Cluster Computing (2026) 29:280
https://doi.org/10.1007/s10586-026-06076-y

CUDA acceleration of feature selection methods based on conditional mutual information

Bieito Beceiro[1] · **Jorge González-Domínguez**[1] · **Juan Touriño**[1]

Received: 27 March 2025 / Revised: 8 October 2025 / Accepted: 2 March 2026

Abstract

Feature Selection (FS) is a fundamental data mining technique that improves machine learning models by eliminating irrelevant or redundant data, especially as dataset dimensions continue to grow. However, greedy FS methods can be computationally intensive, particularly for large datasets. This work focuses on the development of CUDA implementations for FS methods that use Conditional Mutual Information (CMI). Specifically, we address the Interaction Capping (ICAP) method and the family of algorithms that can be included in the BetaGamma space. The proposed implementations leverage memory hierarchy, asynchronous computation, and a reduced-precision approach to efficiently exploit the hardware of NVIDIA GPUs to achieve high performance with relatively low energy costs. The experimental evaluation has been carried out in two systems with different types and numbers of CPUs and GPUs using four publicly available datasets from different fields. We first provide insight into the performance impact of each optimization technique, and also into the combination of configuration parameters that best exploits the GPU architecture. Finally, the novel GPU implementations are compared to their fastest available counterparts (parallel implementations for multi-CPU systems), proving that the GPU versions are significantly faster than these multithreaded approaches in all scenarios using 32 or 64 CPU cores.

Keywords Feature Selection · Conditional Mutual Information · CUDA · GPU

1 Introduction

Feature Selection (FS) is a data mining technique that identifies features that are relevant to the knowledge by discarding irrelevant and/or redundant data [1]. FS has become key in most Machine Learning (ML) studies due to the continuous increase in the dimensions of the datasets. Note that the analysis of extremely large datasets can require too long runtime and even the conclusions may be wrong if redundant and irrelevant data are not removed [2].

There are many methods for FS, each one with its advantages and drawbacks. The greedy algorithms are the most used ones, whose procedure starts with an empty set and adds one feature per iteration up to a maximum indicated by the user. In each iteration, a score is assigned to each feature to decide which one will be added. In general, there are two approaches to assign these scores: wrapper (score based on the accuracy of a certain ML algorithm) and filter methods (score independent of any subsequent training method). In this work, we focus on the latter because they are more general and suitable for a wider range of ML pipelines.

However, greedy FS methods are computationally expensive and can require long execution time when applied to large datasets. GPUs can help alleviate this problem by providing massive parallelism with lower power consumption than traditional multicore systems. To the best of our knowledge, the currently available GPU implementations for FS are exclusively based on the Mutual Information (MI) metric, such as the versions of minimum Redundancy Maximum Relevance (mRMR), Joint Mutual Information (JMI), and Double Input Symmetrical Relevance (DISR) included in the cuFEAST library [3]. Although these methods are

✉ Bieito Beceiro
bieito.beceiro.fernandez@udc.es

Jorge González-Domínguez
jgonzalezd@udc.es

Juan Touriño
juan@udc.es

[1] Universidade da Coruña, CITIC, Computer Architecture Group, Campus de Elviña s/n, A Coruña 15071, A Coruña, Spain

Published online: 01 June 2026

effective in practice, they might fail to capture interactions between features of order higher than two. Some authors have tried to overcome this limitation by including the Conditional Mutual Information (CMI) in the decision of the features to select. Methods that use this metric along with MI evaluate additional relationships between features, enhancing the quality of the selected subset. However, CMI is more computationally expensive than MI, so its usage entails a tradeoff between higher accuracy and faster FS.

In this work, we present a CUDA implementation of the Interaction Capping (ICAP) [4] and BetaGamma FS methods, which are based not only on MI but also on CMI. As explained in [5], BetaGamma is not a standalone method, but a generalization of FS methods that use a linear combination of MI and CMI. It assigns a weight to each metric and includes, for example, the Mutual Information Feature Selection (MIFS) [6] and the Conditional Infomax Feature Extraction (CIFE) [7]. Our work goes beyond a direct translation of the algorithms by introducing GPU-specific optimizations and providing both high-precision and reduced-precision variants. The implementations significantly reduce the FS runtime while preserving accuracy and have been made publicly available as part of the cuFEAST library. The main contributions of this work are:

- The first CUDA implementations of CMI-based FS algorithms (ICAP and the BetaGamma family).
- A reformulation of the algorithms and the addition of GPU-specific optimizations to exploit parallelism, memory hierarchy, and asynchronous computation, while preserving the same results as their sequential counterparts.
- An alternative reduced-precision version (16-bit fixed-point) of the algorithms with negligible accuracy loss and lower computational cost and energy consumption.
- An extensive experimental evaluation on multiple datasets and GPU architectures, analyzing the impact of each optimization and identifying best-performing configurations.
- The open-source release of the implementations in the cuFEAST library (https://gitlab.com/bieito/parallel-fst/-/tree/master/cuFEAST) for reproducibility and community adoption.

The rest of the paper is organized as follows. Related work is presented in Sect. 2. Section 3 explains the FS methods used in this work. The parallel implementation of these methods is described in Sect. 4. Section 5 shows the experimental evaluation, and conclusions are drawn in Sect. 6.

2 Related work

The use of GPUs to accelerate FS has been addressed in several works. To the best of our knowledge, the only publicly available codes are based on the MI metric, which is quite suitable for execution on GPUs [8–10]. Besides the CUDA implementations of several FS algorithms available in the already mentioned cuFEAST library [3], there are other previous works based on the mRMR [11] and the JMI methods [12].

Beyond MI-based criteria, a variety of alternative FS approaches have also been adapted to GPUs. For example, we could mention the CUDA implementations of the Singular Value Decomposition (SVD) [13], a parallel online FS algorithm [14], a CUDA approach for ensemble methods [15], as well as an OpenCL version of the Immunodominance Clone Selection Algorithm (ICSA) [16]. Several domain-specific solutions have also been proposed, such as GPU-based FS for electroencephalogram classification [17], brain image restoration [18], or hyperspectral image classification [19]. While these works demonstrate the applicability of GPUs for FS, they are either not publicly available or tied to specific problems rather than general-purpose FS.

Although none of these methods require CMI, its inclusion along with MI can enable FS to capture higher-order dependencies between features, thus providing more accurate results. The computational cost of CMI is high, but it can be alleviated through high performance computing. To the best of our knowledge, the only publicly available implementations of CMI-based FS are the sequential FEAST library[1] and the multicore Parallel-FST [20], both designed for CPUs. No GPU implementations of these methods have been released so far.

3 Background: Conditional Mutual Information and its use for feature selection

The concepts of the ICAP algorithm and the BetaGamma space, as well as the information metrics needed for them (MI and CMI), are described in this section. The theoretical principles for reduced-precision approaches are also explained.

3.1 Information theory metrics

Many metrics that quantify the information provided by one or more variables are based on the concept of entropy. The entropy of a variable X (i.e., $H(X)$) measures the uncertainty of the values it takes, and is computed as follows:

[1] https://github.com/Craigacp/FEAST

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\log_2 p(x) \tag{1}$$

where $\mathcal{X}$ is the alphabet of values that X can take, and $p(x)$ is the probability of X taking the value x.

The entropy of a variable can be conditioned by another variable. For example, $H(X|Y)$, the entropy of X conditioned by Y, can be interpreted as the amount of uncertainty that X still holds once Y is known. It is computed as:

$$H(X|Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log_2 \frac{p(x,y)}{p(y)} \tag{2}$$

where $\mathcal{X}$ and $\mathcal{Y}$ are the alphabets of values that X and Y can respectively take, $p(x, y)$ is the probability of the joint distribution when X takes the value x and Y takes y, and $p(y)$ is the probability of the marginal distribution of Y. From now on, we will use the abbreviation CH to refer to conditional entropy.

A widely used metric that measures the information provided by two variables is MI. The MI between X and Y, denoted by $I(X; Y)$, can be interpreted as the amount of uncertainty that is removed from X once the value of Y is known. It is calculated as follows:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log_2 \frac{p(x,y)}{p(x)p(y)} \end{aligned} \tag{3}$$

where $\mathcal{X}$ and $\mathcal{Y}$ are the alphabets from which X and Y can take values, $p(x, y)$ is the probability of the joint distribution, and $p(x)$ and $p(y)$ are the probabilities of the marginal distributions. Note that MI is symmetric: $I(X;Y) = I(Y;X)$. MI between a feature and the dataset class can be used as a metric of the variable relevance.

MI between two variables X and Y can also be conditioned by a third variable Z, which can be interpreted as the uncertainty held between X and Y once the value of Z is known. It can be formulated as:

$$\begin{aligned} I(X;Y|Z) &= H(X|Z) - H(X|\mathrm{YZ}) \\ &= \sum_{z\in\mathcal{Z}} p(z)\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y|z)\log_2 \frac{p(x,y|z)}{p(x|z)p(y|z)} \end{aligned} \tag{4}$$

where $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are the alphabets from which X, Y and Z can take values, $p(x, y|z)$, $p(x|z)$ and $p(y|z)$ are the probabilities of the joint and marginal distributions conditioned by the value of Z, and $p(z)$ is the probability of the marginal distribution of Z.

Note that YZ represents the joint state of Y and Z. This can be understood as a single variable that holds the occurrences of pairs of values of Y and Z. Therefore, Eq. 2 can be used to compute not only $H(X|Z)$ but also $H(X|\mathrm{YZ})$.

