CUDA-JMI: Acceleration of Feature Selection on Heterogeneous Systems

Jorge González-Domínguez[a,*], Roberto R. Expósito[a], Verónica Bolón-Canedo[b]

[a] *Universidade da Coruña, CITIC, Grupo de Arquitectura de Computadores, Campus de A Coruña, 15071 A Coruña, Spain*

[b] *Universidade da Coruña, CITIC, Department of Computer Science, Campus de A Coruña, 15071 A Coruña, Spain*

Abstract

Feature selection is a crucial step nowadays in machine learning and data analytics to remove irrelevant and redundant characteristics and thus to provide fast and reliable analyses. Many research works have focused on developing new methods that increase the global relevance of the subset of selected features while reducing the redundancy of information. However, those methods that select features with high relevance and low redundancy are extremely time-consuming when processing large datasets. In this work we present CUDA-JMI, a tool based on Joint Mutual Information that accelerates feature selection by exploiting the computational capabilities of modern heterogeneous systems that contain several CPU cores and GPU devices. The experimental evaluation has been carried out in three systems with different type and amount of CPUs and GPUs using five publicly available datasets from different fields. These results show that CUDA-JMI is significantly faster than its original sequential counterpart for all systems and input datasets. For instance, the runtime of CUDA-JMI is up to 52 times faster than an existing sequential JMI-based implementation in a machine with 24 CPU cores and two NVIDIA M60 boards (four GPUs). CUDA-JMI is publicly available to download from https://sourceforge.net/projects/cuda-jmi.

Keywords: Feature Selection, Machine Learning, CUDA, GPU, Multithreading

1. Introduction

Many scientific and research applications work over datasets that contain the values of different features or characteristics concerning to several individuals or samples. Some examples are bioinformatics or text analytics [1]. In the recent days we have seen a rapid and extreme increase in the amount of stored data, especially the number of features per sample. An example would be genetic datasets, that include the expression values (the degree to

[*] *Corresponding author*

Email addresses: jgonzalezd@udc.es (Jorge González-Domínguez), rreye@udc.es (Roberto R. Expósito), veronica.bolon@udc.es (Verónica Bolón-Canedo)

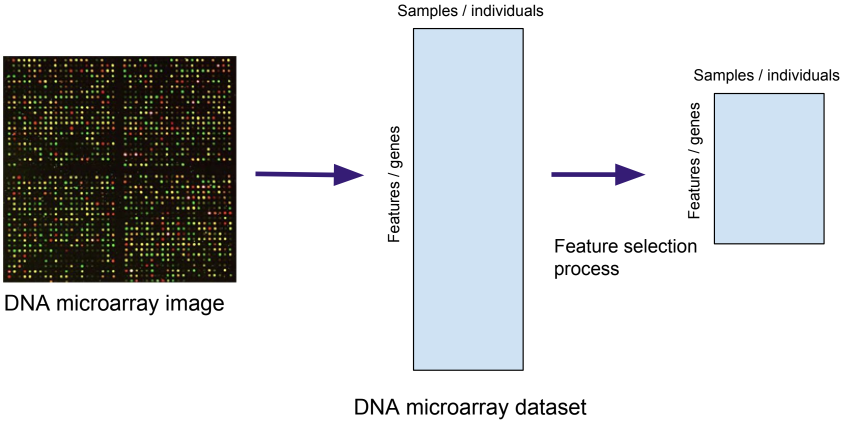

Figure 1: Example of feature selection in DNA microarray data

which an individual exhibits a trait or a genetic mutation) of thousand of genes for different individuals (they are usually persons or any type of living beings). In this case, a DNA microarray image containing expression values, is converted into a DNA microarray dataset in which rows represent the genes (or features) and columns represent the individuals (or samples), as depicted in Figure 1. Analyzing more features increases the average runtime of the applications but it does not always implies obtaining more relevant information. Common solutions to this problem consist in reducing the dimensionality of the dataset discarding those features that do not provide relevant information through Feature Selection (FS) or Feature Extraction (FE) methods. In Figure 1 we can see how by applying a feature selection process, the number of rows (features or genes) is reduced, so a more compact representation of the data is achieved.

FS [2, 3, 4] is a data mining approach used to detect the relevant features and discard the irrelevant ones. Its main advantage over FE is that the selected features are not transformed. A correct selection of the features can lead to an improvement of the related learner or classifier, either in terms of learning speed, generalization capacity or simplicity of the induced model. Furthermore, there are some other benefits associated with a smaller number of features such as a reduced measurement cost and hopefully a better understanding of the domain. There are several scenarios that can benefit from FS, such as the presence of irrelevant and redundant features, noise in the data or interaction between attributes. For instance, the study of a large number of features in microarray analyses does not usually lead to high-quality information as many of them are either irrelevant or redundant with respect to the class concept [5, 6].

FEAST [7], a broad suite of FS methods implemented in C and Matlab was developed in an attempt to facilitate the access to FS methods. It is highly-cited and widely-used by researchers and scientists. Among the different alternatives present in this toolbox, its method based on maximizing the Joint Mutual Information (JMI) [8] selects features with high relevance and low redundancy [7, 9]. However, its quadratical runtime with the number of features makes this method impractical for large datasets. Our work presents CUDA-JMI, a novel parallelization of the JMI-based method included in FEAST that accelerates FS by exploiting the hardware resources of modern heterogeneous systems that include CPU cores and NVIDIA GPU devices. It is implemented following a hybrid parallel approach that uses CUDA and the multithreading support of C++11 so that: 1) it offers multi-GPU support; and 2) not only the GPUs but also several CPU cores can collaborate for FS.

The rest of the paper is organized as follows. Section 2 summarizes the related work and the state of the art. Section 3 presents as background some concepts about FS and parallel computing that are necessary to understand the goal of the work and the implementation of the method. Our parallel implementation is described in Section 4. Section 5 provides the experimental evaluation in terms of runtime. Finally, conclusions and future work lines are presented in Section 6.

2. Related Work

The exploitation of High Performance Computing (HPC) resources to accelerate FS has been previously addressed. For instance, some of the FS algorithms included in the WEKA [10] and MAST [11] toolkits support multithreading in order to exploit the several CPU cores available in current processors. There also exist implementations designed for larger infrastructures such as distributed-memory systems with several nodes connected through a network. Some of them are based on the message-passing paradigm, as the wrapper FS tool DWFS [12], with a parallel genetic algorithm, or an MPI version of online FS [13]. Other alternatives use the MapReduce paradigm to accelerate ReliefF [14], evolutionary algorithms [15], or positive approximation [16].

Another popular HPC platform is the GPU, with a relatively low price and good energy efficiency that provides computing power that can be equivalent to a medium-sized supercomputer, which is orders of magnitude more costly and thus less accessible to many researchers or scientists. The use of GPUs has already proven itself to be enormously useful to deal with big datasets [17, 18, 19, 20]. In fact, GPUs have been extensively employed to accelerate data mining algorithms such as association rule mining [21, 22], classification decision trees [23, 24], or biclustering [25, 26].

