

Lawrence Berkeley National Laboratory

LBL Publications

Title

A multi-backend autotuning study of feature selection on GPUs

Permalink

<https://escholarship.org/uc/item/06p8d0jz>

Journal

The Journal of Supercomputing, 82(13)

ISSN

0920-8542

Authors

Beceiro, Bieito
González-Domínguez, Jorge
Diéguez, Adrián P
[et al.](#)

Publication Date

2026-08-01

DOI

10.1007/s11227-026-08770-5

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at
<https://creativecommons.org/licenses/by/4.0/>

Peer reviewed



A multi-backend autotuning study of feature selection on GPUs

Bieito Beceiro¹ · Jorge González-Domínguez¹ · Adrián P. Diéguez² · Khaled Z. Ibrahim² · Juan Touriño¹

Received: 29 April 2025 / Accepted: 29 July 2026
© The Author(s) 2026

Abstract

Feature selection is an important step in machine learning that can benefit from GPU acceleration. As the number of GPU vendors increases, it is imperative to adapt algorithms such as the minimum Redundancy Maximum Relevance (mRMR) feature selection method to different backends that support several GPU architectures. This work presents a multi-backend implementation of mRMR across CUDA, HIP, and SYCL, and studies its performance when combined with Bayesian optimization and transfer learning to automatically tune execution parameters for different platforms and datasets. Our experimental results show that when tuned, CUDA and HIP achieve comparable performance on NVIDIA architectures, while SYCL exhibits a moderate performance gap. Overall, this work highlights the impact of backend choice and autotuning on GPU-accelerated feature selection and provides insights into deploying mRMR across heterogeneous environments.

Keywords GPU · Feature selection · mRMR · CUDA · HIP · SYCL

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

Jorge González-Domínguez
jgonzalezd@udc.es

Adrián P. Diéguez
aperezdieguez@lbl.gov

Khaled Z. Ibrahim
kzibrahim@lbl.gov

Juan Touriño
juan@udc.es

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

² Applied Mathematics & Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, 94720 Berkeley, California, USA

1 Introduction

In recent years, the demand for efficient high-performance computing (HPC) has surged due to the increasing use of data-intensive applications. Among the various HPC architectures, GPUs can effectively deliver impressive performance for massively parallel problems and have become increasingly powerful with relatively low power consumption. Machine learning (ML) and data analysis workflows are among the most computationally demanding fields today, and many algorithms leverage GPUs to accelerate their execution. For example, feature selection (FS) [1], a key preprocessing technique in many classical ML and data analysis pipelines used to reduce dataset dimensionality, is a good candidate for GPU acceleration due to its high computational intensity, involving highly demanding calculations on large datasets.

Among the various methods for FS [2], one of the most popular is the minimum Redundancy Maximum Relevance (mRMR) [3], which balances high relevance with low redundancy. In practice, mRMR has been applied across diverse domains, including large-scale marketing and personalization systems, industrial process monitoring, and hyperspectral image analysis, where reducing redundancy and dimensionality is critical for both performance and interpretability [4, 5]. Although there is an earlier version of mRMR that can take advantage of GPU hardware [6], it was designed only for NVIDIA's Compute Unified Device Architecture (CUDA¹) framework. CUDA is the default choice for developers targeting NVIDIA GPUs [7] due to its maturity, extensive library support, and optimizations for NVIDIA hardware, but its usability is limited because it cannot run on alternative hardware platforms.

To address the need for cross-platform GPU programming, AMD developed the Heterogeneous-Compute Interface for Portability (HIP²), a framework designed to enable code portability across NVIDIA and AMD GPUs. They have also developed HIPify,³ a tool that automatically adapts CUDA source code to HIP, extending its applicability to more hardware platforms. HIP has become more attractive due to AMD's increasing market share in the HPC sector and the growing demand for a CUDA alternative that is compatible with AMD's ROCm⁴ ecosystem. Despite its potential, the HIP ecosystem is still maturing and performance on non-AMD GPUs can vary, making it a critical focus when comparing GPU frameworks.

In addition, the Khronos Group introduced SYCL,⁵ a single-source, C++-based framework that aims to unify the programming experience for heterogeneous computing by supporting a wide range of processors, including GPUs, CPUs, and FPGAs. As an open standard, SYCL is designed to support multiple vendors and architectures, providing a high degree of portability. With recent advances, such as the development of oneAPI's Data Parallel C++ (DPC++), SYCL has gained relevance in the HPC and ML communities for workloads that require flexibility across different hardware

¹ <https://docs.nvidia.com/cuda/>

² <https://rocm.docs.amd.com/projects/HIP>

³ <https://rocm.docs.amd.com/projects/HIPIFY>

⁴ <https://rocm.docs.amd.com/en/latest/>

⁵ <https://www.khronos.org/sycl/>

environments [8]. However, the performance of SYCL in computationally demanding applications such as FS remains relatively underexplored.

The main goal of this work is to develop multiple backend implementations of mRMR that extend the usability of this FS method across different GPU architectures. The novel implementations include optimizations tailored to CUDA, HIP, and SYCL, achieving high performance across different hardware platforms. In our work, we combine Bayesian Optimization (BO) and Transfer Learning (TL) for selecting the best configuration parameters (adapted to the dataset, problem size, and characteristics of the underlying hardware).

Performance portability has been defined in different ways in the literature, often requiring a single implementation to achieve near-peak performance across multiple architectures. In practice, however, achieving such behavior is challenging due to architectural differences and backend-specific optimizations. In this work, we adopt a pragmatic view of performance portability, focusing on the ability to obtain high performance across platforms with limited tuning effort. This reflects real-world scenarios in which autotuning and backend specialization are commonly used to bridge performance gaps between architectures.

In summary, these are the main contributions of this work:

- Novel versions of the mRMR FS method using HIP and SYCL that can run on non-NVIDIA GPUs, highlighting the implementation particularities of each programming model. They are open source and publicly available⁶ so that the scientific community can use them to accelerate their analyses. We also release further optimizations over our previous CUDA implementation for the mRMR algorithm [6].
- An autotuning study based on BO and TL to efficiently identify high-performance configurations across datasets, backends, and platforms.
- A performance comparison for equivalent CUDA, HIP, and SYCL mRMR implementations on different NVIDIA GPU architectures, providing insight into selecting the optimal programming model for GPU-based FS tasks.
- An exploratory cross-platform evaluation of the HIP backend, illustrating how the same implementation behaves across different GPU architectures under a fixed experimental setup.

The remainder of the manuscript is organized as follows: Sect. 2 reviews previous research on GPU-accelerated FS methods, as well as on comparative analyses of CUDA, HIP, and SYCL. Section 3 provides an overview of key concepts, including FS and cross-platform GPU compatibility. The implementation details and framework-specific optimizations for GPU-based mRMR are described in Sect. 4. The experimental results evaluating the performance of the implementations in the setup described in Sect. 5 with the performance portability search detailed in Sect. 6 are presented in Sect. 7. Finally, Sect. 8 summarizes the findings and highlights avenues for future research.

⁶ <https://gitlab.com/bieito/gpufs>

2 Related work

As previously mentioned, mRMR has already been parallelized for NVIDIA GPUs using CUDA [6] and is publicly available (along with other FS methods) as part of the Parallel-FST library.⁷ Although there have been previous efforts to exploit GPUs to accelerate other FS methods (e.g., [9–11]), none of them are based on either HIP or SYCL for cross-platform execution. This makes our study of great interest to the scientific community.

Regarding performance comparisons between CUDA, HIP, and SYCL, several works have tested their performance using a variety of workloads and applications on different types of GPUs. Many authors have focused on bioinformatics applications due to their high computational cost, which makes GPUs a good choice for accelerating their analyses. One example is ADEPT [12], a tool for DNA and protein sequence alignment. This tool was originally developed in CUDA and designed for NVIDIA GPUs, but was later translated to HIP and SYCL to evaluate its portability [13]. The translation to SYCL proved to be significantly more challenging than the translation to HIP. In terms of performance, the HIP version matches the speed of the CUDA version, while the SYCL implementation is about twice as slow. Another recent study focused on an open-source bioinformatics kernel for all-pairs distance calculation [14], which implements GPU kernels optimized for memory accesses, reduced memory footprints, and effective use of the memory hierarchy on NVIDIA and AMD GPUs. The authors analyzed the GPU instructions generated by the compilers to understand the code optimizations and performance gaps. They concluded that CUDA and HIP kernels outperform SYCL kernels due to factors such as memory address sizes, memory access widths, and sub-word accesses.

A more recent paper [15] completed a very interesting study of NVIDIA, Intel, and AMD GPUs on exascale supercomputers. Their workload performs a massively parallel blood flow (hemodynamic) simulation based on the lattice Boltzmann method for fluid dynamics. The authors found that the three programming models achieved high performance on these GPUs. However, SYCL development required more effort to implement efficient code, while an almost direct translation from CUDA to HIP was sufficient to achieve high performance. Other works have compared CUDA with SYCL only [16, 17], excluding HIP from the analysis.

Although comparing the performance of CUDA, HIP, and SYCL on complete applications is interesting for the HPC community, the conclusions may be closely tied to a specific domain. Another approach is to focus on common kernels or specific functions that are present in different types of applications. An example is the test suite in [18], which uses tensor contractions for high-dimensional general matrix multiplications. Specifically, they compare the original CUDA version generated by COGENT [19], a CUDA generator for tensor contractions, with automatic translations to HIP and SYCL on NVIDIA and AMD GPUs, highlighting that CUDA and HIP outperform SYCL. The authors identified occupancy and data movement resulting from the use of local memory as the primary factors contributing to this performance difference. In addition, another work [20] focused on stencil computations, which are

⁷ <https://gitlab.com/bieito/parallel-fst>

also common operations across several domains. The authors used BrickLib [21], a library that can generate code in different languages, to compare the performance of CUDA, HIP, and SYCL codes over different data layouts, stencil shapes, problem sizes, and GPUs (including NVIDIA, AMD, and Intel). The main conclusion of this work is that CUDA and HIP showed significantly better performance than SYCL.

