Journal of Computer Engineering
ISSN 10003428

Original Research Article
Vol 8, Issue 4, 2019 | ISSN: 1000-3428

Optimization of EEG Feature Selection in High-Dimensional Spaces using Multi-objective Methods and CPU-GPU Architectures

Sofia Rodriguez · Alexander Patel

Sofia Rodriguez, Department of Biomedical Engineering, University of Southern California, Los Angeles, CA, USA; Alexander Patel, Advanced Computing Research Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract Many bioinformatics applications that analyse large volumes of high-dimensional data comprise complex problems requiring metaheuristics approaches with different types of implicit parallelism. For example, although functional parallelism would be used to accelerate evolutionary algorithms, the fitness evaluation of the population could imply the computation of cost functions with data parallelism. This way, heterogeneous parallel architectures, including Central Processing Unit (CPU) microprocessors with multiple superscalar cores and accelerators such as Graphics Processing Units (GPUs) could be very useful. This paper aims to take advantage of such CPU-GPU heterogeneous architectures to accelerate Electroencephalogram (EEG) classification and feature selection problems by evolutionary multi-objective optimization, in the context of Brain Computing Interface (BCI) tasks. In this paper, we have used the OpenCL framework to develop parallel master-worker codes implementing an evolutionary multi-objective feature selection procedure in which the individuals of the population are dynamically distributed among the available CPU and GPU cores.

Parallel Architectures · Memory Access Optimization

Many bioinformatics applications involve high dimensional data mining problems that comprise tasks such as classification, clustering, optimization, and feature selection. Electroencephalogram (EEG) classification is a good example of those applications that process high-dimensional patterns and require feature selection techniques to remove noisy, irrelevant features or to improve the learning accuracy and result comprehensibility, especially whenever the number of features in the input patterns is higher than the number of available patterns. The proposed approach to EEG classification for BCI tasks [1] includes an evolutionary multi-objective optimization algorithm and a clustering algorithm applied to a set of high-dimensional patterns that could require highvolume storage. Thus, as many other bioinformatics applications, the one here considered poses problems that show different types of inherent parallelism.

This paper aims to provide an insight into the design of efficient parallel procedures for high-dimensional classification and optimization tasks, to be executed in heterogeneous parallel architectures involving multiple general-purpose superscalar multicore CPUs and accelerators (mainly GPUs). As they constitute the present mainstream approach to take advantage of technology improvements [2], their use has been proposed in many previous papers on parallel metaheuristics and evolutionary computation [3–5]. Nevertheless, the parallelization on a heterogeneous platform of a whole data mining application with the characteristics of our target application is less frequent in the literature. In our previous papers [6,7], we proposed a multi-objective feature selection that implements both functional and data parallelism which can be executed either in a GPU or in CPU multiple superscalar CPU cores. Moreover, in [7] the effect of memory access optimization on GPU implementations has been demonstrated. In the present paper, a heterogeneous parallel approach that dynamically distributes the evaluation of the individuals among both the GPUs and the CPU cores is proposed, which is also able to efficiently take advantage of the data parallelism available in GPUs. In addition, we analyse the effects on the performance of memory accesses.

After this introduction, Sect. 2 describes the evolutionary multi-objective optimization approach to the feature selection procedure whose implementation has been parallelized. That section also summarizes the characteristics of the main approaches to parallelize an evolutionary algorithm. Section 3 analyses the main issues related with the development of efficient parallel evolutionary algorithms in heterogeneous platforms, and the details of our proposed OpenCL [6] codes. Then, Sect. 4 describes

How to cite: Sofia Rodriguez · Alexander Patel (2019). Optimization of EEG Feature Selection in High-Dimensional Spaces using Multi-objective Methods and CPU-GPU Architectures. Journal of Computer Engineering, 8(4), 1–27. DOI: https://doi.org/10.5281/zenodo.19340041

the experimental results and compares the behaviour of the considered alternatives, Sect. 5 analyses previous works in this area to give an insight into the context of the paper, and Sect. 6 summarises the conclusions.

1 Multi-objective feature selection

This paper deals with parallel processing of feature selection in unsupervised classification of patterns characterised by a high number of features. As the number of patterns to be classified is usually lower than the number of features, we have to cope with a curse of dimensionality [8] problem. Thus, the most relevant features should be selected to achieve an adequate performance of the classifier, decrease the computational complexity of the classification and remove irrelevant or redundant features. Nevertheless, optimal feature selection is an *NP*-hard problem [9] that requires efficient metaheuristics in high-dimensional classification problems. Here, we apply multi-objective optimization to feature selection in applications with a large number of features and propose the use of heterogeneous parallel architectures to accelerate it.

The use of multi-objective optimization in data mining applications is surveyed in [10,11], and the benefits from a multi-objective approach to feature selection in both supervised and unsupervised classification have been reported elsewhere [12–14]. Indeed, [14] shows that feature selection in unsupervised learning problems is inherently a multiobjective problem. Moreover, as the number of involved features in the applications here considered is large, a multiobjective optimization approach would imply high computational costs, which is an important issue to be considered.

Figure 1 describes our approach for feature selection in unsupervised classification of EEG patterns. A multiobjective evolutionary procedure evolves a population of individuals that codify different feature selections. Given a feature selection (an individual in the population of the evolutionary algorithm), the N_P patterns included in the database, *DS*, will be used to define the set of training patterns by choosing the components corresponding to the number of features,

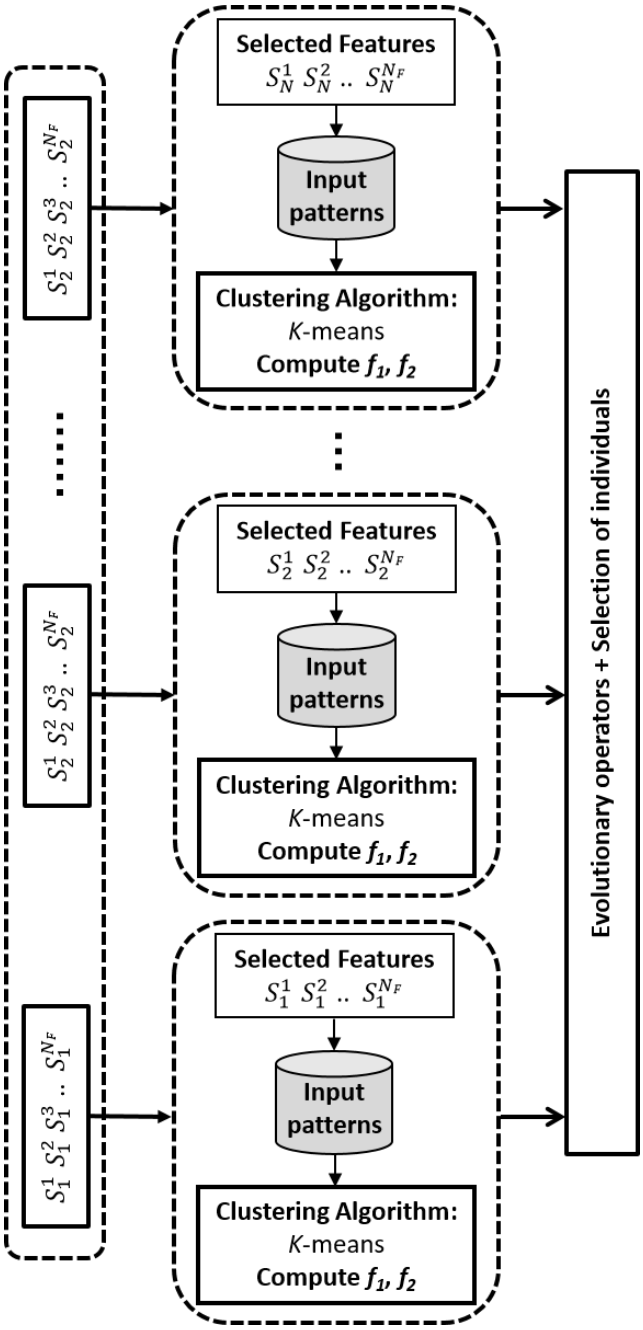

Fig. 1 Wrapper procedure for unsupervised feature selection by means of evolutionary multi-objective optimization (and *K*-means clustering)

N_F, selected. This way, the *K*-means algorithm has been applied to the N_P patterns $P_i = (p_i^1, \ldots, p_i^{N_F})(i = 1, \ldots, N_P)$ to determine the centroids $K^t(j)(j = 1,..,W)$ of the W possible clusters (W is known in our EEG classification problem, and it is equal to the number of classes). *K*-means is a well-known clustering algorithm. Its steps are described below:

1. Set t = 0 and generate W initial centroids $K^t(j)(j = 1,..,W)$ (as many centroids as clusters or classes).
2. Assign each pattern to the nearest cluster.
3. Get the new cluster centroids, $K^{t+1}(j)(j = 1,..,W)$.
4. If the end condition is not met (either changes are observed in the position of the centroids, or a given number of iterations have not been completed yet), set $t = t + 1$ and repeat steps 2 and 3. Otherwise, conclude.

