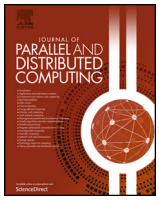
Journal of Parallel and Distributed Computing 190 (2024) 104901

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc



CUDA acceleration of MI-based feature selection methods

Bieito Beceiro [a,*], Jorge González-Domínguez [a], Laura Morán-Fernández [b], Verónica Bolón-Canedo [b], Juan Touriño [a]

[a] CITIC, Computer Architecture Group, Universidade da Coruña, Campus de Elviña s/n, 15071, A Coruña, Spain
[b] CITIC, Department of Computer Science, Universidade da Coruña, Campus de Elviña s/n, 15071, A Coruña, Spain

ARTICLE INFO

Keywords:
Feature selection
Mutual information
Low precision
Fixed point
CUDA

ABSTRACT

Feature selection algorithms are necessary nowadays for machine learning as they are capable of removing irrelevant and redundant information to reduce the dimensionality of the data and improve the quality of subsequent analyses. The problem with current feature selection approaches is that they are computationally expensive when processing large datasets. This work presents parallel implementations for Nvidia GPUs of three highly-used feature selection methods based on the Mutual Information (MI) metric: mRMR, JMI and DISR. Publicly available code includes not only CUDA implementations of the general methods, but also an adaptation of them to work with low-precision fixed point in order to further increase their performance on GPUs. The experimental evaluation was carried out on two modern Nvidia GPUs (Turing T4 and Ampere A100) with highly satisfactory results, achieving speedups of up to 283x when compared to state-of-the-art C implementations.

1. Introduction

In recent years we have witnessed the Big Data phenomenon, where data is in continuous increase in different areas such as bioinformatics, marketing, physics or engineering. These data are interesting when we can extract useful information from them to further guide decisions or make conclusions. However, this data increase leads to computationally costly analyses, and sometimes even to worse conclusions due to the presence of redundant or irrelevant data [5].

The procedure within the Machine Learning (ML) field that chooses only those characteristics that provide relevant information is called Feature Selection (FS) [17]. In recent years, in the field of FS research, many works have emerged focused on the development of different algorithms that use certain criteria for the selection process [5]. These criteria should try to find the most relevant features, but in such a way that the redundancy among them is minimal. Some of the most widely used and well-known methods that reach these goals are those based on Mutual Information (MI) [7], which have been satisfactorily employed in different scenarios, such as medicine [6,13], genetics [1,18], marketing [3,32], biomedicine [30] or electronics [11].

However, MI-based FS algorithms present quadratic complexity (each feature must be compared to the other ones in every step), leading to high runtimes for large datasets. High Performance Computing (HPC) is key to reduce these runtimes and make them more feasible for Big Data. GPUs are nowadays very popular among HPC architectures as they provide high computational power with low energy requirements, and they have been exploited for many years to accelerate the calculation of MI [25]. In this work we present CUDA implementations for three highly used MI-based FS methods that efficiently exploit the computational capabilities of Nvidia GPUs. Concretely, the main strengths of these implementations are:

- They are based on three methods that use MI to distinguish between relevant and irrelevant features. The use of MI for this purpose is well known and widely adopted by the ML community.
- They are highly optimized for modern Nvidia GPUs (Turing and Ampere architectures).
- They include a specific version of the algorithms to work with fixed point that further accelerates FS on GPUs.
- They are publicly available to download from https://gitlab.com/bieito/parallel-fst.

The rest of the paper is organized as follows. The state of the art and related work are summarized in Section 2. Some concepts about MI-based FS that are necessary to understand the algorithms are explained in Section 3. Section 4 describes the CUDA implementations, while Sec-

* Corresponding author.
E-mail addresses: bieito.beceiro.fernandez@udc.es (B. Beceiro), jgonzalezd@udc.es (J. González-Domínguez), laura.moranf@udc.es (L. Morán-Fernández), veronica.bolon@udc.es (V. Bolón-Canedo), juan@udc.es (J. Touriño).

https://doi.org/10.1016/j.jpdc.2024.104901
Received 12 May 2023; Received in revised form 16 November 2023; Accepted 11 April 2024
Available online 18 April 2024

tion 5 shows the experimental evaluation and Section 6 presents the conclusions and future work lines.

2. Related work

There are some works in the literature that already tried to accelerate FS methods on GPUs. Nevertheless, up to our knowledge, CUDA-JMI [16] is the only previous parallel implementation that can be directly compared to any of the three methods addressed in this work. Concretely, CUDA-JMI is focused on the Joint Mutual Information (JMI) method (see Section 3.2) and obtained good performance on past Nvidia GPU architectures such as Kepler and Maxwell. However, as will be seen in Section 5.2, it is not well prepared to exploit the characteristics of more recent Nvidia GPU architectures such as Turing and Ampere. Another work that addressed a GPU implementation of an FS algorithm based on MI is Fast-mRMR [23], with several versions (C++, CUDA and Apache Spark) of a variant of the minimum Redundancy Maximum Relevance (mRMR) algorithm. Nevertheless, the performance of the GPU version is quite limited (speedups of up to 5.32x compared to a sequential C version of mRMR). This is because it is based on an old CUDA version (for instance, it cannot exploit the modern hardware atomic operations) and does not include features to improve memory management such as a memory pool or asynchronism through streams.

FS methods not based on MI have also been adapted to work on GPUs, but none of these works have led to publicly available software. Some examples are CUDA implementations of the Singular Value Decomposition [10], the Local Kernel Density Ratio [2], and an online FS algorithm [29]. Apart from CUDA, OpenCL is also used to exploit the hardware of GPUs, for example, in a version of the Immunodominance Clone Selection Algorithm (ICSA) [33]. There are also parallel implementations of FS methods directly designed for specific fields such as electroencephalogram classification [12], brain image restoration [9], periocular biometric recognition [14], or hyperspectral image classification [24].

Finally, parallel implementations of FS methods for other HPC infrastructures have also been addressed, from clusters and supercomputers using the message-passing paradigm [4,15,27] to Big Data systems using the Hadoop and/or Spark frameworks [21,26,28].

3. Background: feature selection with mutual information and fixed point

As already mentioned in Section 1, FS is the process of selecting the relevant features and discarding the irrelevant or redundant ones. Many datasets include noisy and useless features, which waste a lot of computational resources. Therefore, FS plays a crucial role in the ML framework by removing nonsense features and preserving a small subset to reduce the computational complexity.

One of the most common metrics to capture dependencies between features in ML is MI. Let X be the set of features of a given problem, and Y the class label. MI is defined as the expected logarithm of a ratio in the following way:

$$I(X;Y) = \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}, \tag{1}$$

where $p(x,y)$ is the probability mass function of the joint distribution when the random variable X takes on the value x from its alphabet $\mathcal{X}$ and Y takes on $y \in \mathcal{Y}$, while $p(x)$ and $p(y)$ are the probability mass functions of the marginal distributions. In practice, the sample (maximum likelihood) estimates of the probabilities $\hat{p}$ are used, and the equation results in:

$$I(X;Y) \approx \hat{I}(X;Y) = \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} \hat{p}(x,y) \log \frac{\hat{p}(x,y)}{\hat{p}(x)\hat{p}(y)} \tag{2}$$

The computation of MI is at the heart of several information theoretic FS methods [7]. In this work, we will focus on three of the most used ones, which will be described in the following subsections.

3.1. Minimum redundancy maximum relevance (mRMR)

The minimum redundancy and maximum relevance optimization criteria, both based on MI, are the foundation of the mRMR method [22]. The score of a certain feature X_k, when a subset of features S has already been selected, is calculated as:

$$mRMR(X_k) = I(X_k;Y) - \frac{1}{|S|} \sum_{X_j\in S} I(X_k;X_j) \tag{3}$$

As the feature set S grows, the mRMR criterion has a stronger belief in the assumption that the selected features are pairwise independent. mRMR tries with the two terms of the equation not only to capture the relevance to the class, but also to avoid the redundancy among features of S.

3.2. Joint mutual information (JMI)

The JMI method [31] is focused on increasing complementary information between features given the class labels by using the following score for feature X_k:

$$JMI(X_k) = \sum_{X_j\in S} I(X_k X_j;Y), \tag{4}$$

where the information between the target and a joint variable $X_k X_j$, which associates X_k with each previously chosen feature, is calculated. The main principle is that we should incorporate a new feature if it is "complementary" to already existing features.

3.3. Double input symmetrical relevance (DISR)

The DISR method [19] is a normalized variant of the JMI criterion. DISR combines two well-known properties of FS. First, a set of features might provide more information about an output class than the total of the information provided by each feature considered separately. Second, it seems natural to assume that the combination of the best-performing subsets of features is the most promising set in the absence of any additional knowledge about how subsets of features should be combined. It uses the following modification of the JMI criterion:

$$DISR(X_k) = \sum_{X_j\in S} \frac{I(X_k X_j;Y)}{H(X_k X_j Y)}, \tag{5}$$

where I is the MI and H is the entropy, which quantifies the uncertainty present in the distribution of X. It is defined as:

$$H(X) = -\sum_{x\in\mathcal{X}} p(x) \log p(x) \tag{6}$$

The entropy, however, can be conditioned by other events. The definition of the conditional entropy of X given Y is defined as follows:

$$H(X|Y) = -\sum_{y\in\mathcal{Y}} p(y) \sum_{x\in\mathcal{X}} p(x|y) \log p(x|y) \tag{7}$$

The relation between the entropy, the conditional entropy, the MI and the joint entropy ($H(X,Y)$) can be seen in Fig. 1. More information about these information theoretic quantities can be found in [7].

3.4. Fixed point for MI-based feature selection

FS is usually performed on machines with high-precision representation, i.e. double-precision floating-point computations (64 bits). The use of a more powerful general purpose processor can provide significant benefits in terms of speed and capability to solve more complex

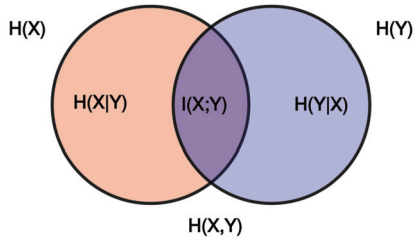


Fig. 1. Illustration of different information theoretic metrics.

problems. But this capability does not come without cost, as a conventional microprocessor can require a substantial amount of off-chip hardware support, memory, and often a complex operating system. Modern Nvidia GPUs provide support to perform operations with lower precision (32, 16 or even 8 bits) at high speed. Therefore, adapting FS methods to work with these data types can help to increase the performance of the CUDA-based implementations. Moreover, being able to reduce the precision of the computations performed by the FS algorithms reduces the memory footprint, thus enabling larger models to fit within the given memory capacity and lowering the bandwidth requirements. Morán-Fernández et al. [20] proposed the use of a lighter procedure for the MI computation based on the usage of low-precision fixed-point operations. Although it was originally designed for embedded systems, Section 4.4 will describe how it was adapted for Nvidia GPUs. A fixed-point representation with b_i as the number of integer bits and b_f as the number of fractional bits was intended as an alternative to the traditional floating-point 64-bit resolution.

Since MI parameters are typically represented in the logarithmic domain, a look-up table is used to determine the logarithm of the probability of a particular event. The look-up table is indexed in terms of number of occurrences of an event (individual counters) and the total number of events (total counter), and stores values for the logarithms in the desired fixed-point representation. To limit the maximum size of the look-up table and the bit-width required for the counters, a maximum integer number M is assumed and the look-up table L is precomputed such that:

$$L(i,j) = \left[\frac{\log(i/j)}{q}\right]_R \cdot q, \tag{8}$$

where $[\cdot]_R$ denotes rounding to the closest integer, q is the quantization interval of the desired fixed-point representation (2^{-b_f}), $\log(\cdot)$ is the logarithm in base 2, and the counters i and j are in the range $[0, M-1]$. We refer to [20] for more detailed information about the calculation of this look-up table.

The total counter S and the individual counters s_j^i are computed using the Algorithm 1 given a set of specific data. We made the assumption that there was some maximum integer number M, where $M = 2^{(bf+bi)} - 1$. The algorithm first checks that all counters are within bounds. Then, the total counter S is computed as the sum of all of them. However, it is important to check that S is still within bounds. In the case that $S \geq M$, an index correction must be applied, and both the total counter and all the individual counters are halved. Finally, the values of S and s_j^i can be used to retrieve the log-probability low-precision values from the look-up table.