While CUDA, HIP, and SYCL require tuning for optimized performance, the best tuning values depend on the programming model, execution model, and compiler efficiency, thus needing specialized tuning strategies. Compared to manual or grid-based tuning approaches, BO-based autotuning has become popular for HPC tuning [22–26], modeling the performance objective function as a probabilistic surrogate and leading to optimal parameter selection with minimal configuration evaluations. This black-box approach is very well suited for tuning complex systems where performance is influenced by multiple interdependent parameters and hardware-specific features, such as CUDA, HIP, and SYCL applications running on different platforms. Specifically, we use our multi-platform tuning tool DFTuning [27] to perform the complex autotuning search of this work. This tool uses GPTune [28] as its underlying BO engine to optimize the execution across different datasets, backends, and hardware platforms by sharing tuning information (through TL) for a predefined set of performance parameters.

Our work contributes to the scientific community by providing a multi-backend implementation of mRMR, which can leverage multiple HPC hardware platforms thanks to state-of-the-art autotuners. In addition, to the best of our knowledge, this study presents the first performance analysis of CUDA, HIP, and SYCL for a FS algorithm.

3 Background

This section describes several concepts that will be necessary to understand the implementations developed as part of this work.

3.1 Feature selection

FS is a technique that finds the best subset of features for a given dataset. Depending on the data analysis step in which FS is applied, we can identify three different approaches:

- Filters, where FS is applied as an independent step before training an ML model. The dimensionality of the dataset is reduced with FS so that the training step runs faster.
- Wrapper approaches, where the FS and the ML model are tightly coupled. The subset of selected features is built iteratively, using the model to evaluate its goodness.
- Embedded approaches that apply FS as an inherent part of the model construction. For example, FS can decide which feature to use as a splitting condition at a given node when building a decision tree.

The mRMR algorithm is an FS filter that uses Mutual Information (MI) to measure the relevance of a feature.

3.1.1 Mutual information

MI is a metric from Shannon's information theory that quantifies the amount of information that two random variables hold together. It can also be understood as the amount of information that is obtained from a random variable when the outcome of another independent variable is known. The MI between two independent features X and Y , denoted by $I(X; Y)$, is calculated as:

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log_2 \frac{p(x, y)}{p(x)p(y)} \quad (1)$$

where $\mathcal{X}$ and $\mathcal{Y}$ denote the alphabet of values that features X and Y can take, respectively; $p(x, y)$ denotes the joint probability of X taking the value x and Y taking the value y ; and $p(x)$ and $p(y)$ denote the marginal probabilities of X taking the value x and Y taking the value y , respectively.

To use this equation, we need to know the marginal and joint probabilities of the values that the variables can take. Since we will be analyzing discrete data, $p(x)$, which denotes the probability of an arbitrary variable X taking the value x , can be approximated as follows:

$$p(x) = p(X = x) \approx \frac{f_x}{N} \quad (2)$$

where f_x is the frequency of the event x (i.e., X takes the value x), and N is the total number of events.

The probabilities are represented by three data structures: two 1D arrays for the marginal probabilities of X and Y (their dimensions are $|\mathcal{X}|$ and $|\mathcal{Y}|$, respectively), and a 2D array for the joint probabilities (dimension $|\mathcal{X}| \times |\mathcal{Y}|$). Therefore, the spatial complexity of computing the MI is $O(|\mathcal{X}| \cdot |\mathcal{Y}|)$. Note that in scenarios with continuous data, both $|\mathcal{X}|$ and $|\mathcal{Y}|$ correspond to the number of bins used for discretizing the values.

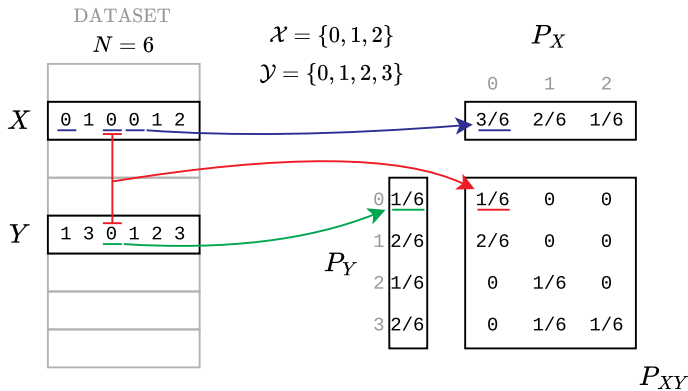
The calculation of the MI between any two variables X and Y can be seen as a two-step process:

1. Calculate the probabilities by computing the frequency (i.e., counting occurrences to construct a 2D histogram) of the values that X and Y can take, both individually (for the marginal probabilities) and jointly (for the joint probability). The complexity of this procedure is $O(N)$.
2. Calculate the MI by applying Eq. 1, which requires iterating over the three data structures. Its complexity is $O(|\mathcal{X}| \cdot |\mathcal{Y}|)$.

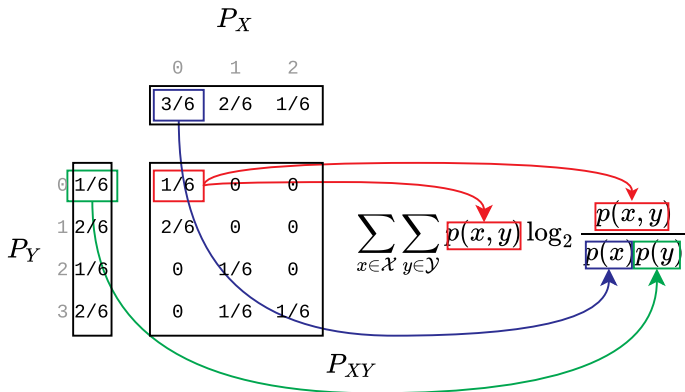
An example of this procedure can be seen in Fig. 1. First, Fig. 1a shows how the probabilities are calculated from the values of the features X and Y . The probabilities are then used to compute the MI, as shown in Fig. 1b

3.1.2 The mRMR algorithm

The mRMR algorithm [3] uses MI to approximate the relevance and redundancy of features in a dataset, providing a score for each feature. The mRMR score J of a



(a) Computation of probabilities for features X and Y with alphabets $\mathcal{X}$ and $\mathcal{Y}$, from a dataset with $N = 6$ samples



(b) Iteration over the probability data structures to compute the MI between X and Y

Fig. 1 Steps involved in the computation of the MI between two example variables, X and Y

feature X_i , given the dataset class Y and the subset of already selected features S , is calculated as:

$$J(X_i, Y, S) = I(X_i; Y) - \frac{1}{|S|} \sum_{X_s \in S} I(X_i; X_s). \quad (3)$$

Two terms can be identified in this equation. The first one, $I(X_i; Y)$, denotes the MI between a feature X_i and the class Y . This indicates relevance, which refers to the amount of information that the feature provides about the class. However, the mRMR score would be incomplete if only the relevance is considered. The second term, $\frac{1}{|S|} \sum_{X_s \in S} I(X_i; X_s)$, denotes the average amount of information that the X_i feature has in common with the already selected features (those in subset S). This gives a measure of redundancy when selecting X_i .

Algorithm 1: Basic structure of FS with mRMR

Input: dataset D , class Y , number of features to select k

Output: subset of selected features S

```

1:  $S \leftarrow \emptyset$ 
2: while  $|S| < k$  do
3:    $P_i \leftarrow \text{zeros}(|D|)$ 
4:   for all  $X_i \in D \setminus S$  do
5:      $P_i \leftarrow I(X_i; Y)$ 
6:     for all  $X_s \in S$  do
7:        $P_i \leftarrow P_i - I(X_i; X_s)/|S|$ 
8:     end for
9:   end for
10:   $l \leftarrow \text{argmax}(P)$  // index of the feature to be selected
11:   $S \leftarrow S \cup \{l\}$ 
12: end while

```

The mRMR algorithm uses a score to measure the goodness of each feature. Algorithm 1 shows a basic pseudocode of the method, highlighting how the output set of selected features is constructed iteratively. Two parts can be distinguished at each iteration of the main loop (line 2). First, the mRMR score P is calculated for each non-selected feature (lines 4–9). Then, the feature with the best score is found and added to the output subset (lines 10–11).

As seen in Sect. 3.1.1, the MI computation is not a lightweight operation. In Algorithm 1, the MI is recomputed several times (line 2) for pairs of variables that do not change between iterations of the loop. Specifically, we can observe that this redundancy occurs between the current variable X_i and the class Y (line 5), and also between the first $|S| - 1$ features added to the selected subset (line 7).

The algorithm can be reformulated more efficiently by computing the MI once for each pair of features (X_i, Y) and storing the results. This approach was already part of the sequential C-based implementation of mRMR available in the FEAST library,⁸ developed to support the research of [29]. The resulting pseudocode of this reformulation is shown in Algorithm 2. Two additional data structures are required: a 1D array to store the MI between each feature in the dataset and the class (N , in line 3), and a 2D array to store the MI between each feature in the dataset and the selected features (M , in line 7). The dimensions of these arrays are $|D|$ and $(k - 1) \times |D|$, respectively. This algorithm starts by computing the MI for all features with the class Y (lines 2–4) and takes as the first feature to select the one with the highest MI (lines 5–6). Then, the main loop (line 8) runs until all requested features are selected. The loop in line 11 starts by calculating the MI of all the features with the last one selected.