Once the clusters are built by including each pattern in the cluster of its nearest centroid, the fitness of each individual in the population is evaluated by using two Clustering Validation Indices (CVIs) [15], defined as:

$$f_2 = -\sum_{j}^{W}\left(\sum \|K^t(i) - K^t(j)\|\right) \quad (1)$$

$$f_1 = \sum_{j=1}^{W}\left(\frac{1}{|C_t(j)|}\sum_{P_i \in C_t(j)}\|P_i - K^t(j)\|\right) \quad (2)$$

where (1) and (2) correspond, respectively, to the intraclass and minus the interclass distances. In these equations, $|C^t(j)|$ is the number of patterns in the cluster $C^t(j)(j = 1,..,W)$ whose centroid is $K^t(j)$, and $\|P_i - K^t(j)\|$ is the Euclidean distance between the pattern P_i and the centroid $K^t(j)$.

2.1 Application complexity

Taking into account Fig. 1, the K-means algorithm previously described and the evaluation of the clustering objectives, the following equations provide a quantitative model to understand the complexity of the application:

$$T_{Kmeans} = N\,W T_{Dist} N_P + T_W + N_P + W^2)\,T_{Dist}) \quad (3)$$

$$T_{Seq} \approx G\,\omega N_2 T_{Ns} + N T_{Ev} + N_r T_{Ni} + T_{Kmeans} \quad (4)$$

In (4), T_{Seq} gives an estimation of the time required by a sequential implementation of the code, G is the number of generations executed by the multi-objective evolutionary algorithm, and ω is the number of objectives (in our case, ω = 2). T_{Ns}, T_{Ev} and T_{Ni} respectively correspond to the determination of the non-dominated individuals (it requires the comparison of the individuals of the population by using their ω objectives); the application of the evolutionary operators (mutation, crossover, etc.) either to the N individuals of the population or to a subset of individuals in the population; and the application of a procedure to maintain a uniform distribution of individuals along the present Pareto front (the complexity of these operations in the considered procedure is taken into account through the exponent r). The last term in (4) correspond to the execution of the K-means algorithm and whose execution time is estimated in (3). K-means implements the evaluation of the two objectives for each of the N individuals of the population. In (3), T_{Dist} corresponds to the computation of the distance between a pattern and a centroid and T_W is the time required to obtain the new centroids, which depends on the number of patterns, N_P (distributed among the W clusters), and the number of components, N_F.

Finally, the expression $N_P + W^2)T_{Dist}$ estimates the cost to compute the two objective functions, f_1 and f_2, respectively, to obtain the distances between each pattern and its nearest centroid, and the distances between each pair of centroids. Both functions are defined in (1) and (2).

2.2 Alternative parallel implementations of evolutionary algorithms

This way, we have to implement a multi-objective evolutionary algorithm where the evaluation of the cost functions of the individuals in the population requires the execution of a K-means algorithm per individual and generation.

There are two main strategies to parallelize an evolutionary algorithm. The first one takes advantage of the parallel evaluation of the individuals in the population, and the second alternative implements concurrent executions of evolutionary algorithms on subpopulations. While the first alternative has the same semantics as the sequential algorithm and is implemented through a master-worker approach, the

second one changes the semantics of the sequential algorithm and could be implemented by an island approach [16]. In what follows, we present some details about the implementation of these parallel models in a CPU-GPU heterogeneous architecture.

The parallel evaluation of the individuals could be implemented on a GPU while the CPU executes the different steps of the evolutionary algorithm. Thus, the GPU is used as a coprocessor in a synchronous way and the population and fitness structures should be transferred between the GPU and the CPU. A drawback of this scheme is the number of copies (a copy in each direction per generation) through a bus, with worse bandwidth and latency than those provided by the CPU memory bus. Moreover, in the application here considered, the size of the dataset to be processed is usually large. The copy of the dataset from main memory to the GPU memory could also require a significant amount of time, although this transfer only has to be done once.

In our target application (Fig. 1), the evaluation of the cost functions for a given individual of the population implies the execution of a *K*-means algorithm, being also possible to accelerate it on GPUs. Thus, two parallelism levels have been implemented in a GPU: the parallel evaluation of the individuals and the parallel execution of the *K*-means to compute the cost functions for each individual. The other evolutionary algorithm steps are implemented in a CPU core.

The second parallel approach has not been implemented here. As it implies the execution of concurrent evolutionary algorithms on subpopulations, the evolutionary algorithm applied to each of the subpopulations assigned to a given GPU should be implemented in parallel. Each subpopulation could be implemented as a thread block, and each individual in the subpopulation, as a thread. This way, the application of selection, mutation, and crossover operators to the individuals requires the use of barriers to synchronize the threads (that implement the individuals). Although this approach could reduce the data transfers between the CPU and GPU, the synchronization requirements have to be taken into account. Moreover, these parallel evolutionary algorithms do not show the same functionality and alternatives as their corresponding sequential ones. Otherwise, the GPU memory hierarchy should be carefully managed. The global memory of the GPU is not cached and its accesses mean many additional cycles. The shared memory available for the threads in a thread block should be used to store the data structures corresponding to the subpopulation assigned to the thread block while the global memory allows the communication among subpopulations according to the devised migration policy. The implementation of these strategies in GPUs can be found in [3], including many details regarding the implementation of the topology to exchange individuals between islands, the selection of these migrants, the replacement/integration topology and the migration criterion, besides the burden of SIMD procedures to find the minimum, and the synchronization between threads required by a synchronous island model.

The parallel procedure proposed here takes advantage of GPU and CPU superscalar cores to accelerate the procedure of Fig. 1. The procedure uses one CPU core as a master that dynamically distributes the individuals evaluation among GPU cores or other CPU cores. Thus, several individuals could be evaluated in parallel in a GPU but this procedure can be also used to implement the computation of the cost functions by taking advantage of its inherent data parallelism. From (3) and (4), in our parallel implementation, a CPU core will determine the time $\omega N^2 T_{Ns} + N T_{Ev} + N^r T_{Ni}$, while the processing of the T_{Kmeans} will be distributed among the available CPU and GPU cores. Of course, the parameters T_{Dist} and T_W have different values depending on if there are assigned to a GPU or to a CPU core. In case of using a GPU core, the computation corresponding to the expression $W T_{Dist} N_P + T_W$ is also parallelized by taking advantage of the data parallel resources in the GPU. As the number of individuals in the population is usually larger than the number of available cores, the individuals are dynamically allocated to the cores that have finished the computation of their previously assigned individuals. It has to be taken into account that as the number of selected features in a solution (an individual) may change, the time required to evaluate the individuals cannot be predicted before executing the algorithm and the dynamic scheduling of individuals is required to reduce the idle time of cores once they have finished their assigned work.

In the next section, a more detailed description of our proposed procedure is given whilst in Sect. 4 its experimental behaviour is analysed.

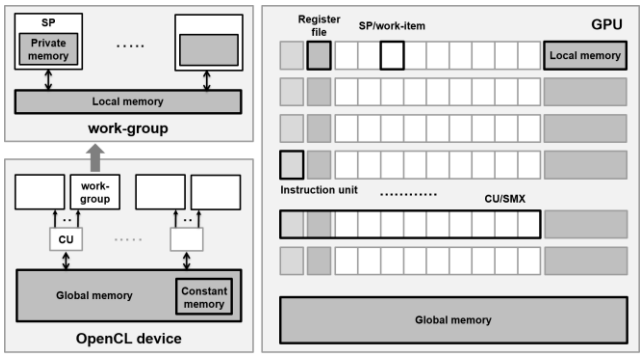

Fig. 2 Elements of the OpenCL device (left block) and schematic diagram of a GPU architecture (right block)

3 A heterogeneous parallel procedure for multi-objective feature selection

In this section, we describe the parallel implementations we have considered to take advantage of heterogeneous CPUGPU architectures. Subsection 3.1 introduces the corresponding OpenCL terms, their relationships with the devices used and the abstract memory model of OpenCL. Subsection 3.2 provides a description of our dynamic parallel implementation and finally, Subsect. 3.3 shows different strategies of optimization, especially at memory level with the objective of reducing computation time and memory consumption.

3.1 Hardware coprocessors and OpenCL

The GPU plays the role of a coprocessor connected, through a bus, to a host including multiple superscalar CPU cores that share the main memory. The basic computing elements or cores of the GPU are the so-called *Stream Processors* (SPs). They do not contain instruction units and are only able to execute scalar operations. Several SPs along with one or more instruction units and a register file comprise a multiprocessor, also called *Streaming Multiprocessor* (SMX). A GPU can include multiple SMXs, which allow the simultaneous execution of the same program on different data, i.e. Single Program Multiple Data (SPMD) model. The threads are organized within thread blocks in such a way that all threads in a block are assigned to a single SMX. Moreover, the blocks are also partitioned into *warps* containing threads with consecutive and increasing identity numbers that start together at the same program address. While the threads in a block are able to cooperate and share the instruction unit and the register file, threads in different blocks can only communicate through the off-chip memory.

We have developed our codes on OpenCL [17] which allows platform-independent parallel programming through programs executed in a host that launch functions, called *kernels*, to other OpenCL devices, such as CPUs or GPUs.

Figure 2 relates the main GPU and OpenCL framework terms and elements. A device in OpenCL is an array of functionally independent *Computing Units* (CUs) divided into *workitems*, which are the minimum units of concurrent execution. Several work-items can execute the same instruction over different data items according to a Single Instruction Multiple Data (SIMD) model, and they can be also organized as a *work-group*, in such a way that several work-groups can be distributed and executed through the available CUs. The GPU scheme previously shown can be translated to the OpenCL terms. This way, the SMXs are Computing Units, the SP cores are work-items, and the thread blocks are workgroups. The abstract memory model of OpenCL defines memory spaces that also resemble the usual memory hierarchies. Thus, the *global memory* (off-chip memory) is visible to all Computing Units in the device, as the *constant memory*, included in the global memory to store variables whose values do not change. All work-items in a given work-group share the corresponding *local memory*, while the *private memory* is only accessed by a work-item.