Although several previous works have addressed the GPU parallelization of different FS algorithms [27, 28, 29, 30, 31], fast-mRMR [32] is the only publicly available GPU-based implementation of a FS algorithm, up to our knowledge. It is a CUDA version of a greedy optimization of the popular minimum-Redundancy-Maximum-Relevance (mRMR) method that maintains the original performance in selecting the adequate features but significantly improving the computational times requirements. From this tool we take the idea of distributing the features (and not the samples) among threads, as well as discarding the parallelization of the loop that selects the features due to dependencies among iterations (as will be explained later). Nevertheless, CUDA-JMI presents the following advantages over fast-mRMR:

- The FS is based on maximizing JMI, which has been demonstrated to obtain better trade-off between relevance and redundancy than those based on Mutual Information

(MI) as mRMR and fast-mRMR. In fact, Brown et al. [7] studied the behavior of several information theoretic based feature selection methods and concluded that JMI was the best option to use, since it has the best trade-off of accuracy and stability.

- Our tool includes two different CUDA kernels that perform the main computation and automatically selects the best one at execution time depending on the characteristics of the GPU and/or the input dataset.
- While fast-mRMR only uses one GPU for speeding up the computation, CUDA-JMI is able to work simultaneously on several CPU cores and multiple GPU devices, being especially suitable for modern heterogeneous systems.
- The performance improvement obtained by CUDA-JMI is much higher than that of fast-mRMR, as will be shown in the experimental evaluation section.

Additionally, GPU implementations of FS algorithms using a non-parametric evaluation criterion or the delta test have been presented in [33] and [34] , respectively, but their code is not available to the scientific community.

3. Background

This section introduces concepts that are necessary to understand our proposal, such as FS, the FEAST library, the JMI measure, and parallel programming with CUDA.

3.1. Feature Selection

FS is a data mining technique commonly used in machine learning to reduce the dimensionality of datasets by selecting only those features that provide interesting information. FS methods can be classified into individual or subset evaluations. On the one hand, individual evaluation is the most simple approach, where each feature has a weight or score assigned according only to its degrees of relevance. The main drawback of this approach is that it is incapable of removing redundant features because they are likely to have similar rankings. On the other hand, in subset evaluations each candidate subset is assessed by a certain evaluation measure and compared with the previous best ones with respect to this measure. Subset evaluation approach can handle feature redundancy with feature relevance. However, these methods are computationally expensive due to searching through all feature subsets.

Additionally, FS methods can also be classified into three classes according to the relationship between a FS algorithm and the inductive learning method used to infer a model:

- Filters. They are applied as a preprocessing step independent of the induction algorithm and define a score used to measure the potential usefulness of a feature when used in a classifier. They are the most commonly employed approaches thanks to its relatively low computational cost and good generalization ability.

- Wrappers. They use a learning algorithm as a black box and consist of using its prediction performance to assess the relative usefulness of subsets of variables.

- Embedded methods. They perform FS in the process of classification and are usually specific to given learning machines.

3.2. FEAST: A Feature Selection Toolbox for C and Matlab

FEAST is a toolbox for feature selection written in C and Matlab by researchers of the University of Manchester to complement their work in which they presented a unifying framework for information theoretic feature selection [7].

FEAST provides the implementation of several popular MI-based feature selection algorithms, as well as an implementation of ReliefF, for the sake of completeness. Thanks to this toolbox, it is easy to perform experiments and check the similarities between these algorithms. The algorithms included are all from the family of filters, and can be seen in Table 1. More details about the algorithms can be found in [7] and the toolbox is available online for downloading[1].

Table 1: Feature selection methods implemented in FEAST

Acronym	Full name
MIM	Mutual Information Maximization
mRMR	Minimum Redudancy Maximum Relevance
MIFS	Mutual Information Feature Selection
CMIM	Conditional Mutual Info Maximization
JMI	Joint Mutual Information
DISR	Double Input Symmetrical Relevance
CIFE	Conditional Infomax Feature Extraction
ICAP	Interaction Capping
CONDRED	Conditional Redundancy
CMI	Conditional Mutual Information
ReliefF	—
FCBF	Fast Correlation Based Filter
Betagamma	—

3.3. Joint Mutual Information for Feature Selection

As previously mentioned, filters are the most commonly employed FS approach as they allow to separate the process of FS and classification (FS is a previous step). Many of those filters use MI as score, assuming that the most relevant features according to their MI value (features with strongest correlation between them and the class) should imply a greater predictive ability when are used in a classifier. However, MI does not consider

[1] http://www.cs.man.ac.uk/~gbrown/fstoolbox/

any dependency among features and thus the results of these methods are not good in the common case of having interdependent features.

When dependency among features arises (for instance, among genes in microarrays) we should not only focus on maximizing the relevance of the selected features, but also on minimizing the redundancy of information provided by them. JMI was presented in [8] as a score able to measure the complementary information among features:

$$JMI(F_i) = \sum_{j \epsilon S} MI(F_i, F_j; Y) \tag{1}$$

where F_i is the feature considered for selection, S is the subset of features already selected, and Y is the class. F_i, F_j is a joint random variable defined by pairing F_i and F_j.

As mentioned in Section 3.2, the FEAST suite provides a method for FS that maximizes the JMI. This is a subset evaluation method whose pseudo-code is presented in Algorithm 1. The first feature is selected as the one with the highest MI with respect to the class (Lines 4 to 6). Then, for every feature to select (outer loop starting at Line 7), FEAST calculates the JMI score of every candidate (inner loop starting at Line 8). This score is the sum of the JMIs joining the candidate and all the features already included in S. The selected feature for each iteration is the one with the highest accumulated score (Line 14).

In FEAST and CUDA-JMI the JMI calculation is divided into two steps (Lines 12 and 13, respectively):

1. Calculate the vector J with the information of the joint variable between the current and the last selected features. J is a vector with the same length as the feature vectors (i.e. the number of individuals M). J stores states (i.e. different combination of joint values for F_i and F_j) so that, for every pair of individuals (x,y), the states $J[x]$ and $J[y]$ are equal if and only if $F_i[x] == F_j[x]$ and $F_i[y] == F_j[y]$. Algorithm 2 shows the pseudo-code for a function that calculates the joint vector of two features. It uses a state map that saves the value applied to each possible combination. If the combination has appeared before, the state is obtained from the map (Line 15). Otherwise, a new state is assigned (Line 11), it is saved in the map (Line 12) and the state count is incremented (Line 13).
2. Compute the MI of J and the class Y. MI is defined as:

$$MI(J; Y) = H(J) + H(Y) - H(J, Y) \tag{2}$$

where $H(J)$ and $H(Y)$ are the marginal entropies and $H(J, Y)$ is the joint entropy of J and Y. They are calculated as:

```
Input: Dataset with $M$ individuals, $N$ features, and one class $Y$; Number of features to
  select $NS$
Output: Subset $S$ with the $NS$ features selected
Initialize $S$ to empty
for every feature $F_i$ with $0 \leq i < N$ do
    Calculate $MI(F_i; Y)$
end
Insert in $S$ the $F_h$ with the highest $MI$
for every $k$ with $0 < k < NS$ do
    for every feature $F_i$ with $0 \leq i < N$ do
        if $F_i$ is not in $S$ then
            $Score = 0$
            for every feature $F_j$ already included in $S$ do
                Create $J$ as the joint of $F_i$ and $F_j$
                $Score = Score + MI(J; Y)$
            end
        end
    end
    Insert in $S$ the $F_h$ with the highest $Score$
end
```

Algorithm 1: Pseudo-code of the sequential FS method based on maximizing JMI provided by FEAST.

$$H(J) = -\sum_{j \epsilon J} p(j) \cdot log(p(j)) \tag{3}$$

$$H(Y) = -\sum_{y \epsilon Y} p(y) \cdot log(p(y)) \tag{4}$$

$$H(J, Y) = -\sum_{j \epsilon J, y \epsilon Y} p(j, y) \cdot log(p(j, y)) \tag{5}$$

with $p(j)$ representing the probability of the states in J taking the value j, $p(y)$ the probability of the class Y taking the value y, and $p(x, y)$ the joint probability of both events.