4. CUDA implementation

As previously mentioned, GPUs are nowadays widely used in the HPC field due to their high computational capabilities. MI computation is a procedure with a high level of parallelism, so GPUs are a suitable architecture to accelerate it.

The CUDA implementations for three MI-based methods (mRMR, JMI and DISR) presented in this work are based on the sequential C

Algorithm 1: Low-precision mutual information.

Input: Individual counters s_j^i and total counter S; look-up table L
for i, j **do**
  // maximum value reached?
  if $s_j^i = M$ **then**
    // half counters (round down)
    $s_j^i \leftarrow s_j^i/2 \ \forall i, j$
  end
end
// sum of the individual counters
$S \leftarrow \sum(s_j^i)$
// ensure that S is in range
while $S \geq M$ **do**
  $S \leftarrow S/2$
  // revise index correction
  $s_j^i \leftarrow s_j^i/2 \ \forall i, j$
end
// get the log probability from look-up table
$l_j^i \leftarrow L(s_j^i, S) \ \forall i, j$
Return: l_j^i

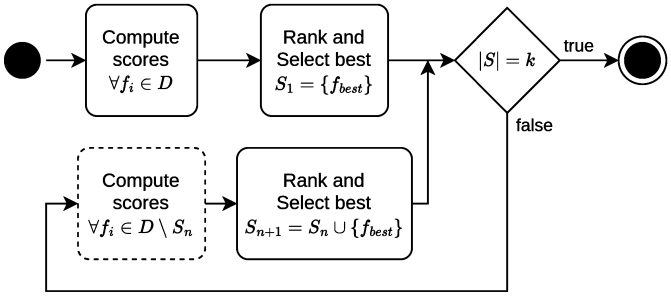


Fig. 2. High-level structure of the FS methods included in FEAST and cuFEAST, for the selection of a subset S with k features from a dataset D.

counterpart methods included in the FEAST[1] library, which are theoretically described in [7] and have been extensively tested and used by scientists. Our implementations are gathered in a library named cuFEAST. Note that these FS methods require discrete data, so the library also includes support to discretize continuous data using a binning approach. Specifically, the range of values of each feature is split into bins of constant size, where the number of bins must be indicated by the user. The cuFEAST implementations accept as input datasets in ARFF, CSV or LIBSVM formats, and share the common structure shown in Fig. 2, with the following two steps:

1. **Selection of the first feature**. At the beginning, the set of selected features S is empty, so redundancy cannot be computed yet. Thus, the value that is used as the score to select the first feature is the relevance, which can be approximated as the MI between each feature and the class.
2. **Selection of the other $k-1$ features**. This part consists of a loop that selects one more feature for the S set in each iteration. At this point, the algorithm has already selected at least one feature, so more sophisticated metrics can be used for score computation (see Section 3). Note that these advanced metrics take S into account to compute scores, as they try to minimize the redundancy with the already selected features. Hence, there is a dependency between iterations which does not allow for parallel execution (the scores of iteration $i+1$ cannot be calculated until the feature corresponding to iteration i has been selected).

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

Algorithm 2: Computation of all scores for an iteration of mRMR, JMI and DISR in FEAST. Green lines correspond to the mRMR algorithm only. Purple lines are exclusive to JMI and DISR.

Input: Dataset D; subset of already selected features S; cache of inter-feature scores PC; cache of MI values with dataset class $classMI$
Output: Computed scores of features P; updated score cache PC

```
1 for feature X_i ∈ D \ S do
2     P_i ← classMI_i
3     P_i ← 0
4     for feature X_j ∈ S do
5         if j = |S| − 1 then
6             // X_j is the most recently selected feature
              L ← X_j
              PC_ji ← computeScore(X_i, L)
          end
7         P_i ← P_i + PC_ji/|S|
          P_i ← P_i + PC_ji
      end
  end
```

Algorithm 3: Computation of all scores for an iteration of mRMR, JMI and DISR in cuFEAST (with the rearrangement). Green lines correspond to the mRMR algorithm only. Purple lines are exclusive to JMI and DISR.

Input: Dataset D; subset of already selected features S; cache of inter-feature scores PC; cache of MI values with dataset class $classMI$
Output: Computed scores of features P; updated score cache PC

```
1 l ← |S| − 1
2 L ← X_l
3 for feature X_i ∈ D \ S do
4     PC_li ← computeScore(X_i, L)
  end
5 for feature X_i ∈ D \ S do
6     P_i ← classMI_i
      P_i ← 0
7     for feature X_j ∈ S do
8         P_i ← P_i + PC_ji/|S|
          P_i ← P_i + PC_ji
      end
  end
```

The focus of this work is the acceleration of the score computation (dashed outline in Fig. 2) since it is the most computationally expensive part of the algorithms. Furthermore, it is highly parallel as the calculation of the score is independent for each feature.

The computation of scores in the three methods (mRMR, JMI and DISR) shares a common structure, which is shown in Algorithm 2. The algorithm iterates over all the features of the dataset D (loop of line 1) to compute their scores P. The scores are initialized (line 2) to the MI of the active feature X_i and the class (denoted by Y in the formulas of Section 3) in mRMR, and to zero in JMI and DISR. Then, the inner loop (line 4) iterates over each already selected feature X_j, so the score P_i is updated depending on the information metric between X_i and X_j (the metric depends on the selected algorithm).

Note that this pseudocode shows an optimization that was already included in the base implementation of the FEAST algorithms, that is, the usage of two caches:

- `classMI` (mRMR only) stores the MI between each feature in the dataset and the class. These values are accessible at the beginning of the algorithm since they were already computed to be used as scores for the selection of the first feature.
- `PC` is a matrix that stores all the partial scores between each feature of the dataset and each previously selected feature. This optimization heavily reduces the complexity of the code: as the scores related to the previously selected features can be directly obtained from the PC matrix, only the score related to the last selected feature must be calculated.

These caches are also used in cuFEAST. Moreover, the code has been rearranged to later benefit from optimizations such as batch processing and asynchronism (explained in detail in later sections). This rearrangement is shown in Algorithm 3 and consists in the extraction of the partial score computation to the outer level of the loops (lines 3-4), so that computations for all the features in the dataset are grouped.

Fig. 3 shows the concrete operations involved in the computation of the score of any feature X, the last selected feature L, and the dataset class Y in each method. Note that MI is computed in the three methods, but whereas in mRMR just the two features are involved, JMI and DISR also use the class and their joint state, which are calculated in a previous stage called merge arrays ("mA" in the figure). Finally, the computation of the DISR scores involves an additional metric with respect to JMI, that is, the joint entropy (H) between the joint state of the features and the dataset class (see Section 3.3).

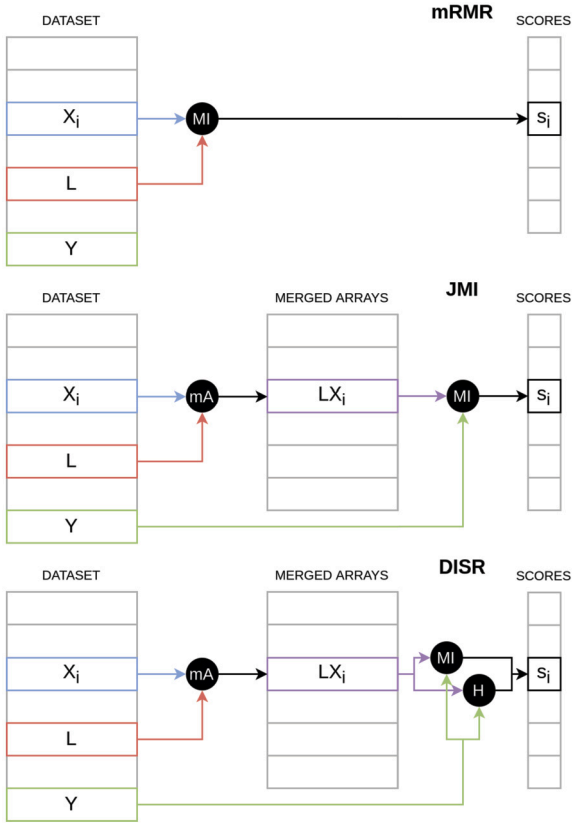


Fig. 3. Operations involved in the computation of the score of an arbitrary feature X_i, where L is the last selected feature and Y the dataset class.

In summary, we have three methods that share a common structure, but differ in the calculations that are performed to compute feature scores, which is the most computationally expensive part of the algorithms. For this reason, we will focus on the adaptation of that part to be accelerated with GPUs. Due to previous optimizations and code refactoring, the heaviest computation has been grouped in a single function, whose parallelization will be explained in the following subsections.

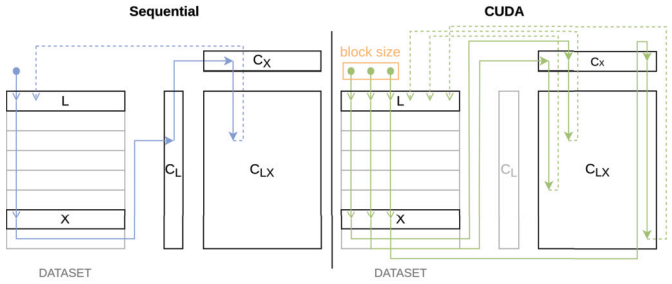


Fig. 4. Comparison of sequential and CUDA implementations of histogram creation.

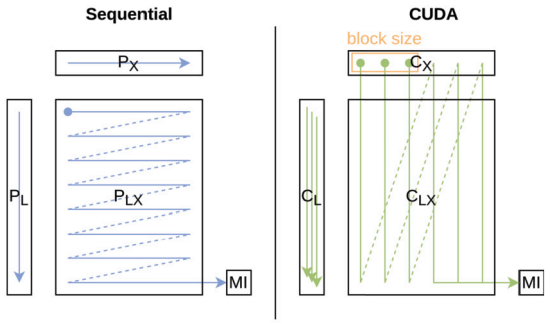


Fig. 5. Comparison of sequential and CUDA implementations of histogram processing for MI.

4.1. CUDA implementation for mRMR

As explained in Section 3.1, the mRMR algorithm uses the MI to compute the feature scores. Due to the caching and code refactoring, the operation that must be performed in parallel is the calculation of MI between every not selected feature in the dataset and the last selected one (lines 3 and 4 in Algorithm 3).

The calculation of the MI between one feature X and the last selected one (L) can be divided into three steps:

1. **Histogram creation**. First, three histograms (i.e., counters of the occurrences for the different values) are created: a unidimensional histogram for L (C_L), a unidimensional one for X (C_X), and a bidimensional one for the joint state of X and L (C_{LX}). See left half of Fig. 4 (labeled as "Sequential").
2. **Conversion to probabilities**. Then, all values in histograms C_X, C_L and C_{LX} are divided by the number of samples to obtain probabilities from counts, and are stored in P_X, P_L and P_{LX}, respectively.
3. **Histogram processing**. Finally, the probability structures are processed to obtain the MI between the two variables, as seen in Equation (2). See left half of Fig. 5.

4.1.1. CUDA kernel

The mRMR implementation in cuFEAST includes only one kernel that covers the three steps explained before. One CUDA block per feature is created, and the work for each MI calculation is divided among the threads of the block as follows:

1. **Histogram creation**. Each thread T_t works with the values corresponding to a same position in the two variables (X_t and L_t), and updates the histograms according to the read values ($C_X[X_t]$ and $C_{LX}[X_t][L_t]$). Note that it is not necessary to calculate C_L for every feature, as it corresponds to the last selected feature, which is used for all the MI calculations. Therefore, performance is improved by computing it only once and storing it in a memory that is accessible by CUDA threads for the third step. The CUDA approach to create the histograms is illustrated in the right half of Fig. 4.
2. **Conversion to probabilities**. The conversion from counts to probabilities is done in a "lazy" fashion. That is, instead of having the values stored, their computation is delayed until the probabilities are needed. This allows to reduce memory consumption since the storage of three additional structures that use 64-bit data (i.e., P_X, P_Y and P_{XY} with `float64`) is avoided.
3. **Histogram processing**. In the CUDA kernel, the histograms are divided among threads in a column fashion. That is, each thread computes the partial MI of a value of the C_X histogram, all C_L, and the corresponding column of C_{LX}. For instance, a thread T_t would process the partial MI with the value $C_X[t]$, all C_L, and the column $C_{LX}[*][t]$ (see the right half of Fig. 5). Then, all threads aggregate their partial solutions to achieve the total MI.