3.2 Scheduler and kernels description

From Fig. 1 and Sect. 2, it is clear that our application involves both evolutionary multi-objective and clustering algorithms. In [16], we have proposed several approaches to parallelize the application through different parallel evolutionary multi-optimization options but we did not parallelize the fitness computation for the individuals in the population. Here, we consider this issue by taking advantage of the GPU resources to run data parallel codes. Thus, as it is illustrated in Fig. 3, we have implemented a master-worker parallel evolutionary algorithm, in which a CPU core in the host is the master that implements the steps of the multi-objective evolutionary algorithm on the population. It also launches kernels to the GPU and CPU cores, to

evaluate the cost functions of the individuals in the population. The master core also calls the scheduler procedure to distribute dynamically the evaluations among the GPU and GPU cores, using OpenMP [18] directives. As it has been said, the GPU kernel implements two levels of parallelism: the parallel evaluation of individuals in the population and the data parallel evaluation of the cost functions for each individual.

Algorithms 1, 2 and 3 provide the pseudocodes corresponding, respectively, to the scheduler running in the master core that dynamically distributes the evaluations of individuals among the OpenCL kernels (running on CPU and GPU devices), and the CPU and GPU kernels. At the beginning of the program, the master core transfers the patterns whose features are to be selected from the host to the GPU memory. Moreover, to start each generation, the master core has to send the individuals of the population to all OpenCL devices. Once the cost functions have been evaluated, their values

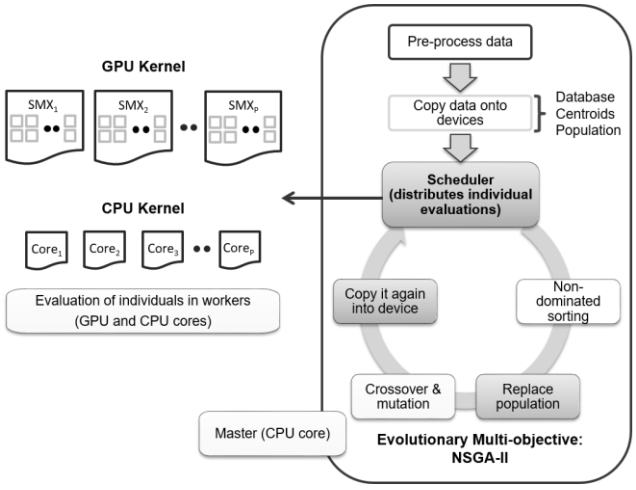

Fig. 3 Scheme of the master-worker OpenCL parallel implementation of the evolutionary multi-objective procedure for feature selection

are sent back to the master core. This way, considering a node where the CPU codes share the main memory, there is only a transfer of the patterns from the master core to the GPU devices. The master core has to send the population to the GPU devices once per generation. In each generation, the master also transfers the indices of the individuals to be evaluated, and takes the values of the evaluated functions, respectively, to and from the corresponding devices.

The pseudocode in Alg. 1 describes the main steps of the dynamic scheduling procedure to distribute the individuals among the OpenCL devices (GPU or CPU cores). It forks a thread per device to select the N_C individuals to be assigned to this device. In line 3 of Alg. 1, each thread transfers the indices of the individuals assigned to each device if the device is a GPU. Line 13 corresponds to the transfer of the corresponding cost functions once they have been evaluated. Lines 5 and 6 correspond to an atomic access to the indices of the individuals, through a pointer that is incremented according to the number of individuals assigned to the device by the corresponding thread.

The pseudocode in Alg. 2 corresponds to the CPU kernel that sequentially evaluates the N_C individuals assigned to the core by the master. In this case, a work-group is composed by only one work-item as the CPU kernel does not implement data parallel processing as GPU. In fact, it would be possible to develop a CPU kernel version which implements the SIMD model to take advantage of vectorization. In OpenCL, it could be implemented using the predefined data types (i.e. *float2*, *float4*, *float8* or *float16*) to process simultaneously 2, 4, 8
or 16 elements respectively.

Algorithm 3 shows the GPU kernel pseudocode to evaluate the fitness of the individuals (the intraclass and interclass distances defining the two cost functions of the multiobjective optimization procedure). As it has been said, both the CPU and GPU kernels evaluate individuals in parallel Algorithm 1: Scheduler pseudocode. The evaluation of the individuals are distributed among all OpenCL devices using OpenMP threads

```
1   Function scheduler(S,N,D,N_D)
    Input :A possible solution for the problem, S
    Input:Number of individuals N to be evaluated
    Input :Object D containing the OpenCL devices
    Input  :Number of available OpenCL devices, N_D
    Output:S, the new solution for the problem

    // Shared pointer between all OpenMP threads
2   Ptr ← 0
    // OpenMP parallel section with N_D devices
3   D_j ← Copy S stored in the host if D_j is a GPU
4   repeat
    /* Atomic section to update the private pointer and
       the shared pointer */
5   Priv_Ptr ← Ptr
6   Ptr ← Update the shared pointer
    // Kernel execution 7
Init the kernel arguments
```

```
8       NC ← The chunk of individuals to be evaluated
9       if Dj is a CPU then
10      SPriv_Ptr ← evaluationsCPU(S,NC,DS,K)
11      else
12      SPriv_Ptr ←
          evaluationsGPU(S,NC,DS,K,DSt)
13      S ← Copy SPriv_Ptr from Dj
14      end
15      until all N individuals are evaluated;
16      S ← normalization(S)
17      return S
18      End
```

by different work-groups, thus implementing the first level of parallelism of the algorithm (line 3 in both algorithms). Moreover, the GPU kernel also implements a second level of parallelism as each work-group is composed by warps of 32 work-items each in ~~the case of the GPUs used for the experimental part of~~ this work. This second level of parallelism corresponds to the parallel implementation of the *K*-means algorithm (lines 5-15 in Alg. 3). The expression << *work-groupID,work-itemID* >> defines the distribution of work-items in each work-group through the different steps of the *K*-means algorithm in the GPU.

3.3 Optimizations

In what follows, we describe the main details of the proposed GPU kernel that make it possible to take advantage of the data parallelism available in the evaluation of the individuals. As it has been seen in our previous paper [7], careful optimizations in the use of the GPU memory hierarchy by the different data structures would improve the performance with respect to

Algorithm 2: Pseudocode for the OpenCL CPU kernel that evaluates a chunk of individuals of the population

```
1  Kernel function evaluationsCPU(S,NC,DS,K)
       Input :A possible solution for the problem, S
       Input:Chunk of individuals to be evaluated, NC
       Input:Dataset DS: NP training patterns of NF features
       Input:Set K of W centroids randomly chosen from DS
       Output: f1(S): intraclass distances in S according to (1)
       Output: f2(S): interclass distances in S according to (2)
2  <<All work-groups,work-item 0>> 3   for i ← 1
   to NC individuals do
4      KC ← Create a copy of the centroids
5      Initialization of the mapping table, MT ← 0
6      repeat
7      MT ← Patterns in DS are assigned to the cluster
8      D ← Store nearest distance for each pattern
9      Check if any pattern has changed its assignment 10
       KC ← Update the centroids using the dataset DS
11     until stop criterion is not reached;
12     f1(S) ← intraclass(KC,DS,D)
13     f2(S) ← interclass(KC,DS)
14     end
15     return (f1(S), f2(S))
16     End
```

a more simple and direct allocation of such structures as it has been done in our first GPU approach to feature selection shown in [6]. In this paper, we also give a more complete analysis about the extent to which these optimizations have allowed efficient data parallel performances.

1. The CPU-GPU kernels receive the input parameters provided by the host code: the individuals of the population, the dataset and the initial centroids for the *K*-means algorithm. An individual, S_i, is a one-dimensional array of contiguous 0's and 1's (according to the selection or not of the corresponding feature) stored in global memory. In addition, in the GPU kernel, S_i will be copied into local memory (line 6 in Alg. 3) as this on-chip memory is faster. The global memory used is $S_{Pop} = N \times N_F$ bytes, where N is the number of individuals and N_F is the whole number of features (among which the selection is to be done). The datasets *DS* and DS^t include the N_P training patterns, each characterized by N_F features. Both sets are stored in global memory due to their large sizes, in a one-dimensional array of $N_P \times N_F$ elements normalized by the host program. In *DS* the patterns are organized in row-major order while column-major order is used in DS^t (DS^t is the transpose of *DS*). Each dataset needs $S_{DB} = 4 \times N_P \times N_F$ bytes of global memory. Instead of the W centroids randomly selected from the dataset, the indices of these centroids are copied from the host memory to the GPU constant memory: the amount of constant memory used is $S_W = 4 \times W$ bytes.

