CUD@ILP: a GPU-based Massively Scalable Inductive Learning Algorithm*

Enrico Santi, Agostino Dovier and Andrea Formisano*

Università degli Studi di Udine, DMIF, Udine, Italy
Gruppo Nazionale per il Calcolo Scientifico — INdAM, Roma, Italy

Abstract

Inductive Logic Programming developed within Logic Programming in the Nineties; nowadays several ILP frameworks capable of learning monotonic and non monotonic programs from hypotheses and examples exist. In this paper we focus on FOLD-RM, an inductive learning algorithm capable of learning hypotheses for multi-classification problems without the need of using any additional ASP solver. We describe how such an algorithm can be effectively scaled to deal with real world datasets by means of a massive parallel implementation. We show that, by exploiting the massive computational power offered by modern GPUs, our parallel implementation CUD@ILP significantly outperforms FOLD-RM in all the tests considered.

Keywords

Inductive Logic Programming, GPU parallelism, Machine Learning

1. Introduction

Graphics processing units (GPUs) are responsible for the widespread diffusion of many machine learning techniques since they have accelerated training and inference of many different machine learning algorithms. Such a success has only been replicated with modest results in automated reasoning: while the usage of parallel computing, especially CPU-based multi-threaded solutions, has been experimented quite widely and with noticeable results in the automated reasoning field [1, 2, 3], only some attempts have been carried out to include also GPU parallel processing. In more recent years, some approaches have shifted the focus towards the usage of GPU based massive parallelism. Approaches space from implementing parallel specific constraint propagators into existing solvers [4, 5] to developing GPU based solvers for Constraint Programming, Satisfiability testing, and Answer Set Programming (ASP) [6, 7, 8]. However, parallelizing Inductive Logic Programming (ILP) frameworks comes with its set of additional challenges, since the majority of such tools rely on external software, such as ASP solvers, to compute the solution [9, 10, 11]. Recently, ILP has been proved to be also a promising approach to Explainability of (black box) AI, even if its use on real-world datasets is restricted by the excessive running time [12, 13].

In this paper, we recall FOLD-RM [14], an ILP framework that does not rely on external solvers, libraries, or tools. We propose and implement CUD@ILP, a GPU parallel version of FOLD-RM based on CUDA. Experimental results show a significant speedup and major scalability of CUD@ILP with respect to its serial counterpart.

The paper is structured as follows: Section 2 recalls some basic information about logic programming, inductive logic programming, and briefly overviews FOLD-RM. Section 3 describes the implementation of CUD@ILP, while Section 4 presents the datasets that were used to compare CUD@ILP with FOLD-RM and the relative results. Lastly, Section 5 draws some conclusions on CUD@ILP while also indicating some future developments of the tool.

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy

*Research partially supported by Interdepartment Project on AI (Strategic Plan UniUD–22-25), and by Unione europea-Next Generation EU, m.4 c.2, project MaPSART "Future Artificial Intelligence (FAIR)", PNRR.
Enrico Santi is supported by FSE/FVG PhD Grant on "XAI-FVG Explainability of Weather Forecasting in FVG" (G23C23001130008).

*Corresponding author.
enrico.santi@uniud.it (E. Santi); agostino.dovier@uniud.it (A. Dovier); andrea.formisano@uniud.it (A. Formisano)
0009-0008-3556-7177 (E. Santi); 0000-0003-2052-8593 (A. Dovier); 0000-0002-6755-9314 (A. Formisano)

2. Background

2.1. Preliminary

We start this section by briefly recalling some fundamental notions of logic programming. Given a set of constant symbols C, variable symbols V, predicate symbols P, and function symbols F, the syntax of logic programming is defined as follows. A *term* is defined inductively as a constant, a variable, or $f(t_1,\ldots,t_n)$, where $f \in F$ is an n-ary function symbol and $t_1,\ldots,t_n$ are terms. An *atom*, is an expression of the form $p(t_1,\ldots,t_n)$ where $p \in P$ is an n-ary predicate symbol and $t_1,\ldots,t_n$ are terms. An atom or a term is said to be *ground* if it does not contain any variables. A *normal rule*, from now on simply referred to as a *rule*, is an expression of the form

$$h \leftarrow \ell_1,\ldots,\ell_m,\texttt{not}\ \ell_{m+1},\ldots,\texttt{not}\ \ell_n$$

where h and $\ell_1,\ldots,\ell_n$, are atoms. The `not` symbol denotes negation as failure. The atom h is the head of the rule, while the conjunction of literals in the body $(\ell_1,\ldots,\ell_m,\texttt{not}\ \ell_{m+1},\ldots,\texttt{not}\ \ell_n)$ specifies when the head holds. In this paper we adopt Prolog-style notation and denote the implication symbol $\leftarrow$ by `:-`.

A normal logic program is a finite set of rules. A predicate occurring in the head of a rule positively (resp., negatively) depends on all predicate occurring in positive (resp., negative) literals of the rule body. A normal logic program is *stratified* if the set of its predicates can be partitioned into layers such that, for each layer λ, each rule head in λ does not positive depend on predicates of layers higher than λ, while negative dependencies can only exist with predicates in strictly lower layers.