⁸ <https://github.com/Craigacp/FEAST>

Algorithm 2: FS with mRMR using MI caching**Input:** dataset D , class Y , number of features to select k **Output:** subset of selected features S

```

1:  $S \leftarrow \emptyset$ 
2: for all  $X_i \in D$  do
3:    $N_i \leftarrow I(X_i; Y)$ 
4: end for
5:  $l \leftarrow \text{argmax}(N)$ 
6:  $S \leftarrow S \cup \{l\}$ 
7:  $M_{ji} \leftarrow \text{zeros}(k \times |D|)$ 
8: while  $|S| < k$  do
9:    $P_i \leftarrow \text{zeros}(|D|)$ 
10:   $j \leftarrow |S|$ 
11:  for all  $X_i \in D \setminus S$  do
12:     $M_{ji} \leftarrow I(X_i; X_l)$ 
13:  end for
14:  for all  $X_i \in D \setminus S$  do
15:     $P_i \leftarrow N_i$ 
16:    for  $s \in [1, |S|]$  do
17:       $P_i \leftarrow P_i - M_{si}/|S|$ 
18:    end for
19:  end for
20:   $l \leftarrow \text{argmax}(P)$ 
21:   $S \leftarrow S \cup \{l\}$ 
22: end while

```

Then, the mRMR score (P) is calculated using the cached data (lines 14–19). Finally, the feature with the highest score is selected (lines 20–21).

Note that the computation of the MI between each feature and the class (line 5 in Algorithm 1) has been moved out of the main loop (lines 2–4 in Algorithm 2), being calculated only once per feature, before the main computation, and stored in an array (N). Thanks to this rearrangement of the code, the heaviest part of the algorithm computation can be isolated in the loops starting at lines 2 and 11. Furthermore, we can abstract a generic function that can be parallelized and used in both places: $I_{all}(D, X) \rightarrow N$. The function I_{all} takes a dataset D and an arbitrary feature X and returns a 1D array of dimension $|D|$ with the MI between each feature in D and X .

3.2 Cross-platform GPU compatibility and autotuning

GPU vendors such as NVIDIA and AMD provide low-level programming models like CUDA or AMD ROCm that deliver high performance by interacting closely with the hardware. However, this comes at a cost, as code written for one platform is not easily

portable to another. To address this, some higher-level programming models support cross-platform compatibility. For example, programs written in OpenCL⁹ can take advantage of parallelism in GPUs and even in multicore CPUs. Built on the principles of OpenCL, but with a more modern API, SYCL targets GPUs from different vendors, CPUs, and even FPGAs, using a single-source codebase. Another example is HIP, which can be compiled for both NVIDIA and AMD GPUs.

Therefore, to target multiple platforms, we can either rely on having multiple backends or use a higher-level programming model that supports a wide range of platforms. The balance between performance and portability, known as performance portability, ensures that applications can achieve high performance on multiple hardware platforms.

Performance is determined by a set of parameters known as performance parameters. By expressing the code in terms of these parameters and assigning different values to them, we can gain control over the performance of the application. The automatic process of finding optimal values for these parameters is known as autotuning, which allows applications to achieve near-optimal performance across a wide range of platforms without extensive manual effort, making it an important component of modern HPC.

However, identifying the optimal values for these parameters is challenging in the exascale era due to the increasing complexity and heterogeneity of systems. The large number of performance parameters (application-, framework-, and architecture-dependent) makes exhaustive search methods unfeasible, since they require evaluating every possible configuration. Instead, ML-based approaches, in particular BO [30], provide a better alternative to address this challenge. BO treats performance optimization as a black-box optimization problem, which enables the discovery of near-optimal parameter configurations while significantly reducing the search cost. BO solves the tuning process as an iterative optimization problem where the objective is to identify optimal parameters with minimal evaluations. It uses a surrogate model, often a Gaussian process, to approximate the application's performance (objective function) based on performance configurations (parameters). The surrogate model predicts both the mean and the uncertainty (variance) of performance for new configurations. This predictive approach allows BO to strategically explore the parameter space while balancing exploration and exploitation in a trade-off fashion:

- Exploration focuses on less known regions of the search space to gain new information.
- Exploitation focuses on refining high-performing regions already identified.

The acquisition function guides this trade-off by assigning a score to each configuration to prioritize either exploration or exploitation. As new data are introduced through evaluations, the surrogate model is constantly retrained and progressively improving its accuracy to guide the search. Unlike grid search or random search, which may require a larger number of evaluations to explore the large search space, BO builds an accurate surrogate model with fewer function evaluations [23, 24, 28].

While traditional BO-based autotuning operates as a black-box framework within a fixed platform, performance portability poses additional challenges when applications

⁹ <https://www.khronos.org/opencv/>

are executed across different architectures. Transfer learning (TL), an ML technique that reuses knowledge from one source task to improve performance in another, can help significantly in this context. By adding platform-specific information and leveraging previous searches, TL facilitates cross-platform tuning, which significantly reduces the cost of the tuning search when migrating applications across heterogeneous platforms [27, 31].

In our study, we use DFTuning [27], which builds upon the GPTune framework [28], to perform BO-based searches, and has been proved capable of finding near-optimal configurations for manageable search spaces that can be explored empirically. GPTune is an open-source autotuner that iteratively builds a Gaussian process surrogate model using multitask learning to integrate data from multiple sources. This multitask approach enables the use of TL. By transferring knowledge from previous optimization runs on other GPU architectures or datasets, DFTuning can be used to minimize redundant search exploration and thus promote performance portability, making it particularly well suited for tuning mRMR on different GPU architectures.

4 GPU implementation of the mRMR algorithm

The GPU implementation of mRMR developed in this work builds on the caching-based version presented in Algorithm 2 of Sect. 3.1.2, which introduces several key modifications to increase performance over any previous implementation [6]. It adopts a resident-dataset approach in which the dataset is transferred to GPU memory only once and remains on the device for the entire FS process, while the host orchestrates the computation by managing batching, memory management, and kernel launches. To ensure portability, our work implements the GPU-based mRMR algorithm across three different backends (CUDA, HIP, and SYCL) to target different hardware platforms. These backends expose a number of performance parameters that need to be tuned. To achieve performance portability, Sect. 6 explains how to use DFTuning to find the optimal values for these parameters for each backend, dataset, and hardware platform.

4.1 Modifications to the mRMR algorithm

Although the joint probability in Eq. 1, $p(x, y)$, depends on the features involved, the marginal probabilities, $p(x)$ and $p(y)$, remain constant for each feature. Therefore, they can be computed only once and cached for later use. Compared to previous implementations, we now cache both marginal probabilities and simplify their computation by keeping them resident in GPU global memory for different kernel executions. A high-level diagram of the modified mRMR is shown in Fig. 2. By precomputing all marginal probabilities and using global memory, we save GPU cycles and reduce reliance on shared memory, which enhances the occupancy of GPU kernels. Note that in our implementation, the marginal probabilities are computed on the GPU, even though the runtime of this step is almost negligible (and therefore it is not highlighted in Fig. 2).

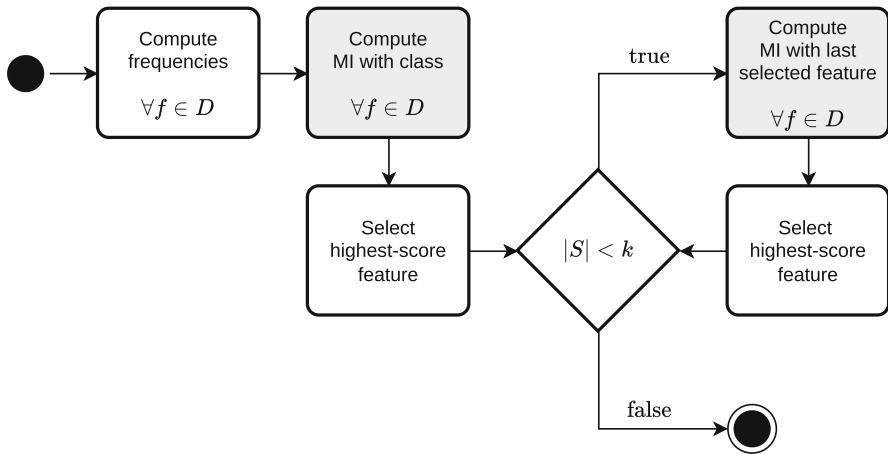


Fig. 2 High-level view of FS for a subset S with k features from a dataset D . Dark-highlighted steps indicate those involving MI calculation for all features, which can be parallelized on GPUs

The amount of memory needed to store these cache structures grows with the number of features (note that probabilities are stored as high-precision data with 64-bit floats). To reduce this memory footprint, our implementation precomputes and caches frequencies (stored as 32-bit unsigned integers) instead of probabilities. Later, it converts them to probabilities on demand. The conversion is done in two steps: first, the integer is converted to a 64-bit float, and then it is divided by the total number of samples. Thus, we can optimize Eq. 1 to work with frequencies, as shown in Eq. 4:

$$I(X; Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \frac{f_{xy}}{N} \log_2 \frac{f_{xy}/N}{f_x/N \cdot f_y/N} = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \frac{f_{xy}}{N} \log_2 \frac{f_{xy} \cdot N}{f_x \cdot f_y}. \quad (4)$$

With the modifications introduced so far, the computation of the MI consists of two steps: (1) calculate the joint frequencies; (2) calculate the MI using the marginal and joint frequencies and converting them to probabilities. These two steps are performed within a single GPU kernel, using one thread block per feature, i.e., given a fixed feature Y , every thread block is responsible for computing the MI between a feature in the dataset and Y . Threads within each block work together on different samples. The amount of memory required for the structure may differ between pairs of features, since their alphabets may be different. Following the approach of previous implementations, we developed a custom memory management system based on pools of GPU global memory managed by the CPU. In addition, we divided the workload into batches, with each kernel invocation computing the MI for a subset of features. This design is motivated primarily by memory constraints. Since the dataset and one joint-frequency structure per processed feature must reside in GPU memory, processing all features in a single kernel invocation may exceed the available memory for datasets with a large number of features. Batching allows us to bound memory usage while avoiding the high overhead of launching one kernel per feature.

