An OpenCL implementation of a forward sampling algorithm for CP-logic

Wiebe Van Ranst, Joost Vennekens*

KU Leuven, Dept. of Computer Science, Campus De Nayer, J.-P. De Nayerlaan 5, 2860 Sint-Katelijne-Waver, Belgium

ARTICLE INFO

Article history:
Received 14 December 2014
Received in revised form 27 April 2015
Accepted 20 May 2015
Available online 28 May 2015

Keywords:
Probabilistic logic programming
CP-logic
OpenCL
Monte Carlo methods

ABSTRACT

We present an approximate query answering algorithm for the Probabilistic Logic Programming language CP-logic. It complements existing sampling algorithms by using the rules from body to head instead of in the other direction. We present an implementation in OpenCL, which is able to exploit the multicore architecture of modern GPUs to compute a large number of samples in parallel, and demonstrate that this is competitive with existing inference algorithms.

1. Introduction

Probabilistic graphical models, such as Bayesian networks [12], have enjoyed great success in a variety of applications. However, their limited expressiveness makes certain patterns of dependencies between random variables difficult to express, and may also lead to models that are too unwieldy to interpret or maintain. In an effort to improve on this, the area of Probabilistic Logic Learning (PLL) leverages concepts and methods from Logic Programming (and/or other logical formalisms) to develop languages with better knowledge representation properties. In other words, the goal is to make models that are more readable, more compact and more modular. Unlike the propositional representation offered by Bayesian networks, PLL models tend to be first-order, using the power of logical variables and logical formulas to describe the relations between a large number of random variables by a comparatively small number of expressions.

An important family of PLL languages has originated from the Distribution Semantics, originally developed by Sato [19]. Examples of languages in this family are Sato’s own PRISM language [19], the Independent Choice Logic [13] and CP-logic [25]. Among these languages, CP-logic offers the most general syntax and will therefore be our focus in this article, even though our results could apply equally to other languages in this family. In all of these languages, a user constructs a set of Logic Programming-like rules, which contain a number of probabilistic parameters. The precise value of these parameters can be supplied by the user or learned from data. Once their values are known, the model can then be used for a variety of inference tasks, such as prediction or explaining observations. A core inference task, which is needed in many applications and also plays a key role in the Machine Learning algorithms that are typically used to estimate the probabilistic parameters, is that of computing the probability of a query. Because, for instance, the ML algorithms need to perform this task numerous times, an efficient implementation is crucial. Here, the additional expressivity of PLL languages may prove problematic.

* Corresponding author.

E-mail addresses: wiebe.vanranst@kuleuven.be (W. Van Ranst), joost.vennekens@kuleuven.be (J. Vennekens).

http://dx.doi.org/10.1016/j.ijar.2015.05.008
0888-613X/© 2015 Elsevier Inc. All rights reserved.
Different algorithms already exist for computing the probability of a query in CP-logic. The Problog system (http://dtai.cs.kuleuven.be/problog) supports the CP-logic language with algorithms based on weighted CNFs [4] and Binary Decision Diagrams [7]. A separate family of inference algorithms for CP-logic [17,16] is based on more traditional Logic Programming methods such as SLG resolution. One of these, the PITA algorithm for CP-logic, is currently part of the XSB Prolog system (http://xsb.sourceforge.net).

Both families of algorithms compute the exact probability of a query. However, because the high computational complexity of this task can be problematic for real applications, several approximate algorithms have also been developed. One possibility [7,14] is to compute only a subset of all proofs of the query, as opposed to computing all proofs to produce an exact probability. An alternative is the Monte Carlo algorithm MCINTYRE [15], which repeatedly samples an SLD proof tree of the query. All these methods use the rules of a CP-logic theory in a backward way, i.e., going from the head to the body.

In this paper, we propose an alternative sampling algorithm, which uses the rules in a forward way, from body to head. As we will argue, this is a useful complement to the backward methods. Moreover, because different samples are sampled independently, our method can easily be parallelised. We demonstrate this by developing an implementation that runs in parallel on GPU (Graphics Processing Unit) hardware. While originally intended only for graphical computations, the processing power available in modern GPUs is becoming more and more popular as a tool for general purpose computation. This GPGPU (General Purpose computing on GPU) trend differs from traditional parallel programming by its massive multi-core approach: instead of using a relatively small number of powerful processors in parallel, a massive number of relatively weak processors are used. Sampling approaches in general depend for their accuracy on the ability to construct a large number of samples, with each individual sample being comparatively easy to compute. This fits in a natural way into the GPGPU paradigm, where a massive number of weak processors can each compute a single sample. Indeed, in the literature, we already find many examples of GPGPU sampling methods, e.g., in the context of computer graphics [22] or financial simulations [1]. However, to the best of our knowledge, this paper is the first to apply GPGPU sampling to PLP query answering.

Our implementation will make use of the OpenCl framework. This is a platform-independent framework for GPGPU programming (in contrast to its predecessor CUDA, which is specific to Nvidia hardware), which allows so-called kernels, written in a dialect of C99, to be executed on a variety of different devices. This paper will not discuss OpenCl in detail. Instead, we will introduce the main concepts as needed in Section 6, when discussing our implementation.

The results in this paper were first presented at the Probabilistic Logic Programming (PLP) 2014 workshop of the International Conference on Logic Programming (ICLP).

2. Preliminaries from logic programming

Before recalling the definition of CP-logic itself, we briefly summarise some relevant concepts from (non-probabilistic) Logic Programming. A vocabulary \( \Sigma \) consists of a set of predicate symbols and functions symbols, each with an associated arity. Function symbols with arity zero are called constants. Throughout this article, we consider only relational vocabularies in which all function symbols are constants. A term is either a constant or a variable \( x, y, z, \ldots \) drawn from some infinite set of available variables. An atom is an expression \( P(t_1, \ldots, t_n) \) where \( P \) is a predicate symbol of arity \( n \) and the \( t_i \) are terms. Formulas are built from atoms using the standard connectives \( \land, \lor, \neg, \forall, \exists \).