The computation of CMI involves three different features. As Eq. 4 shows, it can be computed either as a direct calculation or as a difference of the marginal and the conditional entropy (now referred to as CH). Both approaches have drawbacks: the direct approach suffers from cubic computational and spatial complexity, while the entropy difference method requires additional intermediate steps. In this work, we will use the entropy difference approach due to its lower overall complexity and the additional opportunities it presents for optimization.

3.2 Feature selection based on MI and CMI

In this work, we have focused on greedy FS methods that use MI and CMI to compute their scores: BetaGamma, which exposes a parametric space of methods with the β and γ parameters, and ICAP. BetaGamma computes the score of a feature X as follows:

$$\begin{aligned} &\mathrm{BetaGamma}(X) = I(X;C) \\ &- \sum_{Y\in S} \left(\beta\ I(Y;X) - \gamma\ I(Y;X|C)\right) \end{aligned} \tag{5}$$

where C is the class of the dataset and S is the subset of already selected features. The β and γ parameters balance the weight of redundancy and conditional redundancy, respectively. As explained in [5], the BetaGamma space includes MIM [21] ($\beta = \gamma = 0$), MIFS [6] ($\beta = 1, \gamma = 0$), CIFE [7] ($\beta = \gamma = 1$), and also other criteria in which the parameters depend on the number of selected features such as mRMR [22] ($\beta = \frac{1}{|S|}, \gamma = 0$) and JMI [23] ($\beta = \gamma = \frac{1}{|S|}$).

On the other hand, the ICAP method computes the score of a feature X as:

$$\begin{aligned} &\mathrm{ICAP}(X) = I(X;C) \\ &- \sum_{Y\in S} \max\left\{0, I(Y;X) - I(Y;X|C)\right\}. \end{aligned} \tag{6}$$

Note that both scores depend on the current subset of selected features (S). Therefore, the scores need to be calculated again when S changes, that is, every time a new feature is selected.

Both BetaGamma and ICAP share a common structure shown in Algorithm 1. It presents the mechanism for selecting k features from a dataset D with F features and N samples, and with a class C. First, the MI between each feature and the class is calculated (lines 2–4). The feature

with the highest MI is the first included in the output subset (lines 5–6). Then, the `while` loop at line 7 is used to select the other $k-1$ features. The score of each feature is updated in every iteration, computing both MI and CMI with the latest selected feature (lines 8–12), and the feature with the best score is added to the output subset (lines 13–14). The fn function used in line 11 depends on the FS method. For BetaGamma, fn is equivalent to $\beta \cdot \text{mi} - \gamma \cdot \text{cmi}$, and for ICAP to $\max(0, \text{mi} - \text{cmi})$.

Note that in the inner loop (lines 8–12), the MI and CMI of a feature X_i are computed with only one feature Y (last selected) instead of with every $Y \in S$ (all features already selected) as shown in Eqs. 3 and 4. Both computations are equivalent, the difference being that the one used in Algorithm 1 stores the cumulative score of each feature X_i to avoid recalculating the MI and CMI with the other $|S|-1$ previously selected features.

Input: dataset D, dataset class C, total number of features F, number of features to select k
Output: subset of selected features S

1: $scores$: array$[F]$
2: **for** feature $X_i \in D$ **do**
3: $\quad scores_i \leftarrow I(X_i; C)$
4: **end for**
5: $Y \leftarrow X_{\text{argmax}(scores)}$
6: $S \leftarrow S \cup \{Y\}$

7: **while** $|S| < k$ **do**
8: $\quad$ **for** feature $X_i \in D \setminus S$ **do**
9: $\quad\quad mi \leftarrow I(Y; X_i)$
10: $\quad\quad cmi \leftarrow I(Y; X_i|C)$
11: $\quad\quad scores_i \leftarrow scores_i - fn(mi, cmi)$
12: $\quad$ **end for**
13: $\quad Y \leftarrow X_{\text{argmax}(scores)}$
14: $\quad S \leftarrow S \cup \{Y\}$
15: **end while**

Algorithm 1 Common structure of BetaGamma and ICAP

3.3 Low-precision fixed-point implementation of information metrics

The original versions of BetaGamma and ICAP work with double-precision (64 bits) floating-point data ("high precision" from now on). However, high-precision arithmetic is computationally expensive when compared to, for example, integer arithmetic. The works [24, 25] proposed reduced-precision versions for the computation of MI and CMI, respectively, that make use of low-precision fixed-point data to relax the computational requirements associated with double arithmetic. Both works have shown that using 16-bit fixed-point data ("reduced-precision" from now on) is statistically enough to replicate the quality of the results of the high-precision implementation. However, in [24] the results are tested with MIM, mRMR, and JMI, all of which are based on MI and do not use CMI; conversely, in [25] CMI is used, but MI is not.

The reduced-precision alternatives of [24, 25] were initially developed for lower-end devices (such as wearables) that have limited computing power and need to optimize energy consumption to extend battery life. However, as the target architecture of our work is high-end GPUs, we intend to further reduce the FS runtime by exploiting the faster fixed-point arithmetic and the smaller memory footprint of its lower precision representation. Since GPUs are not constrained like lower-end devices, our proposed implementation differs slightly from [25].

The algorithms of this work are based on both MI and CMI. As will be seen in the next section, our approach requires obtaining the CH values of all features instead of a direct calculation as for MI, in order to optimize the CMI computation on GPUs. To calculate MI and CH, there are some computationally expensive operations that are repeated several times (multiplications/divisions of floating-point data and logarithms), as seen in Eqs. 2 and 3. The reduced-precision versions attempt to mitigate these costs in two ways. First, they use logarithmic properties to convert the multiplications and divisions into additions and subtractions. This way, all logarithms in the equations would remain in the form $\log_2(p(\cdot))$. And second, all these logarithms are precomputed and stored in a fixed-point format in a LookUp Table (LUT). In summary, logarithms are changed for accesses to the LUT, while floating-point multiplications and divisions are changed for fixed-point additions and subtractions.

The LUT stores the values of the logarithm for a given probability. Specifically, the probability of a value $p(x)$ is estimated as $f(x)/N$, where $f(x)$ is the frequency of the value and N is the total number of samples in the dataset. The frequency $f(x)$ is used to index the LUT, while N determines its size, since frequencies cannot exceed the total number of samples.

Fixed-point numbers use a fixed amount of bits b_i to represent the integer part of the number, and a fixed amount of bits b_f for the fractional part. To convert to and from fixed

point, we need to define a quantization interval $q = 2^{-b_f}$. Note that each dataset has a different number of samples N, so the LUT must be computed specifically for each input. In addition, N is used to determine the distribution of bits for the integer and fractional parts of the fixed-point representation: b_i is set as the minimum number of bits required to represent the largest value of the LUT, and b_f corresponds to the remaining bits. This leaves most bits for the fractional part, increasing precision.

The values of the LUT $\mathcal{L}$ are computed as follows:

$$\mathcal{L}_n = \left[\frac{\log_2(n/N)}{q}\right]_R ; \forall n \in [1, N] \tag{7}$$

where n represents any possible frequency value, N is the number of samples of the dataset, q is the quantization interval, and $[\cdot]_R$ denotes rounding to the nearest integer.

During the computations, operations between fixed-point values can be performed as long as they have the same b_i and b_f. Then, to convert the fixed-point value d_{fixed} back to floating point, we can apply the following formula:

$$d_{\text{float}} = d_{\text{fixed}} \cdot q \tag{8}$$

where d_{float} is the floating-point value and q is the quantization interval. Then, applying logarithmic transformations and changing them for accesses to the LUT, the equation for the reduced-precision CH could be expressed as:

$$\begin{aligned}\hat{H}(X|Y) = &-\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y) \\ &\cdot[\mathcal{L}_{f(x,y)} - \mathcal{L}_{f(y)}]_d \cdot q\end{aligned} \tag{9}$$

where $\mathcal{L}$ is the LUT, q the quantization interval, $f(\cdot)$ the frequency of a given value in a marginal or joint distribution, $[\cdot]_d$ a type cast to floating point, and the remaining parameters have the same meaning as in Eq. 2.

Finally, the equation for the reduced-precision version of the MI would be:

$$\begin{aligned}\hat{I}(X;Y) = &\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y) \\ &\cdot[\mathcal{L}_{f(x,y)} - \mathcal{L}_{f(x)} - \mathcal{L}_{f(y)}]_d \cdot q.\end{aligned} \tag{10}$$

4 CUDA implementation

The general structure of the BetaGamma family and the ICAP method was explained in Sect. 3. However, developing an implementation that works on GPUs is not straightforward, and the algorithms need to be adapted. In addition to massive parallelism, GPUs offer other features that can be exploited to improve runtimes. In this section, we will gradually explain the included optimizations, starting with how the original code needs to be reorganized and ending with the fully optimized version. A 16-bit fixed-point version is also proposed.

4.1 High-precision version

Algorithm 1 illustrates that BetaGamma and ICAP share a common structure with two nested for loops. The outer loop (line 7) is used to iteratively select features. Each iteration uses the current subset of selected features, and adds a new one to it. Due to this dependency, this loop cannot run in parallel. In the inner loop (line 8), the scores of all features are computed. The computation of the score of each feature is independent, so this loop is best suited for parallelization.