3.4. CUDA and GPU Technology

CUDA [35] is an architecture for parallel programming on NVIDIA GPUs that extends the general programming languages with a set of abstractions to express parallelism. A CUDA program is comprised of code for the host CPU and kernels for the GPU devices. A kernel is a program launched over a set of lightweight parallel threads on GPUs, where the threads are organized into a grid of blocks. All threads in a block are split for execution

```
Input: Vectors $F_i$ and $F_j$ of length $M$ with the information of each feature.
Output: Joint vector $J$ of length $M$.
$max_i = \text{Max}(F_i)$
$max_j = \text{Max}(F_j)$
Initialize $states$ of size $[max_i][max_j]$ to zero
$iterSt = 1$
for every sample $s$: $0 \leq s < M$ do
    $st = states[F_i[s]][F_i[s]]$
    if $st == 0$ then
        // First time of the combination
        $J[s] = iterSt$
        $states[F_i[s]][F_i[s]] = iterSt$
        $iterSt = iterSt + 1$
    else
        // Same state as samples with the same combination
        $J[s] = st$
    end
end
```

Algorithm 2: Pseudo-code of the joint vector calculation.

into small groups of 32 parallel threads, called warps. These warps are scheduled in a single instruction, multiple thread fashion. Full efficiency and performance can be obtained when all threads in a warp execute the same instruction.

CUDA-enabled GPUs have evolved into highly parallel many-core processors with tremendous compute power and very high memory bandwidth. They are especially well-suited to address computational problems with high data parallelism and arithmetic density. A CUDA-enabled GPU can be conceptualized as a fully configurable array of Scalar Processors (SPs). These SPs are further organized into a set of Streaming Multiprocessors (SMs). The total amount of SMs depends on the GPU model, while the number of SPs per SM can vary depending on the underlying GPU microarchitecture. For instance, GPUs of the Kepler and Maxwell generations are used in the experimental evaluation, whose SMs contain 192 and 128 SPs, respectively.

Figure 2 shows the overall structure and memory hierarchy of the GPU architecture. Remark that the GPU contains several kinds of memories with different characteristics. Global memory is relatively large (in the order of GBs) but has lower bandwidth and higher latency than shared memory, with only tens of KBs. As can be seen in Figure 2, this memory is only shared among the SPs within the same SM. Read-only memories with similar bandwidth and latency as shared memory and also only tens of KBs are available (constant and texture memories).

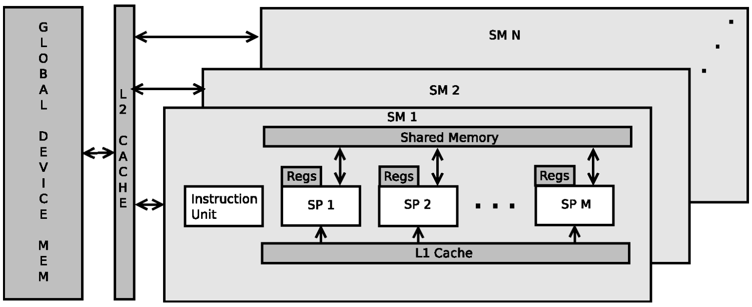

Figure 2: GPU architecture and memory hierarchy.

4. CUDA-JMI Implementation

CUDA-JMI is a command-line tool that receives as input argument a file with the data, supporting different file formats. At the moment of writing this manuscript CUDA-JMI can work with arff, csv and libsvm file formats, but it is expected to increase the number of supported file extensions if required by the users. The data is stored in a $N \times M$ matrix, being N the number of features and M the number of samples. In the case of microarrays the rows and columns represent the genes and individuals, respectively. The class value for each individual is also read from the input file and stored in an array. Remark that MI and JMI work only with discrete values. In case of datasets with continuous values a two-step discretization procedure is performed when reading the input:

1. The continuous range is partitioned into segments or bins of equal size.
2. Each continuous value is assigned to the bin representing the segment covering it.

The input file, the number of bins, the number of features to select, and other configuration parameters are specified in the command line. An explanation of all the arguments, as well as installation instructions, are included in the reference manual available with the tool.

4.1. Conceptual Parallelization Logic

Algorithm 3 illustrates the general pseudo-code of the parallel FS method implemented in CUDA-JMI. Before working on the parallelization, an optimization of the general approach to avoid repeating several calculations was performed. Concretely, the loop of Line 11 in Algorithm 1 was removed by using an array $Score$ that accumulates the JMI of each feature with all the features inserted in S. Therefore, instead of repeating the calculation of the JMI to all the subset on every iteration k, only the JMI with the last inserted feature must be calculated (Lines 12 and 13 in Algorithm 3).

The nested loops included in Algorithm 3 were analyzed when looking for parallelization opportunities. One iteration of the outer loop (Line 8) includes all the computation necessary to determine the next feature to select. Each iteration explores the whole input dataset and the loop continues while more features must be included in the output. It was discarded for parallelization due to the dependency among its iterations: iteration $k+1$ must wait

```
1  Input: Dataset with $M$ individuals, $N$ features, and one class $Y$; Number of features to
     select $NS$
2  Output: Subset $S$ with the $NS$ features selected
3  Initialize $S$ to empty
4  for every feature $F_i$ with $0 \leq i < N$ do
5      Calculate $MI(F_i; Y)$
   end
6  Insert in $S$ the $F_h$ with the highest $MI$
7  Initialize an array $Score[N]$ to zero
8  for every $k$ with $0 < k < NS$ do
9      // Start the parallel region
10     for every feature $F_i$ with $0 \leq i < N$ do
11         if $F_i$ is not in $S$ then
12             Create $J$ as the joint of $F_i$ and $F_h$
13             $Score[i] = Score[i] + MI(J; Y)$
           end
       end
14     // End of the parallel region
15     Insert in $S$ the $F_h$ with the highest $Score$
   end
```

Algorithm 3: Pseudo-code of the parallel FS method based on maximizing JMI provided by CUDA-JMI.

until iteration k has selected its feature, in order to calculate the appropriate joints. The parallelization is focused on the inner loop (Line 10), whose iterations are completely independent. Features are assigned to different hardware (horizontal partitioning of the input dataset) so that their JMI values with respect to the last selected feature are calculated in parallel (see Figure 3). For instance, in the case of genetic data, the genes (features) are distributed among the hardware, while the expression values of all individuals for a certain gene would be assigned to the same thread/process. This procedure is repeated for every iteration of the outer loop.