In this paper we adopt the Answer Set Programming (ASP) semantics. A normal logic program can admit none, one or multiple solutions, called *answer sets*, each one corresponding to a possible model of the program. Stratified normal programs admit a unique answer set.

2.2. Inductive Logic Programming

Inductive logic programming is a subfield of both machine learning and automated reasoning, namely, in the intersection of the two main areas of artificial intelligence [15]. The main goal of ILP is learning models represented as logical rules, starting from a set of *examples* and a *background knowledge* written in some language of logic programming (typically, Prolog or ASP). The learning process consists in searching a *hypothesis space* for rules that are coherent with prior knowledge and that best explain the examples provided. Formally, an ILP task is a 4-tuple $\langle B, S_M, E^+, E^- \rangle$ where:

- B is a set of rules composing the *background knowledge*.
- S_M is a set of rules called hypotheses space.
- E^+ and E^- are disjoint sets of (positive) ground atoms.

Notice that, the hypotheses space S_M can be given explicitly by listing all the rules, or implicitly, for example, by restricting the possible atoms in the body of the rules to the predicates used in the examples, as done in FOLD-RM, or by specifying mode biases as in FastLAS [10].

For coherence with the notation adopted in the description of the FOLD-RM algorithm (cf. Section 2.3) we will refer to the positive (resp., negative) part of the body of the rule as *default* (resp., *abnormal*).

Among the many existing ILP learning settings (cf. [9, 16]), we consider one of the most widespread, namely, entailment based ILP. Its goal, when restricted to the case where $B \cup H$ is stratified, is to find a hypothesis H such that:

- $\forall e \in E^+,\ H \cup B \models e \quad (completeness)$
- $\forall e \in E^-,\ H \cup B \not\models e \quad (consistency)$

Namely, in *all* the models of $H \cup B$ all positive examples are true and no negative example is true (cautious reasoning).

Species	is_fish	is_whale	is_mammal	Habitat
Humpback whale	0	1	1	Water
Blue whale	0	1	1	Water
Atlantic salmon	1	0	0	Water
African elephant	0	0	1	Land

```
B = { species("humpback whale").
      species("blue whale").
      species("atlantic salmon").
      species("african elephant").
      is_whale("humpback whale").
      is_mammal("humpback whale").
      is_whale("blue whale").
      is_mammal("blue whale").
      is_fish("atlantic salmon").
      is_mammal("african elephant"). }

e_1 = { habitat("humpback whale", water). }
e_2 = { habitat("blue whale", water). }
e_3 = { habitat("atlantic salmon", water). }
e_4 = { habitat("african elephant", land). }
```

Figure 1: Top: a simple CSV dataset, the yellow column depicts the predicate to learn. Bottom: an encoding of the corresponding background knowledge and positive examples.

2.3. FOLD-RM

FOLD-RM[1] is an inductive learning algorithm that allows to learn a *stratified* normal logic program (a hypothesis), capable of dealing with multi-class classification problems [14]. A classification problem is defined as a task where the training data comprises examples (usually given as vectors) along with the corresponding category associated for each example, where the objective is to assign each input example to one of a finite number of discrete categories, called classes [17]. A class represents a label that groups together examples sharing common features. The task is said to be binary if the number of possible classes is two. It is a multi-class task if the number is greater than two.

While many algorithms and frameworks to perform ILP exist [18], the majority of them rely on existing systems such as ASP solvers or Prolog interpreters to compute/filter a set of hypotheses rules. FOLD-RM on the other hand, is a standalone implementation, not requiring any additional tool to solve an ILP task. This independence from other systems is in part justified by the stratification of $B \cup H$, and this is a perfect benchmark for experimenting parallelism.

Facts are allowed in the background knowledge B of a FOLD-RM task. The hypothesis learned by FOLD-RM is characterized by two distinct sets of normal rules:

- The set of *labeling rules* that contains all the rules where each head is a predicate describing a class within the classification task.
- The set of *abnormal rules* that contains the rules defining predicates which can only be used in the *abnormal part* of the rules (both labeling and abnormal).

The head of the abnormal rules, which can only be used as predicates in the abnormal part of any rule, are used to exclude false positive misclassifications. For instance, consider a simple classification task where the objective is to learn a `habitat` relation on a set of animals and their characteristics given as examples (see Figure 1). An example of learned hypothesis, taken from [14], is the following:

```
habitat(X,land)  :- species(X), is_mammal(X), not ab1(X).
habitat(X,water) :- species(X), is_fish(X).
```

[1]FOLD stands for *First Order Learner of Defaults*. It is available at https://github.com/hwd404/FOLD-RM.

```
habitat(X,water) :- species(X), is_whale(X).
ab1(X) :- species(X), is_whale(X).
```

The first three rules in this example compose the labeling rules of the hypothesis, while the last one is an abnormal rule.

FOLD-RM can only learn stratified normal logic programs [14] and the current FOLD-RM implementation only admits a set of facts as *background knowledge*. Nonetheless, the tool results very accessible even for users not familiar with logic programming, because the examples used to learn a model can be input directly via a CSV file, without requiring any modeling or logic programming knowledge.