Algorithm 3: Pseudocode for the OpenCL GPU kernel that evaluates a chunk of individuals of the population

```
1  Kernel function evaluationsGPU(S,N_C,DS,K,DS^t)
       Input :A possible solution for the problem, S
       Input:Chunk of individuals to be evaluated, N_C
       Input:Dataset DS: N_P training patterns of N_F features
       Input:Set K of W centroids randomly chosen from DS
       Input :Dataset DS^t is DS in column-major order
       Output: f1(S): intraclass distances in S according to (1)
       Output: f2(S): interclass distances in S according to (2)
2  <<All work-groups,All work-items>> 3      for i ← 1 to
   N_C individuals do
4          <<work-groupID,All work-items>>
5          K_L ←
         Copy the centroids from global to local memory
6          I ← Copy individual S_i from global to local
           memory
7          Initialization of the mapping table, M_T ← 0
8          repeat
9          <<work-groupID,work-itemID >>
10         M_T ← Patterns in DS^t are assigned to the cluster
11         D ← Store nearest distance for each pattern
12         Check if any pattern has changed its
           assignment
13         <<
             work-groupID,All work-items>>
14         K_L ← Update the centroids using the dataset DS
15         until stop criterion is not reached;
16         <<work-groupID,work-item 0>>
17         f1(S) ← intraclass(K_L,DS,D)
18         f2(S) ← interclass(K_L,DS)
19         end
20         return (f1(S), f2(S))
21         End
```

Fig. 4 32 work-items copy data from global memory to local memory providing coalescent access to global memory and minimizing the memory bank conflicts

2. As the positions of the centroids are modified along the iterations of the *K*-means algorithm (otherwise the *K*means algorithm finishes), it is necessary to copy each centroid from global memory to local memory whenever a new individual is going to be evaluated (line 5 in Alg. 3). The operations of lines 5 and 6 are executed in parallel by all work-items of the corresponding work-group. Thus we can benefit from *coalescence*, a technique in which consecutive threads of a warp request data stored in global memory, in consecutive logical addresses. This technique aims to minimize the number of transaction

Fig. 5 work-items accesses to *DS* and DS^t in the different steps of the
GPU kernel

segments requested from global memory by taking advantage of the memory bus width to get multiple data in a single transaction. We have been able to use coalescence as consecutive work-items in the same work-group request data stored in consecutive logical addresses of the global memory. As Fig. 4 shows, the memory bank conflicts in local memory are minimized. When the *WI* work-items in the work-group process the first *WI* data, the next *WI* data are repeatedly requested and processed, until the whole dataset has been processed. In the CPU kernel, the only work-item in a work-group sequentially performs the copy of the centroids and individuals. The centroids need $S_{Kl} = 4 \times W \times N_F$ bytes of local memory and each individual $S_i = N_F$ bytes (W centroids, N_F features and 4 bytes per floating-point data).

3. The mapping table M_T needs $S_{MT} = N_P$ bytes of local memory (N_P is the number of patterns in the dataset *DS*). This table contains the centroid assigned to each pattern along the *K*-means iterations. The initialization (line 7 in Alg. 3) is carried out by all work-items in the same way as the previous initialization of centroids and individuals. Each pattern only stores the index of its corresponding centroid, K_j. Moreover, through the mapping table M_T, it is easier to check the algorithm convergence by taking into account whether a pattern has been assigned to another centroid (line 12), instead of doing that at the end of the iteration (if there are no changes in the centroid assignments).
4. Each work-item has to find the nearest centroid for a specific pattern by using the Euclidean distances between patterns and centroids. The dataset DS^t is stored in the GPU global memory to accelerate this task. The N_P first memory addresses of DS^t store the values of the first feature for all patterns, the following N_P memory addresses store the values of the second feature, and so on. Therefore, as each work-item handles a different pattern in a given

time, consecutive work-items will request consecutive memory addresses, allowing fully coalesced access to global memory. Moreover, when the nearest centroid to a given pattern and the corresponding distance are obtained, they can be written in, respectively, M_T and D with the minimum number of memory bank conflicts. Array D is stored in local memory including the Euclidean distances between each pattern and its closest centroid, occupying a total of $S_D = 4 \times N_P$ bytes.

5. The most complex step of K-means in terms of data parallelization is the update of the centroids (line 14 in Alg. 3). Indeed, some approaches [19,20] directly propose to perform this step sequentially in the host, although the cost per iteration associated to transferring the centroids to the host, processing them, and returning them could be too high, specially in applications with high-dimensional patterns. Thus, we use our GPU kernel and assign each work-item to add the same feature of all patterns belonging to the centroid in question. The dataset DS^t is not adequate as consecutive work-items compute consecutive features. Now, DS is used because its first N_F memory addresses contain all the features of the first pattern, the following N_F addresses contain the features of the second pattern, and so on. Thus, coalesced memory access can be achieved and the memory bank conflicts are minimized when a centroid is updated. Figure 5 shows the relation between DS and DS^t and the work-item accesses to these structures according to the steps 2 and 3 of the K-means algorithm.
6. The GPU and CPU kernels return the fitness values of the individuals (lines 17 and 18 in Alg. 3), built from two components, the intra-cluster and the inter-cluster distances given in (1) and (2) of Sect. 2.

4 Experimental results

In this section, we analyse the performance of our OpenCL
(version 1.2) code, compiled with GCC 4.8.5 and running on Linux CentOS 6.7, in a node with 32 GB of DDR3 memory and two Intel Xeon E5-2620 HT processors at 2.1 GHz including six cores per socket, thus comprising 24 threads. The node also has a Quadro K200D with 2 GB of global memory, 64 GB/s as peak memory bandwidth and two SMXs, each of them including 192 CUDA cores running at 954 MHz, thus a total of 384 cores. Moreover, there is a Tesla K20c with 5 GB of global memory, 208 GB/s as maximum memory bandwidth, and 2,496 CUDA cores at 705.5 MHz, distributed into 13 SMX, thus including 192 cores per SMX.

We have evaluated our proposed procedures on two datasets, b3600a and b480a, containing 178 patterns extracted from the datasets recorded in the BCI Laboratory

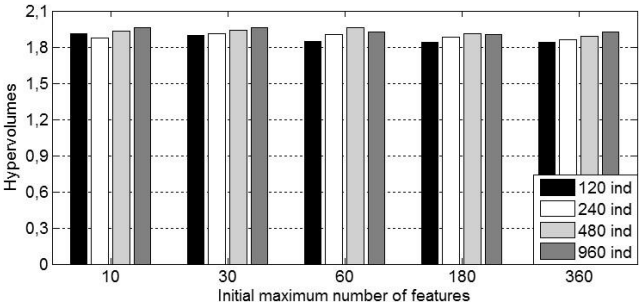

Fig. 6 Mean hypervolumes for different numbers of maximum features
(10, 30, 60, 180, 360) initially set to 1. Population sizes of (120, 240, 480, 960) individuals, dataset b3600a and 50 generations

at the University of Essex [21], and corresponding to the subject coded as 110. Each pattern in the dataset corresponds to an EEG trial. In the EEG signal obtained from each electrode, several segments are defined, which are characterized by a set of details and approximation coefficients belonging to different levels of wavelets [22]. In our case, the databases b3600a and b480a have 15 electrodes, 20 segments, and 6 levels, therefore there are 3,600 sets of coefficients, including from 4 to 128 coefficients in each set. In addition, for each set, as in [21], only one coefficient is built by computing the second moment of the coefficient distribution (variance) in each of the 3,600 sets, and normalizing the obtained value between 0 and 1. This way, 2×15×20×6 = 3,600 features constitute each pattern in the dataset b3600a, of which the first 480 features define each pattern. We have repeated each experiment 10 times, to apply Kolmogorov-Smirnov tests in order to determine whether the data follow a standard normal distribution or not. According to the result of this test, we then apply either an ANOVA test if the data follow a normal distribution or a Kruskal-Wallis test otherwise.

The implemented multi-objective optimization algorithm NSGA-II [23] uses uniform crossover with a probability of 0.75 and mutation by inversion of the selected bit with a probability of 0.25, and selection by

binary tournament. In case of crossover, each bit of the parents' chromosome is exchanged with a probability equal to 0.5, and in case of mutation, each bit is mutated with a probability equal to 0.099 if it is set to 0 or 0.001 if it is set to 1. The hypervolumes [24] are obtained with (1,1) as reference point, and the minimum values for the cost functions f_1 and f_2 are respectively 0 and -1, i. e. (0,-1). Thus, the maximum hypervolume value is 2.

4.1 Comparison of kernels and hypervolumes

Figure 6 shows the mean hypervolumes obtained after 10 repetitions per experiment for different codes and configurations. After analysing the obtained hypervolume results, it has been observed that there are no statistically significant differences with respect to the results obtained by the sequential procedures with the same number of individuals in the population and generations. This is expected, as our OpenCL codes correspond to alternative parallel implementations that keep the behaviour of the base sequential algorithm. Moreover, comparing the different populations and number of generations, there is not any statistical significance among the hypervolumes obtained by procedures with populations of more than 120 individuals and 50 generations. Thus, the differences of mean hypervolumes in Fig. 6 are not statistically significant.

This way, we will analyse the experimental results obtained with 50 generations and an initial population of individuals having 10 features set to 1 at most. As it has been said, the parallel codes implement the optimizations described in Sect. 3 with respect to a baseline parallel code evaluated in [6]. Table 1 compares the memory requirements of both codes and shows the relevant decrease in the memory requirements achieved by the previously described optimizations.

