 125 with 124 samples of overlap). This short length was selected in order to ensure that all stages of the test (some of which were very short) were spread over multiple epochs, thus improving parameter estimation. The theta-slow-wave-index (TSI) was estimated for each epoch as a feature for classification. (e.g. see [19], for its definition and computation). The neuropsychological test was drawn from the Wechsler Adult Intelligence Scale (WAIS, [20]) suite. WAIS is designed to measure intelligence in adults and older adolescents.

The specific sub-test of WAIS used was the Digit Span, which is divided into two stages. In the first stage, called Digit Span Forward, the participant is read a sequence of numbers and then is asked to recall the numbers in the same order. In the second stage, Digit Span Backward, the participant is read a sequence of numbers and is asked to recall the numbers in reverse order. In both stages, the objective is to correctly recall as many sequences as possible. The test is repeated a number of trials until the subject fails two consecutive trials. In the case of Subject #5, the total number of trials performed was 30, of which the data of the first 14 trials was used for training and the data of the last 16 trials was used for testing the developed algorithms. The TSI index was recorded with the goal of classifying the phases of stimuli presentation (phase 1) and subject response (phase 2).

The classification accuracy obtained for the six subjects is shown in Table 1. The results obtained have reasonable accuracy.

Table 1. Classification accuracy results

Subject #	1	2	3	4	5	6
Classification accuracy %	61.05	82.44	78.01	81.60	77.85	80.41

The case selected for testing the algorithms was the data from subject #5. In this case, the classification accuracy was 77.85% and the total number of EEG signals was 115886 (231.77 seconds). Fifty per cent of this data (57943 signals) was used for the training stage of the algorithm and the rest (57943 signals) was used for testing. Thus, the dimensions of the parameters were the following: $k = 1,2$ (1: stimuli presentation; 2: subject response); matrices $W1$ and $W2$ of size $19x19$; bias vectors $b1$ and $b2$ of size $19x1$; transition probability parameter r; signals of the first class $S1$ of size $19x30654$; and signals of the second class S, of size $19x27289$.

5 Analysis of results in terms of efficiency

The data of the experiment described above for Subject #5 was processed with the initial, sequential, and parallel versions. The results of all of the versions were identical (in terms of classification) in all of the cases. We used two machines with different characteristics. The first one was a standard desktop computer with a core-i7 Intel CPU, with 4 cores, 10 GB of RAM, and a GeForce GT 530 GPU with 96 cores. The second computer is a relatively expensive machine, with a Xeon CPU with 24 cores, and a k40 Tesla NVIDIA GPU with 2880 cores.

In the standard desktop, we tested the following versions of the code: single core (to evaluate the benefits of parallelization); version parallelized with OpenMP, running with 4 cores and GPU version running in a GT530 card.

The results are summarized in Table 2:

Table 2. Execution time for the experiment on a standard desktop computer.

Version	Single core	4 cores	GPU GT530
Time (seconds)	951	320	126

This table shows the effect of the different parallelization/optimizations. Since the total duration of the experiment was 231.77 seconds, the last version (with a cheap GPU) would allow real-time processing of the data.

We tested two versions on the second machine: v1: version parallelized with OpenMP, running with 24 cores, and v2 :GPU version running in a k40 Tesla card. The results are shown in Table 3.

Table 3. Execution time for the experiment on a high performance server.

Version	v1:24 cores	v2:GPU k40
Time (seconds)	104	14,5

In order to obtain a better evaluation of the computational cost, we generated different subsets of the EEG data of Subject #5 and tested the code by varying the “size” of the problem. Taking into account that the computational cost per signal received was $6 * M \cdot (Nf1 + Nf2)$ flops and $M \cdot (Nf1 + Nf2)$ exponentials, we defined the “size” of the problem as $M \cdot (Nf1 + Nf2)$. We fixed the number of incoming signals (20000) and executed the different versions of the code for increasing values of problem size: $100000, 300000, 600000$, and 800000 corresponding to $2, 5, 10$, and 14 EEG channels from the total of the 19 EEG channels measured. The results are summarized in Figures 1 and 2.

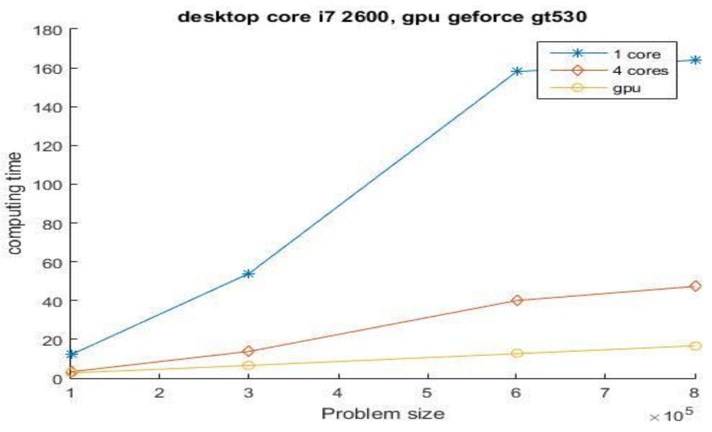

Fig. 1. Computing time (seconds) on a standard desktop

ICCS Camera Ready Version 2019
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-22744-9_1