Note that, as there are several CUDA threads working concurrently, the operations that update memory positions must be performed atomically to keep a consistent memory model. Specifically, the CUDA function `atomicAdd_block` is used for histogram creation and the addition of the partial MI of each thread to the complete MI. Moreover, to get consistency, the threads must be synchronized before advancing from the histogram creation to its processing. This is achieved with the `__syncthreads()` CUDA function.

As mentioned before, each thread is responsible for the computations of a position of the input features during the histogram creation, and of a column of the bidimensional histogram for the calculation of the partial MI. We have also seen that a single block is used to process each feature, but the amount of threads per block is fixed for each execution (selected by the user as input parameter). A cyclic approach is applied to split the computation among threads: for a workload of W items, each item W_n is assigned to thread n mod $block\_size$. Note that $W = \#samples$ for the histogram creation, while $W = len(C_X) = max(X)$ for the processing.

Finally, the GPU memory used by the histograms has to be explicitly managed (allocated, initialized and freed). This introduces a certain overhead, but it will be reduced with some optimizations described in the following subsections.

4.1.2. Batch processing

Now that a kernel that computes the MI between two features has been presented, the implementations need to repeat this calculation for each feature in the dataset. Taking into account that one CUDA block is in charge of the MI value for a certain feature (the total number of features can be much higher than the maximum number of CUDA blocks), three different approaches were identified to calculate the MI between the last selected feature and all the not already selected ones:

1. **Launch one kernel per feature**. This would be the most naive approach, since it would consist of repeatedly resetting GPU memory and launching the kernel with only one CUDA block. The main benefit is that its implementation is quite simple (a `for` loop), but this memory reset would introduce a really high overhead and it would not take advantage of memory bandwidth.
2. **Launch a single kernel for all features**. The opposite approach would be to rely on a single kernel call, with one block per feature. The implementation would be simpler, since the `for` loop is avoided, but we would need to allocate a huge amount of GPU memory at a time. Depending on the properties of the processed dataset, this could even cause the execution to fail due to excessive memory requirements.
3. **Batched approach**. This would be a trade-off between the two previous options. We could try to reduce the average overhead by using fewer kernels than the "one kernel call per feature" approach, while reducing the impact of memory management (allocations

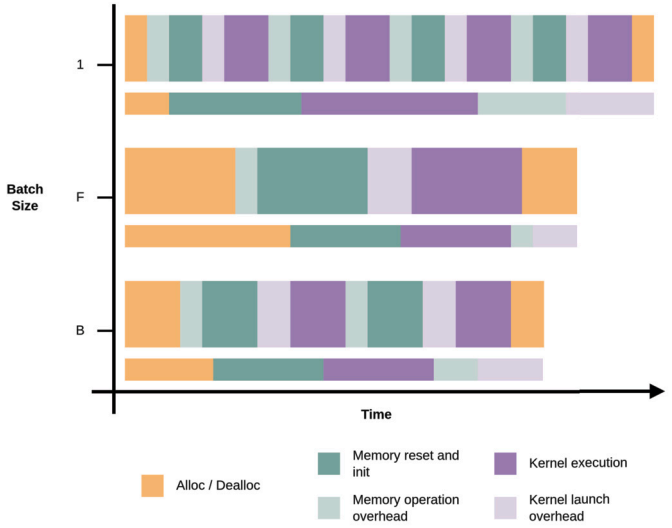


Fig. 6. Conceptual comparison of runtimes and overheads of each kind of operation involved in the CUDA calculation of scores.

and frees) by having fewer features per kernel than with the "single kernel call for all features" approach. However, this would lead to an additional configuration parameter (the number of features per batch) with a significant impact on performance that would depend on the properties of each dataset.

Fig. 6 illustrates the differences among the approaches. Note that F refers to the number of features in the dataset and B to the batch size (given that $1 < B < F$). Two bars are shown for each batch configuration: the highest one shows a timeline view for the duration of each operation, while the shortest one displays the aggregate times of each kind of operation. The chart clearly shows that a smaller batch spends less time with memory allocations and deallocations, while a larger one reduces overhead and enhances kernel performance (it better exploits GPU parallelism). It is also easy to see how the batched approach is an intermediate case that, if balanced, can outperform both extreme cases in overall runtime. Therefore, a batch size that provides a compromise between low overhead highly parallel executions and low memory allocations should be selected.

In order to achieve adaptability, the number of features per batch is left as a configuration parameter to the user. This way, the execution can be tuned depending on the properties of the dataset and also adapted to other CUDA architectures that potentially have different ratios between computation and memory performance.

4.1.3. Custom GPU memory management

Even with the batched approach, there is still room to optimize memory accesses. As explained before, we need to allocate and free memory for the histograms of the features that are packed in each batch, but these memory operations come with a performance overhead. The cuFEAST implementation of mRMR includes a custom memory management system based on pools: instead of using `cudaMalloc` and `cudaFree` at the beginning and end of each batch processing, a pool of memory is allocated at the beginning and deallocated at the end of the whole execution, and it is just reset before launching each batch kernel.

Note that this approach works because a top limit for the amount of required memory per batch can be estimated beforehand. The maximum can be estimated as $B \times (max(len(C_X)) + max(len(C_X)) \times len(C_L) \times d$, where B is the batch size, C_L is the histogram of the last selected feature, $max(len(C_X))$ is the maximum length of the histogram of any variable X in the dataset (usually equal to the $\#bins$ used for discretization), and d is the size (in bytes) of the data type used for the histograms.

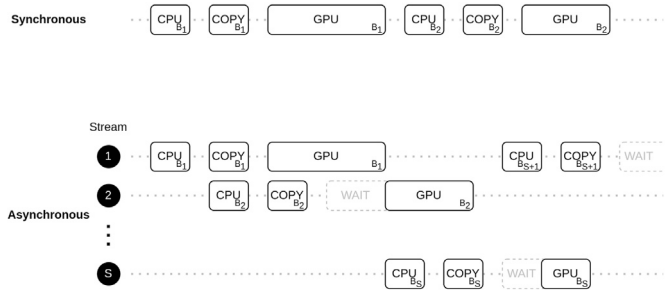


Fig. 7. Comparison of the processing using a synchronous and an asynchronous (with S streams) approach.

Regarding memory requirements, cuFEAST uses `uint16` for the histograms, so these requirements are reduced to a half if compared to the 32-bit `uint`. Moreover, the original C implementation in FEAST saves in memory both the histograms with counters (32-bit `uint`) and probabilities (64-bit `float64`), so the real reduction is higher (from 96 bits to just 16 bits per element). However, as a side effect, the maximum value that can be represented is $2^{16} - 1 = 65535$, so it is possible that overflow occurs for datasets with $\#samples > 65535$. This is only common for the class histogram, since it usually takes values from a low amount of possibilities (i.e., the number of classes). So, in order to avoid this particular error, the class histogram still uses 32-bit `uint`s.

Being able to compute the amount of memory needed by the histograms of a feature with a simple operation based on its length helps to optimize even more the behavior of the memory pool. It is common that the actual size of the memory space required for the histograms of some features is lower than the maximum. Therefore, there would be allocated but uninitialized space in the pool. Since the computation of larger batches is more efficient (the drawback is the overload of memory operations), cuFEAST uses this uninitialized space to fit more features in the batch, allowing to compute more features without adding memory operation overload.

4.1.4. Asynchronism

In a system with a GPU, there are usually three "units" that can work simultaneously: host CPU, GPU compute engine, and GPU transfer engine. cuFEAST uses CUDA streams to overlap the work of all units in a pipelined structure. Specifically, the overlapped work is the preparation of the batch (CPU), the transfer of batch data (GPU transfer) and the kernel execution (GPU compute). Fig. 7 shows a comparison of the resource usage between a synchronous and an asynchronous approach. The batches are distributed cyclically among streams: when using S streams, a batch B_b would be processed in the stream $b \bmod S$.

Ideally, the configuration that achieves the best performance would be to use one stream per batch. But, in practice, the management of a high number of streams increases memory consumption: each stream needs its own memory space so that memory requirements rise linearly with the number of streams. Therefore, the number of streams is a parameter that also impacts performance and, as it multiplies the required memory allocation, the variations in performance will be linked to the batch size (as will be shown in Section 5.3.1). Again, the number of CUDA streams used for the asynchronism is left to the user as a configuration parameter, so that the execution can be tuned according to the specific properties of each dataset and CUDA architecture.

4.1.5. Shared memory

In the kernel explained so far the histograms are stored in the global memory of the GPU. This type of memory is the largest but slowest one in the CUDA memory hierarchy, so cuFEAST tries to take advantage of faster memories. This is the case of shared memory, since accesses are about 10x faster than in global memory and it is accessible from all the threads in each block (remember that cuFEAST launches one block of threads per feature, so only that block would need to access its

own shared memory). However, besides being faster, shared memory is much smaller than global (e.g., up to 64 KB per block for the Turing microarchitecture), so it cannot be used for all data structures in all scenarios.

A previous optimization, the usage of the `uint16` data type for the histograms, helps with the reduction of memory requirements, making it possible to fit larger histograms in shared memory. The implementation follows an "adaptive" shared memory usage, which tries to fit structures in it while there is remaining free space. The procedure consists of the following steps:

1. cuFEAST obtains a structure with, among other information, the amount of shared memory per block.
2. Then, for each feature in the batch, it checks which structures fit in shared memory:
   2.1. First, it checks the size of the bidimensional histogram, since it is the most accessed one. If this histogram fits, it is created and processed in shared memory.
   2.2. Second, it checks for the unidimensional one.
3. Finally, the CUDA kernel uses global or shared memory depending on the results from the previous steps.

Therefore, there are four possible scenarios (ordered from best to worst performance):

1. Both histograms fit in memory.
2. The bidimensional histogram fits, but not both, so the unidimensional one uses global memory.
3. The bidimensional histogram does not fit, so it uses global memory, but the unidimensional one fits in shared memory.
4. Both histograms need more space than available in shared memory, thus they are created in global memory.

Note that we first try to fit the largest histogram (the bidimensional one) in shared memory, since it is the most accessed one. During their creation, both the unidimensional and bidimensional histograms are written NS times, where NS is the number of samples in the dataset. However, the difference appears during the computation of the MI, since all positions of both structures are read. Therefore, the bidimensional histogram should be prioritized to use shared memory.

We have previously seen how the maximum amount of required memory per feature (i.e., per CUDA block) can be calculated. Note that the size of the histograms is limited by the bins used for the discretization, so the user can assert that shared memory will be used for both histograms when $(\#bins + \#bins^2) \times d < sharedMemPerBlock$, where d is equal to 16 bits and $sharedMemPerBlock$ depends on the GPU microarchitecture.

Furthermore, shared memory can work together with the custom memory management (see Section 4.1.3) to improve performance even further. When the histograms of a certain feature fit in shared memory, they do not consume any space of the allocated pool, so that space can be used to pack additional features in the batch, improving its workload. The implementation works as follows:

- **Host side**. An offset to a pointer (`offset_pointer`) is assigned to the CUDA block to know what space to use. When computing the offsets and memory sizes of each feature, if a histogram fits in shared memory, the pointer is set to `null`. Otherwise, it stores the address of global memory that the block will use.
- **CUDA kernel side**. When the CUDA block starts the execution, it checks the value of the `offset_pointer`. If it is `null`, then shared memory is initialized (in parallel) and used for the histograms. Otherwise, the memory area pointed by `offset_pointer` (which is already initialized) is used.