Table 1 Memory (in bytes) used by (Ref) code [6] and our proposed code (Opt) in [7]. N, W, N_F and N_P are the number of individuals, centroids, features and patterns respectively

Memory type		Global		Constant	Local			
Description		Population	Databases	Centroids		Individual	Tables	Distances
Array		S_{Pop}	S_{DB}	S_W	S_{Ki}	S_i	S_{MT}	S_D
Size	Ref	$N \times N_F$	$4 \times N_P \times N_F$	$4 \times W \times N_F$	$4 \times W \times N_F$	N_F	$3 \times W \times N_P$	$4 \times W \times N_P$
	Opt		$8 \times N_P \times N_F$	$4 \times W$			N_P	$4 \times N_P$
Total size	Ref	$N \times N_F + 4 \times N_P \times N_F$		$4 \times W \times N_F$	$4 \times W \times N_F + 7 \times W \times N_P + N_F$			
	Opt	$N \times N_F + 8 \times N_P \times N_F$		$4 \times W$	$4 \times W \times N_F + 5 \times N_P + N_F$			

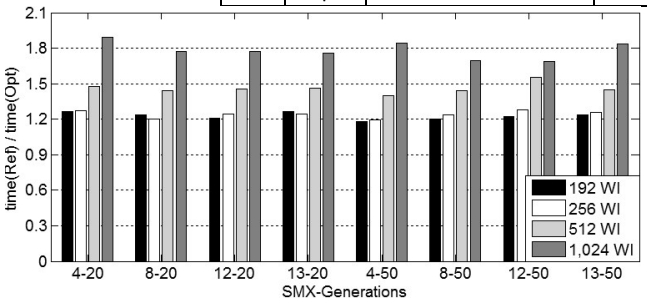

Fig. 7 Mean speedups achieved by the optimized GPU kernel with respect to the GPU kernel in [6]. Population size of 960 individuals, 48 features initially set to 1, dataset b480a and (20, 50) generations

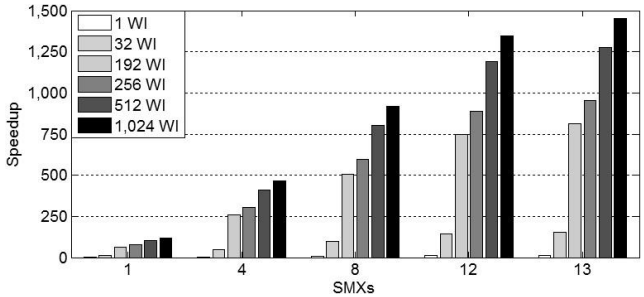

Fig. 8 Mean speedups with respect to only one work-item. Population size of 240 individuals, 10 features initially set to 1, dataset b3600a and
50 generations

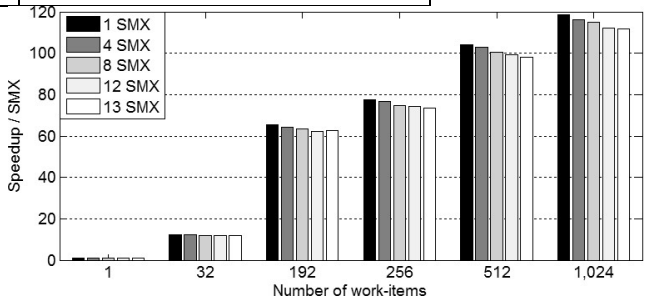

Fig. 9 Speedups per number of SMXs. Population size of 240 individuals, 10 features initially set to 1, dataset b3600a and 50 generations

The GPU kernels which we have implemented here take advantage of the coalescence technique and show better distributed memory access patterns. This way, the running times measured for the GPU kernels are lower than the ones obtained for the codes of [6] in all experiments accomplished under the same conditions of SMXs and work-items. As an example of the obtained results, Fig. 7 shows the speedups obtained by the GPU kernels here considered (called optimized kernels) with respect to the GPU kernels in [6] (known as reference kernels) for different numbers of SMX multiprocessors and work-items, populations of 960 individuals, and the dataset b480a (it has not been possible to run the reference kernels with the size of the b3600a dataset due to its memory requirements as the local memory consumption is very high). As can be seen, in all cases, speedups higher than one are obtained. Moreover, from 256 work-items, the amount of time-cutting provided by the optimized GPU kernel grows as more work-items are used, due to the effect of the applied coalescence technique. The differences in the speedups obtained for 512 and 1,024 work-items are statistically significant after applying the Kolmogorov-Smirnov and the Kruskal-Wallis tests. We have also applied the Kolmogorov-Smirnov and the Kruskal-Wallis tests to analyse the statistical significance of the differences in the running times of reference [6] and the optimized ones. In all cases, the differences are statistically significant, with p-values lower than 0.009.

On the other hand, Fig. 8 provides the speedups obtained by the GPU kernels for different SMX multiprocessors and work-items with respect to the running time of a work-item in only one SMX multiprocessor. It can be seen that the speedups grow with the number of SMXs and work-items. To get some insight on the effect of the number of these SMXs and work-items, Fig. 9 provides the speedups of Fig. 8 divided by the number of SMXs. As can be seen, these speedups are quite similar for a given number of work-items (there is only a slight decrease as the number of SMXs grows). As the best speedup values for a given number of SMXs are obtained for the maximum number of work-items, in the following experiments we will use this number (1,024 workitems) in the evaluations of the individuals done by the GPU kernels.

4.2 Dynamic scheduler evaluation

4.2.1 Overhead analysis

Table 2 shows the time values obtained from traces of execution of the parallel procedure in the different devices, while using their maximum number of available CPU cores, or SMX processors in the Quadro and Tesla GPUs. For each device, we have considered two different sizes of individuals assigned each time by the implemented scheduler (Alg. 1): the whole population and the maximum number of CPU cores or SMX processors in the GPU. The column $Ov(\%)$ gives the percentage of overhead time measured in each trace evaluated as:

$$Ov(\%) = \frac{T_C + T_{CL}}{T_T} \times 100 \quad (5)$$

where T_C is the time to transfer the individuals to be evaluated between the master CPU core and the corresponding device, T_{CL} is the time required to setup and kill the corresponding OpenCL kernel and T_T is the execution time of the parallel procedure. Table. 2 demonstrates that the percentage of overhead is higher whenever the individuals are evaluated after several assignations of individuals instead of sending all the population once. Moreover, despite the lower percentage overhead per assignment of individuals in the Quadro GPU with respect to the Tesla GPU (0.56% with respect to 1.26%), the percentage of overhead is higher in the Quadro GPU when the master assigns as many

Table 2 Time values from execution traces in different platform configurations and 120 individuals in the population with database b3600a. The *Size* column shows the number of individuals assigned each time by the master core

Device	Size K	means Time (ms)	Copy Time (T_C) (ms)	Bandwidth (GB/s)	OpenCL Time (T_{CL}) (ms)	T. Time (T_T) (ms)	Ov(%)
Quadro	2	587.09	0.33	2.44	47.91	635.33	7.6
	120	575.89	0.21	6.24	3.05	579.15	0.56
Tesla	13	263.7	0.23	4.85	11.29	275.22	4.18
	120	257.37	0.21	6.26	3.07	260.65	1.26
Xeon	24	230.75	-	-	7.95	238.7	3.33
	120	229.05			1.64	230.68	0.71
Seq.	120	2,546.6	-	-	-	2,546.6	0

individuals as SMXs in the GPU, because the *K*-means time is much higher in Quadro than in Tesla (the Quadro GPU only has 2 SMXs in comparison to the Tesla one, which has 13 SMXs). However, to compute the 120 individuals, the Quadro GPU consumes 7.6% of overhead in comparison to 4.18% of the Tesla. The reason is that the Quadro GPU needs 60 kernel executions to process the 120 individuals, and Tesla only 5.

Indeed, the overhead generated by the multiple calls to the CPU and GPU kernels is quite high. It would be interesting to explore other implementations that reduce such overhead. A benchmark could be run at the beginning of the program to study the relative performance between the multiple devices running the program and apply that proportion to the distribution of individuals. Despite being a good optimization approach, the individuals distribution by using this strategy is static. Our present approach provides a dynamic workload distribution. Although in our applications, all parameters could be known a priori and we are using heterogeneous platform with only two different processors (CPU and GPU), a static strategy based on a cost analysis of tasks in different CPU-GPU cores could be difficult to apply to more heterogeneous platforms with several types of cores. Moreover, other multi-objective feature selection applications may have dynamic parameters at runtime where static partitioning is not adequate or cannot be performed. Indeed, we could change the stop criterion of our *K*-means algorithm and to use the achievement of an error level. Our intention is to give readers a general idea of how to perform a dynamic distribution of "general data" using OpenMP to be processed by heterogeneous devices such as CPUs and GPUs. In problems with more irregular sizes of tasks, our parallel procedure would be even better.