Figure 1 presents a simple CSV dataset together with its corresponding representation as ASP examples. Each row of the CSV depicts an example and the columns (features) are used to derive the predicates which will be used in the bodies of learned rules.

Algorithm 1 recalls the main procedure of FOLD-RM. Once loaded the training dataset, FOLD-RM works as follows:

1. Consider all remaining examples (at the beginning all the examples in the training dataset). A class c to predict is selected, the most frequent class that labels the remaining examples is used (call of function `most()` in Algorithm 1, line 4).
2. Split the remaining examples according to whether the label is equal to the most frequent one or not. The examples with label equal to c compose E^+, all remaining examples make up E^- (`split_by_literal()`, line 5 in Algorithm 1).
3. Learn a rule (and relative abnormal rules) to correctly classify the most number of examples in the chosen class (`learn_rule()`, line 6).
4. Compute the examples of the class still not covered by the learned rule and repeat the process until all examples are covered.

Profiling the functions called in the main loop of `FOLD-RM()` revealed that `learn_rule()` (also listed in Algorithm 1) accounts for the vast majority of the computational load, while the execution time of other functions is negligible. Consequently, we decided to parallelize only such function as CUDA kernels of CUD@ILP. We will discuss this implementation in Section 3.3. Algorithm 2 lists the main functions called by `learn_rule()` and that will be parallelized. We left essentially unchanged the remaining functions called by `FOLD-RM()` (i.e., `find_most()`, `split_by_literal()`, and some additional functions for whose descriptions we refer the reader to [14]).

3. The CUD@ILP Algorithm

In this section we outline the main design choices we made in implementing a GPU-empowered variant of FOLD-RM. Since on the device (i.e., GPU) only numerical data and simple data structures are allowed (there is no support for strings, for instance), we developed data pre- and post-processing procedures, presented in Section 3.2. In Section 3.3 the kernel implementation is discussed.

We briefly recall that in CUDA programming, kernels are functions called into execution by the CPU (host) and execute in parallel on the GPU (device).

3.1. Where Parallelism Can Be Exploited in FOLD-RM

As mentioned, profiling revealed that the most computationally intense function of FOLD-RM is `learn_rule()`; actually, on the set of benchmarks we used, it is responsible for more than the 70% of the overall computation time (see Figure 3). The reason is that a single call to `find_best_literal()` in `learn_rule()` (Algorithm 1, line 16) requires to scan a consistent part of the whole dataset. For this reason, we mainly focused on the task of parallelizing `find_best_literal()`, and this allowed us to achieve a significant speedup (cf. Section 4).

We observed that for some of the instances in our benchmark set, other portions of the code exhibit relevant running time. This is the case, for instance, of the code performing the update of the set of examples still to be covered (i.e., the update of E in `FOLD_RM()`, lines 7 and 9 of Algorithm 1).

Algorithm 1: FOLD-RM Algorithm

Input: E: examples, *ratio*: exception ratio, B: background knowledge (as a global parameter)
Output: $R = \{r_1, \ldots, r_n\}$: a set of defaults rules with exceptions

```
1  function FOLD_RM(E)
2      R ← ∅
3      while |E| > 0 do
4          l ← most(E)                    // l: most popular target literal as the learning target
5          E⁺, E⁻ ← split_by_literal(E, l)
6          r ← learn_rule(E⁺, E⁻, ∅)
7          E_FN ← covers(r, E⁺, false)                    // get the examples not coverd by r
8          if |E_FN| = |E⁺| then break
9          E ← E⁻ ∪ E_FN                          // rule out the already covered examples
10         r ← add_head(r, l)
11         R ← R ∪ {r}
12     return R
13 function learn_rule(E⁺, E⁻, L_used)
14     L ← ∅
15     while true do
16         l ← find_best_literal(E⁺, E⁻, L_used)
17         L ← L ∪ {l}
18         r ← set_default(r, L)                          // set default part of rule r as L
19         E⁺ ← covers(r, E⁺, true)
20         E⁻ ← covers(r, E⁻, true)
21         if l is invalid or |E⁻| ≤ |E⁺| · ratio then
22             if l is invalid then                  // remove the invalid literal l from rule r
23                 r ← set_default(r, L \ {l})
24             else                          // learn exception rules for r and add them to the rule
25                 AB ← FOLD(E⁻, E⁺, L_used ∪ L)
26                 r ← set_exception(r, AB)
27             break
28     return r                                  // the head of rule r is a class to characterize
```

3.2. Pre- and Post-processing

CUDA does not provide native support for the string data type on the device, hence strings cannot be directly stored and used in GPU's memory. This poses a problem when dealing with categorical features of FOLD-RM. For this reason a pre- and post-processing phase has been implemented. The dataset in FOLD-RM is loaded in memory as a matrix implemented by a list of lists, which can contain heterogeneous data types. Consider the example presented in Section 2.3, the first column of the CSV contains a categorical feature and the second one a numerical one, so the first element of each list will be a string and the second one an integer. The pre-processing, carried out on the whole training dataset, consists in mapping all the possible values of categorical features to 64-bit integers. Additionally, since `NumPy` arrays must have a homogeneous data type, all numerical feature values are mapped to 64-bit integers. This last mapping step does not introduce any accuracy issues, even when converting floating-point values to integers, as FOLD-RM does not consider distances between numerical values, but only their relative ordering. Therefore, as long as the mapping for each numerical column preserves the ordering of values, the algorithm operates correctly. The obtained 64-bit integer matrix, representing the dataset, can then be copied once to the device. Such a pre-processing phase, implemented and optimized using `NumPy` primitives, requires a negligible amount of time even for large datasets. Once each rule has been learned (see Section 3.3) its representation is copied back to the host, and it is post-processed to substitute the 64-bit integer values with their proper original string values.