A Logic Program is a finite set of rules \( r \) of the form:

\[ \forall \bar{x} \ A \leftarrow \phi \]

where \( A \) is an atom (denoted head(\( r \))), \( \phi \) a formula (denoted body(\( r \))) and \( \bar{x} \) a tuple of variables that contains all free variables of \( A \) and \( \phi \). It is common to restrict \( \phi \) to a conjunction of literals. A rule is ground if it does not contain any variables. In the absence of function symbols of arity \( > 0 \), each program can be transformed to a (finite) ground program by replacing each variable by all available constants in all possible ways, turning a single rule (with a non-empty tuple \( \bar{x} \)) into a set of rules (with empty \( \bar{x} \)), turning \( \forall \) into \( \land \), and turning \( \exists \) into \( \lor \). The resulting ground program is called the grounding of the original program.

A (Herbrand) interpretation \( I \) for a vocabulary \( \Sigma \) assigns to each \( n \)-ary predicate \( P \) a set \( P^I \) of \( n \)-tuples of constants. Such an interpretation can equivalently be viewed as the set of all ground atoms \( P(\bar{c}) \) for which \( \bar{c} \in P^I \). For an atom \( A \), we write \( I \models A \) when \( A \in I \). By the usual induction over the logical connectives (e.g., \( I \models \phi \land \psi \) if both \( I \models \phi \) and \( I \models \psi \)) this notion is extended to arbitrary formulas in vocabulary \( \Sigma \).

The semantics of a positive logic program (i.e., one in which no negation \( \neg \) appears) is uncontroversially defined by means of a least fixpoint construction. For a program \( P \), we define the operator \( T_P \) which maps each interpretation \( I \) to the interpretation \( I^* = \{ \text{head}(r) \mid r \in P \text{ and } I \models \text{body}(r) \} \). The model of \( P \) is then the least fixpoint of this operator \( T_P \), which can be constructed by iteratively applying \( T_P \) to the empty interpretation \( \emptyset \).

For programs containing negation, different semantics exist. CP-logic makes use of the well-founded semantics [23], for reasons explained elsewhere [25]. This semantics uses three-valued interpretations \( I \), in which atoms and formulas can be Unknown in addition to True or False. Such a three-valued interpretation \( I \) corresponds in a natural way to a pair of two-valued interpretations \( (I^r, I^c) \) with \( I \subseteq J \): the atoms \( A \in I \) are those for which \( A^r = \text{True} \); the atoms \( A \notin J \) are those for which \( A^c = \text{False} \); and the remaining atoms \( A \in J \setminus I \) are those for which \( A^c = \text{Unknown} \). We say that a three-valued interpretation \( I \) is less precise than \( I^r \), denoted \( I \preceq I^r \), if whenever \( A^r = \text{Unknown} \) also \( A^c = \text{Unknown} \), and whenever
A^{I'} \neq \text{Unknown} \; \text{then} \; A^{I} = A^{I'}. \; \text{Viewing} \; I \; \text{and} \; I' \; \text{as pairs} \; (I, \; J) \; \text{and} \; (I', \; J') \; \text{of two-valued interpretations, this corresponds to} \; I \subseteq I' \; \text{and} \; I' \subseteq J.

Using the standard Kleene truth tables (see Fig. 1) the three-valued truth assignment to atoms can again be extended to arbitrary formulas. When the three-valued interpretation \( I \) is viewed as a pair \((I, \; J)\) of two-valued interpretations, this truth assignment \( A^I \) can also be characterised in a different way. For a formula \( \phi \), denote by \( \phi^{(I, \; J)} \) the result of evaluating all positive occurrences of atoms (i.e., those within the scope of an even number of negations) in \( I \), all negative occurrences of atoms in \( J \), and then simply applying the usual induction over the logical connectives. Then \( \phi^I = \text{True} \) if and only if both \( \phi^{(I, \; J)} = \text{True} \) and \( \phi^{(J, \; I)} = \text{True} \); \( \phi^I = \text{False} \) iff both \( \phi^{(I, \; J)} = \text{False} \) and \( \phi^{(J, \; I)} = \text{False} \); and \( \phi^I = \text{Unknown} \) iff \( \phi^{(I, \; J)} = \text{False} \) but \( \phi^{(J, \; I)} = \text{True} \). (Note that because \( I \subseteq J \), it is impossible that \( \phi^{(I, \; J)} = \text{True} \) and \( \phi^{(J, \; I)} = \text{False} \).

For a given interpretation \( J \), we can define an operator \( U_P(\cdot, \; J) \) which maps each \( I \) to the interpretation \( I' = \{ \text{head}(r) \mid r \in P \; \text{and} \; \text{body}(r)^{(I, \; J)} = \text{True} \} \). Such an operator \( U_P(\cdot, \; J) \) interprets all negative occurrences of atoms by the fixed interpretation \( J \) and acts like the standard \( T_P \) operator on the remaining, positive occurrences. As with \( T_P \), we are interested in the least fixpoint of \( U(\cdot, \; J) \). By \( C_P \), we denote the operator that maps each interpretation \( J \) to this least fixpoint, i.e., \( C_P(J) = \text{lfp}(U(\cdot, \; J)). \)

Using this \( C_P \), we can now define the well-founded model \( \text{WFM}(P) \) of \( P \) as follows. An alternating fixpoint of \( C_P \) is a pair \((I, \; J)\) of interpretations such that \( C_P(I) = J \) and \( C_P(J) = I \). The well-founded model of \( P \) is the least precise alternating fixpoint of \( C_P \).

3. Preliminaries: CP-logic

A theory in CP-logic is a finite set of CP-laws of the form:

\[
\forall x \; (A_1 : \alpha_1) \lor \cdots \lor (A_n : \alpha_n) \leftrightarrow \phi.
\]

Here, \( \phi \) is a formula of first-order logic such that the free variables of \( \phi \) belong to \( x \) and the \( A_i \) are atoms, such that the tuple of variables \( x \) contains all free variables in \( \phi \) and the \( A_i \). The \( \alpha_i \) are non-zero probabilities with \( \sum \alpha_i \leq 1 \). Such a rule expresses that \( \phi \) causes some (implicit) non-deterministic event, of which each \( A_i \) is a possible outcome with probability \( \alpha_i \). If \( \sum \alpha_i = 1 \), then at least one of the possible effects \( A_i \) must result if the event caused by \( \phi \) happens; otherwise, the event may happen without any (visible) effect on the state of the world. For a CP-law \( r \), we refer to \( \phi \) as \( \text{body}(r) \), and to the sequence \((A_i, \alpha_i)_{i=1}^n\) as \( \text{head}(r) \). The body may be omitted for events that are vacuously caused.