4.2. GPU Kernels to Calculate Mutual Information

In order to exploit the computational capabilities of heterogeneous platforms, each step of the JMI calculation is assigned to different hardware: the joint vector calculation is performed by the CPU cores, while the MI computation by the GPUs. This workload distribution has been chosen because the vector joint would obtain poor performance on GPUs due to many random memory accesses to a map large enough to not fit in the shared memory of the GPU (see Algorithm 2). The features are divided into batches so that the CPU hardware can work over some features at the same time that the GPUs calculate the MI of the previous ones. Figure 4 illustrates this concept. For instance, in genetic scenarios the genes would be divided into different batches, and the CPU cores would analyze a group

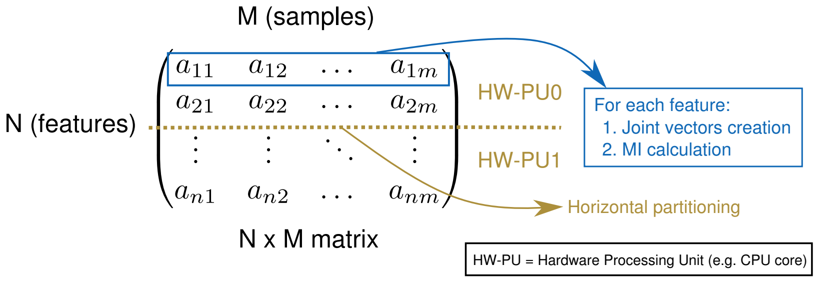
M (samples)
N (features)
a_{11} a_{12} … a_{1m}
a_{21} a_{22} … a_{2m}
a_{n1} a_{n2} … a_{nm}
N x M matrix
HW-PU0
HW-PU1
For each feature:
1. Joint vectors creation
2. MI calculation
Horizontal partitioning
HW-PU = Hardware Processing Unit (e.g. CPU core)

Figure 3: Conceptual parallelization logic of the input dataset ($N \times M$ matrix)

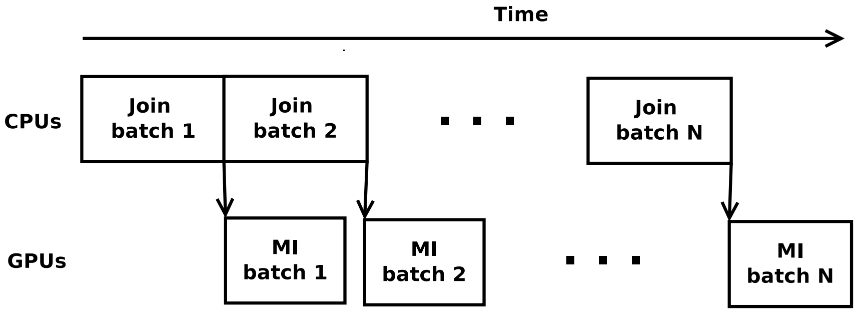

Figure 4: Overlapping between CPU and GPU work. The CPUs join features while the GPUs calculate the MI of the previous batch.

of genes while the GPU analyzes the previous one. The overlapping of computation between CPUs and GPUs is achieved thanks to the proper use of asynchronous CUDA functions to transfer the joint vectors of each batch. Remark that CUDA-JMI provides support to perform the joints of the features within a batch in parallel on several CPU cores thanks to the use of C++ threads [36], as will be explained later. The number of batches is dynamically and automatically defined by the aplication (transparently to the user), looking for a number high enough to allow CPU-GPU overlapping, but not too high to create so small batches that are not able to exploit all the resources of the GPU.

The calculation of the MI values between a batch of joint vectors and the class is performed by GPUs. Algorithm 4 summarizes the behavior of the kernel for each batch. Remark that the term batch represents a completely different concept from the already explained CUDA block. A batch groups a subset of features that are computed together, first by the CPU and later by the GPU. Once the batch arrives to the GPU, its computation is divided into CUDA threads, which are grouped into blocks (a CUDA concept), so that each one is in charge of different joint vectors. Each block i starts working over the $i-th$ vector and then continues iterating on the batch with a stride of $gridDim.x$ (i.e. distributes the joint vectors in a round-robin or cyclic fashion as specified in Line 3 of Algorithm 4). The number of threads per block is predefined by the user.

As previously explained, it is necessary to calculate the joint probability of each state in J and class Y in order to calculate the entropies and the MI. This is performed through a map that counts the number of appearances of each combination. The threads of the block

```
1  Input: $b$x$M$ array $J$, where $b$ is the number of joint vectors in the batch and $M$ the number
     of individuals; Vector $Y$ with the class of the individuals; $maxSt$ and $maxY$ with the
     maximum state and class, respectively
2  Output: Array $Score$ with the MI of the features in the batch
3  for $i = blockIdx.x;\ i < b;\ i = i + gridDim.x$ do
4      for $pos = threadIdx.x;\ pos < maxY * maxSt;\ pos = pos + blockDim.x$ do
5          Initialize position $pos$ in the map as zero
       end
6      Synchronize threads of the CUDA block
7      for $s = threadIdx.x;\ s < M;\ s = s + blockDim.x$ do
8          $atomicAdd$ of one to position $J[i][s]$ of the map
       end
9      Synchronize threads of the CUDA block
10     $myMI = 0$
11     for $pos = threadIdx.x;\ pos < maxY * maxSt;\ pos = pos + blockDim.x$ do
12         $myMI -=$ Entropy with position $pos$ of the map
13         Reduce the $myMI$ values of the block into $Score[i]$
       end
   end
```

Algorithm 4: Pseudo-code of the MI kernel for a batch of joint vectors.

collaborate during the calculation of the MI of each joint vector during the three steps of the kernel, resulting in a two level parallelization:

1. Initializing the map to zero (Lines 4-5).
2. Counting the number of appearances of each combination and storing them in the map (Lines 7-8). An atomic operation is used in order to avoid race conditions when several threads need to update the same position of the map.
3. Calculating the MI taking into account the values of the map (Lines 11-13). This step needs to reduce the partial results of all threads, which is also performed through atomic operations.

Accesses to the map, with and without atomic operations, are present on the three steps and thus have significant impact on the performance of the kernel. CUDA-JMI tries to store the map in the shared memory of the GPU in order to reduce the performance degradation due to the irregular accesses to it necessary for the second step. However, sometimes the use of shared memory is not possible or not advisable. On the one hand, the space necessary for the map can be too large to fit in the shared memory. The map requires $maxY * maxSt$ integers, being $maxSt$ and $maxY$ the maximum state of the joint vector J and the maximum value of the class Y, respectively. Moreover, each integer must be represented with 32 bits so that the atomics can work efficiently. On the other hand, support for efficient atomic operations over shared memory has been included in CUDA only after version 5.0 (Maxwell). Therefore, CUDA-JMI includes two kernels: one with the map stored in global memory, used either when the map does not fit in shared memory or the architecture of the GPU is lower than 5.0; and another one with the map stored in shared memory that is used otherwise. The suitability of this approach to select the best kernel on each scenario will be assessed in Section 5.1.

Among other optimizations included in the CUDA-JMI code to improve its performance we should remark:

- The joint vectors in CUDA-JMI are based on 16-bit integers instead of the 32-bit ones used by FEAST. This reduces the cost of the memory transfers between CPU and GPU while still being large enough to represent all the state combinations.
- The probabilities of the classes necessary to calculate their entropy (see Formula 4) are the same for all MI calculations. Thus, they are calculated only once at the beginning of the CUDA-JMI execution and stored in the constant memory of the GPU.
- The calculation of the probabilities of the states of the joint vector need a division by the number of samples M. The kernel starts calculating once the value $\frac{1}{M}$ and then multiplies to this value which is faster than directly dividing by M.
- Accesses to the array J in Line 8 of Algorithm 4 are coalesced as threads within each block work over different samples with a round-robin/cyclic distribution.
- As previously mentioned, the CPU-GPU transfers of the joint vectors are carried out with asynchronous CUDA copies, which allows to overlap the computation of CPU and GPU hardware.