Algorithm 2: FOLD-RM auxiliary functions, Find Best Literal and best information gain

Input: E^+: positive examples, E^-: negative examples, L_{used}: used literals, B: background knowledge (as a global parameter)

Output: *best_lit*: the best literal that provides the most information

function `find_best_literal`(E^+, E^-, L_{used})
  $best\_ig, best\_lit \leftarrow -\infty, invalid$
  for $i \leftarrow 1$ **to** N **do** // N is the number of features, i.e., columns in the CSV
    $ig, lit \leftarrow$ `best_info_gain`(E^+, E^-, i, L_{used})
    if $best\_ig < ig$ **then** $best\_ig, best\_lit \leftarrow ig, lit$
  return $best\_lit$

function `best_info_gain`(E^+, E^-, i, L_{used})
  // i: feature index
  // pos (neg): dictionaries holding the frequencies of the values in E^+ (E^-)
  $pos, neg \leftarrow$ `count_classification`(E^+, E^-, i)
  // xs (cs): lists holding the unique numerical (categorical) values
  $xs, cs \leftarrow$ `collect_unique_values`(E^+, E^-, i)
  // xp (xn): the total number of E^+ (E^-) with numerical values
  // cp (cn): the total number of E^+ (E^-) with categorical values
  $xp, xn, cp, cn \leftarrow$ `count_total`(E^+, E^-, i)
  $xs \leftarrow$ `counting_sort`(xs)
  for $j \leftarrow 1$ **to** `size`(xs) **do** // compute prefix sum for E^+ & E^- numerical values
    $pos[xs_i] \leftarrow pos[xs_i] + pos[xs_{i-1}]$
    $neg[xs_i] \leftarrow neg[xs_i] + neg[xs_{i-1}]$
  for $x \in xs$ **do** // compute information gain for numerical comparison literals
    $lit\_dict[literal(i, \leq, x)] \leftarrow$ `IG`$(pos[x], xp - pos[x] + cp, xn - neg[x] + cn, neg[x])$
    $lit\_dict[literal(i, >, x)] \leftarrow$ `IG`$(xp - pos[x], pos[x] + cp, neg[x] + cn, xn - neg[x])$
  for $c \in cs$ **do** // compute info gain for equality comparison literals
    $lit\_dict[literal(i, =, c)] \leftarrow$ `IG`$(pos[c], cp - pos[c] + xp, cn - neg[c] + xn, neg[c])$
    $lit\_dict[literal(i, \neq, c)] \leftarrow$ `IG`$(cp - pos[c] + xp, pos[c], neg[c], cn - neg[c] + xn)$
  $best\_ig, lit \leftarrow$ `best_pair`(lit_dict, L_{used})
  return $best\_ig, lit$ // return the best info gain and its corresponding literal

function *IG*(tp, fn, tn, fp)
  if $tp + tn < fp + fn$ **then return** -1×10^{20}
  $tot\_p \leftarrow tp + fp$
  $tot\_n \leftarrow tn + fn$
  $tot \leftarrow tot\_p + tot\_n$
  $ret \leftarrow 0.0$
  if $tp > 0.0$ **then** $ret \leftarrow ret + \left(\frac{tp}{tot}\right) \times \ln\left(\frac{tp}{tot\_p}\right)$
  if $fp > 0.0$ **then** $ret \leftarrow ret + \left(\frac{fp}{tot}\right) \times \ln\left(\frac{fp}{tot\_p}\right)$
  if $tn > 0.0$ **then** $ret \leftarrow ret + \left(\frac{tn}{tot}\right) \times \ln\left(\frac{tn}{tot\_n}\right)$
  if $fn > 0.0$ **then** $ret \leftarrow ret + \left(\frac{fn}{tot}\right) \times \ln\left(\frac{fn}{tot\_n}\right)$
  return ret

3.3. The CUD@ILP Implementation

In this section, the implementation of the main kernels of CUD@ILP is discussed, also addressing some of the main design choices we made.

Let us recall that each computation on the device is described as a collection of concurrent threads, each executing the same function, called kernel. Threads are hierarchically organized in equally sized blocks that are in turn structured in a grid and executed in warps (groups of 32 threads).