The semantics of a theory in CP-logic is defined in terms of its grounding, so from now on we will restrict attention to ground theories. Moreover, we assume without loss of generality that the sum \( \sum \alpha_i \) of all probabilities in any \( \text{head}(r) \) is precisely 1, since this can always be achieved by adding a fresh atom as an additional disjunct. A final assumption, made for simplicity, is that we assume that each \( \text{body}(r) \) is a conjunction of literals.

Example 1. Suzy and Billy may each decide to throw a rock at a bottle. Suzy throws with probability 0.5 and if she does, her rock breaks the bottle with probability 0.8. Billy always throws and his rock hits with probability 0.6.

\[
\begin{align*}
(\text{Throws(Suzy)} &: 0.5) \quad \text{(1)} \\
(\text{Throws(Billy)} &: 1) \quad \text{(2)} \\
(\text{Broken} &: 0.8) &\leftrightarrow \text{Throws(Suzy)} \quad \text{(3)} \\
(\text{Broken} &: 0.6) &\leftrightarrow \text{Throws(Billy)} \quad \text{(4)}
\end{align*}
\]

This theory consists of two vacuous laws and two conditional laws. The second law is deterministic, i.e., it always has the same effect.

The semantics of CP-logic can be defined as a straightforward instantiation of Sato’s distribution semantics. An instance of a CP-logic theory \( T \) is a logic program that is constructed by choosing a single atom \( A_i \) from the head of each rule \( r \) of \( T \). In other words, if we denote this chosen atom by \( \sigma(r) \), then the instance \( T^\sigma \) is the logic program that consists of the rules:

\[
\sigma(r) \leftarrow \text{body}(r), \quad \text{for all} \; r \in T.
\]
To each such instance, we associate a probability $\pi_\sigma$, which is the product $\prod_{r \in T} \alpha_r$, where each $\alpha_r$ is the probability of $\sigma(r)$ in head$(r)$. This probability distribution over instances is then mapped to a probability distribution over interpretations by applying the well-founded model semantics to these logic programs. In other words, we define a probability distribution $\pi_T$ over interpretations $I$ as:

$$\pi_T(I) = \sum_{\sigma \text{ s.t. } \text{WFM}(T) = I} \pi_\sigma.$$  \hspace{1cm} (5)

This semantics is only well-defined for CP-theories which have the property that each instance $T^\sigma$ has a two-valued well-founded model. Therefore, CP-logic only allows such theories. This can be ensured by imposing a syntactic condition such as stratification.

In [25], an alternative— but equivalent— characterisation of the CP-logic semantics was developed. Following the work of Shafer [20], it makes use of probability trees. In particular, the concept of an execution model of a CP-theory is defined. This is a probability tree in which each node $s$ is labelled with an interpretation $I(s)$. Such trees are constructed, starting from a root node $s_0$ in which all atoms are false (i.e., $I(s_0) = \{\}$), by “firing” rules whose body holds. Figure 2 shows an execution model for Example 1. States $s$ in which the bottle is broken (i.e., $I(s) \models \text{Broken}$) are represented by an empty circle, and those in which it is still whole by a full one. As usual, the probability of a branch of such a tree is defined as the product of all probabilistic labels of edges that are contained in the branch.

To each branch $b$ of such a tree corresponds a set of instances of the theory, namely all those which make the same choices as $b$ for all the rules that actually fire in $b$. For the rules that do not fire in $b$, it does not matter which choices the instances make. As shown in [25], the sum of the probabilities of all these instances is precisely the probability of the branch, and they all have the interpretation $I(l)$ associated with the leaf $l$ of the branch as their WFM. For positive theories, i.e., those without negation, this property is rather obvious. In the general case, however, some additional care is required when constructing the execution models. We discuss this in Section 5.

A CP-theory may have many execution models, which differ in the order in which they fire rules. The differences between these trees are irrelevant, in the sense that they all produce the same probability distribution in the end, namely $\pi_T$ as defined by equation (5) [25].

4. A forward sampling method for positive CP-theories

The probability tree characterisation of the semantics of CP-logic suggests a straightforward sampling method. To approximate the probability $\pi_T(Q)$ of a query $Q$, we can simply perform repeated random walks in the tree and report the ratio $\hat{p}$ of how often we end up in a leaf $s$ such that $I(s) \models Q$ versus the total number of walks. The more walks we take, the more precise this estimate.

To turn this idea into a concrete algorithm, we must decide which of the execution models of the theory will be used for the random walks. For efficiency reasons, we construct the tree by first building a list of all rules that are applicable in a given node, and then firing all of these rules in sequence, thereby traversing a number of nodes in which we do not have to evaluate any rule bodies. This results in Algorithm 1, which produces a single random walk when called with the CP-theory $T$ as its first argument and the interpretation $\{\}$ that assigns $\text{False}$ to all atoms as the second.

At the end of this procedure, the interpretation $I$ is saturated, in the sense that all rules $r \in T$, such that $I \models \text{body}(r)$, have been fired, and therefore $I$ contains at least one true atom from each such head$(r)$.

The following theorem proves the correctness of this procedure.

Theorem 1. Let $T$ be a positive CP-theory and let $F$ be the interpretation that assigns false to all atoms. Let $\hat{P}_T$ be the probability distribution over all possible runs $\rho$ of the probabilistic Algorithm 1 when called as Saturate$(T, F)$. For each possible run $\rho$, let $n^\rho$ be the total (i.e., summing over all iterations of the outer loop) number of iterations of the inner for-loop (line 6), and, for each $1 \leq i \leq n^\rho$, let $l_i^\rho$ be the value of $I$ at the start of the $i$th iteration of this loop. Then there exists an execution model $\chi$ of $T$ with a one-to-one mapping $f$ between the branches of $\chi$ and the possible runs of the algorithm, such that, for each branch $b = (s_1, \ldots, s_n)$ of $\chi$ and the corresponding run $\rho = f(b)$, the probability of $b$ is equal to $P_T(\rho)$ and $I(s_i) = l_i^\rho$ for all $1 \leq i \leq n^\rho$. 