We will restructure the code in several ways to make it more suitable for GPU execution. First, line 10 can be rewritten as $\text{cmi} \leftarrow H(X_i|C) - H(X_i|\text{YC})$ (see Eq. 4). This change splits the CMI into two terms. The first one, $H(X_i|C)$, is constant for each feature X_i in the dataset, so it can be computed once and cached. The same approach cannot be applied to the second term, $H(X_i|\text{YC})$, because it depends on the last selected feature (Y), which changes between iterations of the outer loop (line 7). The second term calculates the entropy of X_i conditioned by YC. YC represents the joint state of feature Y and the class C, and it must be computed before the CH. Since it is constant within the inner loop (line 8), it can also be extracted from it and computed only once per selected feature.

The pseudocode after these changes is shown in Algorithm 2. The entropy of each feature X_i conditioned by the class C (the first term) is precomputed in line 5 and stored in the cch data structure, which is later used in line 13 for the CMI. The joint state of Y and C is computed in line 10 and used in line 13 for the entropy of X_i conditioned by YC (the second term). Note that YC has been renamed to J for simplicity.

Input: dataset D, dataset class C, total number of features F, number of features to select k
Output: subset of selected features S

1: $scores$: array$[F]$
2: cch : array$[F]$
3: **for** feature $X_i \in D$ **do**
4: $scores_i \leftarrow I(X_i; C)$
5: $cch_i \leftarrow H(X_i|C)$
6: **end for**
7: $Y \leftarrow X_{\text{argmax}(scores)}$
8: $S \leftarrow S \cup \{Y\}$

9: **while** $|S| < k$ **do**
10: $J \leftarrow \text{merge}(Y, C)$
11: **for** feature $X_i \in D \setminus S$ **do**
12: $mi \leftarrow I(Y; X_i)$
13: $cmi \leftarrow cch_i - H(X_i|J)$
14: $scores_i \leftarrow scores_i - fn(mi, cmi)$
15: **end for**
16: $Y \leftarrow X_{\text{argmax}(scores)}$
17: $S \leftarrow S \cup \{Y\}$
18: **end while**

Algorithm 2 Common structure of BetaGamma and ICAP, precomputing $H(X_i|C)$ and $J = \text{YC}$

The algorithm can be further restructured to better adapt to GPUs. A good approach to offloading work to the GPU is to group similar computations together so that different threads can perform these computations on different data simultaneously. In our case, we split the inner loop (line 11 in Algorithm 2) into four loops: computing the MI, computing the CH, computing the CMI (with the CH), and finally computing the scores (with the MI and CMI). The resulting pseudocode is shown in Algorithm 3. Two new functions were created for the most computationally expensive steps:

- $\text{MI}_{\text{all}} : (D, Y) \rightarrow N$, which takes a subset D and a feature (or class) Y and returns a 1D array of dimension $|D|$ with the $\text{I}(X; Y)$ for all $X \in D$.
- $\text{CH}_{\text{all}} : (D, Y) \rightarrow N$, which takes a subset D and a feature (or joint state, or class) Y and returns a 1D array of dimension $|D|$ with the $H(X|Y)$ for all $X \in D$.

Now, the outer loop (line 5 in Algorithm 3) is structured in a task-based manner. First, the MI is calculated and stored in the mi array (line 6). Then, the CMI is computed and stored in cmi (lines 7 to 12) using the following intermediate steps: the calculation of the joint state of Y and C in line 8, the computation of the CH for all features in line 9, and the computation of the CMI as a difference of entropies from line 10 to 12. Finally, the MI and CMI are used to compute the scores (lines 13–15). Note that MI_{all} and CH_{all} are also used to compute the scores for the first time (line 1) and the cached entropy of each feature conditioned by the class (line 2), respectively.

Input: dataset D, dataset class C, total number of features F, number of features to select k
Output: subset of selected features S

1: $scores \leftarrow MI_{all}(D, C)$
2: $cch \leftarrow CH_{all}(D, C)$
3: $Y \leftarrow X_{\text{argmax}(scores)}$
4: $S \leftarrow S \cup \{Y\}$

5: **while** $|S| < k$ **do**
6: $mi \leftarrow MI_{all}(D \setminus S, Y)$
7: cmi : array$[F]$
8: $J \leftarrow \text{merge}(Y, C)$
9: $ch \leftarrow CH_{all}(D \setminus S, J)$
10: **for** feature $X_i \in D \setminus S$ **do**
11: $cmi_i \leftarrow cch_i - ch_i$
12: **end for**
13: **for** feature $X_i \in D \setminus S$ **do**
14: $scores_i \leftarrow scores_i - fn(mi_i, cmi_i)$
15: **end for**
16: $Y \leftarrow X_{\text{argmax}(scores)}$
17: $S \leftarrow S \cup \{Y\}$
18: **end while**

Algorithm 3 Common structure of the restructured BetaGamma and ICAP

With this structure of the algorithm, the most computationally expensive steps (computation of MI and CMI) are isolated and better structured to take advantage of GPU parallelism. Furthermore, we can use the GPU-optimized version of MI_{all} implemented for our previous work [3] and focus on developing a GPU-optimized version of CH_{all}.

4.1.1 Base CUDA version

The novel implementation of CH_{all} consists of two parts: a CUDA kernel that computes the CH and a CPU function responsible for managing the workload, preparing the data, and launching the GPU computation.

Memory management is a key challenge in this implementation, since computing the CH for a feature X_i requires a 2D data structure of dimension $|\mathcal{X}_i| \times |\mathcal{J}|$, where $\mathcal{X}_i$ and $\mathcal{J}$ are the alphabets of X_i and J, respectively. The CH is computed in the GPU, and since datasets can contain many features, the total memory required for all features X_i can easily exceed the available GPU capacity. One option would be to process each feature in a separate kernel, but this would introduce significant overhead due to frequent

memory operations and synchronizations. Instead, we split the workload into batches so that multiple features are processed together in a single kernel launch.

The aforementioned 2D data structure is needed to store the joint probabilities of X_i and J (see Eq. 2). Note that a 1D data structure is also required for the marginal probabilities of J. However, the CH of all features X_i is conditioned by the same J, so only one 1D structure is needed for all $\mathrm{CH}_{\mathrm{all}}$ executions.

To reduce memory consumption, the probabilities are computed in a lazy way. That is, instead of storing probabilities directly, we compute and store frequencies (i.e., the data structures are histograms). This allows the use of `uint16` (16-bit unsigned integers) to represent the data, instead of the 64-bit floating-point representation that would be needed to store the probabilities.

Probabilities are then calculated on demand by dividing the frequencies by the total number of samples N. Although this is equivalent to Eq. 2, we can reformulate it to reflect these changes as:

$$H(X|Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} \frac{f_{\mathrm{xy}}}{N}\log_2\frac{f_{\mathrm{xy}}/N}{f_y/N} \tag{11}$$

where f_{xy} is the frequency of both $X = x$ and $Y = y$, and f_y is the frequency of $Y = y$. Remember that this equation is used to compute $H(X_i|J)$ for all X_i in the dataset. That is, the 1D histogram (henceforth F_J) contains the frequencies of J, and the 2D histogram (henceforth F_{JX_i}) contains the joint frequencies of X_i and J.

To further optimize memory usage, a custom memory management system based on pools was implemented. Instead of allocating and deallocating memory for the 2D histograms of each batch, a large memory pool is preallocated at the beginning of the execution. This is possible because we know the number of features per batch and we can compute the maximum bound for the size of a single 2D histogram. The pool size is calculated as:

$$\begin{aligned} max\_hsize &= \max_{X_i\in D}\{|\mathcal{X}_i|\}\cdot|\mathcal{J}|\cdot \mathrm{d}_{\mathrm{size}} \\ pool\_size &= B\cdot max\_hsize \end{aligned} \tag{12}$$

where B is the number of features per batch, $\mathcal{X}_i$ and $\mathcal{J}$ are the alphabets of X_i and J, D is the dataset, and $\mathrm{d}_{\mathrm{size}}$ is the size of the datatype used to store the frequencies (two bytes for a `uint16`). Later, before launching a kernel, the CPU computes the exact sizes of the 2D histograms for the whole batch and assigns a memory region from the pool to each one. This approach allows us to reduce the overhead associated with frequent allocations and deallocations. Note that the number of features per batch can heavily impact performance, so it is exposed as a configurable parameter that allows tuning based on dataset characteristics and available GPU memory.

While the preallocation of the pool is based on the maximum size of the joint histogram, in practice many features require less memory, so some space in the pool may remain unused. To mitigate this and minimize the number of kernel launches, the CPU will add more features to the batch as long as there is enough space in the pool to fit their 2D histograms.

We have already mentioned that, in addition to the joint frequencies (2D histogram), the CH also needs the marginal frequencies of J. This is represented as a 1D histogram, which is constant for the computation of the CH for all X_i. Therefore, in our implementation, it is computed by the CPU at the beginning and then transferred to the GPU, with negligible impact on either performance or memory consumption.

Figure 1 summarizes the behavior of the $\mathrm{CH}_{\mathrm{all}}$ function from the CPU point of view, including the optimizations described above.

We can now move on to the GPU side. After the CPU prepares a batch, the kernel is launched to compute the CH for its features. It is launched using a grid of B blocks, where B is the number of features in the batch. That is, a single CUDA thread block will be in charge of computing the CH for a single feature X_i.