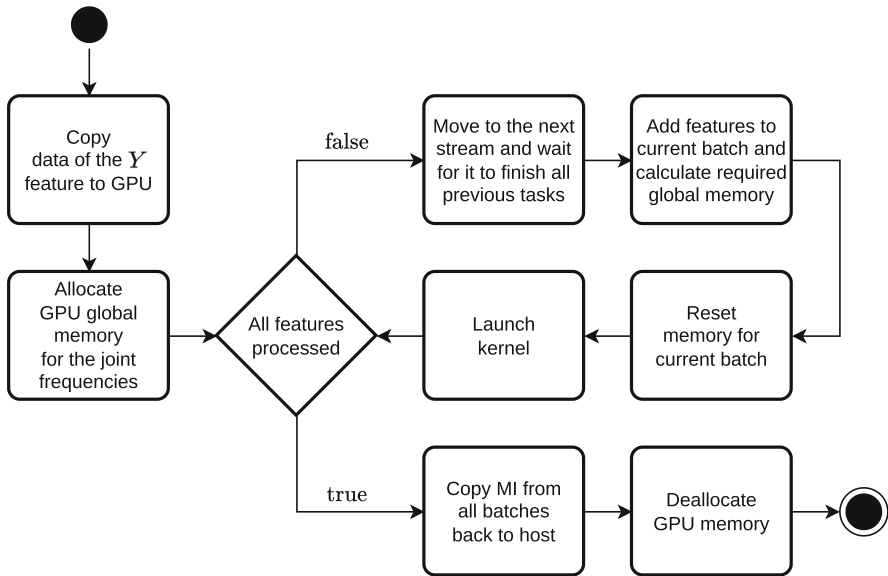


Fig. 3 Diagram of the batched and asynchronous GPU computation of MI

This design also enables the overlap of host-side preparation and device execution using asynchronous streams, as illustrated in Fig. 3. In particular, the host iterates over batches, performing memory allocation, data transfers, and initialization for each batch, while previously submitted batches are executed on the GPU. A similar strategy was used in the original CUDA implementation of mRMR [6] and is adapted here to the multi-backend setting.

Theoretically, it is possible to use an arbitrarily large number of streams in some devices, which would eliminate synchronizations by assigning a unique stream per batch. However, this approach is not feasible because a maximum number of streams must be set beforehand in order to perform memory allocations. Note that the allocated memory grows not only with the batch size but also with the maximum number of streams, which can significantly impact performance. Therefore, a bounded number of streams is used and reused in a cyclic manner across batches. Consequently, before submitting a new batch to a stream that has already been used, the host must synchronize that stream to ensure that all previously enqueued operations have completed. These synchronizations are thus required to guarantee correctness when reusing streams, rather than due to data dependencies between batches. The optimal number of streams used may vary from dataset to dataset. Therefore, it has been exposed as a performance parameter to be tuned in our implementation.

4.1.1 Shared memory optimization

While the marginal frequencies can be precomputed, the joint frequencies must be calculated within the GPU kernel. In particular, the creation of the joint frequencies requires N writes and the MI calculation requires $|\mathcal{X}| \cdot |\mathcal{Y}|$ reads, using shared memory

for this. However, the capacity of shared memory is limited (tens of kiB per block), and in most cases it is not sufficient to store the joint frequencies. In addition, the amount of shared memory used per thread block can be a limiting factor in GPU occupancy.

Although previous implementations used 16-bit precision for both marginal and joint frequencies, this work employs 16-bit precision only for joint frequencies and 32-bit for marginal frequencies. By moving marginal frequencies out of shared memory, we increase the capacity to store joint frequencies in shared memory. However, from a performance perspective, there is an inflection point where the reduced occupancy caused by using shared memory does not offset the increased latency of accessing global memory. To address this, this work also considers the maximum shared memory usage per thread block as a performance parameter. If the data structure does not fit in the specified amount of shared memory, then global memory is selected specifically for that feature. This performance parameter allows to establish a balance between: (1) maximizing the likelihood that shared memory is used for the joint frequencies, and (2) maximizing the number of active thread blocks on the GPU (related to occupancy). Section 4.2 explains some characteristics that each platform and programming model natively exposes, which can improve GPU performance and were considered as performance parameters in our work.

4.2 Backends for performance portability

Our GPU strategy, described in Sect. 4.1, can be adapted to different programming models. Adaptation to each model requires adhering to implementation-specific details, including variations in naming conventions (e.g., the concept of “streams” in CUDA and HIP is equivalent to “queues” in SYCL). This work adopts the CUDA terminology, as it is the most widely recognized standard in the HPC community.

Although CUDA, HIP, and SYCL provide different APIs for GPU control, they share several core functionalities, such as kernel execution, memory management, and asynchronous task handling. CUDA and HIP provide similar mechanisms for these functionalities, while SYCL often takes a different approach. The following examples from our mRMR implementation illustrate key differences and necessary adjustments in the codebase:

- *Writing and compiling kernels* While CUDA and HIP define a kernel as a C-style function, SYCL provides a higher-level approach that allows kernels to be defined as named function objects or lambda functions.
- *Starting kernel executions* CUDA and HIP support a common syntax `kernel<< gridDim, blkDim, shMem, stream>>(args...)` to configure the grid dimension, block dimension, shared memory per block, and id of the stream for which the kernel is launched. This call acts as a wrapper to the `cudaLaunchKernel` and `hipLaunchKernelGGL` functions, respectively. SYCL follows an object-oriented syntax. Instead of having grid and block dimensions, SYCL specifies the partitioning of the workload by a global range (total number of threads) and a local range (number of threads per block). In all three frameworks, kernel launches are asynchronous to the host, meaning that kernels are scheduled for execution on the GPU and control is returned to the CPU.

- *Managing memory* CUDA and HIP provide functions for allocating and deallocating buffers on global memory with different characteristics (e.g., non-pageable memory), specific memory layouts (e.g., padded), and support for asynchronous operations. Shared memory can be allocated statically, if the size is known at compile time, by specifying it in the kernel. If the size is known at runtime, it must be specified when launching the kernel. Instead, SYCL exposes two mechanisms for dealing with memory management:
 - The Buffer-Accessor (BA) model provides a high-level abstraction of the memory. Buffers are an abstraction for a view of the memory, accessible by the host or the device, and accessors are the abstractions used to read or write memory from a buffer.
 - The Unified Shared Memory (USM) is the lower-level option because it relies on pointers and various functions to allocate different types of memory.

The USM approach was used in this work to keep more control over global-memory operations and because the resulting code structure is similar to that of CUDA and HIP. However, the shared memory must be handled in a BA fashion. To set memory, the `{cuda, hip}Memset` functions are used in CUDA and HIP, while the `fill` method of the `Queue` class is used in SYCL.

- *Managing asynchronous streams* The main difference in handling streams is that CUDA and HIP have an implicit stream where tasks are enqueued by default. If more streams are needed, they must be created and destroyed using the appropriate functions. Instead, there are no default streams in SYCL, and at least one queue must be explicitly created to submit any work to the GPU. In addition, more queues can be created to take advantage of inter-stream asynchrony.
- *Identifying the active thread and block* In our approach, mRMR uses the thread block index to identify the feature being processed, while the thread index is used to calculate the memory addresses accessed by each thread. In CUDA and HIP, these indices (as well as the dimensions of the grid and the thread blocks) are obtained within the kernel from specific built-in variables. This is different in SYCL, where an object is mapped to each unit of parallel execution, such as a work-item, and indices and dimensions are obtained using `get_*` methods.
- *Addressing atomic operations* Atomic operations are executed in the MI computation to accumulate the data frequencies as well as the partial MI of each thread. In CUDA and HIP, there are specific functions used for this purpose. In SYCL, atomics require some more work to set them up. First, an atomic reference must be constructed for a given memory address, and then a fetch method is used to increment the value.

In addition to the common functionalities shared by the three frameworks, CUDA and HIP provide one aimed at improving performance: occupancy tuning. In both CUDA and HIP, kernel resource utilization can be optimized if a maximum number of threads per block is set at compile time. This is achieved by adding the `__launch_bounds__(maxThreadsBlk)` qualifier to the kernel, where `maxThreadsBlk` specifies the maximum number of threads per block. This provides the compiler with hints about resource usage, allowing it to apply informed

Table 1 Datasets used in the experiments

Dataset	Features	Samples	Classes
MNIST	780	60,000	10
Epsilon	2000	400,000	2
RCV1 train	47,236	20,242	2
SVHN	3072	531,131	10
News20 TFIDF	130,107	11,314	20

optimizations rather than generating a more generic binary that works across different block sizes. The `__launch_bounds__` qualifier can take a second parameter that is interpreted differently by CUDA (minimum number of blocks per multiprocessor) and HIP (minimum number of warps per execution unit). Both refer to the same concept: a request for a minimum number of resident threads (measured in blocks or warps) per hardware component (multiprocessor or execution unit). With the usage of `__launch_bounds__`, hardware resources are redistributed so that there can be more resident warps at the same time, which is beneficial for workloads where threads spend a significant amount of time stalled. Since choosing the best value for the maximum number of threads is a complex task, our work includes this performance parameter in the set of configuration parameters explored by DFTuning.

5 Computational setup for experiments

The experimental evaluation presented in the next sections focuses on assessing the performance differences between programming frameworks (CUDA, HIP, and SYCL) running on different GPUs for datasets with distinct properties. All datasets were obtained from the LIBSVM [32] website,¹⁰ and their properties are listed in Table 1. These datasets contain continuous data, which is not valid for mRMR. Note that these data were discretized using 128 bins, i.e., the values of each feature were mapped into 128 intervals to obtain a discrete representation suitable for the method.

The characteristics of the GPUs are detailed in Table 2: an NVIDIA RTX4090 installed on a workstation (“Lovlace” from now on), an NVIDIA A100 GPU from a node of the Perlmutter supercomputer, and an AMD MI250X from the Frontier supercomputer. Note that the table shows the hardware limits of some factors that influence GPU occupancy, but the actual occupancy depends on the final characteristics of each executed kernel (e.g., register pressure). Also note that the AMD MI250X GPU consists of two dies and behaves as two separate GPUs from a programming perspective. Our implementation targets single-GPU execution and therefore uses only one of these dies. Consequently, the available resources on this platform are effectively reduced (resources marked with an asterisk in the table are limited to half). In addition, for the MI250X, the register file size and the maximum 32-bit registers per SM are shown as a sum (scalar + vector) because the CDNA2 microarchitecture has separate registers for scalar and vector data.