Fig. 2. An execution model for Example 1.
Algorithm 1 Saturates a given interpretation $I$ by applying rules from a set $R$

1: procedure SATURATE($T$, $I$) $\triangleright$ Updates $I$
2: $T' \leftarrow T$
3: $\text{Fired} \leftarrow \{\}$
4: repeat
5: $\text{ToFire} \leftarrow \{ r \in T' \mid I \models \text{body}(r) \}$
6: for all $r \in \text{ToFire}$ do
7: $a \leftarrow \text{RandomChoice}($head$(r))$
8: $I(a) \leftarrow \text{True}$
9: $T' \leftarrow T' \setminus \text{ToFire}$
10: until $\text{ToFire} = \{\}$
11: function $\text{RandomChoice}((\alpha_1, \alpha_1), \ldots, (\alpha_n, \alpha_n))$
12: $r \leftarrow \text{random floating point number} \in [0, 1]$
13: sum $\leftarrow 0$
14: for all $i \in [1, n]$ do
15: sum $\leftarrow \text{sum} + \alpha_i$
16: if $r \leq \text{sum}$ then
17: return $\alpha_i$

Proof. For a run $\rho$, we construct part of an execution model containing a branch $b$ such that $f(b) = \rho$ as follows. We start from a root node $S_0$ and assign $I(s) = F$ (as must be the case for a root node in an execution model). We iteratively extend the node $s_i$ by firing the rule $r$ of line 6: if the head of $r$ is $((\alpha_1, \alpha_1), \ldots, (\alpha_n, \alpha_n))$, then the children of $s_i$ are nodes $s_{i,j}$ for $j \in 1..n$, with $I(s_{i,j}) = I(s_i) \cup \{\alpha_j\}$ and the probability of the edge from $s_i$ to $s_{i,j}$ is $\alpha_j$. The node $s_{i,j}$ such that $\text{RandomChoice}($head$(r))$ returned $a_j$ is then chosen as $s_{i+1,j}$, and this construction is repeated. It is obvious by induction that, for each $i$, $I(s_i) = I^i$. Moreover, the probability tree thus constructed can be extended to an execution model by applying the usual construction process to those nodes that do not occur in the sequence $(s_i)_i$.

We now have established a one-to-one mapping $f$ between runs and branches of a number of different (potentially as many as there are runs) execution models of $T$. What remains to be shown is that there exists a single execution model $\chi$ such that the branches of $\chi$ are precisely all $f^{-1}(\rho)$ for which $\rho$ is a possible run of the algorithm.

Whenever two runs $\rho, \rho'$ of the algorithm coincide on the first $n$ probabilistic choices returned by $\text{RandomChoice}$, then the initial fragments $(s_0, \ldots, s_n)$ and $(s'_0, \ldots, s'_n)$ of the sequences $(s_i)_i$ and $(s'_i)_i$ constructed in the preceding paragraph will also be identical. It follows that we can merge the two corresponding partial probability trees into a single probability tree:

In this picture, the bullets denote nodes that are constructed but not expanded, because they do not belong to the sequence $(s_i)_i$ ($(s'_i)_i$ respectively). Moreover, $n$ is the maximal number for which the initial fragments up to $n$ of $(s_i)_i$ and $(s'_i)_i$ coincide. It follows that we can merge all branches $f^{-1}(\rho)$ for which $\rho$ is a possible run into a single probability tree $\chi$. By construction, this single probability tree is a complete execution model of the theory. The fact that the probabilities with which $\text{RandomChoice}$ returns a particular atom are the same as the labels $\alpha_i$ placed on the corresponding edge of this execution model concludes the proof. \hfill $\square$

With this result in hand, we can compute the probability of a given query by repeatedly sampling this distribution. This results in the following Algorithm 2.

So far, we have only considered positive CP-logic theories. When negation is allowed to appear, some more work is needed.

5. A forward sampling method for general CP-theories

Negation in a CP-theory is interpreted by the WFM semantics. In [25], the execution model semantics was also extended to this case. Because the WFM uses three-valued interpretations, each node $s$ in an execution model is now also labelled
Algorithm 2 Approximates the probability of a query $Q$ according to $\pi_T$

1: function SAMPLE($T, Q$)  
2: $q \leftarrow 0, n \leftarrow 0$  
3: repeat  
4: $I \leftarrow$ the interpretation such that all atoms are False  
5: Saturate($T, I$)  
6: if $I \models Q$ then $q \leftarrow q + 1$  
7: $n \leftarrow n + 1$  
8: until desired accuracy reached
9: return $q/n$  
\[\text{▷ Section 6.2}\]

Algorithm 3 Samples single branch of execution model of $T$, returning the leaf

1: function SINGLE SAMPLE($T$)  
2: $I \leftarrow$ interpretation that maps each atom appearing in $T$ to Unknown  
3: repeat  
4: Saturate($T, I$)  
5: Shrink($T, I$)  
6: until fixpoint reached
7: return $I$  
8: function Shrink($T, I$)  
9: \[\text{▷ updates $I$ by falsifying atoms}\]
10: for all atoms $a$ such that $I(a) = \text{Unknown}$ do  
11: $I(a) \leftarrow \text{False}$  
12: \[\text{▷ that can no longer be derived}\]
13: ToFire $\leftarrow \{ r \in T \mid \text{body}(r)^T \neq \text{False} \}$  
14: for all atoms $a$ in some head(r) with $r \in \text{ToFire}$ do  
15: $I(a) \leftarrow \text{Unknown}$  
16: $T \leftarrow T \setminus \text{ToFire}$  
17: until ToFire $= \{}$

with a three-valued interpretation $I(s)$, in addition to the two-valued $I(s)$. The true atoms of $I(s)$ and $I(s)$ are always the same, but false atoms of $I(s)$ may be unknown in $I(s)$. At the root $r$ of the tree, $I(r)$ assigns Unknown to all atoms.