As explained earlier, two histograms are needed to compute the CH: the marginal frequencies of J (F_J) and the joint frequencies of J and X_i (F_{JX_i}). F_J is already precomputed and transferred to the GPU, so the kernel only needs to compute F_{JX_i}. Therefore, each thread block performs two steps: the creation of F_{JX_i}, and the computation of $H(X_i|J)$ using F_J and F_{JX_i}.

Both steps are distributed among the threads in the block, as shown in Fig. 2. Specifically, the workload of each thread block is distributed among its T threads as follows:

1. **Creation of the joint frequencies.** Each thread reads a pair of values from the same position p of J and X_i (i.e., $j = J[p]$ and $x = X_i[p]$). This means that there exists an occurrence of (j,x), and thus the thread increments the element (x, j) at F_{JX_i} ($F_{\mathrm{JX}_i}[x][j] \leftarrow F_{\mathrm{JX}_i}[x][j] + 1$). The workload is cyclically distributed among the T threads, so that each thread t is responsible for reading all positions $p = t + kT,\ k = 0, 1, 2, \ldots,\ p < N$, where N is the number of samples of both J and X_i. This distribution ensures coalesced memory accesses when reading J and X_i. Note that the updates are performed on random positions of F_{JX_i}, so the increments must be atomic to avoid data races between concurrent threads. This first step is shown in Fig. 2a.

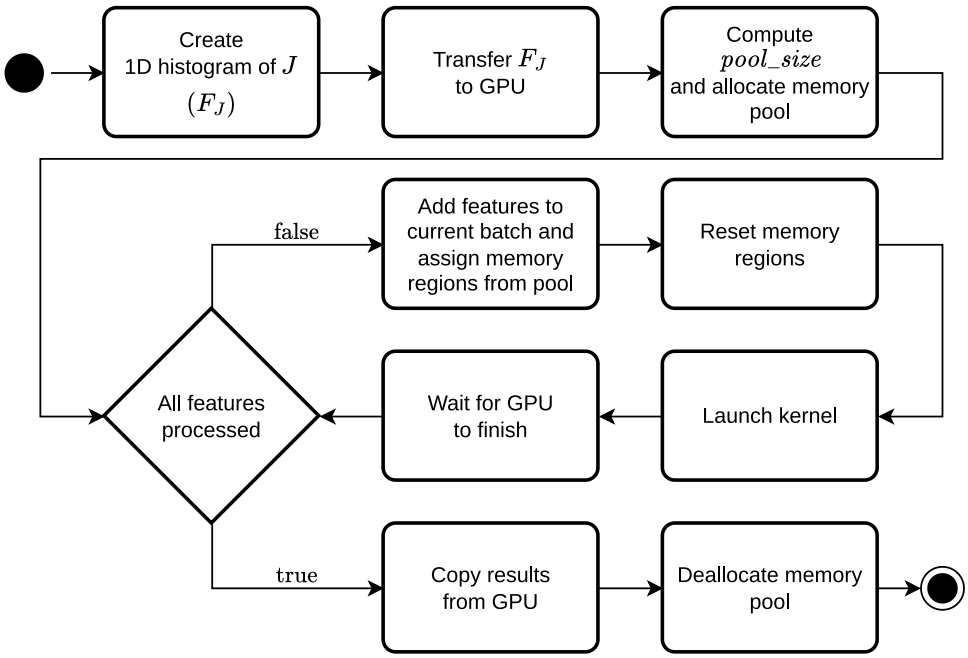


Fig. 1 High-level view of the $\mathrm{CH}_{\mathrm{all}}$ function

2. **Synchronization.** The histogram will be read in the next step, so we need to make sure that it has been completely created (i.e., all threads have finished the previous step) before proceeding. The CUDA function `__syncthreads()` is used to synchronize all the threads in the block.
3. **Computation of the CH.** To calculate the CH, each joint frequency f_{jx} (stored in F_{JX_i}) must be combined with the corresponding marginal frequency f_j (stored in F_J), as shown in Eq. 11. The CH is calculated as the sum of the results of these operations. In our approach, we distribute the workload by columns of F_{JX_i}, as shown in Fig. 2b. To process a column q, the thread first reads $f_j = F_J[q]$ and stores it in a private register. It then iterates over the column $F_{\mathrm{JX}_i}[*][q]$, reading the values f_{jx} and combining them with f_j. The columns are cyclically distributed to achieve an efficient parallel access pattern. The workload is distributed so that each thread t processes one or more columns and accumulates a partial result CH_t. At the end, the partial results of all threads are summed to obtain the CH.

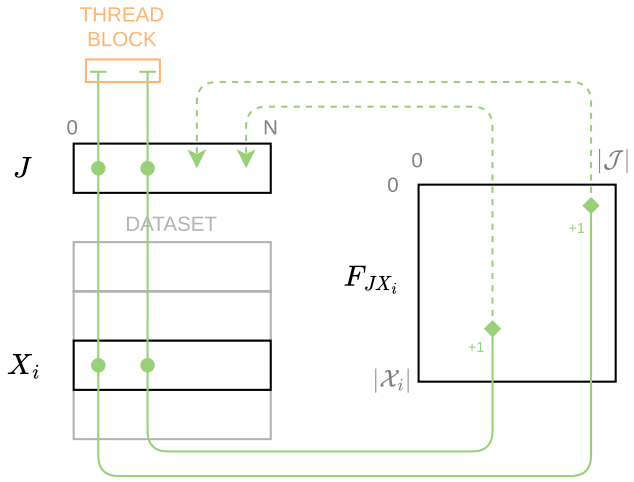
Note that this approach works for blocks with any number of threads. The block size is exposed to the user as a tunable parameter because its configuration can affect performance.

4.1.2 Shared memory

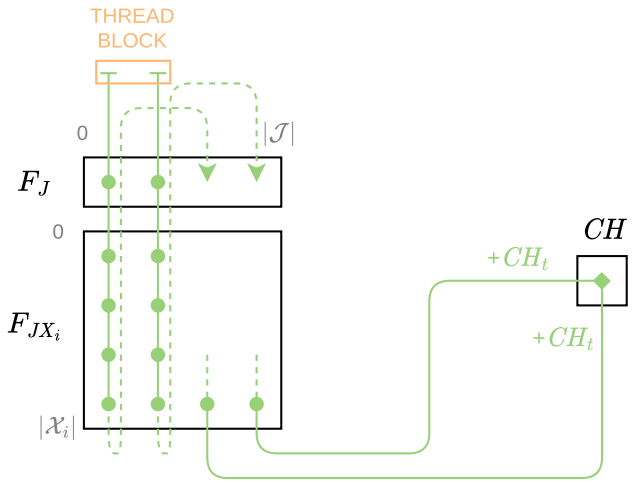
As explained earlier, each CUDA block computes the entropy of a single feature X_i conditioned by J. This is a two-step process that involves computing a joint histogram for both features (F_{JX_i}) and then accessing all its positions to compute the CH. The shared memory available to each thread block can be used to speed up all memory accesses to the joint histograms, as it provides higher bandwidth than the global GPU memory. The main limitation is the small size of the shared memory, which ranges from tens to hundreds of kiB on modern GPUs (MAX_SHR_SM bytes from now on). The dimensions of the joint histograms depend on the alphabet sizes of the features involved (i.e., X_i and J). This optimization should be valid for different alphabets and GPU architectures (with different shared memory sizes).

An initial approach was to fit part of the histogram in shared memory while keeping the rest in global memory. However, early experiments showed that this approach was inefficient because the overhead introduced outweighed the performance gains. Our final solution is simpler and integrates well with the custom memory management system. It works as follows:

1. When the CPU prepares a batch, it calculates the size of the joint histogram (hsize_i) for each feature X_i in the batch. If the histogram fits in shared memory ($\mathrm{hsize}_i \leq MAX\_SHR\_SM$), it will be stored there. This is indicated to the kernel by passing a `nullptr` for the histogram of that feature. If it does not fit, the CPU assigns a block from the memory pool and passes its address to the kernel. In addition, the CPU keeps track of the maximum amount of shared memory required by any feature in the batch (shr_max bytes, i.e., $shr\_max = \max\{\mathrm{hsize}_i\}$).



(a) Creation of the joint frequencies



(b) Computation of the CH

Fig. 2 Steps performed by the GPU kernel to compute $H(X_i|J)$

2. The CPU issues a `memset` operation to reset all the global memory used by the features in that batch.
3. The CPU launches the kernel for that batch, with one block per feature and shr_max bytes of dynamic shared memory per block.
4. Inside the kernel, the GPU checks the pointer. If it points to a global address, that space (which we are sure that is reset to zero) is used for the histogram. Otherwise, if the pointer contains a `nullptr`, shared memory is used. In this case, shared memory is explicitly reset by the threads in the block before the histogram is created.

Creating the joint histograms is a task that involves random memory writes. Since it is performed in parallel in our kernel, atomic operations are required to avoid data races. Empirical experiments show that the benefits of using shared memory depend on the dataset characteristics, and in some cases it does not provide a clear performance advantage. Therefore, the implementation included in cuFEAST allows users to specify whether or not to use shared memory in the kernels for $\mathrm{CH}_{\mathrm{all}}$.

4.1.3 Asynchronous computing

A batch processing that is fully synchronous means that the CPU prepares a batch, resets global memory, launches the kernel, and then waits for the GPU to finish before moving on to the next batch. This results in idle time for both the CPU and the GPU: while the GPU is running the kernel, the CPU remains idle, and while the CPU is preparing the next batch and resetting memory, the GPU is not being used.