Algorithm 4: Computation of the merged state XL of two variables X and L.

```
Input: Feature X; last selected feature L; number of samples NS
Output: Merged state XL
dimension ← (max(X), max(L))
map ← zeros(dimension)
num_states ← 0
for index i ∈ [0, NS − 1] do
    pos ← (X_i, L_i)
    if map_pos = 0 then
        // first appearance of the state
        num_states ← num_states + 1
        map_pos ← num_states
    end
    XL_i ← map_pos
end
```

4.1.6. Complexity analysis

Although the mRMR algorithm is too dependent on the properties of the dataset, we can estimate its complexity. First, the number of CUDA blocks that are executed can be expressed as $O(k * F)$, where k is the number of features to select and F the number of features in the dataset. Then, the complexity of the workload of each block is given by $\Theta(NS) + O(\#bins^2)$, where NS is the number of samples in the dataset and $\#bins$ the number of bins used for the discretization. Note that the first term, $\Theta(NS)$, corresponds to the creation of the histograms, and the second one, $O(\#bins^2)$, to their processing.

Therefore, the overall complexity can be expressed as $O(k * F) * (\Theta(NS) + O(\#bins^2))$.

4.2. CUDA implementation for JMI

It was previously explained that the three methods addressed in this work (mRMR, JMI and DISR) share a common structure (see Algorithms 2 and 3). However, JMI and DISR have an additional step before the MI computation (see Fig. 3). While in mRMR the MI was computed between two features of the dataset, in JMI and DISR there are three features involved. Hence, four steps arise in JMI:

1. **Merge arrays**. Two features must be merged in order to obtain a new one that represents the "joint" state of both variables. For a feature X and the last selected one L, we refer to the merged state as XL.
2. **Histogram creation**. Same as in mRMR, but instead of using variables X and L, the merged one XL and the dataset class Y must be used.
3. **Conversion to probabilities**. Same as in mRMR.
4. **Histogram processing**. Same as in mRMR.

First, we need to understand how the original C function to merge arrays (already presented in Fig. 3) works in FEAST. This operation is used to compute the joint state of two variables, which is then used to calculate the JMI score. Each unique pair of X and L constitutes a state of XL (i.e., XL_i represents (X_i, L_i)). Note that XL will therefore have the same length as X and L (the number of samples of the dataset, denoted by NS).

In terms of implementation, the computation of the joint state of two batches needs a bidimensional memory structure ("merge map" from now on). This way, cuFEAST can track whether each pair (X_i, L_i) appears more than once. That is, each possible combination of values of X and L is represented univocally by one and just one position of the map. Note that the map needs to have a dimension of $max(X) \times max(L)$. The procedure to compute XL is detailed in Algorithm 4.

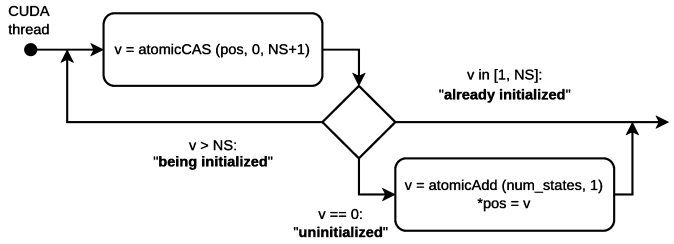


Fig. 8. Flow diagram for the critical read/read-and-update operation used in the CUDA implementation of merge arrays.

4.2.1. CUDA kernel

A naive CUDA approach for JMI would consist of dividing the workload into two kernels. First, all the X features would be merged with L to create a new dataset of merged features, and then a second kernel would compute the MI as in mRMR. However, this approach would increase the overhead of the kernel launches, as well as memory requirements.

Therefore, a custom operation that joins the first two steps of JMI into a single one was developed, merging the features "on the fly" and contributing to the histogram creation. This operation avoids having to allocate an extra structure with the same size of the dataset, and reduces global memory reads and writes.

The CUDA kernel for JMI implements all the steps involved in the computation of the scores in just two stages:

- **Merge arrays**. Each thread T is responsible for the same position in both variables (X_t and L_t). Once read, the values of the variables are used to compute the position in the merge map. This map is read (and updated if the state appears for the first time), and this way the value of the joint state XL_t is known. However, as this value is read only once for the histogram creation, instead of storing it, the thread updates the histograms accordingly (i.e., from the values XL_t and Y_t).
- **Histogram processing**. It is implemented as for mRMR. Remember that a synchronization is required before starting this stage.

The creation of the merge map is a critical task, since several threads can try to access the same position of the map simultaneously. Therefore, these memory operations must be performed atomically. A novel, fast, and lightweight atomic operation has been developed to reduce the synchronization overhead. This procedure is illustrated by the diagram shown in Fig. 8 and detailed in Algorithm 5. This operation avoids heavier synchronization functions (such as kernel-level) and reduces memory usage and the complexity of using mutexes. Each map position is written at most once (when its value is 0, as shown in line 6 of Algorithm 4), and then race conditions must be avoided only when two threads reach an unused position for the first time. The *atomicCAS : (address, compare, value) → (old)* CUDA function is used for it. This operation computes atomically[2][3]:

$$*address = ((old == compare)\ ?\ value : old)$$

and returns *old*. Two of the parameters are simple. On the one hand, the *address* is already known. On the other hand, *compare* is set to 0 as the map must only be written when the original value is still 0.

However, working with *old* and *value* is more tricky. In order to track the new states, a counter (*num_states*) is used, which must be shared among CUDA threads. Therefore, two shared variables (the map position and the counter) must be updated at once every time a thread finds a new state, so the *atomicCAS* function is not enough. To solve this problem, the memory position is used as a mutex to avoid race conditions. This way, when a thread reaches a position, its value can be "uninitialized", "being initialized" or "already initialized", and its action has to be "initialize the position", "wait until position is initialized" or "do nothing and continue", respectively. Since the map is set to zero at the beginning, a position that stores a zero can be considered as "uninitialized". However, other values are necessary to represent the other two states. When a state appears for the first time, the merge map saves the state index (which starts at one to avoid confusion with an "uninitialized" position). Note that the counter must also be updated atomically (in this case with *atomicAdd*). So, each position of the map stores the index of a distinct joint state, and the number of distinct states is limited by the number of samples (a used position will hold a value in $[1, NS]$). Therefore, any map position with a value in that range will be considered as "already initialized". Finally, $NS + 1$ is used to represent the "being initialized" state.

Algorithm 5: Concurrent-safe implementation of merge arrays for a thread T_t.

```
   Input: Feature X; last selected feature L; number of samples NS;
          shared merge map map; shared state counter num_states
   Output: Safe value of the joint state XL_t
1  pos ← (X_t, L_t)
2  repeat
       // atomically read position
       // try to set it to "being initialized" if "uninitialized"
3      v ← atomicCAS(map_pos, 0, NS + 1)
       // retry if position was "being initialized"
   until v > NS
4  if v = 0 then
       // position is "being initialized" by T_t (current thread)
       // add new state - atomic
       // other thread could be doing the same for a different pos
5      v ← atomicAdd(num_states, 1)
       // T_t can safely update the map position, no other threads
       // will write it
6      map_pos ← v
   else
       // nothing to do; position already initialized
   end
   // now v holds the value of XL_t and can be used safely
```

4.2.2. Batch processing and asynchronism

The same approach for batch processing and asynchronism already explained in Sections 4.1.2 and 4.1.4, respectively, has been applied to JMI. The only difference is that this method needs an additional memory structure (the merge map) and so the optimal batch size might vary from mRMR.

4.2.3. Custom GPU memory pool

The memory pool and its custom management presented for mRMR in Section 4.1.3 can also be applied to JMI. However, due to the fact that there is an additional step (merge arrays) previous to the MI computation, there is an aspect that must be taken into account. While for mRMR the size of the histograms can be calculated in advance before launching the kernel, it is not possible for JMI due to the fact that the size depends on the outcome of the merged array. Thus, the optimization of packing more features per batch to further improve performance cannot be applied here since we cannot assure whether allocated memory of the pool is left unused.

[2] Note that CUDA atomic operations are not available for architectures prior to Kepler.

[3] For the implementation, the atomic operation variants with a "`_block`" suffix were used because their performance is better and data consistency is maintained since they are atomic at a block scope.

Table 1
Maximum theoretical sizes of the data structures used in each algorithm for a given number of samples and number of bins. Sizes are indicated in number of elements.

NS	$\#bins$	mRMR		JMI		
		1D	2D	map	1D	2D
2,000	64	64	4,096	4,096	2,000	128,000
	128	128	16,384	16,384	2,000	256,000
	256	256	65,536	65,536	2,000	512,000
20,000	64	64	4,096	4,096	4,096	262,144
	128	128	16,384	16,384	16,384	2,097,152
	256	256	65,536	65,536	20,000	5,120,000

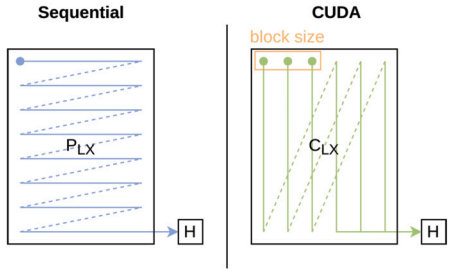


Fig. 9. Comparison of sequential and CUDA implementations of histogram processing for H.

However, an upper bound must be calculated in order to know the total memory size allocated for the pools. As mentioned above, the size of the histograms depends on the merged array, specifically on its maximum value. Given a feature X and the last selected one L with NS samples, we already know that the resulting joint state XL also has length NS. Moreover, the merge map has a position for each possible combination of values between X and L, so its size would be $map\_size = max(X) \times max(L)$. As previously seen, during the merge arrays step the state values are assigned sequentially, so we know that $max(XL)$ is a value between one (all states are equal) and $min(NS, map\_size)$ (all states are different).

Now, the maximum size of the histograms can be calculated. Remember that in JMI the score of a feature X, given the last selected one L, is computed as the MI between the class Y and XL (the joint state of X and L). The size of the bidimensional histogram can be computed as the product of the sizes of the unidimensional ones. That is, the one of the class Y (size $max(Y)$), and the one of the joint state XL (size $max(XL) = min(NS, max(L) \times max(X))$). Therefore, the maximum size of the bidimensional histogram would be:

$$max(Y) \times min(NS, max(L) \times max(X)) \tag{9}$$

A comparison of the maximum sizes for each data structure when using different numbers of bins and samples is provided in Table 1.

Due to the discretization, the maximum value of a feature is limited by the number of bins ($\#bins$). Therefore, the required memory for the bidimensional histograms of a batch of B features would be $B \times max(Y) \times min(NS, \#bins^2) \times d$, where d is the data size (16 bits). Note that $max(Y)$ is constant for the dataset, so cuFEAST precomputes and uses it instead of its theoretical maximum of $\#bins$.

4.2.4. Shared memory

cuFEAST modifies the "adaptive" approach explained for mRMR in Section 4.1.5 so that it can efficiently work with JMI. The main difference is that JMI has an additional memory structure (the merge map) that can also be stored in shared memory. Obviously, the ideal case would be to fit all three structures (the merge map and the two histograms) in shared memory, but, due to the large size of the merge map and the bidimensional histogram, mostly only one of them fits. The cuFEAST implementation of JMI gives the highest priority to the merge map as this worked best in most of the scenarios we tested. Therefore, the adaptive strategy in JMI tries to fit the structures in shared memory in this order: merge map, bidimensional histogram, unidimensional histogram.

4.2.5. Complexity analysis

The complexity of the JMI algorithm also depends on the input data, even more critically than mRMR. The number of CUDA blocks is the same as in mRMR, $O(k * F)$, where k is the number of features to select and F the number of features in the dataset. However, the workload of each block is different now, since JMI includes the step of merging arrays. The complexity of the work performed by each CUDA block is $\Theta(NS) + O(\#bins * \min(NS, \#bins^2))$. The first term corresponds to the array merging and histogram creation, and the second term to the processing of histograms.

Therefore, the overall complexity would be $O(k * F) * (\Theta(NS) + O(\#bins * \min(NS, \#bins^2))$.

4.3. CUDA implementation for DISR

As previously explained, the DISR method works similarly to JMI, but the computation of the entropy (H) is also necessary (see Fig. 3). A major advantage is that the entropy, besides having less arithmetic complexity, makes use of one of the structures that are also generated to calculate the MI: the bidimensional histogram. Fig. 9 shows the conceptual differences between the sequential and CUDA implementations of the entropy calculation (compare with Fig. 5 to see the implementation differences with the histogram processing for MI). Since both MI and H are needed for DISR, cuFEAST uses the same bidimensional histogram for the computation of both metrics.