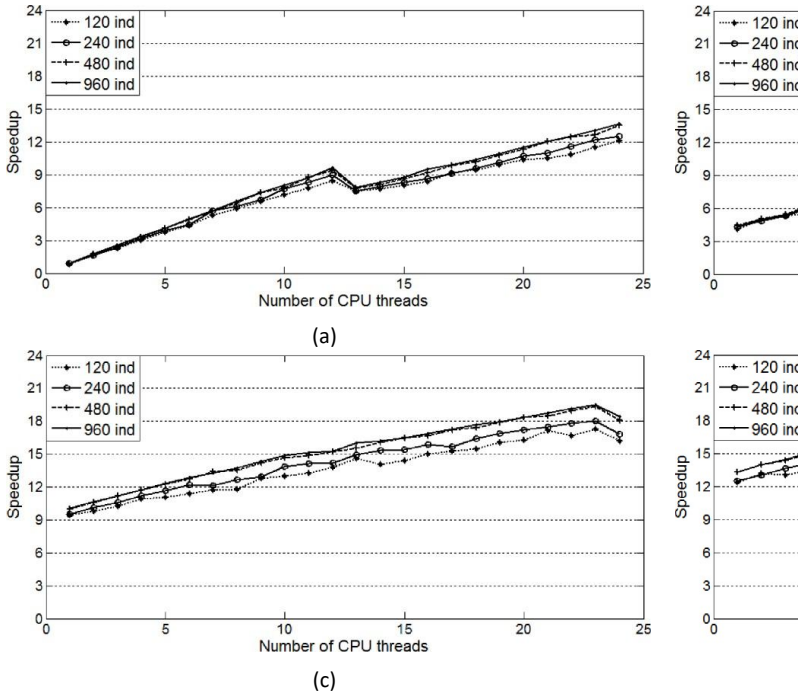

Fig. 10 Mean speedups for different number of CPU threads, population sizes, and plat
Quadro GPU; (c) CPU cores + Tesla GPU; (d) CPU cores + Quadro + Tesla GPUs. 10 featu

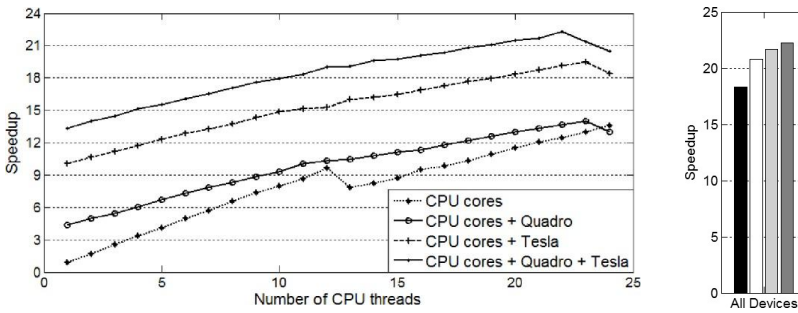

Fig. 11 Mean speedups for different number of CPU threads and platform configurations. Population size of 960 individuals, 10 features initially set to 1, dataset b3600a and 50 generations

4.2.2 Performance and scalability

Figures 10, 11 and 12 provide results about the speedups obtained by different platform configurations. We have defined seven configurations according to the devices that can be chosen by the master to assign evaluations of individuals in the population. This way, we have configurations that include only CPU cores ("CPU cores"), SMXs in a GPU or both GPUs ("Tesla", "Quadro", and "Tesla + Quadro"), heterogeneous configurations including CPU cores and SMXs of a given GPU ("CPU + Tesla", "CPU + Quadro"), and CPU cores and SMXs from Tesla GPU and Quadro GPU ("All Devices"). In Figs. 10, 11 and 12, the evolutionary multi-objective algorithm runs 50 generations and the individuals in the initial population are feature selections with 10 features at most. Populations of 120, 240, 480, and 960 individuals have been used in the experiments. Although in the benchmarks used in this paper a population of 120 individuals provides hypervolumes results without significant differences with respect to those provided by bigger populations, it is useful to consider them to analyse the scalability of our proposal for other datasets that could require these

larger populations. In any case, a population including 120 individuals shows a similar behaviour to bigger populations with respect to changes in the CPU-GPU configurations and

Fig. 12 Mean speedups for different configurations and population sizes. 10 features initially set to 1, dataset b3600a and 50 generations

changes in the number of cores. As it has been said, each time the master assigns individuals to be evaluated in the GPUs, and the number of individuals assigned is equal to the number of SMXs in the corresponding GPU: 2 for Quadro and 13 for Tesla.

Figure 10 shows the speedup changes when increasing the number of CPU threads in four configurations that use the CPU to evaluate individuals. As can be seen, the speedups grow as more threads are used except in the case of 13 threads, because there are 12 CPU cores in the node and it has to be taken into account that the scheduler also uses at least one thread to assign individuals. The same effect can be seen as we approach to 24 threads, which is the number of possible CPU threads in the node.

The improvement in the speedups when the GPUs are involved in the evaluation of individuals is shown in Fig. 11 considering a population of 960 individuals to illustrate the behaviour. It clearly shows that the "All Devices" configuration including CPU cores and SMXs from both Tesla and Quadro GPUs ("CPU cores + Quadro + Tesla"), provides the best speedups independently of the number of CPU threads involved. The Quadro GPU only provides a small improvement in the speedups with respect to the use of CPU cores alone. It can be explained from the results in Tab. 2, that show higher percentages of overhead time for this GPU. This way, when 24 CPU threads are used to evaluate individuals, it has to be taken into account that, along the higher overhead involved in the use of the Quadro GPU, one CPU thread is required to assign individuals to the GPU kernels. These circumstances counter the effect of using the SMXs to accelerate the individuals evaluation. The speedup reduction as we approach to 24 CPU threads is also apparent in the other configurations involving the Tesla GPU, but this reduction is not so high, and the speedup improvement still remains.

Following with the speedup curves shown in Fig 11, as it has been previously said, in the "CPU cores" configuration the speedup shows a reduction for 13 CPU threads because, although there are only 12 cores in the node, the overhead associated to the master thread only has a relatively low effect. Thus, the fact of having threads that share cores is not apparent for 12 threads (12 threads used as workers and one more as master thread) but for 13 (13 threads used as workers and one more as master thread). This can be demonstrated by taking into account that there is no speedup in case of 24 CPU threads (the maximum number of simultaneous threads in the node is 24). In the configurations including GPUs, there are also slight reductions in the speedups for numbers of CPU threads near to 13 threads. In these curves ("CPU + Tesla", "CPU + Quadro" and "CPU cores + Quadro + Tesla") the reduction in the speedups is smaller and, in general, the speedup curves are smoother due to the speedups provided by the GPUs, that also take advantage of the data parallelism. Moreover, although there are only 12 cores, there are two simultaneous threads per core and there are still threads available to execute the master. The effect of higher overheads in the configurations including GPUs is apparent if it is taken into account that the highest speedups is shown for 23 CPU threads in the "CPU + Tesla" and "CPU + Quadro" configurations, and for 22 threads in the "CPU cores + Quadro + Tesla" configuration. As the overheads to assign workloads and to communicate with the GPUs is higher, the cost of the master thread is more important and the reduction in the speedups is clear. In the case of the "CPU cores + Quadro + Tesla" configuration, the master has to assign workload and communicate with two GPUs, and the maximum speedup is shown for 22 threads, i.e. two threads less than the maximum number of parallel CPU threads in the node.

Figure 12 summarizes the highest speedups attained by each platform configuration for different population sizes. It

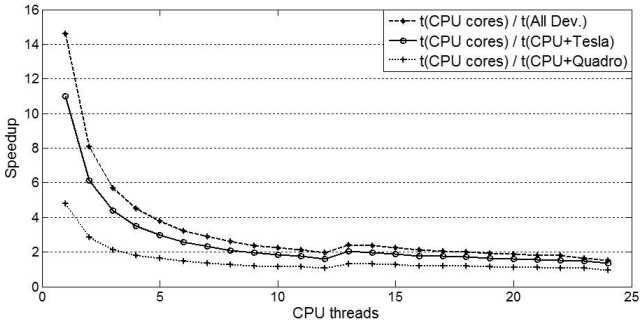

Fig. 13 Mean speedups of platform configurations including GPUs with respect to the use of CPU. Population size of 960 individuals, 10 features initially set to 1, dataset b3600a and 50 generations

clearly shows the improvement achieved as more OpenCL devices can be used by the master to distribute the evaluation of the individuals in the population. In addition, it has been checked that the differences in the highest speedups attained for a given population size are statistically significant among all configurations (i.e. “All Devices”, “CPU + Tesla”, “CPU + Quadro”, “CPU cores”, “Tesla + Quadro”, “Tesla” and “Quadro”). With respect to changes in the speedups as the size of the population is increased, statistically significant differences for populations with 480 and 960 individuals in the “CPU + Tesla” configuration, and with 240, 480, and 960 individuals in the “All Devices” one have been observed.

Figure 13 shows, for a population of 960 individuals, the mean speedups obtained by the platform configurations including GPUs (“All Devices”, “CPU + Tesla”, and “CPU + Quadro”) with respect to the use of CPU cores alone. The behaviours shown in Fig. 13 are similar for all the population sets considered and correspond to improvements of at least about 50% for the configuration including Tesla and Quadro GPUs and about 35% for the platform including the Tesla GPU. In the case of the platform including only the Quadro GPU, as it has been indicated before, the speedups could be less than one in the case of using 24 CPU threads, although the values obtained are larger than 0.9.

Nevertheless, with all considered populations, to get the speedups attained by the 2 SMX processors of the Quadro GPU, the 13 SMX processors of the Tesla GPU and the 15 SMX processors of the Tesla + Quadro GPUs, respectively 5, 19, and 23 CPU threads are required in the configuration using only CPU cores. This does not mean that CPU cores are worse but the usefulness of data parallelism implemented in our GPU kernel. It has to be taken into account that vectorization could be also implemented in the CPU kernel to take advantage of the SIMD available in the CPU cores. In this case, even better speedups may be obtained, as we will check in future versions of our procedure.