4.3. Exploitation of Multiple CPUs and GPUs

It has already been explained how a CPU and a GPU can collaborate to complete the FS method of CUDA-JMI, with each hardware associated to a different part of the algorithm. However, many computing platforms contain several CPU cores and GPU devices. Therefore, CUDA-JMI applies another level of parallelism using C++ threads [36] so that each phase can be performed on parallel by multiple CPU cores and/or GPU devices.

The number of CPU threads nt for the first phase (creation of the joint vectors) can be indicated as an input argument through command line. CUDA-JMI then applies a pure block distribution that assigns $\frac{b}{nt}$ consecutive features to each thread, being b the number of features per batch. Remark that a distribution of the workload by features is the only advisable solution, as determining the state for certain individual depends on the values of the map generated by the previous ones (see Algorithm 2).

Regarding the multi-GPU implementation of the second phase (MI calculation), one C++ thread per GPU is created. Each thread is mapped into a different CPU core, being in charge of transferring the data to and from its assigned GPU. Again, a pure block distribution that divides the joint vector evenly among the GPUs, and with consecutive vectors associated to the same GPU, is applied.

An example of a CUDA-JMI execution in a machine with several CPU cores and GPUs is presented in Figure 5. This approach is designed for systems where all the GPUs have the

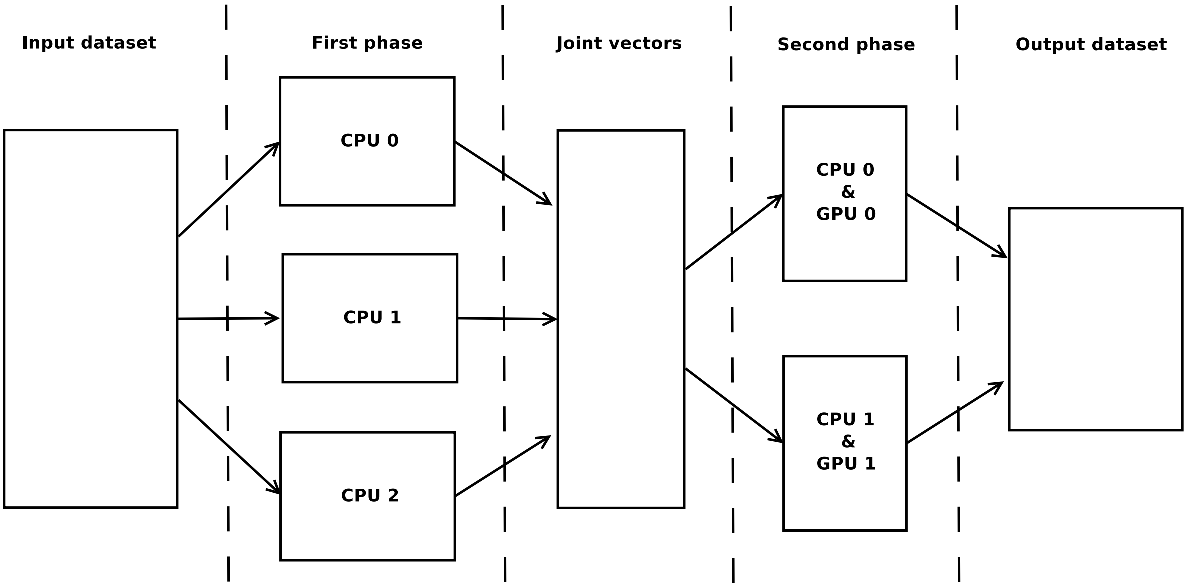

Figure 5: An example of CUDA-JMI execution in a platform with three CPU cores and two GPUs.

same performance, which is the most common scenario in current HPC platforms. In this case, the workload per GPU is completely balanced (i.e. they work over the same number of features).

5. Experimental Evaluation

The experimental evaluation of our work is focused on performance in terms of execution time, as the features selected by CUDA-JMI are identical to those provided by the JMI-based method included in FEAST, which is extensively used by researchers. Three systems were used for this evaluation:

- *Sys1*: A multicore system with two octa-core Intel Xeon E5-2660 Sandy Bridge-EP processors (in total, 16 CPU cores at 2.20GHz) that contains three NVIDIA Tesla Kepler K20m GPUs. CUDA-JMI was compiled with the GNU suite v4.8.2 and the CUDA toolkit v8.0.61.

- *Sys2*: This system provides the same CPU model and compilers as *Sys1* but has one NVIDIA Tesla Kepler K40c GPU.

- *Sys3*: A shared-memory machine with 24 cores (two 12-core Intel Xeon E5-2690 v3 Haswell-EP processors at 2.60GHz). This machine contains two NVIDIA Tesla Maxwell M60 boards, each one providing two GPUs. The compilers used for the evaluation were the GNU suite v4.8.5 and the CUDA toolkit v9.1.85.3.

More information about the characteristics of the GPUs are shown in Table 2. Five publicly available datasets with different characteristics (summarized in Table 3) have been

Table 2: Characteristics of the GPU boards (*these values are for each GPU of the board).

	K20m	K40c	M60
Microarchitecture	Kepler	Kepler	Maxwell
GPU chip	GK110	GK110B	GM204
Number of GPUs	1	1	2
Compute capability	3.5	3.5	5.2
Number of SMs	13	15	16*
Number of cores	2,496	2,880	2,048*
Core frequency (MHz)	706	745	899
Memory size (GB)	5	12	8*
Shared memory (KB)	64	64	64*
Memory bandwidth (GB/s)	208	288	320
Power consumption (W)	225	235	300

Table 3: Characteristics of the datasets.

	Features	Samples	Classes
BreastCancer	24,481	97	2
ECML	27,679	90	43
rcv1train	47,236	20,242	2
news20	62,061	15,935	20
SVHN	3,072	531,131	10

evaluated. BreastCancer is a microarray used for genetic experiments [37], which usually present a higher number of features (genes) than samples (individuals). A similar proportion of features/samples but with multiple classes is followed by the ECML dataset, which was prepared by downloading and processing various information from the SAGEmap website as of December 2002 (http://eps.upo.es/bigs/datasets.html). The last three datasets were obtained from the LibSVM collection [38]. While recv1train and news20 are respectively biclass and multiclass examples of datasets with tens of thousands of features and samples, SVHN is used as an example of dataset with a much higher amount of individuals than features. All the experiments shown in this manuscript have been obtained after applying a discretization with 64 bins. Although additional experiments with 128 and 256 bins were run, they are not included in this section for simplicity as the conclusions obtained from them are similar.

The number of selected features has also been fixed to 200. This number is high enough to show the impact of discarding the selected features from the candidates list, and not too large to avoid selecting an extremely high number of features, which will never be the case in a real scenario. Additionally, the speedup between the original FEAST implementation and CUDA-JMI does not depend on the number of selected features as the parallel approach is repeated for every feature selected (every iteration of the loop in Lines 7 and 8 of Algorithms 1 and 3, respectively).