To improve resource utilization and overlap computation with batch preparation, asynchronous execution using multiple CUDA streams is introduced. Before processing begins, S streams are created, which are then reused cyclically. That is, batch b is assigned to stream $s = b \mod S$, allowing the CPU to prepare batch $b + 1$ while the GPU is still processing batch b.

However, introducing asynchronism comes with additional memory requirements, as some memory structures must be replicated for each stream to prevent one stream from overwriting the data of another before execution is complete. While most data structures have a negligible footprint, the memory pools must also be replicated S times. In extreme cases (e.g., when using large batches), this can exceed the available GPU memory.

The number of streams is also exposed as a configurable parameter to address this balance between asynchronism and memory consumption. Choosing an appropriate tradeoff between batch size and the number of streams is critical, as using small batches with many streams can introduce overhead from frequent kernel launches and memory operations, potentially diminishing performance gains.

4.2 Reduced-precision version

As explained in Sect. 3.3, using a low-precision fixed-point representation to compute information metrics reduces the computational cost of frequent operations such as double-precision arithmetic and logarithms. However, implementing this approach requires addressing three key issues: selecting an appropriate data representation, determining bit-width precision, and deciding when to compute the LUT.

For data representation, CUDA does not provide native support for fixed-point arithmetic, so values must be stored in an alternative encoding. We adopt a 16-bit fixed-point representation, following the results of [24, 25], which show that this precision is sufficient for the FS algorithm to select feature subsets of the same quality as the double-precision

Table 1 Characteristics of the systems used for the experiments

System	Pluton
CPU	2×Intel Xeon Silver 4216 (Cascade Lake-SP)
Cores/CPU	16
Total cores	32
RAM	256 GiB
GPU	NVIDIA T4
GPU micro-architecture	Turing
CPUs TDP	2×100 W
GPU TDP	70 W
System	Finisterrae III (FT3)
CPU	2×Intel Xeon Platinum 8352Y (Ice Lake-SP)
Cores/CPU	32
Total cores	64
RAM	256 GiB
GPU	NVIDIA A100
GPU micro-architecture	Ampere
CPUs TDP	2×205 W
GPU TDP	250 W

Table 2 Characteristics of the datasets used for the experiments

Dataset	#Features	#Samples	#Classes
MNIST	780	60,000	10
Epsilon	2,000	400,000	2
RCV1	47,236	20,242	2
News20	62,061	15,935	20

Table 3 Parameters and their tested values used in the performance experiments

Scenario parameters	Tested values
System	Pluton, FT3
Algorithm	BetaGamma, ICAP
Dataset	MNIST, Epsilon, RCV1, News20
Precision	double, reduced
Performance parameters	Tested values
Block size (#threads)	64, 128, 256, 512, 1024
Batch size (#features)	64, 256, 512, 1024, 4096
Async. (#streams)	0, 1, 2, 4, 8
Shared memory	Disabled, enabled

counterpart. Instead of using a fixed allocation of bits between the integer and fractional parts, a dynamic bit distribution strategy that adapts to each dataset is applied. This approach calculates the largest magnitude that needs to be represented based on the number of samples of the dataset and uses as b_i the number of bits required to represent this maximum value, and b_f for the rest. Thus, the dynamic approach always maximizes the number of fractional bits (b_f) to improve precision while ensuring that the value with the largest magnitude can be stored in the 16-bit fixed-point format.

Finally, the LUT is created and transferred to the global memory of the GPU only once at the beginning of the whole process. The computational cost of generating the LUT is negligible compared to the total FS runtime in real-world scenarios.

5 Experimental evaluation

The performance of the proposed CUDA implementations has been assessed by running several experimental tests with different input datasets on two systems (Pluton and Finisterrae III, abbreviated as FT3) with different GPU architectures (Turing and Ampere, respectively). The characteristics of the systems are shown in Table 1. Besides the performance, the correctness of the GPU implementations has been validated by comparing their results with those produced by the sequential algorithms of the FEAST library.

The runtime of both ICAP and BetaGamma methods is tied to the input data, so we need to measure their performance using datasets with different characteristics, which are shown in Table 2. These datasets are publicly available in the LIBSVM [26] collection[2]. Two of them, MNIST and Epsilon, have more samples than features, while RCV1 and News20 have more features than samples. It is also worth noting the number of classes, as this can also affect performance. MNIST and RCV1 are biclass datasets, while Epsilon and News20 are multiclass. All of these datasets contain continuous data that must be discretized before running the FS with BetaGamma or ICAP. Although tests with 128 and 256 bins have been performed, the results in this section are only shown for 128 bins for simplicity, as the conclusions that can be drawn from both cases are similar.

As explained throughout Sect. 4, there are four parameters (block size, number of features per batch, number of streams, and whether or not to use shared memory) that can be configured by the user and affect performance. Over 40,000 experiments have been run to find the configuration that works best. This number of executions results from the combinations of the parameters shown in Table 3.

In addition, the execution of each configuration was repeated several times to discard outliers. Specifically, each configuration was run five times, from which the minimum and maximum were discarded (to avoid the influence of

[2] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

outliers), and the results shown in this section are the average of the three central values.

Note that the BetaGamma algorithm, in particular, uses two parameters (β and γ) to balance the weight of the MI and CMI for the computation of the scores. In our experiments, BetaGamma was configured with $\beta = 1.0$ and $\gamma = 1.0$. These parameters do not have a significant impact on performance, i.e., they do not directly affect the execution time, but only the numerical results.

In all experiments, 200 features were selected because this number is low enough to be used in a real case, but high enough to draw meaningful conclusions from the runtimes. Also, all runtimes (and derived metrics) shown include data transfers to and from the GPU.

In the following subsections (Sects. 5.1 and 5.2), we first analyze the benefits of each optimization separately. Then, two additional subsections (Sects. 5.3 and 5.4) are dedicated to finding out which kernel configuration works best, and to comparing our CUDA versions to the multicore state of the art, respectively. Finally, the main findings of the experimental evaluation are summarized in Sect. 5.5.

5.1 High-precision version

Our first CUDA implementation of the algorithms included a basic parallelization and division of the workload in batches (see Sect. 4.1). The runtimes are shown in Table 4 by algorithm, GPU, and dataset. Note that each cell corresponds to the best runtime for the block and batch size configurations tested (see Table 3).

Our next proposed optimization was the use of shared memory with the adaptive procedure explained in Sect. 4.1.2. Table 5 shows the best runtimes with shared memory enabled, as well as the relative runtime difference from the previous best configurations.

Table 4 Best runtimes (in seconds) for the base version of CUDA

Algorithm	GPU	Dataset			
		MNIST	Epsilon	RCV1	News20
BetaGamma	T4	5.73	31.65	58.26	55.44
	A100	2.55	10.59	20.10	20.53
ICAP	T4	5.73	30.46	61.84	58.25
	A100	2.49	11.70	26.32	23.66

The results show that the runtime reduction achieved by using shared memory reaches up to 30% for T4 and 41% for A100. We can also see that the FS for the RCV1 dataset is slower when using the T4 GPU for both algorithms. The reason is that RCV1 is the dataset that requires the most shared memory (on average 32 kiB for MI and 35 kiB for CH), which can limit the number of blocks per streaming multiprocessor (SM). This limitation, which is less severe for the other datasets, is outweighed by the reduction of memory access latency. Regarding the GPU, the T4 is restricted to one block per SM, while the A100 allows up to four blocks per SM. Therefore, the A100 can achieve a higher level of parallelism when using shared memory.

The final optimization was to create asynchronous streams to overlap CPU and GPU computations. The best runtimes achieved with asynchronism are shown in Table 6. We see improvements of up to 6% and 30% for the T4 and A100, respectively, over the best synchronous results (best runtimes between Tables 4 and 5). The results also show that asynchronism is not beneficial for the RCV1 dataset on the T4, but in this case the increase in runtime is negligible.

Note that the runtime improvements shown in Table 6 are due not only to asynchronism, but also to the interplay between optimizations. For example, using larger batches imposes an overhead on memory operations, while smaller batches increase the overhead of launching kernels. This latter overhead can be hidden with asynchronism, so that

Table 5 Best runtimes (in seconds) for the CUDA version with adaptive shared memory and relative runtime differences from the best results of the version using global memory (base version)

Algorithm	GPU	Dataset			
		MNIST	Epsilon	RCV1	News20
BetaGamma	T4	5.53 (−4%)	22.06 (−30%)	63.71 (+9%)	40.32 (−27%)
	A100	2.33 (−8%)	6.27 (−41%)	12.93 (−36%)	14.00 (−32%)
ICAP	T4	5.40 (−6%)	21.58 (−29%)	66.99 (+8%)	43.46 (−25%)
	A100	2.24 (−10%)	7.00 (−40%)	18.43 (−30%)	17.60 (−26%)

Table 6 Best runtimes (in seconds) for the CUDA version with asynchronism and relative runtime differences from the best synchronous results

Algorithm	GPU	Dataset			
		MNIST	Epsilon	RCV1	News20
BetaGamma	T4	5.52 (0%)	21.72 (−2%)	58.28 (0%)	37.71 (−6%)
	A100	2.36 (+1%)	5.90 (−6%)	9.05 (−30%)	10.15 (−27%)
ICAP	T4	5.40 (0%)	21.27 (−1%)	61.87 (0%)	40.78 (−6%)
	A100	2.24 (0%)	6.62 (−5%)	14.35 (−22%)	14.12 (−20%)

Table 7 Bit distribution for the 16-bit fixed-point version

Dataset	#Samples	Bits	
		b_i	b_f
MNIST	60,000	4	12
Epsilon	400,000	5	11
RCV1	20,242	4	12
News20	15,935	4	12

configurations that were previously slower can now stand out.