¹⁰ <https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/>

Table 2 Characteristics of the GPUs used in the experiments

System	Lovelace	Perlmutter	Frontier
GPU	NVIDIA RTX4090	NVIDIA A100	AMD MI250X
Microarchitecture	Ada Lovelace	Ampere	CDNA2
Frequency/Boost (GHz)	2.24/2.52	0.76/1.41	1.00/1.70
RAM (GiB)	24	40	128*
Bandwidth (TiB/s)	0.98	1.52	3.20*
Streaming Multiprocessors (SMs)	128	108	220*
Cores per SM	128	96	64
Total cores	16,384	10,368	14,080*
L1 cache per SM (kiB)	128	192	16
L2 cache (MiB)	72	40	16*
Threads per warp	32	32	64
Max warps per SM	48	64	32
Max threads per SM	1,536	2,048	2,048
Max threads per block	1,024	1,024	1,024
Shared memory per SM (kiB)	100	164	64
Register file size per SM (kiB)	256	256	12.5+512
Max 32-bit registers per SM	65,536	65,536	3,200+131,072
CPU	Intel i9-14900KF	AMD EPYC 7763	AMD EPYC 7713
RAM	64 GiB	256 GiB	512 GiB

For comparison across different backends, Table 3 summarizes the software environment used to run the experiments on the Lovelace and Perlmutter systems. The main host compiler was GCC in both cases: v.13.2.0 on Lovelace and v.11.2.0 on Perlmutter. Lovelace ran Ubuntu 24.04 LTS, while Perlmutter used the NERSC-managed HPE Cray software environment. All backends were compiled with `-O3`. Host compilation used OpenMP via `-fopenmp`, and CUDA/HIP builds passed `-march=native` to the host compiler.

6 Implementing performance portability as a search problem

Our GPU backends for mRMR expose four tunable performance parameters: threads per block, features per batch, shared memory per block, and number of streams. In addition, the minimum number of blocks per SM can be tuned in CUDA and HIP with the `__launch_bounds__` qualifier. Table 4 lists the performance parameters defined for the DFTuning search and their ranges. As shown in the table, each of these parameters can take many possible values, resulting in a total search space of 106,229,760 combinations. To find the best configuration exhaustively would require a huge number of experiments. If all combinations were valid, and considering an

Table 3 Backend-specific software environment used on the Lovelace and Perlmutter platforms

System	Backend	Compiler / Toolchain
Lovelace	CUDA	system nvcc; linked with libcudart.so.12
	HIP	system hipcc; linked with libcudart.so.12
	SYCL	Intel LLVM/SYCL clang 19.0.0; linked with libsycl.so.8
Perlmutter	CUDA	NVIDIA HPC SDK 22.7 nvcc
	HIP	ROCm 5.5.1 hipcc
	SYCL	NERSC Intel LLVM 2023-WW13

Table 4 Definition of the GPU-mRMR search parameters for DFTuning

Parameter	Formulated as	Range for n
Warps per block	n	[1, 32]
Features per batch	2^n	[6, 12]
Shared memory per block (B)	$1,024 \times n$	[0, 227]
Streams	n	[0, 64]
Minimum blocks per SM	n	[1, 32]

average execution time of 0.5 s per evaluation, the total evaluation time for an exhaustive search would be 3074 days per backend and platform to complete five datasets. To perform this search in a feasible time, we used DFTuning to find the (near-) optimal values of these parameters for each platform and backend.

As explained in Sect. 3.2, DFTuning uses a BO model to find the configuration (performance parameters) that achieves this near-optimal performance. DFTuning categorizes parameters into two types: input and performance parameters. Input parameters define the specific problem instance to be tuned (e.g., problem size, selected dataset) and are used uniquely to describe the problem. Performance parameters, on the other hand, directly affect the execution performance and are explored.

In our mRMR case, the input parameters define the dataset used (*MNIST*, *Epsilon*, *RCV1 train*, *SVHN*, or *News20 TFIDF*), the platform (Lovelace, Perlmutter, or Frontier), and the backend (CUDA, HIP, or SYCL). Adding the platform and backend as input parameters allows us to characterize the search for each portability scenario. Note that some combinations are incompatible, e.g., CUDA cannot be used with the AMD GPU on the Frontier system.

Table 4 includes all possible combinations, but not all of them are necessarily valid. For example, there are GPU architectures with up to 227 kiB of shared memory per block (e.g., NVIDIA H100¹¹), but we did not use these architectures in our work. While theoretical bounds describe the problem, each portability search must be constrained

¹¹ <https://www.nvidia.com/en-us/data-center/h100>

by its specific characteristics. To do this, we defined the following constraints in DFTuning:

- Number of threads per block (GPU warp size $\times$ warps per block) is less than or equal to the hardware maximum (1,024 for all GPUs tested, but may vary by architecture). Note that the block size is defined in terms of warps per block instead of threads per block. This allows DFTuning to abstract the search process from the GPU warp size and then apply TL between platforms where it differs.
- Number of threads per SM (threads per block $\times$ blocks per SM) is less than or equal to the hardware maximum (varies by GPU).
- Shared memory per block is less than or equal to the hardware maximum (varies by GPU).
- Shared memory per block is less than or equal to the maximum needed to compute the MI. Since discretization was fixed to 128 bins, we can only calculate a maximum bound of $128^2 \times 2 \text{ B} = 32 \text{ kiB}$. This constraint avoids limiting the occupancy by allocating more shared memory than needed.
- Number of features per batch is less than or equal to the number of features in the dataset (varies by dataset, see Table 1).
- Number of streams is less than or equal to the number of batches (features in the dataset divided by features per batch).

Some constraints are enforced directly in the mRMR code to prevent performance losses due to inefficient configurations. However, they must also be defined in DFTuning to avoid wasting resources evaluating ineffective regions of the search space.

In addition, we can leverage the knowledge from previous searches to guide new searches (i.e., for different input parameters) by focusing only on already discovered high-performance regions of a similar search space. This is achievable by TL, which effectively reduces the number of executions BO needs to find near-optimal configurations. For example, in a new tuning search where the only difference is the selected dataset, some good values of the performance parameters are likely to remain similar across datasets. BO can take advantage of this prior information to reduce the number of evaluations needed. A key innovation introduced in [27] is the use of TL to incorporate search knowledge from previous platforms into new ones, thereby improving portability optimization.

DFTuning builds the initial surrogate model by evaluating some random configurations. Then, new configurations are interactively evaluated following the exploration/exploitation paradigm until the stopping criterion is met. Specifically, DFTuning was configured with a stopping criterion based on a window size of the last $W = 20$ executions and a threshold of $\tau = 1.5\%$. If no configuration improves the previous runtime by at least τ within W executions, the search is terminated. To reduce noise in the measurements, each configuration was run five times and the average was calculated.

The tuning experiments were conducted with $k = 20$ selected features to keep the search cost manageable. In our GPU implementation of mRMR, the configurable performance parameters affect the computation of feature scores, which is the device-side step repeated at each selection iteration. These parameters do not alter the algorithmic structure for different values of k ; therefore, increasing k primarily results in addi-

tional repetitions of the same score-computation procedure. Consequently, runtime is expected to scale linearly with k for both optimized and unoptimized configurations, and the tuning conclusions should remain valid for larger values of k .

Note that the autotuning process is treated as an offline configuration step rather than as part of the execution of mRMR itself. In practice, autotuning would be performed once for a given hardware platform and a representative set of workload characteristics (e.g., number of samples and feature cardinality). The resulting configuration can then be reused across multiple runs and datasets with similar characteristics. In this study, we perform tuning independently for each dataset to provide a controlled comparison across backends and platforms. This design choice allows us to isolate backend behavior without confounding effects from cross-dataset generalization. However, it represents a worst-case scenario in terms of autotuning cost, as practical deployments would amortize this cost across multiple executions.

7 Experimental results

The experimental evaluation carried out in this work, based on the data collected from the DFTuning execution, was twofold. On the one hand, a performance comparison of the three optimized backends was conducted on NVIDIA architectures. On the other hand, we include an exploratory evaluation of the HIP implementation across platforms. This analysis is intended to illustrate how the same implementation behaves under different execution environments.

The performance results reported in this section consider only the execution time of the mRMR algorithm using the configurations obtained through DFTuning. The cost of the autotuning process is not included, as it is treated as an offline configuration step performed once per platform and amortized across multiple executions.

7.1 DFTuning execution

Table 5 shows the DFTuning evaluation setup for each combination of input parameters. It includes the DFTuning configuration for evaluations (number of initial random evaluations in the BO search plus maximum number of BO-guided evaluations allowed per search), and the evaluation at which the BO search stopped, based on the previously defined stopping criterion that ends the search after a window of no progress. In addition, for searches that used TL, the platform and backend employed as the source of prior knowledge are also displayed. The same setup (random evaluations and a maximum number of BO-guided evaluations) was applied to the five test datasets for each platform and backend search. Consequently, each row shows the average number of actual evaluations performed before reaching the stopping criterion, along with the average runtime speedup achieved by the best BO-found configuration compared to the average runtime achieved by the initial evaluations across the five datasets. Also, when TL was used for a particular combination of platform, backend, and dataset, the source task was a previous search for the same dataset but on the corresponding platform and backend shown in the “TL source” column of Table 5. Finally, for searches

Table 5 Setup for the DFTuning searches and performance of the best configurations found

Platform	Backend	BO evaluations Random+Max.	Executed	TL source	Runtime speedup
Lovelace	CUDA	30+120	50	No TL	1.20x
	HIP	20+100	53	Lovelace/CUDA	1.30x
	SYCL	20+100	48	Lovelace/CUDA	1.30x
Perlmutter	CUDA	30+120	54	No TL	1.18x
	HIP	20+100	49	Perlmutter/CUDA	1.23x
	SYCL	20+100	43	Perlmutter/CUDA	1.27x
Frontier	HIP	10+80	48	Perlmutter/CUDA	1.84x

that did not use TL to accelerate the process, we allowed a higher maximum number of BO-guided evaluations and increased the number of initial random evaluations. This compensates for the lack of prior knowledge, allowing the surrogate model to be refined with more data.