The implementation of DISR is, therefore, almost the same as JMI. The only difference is reflected in the modification of the CUDA kernel, so that the threads compute both MI and H from the created histograms. Apart from that, in order to maintain consistency with the mRMR and JMI kernels, DISR returns only one value per feature (instead both MI and H). This value is $\frac{MI}{H}$, as can be seen in Equation (5). Note that, due to their only slight differences, the complexity of DISR is the same as that of JMI (see Section 4.2.5).

4.4. Adaptation to fixed point

Section 3.4 introduced a variant of the MI calculation for fixed-point data. This idea was presented in [20] for low-end devices (e.g., wearables), but it can potentially help to enhance performance on modern Nvidia GPUs. This approach consists in avoiding the high computational cost of logarithms and floating-point arithmetic by using a look-up table (i.e., memory accesses) and integer arithmetic.

As explained in Section 3.4, the look-up table is key to compute the MI with fixed point, so it must be created beforehand. Nevertheless, the cuFEAST implementation includes the following modifications with respect to the original approach [20], as GPUs have different characteristics to low-end devices:

- **No limit on table size**. Low-end devices have less memory than modern Nvidia GPUs, so cuFEAST can afford to have larger tables and reduce some computation overhead in the precision conversion. Specifically, counter halving and index correction (see [20] for a detailed explanation of these terms) can be suppressed. Note that, due to this change, the table in the GPU will be unidimensional since only one value (the number of samples of the dataset, NS) is used as secondary indexing term (variable j is always equal to NS in Equation (8)). Therefore, the look-up table will have a size of NS elements.
- **Computation instead of precomputation**. The authors of [20] proposed to calculate the look-up table only once for each device,

Table 2
Characteristics of the datasets (size is calculated using 8B per value).

	#Features	#Samples	#Classes	Size
`MNIST`	780	60,000	10	0.36 GB
`Epsilon`	2,000	400,000	2	6.96 GB
`RCV1`	47,236	20,242	2	7.12 GB
`News20`	62,061	15,935	20	7.37 GB

store it on disk and read it prior to each FS execution. However, since the computation of the table is cheap when performed on HPC systems, cuFEAST computes it at the beginning instead of reading a cached copy stored on disk.

- **No fixed-point data types**. The original approach represents the elements of the table with a fixed-point data type. However, such type does not directly exist in the GPU hardware, so the algorithms must be adapted to work with integers. This was achieved by delaying the conversion from integer to fixed point from the table creation to the use of the data. Specifically, Equation (8) is modified as follows:

$$L'(i,j) = \left[\frac{\log(i/j)}{q}\right]_R \tag{10}$$

Then, when needed for the computations, data is converted back to floating point as:

$$d_{float} = L'(i,j) \cdot q \tag{11}$$

- **Adaptive precision distribution**. The original approach is generic for the use of an arbitrary number of bits for both the integer and fractional parts of the number. However, cuFEAST only implements fixed-point algorithms with `uint16` data for the look-up table. This choice was based on the experimental evaluation of [20], which proves that a 16-bit width provides accuracy high enough when compared to high-precision floating-point values, while keeping a low number of bits. Nevertheless, the 16 bits must still be split between the integer and fractional parts. cuFEAST includes a solution consisting of a dynamic approach where the precision is determined by the dataset properties, in order to achieve the best possible results. That is, it chooses a precision with no more than the bits needed to represent the highest possible value in order to leave as many bits as possible for the fractional part.

The computations that use logarithms are executed within the GPU kernels, whose adaptation to fixed point required to make the look-up table available in memory to the CUDA threads. Consequently, it is transferred to the global memory of the device. Then, the fixed-point approach allows to substitute multiplications, divisions and logarithms by additions, subtractions and memory accesses.

5. Experimental evaluation

Four publicly available datasets with different characteristics (summarized in Table 2) have been used for the experimental evaluation. They were all obtained from the LIBSVM collection [8]. `MNIST` and `Epsilon` are multiclass and biclass datasets, respectively, with more samples than features, while `RCV1` (biclass) and `News20` (multiclass) present more features than samples. Since the values of these datasets are continuous a discretization with 128 bins (a quite common scenario) has been applied. Although additional experiments with 256 bins were run, they are not included in this section for simplicity as the conclusions drawn from them are similar.

Two Nvidia GPUs from different generations (Turing T4 and Ampere A100), whose characteristics are shown in Table 3, were used for the experimental evaluation. The GPUs are installed in two CPU systems, Pluton and FinisTerrae III, whose characteristics are listed in Table 4.

Table 3
Characteristics of the GPUs.

	T4	A100
Microarchitecture	Turing	Ampere
Number of SMs	40	108
Number of cores	2560	6912
Core frequency (MHz)	585	765
Global memory size (GB)	16	40
Shared memory size (KB)	64	163
Memory bandwidth (GB/s)	320	1555

Table 4
Characteristics of the systems containing the GPUs.

	Pluton	FinisTerrae III
CPU	2 x Intel Xeon Silver 4216 Cascade Lake-SP	2 x Intel Xeon Ice Lake 8352Y
Cores/CPU	16	32
Total Cores	32	64
RAM	256 GiB	256 GiB
GPU	Nvidia T4	Nvidia A100

These CPUs were also used to execute the sequential and multithreaded counterparts of mRMR, JMI and DISR available in FEAST[4] and ParallelFST [4], respectively, whose performance will be compared to that of cuFEAST. The sequential implementations have also been used to prove the correctness of the numerical results as our CUDA codes select exactly the same features. We can assume that the results are correct as FEAST has been widely used and tested by researchers from different fields of computational science. The output of the low-precision fixed-point approaches has been compared to that of a new sequential C++ implementation developed by the authors following the instructions explained in [20]. This paper includes a detailed accuracy evaluation of the methods with low precision, concluding that the use of 16 bits in FS methods (as in our CUDA fixed-point implementations) is enough to select features with the same quality as double-precision versions.

The number of selected features has been fixed to 200. This number is high enough to show the impact of discarding the selected features from the candidate 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 of the CUDA-based implementations with respect to the sequential counterparts does not depend on the number of selected features as the parallel approach is repeated for every feature selected (see dashed outline in Fig. 2).

The CUDA implementations have been executed with different configurations of batch size, number of streams, and number of threads per block. The speedups shown in the following subsections were calculated using only the runtime of the best configuration, but some insights about the impact of these parameters will be provided in Section 5.3. Also note that all the GPU times shown in this section include the time for data transfers between GPU and CPU, as well as the time needed to create the look-up table in the case of fixed-point versions. We have repeated five times each experiment, discarding the maximum and minimum values, and providing as runtime for each configuration the average of the three intermediate results. This prevents outliers from influencing our evaluation. It means that more than 60,000 executions were performed in this experimental evaluation.

Regarding the fixed-point version of the algorithms, Table 5 shows the fractional bits, sizes and generation times of the look-up tables for each dataset and CPU. Note that generation times are shown in milliseconds and are negligible when compared with FS runtimes.

[4] The latest available version of FEAST (v2.0.0) was used.

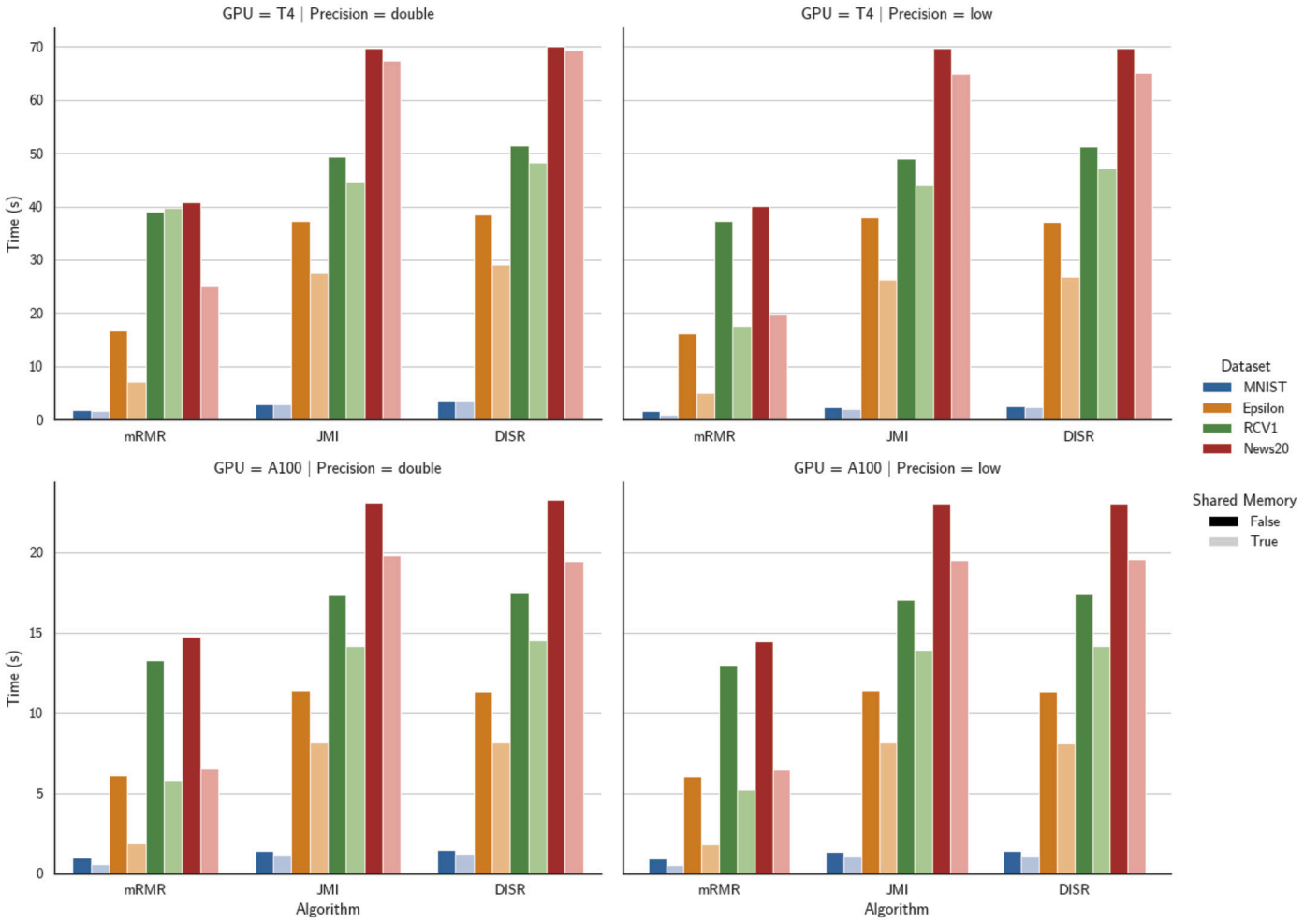


Fig. 10. Comparison of the runtimes with and without exploiting the shared memory of the GPUs. Darker colors are used for versions that do not exploit shared memory.

Table 5
Bit distribution, sizes and generation times of the look-up tables for low-precision fixed-point executions.

Dataset	Fractional bits	Table size (KB)	Time (ms)	
			Cascade Lake-SP	Ice Lake
MNIST	12	117.19	1.70	1.24
Epsilon	11	781.25	10.50	7.25
RCV1	12	39.54	0.66	0.54
News20	12	31.12	0.78	0.46

5.1. Impact of shared memory

The mRMR, JMI and DISR methods have been adapted to exploit the shared memory available in the GPUs, which involved significant modifications to their implementations, as was explained throughout Section 4. The experimental evaluation started by proving that the use of shared memory is beneficial for the MI-based FS methods. Fig. 10 compares the runtimes (in seconds) of the three implementations that only use global memory versus the three that exploit shared memory, for the two GPUs, the four datasets, and the double-precision and fixed-point versions. As mentioned above, the runtimes shown in the graphs are those obtained with the best combination of batch size, number of streams, and number of threads per block for each case.