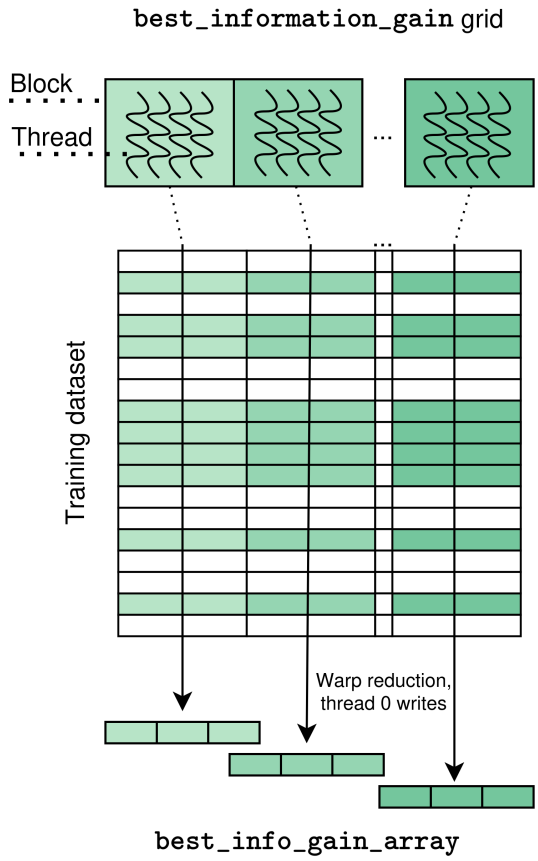

Figure 2: Overview of the how the different blocks of the kernel `find_best_literal()` behave.

As briefly anticipated, the main speedup has been achieved by replacing the `find_best_literal()` function with a kernel. Such kernel is executed on the device by a grid having one block per each feature in the dataset. This is possible because each feature is treated independently in `find_best_literal()`.

In the original FOLD-RM sequential function `find_best_literal()`, a single thread scans all examples that still need to be covered, for each feature. At the end a triple of values is returned. The triple contains the index of the column were the highest information gain was found, the splitting value as well as a comparison operator: $\{=, \neq\}$ for categorical features and $\{>, \leq\}$ for numerical features. Such a triple is then converted into a body literal in the rule being learned.

The kernel, on the other hand, uses a block of threads to scan and compute the information gain of a single column, moreover different blocks can independently scan different features. By using blocks of warp size (32 threads) no inter-block synchronization is needed and all the operations to be performed can be carried out via efficient intra-warp primitives.

At the end of each computation, the first thread of each block stored into an array `best_info_gain_array` its block id (equal to the index of the column), the information gain, and the computed comparison operator. After the grid execution completed, the array is transferred back to the host, where the element with the highest information gain is selected. Such a computation can be efficiently carried out on the host since in general number of features in a dataset is reasonably small.

Figure 2 sums up the `find_best_literal()` grid and the data each blocks considers. Alongside the `find_best_literal()` kernel, a second kernel, currently launched with a fixed number of blocks, parallelizing the update of the set of examples E^+, E^- (lines 19 and 20 of Algorithm 1) has been implemented. Such a kernel, while not scaling yet on an arbitrary grid, allows CUD@ILP to compute directly on the device the second most time consuming phase of FOLD-RM (cf. timing of function *cover()* in Figure 3).

Dataset name	Features no.	Trainining examples no.	Task classification type
Jannis [19]	46	∼ 70k	multiclass
MiniBooNE [20]	50	∼ 105k	binary
Human [21]	562	∼ 9k	binary
LifeStyle [22]	31	∼ 360k	multiclass
Sloan [23]	41	∼ 80k	multiclass
Diabetes [24]	21	∼ 203k	multiclass
Weather [25]	11	∼ 320k	binary
SmokeDrink [26]	23	∼ 392k	binary
CoverType [27]	54	∼ 465k	multiclass
MNIST [28]	784	∼ 60k	multiclass
Crops [29]	174	∼ 260k	multiclass
Earthquake [30]	29	∼ 610k	multiclass

Table 1
The datasets used for comparing FOLD-RM and CUD@ILP. For each dataset, the third column reports the size of the training set, which was obtained by selecting approximately 80% of the data.