The use of DFTuning significantly aided the search for near-optimal configurations of the mRMR computation across different backends and platforms. On average, the search required 49 attempts per portability scenario, a small fraction of the executions (and total runtime) that an exhaustive search would have required. In addition, when analyzing the total number of evaluations executed (which includes both random and BO-guided evaluations), we observe that the number of evaluations performed by the random phase is similar to those conducted during the guided phase of DFTuning. It is noteworthy that the configurations identified by the BO-guided phase, which use information from the previous random evaluations, achieved runtimes that were on average 22.53% faster than the average runtimes of the initial random configurations.

Finally, note that the goal of TL is not solely to reduce the total number of evaluations, but to improve the quality of the search trajectory and the robustness of the final configuration. Without TL, the search on a new platform requires more extensive exploration to rediscover high-performing regions. This can result in slower convergence and, in some cases, convergence to a different near-optimal configuration that is farther from the best-performing region identified with TL.

7.2 Performance comparison between backends

The mRMR versions using all three backends were tested on both the Lovelace and Perlmutter systems to analyze the performance differences. SYCL results are not reported for Frontier because we could not obtain a working SYCL configuration for the AMD GPU environment used in the experiments. Therefore, the direct CUDA/HIP/SYCL comparison is limited to NVIDIA GPUs, while the Frontier evaluation is restricted to the HIP backend. The average runtimes of the initial random evaluations, used as the baseline, and the best runtimes achieved by DFTuning for each combination of input parameters (dataset, backend, and platform) are shown in Tables 6 and 7, respectively.

Table 6 Average runtimes and standard deviations (in seconds) for the initial random parameter configurations for each combination of dataset, platform, and backend

Platform	Dataset	CUDA	HIP	SYCL
Lovelace	MNIST	0.08 ± 0.01	0.08 ± 0.01	0.08 ± 0.01
	Epsilon	0.68 ± 0.08	0.73 ± 0.20	0.74 ± 0.21
	RCV1 train	0.93 ± 0.07	0.92 ± 0.12	1.06 ± 0.17
	SVHN	1.57 ± 0.50	1.78 ± 0.76	1.77 ± 0.66
	News20 TFIDF	1.69 ± 0.19	1.67 ± 0.24	2.01 ± 0.32
Perlmutter	MNIST	0.16 ± 0.01	0.16 ± 0.01	0.16 ± 0.01
	Epsilon	0.83 ± 0.15	0.89 ± 0.41	0.90 ± 0.42
	RCV1 train	1.24 ± 0.08	1.27 ± 0.19	1.50 ± 0.27
	SVHN	1.71 ± 0.50	1.86 ± 0.55	1.90 ± 0.58
	News20 TFIDF	2.03 ± 0.19	2.10 ± 0.35	2.62 ± 0.57

Table 7 Runtimes (in seconds) and relative runtime reduction over average random configurations for the best parameter configuration found for each combination of dataset, platform, and backend. Note that TL was used for HIP and SYCL, with CUDA as the source task under the same combination of platform and dataset

Platform	Dataset	CUDA	HIP	SYCL
Lovelace	MNIST	0.06 (−19%)	0.06 (−23%)	0.06 (−22%)
	Epsilon	0.61 (−11%)	0.59 (−20%)	0.60 (−19%)
	RCV1 train	0.83 (−11%)	0.76 (−17%)	0.85 (−20%)
	SVHN	1.18 (−25%)	1.15 (−36%)	1.17 (−34%)
	News20 TFIDF	1.43 (−15%)	1.42 (−15%)	1.63 (−19%)
Perlmutter	MNIST	0.13 (−15%)	0.14 (−14%)	0.14 (−10%)
	Epsilon	0.74 (−10%)	0.73 (−18%)	0.74 (−18%)
	RCV1 train	1.09 (−12%)	1.08 (−15%)	1.14 (−24%)
	SVHN	1.31 (−23%)	1.33 (−29%)	1.36 (−28%)
	News20 TFIDF	1.72 (−15%)	1.74 (−17%)	1.97 (−25%)

The performance of the best configuration identified for HIP and CUDA is very similar. Considering all datasets, HIP's best configurations outperform CUDA's by an average of 4.24% on Lovelace, while this difference is reversed on Perlmutter (0.78% slower on average). Despite potential fluctuations, the HIP search seems to benefit from TL (using CUDA as the source task), effectively biasing the search toward high-performance regions and likely identifying optimal configurations, slightly better than the CUDA ones in many cases. In contrast, SYCL shows slightly lower performance in all cases. On Lovelace, SYCL is on average 2.48% slower than CUDA and 6.99% slower than HIP. On Perlmutter, SYCL is 6.14% and 5.32% slower than CUDA and HIP, respectively. Our multi-backend approach ensures that each platform benefits from an optimized implementation tailored to its specific architecture, rather than

relying on a single portable implementation that may not fully exploit hardware-specific optimizations.

For the sake of fairness, the performance differences across the three backends are also analyzed for the same configurations (i.e., the parameters of the n -th configuration are the same for CUDA, HIP, and SYCL). This avoids noise from the DFTuning search, as there is no guarantee that the best configuration found in one backend is also explored in the search of the other backends. It also allows for a direct analysis of how performance varies between CUDA, HIP, and SYCL under identical execution settings. To do this, we evaluated the same 30 randomly sampled configurations across the three backends. Although the configurations were selected randomly, they are identical across CUDA, HIP, and SYCL, ensuring a controlled and fair comparison independent of the DFTuning search. The achieved runtimes are shown in Fig. 4, sorted by the CUDA runtime for better visualization. We can see that HIP behaves in a very similar way to CUDA. The most notable difference is for the *MNIST* dataset. However, since this is the smallest dataset, the runtimes are short and these variations could be explained by execution noise. In the case of SYCL, there are two interesting scenarios:

- The SYCL implementation is noticeably slower for the *RCV1 train* and *News20 TFIDF* datasets, which are the ones with the largest number of features relative to the number of samples. For our mRMR implementations, this results in executions with a high number of batches and thus a high number of kernel launches and synchronizations. Empirical results suggest that the overhead for these operations is higher in SYCL than in CUDA and HIP.
- SYCL runtimes do not mimic those of CUDA as smoothly as HIP. This results in noticeable “spikes” in the plots, with SYCL occasionally outperforming both CUDA and HIP in certain cases. This effect may be explained, among other factors, by the `__launch_bounds__` qualifier introduced at the end of Sect. 4.2. Although this qualifier can help improve performance by tuning the register pressure (thus improving GPU occupancy), an ineffective configuration can result in runtimes worse than if it is not used. As this qualifier is unique for CUDA and HIP, the SYCL implementation ignores it during the tuning search. As a result, CUDA and HIP take advantage of good configurations, while SYCL does not. However, if the tuning search does not explore the correct regions for this qualifier, CUDA and HIP executions are susceptible to performance degradation due to suboptimal configurations, while SYCL execution is not.

On Lovelace, HIP and SYCL runtimes were on average 0.48% and 8.86% slower, respectively, than CUDA for the same configuration. On Perlmutter, HIP runtimes were 0.06% faster than CUDA, while SYCL runtimes were 9.80% slower. While we can conclude that HIP effectively achieved the same performance as CUDA in our analysis, the case of SYCL shows the need to carry out a tuning search for each backend rather than using the same performance parameters across backends.

Note that the runtimes shown in this section are relatively short because the experiments were conducted with a small number of features. This was necessary to ensure that the BO search remained feasible, as it requires running the FS repeatedly along the features. However, in real-world scenarios, selecting more features increases the