The main conclusion that can be drawn from these results is that the version with shared memory is clearly beneficial. Its impact on performance heavily depends on the hardware, the method, and the characteristics of the dataset. For instance, on the A100 GPU it is on average 32% faster than the version that only uses global memory, while this percentage drops to 21% for the T4 GPU. The reason is that the A100 has much more shared memory than the T4 (see Table 3). Regarding the methods, the use of shared memory has a greater impact for mRMR (on average, 49% faster than the global memory version), while for JMI and DISR the average benefit remains at 16% and 15%, respectively. This difference can be explained because mRMR can better exploit the memory pools, as its implementation is able to use uninitialized space to fit more features in the batch (see Section 4.1.3). Another conclusion is that the higher the number of samples in the dataset, the higher the benefit of using shared memory. For instance, the average runtime reduction over the global memory version when working with the `Epsilon` dataset (400,000 samples) reaches 41%, while it drops to 20%, 22% and 23% when selecting features of `MNIST`, `RCV1` and `News20`, respectively. Finally, we can also conclude that the precision of the algorithms (double or fixed) does not influence the impact of using shared memory.

5.2. Comparison with the state of the art

Once we have proved that the CUDA implementation that makes use of shared memory obtains the best performance, this section compares it with the state of the art. First, Fig. 11 shows the speedup obtained by this version of the CUDA codes over their sequential counterparts available in FEAST. The speedups for each GPU are calculated related to the sequential time when running the FEAST implementations on

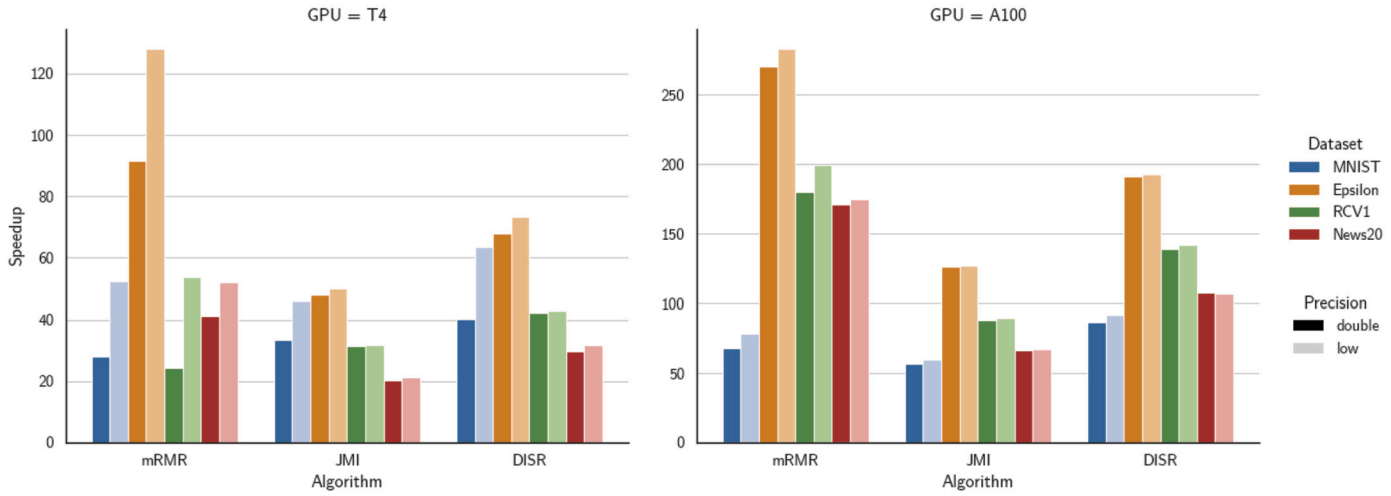


Fig. 11. Speedups obtained by the CUDA implementations (using shared memory) over the sequential C codes.

Table 6
Runtime (in seconds) of the sequential implementations available in FEAST (C version).

		mRMR	JMI	DISR
Pluton (Cascade Lake-SP)	MNIST	44.79	93.62	143.40
	Epsilon	647.35	1317.74	1971.80
	RCV1	943.51	1396.39	2031.61
	News20	1025.16	1372.20	2064.09
FinisTerrae III (Ice Lake)	MNIST	39.37	66.54	104.40
	Epsilon	514.01	1034.83	1560.73
	RCV1	1043.86	1253.14	2010.77
	News20	1129.98	1316.60	2091.47

their corresponding host CPUs (Cascade Lake-SP for the T4 and Ice Lake for the A100). Table 6 shows these sequential runtimes (in seconds) for each CPU, method and dataset. All the experiments for the CUDA version with low-precision fixed point were executed with 16 bits, as suggested in [20].

The results shown in Fig. 11 prove that our CUDA implementations are significantly faster than the original sequential codes in all scenarios. The magnitude of the acceleration depends on several factors:

- **The GPU hardware**. The speedups achieved on the A100 (132x on average) are always higher than on the T4 (48x on average). This proves that the CUDA implementations efficiently exploit the resources available in modern GPUs. As can be seen in Table 3, the A100 not only includes more cores, but also these cores and the memory transfers are faster (higher memory bandwidth).
- **The FS method**. Although mRMR is the fastest method even in its sequential implementation, it is the one that better adapts to the GPU, obtaining an average speedup of 118x (with a 283x peak for the `Epsilon` dataset on the A100). Nevertheless, DISR and JMI are also very suitable for GPUs, reaching speedups of up to 193x and 127x, respectively.
- **The input dataset**. The highest speedups are achieved for `Epsilon` (137x on average), the dataset with the largest number of samples. The behavior of `RCV1` and `News20` is quite similar, with average speedups of 89x and 74x, respectively, while `MNIST` obtains the worst results (average speedup of 59x). The reason is not a poor implementation of the methods but the low computational requirements for the analysis of this dataset (it is the smallest one). Note that all CUDA implementations complete the FS on `MNIST` in less than three seconds.
- **The precision**. The low-precision fixed-point versions of the algorithms are faster than their double-precision counterparts in 94% of the experiments. The low-precision version achieves an average improvement in runtime of 23%, 6% and 8% for mRMR, JMI and DISR, respectively. Note that the runtime improvement is not impressive, since this optimization not only involves changing the data type, but also introduces suboptimal memory accesses (random reads). Furthermore, the optimization only applies to a limited section of the program, so the speedup of the entire algorithm is less than that of the optimization itself.

Summarizing, the CUDA implementations presented in this paper efficiently exploit the GPU hardware and are able to obtain speedups of up to 283x and 128x when executing the mRMR method with 16-bit fixed point over the `Epsilon` dataset on the A100 and T4 GPUs, respectively. Regarding the runtime, the FS was completed in less than 20 seconds for any method and dataset on the A100, and in around one minute in the worst case for the T4, which is a huge reduction compared to sequential runtimes of up to 35 minutes.

We have compared the performance of the GPU versions in cuFEAST with the multithreaded versions available in our Parallel-FST library [4]. The number of cores available varies by system and the multithreaded versions were run with one thread per core (32 threads in Pluton and 64 threads in FinisTerrae III, according to Table 4). The speedups of the cuFEAST and Parallel-FST versions with respect to the sequential version on the corresponding host system are shown in Fig. 12. Although the multithreaded versions achieve high parallel efficiency, the further optimizations present in cuFEAST make the GPU versions achieve far superior speedups.

As mentioned in Section 2, CUDA-JMI [16] is, up to our knowledge, the only available implementation in the state of the art for FS on GPUs. Table 7 compares the runtime (in seconds) of CUDA-JMI and our novel JMI implementation on the two GPUs for the four datasets, only for the double-precision version, as CUDA-JMI is not able to work with low-precision (16 bits) fixed point. First, note that CUDA-JMI failed when working with the `MNIST` dataset due to an inefficient use of memory that makes it run out of memory. For the other datasets our implementation is, on average, around 11 and 41 times faster on the T4 and A100, respectively, proving that it is better suited to the most modern and powerful GPUs. This runtime improvement is due to the exploitation of the GPU: while CUDA-JMI just computes the MI, cuFEAST tries to maximize the use of the GPU by also offloading the merge arrays stage. Moreover, cuFEAST includes optimization techniques (those explained in Section 4.2) that are not present in CUDA-JMI.

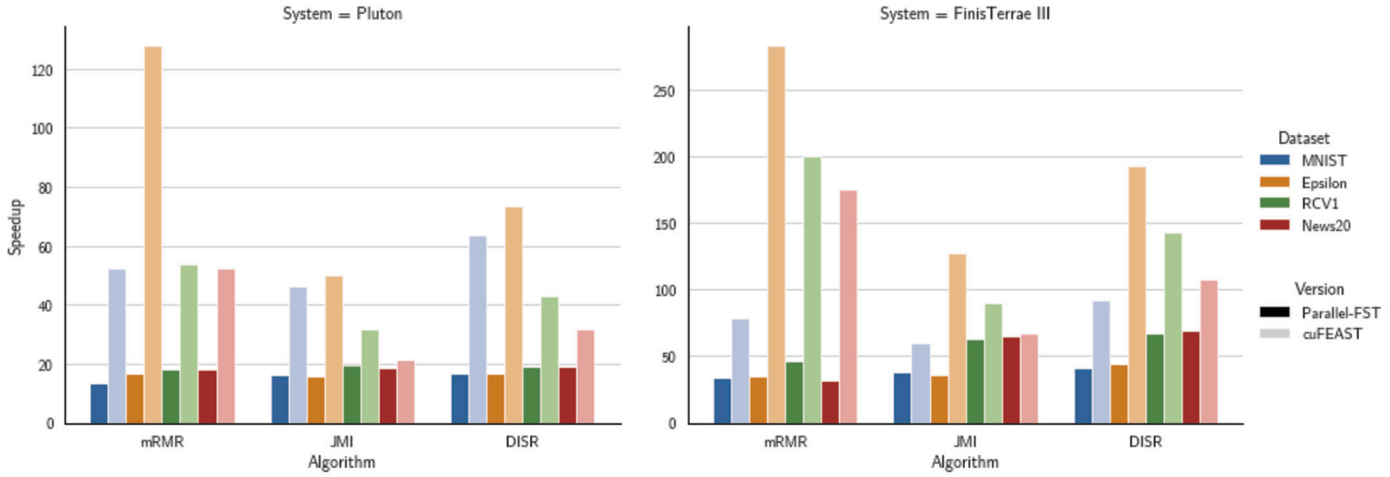


Fig. 12. Comparison of the best speedups obtained by the CUDA implementations (cuFEAST) and the multithreaded versions (Parallel-FST) over the sequential C codes.

Table 7
Runtime (in seconds) of CUDA-JMI and our novel double-precision implementation of JMI ("-" is a failed out-of-memory execution).

Dataset	GPU	Time (s)	
		CUDA-JMI	cuFEAST JMI
MNIST	T4	–	2.80
	A100	–	1.17
Epsilon	T4	528.24	27.55
	A100	475.47	8.19
RCV1	T4	366.73	44.70
	A100	531.64	14.187
News20	T4	378.50	67.38
	A100	545.56	19.77

5.3. Selection of the configuration parameters

As previously explained, the experimental evaluation carried out for this work has been very extensive, with around 60,000 executions, testing different configurations of batch size, number of streams and number of threads per block. However, all the runtimes presented up to now are the best ones for each scenario (i.e., using the best configuration). In this section we provide some insight about which are these best configurations.

5.3.1. Batch size and streams

The first version of our CUDA FS code introduced the configuration parameter of "block size", that is, the number of threads that work in parallel in a block of a CUDA kernel (remember that just one block per feature is launched). However, due to the amount of configuration parameters and variability in test executions (distinct datasets, microarchitectures, etc.), it is difficult to isolate the impact of the block size. Therefore, we have initially tried to find a good configuration for the batch size and number of streams, and then, select the block size that works best for that case. We introduce the "normalized speedup" metric to provide a fair comparison between configurations, calculated as follows:

1. The speedup of each tested configuration is calculated with respect to the sequential implementation available in FEAST as $S_{cfg} = \frac{time_{FEAST}}{time_{cfg}}$.
2. Speedups are grouped by GPU type, dataset, algorithm and precision.
3. Speedups are normalized within each group. That is, for a speedup of a given configuration S_{cfg} in a group g, we compute the normalized speedup metric as $S'_{cfg} = (S_{cfg} - mean_g)/std_g$. This transforms each speedup in a value that can be interpreted as how well a configuration performs compared to the others in the same group, while maintaining that value in a common range among all groups.