The key difference with the positive case is that rules are now only allowed to fire if their body is true in both $I(s)$ and in $I(s)$ (in particular, rules which are true in $I(s)$ but unknown in $I(s)$ may not fire). The effect of firing a rule is the same as before: one atom from the head of the rule becomes true, both in $I(s)$ and $I(s)$. In this way, an atom that was originally unknown in some $I(s)$ may become true in a child $I(s')$. There is also a way in which atoms that are originally unknown may become false. This is done by a "look ahead"-mechanism, that makes atoms false in any case there is no longer any way in which they could still be caused. The details are as follows.

The three-valued interpretation $I(s)$ in a node $s$ is not derived from the three-valued interpretation $I(s')$ associated to the parent $s'$ of $s$, but it is instead constructed from the two-valued interpretation $I(s)$. In particular, $I(s)$ is defined as the limit of a sequence $(I'_s)_{i=0}$ of three-valued interpretations. This sequence starts from the interpretation $I'_0$ that coincides completely with $I(s)$ (and therefore has no unknown atoms). Given an $I'_i$, we then select a rule $r$ that has not yet fired and whose body is either True or Unknown in $I'_i$. We derive $I'_{i+1}$ from $I'_i$ by changing the truth value of all atoms in the head of $r$ that are still False to Unknown. In this way, we end up making some of the atoms that are False in $I(s)$, but not necessarily all of them. This is then defined as the sequence. It can be shown that, by constructing $I(s)$ in this way, an overestimate is obtained of the atoms that still might be caused in a descendant of $s$: atoms that are False in $I(s)$ are also false in $I(l)$ for any leaf $l$ that can be reached from $s$. Relating this construction to the concepts of Section 2, it was shown in [25] that in the special case of deterministic CP-theories, this $I(s)$ is in fact equal to $C_T(I(s))$. From this result, it follows that, also for theories containing negation, the execution model characterisation of the semantics coincides with the Sato-style distribution semantics characterisation of equation (5).

Even though the three-valued interpretation $I(s)$ is constructed from $I(s)$, and not from the three-valued interpretation $I(s')$ of the parent $s'$ of $s$, it can nevertheless be shown that $I(s)$ is always more precise than $I(s')$, in the sense that its true and false atoms, respectively, are a superset of those of $I(s')$. This allows us to postpone the expensive operation of actually computing $I(s)$: if the body of a rule $r$ is true in $I(s')$, for some ancestor $s''$ of $s$, then we can be sure that it is still true in $I(s)$. Therefore, we can safely fire $r$, thus proceeding to a child of $s$, without ever having to compute $I(s)$ at all.

A further improvement on this is that it is also safe to fire a rule $r$ in a child $s$ of $s'$ even if its body is not true in $I(s')$ itself, but is true in the three-valued interpretation $I'$ that we can construct from $I(s')$ by switching the atom $a$ that was chosen on the transition from $s'$ to $s$ from Unknown to True (in other words, $I'$ has the same False atoms as $I(s')$ and the same True atoms as $I(s)$). This is because the real $I(s)$, that we are trying to avoid having to construct, must also be more precise that this $I'$, and therefore all rule bodies that are true in $I'$ must also be true in $I(s)$.

This leads us to Algorithm 3. Here, we are now calling the procedure Saturate with a three-valued interpretation as its argument. In this case, the expression $I \models \text{body}(r)$ in Line 13 of Algorithm 1 has to be interpreted as that the truth value of $\text{body}(r)$ has to be True according to $I$ (i.e., Unknown is not enough).
Theorem 2. For a CP-logic theory $T$ that may contain negation, the same correspondence holds between runs of the non-deterministic Algorithm 3 and branches of an execution model of $T$, as formulated in Theorem 1 for Algorithm 1 and execution models of a positive CP-logic theory.

Proof. The procedure SHRINK of Algorithm 3 is a straightforward implementation of the construction process used to define the three-valued interpretation $I(s)$. From this, together with Theorem 1, the result immediately follows. □

6. OpenCL implementation

To compute a good approximation of the probability of a query, our sampling algorithm may need a large number of samples. Because different samples are sampled independently, the sampling process can be executed in parallel to achieve a significant speed-up. We demonstrate this in OpenCL, a recent framework for programming parallel hardware (typically, but not exclusively, GPUs) in a platform-independent way.

6.1. A short introduction to OpenCL

An OpenCL program consists of two parts: there are kernels, which are executed in parallel on OpenCL-capable devices such as GPUs, and there is host code which runs on the CPU and is responsible for starting the kernels and loading the necessary data onto the device where the kernels will run. The kernels are written in the OpenCL language, which is a variant of C99. Host code can be written in a variety of standard programming languages, such as C or C++.

When executing a kernel on an OpenCL device, the host code specifies how many threads, also called work items, have to be run. These work items are grouped into workgroups. All work items in the same work group are executed in SIMT (Single Instruction Multiple Thread) parallel, meaning that they share a single program counter and are therefore always executing the same instruction.

Typically, memory management plays a key role in producing efficient OpenCL code. On an OpenCL device, there are three different kinds of memory. Global memory is shared across all work items and is the slowest memory. It is also the only memory that can be accessed from the host, so all data transfer to and from the OpenCL device has to pass through it. Local memory belongs to a single workgroup and is typically both smaller and faster than global memory. The smallest and fastest kind is the private memory of a single work item.

6.2. Our algorithm in OpenCL

To compute $n$ samples, we create $n$ work items, each running a kernel consisting of Algorithm 3. To make this possible, the host code first copies the CP-theory $T$ to the global memory of the OpenCL device. It then also allocates the data structures that the work items need for their operation: an array atoms in which they can store the truth value of each atom, initialised to Unknown; an array rules in which they can store the truth value of each rule body (True, False, Unknown or “already fired”), also initialised to unknown; and a sequence of random numbers for the required random decisions. In the non-optimised version of our implementation, all these arrays are simply stored in global memory.

When a work item finishes, it also leaves its end result in global memory. Once all work items have finished, the host copies these individual samples back to main memory and joins them together in order to answer the query. To improve the performance of this basic OpenCL algorithm, we can apply the following optimisations.