5.2 Reduced-precision version

Although there are works that have applied a low-precision fixed-point procedure to CMI, its accuracy has been measured in a Markov blanket context [25]. In this section, we will analyze how this approach affects our GPU versions of BetaGamma and ICAP, and also provide some insight into the accuracy of the procedure when applied to the pure FS methods.

As explained in Sect. 4.2, our implementation uses a datatype of 16 bits, which are automatically distributed between the integer and fractional parts (see Sect. 3.3). Table 7 shows the bit distribution for each dataset (b_i bits for the integer part and b_f for the fractional part).

5.2.1 Accuracy

The impact of using low-precision fixed-point data on the accuracy of FS has previously been tested for MI [24] or CMI [25], but not for both. Since BetaGamma and ICAP mix MI with CMI for the calculation of the scores, we needed to be sure that the accuracy of the scores would be maintained when using low-precision fixed-point data. Note that only the accuracy of the fixed-point version with 16 bits was tested. Although it might be interesting to understand the effects of using different numbers of bits, it would be beyond the scope of our work, which is focused on exploiting GPUs for FS acceleration.

These FS methods provide a ranking of k features with their scores. In order to measure accuracy, a "true" ranking must be established beforehand so that it can be used as a reference point for comparisons. In our case, the output of the original algorithms from the FEAST library was used as a reference, since they are widely used by the scientific community and can be considered as "ground truth". Two metrics are then used to measure accuracy:

- **True Positive Rate** (TPR). In this context, the output of the FS is a set, so we do not consider the order. TPR is calculated as follows:

$$\mathrm{TPR} = 1 - \frac{|S_{\mathrm{ref}} \setminus S|}{k}$$

where k is the number of features selected, S_{ref} is the "true" output subset, and S is the output subset of our implementation. The term $|S_{\mathrm{ref}} \setminus S|$ would be the number of incorrectly selected features.

- **Mean Squared Error** (MSE). With this metric we try to obtain the deviation of the scores compared to the reference version. The MSE is calculated as:

$$\mathrm{MSE} = \frac{1}{k}\sum_{i=1}^{k}(\mathrm{score}_{\mathrm{ref}}[i] - \mathrm{score}[i])^2$$

where k is the number of features selected, and $\mathrm{score}_{\mathrm{ref}}$ and score are the scores obtained by the reference implementation and our reduced-precision version, respectively.
Table 8 shows both TPR and MSE for each algorithm and dataset when 5, 10, and 20 features are selected. As expected, MSE increases as k increases because the loss of precision in the computations is accumulated in the calculation of the scores. However, the MSE values are extremely low in all scenarios. The TPR is 1.0 in most cases, which is similar to the results in [24] when using 16 bits. Therefore, we can

Table 8 TPR and MSE of both algorithms using the 16-bit fixed-point computation of MI and CMI, and varying the number of features selected (k)

Algorithm	Dataset	$k = 5$		$k = 10$		$k = 20$	
		TPR	MSE	TPR	MSE	TPR	MSE
BetaGamma	MNIST	1.00	6.62e−9	1.00	1.41e−8	1.00	6.65e−8
	Epsilon	1.00	2.49e−9	0.90	9.02e−9	0.95	7.83e−8
β=1,γ=1	RCV1	1.00	6.97e−9	1.00	1.36e−7	1.00	8.46e−7
	News20	1.00	1.34e−8	1.00	1.17e−7	1.00	4.56e−7
ICAP	MNIST	1.00	3.43e−10	1.00	5.63e−10	0.95	1.06e−9
	Epsilon	1.00	4.64e−11	1.00	3.34e−10	1.00	4.25e−10
	RCV1	1.00	9.90e−9	1.00	6.86e−9	1.00	1.57e−8
	News20	1.00	6.70e−9	1.00	1.53e−8	1.00	2.19e−8

also conclude that 16 bits are sufficient for the fixed-point version to select the same features as the 64-bit floating-point version in the BetaGamma and ICAP methods.

As a final thought, note that the versions of the algorithms that work with double-precision floating-point data produce exactly the same results as their original sequential counterparts, so the accuracy is also the same and does not need to be measured again.

5.2.2 Performance

Using 16-bit integer datatypes instead of double floating-point arithmetic greatly reduces the computational cost of MI and CMI calculations. However, applying this optimization requires more than just changing the datatype (e.g., adding random memory accesses and type casts), so the overall impact on performance cannot be estimated and must be measured.

Table 9 shows the best runtimes achieved by the 16-bit fixed-point version for each algorithm and dataset. The runtime difference relative to the best double floating-point results (mainly those in Table 6) is also shown. Note that these runtimes already include the time to compute the LUT and are for the best configuration of block and batch size, number of streams, and shared memory usage.

The results show that the 16-bit fixed-point version significantly reduces runtimes, up to 49% and 23% on the T4 and A100 GPUs, respectively.

5.3 Kernel configuration

As seen throughout Sect. 4.1 and summarized in Table 3, our CUDA implementation exposes three configuration parameters that have a direct impact on performance (block size, batch size, and asynchronism), so finding the best combination would be of great interest. However, the best performing configuration may depend on the actual scenario (system, algorithm, dataset, and precision). Therefore, we normalize the runtime to search for this configuration. Users can also disable the use of shared memory, but since it is beneficial in most cases, we assume that it is always enabled. The procedure is as follows:

1. Runtimes are grouped by the parameters that are external to the configuration (i.e., system, algorithm, dataset, and precision). Note that precision is considered as an external parameter because using the reduced-precision version may not always be appropriate.
2. The runtimes are normalized within each group as:

$$\hat{t_{\mathrm{cfg}}} = \frac{t_{\mathrm{cfg}} - \bar{t}}{\sigma}$$

where t_{cfg} is the runtime of a given configuration cfg, and $\bar{t}$ and σ are the mean runtime and its standard deviation within the group, respectively. This value can now be interpreted as a measure of how well each configuration performs within its group (i.e., independently of the external parameters) and can be used for comparison purposes.

The normalized runtimes are shown in Fig. 3 for block size, batch size, and asynchronism. Since the impact of the different parameters can vary depending on the GPU microarchitecture, we have also separated the plots by system.

The first conclusion that can be drawn is that faster executions are obtained when using more threads per block in both architectures. However, the plots for the batch and the streams should be interpreted with caution, as these two parameters are highly interdependent. For example, the distribution for eight streams shows that the runtimes are skewed to higher values. This is due to the fact that a large number of streams is very inefficient when the batch size is also large, since the amount of memory that needs to be allocated is huge. Therefore, when eight streams are combined with batches larger than 64 or 128, both optimizations cancel each other out.

The recommended parameter configurations, based on the combinations that performed best on average across all datasets, are shown in Table 10. The use of shared memory showed an improvement in all of these configurations, so it is always convenient to enable it. We also recommend using the reduced-precision version, although the user should check whether the loss of accuracy is acceptable (see Sect. 5.2.1). For this reason, the recommended configurations for the double-precision version are also shown. Note that the parameters of these configurations are meant to be

Table 9 Best runtimes (in seconds) for the reduced-precision version and relative runtime differences from the best runtimes of the double-precision floating-point version

Algorithm	GPU	Dataset			
		MNIST	Epsilon	RCV1	News20
BetaGamma	T4	2.83 (−49%)	16.53 (−24%)	33.07 (−46%)	33.46 (−11%)
	A100	1.92 (−18%)	5.79 (−2%)	7.80 (−14%)	9.42 (−7%)
ICAP	T4	2.77 (−49%)	16.50 (−22%)	35.19 (−46%)	35.28 (−13%)
	A100	1.87 (−16%)	6.40 (−3%)	10.99 (−23%)	12.72 (−10%)

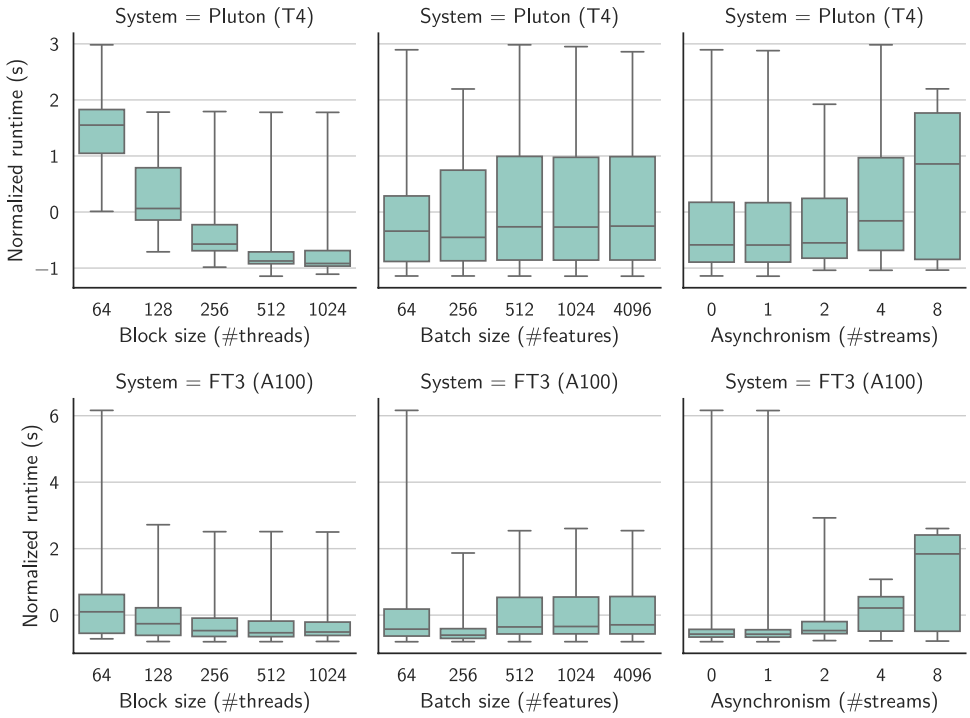


Fig. 3 Distribution of normalized runtimes for different block sizes, batch sizes, and number of streams. Each graph shows how variation in the X-axis parameter affects the distribution of runtimes in general, independently of the values of the other parameters