Figs. 13 and 14 show the normalized speedups for configurations of different batch sizes and number of streams with double precision and low-precision fixed point, respectively. Within each figure, the results are also separated by GPU and algorithm. Note that each square of the heatmap (configuration with a specific batch size and number of streams) aggregates the average normalized speedups for the non-mentioned parameters (i.e., dataset and block size). The figures clearly show the negative speedup correlation between using a high/low number of streams and a low/high batch size. In order to understand this correlation, we will examine the advantages and drawbacks of the cases in each sector of the heatmaps:

- **Top-left**: many streams, small batches. The code is capable of achieving a good level of asynchronism since there are lots of streams and low workload for each batch. Furthermore, the amount of memory that needs to be allocated for the pools is not excessive.
- **Top-right**: many streams, large batches. The amount of memory required for the pools is so high that the overhead it introduces highly impacts runtime.
- **Bottom-left**: few streams, small batches. The amount of memory required for the pools is really low, but there is a high amount of kernel launches which introduces some overhead that influences the overall runtime.
- **Bottom-right**: few streams, large batches. There are fewer kernel launches and fewer streams to manage, so the introduced overhead is minimum. However, due to this, the code cannot achieve a good level of asynchronism. The memory required for the pools is not excessive.

Finally, after examining these results, we can conclude that a good configuration is a batch size of 256 features and asynchronism with two streams. It is recommended as it is a good configuration overall and, in some cases, the best one.

5.3.2. Block size

Now that the best configuration for the other parameters is known, the impact on performance of the CUDA block size can be measured. The number of threads that work concurrently in a CUDA block is rel-

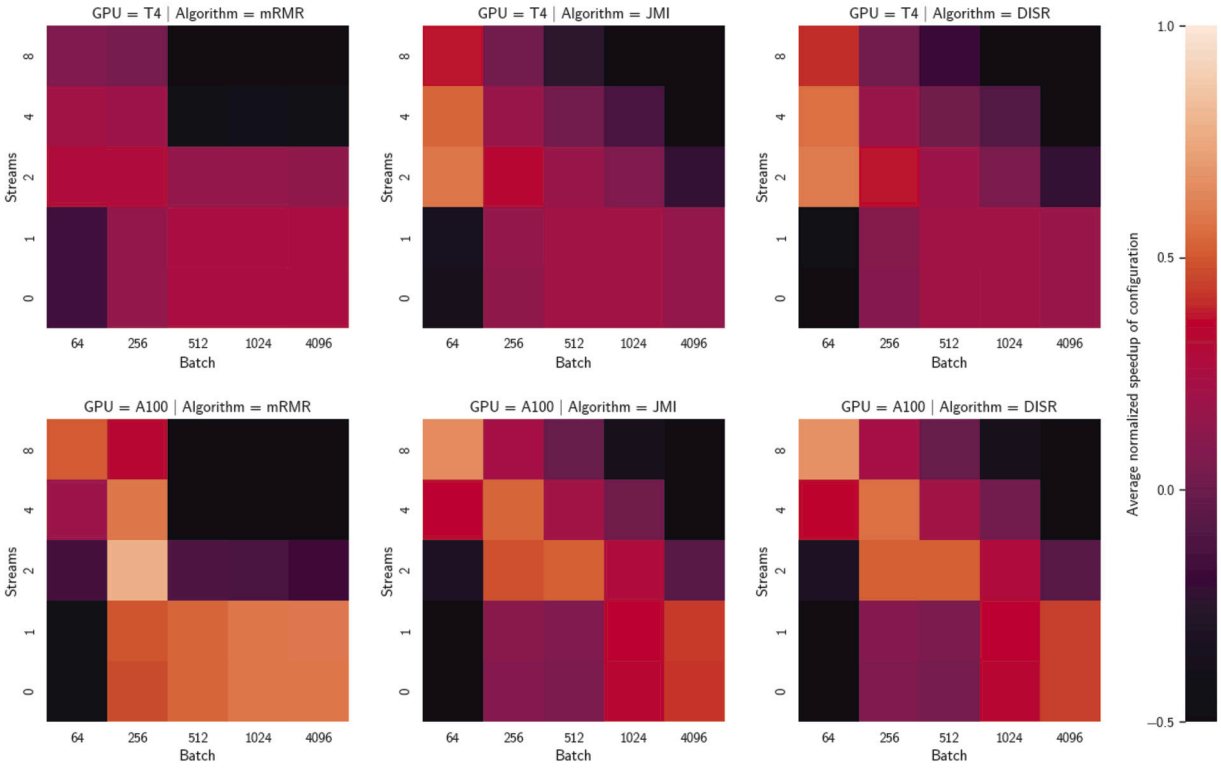


Fig. 13. Comparison of the average normalized speedups obtained for different combinations of batch sizes and number of CUDA streams (double-precision implementation).

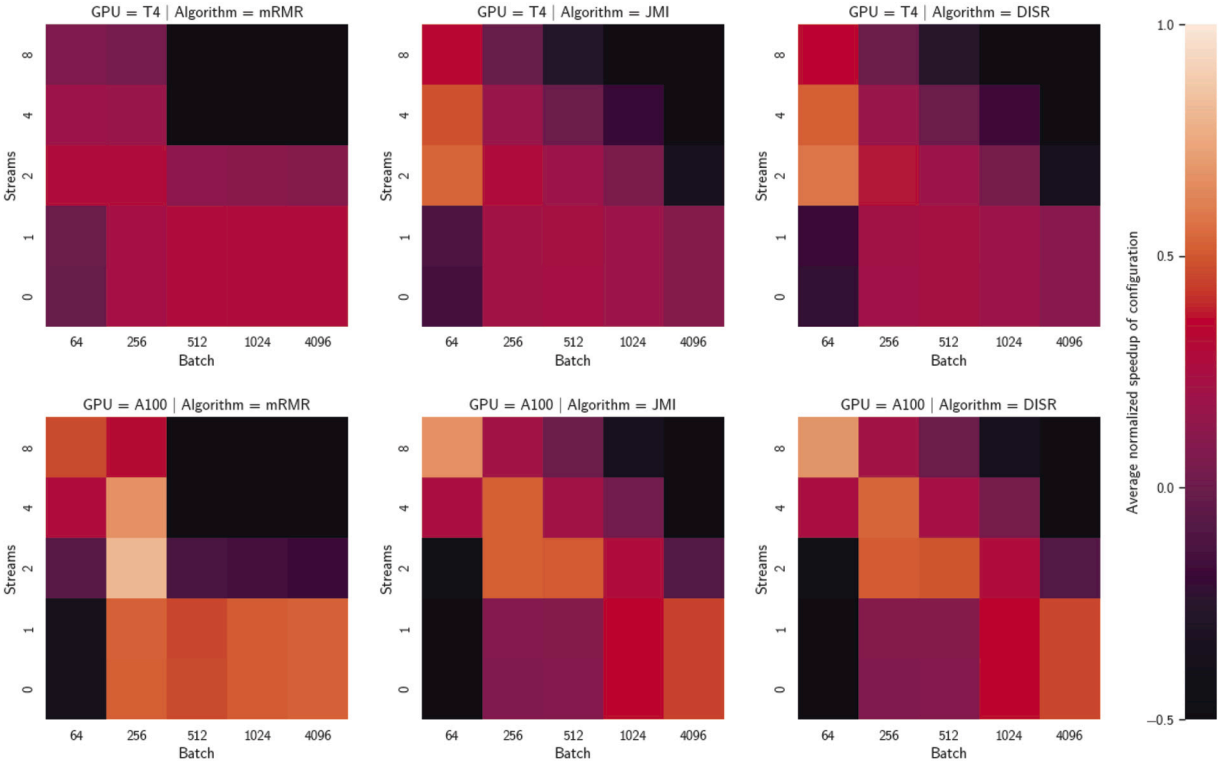


Fig. 14. Comparison of the average normalized speedups obtained for different combinations of batch sizes and number of CUDA streams (fixed-point implementation).

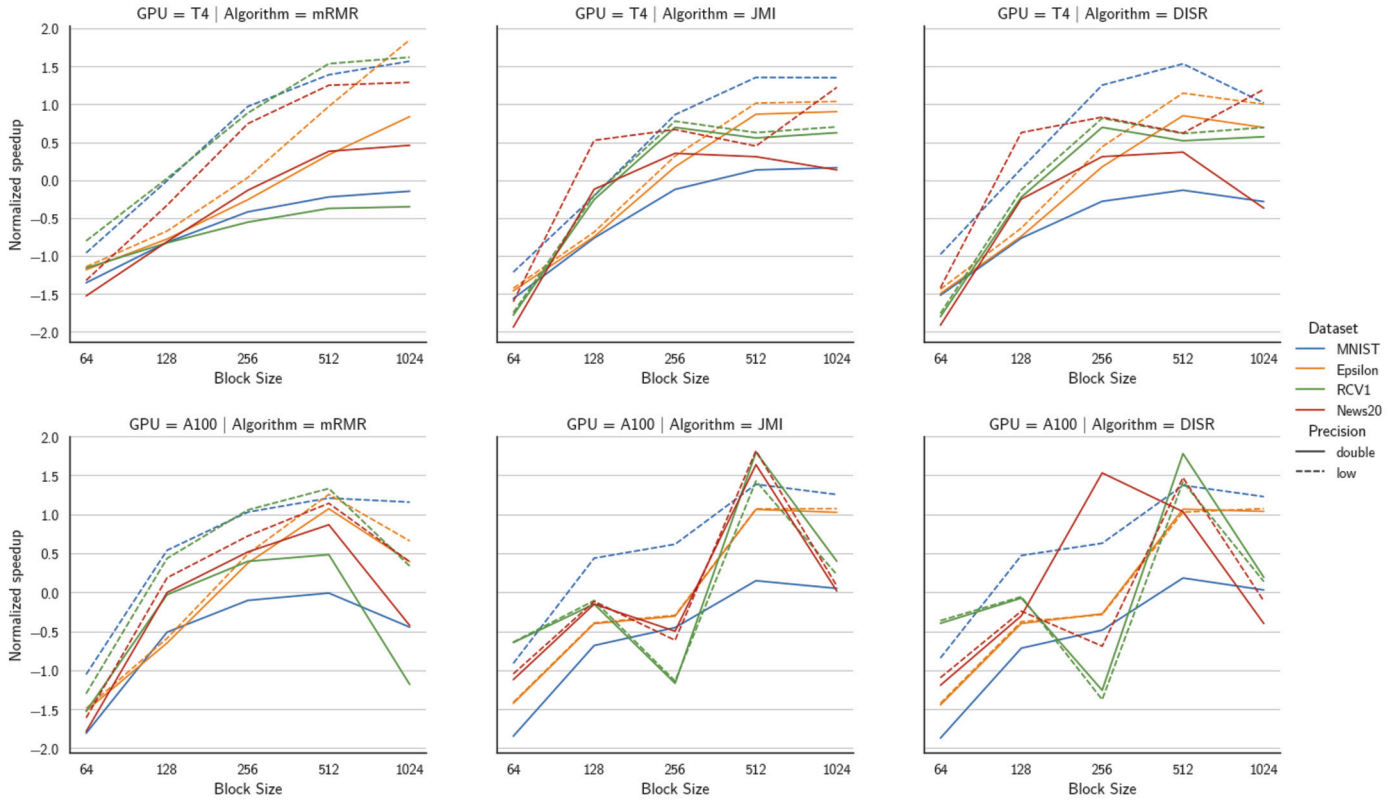


Fig. 15. Comparison of the normalized speedups for different block sizes (number of threads per CUDA block).

evant since each block runs in a physical Streaming Multiprocessor (with its own shared memory), so its hardware is shared among the block threads. Therefore, on the one hand, codes with threads that require many resources would be able to achieve more parallelism with a smaller block size, while a larger block would cause some threads to stall waiting for resources. On the other hand, in codes with more lightweight threads, a larger block size would be able to achieve more parallelism, while a smaller one would cause some hardware to be idle.