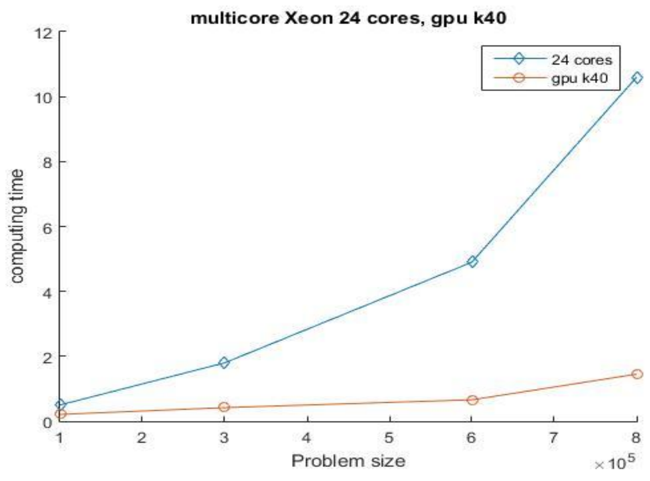

Fig. 2. Computing time (seconds) on a high performance computing server

6 Conclusion

The results show the large reduction of computational time due to the proposed parallelization. GPU versions are especially appropriate in problems of this kind. Specifically, a small laptop with a GPU with some data-acquisition equipment may be enough to register and process large amounts of biomedical data in an affordable period of time.

The approach has been illustrated in the automatic classification of electroencephalographic signals from epileptic patients that were performing a neuropsychological test. The state-of-the-art SICAMM algorithm has been considered due to its computational complexity and good results. Moreover, it is especially suited to parallelization. This parallelization opens up the use of SICAMM in real time on a realistic clinical setting. There are many possible applications for this setting, for instance, the real-time detection of epileptic seizures would allow the activation of certain clinical procedures or analyses.

As a future work, the proposed approach might be implemented in wearable devices for medical applications, thus facilitating fast, real-time monitoring and diagnosis.

Acknowledgements

This work was supported by Spanish Administration (Ministerio de Economía y Competitividad) and European Union (FEDER) under grants TEC2014-58438-R and TEC2017-84743-P; and Generalitat Valenciana under grants PROMETEO II/2014/032 and PROMETEO II/2014/003.

References

1. Shi, L., Liu, W., Zhang, H., Xie, Y., Wang, D.: A survey of GPU-based medical image computing techniques. Quantitative Imaging in Medicine and Surgery 2(3):188-206. (2012) doi:10.3978/j.issn.2223-4292.2012.08.02.
2. Montagnat, J., Bellet, F. , Benoit-Cattin, H., Breton, V. Brunie, L. Duque, H. Legré, Y. Magnin, I. E. Maigne, L, Miguet, S., Pierson, J. -M., Seitz, L., Tweed, T.,: Medical Images Simulation, Storage, and Processing on the European DataGrid Testbed, Journal of Grid Computing, vol.2, 4, (2005), doi=10.1007/s10723-004-5744-y
3. Saxena, S., Sharma, S., Sharma, N.: Image registration techniques using parallel computing in multicore environment and its applications in medical imaging: An overview, In: International Conference on Computer and Communication Technology (ICCCT) (2014)
4. Salazar, A., Vergara, L., Miralles, R.: On including sequential dependence in ICA mixture models, Signal Processing, vol. 90, 2314-2318, (2010).
5. Niedermeyer, E., Lopes da Silva, F.: Electroencephalography: Basic Principles, Clinical Applications, and Related Fields, Fifth Edition, Lippincott Williams & Wilkins, (2005).
6. Cuda C Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
7. Common P., Jutten, C.: Handbook of Blind Source Separation: Independent Component Analysis and Applications. Oxford, UK: Academic Press, (2010).
8. Salazar, A., Igual, J., Vergara, L., Serrano, A.: Learning hierarchies from ICA mixtures. In: IEEE International Conf. on Neural Networks, IJCNN 2007, 2271-2276, (2007).
9. Salazar, A. : On Statistical Pattern Recognition in Independent Component Analysis Mixture Modelling. New York: Springer, (2013).
10. Safont, G., Salazar, A., Vergara, L., Gomez, E., Villanueva, V.: Probabilistic Distance for Mixtures of Independent Component Analyzers, IEEE Transactions on Neural Networks and Learning Systems, (29) 4, 1161-1173, (2018).
11. Salazar, A., Vergara, L., Serrano, A., Igual, J. : A general procedure for learning mixtures of independent component analyzers, Pattern Recognition, (43) 1, 69-85, (2010).
12. The Mathworks Inc., MATLAB R14 Natick MA (2004).
13. http://www.openmp.org/
14. Intel(R) AVX - Intel(R) Software Network, http://software.intel.com/en-us/avx/
15. Golub, G. H., Van Loan, C. F. : Matrix Computations. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, (1996).
16. Anderson, E., Bai, Z. , Bischof, C., Blackford, S., Demmel, J. , Dongarra, J. , Du Croz, J. , Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia, (1999).
17. CUBLAS http://docs.nvidia.com/cuda/cublas/index.html
18. Harris, M.: Optimizing parallel Reduction in CUDA, https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.
19. Motamedi-Fakhr, S., Moshrefi-Torbati, M. , Hill, M., Hill, C.M. , White, P.R. :Signal processing techniques applied to human sleep EEG signals-A review, Biomedical Signal Processing and Control, vol. 10, 21-33, (2014).
20. Strauss, E.: A Compendium of Neuropsychological Tests. Oxford University Press, (2006).