Table 10 Recommended configurations for the CUDA versions of BetaGamma and ICAP algorithms on different GPU architectures

GPU	Algorithm	Precision	Block size (#threads)	Batch size (#features)	Async. (#streams)	Shared memory
T4	Both	Double	512	4096	1	Enabled
		Reduced	1024	512	1	
A100	Both	Double	512	256	2	Enabled
		Reduced	512	256	1	

used altogether, as they were the best performing ones on average, and are not based on values that work well in isolation (e.g., as explained earlier, a batch size of 4096 features is not a good "overall" configuration, although it can achieve the best performance with the appropriate block size and asynchronism).

5.4 Comparison with the state of the art

We have already explored the effectiveness of each optimization when applied incrementally to our CUDA versions of BetaGamma and ICAP. This subsection compares our results with the state of the art, in terms of both performance and energy consumption. To the best of our knowledge, the only implementations of BetaGamma and ICAP are provided by the FEAST and Parallel-FST[3] libraries. On the one hand, FEAST is the library on which both cuFEAST and Parallel-FST are based. Its implementations are sequential and have been used to validate that our double-precision GPU versions provide the expected results. On the other hand, Parallel-FST contains multithreaded implementations targeted at multicore CPUs.

5.4.1 Performance

The three libraries implementing BetaGamma and ICAP were run on both Pluton and FT3 (see Table 1). The runtimes are shown in Table 11. In the worst-case scenario, the T4 GPU requires one minute to complete the FS procedure, while the A100 always needs less than 15 seconds. This represents a remarkable runtime reduction compared to the FEAST methods, which can require even more than 50 minutes.

In Pluton, the multicore implementations were run with 32 threads and achieved an average efficiency of $52\% \pm 6\%$. In FT3, the average efficiency was $86\% \pm 6\%$ using 64 threads. Figure 4 shows the speedups of the BetaGamma and ICAP algorithms from cuFEAST (both the double- and reduced-precision versions) over their multithreaded counterparts from Parallel-FST. Note that reduced-precision implementations are not available in Parallel-FST, so the double-precision versions were used.

[3] https://gitlab.com/bieito/parallel-fst/-/tree/master/mpi-feast

Table 11 Runtimes (in seconds) for the BetaGamma and ICAP implementations in FEAST, Parallel-FST (P-FST), and cuFEAST to select 200 features from each dataset. The three libraries used double precision and produced the same results

Algorithm	Dataset	Pluton			FT3		
		FEAST	P-FST	cuFEAST	FEAST	P-FST	cuFEAST
BetaGamma	MNIST	183.70	12.65	5.52	143.34	3.55	2.33
	Epsilon	2379.33	152.87	21.72	1924.58	43.07	5.90
	RCV1	2960.34	159.86	58.26	2981.07	49.71	9.05
	News20	3159.62	171.69	37.71	3248.76	56.36	10.15
ICAP	MNIST	182.38	12.89	5.40	144.66	3.73	2.24
	Epsilon	2342.92	151.35	21.27	1918.52	42.93	6.62
	RCV1	2964.22	159.89	61.84	2986.50	50.07	14.35
	News20	3176.75	171.87	40.78	3262.39	57.32	14.12

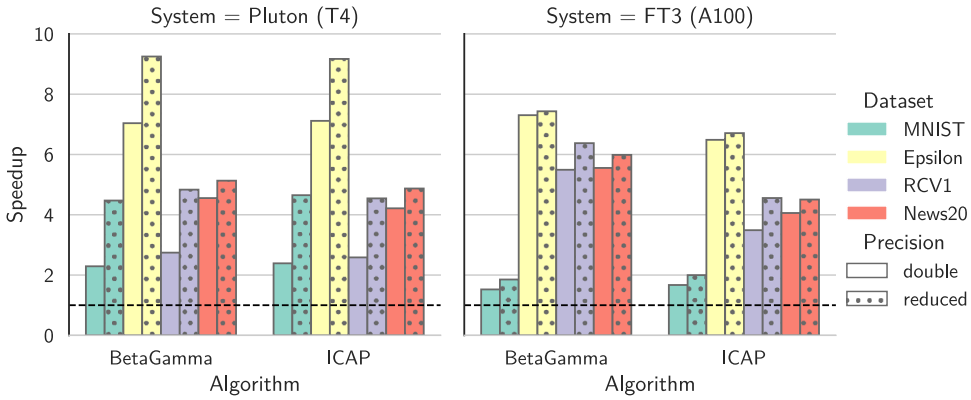


Fig. 4 Best speedups achieved by the GPU versions of BetaGamma and ICAP relative to the multi-threaded double-precision versions of Parallel-FST; a dashed line is used to indicate a 1x speedup

Table 12 Estimated energy consumption (W h) per dataset, system, and implementation, averaged over BetaGamma and ICAP

Dataset	Pluton			FT3		
	Parallel-FST	cuFEAST		Parallel-FST	cuFEAST	
	Double	Double	Reduced	Double	Double	Reduced
MNIST	0.71	0.11	0.05	0.41	0.16	0.13
Epsilon	8.45	0.42	0.32	4.90	0.43	0.42
RCV1	8.88	1.17	0.66	5.68	0.81	0.65
News20	9.54	0.76	0.67	6.47	0.84	0.77

Although the A100 GPU is more powerful than the T4, the results generally show superior speedups for the T4. This can be explained by the reference times, as the multi-threaded implementations of Parallel-FST are much slower in the Pluton system. Not only they are executed with half the cores of the FT3, but their achieved efficiencies are also much lower.

5.4.2 Energy consumption

The energy consumption of our GPU implementations and the multicore versions of Parallel-FST was also analyzed. As power measurements were not available in the clusters, the consumption was estimated as the product of the runtime (see Table 11), the number of CPUs or GPUs used, and the Thermal Design Power (TDP) of each CPU/GPU (shown in Table 1). Table 12 summarizes the results for both systems and all datasets.

The results show that cuFEAST, in addition to being faster than Parallel-FST, requires less energy to complete the same tasks. On average, the GPU implementations use $90\% \pm 4\%$ and $81\% \pm 13\%$ less energy than their multicore counterparts in Pluton and FT3, respectively. The reduced-precision versions are even more energy-efficient, consuming $32\% \pm 17\%$ (Pluton) and $12\% \pm 8\%$ (FT3) less energy.

5.5 Summary and discussion

Table 13 summarizes the relative impact of each optimization step applied to the CUDA implementations, depending on the GPU. The benefits of using shared memory and/or asynchronism vary with the dataset and GPU architecture, while the reduced-precision version consistently provides additional speedups for all datasets and systems.

Overall, the results show that our CUDA implementations of BetaGamma and ICAP achieve substantial acceleration over the sequential and multicore versions provided

Table 13 Relative runtime improvements of the proposed optimizations. Values show the average variations across datasets, with ranges in parentheses (lower is better)

Optimization	Pluton (T4)	FT3 (A100)
1. Shared memory	−13% (+9%, −30%)	−28% (−8%, −41%)
2. Asynchronism	−2% (0%, −6%)	−14% (+1%, −30%)
3. Reduced-precision	−32% (−11%, −49%)	−12% (−2%, −23%)

by FEAST and Parallel-FST, respectively. The incremental optimizations significantly reduce execution time, the reduced-precision versions provide further improvements with minimal loss of accuracy, and the comparison with the state of the art demonstrates clear advantages in both performance and energy efficiency.

6 Conclusions

FS is a fundamental technique in data mining because it improves the performance of ML models by eliminating irrelevant or redundant data, which is especially critical as the dimensionality of datasets increases. However, traditional FS methods, particularly those based on greedy approaches, can be computationally expensive and require long execution times when applied to large datasets. This situation drives the search for more efficient solutions that optimize the use of available computational resources.

In this work, CUDA implementations of FS methods that use CMI have been developed, specifically the algorithms belonging to the BetaGamma family and the ICAP method. These techniques, widely used in FS, allow the evaluation of relevance, redundancy, and conditional redundancy of attributes within a dataset. However, their computationally intensive nature poses a challenge, especially when dealing with large amounts of information.

Two approaches have been taken to improve the performance of the FS implementations. On the one hand, we optimized the use of GPU resources with multiple asynchronous streams that allow overlapping CPU and GPU computations, as well as an efficient use of shared memory that adapts to the characteristics of the GPUs and the input datasets. On the other hand, a reduced-precision version based on 16-bit datatypes has been developed to further accelerate computation in those common scenarios where 64-bit precision is not required.