MEM: Each work item $w_1 \ldots w_n$ needs its own array $(a_1^{w_1}, \ldots, a_m^{w_1})$ in which to store the truth value of the atoms $(a_1, \ldots, a_m)$. Instead of storing these arrays as $((a_1^{w_1}, \ldots, a_m^{w_1}), (a_1^{w_2}, \ldots, a_m^{w_2}), \ldots, (a_1^{w_n}, \ldots, a_m^{w_n}))$, it is more efficient to store them as $((a_1^{w_1}, \ldots, a_1^{w_n}), (a_2^{w_1}, \ldots, a_2^{w_n}), \ldots, (a_m^{w_1}, \ldots, a_m^{w_n}))$. In this way, coalesced memory access allows all work items to access their own truth value for the same atom $a_i$ in a single transaction.

INI: Initialising the arrays atoms and rules can be done as the first operation of the kernel, to avoid copying two large arrays of Unknown values from host to device memory.

PRI: The array atoms can be cached in private memory to reduce the time spent accessing global memory. For several of our larger benchmarks, the array rules does not fit in private or even local memory, so we leave this in global memory.

RAN: Instead of generating random numbers on the host and transferring them to the OpenCL device, we can also run a random number generator on the OpenCL device, so that we only have to transfer a single seed. We will use the MWC64X OpenCL random number generator.²

RED: Instead of copying the results of the individual samples back to the host, we can run a second kernel Reduce on the OpenCL device to aggregate the results there, so only the final count has to be transferred back. This is

¹ http://www.khronos.org/opencl/
² http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html
done using the standard parallel reduction method of iteratively reducing $n$ results to $n/2$ results by having $n/2$ workitems that each combine 2 results. The necessary intermediate data structures are stored in local memory.

**CHU:** While the OpenCL device is constructing samples, the host is idling. To avoid this, we can opt for a chunking strategy, in which we compute our $n$ samples in chunks of $m \leq n$ at a time. While the OpenCL device is computing a chunk of samples, the host can transfer the results of the previous chunk back to host memory and aggregate them. Because this is a relatively cheap operation, it should be finished before the $i$th chunk of samples is done, so the host should be ready in time to initiate the next chunk. In this way, we no longer need a separate *Reduce* kernel and keep the OpenCL device continually busy generating samples. A possible downside of this approach is that we now incur multiple times the overhead of starting a kernel on the OpenCL device and waiting for it to finish.

All of the above optimisations can be combined, with the exception of the mutually incompatible RED and CHU. In addition, we also experimented with caching part of the theory (in particular, the heads of rules) in local memory. However, due to limited size of local memory this was only possible for very small benchmarks. Moreover, it did not significantly improve performance. We have therefore not considered this further.

Instead of running a fixed number of iterations, we can also stop the algorithm once a desired accuracy has been reached. This is particularly useful in combination with the optimisation CHU, because we can then check the accuracy after each chuck. The Problog Monte Carlo algorithm [7] uses a stopping criterion based on the central limit theorem:

$$2z_{1-\alpha/2} \sqrt{\hat{p}(1-\hat{p})/k} \leq \delta,$$

where $\hat{p}$ is the measured probability of the query, $z_x$ is the $x$ percentile of a standard normal distribution, $k$ is the number of samples and $\delta$ is the desired width of the confidence interval. The MCINTYRE algorithm uses the same criterion, with the additional requirement that $k\hat{p} > 5$ and $k(1-\hat{p}) > 5$ to ensure that approximation provided by the central limit theorem is trustworthy. We will use the same criterion with the same parameters of $\alpha = 0.05$ (so $z_{1-\alpha/2} = 1.96$) and $\delta = 0.01$. With these parameters, the worst case value for $k$ (reached for $\hat{p} = 0.5$) is 38416.

### 7. Experimental results

Source code and graphs of the experiments described in this section can be downloaded from:


All our experiments were run on a Linux machine with an Intel Core i7 965 CPU (3.20 GHz) and an NVIDIA GeForce GTX 295 (GT200) GPU. This GPU has 896 MiB of global memory and consists of 30 streaming multiprocessors, each with 16 KiB of local memory and consisting of 8 individual cores operating at 1242 MHz. When executing OpenCL code, each work item occupies a single core, with work items in the same work group running on the same streaming multiprocessor. Each core has an additional 8 KiB of private memory available. Our experiments use the BLOODTYPE, GROWINCHEAD and GROWINCBODY benchmarks from [10]. The latter two benchmarks consist of ground programs. However, the BLOODTYPE benchmark first needs to be grounded before we can apply our algorithm.

**Fig. 3** investigates the effect of the different optimisations proposed in the previous section. As a baseline, we also include a plain C version of our algorithm, that runs on a single CPU core and computes the required number of samples in a sequential *for*-loop. We compare this baseline to the following variants of our OpenCL implementation on GPU.

<table>
<thead>
<tr>
<th>Name</th>
<th>Applied optimisations</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU original</td>
<td>–</td>
</tr>
<tr>
<td>RAN</td>
<td>RAN</td>
</tr>
<tr>
<td>RAN + MEM</td>
<td>RAN, MEM</td>
</tr>
<tr>
<td>Reduction</td>
<td>RAN, MEM, RED</td>
</tr>
<tr>
<td>Private</td>
<td>RAN, MEM, RED, PRI</td>
</tr>
<tr>
<td>Chunking</td>
<td>RAN, MEM, CHU, PRI</td>
</tr>
</tbody>
</table>

Recall that the optimisations CHU and RED are incompatible. Therefore, the final two variants are two alternative ways of fully optimising the code. For each version, we have run a single instance of the three benchmarks.

**Fig. 3** shows first of all that relatively little is gained by simply porting the CPU code to GPU in a naive way. Indeed, for one of the benchmarks, the additional overhead of copying data and starting an OpenCL kernel even leads to a slow-down instead of a speed-up. This figure also shows that the optimisations have a significant impact. In general, more optimisations are better, even though it is not always the same optimisation that leads to the biggest gain. Of the two fully optimised variants, Chunking outperforms Private on two of the three benchmarks. When comparing it to GPU original, the smallest
gain is observed in the GrowingBody benchmark, where Chunking is only about 4× faster. For Bloodtype and GrowingHead, the speedup is 15× and 37×, respectively. When compared to the C version on a single CPU core, the speedups are, respectively, 19×, 22× and 10×. We conclude that use of the GPU and our optimisations is indeed worthwhile. Our remaining experiments will be conducted using the Chunking variant.