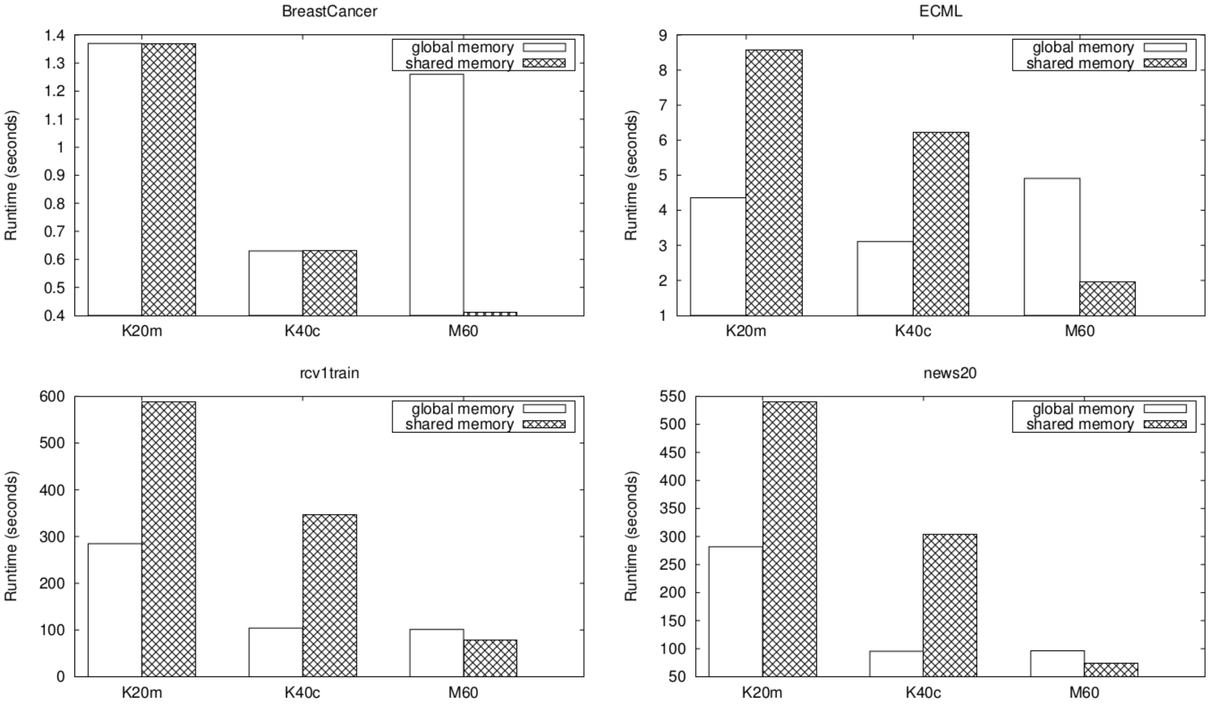

Figure 6: Runtime of the kernels to calculate MI on three different GPUs when selecting 200 features.

5.1. Best Kernel Configuration

CUDA-JMI includes two kernels for MI calculations, depending on the type of memory (global or shared) used to store the maps. As mentioned in Section 4.2, the kernel choice not only depends on the memory requirements of the map but also on the characteristics of the GPU. The experimental evaluation starts checking which kernel is the most appropriate for each available GPU.

Figure 6 shows the runtime of the two kernels executed by the three different GPUs over four datasets. The SVHN dataset is not present in this analysis as it is the only one whose map does not fit in shared memory. Consequently, the global-memory-based kernel must be always used and this comparison does not make sense for this dataset. Remark that the runtime shown for the M60 board was obtained with only one GPU.

The results show that the use of shared memory to store the map is only beneficial on the GPU of the M60 board, which provides CUDA computational capability higher than 5.0 and hardware support for atomics on shared memory. Shared-memory atomics on the K20m and K40c GPUs are implemented in software, being significantly slower. Consequently, the already explained protocol to choose the best kernel for computation is correct: the shared-memory-based kernel is only selected when: 1) the map fits in it and 2) the GPU provides support for hardware shared-memory atomics (computational capability higher than 5.0). CUDA-JMI automatically checks at execution time if both conditions are fulfilled. Otherwise, the global-memory-based kernel is selected.

Table 4: Runtime (in seconds) of the MI calculation in several K20m GPUs and acceleration compared to the single-GPU runtime.

Datasets	2 GPUs		3 GPUs	
	time	acc.	time	acc.
BreastCancer	0.78	1.68	0.56	2.35
ECML	5.63	1.97	3.82	2.91
rcv1train	142.94	1.99	95.53	2.98
news20	141.43	1.99	94.29	2.99
SVHN	531.32	2.0	348.21	3.0

Table 5: Runtime (in seconds) of the MI calculation in several M60 boards and acceleration compared to the single-GPU runtime.

Datasets	2 GPUs				4 GPUs	
	Same board		Diff. board			
	time	acc.	time	acc.	time	acc.
BreastCancer	0.30	1.37	0.22	1.83	0.17	2.45
ECML	1.39	1.40	1.21	1.61	0.79	2.48
rcv1train	69.55	1.12	39.17	1.98	35.01	2.22
news20	69.23	1.08	37.78	1.98	34.83	2.15
SVHN	544.10	1.65	490.53	1.82	277.95	3.22

5.2. Evaluation for Multiple GPUs

Table 4 shows the runtime of the MI calculation using several K20m GPUs in *Sys1*, as well as the corresponding acceleration compared to the single-GPU execution. Remark that CUDA-JMI selects the global-memory-based kernel in all cases as these GPUs do not provide support for efficient hardware memory atomics. The multi-GPU execution of CUDA-JMI is advantageous in this system, with a performance improvement almost linear with the number of GPUs (excepting the smallest dataset whose runtime with only one GPU is already very low).

Table 5 shows similar results using up to the four GPUs available in *Sys3*. In this system CUDA-JMI uses shared memory to store the maps during the experiments with BreastCancer, ECML, rcv1train and news20, while global memory for SVHN as its map does not fit in shared memory. Two different results using two GPUs are provided, distinguishing when the MI calculation is executed on the two GPUs of the same M60 board or selecting one GPU of each board. The runtime is reduced almost to half of the single-GPU execution when only one GPU per board is used. Nevertheless, the speedup is significantly reduced when CUDA-JMI tries to exploit one whole board, probably because the two CPU threads associated to the GPUs share the link between CPU and GPU. This fact also penalizes the performance of the four-GPU execution as both GPUs of the two boards are used.

Table 6: Speedup of CUDA-JMI over the JMI-based sequential method of FEAST when selecting 200 features.

Datasets	*Sys1*	*Sys2*	*Sys3*
BreastCancer	42.95	36.35	45.94
ECML	20.06	10.09	52.26
rcv1train	13.15	12.41	33.00
news20	13.03	12.90	32.23
SVHN	10.53	5.74	13.57

5.3. Evaluation in Heterogeneous Systems

The experimental evaluation finishes comparing the JMI-based FS method of FEAST (the most commonly used sequential counterpart) with CUDA-JMI on the three systems, with the CPU cores joining the vectors while the GPUs calculate the MI of the vectors joint in the previous batch (see Figure 4). Figure 7 shows the runtime of both approaches, with CUDA-JMI fully exploiting the systems (using all the available CPU cores and GPU devices). I/O times are not included as they are identical for FEAST and CUDA-JMI and they hardly depend on the file format of the input dataset. Table 6 shows the speedups of CUDA-JMI over FEAST for each dataset and system.

These results prove that CUDA-JMI is able to significantly reduce the time needed to complete FS. The magnitude of the acceleration depends on the hardware available in the system (amount and computational capability of CPU cores and GPU devices) as well as on the characteristics of the datasets (amount of features, samples and classes). As expected, the fastest runtime and the highest speedup for each dataset has been obtained in *Sys3*, as this system does not only provide more CPU and GPU resources than the other ones, but also at the highest clock frequencies. Comparing the datasets, the speedups are on average higher for those scenarios with not many samples (very common on biological studies). The lowest acceleration is achieved for SVHN as the map is too large to fit in shared memory. Finally, remark that the use of CUDA-JMI is beneficial for all combinations of datasets and systems, reaching an impressive speedup of more than 50x when analyzing ECML in *Sys3*. A comparison to the GPU implementation of fast-mRMR [32] has not been included in this paper as the underlying FS method is different. Nevertheless, their authors reflect a maximum speedup over a sequential implementation of only 5x, which is significantly lower than the speedups obtained by CUDA-JMI.