The impact of the block size is depicted in Fig. 15. The same as for the batch and streams, the graphs show normalized speedups. However, the results presented correspond to the best configuration of batch size and number of streams obtained in the previous step, so now each line shows the normalized speedup for each case instead of an aggregation for several parameters.

The results show that, in general, speedups increase with the block size. Specifically, the maximum block size of 1024 threads achieves the best speedup in some cases. However, this configuration can cause performance drops in some other cases, so a better general recommendation is to use a block size of 512 threads.

5.3.3. General recommendations

It should be noted that the configuration that achieves the best performance would largely depend on the properties of the dataset, as well as on the distribution of the values that each feature can take. Therefore, we can recommend the configurations shown in Table 8 based on their overall performance achieved in the experimental evaluation (i.e., they might not get the best speedup for a specific case, but work well in general).

Table 8
Recommended configurations with overall good performance for each microarchitecture and algorithm.

Microarch.	Algorithm	Block	Batch	Streams
Turing	mRMR	1024	64	2
	JMI	1024	64	2
	DISR	512	64	2
Ampere	mRMR	512	256	2
	JMI	512	64	8
	DISR	512	64	8

6. Conclusion

Feature Selection is nowadays a common and extremely important step in Machine Learning, especially with the continuous increase in the average size of datasets from different fields such as text mining, genetics or bioinformatics. Among the many existing methods for FS, those based on MI are widely employed. However, FS procedures require long computation times for large datasets due to their quadratic complexity with the number of features.

This work has presented CUDA implementations for three MI-based FS methods: mRMR, JMI and DISR. Two CUDA versions were developed for each method: one that works with data in double precision and another one that uses fixed point and low precision, thus reducing the computational requirements at the cost of a slightly lower accuracy [20]. All codes are highly optimized with the use of shared memory, asynchronism through streams and a custom memory pool. The extensive experimental evaluation with two Nvidia GPUs, four datasets, and multiple algorithm configurations (a total of about 60,000 executions) have proved that our implementations are able to efficiently exploit the computing resources of modern and powerful GPUs. For instance, all the experiments (even for double precision) finished in less than 20 seconds on an Ampere A100 GPU, obtaining speedups of up to 283x compared to the popular sequential C implementations included in the FEAST library. The versions of the algorithms that work with fixed-point data are also beneficial in terms of performance, since they are on average 12% faster than their double-precision counterparts. All the CUDA implementations described in this work are publicly available at https://gitlab.com/bieito/parallel-fst under a 3-Clause BSD license (open source).

As future work we aim to extend our CUDA approach to multi-GPU platforms, as well as to other FS methods with the goal of providing a broad suite where potential users can choose the parallel algorithms that best fit their data and analyses. We will also try to design an "autotuning" system that automatically selects the best configuration parameters according to the characteristics of the GPU and the input dataset.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by grants PID2019-104184RB-I00, PID2019-109238GB-C22, TED2021-130599A-I00 and PID2022-136435NB-I00, funded by MCIN/AEI/ 10.13039/501100011033 (TED2021 also funded by "NextGenerationEU"/PRTR and PID2022 by "ERDF A way of making Europe", EU). Grant TSI-100925-2023-1, funded by Ministry for Digital Transformation and Civil Service. FPU predoctoral grant of Bieito Beceiro ref. FPU20/00997, funded by the Ministry of Science, Innovation and Universities. We gratefully thank the Galician Supercomputing Center (CESGA) for the access granted to its supercomputing resources. Funding for open access charge: Universidade da Coruña/CISUG.

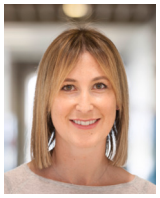
References

[1] H. Alshamlan, G. Badr, Y. Alohali, mRMR-ABC: a hybrid gene selection algorithm for cancer classification using microarray gene expression profiling, BioMed Res. Int. 2015 (2015) 604910.
[2] F. Azmandian, A. Yilmazer, J.G. Dy, J.A. Aslam, D.R. Kaeli, GPU-accelerated feature selection for outlier detection using the local kernel density ratio, in: IEEE 12th International Conference on Data Mining, 2012, pp. 51–60.
[3] N. Barraza, S. Moro, M. Ferreyra, A. de la Peña, Mutual information and sensitivity analysis for feature selection in customer targeting: a comparative study, J. Inf. Sci. 45 (1) (2019) 53–67.
[4] B. Beceiro, J. González-Domínguez, J. Touriño, Parallel-FST: a feature selection library for multicore clusters, J. Parallel Distrib. Comput. 169 (2022) 106–116.
[5] V. Bolón-Canedo, N. Sánchez-Maroño, A. Alonso-Betanzos, A review of feature selection methods on synthetic data, Knowl. Inf. Syst. 34 (3) (2013) 483–519.
[6] N.M. Braman, M. Etesami, P. Prasanna, C. Dubchuk, H. Gilmore, P. Tiwari, D. Plecha, A. Madabhushi, Intratumoral and peritumoral radiomics for the pretreatment prediction of pathological complete response to neoadjuvant chemotherapy based on breast DCE-MRI, Breast Cancer Res. 19 (1) (2017) 1–14.
[7] G. Brown, A. Pocock, M.-J. Zhao, M. Luján, Conditional likelihood maximisation: a unifying framework for information theoretic feature selection, J. Mach. Learn. Res. 13 (2012) 27–66.
[8] C.-C. Chang, C.-J. Linn, LIBSVM: a library for support vector machines, ACM Trans. Intell. Syst. Technol. 2 (3) (2011) 27.
[9] H.-H. Chang, C.-Y. Li, An automatic restoration framework based on GPU-accelerated collateral filtering in brain MR images, BMC Med. Imaging 19 (1) (2019) 1–13.
[10] 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 Comput. 22 (5) (2018) 1525–1532.
[11] A. Das, N. Borisov, M. Caesar, Tracking mobile web users through motion sensors: attacks and defenses, in: 23rd Annual Network and Distributed System Security Symposium, 2016, pp. 282–296.
[12] 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, Clust. Comput. 20 (3) (2017) 1881–1897.
[13] H. Estiri, Z.H. Strasser, J.G. Klann, P. Naseri, K.B. Wagholikar, S.N. Murphy, Predicting COVID-19 mortality with electronic medical records, npj Digit. Med. 4 (1) (2021) 1–10.
[14] P. Fazendeiro, C. Padole, P. Sequeira, P. Prata, OpenCL implementations of a genetic algorithm for feature selection in periocular biometric recognition, in: 3rd International Conference on Swarm, Evolutionary, and Memetic Computing, 2012, pp. 729–737.
[15] J. González-Domínguez, V. Bolón-Canedo, B. Freire, J. Touriño, Parallel feature selection for distributed-memory clusters, Inf. Sci. 496 (2019) 399–409.
[16] J. González-Domínguez, R.R. Expósito, V. Bolón-Canedo, CUDA-JMI: acceleration of feature selection on heterogeneous systems, Future Gener. Comput. Syst. 102 (2020) 426–436.
[17] I. Guyon, S. Gunn, M. Nikravesh, L.A. Zadeh, Feature Extraction: Foundations and Applications, Springer, 2006.
[18] F. Jalali-Najafabadi, M. Stadler, N. Dand, D. Jadon, M. Soomro, P. Ho, H. Marzo-Ortega, P. Helliwell, E. Korendowych, M.A. Simpson, et al., Application of information theoretic feature selection and machine learning methods for the development of genetic risk prediction models, Sci. Rep. 11 (1) (2021) 1–14.
[19] P.E. Meyer, G. Bontempi, On the use of variable complementarity for feature selection in cancer classification, in: Workshops on Applications of Evolutionary Computation, 2006, pp. 91–102.
[20] L. Morán-Fernández, K. Sechidis, V. Bolón-Canedo, A. Alonso-Betanzos, G. Brown, Feature selection with limited bit depth mutual information for portable embedded systems, Knowl.-Based Syst. 197 (2020) 105885.
[21] R.-J. Palma-Mendoza, L. de Marcos, D. Rodriguez, A. Alonso-Betanzos, Distributed correlation-based feature selection in Spark, Inf. Sci. 496 (2019) 287–299.
[22] H. Peng, F. Long, C. Ding, Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy, IEEE Trans. Pattern Anal. Mach. Intell. 27 (8) (2005) 1226–1238.
[23] 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, Int. J. Intell. Syst. 32 (2) (2017) 134–152.
[24] A. Samat, E. Li, P. Du, S. Liu, J. Xia, 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 (2021) 3200–3214.
[25] R. Shams, N. Barnes, 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, 2007, pp. 555–560.
[26] M. Soheili, A.M. Eftekhari-Moghadam, DQPFS: distributed quadratic programming based feature selection for big data, J. Parallel Distrib. Comput. 138 (2020) 1–14.
[27] 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) e0117988.
[28] L. Venkataramana, S.G. Jacob, R. Ramadoss, A parallel multilevel feature selection algorithm for improved cancer classification, J. Parallel Distrib. Comput. 138 (2020) 78–98.
[29] B. Venkatesh, J. Anuradha, Fuzzy rank based parallel online feature selection method using multiple sliding windows, Open Comput. Sci. 11 (1) (2021) 275–287.
[30] Z. Yan, Z. Wang, H. Xie, The application of mutual information-based feature selection and fuzzy LS-SVM-based classifier in motion classification, Comput. Methods Programs Biomed. 90 (3) (2008) 275–284.
[31] H.H. Yang, J.E. Moody, Data visualization and feature selection: new algorithms for nongaussian data, in: Advances in Neural Information Processing Systems, vol. 12, 1999, pp. 687–693.
[32] Z. Zhao, R. Anand, M. Wang, Maximum relevance and minimum redundancy feature selection methods for a marketing machine learning platform, in: IEEE International Conference on Data Science and Advanced Analytics, 2019, pp. 442–452.
[33] H. Zhu, Y. Wu, P. Li, P. Zhang, Z. Ji, M. Gong, An OpenCL-accelerated parallel immunodominance clone selection algorithm for feature selection, Concurr. Comput., Pract. Exp. 29 (9) (2017) e3838.

Bieito Beceiro received the B.S. in computer science and the M.S. in High Performance Computing (HPC) from the Universidade da Coruña (UDC), Spain, in 2020 and 2021, respectively. He is currently a Ph.D. student at the Computer Architecture Group of the UDC. His work is focused on the acceleration of machine learning methods for computational science using HPC techniques.

Jorge González-Domínguez received the B.S., M.S., and Ph.D. degrees in computer science from the Universidade da Coruña (UDC), Spain, in 2008, 2009, and 2013, respectively. He is currently an Associate Professor with the Department of Computer Engineering, UDC. His main research interests include the development of parallel applications on multiple fields, such as bioinformatics, data mining, and machine learning, focused on different architectures (multicore systems, GPUs, clusters, and so on).

Laura Morán-Fernández received her B.S. (2015) and Ph.D. (2020) degrees in Computer Science from the Universidade da Coruña (Spain). She is currently an Assistant Lecturer in the Department of Computer Science of the Universidade da Coruña. She received the Frances Allen Award (2021) from the Spanish Association of Artificial Intelligence (AEPIA). Her research interests include machine learning, feature selection and big data. She has co-authored four book chapters, and more than 15 research papers in international journals and conferences.



Verónica Bolón-Canedo received her B.S. (2009), M.S. (2010) and Ph.D. (2014) degrees in Computer Science from the Universidade da Coruña (Spain). After a postdoctoral fellowship in the University of Manchester, UK (2015), she is currently an Associate Professor in the Department of Computer Science of the Universidade da Coruña. Her main current areas are machine learning and feature selection. She is co-author of more than 100 papers on these topics in international conferences and journals.

Juan Touriño is a Full Professor with the Department of Computer Engineering, Universidade da Coruña, where he also leads the Computer Architecture Group. He has extensively published in the area of High Performance Computing (HPC): HPC & AI convergence, programming languages and compilers for HPC, high-performance architectures and networks, parallel algorithms and applications in computational science and engineering. He is coauthor of more than 170 papers on these topics in international conferences and journals.