Despite the high floating-point peak performance of the GPU, about 3,524 GFLOPS [25], which is also much higher than the 120 GFLOPS of the E5-2620 processor [26], it is quite difficult to reach speeds near the GPU peak performance. On the one side, the GPU performance is greatly impaired by the conditional branches (if-then-else statements) of the code, causing the work-items of the same warp in the *if* branch and the work-items in the *else* branch to wait for each other. Moreover, synchronizations between workitems and the copy of the individual to be evaluated also decreases performance. These drawbacks are not present in CPU as there is only one work-item per individual and no extra copy of the individual is needed because RAM memory is accessed directly. Moreover, the problem with synchronization of work-items in conditional instructions does not appear either in the CPU cores. Nevertheless, it has to be taken into account that, as it has been said, we have not used vectorization in the CPU cores, and our GPU kernel allows us to take advantage of the data parallelism available in the GPU. Despite less individuals can be processed in parallel in the GPU than in the available CPU cores, the achieved speedups are similar in both configurations (only CPU cores and only the GPU).

5 Previous related works

The procedure here considered includes an evolutionary multi-objective optimization and a clustering algorithm applied to a set of high-dimensional patterns usually requiring high-volume storage. The use of heterogeneous architectures including data parallel architectures such as GPUs has been proposed in previous papers. In [27,28] some approaches are proposed to cope with some difficulties appearing in the parallelization of algorithms frequent in data mining applications. Paper [27] proposes parallel approaches to problems such as the presence of irregular patterns, the selection of the K minimum or maximum elements in a given set or the high dimension reduction problem, illustrating the effect of the proposed approaches in widely used data mining algorithms such as *Apriori*, *K-Nearest Neighbor* and *K-means*. Nevertheless, the parallelization on a heterogeneous platform of a whole data mining application with the characteristics of our target application is less frequent in the literature. Paper [28] analyses the effect of factors such as the communication patterns and the data partition on the performance of data mining applications. Our approach takes advantage of heterogeneous platforms and uses both CPU cores and GPUs at a given moment to speed up the application.

In paper [29], some strategies fuelling the trend towards heterogeneous processors have been considered. That paper also analyses the benefits of mechanisms to migrate tasks to available cores,

memory optimizations, and some optimizations to reduce the distance between producer and consumer tasks through fine-grained communication. Precisely we have shown, for example through the analysis of Fig. 11, the effects of overheads in platform configurations that need to communicate with a GPU through a PCIe bus. Moreover, Fig. 7 clearly shows the effect of memory optimizations, and our parallel codes also allow speedup improvements by dynamically distributing the workload among the available CPU cores and GPUs (e.g. Fig. 10).

Different approaches for GPU-based implementations of evolutionary algorithms are analysed in [3], and the issue of implementations of parallel metaheuristics on multicore platforms has been surveyed in [4]. However, works analysing the effect in the parallel performances of heavy fitness functions requiring high-volume datasets are less frequent.

In general, for an evolutionary algorithm, as the evaluation of the fitness can be independently done for each individual in the population, this step is usually implemented in parallel. Nevertheless, other steps in the evolutionary algorithms, such as the individuals selection or the evolutionary operators require interaction among individuals, thus involving some kind of synchronization among the computing elements. This way, two main researching lines can be distinguished among the proposals on a GPU implementation of evolutionary algorithms, i.e. a parallel implementation that shows the same behaviour than the sequential one, and the implementation of an evolutionary parallel algorithm tuned to the characteristics of the GPU architecture. However, its characteristics could be different from those of the corresponding sequential algorithm. In this last alternative, an analysis of the suitability of the attained solutions should be done.

Pospichal et al. [5] describes a CUDA implementation of a parallel genetic algorithm based on an island model. The paper provides new implementations of genetic operators especially devised to be efficiently executed in the CUDA architecture, and uses the main memory of the GPU to allow asynchronous exchanges of individuals between subpopulations interconnected according to a unidirectional ring topology. This paper is an example of the approaches that modify the evolutionary algorithm to reach a more suitable version for the available GPU architecture and resources.

An alternative GPU implementation of the nondominance rank used in NSGA-II, the Archived-based Stochastic Ranking Evolutionary Algorithm (ASREA), is provided in [30]. It is based on an archive of distinct nondominated solutions that allow the evaluation of the rank of a given solution as one plus the number of solutions in the archive that dominate this solution. Using the ZDT3, ZDT4 and ZDT5 benchmarks with two objectives and a population of 10,000 individuals, the results show speedups of about 5,000 for GPU implementations, with respect to a CPU implementation of NSGA-II, although these speedups are low with respect to the sequential implementation of ASREA.

Paper [31] provides a parallel GPU implementation of a multi-objective evolutionary algorithm for a data mining application on marketing that predicts potential prospects from records of customers. This approach executes all the steps of an NSGA-II algorithm in the GPU except for the nondominated selection, as it requires to sort the non-dominated individuals according to their objectives, the number of elements to be sorted varies, the efficiency of sorting a small number of values in a GPU could be low, and there should be many synchronizations and accesses to variables by different threads. This way, besides the parallel implementation of the independent fitness evaluation for the individuals in the population, a fast procedure is proposed for the non-dominated selection. With respect to the non-dominated sort, a procedure is implemented to determine the number of individuals that dominate a given one and the set of individuals dominated by each individual, in parallel, without interaction among kernels. The results provided by [31], from a dataset with records of 361 components (only 17 selected variables are finally used) corresponding to more than 100,000 customers, show an overall GPU speedup of about 23 with respect to the CPU implementation. In this application, the fitness evaluation times are higher than the rest of steps of the application (representing more than 98% of the execution time).

The parallelization of the *K*-means algorithm on a GPU has been considered in many papers [32,33]. In [34], a speedup factor of up to 68 is reported for a GeForce GTX 8800 Ultra, with respect to a base implementation (CPU at 3 GHz), and a dataset consisting of one million unsigned long integers (4 bytes per element) defining 4,000 clusters. The use of large datasets for *K*-means clustering is considered in

[35], where clustering is applied to datasets including one billion patterns and 1,000 centroids, although they are only bi-dimensional, and achieves a speedup factor of more than 11 with respect to an optimized CPU version executed on 8 cores. *K*-means for clustering higher dimensional data on GPUs is considered in [36]. This paper provides experimental results by using datasets of up to 500,000 instances with 2, 20, and 200 dimensions and shows speedups of 1.5 to 14

with respect to a fully optimized C++ version that makes use of the SIMD instruction set of the CPU (speedups of 4 to 43 with respect to a not fully optimized sequential version). It also points out that performances are better as higher dimensions and clusters are considered, and that many real-data applications (such as the one here considered) process sparse patterns that imply difficulties to apply memory coalescing transformations, thus requiring specific procedures to be optimally implemented in GPUs. Paper [27] proposes a parallel high dimension reduction scheme which is used to get an efficient parallel GPU implementation for *K*-means.

With respect to other OpenCL implementations, the paper [37] shows a genetic algorithm for feature selection in a biometric recognition application. Although this paper does not implement a multi-objective evolutionary algorithm, its approach follows a quite similar strategy to the one considered in the present paper. Thus, the evaluation of the individuals is implemented in parallel besides a new level of parallelism related to the characteristics of the application. While paper [37] takes advantage of the parallelism related to the processing of the different classes involved in the application, our procedure parallelizes the clustering algorithm and the computation of the cost functions involved in the evaluation of the individuals. Moreover, it can also benefit from both GPU and CPU cores available in the platform.

6 Conclusions

Although many works in the literature have shown important speedups achieved by different parallel evolutionary algorithms implemented on GPUs, fewer details have been reported about the benefits of such architectures in data mining applications with irregularities in the data accesses, along with high dimensional patterns and/or high volume data. This paper proposed and evaluated a master-worker parallel multiobjective procedure to a high-dimensional feature selection problem related with EEG classification on BCI tasks. The OpenCL parallel procedure here proposed allows a dynamic workload distribution among the cores of heterogeneous platforms including multicore CPU and GPU architectures.

The procedure includes a multi-objective optimization evolutionary algorithm where the fitness evaluation for a given individual implies the computation of two CVIs through a *K*-means algorithm applied to the patterns of the dataset. This way, we have implemented a master-worker parallel algorithm in which a master CPU core distributes the individuals evaluation among the available CPU cores and GPU SMXs. The master core launches two different OpenCL kernels. While the CPU kernel implements the individuals evaluation through *K*-means, the GPU kernel allows not only the parallel evaluation of individuals but also taking advantage of the data parallelism available in *K*-means. This way, the individuals are distributed among the SMXs of the GPU and the *K*-means algorithm is parallelized among the workitems in the SMXs. The use of the GPU memory hierarchy has been optimized through some techniques among which the coalescing of memory accesses and the minimization of memory bank conflicts have been the most efficient ones.

From the accomplished experiments, we have demonstrated the relevance of GPU memory optimizations to take advantage of coalescing and decreasing memory contention, which has allowed us to increase the achieved GPU speedups from 20% up to 80% when 1024 work-items are used, with respect to a first GPU kernel previously provided in [6]. It is clear that platform configurations comprising both GPUs and CPU cores improve the speedups provided by the parallel codes executed either on CPU cores or GPU only. Moreover, in all platform configurations, the results show good behaviour for scalability as the CPU threads increase up to the maximum number of threads in the CPU.