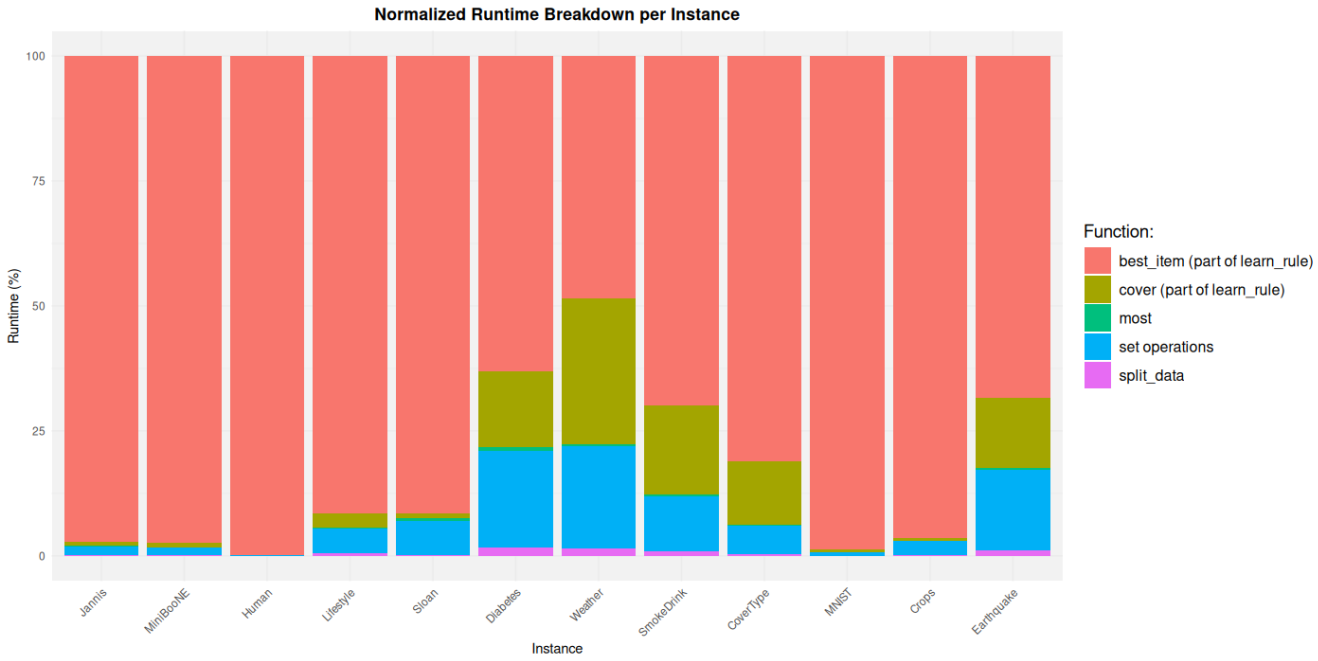

Figure 3: Stacked barplot presenting the average percentages of time used by the main functions of FOLD-RM for all the tested instances.

4. Experimental Results

4.1. Datasets

The datasets used to compare FOLD-RM and CUD@ILP concern *real-world* classification tasks in different fields, from medicine to engineering. They present a very different number of features, ranging from tens to hundreds, both numerical and categorical. Such datasets also contain a very wide range of examples, from tens to hundreds of thousands. This diversity enables a very realistic comparison of the two implementations. In Table 1 we have summarized the characteristics of the datasets we used.

For each dataset, we partitioned the data into approximately 80% training and 20% validation sets. The ILP solver uses the training portion to compute a set of hypotheses whose (unique) model contains all positive examples but excludes all negative ones. As usual, we used the remaining 20% portion of dataset as a validation set to check the behavior of the learned program when applied to unknown data.

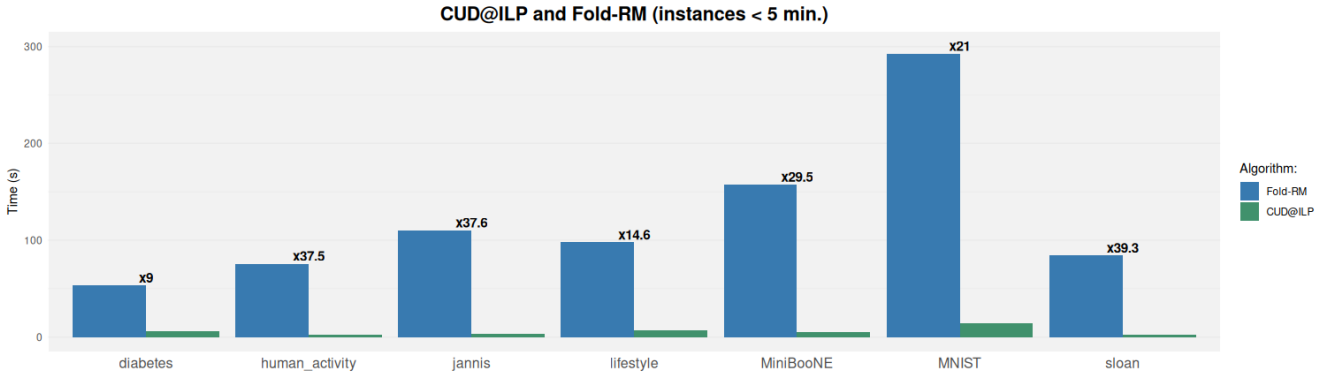

Figure 4: Barplot comparing the average serial and CUDA runtime obtained with five iterations. We have reported only the instances that were solved by FOLD-RM in less than 5 minutes.