The convergence behaviour of our algorithm is examined in Figs. 4a, 4b, 4c. These figures show boxplots for the probability estimates produced by 10 different runs of our algorithm, each time using a fixed number of samples. The benchmarks used here are the same as before. As these results show, even a relatively small number of samples can already produce reasonably good approximations of the true probability. This is most pronounced for Bloodtype, where 10 samples already provide a very good estimate (even though the stopping criterion of Section 6.2 would run until about 350 samples). For GrowingBody, the stopping criterion suggests about 17 000 samples, which is indeed about where the results of the algorithm begin to seem accurate. The same can be said for GrowingHead, where the stopping criterion suggests about 30 000 samples.

Finally, we compare our algorithm to existing query answering methods for CP-logic:

- The approach of [9], which transforms CP-logic into a Bayesian network and then applies the Variable Elimination (VE) algorithm or Contextual Variable Elimination (CVE) [10].
- The default and Monte Carlo inference algorithms of the ProbLog system [7].
- The PITA algorithm [18] that is built into XSB Prolog.
- The MCINTYRE backwards sampling algorithms [15].

For the k-best [7] and k-optimal [14] algorithms, we were unable to run the most recent version (in YAP 6.3.3) on our system, even after contacting the authors. We were able to run an older version of k-best (in YAP 6.2.2), but this does
not support negation of arbitrary atoms and was therefore unable to handle the GrowingBody benchmark. The k-optimal algorithm is not available in this version and could not be included in our experiments.

The results for the three benchmarks are shown in Figs. 5, 6 and 7. Here, each data point is the lowest runtime observed in three runs of the systems. Three variants of our OpenCL algorithm are included in the comparison: \textit{inc} = 20000, \textit{inc} = 5000 and \textit{k} = 38416. The first two variants use the MCINTYRE stopping criterion, where \textit{inc} is the chunk size (i.e., the number of samples that are run at once). A smaller \textit{inc} allows the total number of samples computed to be closer to the actual minimal value for which the stopping criterion is satisfied, but may increase the likelihood of GPU processors having to wait for data transfers. Our experiments show that the large chunk size consistently performs better. Our third variant does not actually use the stopping criterion at all, but just uses its worst case value of \textit{k} = 38416 as a fixed number of samples. For k-best, we have included experiments with \textit{k} = 2000, \textit{k} = 200 and \textit{k} = 20. As the results below show, the precise value of \textit{k} has only a limited impact on the way in which this algorithm compares to the others.

For Bloodtype, the sampling approaches perform quite poorly in general. While several exact inference methods (PITA, CVE) are able to handle problems up to size 16 in a second or less, MCINTYRE (not visible in the graph, because all measured runtimes exceed 10 s), Problog Monte Carlo and the three version of k-best already require more than 100 s for size 4. Our algorithm does better and can handle up to size 6 in about a second. After that, however, the theory no longer fits in the global memory of our GPU. We have also run some experiments with a Tesla C 2075 GPU, which has 6 GiB of global memory. This allows instances up to size 11 to fit in GPU memory. The runtimes for these problem sizes were still in the order of a few (< 5) seconds, for all three variants of our algorithm, following the CVE curve. As this benchmark is the only non-ground one, the times reported for our algorithm include the time needed to ground the theory (~ 0.01 s–0.1 s).

A similar phenomenon can be seen for the GrowingBody benchmark. Again, CVE and PITA perform best, comfortably handling problem sizes \( \geq 20 \), while the sampling methods of MCINTYRE and Problog MC already struggle to reach size 10. Here, however, our sampling approach does significantly better. While its runtimes do grow faster than that of CVE and PITA, it is still able to handle sizes \( \geq 20 \) in under a second.

Finally, the GrowingHead benchmark is clearly better suited for the sampling methods. Here, Problog MC is the worst of the sampling methods, being (slightly) outperformed by the exact inference of PITA, but with the gap decreasing for the larger problem sizes. For \textit{k} = 20, the k-best algorithm runs slightly faster than PITA, but produces an estimate that is almost 0.2 below the actual probability. MCINTYRE performs best in this benchmark, handling problem sizes \( \geq 20 \) in a fraction of a second. Our algorithm is somewhat slower, probably due to the overhead of copying data to the GPU, but, like MCINTYRE, it also shows an almost constant runtime (around 0.1 s) for this benchmark.

8. Discussion and future work

Sampling methods have a long history as ways of approximating complex computations. Typically, the development of a new sampling method requires a significant amount of domain knowledge and creativity in order to be effective. PLL
may help to improve on this situation, by allowing the development of generic sampling methods, that can be applied to a wide range of probabilistic models. For instance, [5,6] developed a general framework for constructing sampling algorithms, which they apply both to estimate the number of models of a DNF formula and to estimate the failure probability of a network with possibly faulty edges. In CP-logic, representing such a network is almost trivial (see e.g. [3]); moreover, the Prolog system is able to reduce the problem of computing its failure probability to that of computing the probability of a DNF—and to solve this latter problem, precisely the sampling method of [5] may be used [21]. In this way, the Prolog PLL system has in effect incorporated Karp et al.’s methodology.

The fact that the efficacy of sampling methods typically depends on having a large number of independently drawn samples of course makes these methods in general a suitable target for GPU hardware. This has been investigated for instance in the context of physics [11], biochemical networks [26] or finances [1]. A general study of a number of different sampling methods on GPU from a statistical point-of-view was performed in [8]. Their results demonstrate that remarkable
speed-ups (from $30 \times$ to $500 \times$) can indeed be achieved. Here again, we may hope that eventually PLL methods will be able to offer a uniform framework which is able to reproduce the results from these special-purpose methods.