6. Conclusions

FS is a data mining technique that is nowadays a common and extremely important step in machine learning, especially with the continuous increase of the average dataset sizes on different fields such as text mining, genetics or bioinformatics. Among the many methods existing for FS, those based on JMI have been proved to provide a set of features with more complementary information than those based on pure MI. However, the quadratic complexity of these methods with the number of features prevents their use over large datasets.

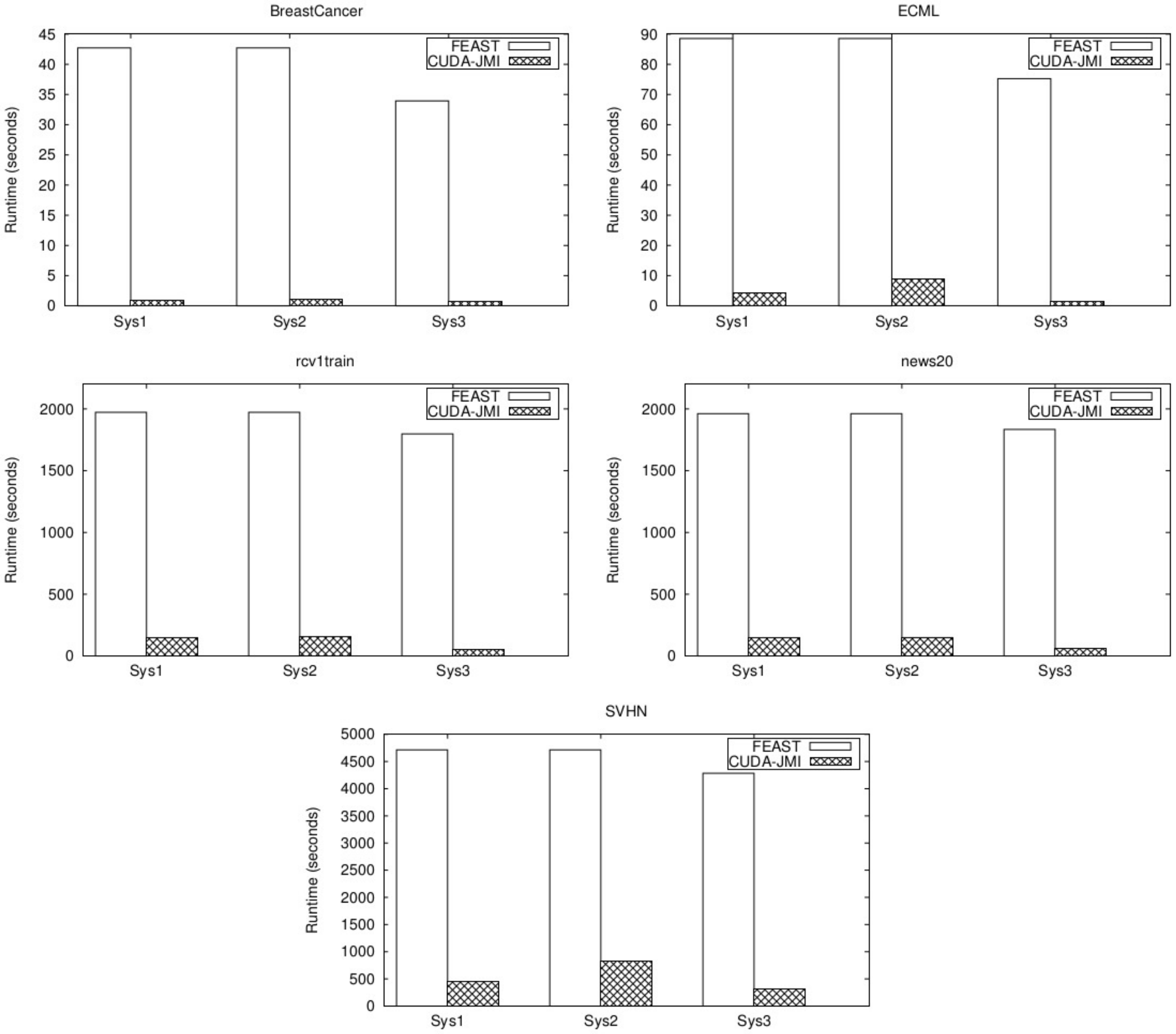

Figure 7: Runtime of CUDA-JMI and the JMI-based sequential method of FEAST when selecting 200 features.

This work has presented CUDA-JMI, a high-performance C++-based FS tool for heterogeneous systems with several CPU cores and NVIDIA GPU devices. Taking as basis a fast JMI implementation available in the FEAST toolkit, CUDA-JMI is able to efficiently exploit all the computing resources of modern shared-memory platforms and thus processing large datasets in a reasonable period of time. The computation is divided into two main steps and each one is processed simultaneously by the most suitable hardware: joint of candidate features with the last selected one by the CPU cores while MI calculation for the joint vectors by the GPU devices. The scalability has been tested on three heterogeneous shared-memory systems using five public datasets, all of them with different characteristics. CUDA-JMI obtains high speedups (up to 50x) compared to the state-of-the-art toolkit FEAST.

As future work we aim to adapt our parallel approach to other FS methods with the goal of providing a broad suite where the future users can choose the parallel algorithms that better adapts to their data and analyses. Once several FS methods have been parallelized, another future step would be the development of parallel ensembled methods, where several FS algorithms are executed at the same time and the final output is a conjunction of the partial results. The input formats accepted by CUDA-JMI are also expected to be extended after feedback with the users. Finally, we will also try to include a dynamic distribution of the joint vectors in order to improve the performance of CUDA-JMI in platforms with different types of GPUs.

Acknowledgements

This research has been partially funded by projects TIN2016-75845-P and TIN-2015-65069-C2-1-R of the Ministry of Economy, Industry and Competitiveness of Spain, as well as by Xunta de Galicia projects ED431D R2016/045, ED431G/01 and GRC2014/035, all of them partially funded by FEDER funds of the European Union.

References

[1] Y. Zhai, Y.-S. Ong, I. W. Tsang, The Emerging Big Dimensionality, IEEE Computational Intelligence Magazine 9 (3) (2014) 14–26.

[2] I. Guyon, A. Elisseeff, An Introduction to Variable and Feature Selection, Journal of Machine Learning Research 3 (2003) 1157–1182.

[3] H. Liu, H. Motoda, Feature Selection for Knowledge Discovery and Data Mining, Springer Science & Business Media, 2012.

[4] V. Bolón-Canedo, N. Sánchez-Maroño, A. Alonso-Betanzos, Feature Selection for High-Dimensional Data, Springer, 2015.

[5] V. Bolón-Canedo, N. Sánchez-Marono, A. Alonso-Betanzos, J. M. Benítez, F. Herrera, A Review of Microarray Datasets and Applied Feature Selection Methods, Information Sciences 282 (2014) 111–135.

[6] Z. M. Hira, D. F. Gillies, A Review of Feature Selection and Feature Extraction Methods Applied on Microarray Data, Advances in Bioinformatics 2015.