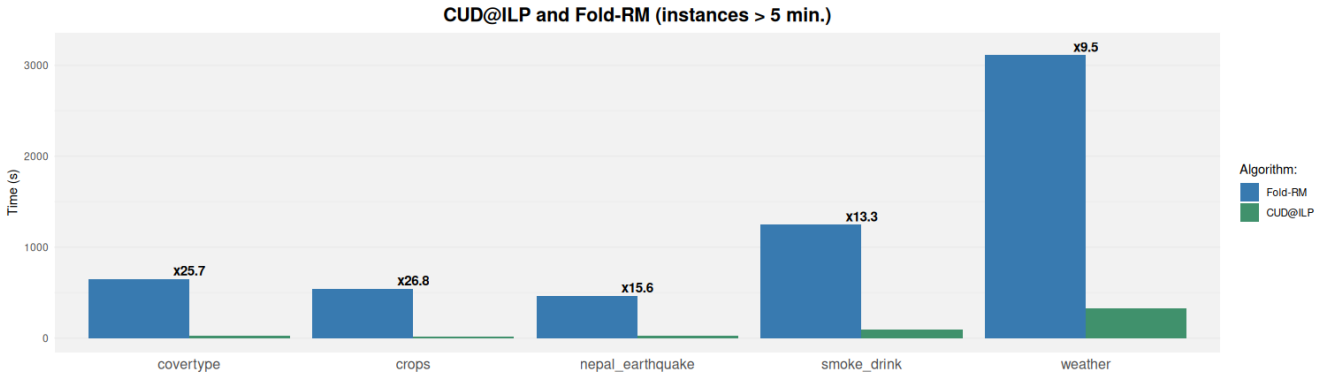

Figure 5: Barplot comparing the average serial and CUDA runtime obtained with five iterations. We have reported only the instances that were solved by FOLD-RM in more than 5 minutes.

4.2. Results

All the tests have been executed on a system equipped with an Intel Core i7-13700KF at base clock 3.4 GHz (up to 5.4 GHz in boost), 128 GB of DDR4 RAM and a NVIDIA GeForce RTX 4090 with 16K CUDA cores, grouped into 128 streaming multiprocessors, running at 2.23 GHz (up to 2.52 GHz). On the software side, the system runs Ubuntu 24.04 and CUDA 13.1.

Barplots summarizing the solve times of both implementations are shown in Figure 4 and 5. For clarity, the instances are grouped in the figures according to their serial runtime.

Note that during the tests, especially the smallest ones, GPU capabilities were not fully utilized. For this reason and for sake of completeness, we report that all the instances were also tested on a lower-end system equipped with a GPU RTX3060, with 3.5K CUDA cores grouped into 28 streaming multiprocessors, and a lower-end i7 processor as host. Results showed very similar speedups.

Although speedups vary with respect to both the number of `learn_rule()` loops and the dimensionality of the dataset, CUD@ILP consistently achieves an overall average speedup exceeding 13× across both systems.

We do not report results concerning the accuracies obtained by CUD@ILP because the parallelized algorithm simply executes in parallel the same computation FOLD-RM does. Hence, the accuracies obtained by the two implementations always coincide.

Source code, tests instances, and results are available at https://clp.dimi.uniud.it/sw.

5. Conclusions and Future Works

The GPU parallel version of the FOLD-RM algorithm described in this paper presents a significant speedup against its serial counterpart, making CUD@ILP an actually scalable algorithm usable on real-world datasets. Nonetheless, compared to ILP tools such as FastLAS [10], CUD@ILP remains easier to use due to CSV-formatted inputs. On the other hand FOLD-RM and CUD@ILP results are less general than FastLAS since only ground facts can be used as background knowledge. To this regard, future efforts will be put into extending CUD@ILP to consider stratified user-defined rules which can be used in the `learn_rule()` function to capture more expressive background knowledge. Additional effort will also be put into implementing CUDA kernels for set operations, which, as shown in Figure 3, may contribute significantly to the total computation time.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] Y. Hamadi, L. Sais (Eds.), Handbook of Parallel Constraint Reasoning, Springer, 2018. doi:10.1007/978-3-319-63516-3.

[2] I. P. Gent, I. Miguel, P. Nightingale, C. McCreesh, P. Prosser, N. C. A. Moore, C. Unsworth, A review of literature on parallel constraint solving, Theory Pract. Log. Program. 18 (2018) 725–758. doi:10.1017/S1471068418000340.

[3] A. Dovier, A. Formisano, G. Gupta, M. V. Hermenegildo, E. Pontelli, R. Rocha, Parallel logic programming: A sequel, Theory Pract. Log. Program. 22 (2022) 905–973. doi:10.1017/S1471068422000059.

[4] F. Tardivo, A. Dovier, A. Formisano, L. Michel, E. Pontelli, Constraint propagation on GPU: A case study for the alldifferent constraint, J. Log. Comput. 33 (2023) 1734–1752. doi:10.1093/LOGCOM/EXAD033.

[5] E. Santi, A. Dovier, A. Formisano, F. Tardivo, GPU accelerated compact-table propagation, Theory Pract. Log. Program. 25 (2025) 756–774. doi:10.1017/S1471068425100197.

[6] P. Talbot, A GPU-based constraint programming solver, in: S. Koenig, C. Jenkins, M. E. Taylor (Eds.), 40th AAAI Conference on Artificial Intelligence, 38th Conference on Innovative Applications of Artificial Intelligence, 16th Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, AAAI Press, 2026, pp. 14331–14341. doi:10.1609/AAAI.V40I17.38448.

[7] A. Dal Palù, A. Dovier, A. Formisano, E. Pontelli, CUD@SAT: SAT solving on GPUs, J. Exp. Theor. Artif. Intell. 27 (2015) 293–316. doi:10.1080/0952813X.2014.954274.

[8] A. Dovier, A. Formisano, F. Vella, GPU-based parallelism for ASP-solving, in: P. Hofstedt, S. Abreu, U. John, H. Kuchen, D. Seipel (Eds.), Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Revised Selected Papers, volume 12057 of *Lecture Notes in Computer Science*, Springer, Cham, 2019, pp. 3–23. doi:10.1007/978-3-030-46714-2_1.