The experimental evaluation carried out on two systems with different CPU and GPU architectures (Turing and Ampere), using four publicly available datasets from different domains, has allowed a detailed analysis of the impact of each proposed optimization. The adaptive use of shared memory provides up to 41% of performance improvement, while the appropriate choice of the number of streams can improve the performance by another 30%. In scenarios where 16 bits are sufficient, the reduced-precision version achieves up to 49% improvement. In addition, an analysis of more than 40,000 experiments was performed, allowing the identification of parameter configurations that maximize performance for each GPU architecture.

Finally, the GPU implementations developed in this work have been shown to be significantly faster than parallel counterparts optimized for multi-CPU systems, even when the latter use 32 or 64 CPU cores. This performance improvement confirms that using GPUs for FS is not only feasible, but also highly efficient compared to CPU-only alternatives. As a result, this study reinforces the importance of parallel architectures and hardware optimization in the context of data mining and ML, opening up new possibilities for the efficient processing of large-scale datasets in various scientific and industrial applications. Furthermore, these optimizations have been included in the publicly available cuFEAST library[4] for use by the research community. Our future plans include multi-GPU versions for further acceleration.

Acknowledgements We gratefully thank the Galician Supercomputing Center (CESGA) for the access granted to its supercomputing resources.

Author Contributions B.B. : Methodology, Software, Validation, Data curation, Writing – original draft, Writing – review & editing, Visualization. J.G-D. : Conceptualization, Writing – original draft, Writing – review & editing, Supervision, Funding acquisition. J.T. : Conceptualization, Resources, Writing – review & editing, Supervision, Funding acquisition.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This work was supported by grant PID2022-136435NB-I00, funded by MICIU/AEI/10.13039/501100011033 and "ERDF A way of making Europe" (EU), grant TSI-100925-2023-1, funded by the Ministry for Digital Transformation and Civil Service, and FPU predoctoral grant of Bieito Beceiro ref. FPU20/00997, funded by the Ministry of Science, Innovation and Universities. It was also supported by the Consolidation Program of Competitive Reference Groups of Xunta de Galicia (ref. ED431C 2025/33). Funding for open access charge: Universidade da Coruña/CISUG.

Data Availability The datasets analyzed in this study are available in the LIBSVM repository, https://www.csie.ntu.edu.tw/cjlin/libsvmtools/datasets/.

[4] https://gitlab.com/bieito/parallel-fst/-/tree/master/cuFEAST

Declarations

Competing interests The authors declare no competing interests.

Ethics approval and consent to participate Not applicable.

References

1. Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L.A.: Feature Extraction: Foundations and Applications. Studies in Fuzziness and Soft Computing, vol. 207. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/978-3-540-35488-8
2. Bolón-Canedo, V., Sánchez-Maroño, N., Alonso-Betanzos, A.: A review of feature selection methods on synthetic data. Knowl. Inf. Syst. **34**(3), 483–519 (2013). https://doi.org/10.1007/s10115-012-0487-8
3. Beceiro, B., González-Domínguez, J., Morán-Fernández, L., Bolón-Canedo, V., Touriño, J.: CUDA acceleration of MI-based feature selection methods. J. Parallel Distrib. Comput. **190**, 104901 (2024). https://doi.org/10.1016/j.jpdc.2024.104901
4. Jakulin, A.: Machine learning based on attribute interactions. Ph.D. dissertation, University of Ljubljana, Sežana (2005). https://repozitorij.uni-lj.si/IzpisGradiva.php?id=24291&lang=eng
5. Brown, G., Pocock, A., Zhao, M.-J., Luján, M.: Conditional likelihood maximisation: a unifying framework for information theoretic feature selection. Journal of Machine Learning Research **13**, 27–66 (2012)
6. Battiti, R.: Using mutual information for selecting features in supervised neural net learning. IEEE Trans. Neural Networks **5**(4), 537–550 (1994). https://doi.org/10.1109/72.298224
7. Lin, D., Tang, X.: Conditional infomax learning: an integrated framework for feature extraction and fusion. In: Computer Vision - ECCV 2006. Lecture Notes in Computer Science, vol. 3951, pp. 68–82. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11744023_6
8. Shams, R., Barnes, N.: Speeding up mutual information computation using NVIDIA CUDA hardware. In: 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA 2007), pp. 555–560 (2007). https://doi.org/10.1109/dicta.2007.4426846
9. Ikeda, K., Ino, F., Hagihara, K.: Efficient acceleration of mutual information computation for nonrigid registration using CUDA. IEEE J. Biomed. Health Inform. **18**(3), 956–968 (2014). https://doi.org/10.1109/jbhi.2014.2310745
10. Shi, H., Schmidt, B., Liu, W., Müller-Wittig, W.: Parallel mutual information estimation for inferring gene regulatory networks on GPUs. BMC. Res. Notes **4**(1), 189 (2011). https://doi.org/10.1186/1756-0500-4-189
11. Ramírez-Gallego, S., Lastra, I., Martínez-Rego, D., Bolón-Canedo, V., Benítez, J.M., Herrera, F., Alonso-Betanzos, A.: Fast-mRMR: fast minimum redundancy maximum relevance algorithm for high-dimensional big data. Int. J. Intell. Syst. **32**(2), 134–152 (2017). https://doi.org/10.1002/int.21833
12. González-Domínguez, J., Expósito, R.R., Bolón-Canedo, V.: CUDA-JMI: acceleration of feature selection on heterogeneous systems. Futur. Gener. Comput. Syst. **102**, 426–436 (2020). https://doi.org/10.1016/j.future.2019.08.031
13. Cuomo, S., Galletti, A., Marcellino, L., Navarra, G., Toraldo, G.: On GPU-CUDA as preprocessing of fuzzy-rough data reduction by means of singular value decomposition. Soft. Comput. **22**(5), 1525–1532 (2018). https://doi.org/10.1007/s00500-017-2887-x
14. Venkatesh, B., Anuradha, J.: Fuzzy rank based parallel online feature selection method using multiple sliding windows. Open Computer Science **11**(1), 275–287 (2021). https://doi.org/10.1515/comp-2020-0169
15. Firouznia, M., Ruiu, P., Trunfio, G.A.: Robust feature selection for high-dimensional datasets using a GPU-accelerated ensemble of cooperative coevolutionary optimizers. In: 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2023), pp. 291–298 (2023). https://doi.org/10.1109/pdp59025.2023.00052
16. Zhu, H., Wu, Y., Li, P., Zhang, P., Ji, Z., Gong, M.: An OpenCL-accelerated parallel immunodominance clone selection algorithm for feature selection. Concurrency and Computation: Practice and Experience **29**(9), 3838 (2017). https://doi.org/10.1002/cpe.3838
17. Escobar, J.J., Ortega, J., González, J., Damas, M., Díaz, A.F.: Parallel high-dimensional multi-objective feature selection for EEG classification with dynamic workload balancing on CPU-GPU architectures. Cluster Computing **20**(3), 1881–1897 (2017). https://doi.org/10.1007/s10586-017-0980-7
18. Chang, H.-H., Li, C.-Y.: An automatic restoration framework based on GPU-accelerated collateral filtering in brain MR images. BMC Med. Imaging **19**(1), 8 (2019). https://doi.org/10.1186/s12880-019-0305-9
19. Samat, A., Li, E., Du, P., Liu, S., Xia, J.: GPU-accelerated CatBoost-Forest for hyperspectral image classification via parallelized mRMR ensemble subspace feature selection. IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. **14**, 3200–3214 (2021). https://doi.org/10.1109/jstars.2021.3063507
20. Beceiro, B., González-Domínguez, J., Touriño, J.: Parallel-FST: A feature selection library for multicore clusters. J. Parallel Distrib. Comput. **169**, 106–116 (2022). https://doi.org/10.1016/j.jpdc.2022.06.012
21. Lewis, D.D.: Feature selection and feature extraction for text categorization. In: Proceedings of the Workshop on Speech and Natural Language, pp. 212–217 (1992). https://doi.org/10.3115/1075527.1075574
22. Peng, H., Long, F., Ding, C.: Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. IEEE Trans. Pattern Anal. Mach. Intell. **27**(8), 1226–1238 (2005). https://doi.org/10.1109/tpami.2005.159
23. Yang, H.H., Moody, J.E.: Data visualization and feature selection: new algorithms for nongaussian data. In: Advances in Neural Information Processing Systems, vol. 12, pp. 687–693. MIT Press, Cambridge, MA (1999). https://proceedings.neurips.cc/paper_files/paper/1999/hash/8c01a75941549a705cf7275e41b21f0d-Abstract.html
24. Morán-Fernández, L., Sechidis, K., Bolón-Canedo, V., Alonso-Betanzos, A., Brown, G.: Feature selection with limited bit depth mutual information for portable embedded systems. Knowledge-Based Systems **197**, 105885 (2020). https://doi.org/10.1016/j.knosys.2020.105885

25. Morán-Fernández, L., Blanco-Mallo, E., Sechidis, K., Alonso-Betanzos, A., Bolón-Canedo, V.: When size matters: Markov blanket with limited bit depth conditional mutual information. In: IoT Streams for Data-Driven Predictive Maintenance and IoT. Edge, and Mobile for Embedded Machine Learning, vol. 1325, pp. 243–255. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-66770-2_18

26. Chang, C.-C., Linn, C.-J.: LIBSVM: a library for support vector machines. ACM Trans. Intell. Syst. Technol. **2**(3), 27 (2011). https://doi.org/10.1145/1961189.1961199