[7] G. Brown, A. Pocock, M.-J. Zhao, M. Luján, Conditional Likelihood Maximisation: a Unifying Framework for Information Theoretic Feature Selection, Journal of Machine Learning Research 13 (2012) 27–66.

[8] H. Yang, J. Moody, Feature Selection Based on Joint Mutual Information, in: 3rd ACM SIGKDD International Symposium on Advances in Intelligent Data Analysis (IDA 1999), Amsterdam, The Netherlands, 1999, pp. 22–25.

[9] H. Yang, J. Moody, Data Visualization and Feature Selection: New Algorithms for Nongaussian Data, in: 12th International Conference on Neural Information Processing Systems (NIPS 1999), Denver, CO, USA, 1999, pp. 687–693.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The WEKA Data Mining Software: an Update, ACM SIGKDD Explorations Newsletter 11 (1) (2009) 10–18.

[11] A. Kleerekoper, M. Pappas, A. Pocock, G. Brown, M. Luján, A Scalable Implementation of Information Theoretic Feature Selection for High Dimensional Data, in: 3rd IEEE International Conference on Big Data (Big Data 2015), Santa Clara, CA, USA, 2015, pp. 339–346.

[12] O. Soufan, D. Kleftogiannis, P. Kalnis, V. B. Bajic, DWFS: A Wrapper Feature Selection Tool Based on a Parallel Genetic Algorithm, PLoS ONE 10 (2) (2015) 1–23.

[13] H. Yang, R. Fujimaki, Y. Kusumura, J. Liu, Online Feature Selection: A Limited-Memory Substitution Algorithm and its Asynchronous Parallel Variation, in: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016), London, UK, 2016, pp. 1945–1954.

[14] J. Yazidi, W. Bouaguel, N. Essoussi, A Parallel Implementation of Relief Algorithm Using MapReduce Paradigm, in: 8th International Conference on Computational Collective Intelligence (ICCI 2016), Halkidiki, Greece, 2016, pp. 418–425.

[15] D. Peralta, S. Río, S. Ramírez-Gallego, I. Triguero, J. M. Benítez, F. Herrera, Evolutionary Feature Selection for Big Data Classification: A MapReduce Approach, Mathematical Problems in Engineering (2015) 1–11.

[16] Q. He, X. Cheng, F. Zhuang, Z. Shi, Parallel Feature Selection Using Positive Approximation Based on MapReduce, in: 11th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD 2014), Xiamen, China, 2014, pp. 397–402.

[17] D. Singh, C. K. Reddy, A Survey on Platforms for Big Data Analytics, Journal of Big Data 2 (1) (2015) 8.

[18] I. Anagnostopoulos, S. Zeadally, E. Exposito, Handling Big Data: Research Challenges and Future Directions, The Journal of Supercomputing 72 (4) (2016) 1494–1516.

[19] H. Jiang, Y. Chen, Z. Qiao, T.-H. Weng, K.-C. Li, Scaling up MapReduce-based Big Data Processing on Multi-GPU Systems, Cluster Computing 18 (1) (2015) 369–383.

[20] J. A. Stuart, J. D. Owens, Multi-GPU MapReduce on GPU clusters, in: 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011), Anchorage, AK, USA, 2011, pp. 1068–1079.

[21] A. Cano, J. M. Luna, S. Ventura, High Performance Evaluation of Evolutionary-Mined Association Rules on GPUs, The Journal of Supercomputing 66 (3) (2013) 1438–1461.

[22] Y. Djenouri, A. Bendjoudi, M. Mehdi, N. Nouali-Taboudjemat, Z. Habbas, GPU-Based Bees Swarm Optimization for Association Rules Mining, The Journal of Supercomputing 71 (4) (2015) 1318–1344.

[23] L. Jian, C. Wang, Y. Liu, S. Liang, W. Yi, Y. Shi, Parallel Data Mining Techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA), The Journal of Supercomputing 64 (3) (2013) 942–967.

[24] K. Jurczuk, M. Czajkowski, M. Kretowski, Evolutionary Induction of a Decision Tree for Large-Scale Data: a GPU-Based Approach, Soft Computing 21 (24) (2016) 7363–7379.

[25] B. Liu, Y. Xin, R. C. Cheung, HongYan, GPU-Based Biclustering for Microarray Data Analysis in Neurocomputing, Neurocomputing 134 (2014) 239–246.

[26] P. Orzechowski, K. Boryczko, Rough Assessment of GPU Capabilities for Parallel PCC-Based Biclustering Method Applied to Microarray Data Sets, Bio-Algorithms and Med-Systems 11 (4) (2015) 243–248.

[27] H.-H. Chang, C.-Y. Li, An Automatic Restoration Framework Based on GPU-Accelerated Collateral Filtering in Brain MR Images, BMC Medical Imaging 19 (1) (2019) 8.

[28] S. Cuomo, A. Galletti, L. Marcellino, G. Navarra, G. Toraldo, On GPU-CUDA as Preprocessing of Fuzzy-Rough Data Reduction by Means of Singular Value Decomposition, Soft Computing 22 (5) (2018) 1525–1532.

[29] J. Yang, S. Jing, Acceleration of Feature Subset Selection Using CUDA, in: 14th International Confer-

ence on Computational Intelligence and Security (CIS 2018), Hangzhou, China, 2018, pp. 140–144.

[30] J. J. Escobar, J. Ortega, J. González, M. Damas, A. F. Díaz, Parallel High-Dimensional Multi-Objective Feature Selection for EEG Classification with Dynamic Workload Balancing on CPU–GPU Architectures, Cluster Computing 20 (3) (2017) 1881–1897.

[31] H. Zhu, Y. Wu, P. Li, P. Zhang, Z. Ji, M. Gong, An OpenCL-Accelerated Parallel Immunodominance Clone Selection Algorithm for Feature Selection, Concurrency and Computation: Practice and Experience 29 (9) (2017) e3838.

[32] S. Ramírez-Gallego, I. Lastra, D. Martínez-Rego, V. Bolón-Canedo, J. M. Benítez, F. Herrera, A. Alonso-Betanzos, Fast-mRMR: Fast Minimum Redundancy Maximum Relevance Algorithm for High-Dimensional Big Data, International Journal of Intelligent Systems 32 (2) (2017) 134–152.

[33] F. Azmandian, A. Yilmazer, J. G. Dy, J. A. Aslam, D. R. Kaeli, Harnessing the Power of GPUs to Speed Up Feature Selection for Outlier Detection, Journal of Computer Science and Technology 29 (3) (2014) 408–422.

[34] A. Guillén, M. I. García-Arenas, M. Heeswijk, D. Sovilj, A. Lendasse, L. J. Herrera, H. Pomares, I. Rojas, Fast Feature Selection in a GPU Cluster Using the Delta Test, Entropy 16 (2) (2014) 854–869.

[35] NVIDIA Developer CUDA Zone (Last visit October 2018), https://developer.nvidia.com/category/zone/cuda-zone (2018).

[36] B. Yablonsky, C++11 Standard Library: Usage and Implementation, CreateSpace Independent Publishing Platform, 2013.

[37] Z. Zhu, Y.-S. Ong, M. Dash, Markov Blanket-Embedded Genetic Algorithm for Gene Selection, Pattern Recognition 40 (11) (2007) 3236–3248.

[38] C.-C. Chang, C.-J. Linn, LIBSVM: a Library for Support Vector Machines, ACM Transactions on Intelligent Systems and Technology (TIST) 2 (3) (2011) 27.