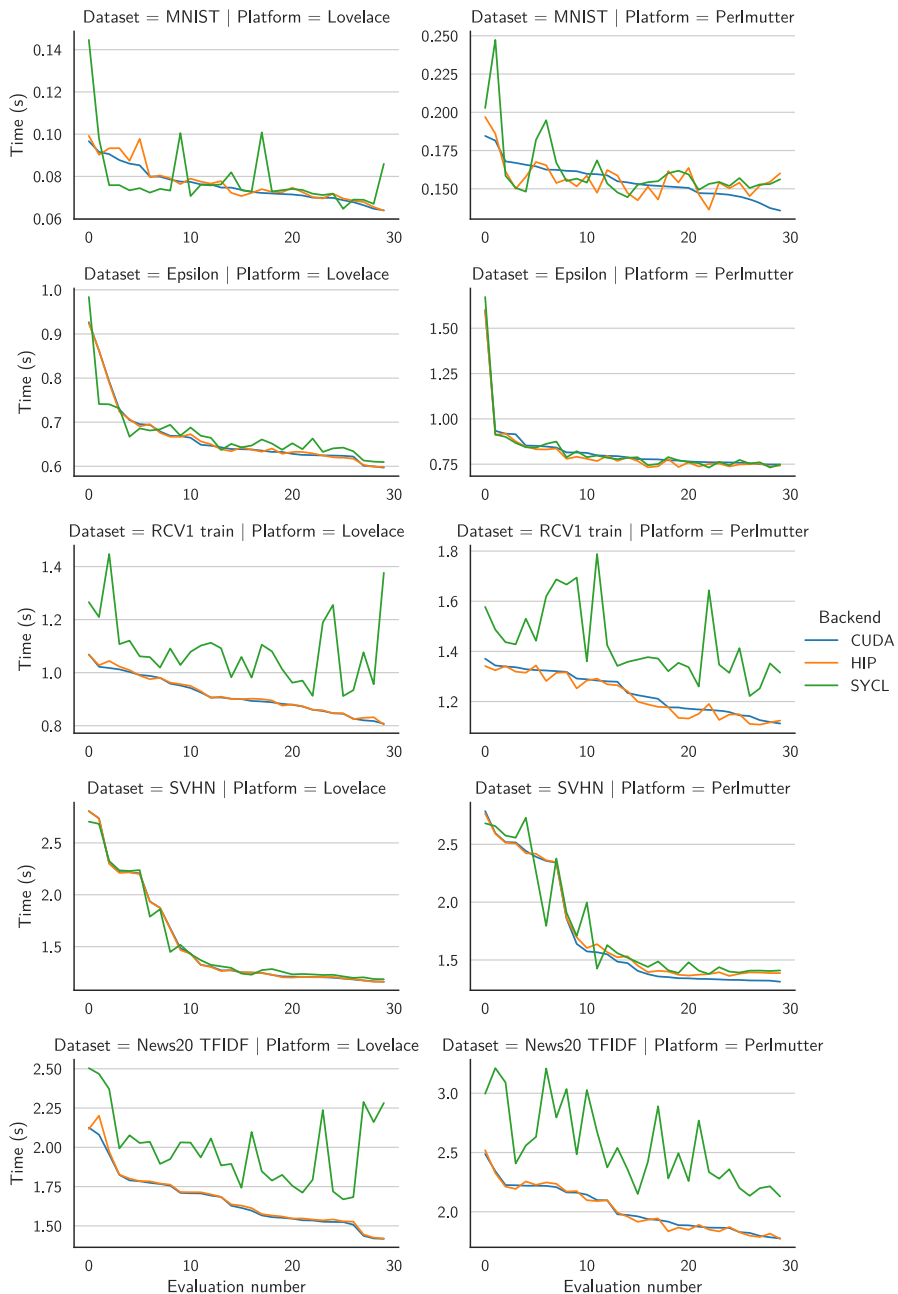


Fig. 4 Runtimes (in seconds) of the same 30 random configurations for CUDA, HIP, and SYCL, sorted by CUDA runtimes. CUDA and HIP exhibit similar trends, while SYCL shows more variation and does not always follow the same pattern. Each subplot uses an independent Y-axis range that does not start at zero to improve the readability of the differences between backends

Table 8 Best runtimes for HIP on each platform along with the corresponding speedup relative to Frontier

Dataset	Lovelace Time (s)	Spd. Frontier	Perlmutter Time (s)	Spd. Frontier	Frontier Time (s)
MNIST	0.06	3.91x	0.14	1.68x	0.23
Epsilon	0.59	1.72x	0.73	1.39x	1.01
RCV1 train	0.76	1.94x	1.08	1.38x	1.48
SVHN	1.15	1.68x	1.33	1.45x	1.93
News20 TFIDF	1.42	1.86x	1.74	1.51x	2.63

overall runtime. As a result, performance differences accumulate, amplifying their impact and making this performance tuning even more valuable.

7.3 Cross-platform behavior of HIP

Table 8 shows the runtimes achieved by the best configuration found for each dataset on each platform using the HIP implementation. The table also includes the speedup of each platform relative to Frontier, which is the slowest platform in our results. As noted above, Frontier did not have full SYCL support at the time of this work.

These results provide an exploratory view of how the same HIP implementation behaves across different execution environments. We emphasize that these platforms differ substantially in architecture, memory hierarchy, clock frequency, and software stack, and that the MI250X GPU is evaluated using a single die. Therefore, the results presented in this section should not be interpreted as a controlled comparison across GPU vendors, but rather as observations specific to the considered implementations and experimental setup.

Under these conditions, we observe that the Lovelace system achieves an average speedup of 1.47x and 2.22x over Perlmutter and Frontier, respectively, while Perlmutter achieves an average speedup of 1.48x over Frontier. For *MNIST*, the smallest dataset, the differences are more pronounced. This can be attributed to fixed overheads (such as data transfers, CPU-side FS, and kernel launch costs) having a relatively larger impact when the computational workload is small.

The observed performance differences across systems are influenced by multiple interacting factors related to both hardware characteristics and software execution. In particular, the mRMR workload relies heavily on atomic operations with irregular memory access patterns during frequency computation, followed by a more structured but computationally intensive MI calculation. The first step has been empirically identified as the dominant bottleneck due to its reliance on global-memory atomics and limited locality, while the second step benefits from more regular access patterns. Several architectural factors may contribute to the observed behavior, including differences in atomic operation latency, cache hierarchy (especially L2 cache capacity and behavior), register usage and its impact on occupancy, and operating frequency. For example, global-memory atomics are coordinated across thread blocks and may rely on different memory subsystems depending on the architecture. Larger L2 caches

can help mitigate contention and reduce effective latency, while higher clock frequencies may improve responsiveness to memory stalls. In our experiments, the best configurations identified by the autotuner tend to favor the use of global memory with atomics over shared-memory-based approaches for frequency computation.

However, we did not perform detailed hardware-level profiling to isolate and quantify the contribution of each of these factors. Therefore, the above considerations should be interpreted as potential contributing factors consistent with the observed behavior, rather than definitive explanations of the performance differences.

Overall, these results highlight that cross-platform performance differences are highly dependent on the interaction between workload characteristics, implementation choices, backend behavior, and system configuration. Consequently, they should be interpreted within the scope of the specific experimental setup used in this study, rather than as general conclusions about GPU architectures.

7.4 Discussion

Regarding performance, the experimental results from the previous subsection do not show a significant difference between the runtimes of the HIP and CUDA implementations. Although there is a high variance depending on the characteristics of the analyzed dataset, the SYCL implementation shows an average performance loss of about 6% (see Sect. 7.2). This validates our decision to implement a multi-backend approach rather than adopting SYCL (the most portable of the three programming models as of today). The observed SYCL behavior is in line with the conclusions of previous works [13, 18] already cited in Sect. 2. In our case, the performance drop may be related to overheads in certain operations (such as kernel invocation and data transfer to the GPU), compiler/runtime behavior, and the lack of an equivalent to the `__launch_bounds__` qualifier at the time of this research.

These results should be interpreted in the context of the software environments reported in Table 3. Since this work compares backend implementations compiled with different toolchains and runtimes, the observed differences reflect the combined effect of the programming model, compiler/runtime maturity, and system configuration. In particular, the small differences observed between CUDA and HIP on NVIDIA GPUs should be interpreted as comparable performance rather than as a meaningful advantage of either backend.

Previous studies have demonstrated the effectiveness of BO-based searches compared to random searches [23, 24, 28], as well as search acceleration when using TL from previous searches in similar architectures [27]. In our results, the BO-guided step that follows the random evaluation achieved 22% faster configurations. For example, compared to the HIP implementation, SYCL shows an average performance drop of 9.33% when running random configurations, while this drop is reduced to about 6.16% when using the guided search. The search space for each backend, platform, and dataset consists of more than 100 million combinations in total, while the BO-based search finishes after an average of 49 evaluations. While we recognize that these BO searches may not have explored the optimal configuration during these evaluations, as

practitioners we are satisfied with a near-optimal solution that reduces the time spent navigating such a large search space.

Furthermore, as shown in Sect. 7.3, performance differences are observed across the evaluated systems when using the same HIP implementation. Under the experimental conditions considered in this work, the AMD MI250X exhibits lower performance than the NVIDIA A100. These results depend on multiple factors, including the characteristics of the mRMR workload (which relies heavily on atomic operations), the implementation strategy, the HIP backend, the compiler and runtime environment, and the use of a single die of the MI250X. Several factors may contribute to this behavior. In particular, since the implementation is latency-bound due to the use of atomics, it is relevant to consider how atomic operations are handled on different platforms. While we use INT32 atomics to compute frequencies, some architectures may require specific compiler options (e.g., `-munsafe-fp-atomics`) to fully exploit hardware capabilities, and not using them may result in missed optimization opportunities^{12,13}.

While tools exist to quantify portability effort [33], such an analysis is beyond the scope of this work. However, our experience as developers provides valuable insights. Translating CUDA code to HIP was straightforward, given the similar C-based syntax and programming structure. Perhaps the most significant difference is that some recent CUDA features may not yet be available or mature in HIP. Porting the implementation to SYCL required more substantial restructuring. The underlying GPU execution hierarchy is conceptually similar, but SYCL exposes it through different abstractions. These differences made the SYCL implementation less direct and introduced a steeper learning curve, while also providing flexibility to target both GPUs and multicore CPUs.

Finally, we note that this study focuses on the mRMR algorithm, which is representative of MI-based FS methods. Its implementation is dominated by histogram construction, atomic operations, and irregular memory accesses, which are also present in other MI- and Conditional MI (CMI)-based methods. While different algorithms may combine these operations in different ways, potentially affecting portability and tuning behavior, we expect that several of the observed trends extend to this broader class of methods. Therefore, the conclusions of this work should be interpreted within the scope of MI-based FS. In this context, and considering our mRMR implementation and the tested platforms, our observations suggest the following guidelines:

- For porting an existing CUDA mRMR codebase to AMD GPUs, HIP is a natural choice, as it requires limited restructuring and enables execution on the tested AMD platform.
- For this workload on NVIDIA GPUs, CUDA and HIP achieve comparable optimized performance, making HIP a reasonable option when a CUDA-like codebase targeting both NVIDIA and AMD GPUs is desired.
- SYCL requires more substantial restructuring and is slower in our NVIDIA experiments, but remains attractive when broader source-level portability is the main objective.

¹² <https://github.com/intel/llvm/issues/7179>

¹³ https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

8 Conclusions

We have studied the implementation of GPU-accelerated FS using the mRMR algorithm across multiple programming backends (CUDA, HIP, and SYCL), analyzing how autotuning techniques can be used to obtain efficient configurations across different execution environments. In this work, we formulated the optimization process as a search problem and applied BO with TL to tune the performance parameters of each backend–platform combination. On average, the search required only 49 evaluations per scenario, representing a small fraction of the total search space.