While the suitability of GPU hardware for sampling algorithms has thus already been demonstrated, we are not aware of any attempts to exploit this possibility in the context of PLL. Our efforts, using the OpenCL framework, are therefore, to the best of our knowledge, the first in this respect. When we compare our work to the existing studies of sampling on GPU, we find that focus of these methods is somewhat orthogonal to the results we have presented in this paper. This is due to the fact that logic programming concepts play a significant role in our method, whereas existing approaches are more statistical in nature. As a consequence, the bulk of our algorithm is concerned with the computation of a single sample, which is typically considered to be an atomic operation in more statistically oriented work. On the other hand, our method directly samples from the target distribution in a very straightforward way, whereas statistical methods often sample from a different distribution in order to reduce the number of samples that is needed. In future work, it could be examined whether these methods can also be integrated with our work. In particular, this could prove very useful when the goal is not to compute an \textit{a priori} probability $P_T(\phi)$ but an \textit{a posteriori} $P_T(\phi \mid \psi)$ where $\psi$ itself has a low probability. However, this is beyond the scope of the current paper.

When comparing our sampling method on GPU with other inference methods for the PLL language of CP-logic, we have first of all shown that a careful optimisation of the code is important in order to achieve decent performance. However, with these optimisations in place, our implementation was generally able to outperform the existing PLL sampling methods. A single exception was the GROWINGHEAD benchmark, where our algorithm shows similar performance to MCINTYRE, but was a constant few hundredths of a second slower, probably due to the overhead of using the GPU.

When comparing the sampling methods in general to the exact inference algorithms, we saw that the PITA algorithm and theCVE algorithm both had clearly better performance on one of the benchmarks. However, for the other benchmarks, they performed significantly worse.

We can conclude from these results that:

- Certain benchmarks are more suited towards exact methods while others favour sampling methods, which demonstrates again that both kinds of algorithms have their place;
- Our method is an improvement on the state-of-the-art of current sampling algorithms for CP-logic, delivering a more robust performance across different benchmarks.

A particular challenge in our experiments is that we have had to compare the performance of our method, running mainly on GPU, with existing algorithms that only use the CPU. Hence, a bias could have been introduced into our results by using a comparatively more or less powerful GPU. To avoid this, we have chosen to use similarly priced processors: at the time of writing, both the CPU and GPU that were used are worth around $100–$200. Moreover, they are commonly found together in typical desktop machines.

One limitation of such low-end GPUs is that they typically have a relatively small amount of memory available, causing our approach to become useless when the CP-logic theory grows too large. However, this problem does not appear to be too severe in practice: we only encountered it for one of our three benchmarks and, even there, we were able to reach a problem size where several other approaches (standard Prolog inference, the k-best method and MCINTYRE) had already either timed-out or run out of (main) memory as well. The problem can of course be mitigated further by switching to a higher end GPU: by using one in the price range of $\pm$1500 instead of $\pm$150, we could reach problem size $n = 11$ instead of $n = 7$ for this benchmark.

Our method not only differs from existing sampling methods by using GPU hardware, but also by not being query-oriented: each sample constructs the entire interpretation at one of the leaves of an execution model. In our experiments, we have used this interpretation only to check that a single query is satisfied. However, with essentially the same effort, we can compute the probability of a set of queries at once. By contrast, all of the other algorithms that we have considered would need to be run for each query separately. In this sense, the experiments that we have conducted therefore represent the worst case for our algorithm, namely, that in which only a single query is of interest. The ability to compute the probability of multiple queries at once is useful in, e.g., applications where the goal is to find the most probable of a set of atoms, or to compute their entire joint probability distribution.

The fact that our method is not query-based also opens up interesting possibilities for future work. For instance, our method could be applied in cases where query-based methods have problems with an infinite number of explanations. Suppose a player repeatedly rolls a die until he gets the number six. Computing the expected number of rolls is not a trivial task for algorithms that depend on a completely constructed proof tree, as they may end up in an infinite recursion. Our approach does not suffer from this problem, even though our current ground-and-solve method obviously cannot cope with the fact that this example would generate an infinite grounding. However, there appears to be no conceptual difficulty in switching to a lazy grounding method, where we would load the non-ground theory directly onto the GPU and then generate instantiations of the rules only when they can actually be fired.

A second interesting topic for future work is that our algorithm may also be easier to extend to new language features, because it more closely follows the execution model semantics of CP-logic. This may in particular be the case for language features that extend the expressive power of the heads of CP-logic rules. For instance, while the head of a CP-logic rule is currently a choice between a fixed number of alternatives, we could allow the set of possible alternatives and/or their prob-
abilities to be dynamically determined, based on the interpretation I(s) in the state s where it is executed. A similar feature can be found in the P-log language [2], which has some similarities to CP-logic, but does not follow Sato’s distribution semantics. (A comparison between CP-logic and P-log can be found in [25].) A second example is that, instead of allowing only a discrete choice in the head of rules, we could also allow a value to be selected from a continuous distribution. Finally, recent work [24] has investigated the possibility of extending CP-logic with negated atoms in the head. Integrating such features into our forward sampling algorithm seems much easier than extending one of the proof-based methods.

9. Conclusions

We have presented a sampling method for the expressive PLP language of CP-logic. It differs from the existing state-of-the-art in two ways: it makes use of the OpenCL framework to take advantage of massively parallel GPU hardware; and it is a forward method, i.e., it is not query-based. As is common for OpenCL programs, the efficiency of our implementation depends to a large extent on which optimisations have been applied: in one case, our most optimised variant was 37 times faster than the original.

With the necessary optimisations in place, our algorithm proved to be an improvement on existing sampling methods (MCINTYRE and Problog Monte Carlo), providing a more robust performance across different benchmarks. In addition to its greater efficiency, our algorithm can also compute the probabilities of a set of queries almost as quickly as that of a single query. Our results also show that neither the exact inference algorithms nor the sampling methods are able to consistently outperform the other, suggesting that both have their place among PLP inference methods.

One limitation of our current work is that we have focused on ground CP-theories, which means that grounding is a potential bottleneck. In future work, a lifted method could be developed, which would allow rules to be stored on the GPU in a first-order format. During the sampling process, these could then be instantiated on the fly. This could also help to avoid the problems we observed with memory limitations on the GPU. Moreover, it would also allow us to handle problems with an infinite grounding.

References