[9] M. Law, Inductive learning of answer set programs, Ph.D. thesis, Imperial College London, UK, 2018. URL: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.762179.

[10] M. Law, A. Russo, E. Bertino, K. Broda, J. Lobo, FastLAS: Scalable inductive logic programming incorporating domain-specific optimisation criteria, in: 34th AAAI Conference on Artificial Intelligence, 32nd Conference on Innovative Applications of Artificial Intelligence, 10th Symposium on Educational Advances in Artificial Intelligence, AAAI 2020, AAAI Press, 2020, pp. 2877–2885. doi:10.1609/AAAI.V34I03.5678.

[11] R. Borelli, A. Dovier, Learning from answer sets via single-shot disjunctive ASP encoding, in: S. Koenig, C. Jenkins, M. E. Taylor (Eds.), 40th AAAI Conference on Artificial Intelligence, 38th

Conference on Innovative Applications of Artificial Intelligence, 16th Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, AAAI Press, 2026, pp. 18961–18968. doi:10.1609/AAAI.V40I23.38967.

[12] T. Dreossi, A. Dovier, A. Formisano, M. Law, A. Manzato, A. Russo, M. Tait, Towards explainable weather forecasting through FastLAS, in: C. Dodaro, G. Gupta, M. V. Martinez (Eds.), Logic Programming and Nonmonotonic Reasoning - 17th International Conference, LPNMR 2024, Dallas, TX, USA, October 11-14, 2024, Proceedings, volume 15245 of *Lecture Notes in Computer Science*, Springer, Cham, 2024, pp. 262–275. doi:10.1007/978-3-031-74209-5_20.

[13] A. Dovier, T. Dreossi, A. Formisano, B. Strizzolo, XAI-LAW: A logic programming tool for modeling, explaining, and learning legal decisions, in: M. Gebser, D. Inclezan, F. Ricca, M. Carro, M. Truszczynski (Eds.), Proceedings 41st International Conference on Logic Programming, ICLP 2025, volume 439 of *EPTCS*, 2025, pp. 405–419. doi:10.4204/EPTCS.439.28.

[14] H. Wang, F. Shakerin, G. Gupta, FOLD-RM: A scalable, efficient, and explainable inductive learning algorithm for multi-category classification of mixed data, Theory Pract. Log. Program. 22 (2022) 658–677. doi:10.1017/S1471068422000205.

[15] S. H. Muggleton, Inductive logic programming, New Gener. Comput. 8 (1991) 295–318. doi:10.1007/BF03037089.

[16] A. Cropper, S. Dumancic, Inductive logic programming at 30: A new introduction, J. Artif. Intell. Res. 74 (2022) 765–850. doi:10.1613/JAIR.1.13507.

[17] C. M. Bishop, Pattern recognition and machine learning, 5th Edition, Information science and statistics, Springer, 2007. URL: https://www.worldcat.org/oclc/71008143.

[18] H. V. Agun, Inductive logic programming beyond Prolog: Solvers and optimization, SSRN (2025). doi:10.2139/ssrn.5682668.

[19] J. Thomas, Jannis dataset, https://www.openml.org/search?type=data&id=41168, 2018. Accessed: 2026-04-13.

[20] P. Gijsbers, Miniboone, https://www.openml.org/search?type=data&id=41150, 2018.

[21] D. Anguita, A. Ghio, L. Oneto, X. Parra, J. L. Reyes-Ortiz, Human activity with smartphones dataset, https://www.kaggle.com/datasets/georgesaavedra/logistic-regression, 2022.

[22] A. Therrien, Half a million lifestyle, 2023. doi:10.34740/KAGGLE/DS/4094599.

[23] A. Almeida, et al., The eighteenth data release of the Sloan Digital Sky Surveys: Targeting and first spectra from SDSS-V, https://www.kaggle.com/datasets/diraf0/sloan-digital-sky-survey-dr18, 2023.

[24] A. Teboul, Diabetes health indicators dataset, https://www.kaggle.com/datasets/alexteboul/diabetes-health-indicators-dataset, 2022.

[25] M. Riyas, TN weather 2020-2025, 2025. doi:10.34740/KAGGLE/DSV/12975437.

[26] Soo.Y, Smoking and drinking dataset, https://www.kaggle.com/datasets/sooyoungher/smoking-drinking-dataset, 2024. Accessed: 2026-04-13.

[27] J. Blackard, Covertype, UCI Machine Learning Repository, 1998. doi:10.24432/C50K5N.

[28] D. Dato-on, MNIST in CSV format, https://www.kaggle.com/datasets/oddrationale/mnist-in-csv, 2018. Accessed: 2026-04-13.

[29] Crop mapping using fused optical-radar data set, UCI Machine Learning Repository, 2020. doi:10.24432/C5G89D.

[30] A. Acharya, Nepal 2015: Building damage & seismic intensity, https://www.kaggle.com/datasets/sirorororo/nepal-earthquake-ward-level-intensities-and-impact, 2024.