The performance evaluation showed that when properly tuned, CUDA and HIP achieve comparable performance on NVIDIA platforms. This indicates that HIP can effectively exploit the underlying hardware while maintaining a CUDA-like programming model, making it a practical option when targeting multiple platforms with similar codebases. In contrast, SYCL exhibited a moderate performance gap in our experiments on NVIDIA GPUs, which may be related to differences in runtime behavior, compiler maturity, and the lack of certain low-level optimization mechanisms available in CUDA and HIP at the time of this study. The results also illustrate the importance of autotuning in adapting performance parameters to each backend and execution environment. The use of BO enabled the efficient exploration of a large configuration space and consistently identified high-performance configurations without requiring exhaustive searches.

Acknowledgements We gratefully thank the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the access granted to their supercomputing resources.

Author Contributions B.B.: Methodology, Software, Validation, Writing - original draft, Writing - review & editing, Visualization. J.G-D.: Conceptualization, Writing - original draft, Writing - review & editing, Supervision, Funding acquisition. A.P.D.: Resources, Methodology, Software, Writing - original draft, Writing - review & editing, Supervision. K.Z.I.: Resources, 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 grants PID2022-136435NB-I00 and PID2025-167767NB-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 Next-GenerationEU/RRF; FPU predoctoral grant of Bieito Beceiro ref. FPU20/00997, funded by the Ministry of Science, Innovation and Universities; and INDITEX-UDC Predoctoral Research Stay Grants Program 2024. In addition, this material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program under Award Number DE-AC02-05CH11231. 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/>.

Declarations

Code availability The code for the CUDA, HIP, and SYCL backends for mRMR used in the experimental evaluation is available at <https://gitlab.com/bieito/gpufs>.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Guyon I, Elisseeff A (2003) An introduction to variable and feature selection. *J Mach Learn Res* 3:1157–1182 (<https://www.jmlr.org/papers/v3/guyon03a.html>)
2. Bolón-Canedo V, Sánchez-Marño N, Alonso-Betanzos A (2013) A review of feature selection methods on synthetic data. *Knowl Inf Syst* 34(3):483–519. <https://doi.org/10.1007/s10115-012-0487-8>
3. Peng H, Long F, Ding C (2005) 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. <https://doi.org/10.1109/TPAMI.2005.159>
4. Papaioannou N, Myllis G, Tsimpliris A, Vrana V (2025) The role of mutual information estimator choice in feature selection: an empirical study on mRMR. *Information* 16(9):724. <https://doi.org/10.3390/info16090724>
5. Bhadra T, Mallik S, Hasan N, Zhao Z (2022) Comparison of five supervised feature selection algorithms leading to top features and gene signatures from multi-omics data in cancer. *BMC Bioinformatics* 23(Suppl 3):153. <https://doi.org/10.1186/s12859-022-04678-y>
6. Beceiro B, González-Domínguez J, Morán-Fernández L, Bolón-Canedo V, Touriño J (2024) CUDA acceleration of MI-based feature selection methods. *J Parallel Distrib Comput* 190:104901. <https://doi.org/10.1016/j.jpdc.2024.104901>
7. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA: is CUDA the parallel programming model that application developers have been waiting for? *Queue* 6(2):40–53. <https://doi.org/10.1145/1365490.1365500>
8. Smirnov V, Storublevtcev N (2019) Development tools for heterogeneous computing. In: *Computational Science and its Applications – ICCSA 2019. Lecture Notes in Computer Science*, vol. 11622, pp. 812–824. Springer, Cham. https://doi.org/10.1007/978-3-030-24305-0_62
9. Ramírez-Gallego S, Lastra I, Martínez-Rego D, Bolón-Canedo V, Benítez JM, Herrera F, Alonso-Betanzos A (2017) Fast-mRMR: fast minimum redundancy maximum relevance algorithm for high-dimensional big data. *Int J Intell Syst* 32(2):134–152. <https://doi.org/10.1002/int.21833>
10. Cuomo S, Galletti A, Marcellino L, Navarra G, Toraldo G (2018) On GPU-CUDA as preprocessing of fuzzy-rough data reduction by means of singular value decomposition. *Soft Comput* 22(5):1525–1532. <https://doi.org/10.1007/s00500-017-2887-x>
11. Venkatesh B, Anuradha J (2021) Fuzzy rank based parallel online feature selection method using multiple sliding windows. *Open Comput Sci* 11(1):275–287. <https://doi.org/10.1515/comp-2020-0169>
12. Awan MG, Deslippe J, Buluc A, Selvitopi O, Hofmeyr S, Olikier L, Yelick K (2020) ADEPT: a domain independent sequence alignment strategy for GPU architectures. *BMC Bioinformatics* 21:406. <https://doi.org/10.1186/s12859-020-03720-1>
13. Haseeb M, Ding N, Deslippe J, Awan M (2021) Evaluating performance and portability of a core bioinformatics kernel on multiple vendor GPUs. In: *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 68–78. <https://doi.org/10.1109/P3HPC54578.2021.00010>
14. Jin Z, Vetter JS (2023) Understanding performance portability of SYCL kernels: a case study with the all-pairs distance calculation in bioinformatics on GPUs. In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 366–372. <https://doi.org/10.1109/IPDPSW59300.2023.00067>
15. Martin A, Liu G, Ladd W, Lee S, Gounley J, Vetter J, Patel S, Rizzi S, Mateevitsi V, Insley J, Randles A (2023) Performance evaluation of heterogeneous GPU programming frameworks for hemodynamic

- simulations. In: Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W'23), pp. 1126–1137. <https://doi.org/10.1145/3624062.3624188>
16. Costanzo M, Rucci E, García-Sánchez C, Naiouf M, Prieto-Matías M (2023) Comparing performance and portability between CUDA and SYCL for protein database search on NVIDIA. AMD and Intel GPUs. In: 2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 141–148. <https://doi.org/10.1109/SBAC-PAD59825.2023.00023>
 17. Faqir-Rhazoui Y, García C (2024) SYCL in the edge: performance and energy evaluation for heterogeneous acceleration. *J Supercomput* 80(10):14203–14223. <https://doi.org/10.1007/s11227-024-05957-6>
 18. Ozturk ME, Asudeh O, Sabin G, Sadayappan P, Sukumaran-Rajam A (2023) A performance portability study using tensor contraction benchmarks. In: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 591–600. <https://doi.org/10.1109/IPDPSW59300.2023.00102>
 19. Kim J, Sukumaran-Rajam A, Thumma V, Krishnamoorthy S, Panyala A, Pouchet L-N, Rountev A, Sadayappan P (2019) A code generator for high-performance tensor contractions on GPUs. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 85–95. <https://doi.org/10.1109/CGO.2019.8661182>
 20. Antepara O, Williams S, Johansen H, Zhao T, Hirsch S, Goyal P, Hall M (2023) Performance portability evaluation of blocked stencil computations on GPUs. In: Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W'23), pp. 1007–1018. <https://doi.org/10.1145/3624062.3624177>
 21. Zhao T, Basu P, Williams S, Hall M, Johansen H (2019) Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19), p. 52. <https://doi.org/10.1145/3295500.3356210>
 22. Wu X, Balaprakash P, Kruse M, Koo J, Videau B, Hovland P, Taylor V, Geltz B, Jana S, Hall M (2025) ytopt: autotuning scientific applications for energy efficiency at large scales. *Concurr Comput Practice Exp* 37(1):8322. <https://doi.org/10.1002/cpe.8322>
 23. Menon H, Bhatele A, Gambin T (2020) Auto-tuning parameter choices in HPC applications using Bayesian optimization. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 831–840. <https://doi.org/10.1109/IPDPS47924.2020.00090>
 24. Roy RB, Patel T, Gadepally V, Tiwari D (2021) Bliss: Auto-tuning complex applications using a pool of diverse lightweight learning models. In: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21), pp. 1280–1295. <https://doi.org/10.1145/3453483.3454109>
 25. van Werkhoven B (2019) Kernel Tuner: a search-optimizing GPU code auto-tuner. *Futur Gener Comput Syst* 90:347–358. <https://doi.org/10.1016/j.future.2018.08.004>
 26. Dorier M, Egele R, Balaprakash P, Koo J, Madireddy S, Ramesh S, Malony AD, Ross R (2022) HPC storage service autotuning using variational-autoencoder-guided asynchronous Bayesian optimization. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER), pp. 381–393. <https://doi.org/10.1109/CLUSTER51413.2022.00049>
 27. Dieguez AP, Choi M, Zhu X, Wong BM, Ibrahim KZ (2022) ML-based performance portability for time-dependent density functional theory in HPC environments. In: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 1–12. <https://doi.org/10.1109/PMBS56514.2022.00006>
 28. Liu Y, Sid-Lakhdar WM, Marques O, Zhu X, Meng C, Demmel JW, Li XS (2021) GPTune: multitask learning for autotuning exascale applications. In: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'21), pp. 234–246. <https://doi.org/10.1145/3437801.3441621>
 29. Brown G, Pocock A, Zhao M-J, Luján M (2012) Conditional likelihood maximisation: a unifying framework for information theoretic feature selection. *J Mach Learn Res* 13:27–66 (<https://jmlr.org/papers/v13/brown12a.html>)
 30. Mockus JB, Mockus LJ (1991) Bayesian approach to global optimization and application to multi-objective and constrained problems. *J Optim Theory Appl* 70(1):157–172. <https://doi.org/10.1007/bf00940509>
 31. Malakar P, Balaprakash P, Vishwanath V, Morozov V, Kumaran K (2018) Benchmarking machine learning methods for performance modeling of scientific applications. In: 2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 33–44. <https://doi.org/10.1109/PMBS.2018.8641686>

32. Chang C-C, Linn C-J (2011) LIBSVM: a library for support vector machines. *ACM Trans Intell Syst Technol* 2(3):27. <https://doi.org/10.1145/1961189.1961199>
33. Lin W-C, Deakin T, McIntosh-Smith S (2024) A metric for HPC programming model productivity. In: *Workshops of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC-W'24)*, pp. 1192–1205. <https://doi.org/10.1109/SCW63240.2024.00160>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.