Despite the relatively good results shown in this paper, we are exploring other strategies to increase the efficiency of heterogeneous parallel architecture, which are related to the improvement of core, cache efficiencies and the power-aware scheduling. Indeed, energy consumption offers a good approach to evaluate efficiencies in heterogeneous architectures. Moreover,

the implementation of evolutionary subpopulations through island approaches could offer new insights into the possibilities of heterogeneous parallel architectures.

References

1. Rupp, R., Kleih, S., Leeb, R., Millan, J., Kübler, A., Müller-Putz, G.: Brain-computer interfaces and assistive technology. In: Grübler, G., Hildt, E. (eds.) Brain-Computer-Interfaces in their Ethical, Social and Cultural Contexts, pp. 7–38. The International Library of Ethics, Law and Technology, Springer (2014)
2. Collet, P.: Why gpgpus for evolutionary computation? In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs, pp. 3–14. Natural Computing Series, Springer (2013)
3. Luong, T., Melab, N., Talbi, E.G.: Gpu-based island model for evolutionary algorithms. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation. pp. 1089–1096. GECCO'2010, ACM, Portland, OR, USA (July 2010)
4. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: Recent advances and new trends. International Transactions in Operational Research 20(1), 1–48 (2013)
5. Pospichal, P., Jaros, J., Schwarz, J.: Parallel genetic algorithm on the cuda architecture. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A., Goh, C.K., Merelo, J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G. (eds.) Proceedings of the 13th European Conference on the Applications of Evolutionary Computation. pp. 442–451. EvoApplications'2010, Springer, Istambul, Turkey (April 2010)
6. Escobar, J., Ortega, J., González, J., Damas, M.: Assessing parallel heterogeneous computer architectures for multiobjective feature selection on eeg classification. In: Ortuño, F., Rojas, I. (eds.) Proceedings of the 4th International Conference on Bioinformatics and Biomedical Engineering. pp. 277–289. IWBBIO'2016, Springer, Granada, Spain (April 2016)
7. Escobar, J., Ortega, J., González, J., Damas, M.: Improving memory accesses for heterogeneous parallel multi-objective feature selection on eeg classification. In: Proceedings of the 4th International Workshop on Parallelism in Bioinformatics. PBIO'2016, Springer, Grenoble, France (August 2016)
8. Bellman, R.: Adaptive Control Processes: A Guided Tour. Princeton University Press (1961)
9. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. The Journal of Machine Learning Research 3, 1157–1182 (March 2003)
10. Mukhopadhyay, A., Maulik, U., Bandyopadhyay, S., Coello Coello, C.: A survey of multiobjective evolutionary algorithms for data mining: Part i. IEEE Transactions on Evolutionary Computation 18(1), 4–19 (2014)
11. Mukhopadhyay, A., Maulik, U., Bandyopadhyay, S., Coello Coello, C.: A survey of multiobjective evolutionary algorithms for data mining: Part ii. IEEE Transactions on Evolutionary Computation 18(1), 20–35 (2014)
12. Emmanouilidis, C., Hunter, A., MacIntyre, J.: A multiobjective evolutionary setting for feature selection and a commonality-based crossover operator. In: Proceedings of the 2000 Congress on Evolutionary Computation. CEC'2000, vol. 1, pp. 309–316. IEEE, La Jolla, CA, USA (July 2000)
13. Handl, J., Knowles, J.: Feature subset selection in unsupervised learning via multiobjective optimization. International Journal of Computational Intelligence Research 2(3), 217–238 (2006)
14. Morita, M., Sabourin, R., Bortolozzi, F., Suen, C.: Unsupervised feature selection using multi-objective genetic algorithms for handwritten word recognition. In: Proceedings of the Seventh International Conference on Document Analysis and Recognition. pp. 666–670. ICDAR'2013, IEEE (August 2003)
15. Arbelaitz, O., Gurrutxaga, I., Muguerza, J., Pérez, J., Perona, I.: An extensive comparative study of cluster validity indices. Pattern Recognition 46(1), 243–256 (2013)
16. Kimovski, D., Ortega, J., Ortiz, A., Baños, R.: Leveraging cooperation for parallel multi-objective feature selection in highdimensional eeg data. Concurrency and Computation: Practice and Experience 27(18), 5476–5499 (2015)
17. Khronos Group: Khronos opencl registry. https://www.khronos.org/registry/cl/ (Accessed: 2015-11-30)
18. OpenMP Community: Openmp specifications. http://www.openmp.org/specifications/ (Accessed: 2016-11-21)
19. Gunarathne, T., Salpitikorala, B., Chauhan, A., Fox, G.: Optimizing opencl kernels for iterative statistical algorithms on gpus. In: Proceedings of the Second International Workshop on GPUs and Scientific Applications. pp. 33–44. GPUScA'2011, Galveston Island, Texas, USA (October 2011)
20. Dhanasekaran, B., Rubin, N.: A new method for gpu based irregular reductions and its application to k-means clustering. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units. pp. 729–737. GPGPU-4, ACM, Newport Beach, California, USA (March 2011)
21. Asensio-Cubero, J., Gan, J., Palaniappan, R.: Multiresolution analysis over simple graphs for brain computer interfaces. Journal of Neural Engineering 10(4) (2013)
22. Daubechies, I.: Ten Lectures on Wavelets. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1992)
23. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist nondominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In: Proceedings of the 6th International Conference on Parallel Problem Solving from

Nature. pp. 849–858. PPSN VI, Springer, Paris, France (September 2000)
24. Fonseca, C., López-Ibáñez, M., Paquete, L., Guerreiro, A.: Computation of the hypervolume indicator. http://lopez-ibanez.
eu/hypervolume (Accessed: 2015-11-30)
25. Nvidia Corporation: Nvidia tesla k20c datasheet. http://www.nvidia.com/content/tesla/pdf/ nvidia-tesla-kepler-family-datasheet.pdf (Accessed: 2017-05-17)
26. Intel Corporation: Intel xeon processor e5-2600 series specifications.
http://download.intel.com/support/processors/
xeon/sb/xeon_E5-2600.pdf (Accessed: 2017-05-17)
27. Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W., Shi, Y.: Parallel data mining techniques on graphics processing unit with compute unified device architecture (cuda). The Journal of Supercomputing
64(3), 942–967 (June 2013)
28. Gainaru, A., Slusanschi, E., Trausan-Matu, S.: Mapping data mining algorithms on a gpu architecture: A study. In: Kryszkiewicz, M., Rybinski, H., Skowron, A., Ras, Z.W. (eds.) Proceedings of the´ 19th International Symposium. Foundations of Intelligent Systems. pp. 102–112. ISMIS'2011, Springer Berlin Heidelberg, Warsaw, Poland (June 2011)
29. Hestness, J., Keckler, S., Wood, D.: Gpu computing pipeline inefficiencies and optimization opportunities in heterogeneous cpu-gpu processors. In: Proceedings of the 2015 IEEE International Symposium on Workload Characterization. pp. 87–97. IISWC'15, IEEE Computer Society, Atlanta, GA, USA (October 2015)
30. Sharma, D., Collet, P.: Implementation techniques for massively parallel multi-objective optimization. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs, pp. 267–286. Natural Computing Series, Springer (2013)
31. Wong, M., Cui, G.: Data mining using parallel multi-objective evolutionary algorithms on graphics processing units. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs, pp. 287–307. Natural Computing Series, Springer
(2013)
32. Baramkar, P., Kulkarni, D.: Review for k-means on graphics processing units (gpu). International Journal of Engineering Research & Technology 3(6), 1911–1914 (2014)
33. Kijsipongse, E., U-ruekolan, S.: Dynamic load balancing on gpu clusters for large-scale k-means clustering. In: Proceedings of the 9th International Joint Conference on Computer Science and Software Engineering. pp. 346–350. JCSSE'2012, Bangkok, Thailand
(May 2012)
34. Farivar, F., Rebolledo, D., Chan, E., Campbell, R.: A parallel implementation of k-means clustering on gpus. In: Proceedings of the International Conference on Parallel and Distributed Processing
Techniques and Applications. pp. 340–345. PDPTA'08, Las Vegas, Nevada, USA (July 2008)
35. Wu, R., Zhang, B., Hsu, M.: Clustering billions of data points using gpus. In: Hast, A., Buchty, R., Tao, J., Weidendorfer, J. (eds.) Proceedings of the Combined Workshops on UnConventional High Performance Computing workshop plus Memory Access Workshop. pp. 1–6. UCHPC-MAW'09, ACM, Ischia, Italy (May 2009)
36. Zechner, M., Granitzer, M.: Accelerating k-means on the graphics processor via cuda. In: Proceedings of the First International Conference on Intensive Applications and Services. pp. 7–15. INTENSIVE'09, IEEE, Valencia, Spain (April 2009)
37. Fazendeiro, P., Padole, C., Sequeira, P., Prata, P.: Opencl implementations of a genetic algorithm for feature selection in periocular biometric recognition. In: Panigrahi, B., Das, S., Suganthan, P., Nanda, P. (eds.) Third International Conference on Swarm, Evolutionary and Memetic Computing. pp. 729–737. SEMCCO'2012,