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

Wiebe Van Ranst and Joost Vennekens

firstname.lastname@kuleuven.be
Dept. of Computer Science, KU Leuven
Campus De Nayer
J.-P. De Nayerlaan 5, 2860 Sint-Katelijne-Waver

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

The Distribution Semantics, originally developed by Sato [13], has given rise to a family of Probabilistic Logic Programming (PLP) languages. Examples of such languages are Sato's own PRISM language [13], the Independent Choice Logic [7] and CP-logic [17]. The most common inference task for these languages is that of computing the probability of a query. For CP-logic,which subsumes the other mentioned languages, this task has been implemented in a number of different ways. The Problog system (`http://dtai.cs.kuleuven.be/problog`) supports the CP-logic language with algorithms based on weighted CNFs [3] and Binary Decision Diagrams [4]. A separate family of inference algorithms for CP-logic [11, 10] 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 algoritms 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 [4, 8] is to compute only a subset of all proofs of the query, as opposed to computing all proofs which produces an exact probability. An alternative is the Monte Carlo algorithm MCINTYRE [9], which repeatedly samples an SLD proof tree of the query. All these methods use the rules of a CP-logic theory in a backwards 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 forwards way, from body to head. As we will argue, this is a useful complement to the backwards methods. Moreover, because different samples are

independent, our method can easily be parallellised. 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 [14] 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 5, when discussing our implementation.

2 Preliminaries: CP-logic

A theory in CP-logic consists of a set of *CP-laws* of the form: $\forall \boldsymbol{x}\ (A_1 : \alpha_1) \vee \cdots \vee (A_n : \alpha_n) \leftarrow \phi$. Here, ϕ is a conjunction of literals and the A_i are atoms, such that the tuple of variables $\boldsymbol{x}$ contains all free variables in ϕ and the A_i. The α_i are non-zero probabilities with $\sum \alpha_i \leq 1$. Such a rule expresses that ϕ causes some (implicit) non-deterministic event, of which each A_i is a possible outcome with probability α_i. If $\sum_i \alpha_i = 1$, then at least one of the possible effects A_i must result if the event caused by ϕ happens; otherwise, the event may happen without any (visible) effect on the state of the world. For a CP-law r, we refer to ϕ as $body(r)$, and to the sequence $(A_i, \alpha_i)_{i=1}^n$ as $head(r)$. 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, in which each tuple of variables $\boldsymbol{x}$ is empty. Moreover, we assume that the sum $\sum_i \alpha_i$ of all probabilities in any $head(r)$ is always 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 $body(r)$ is a conjunction of literals.

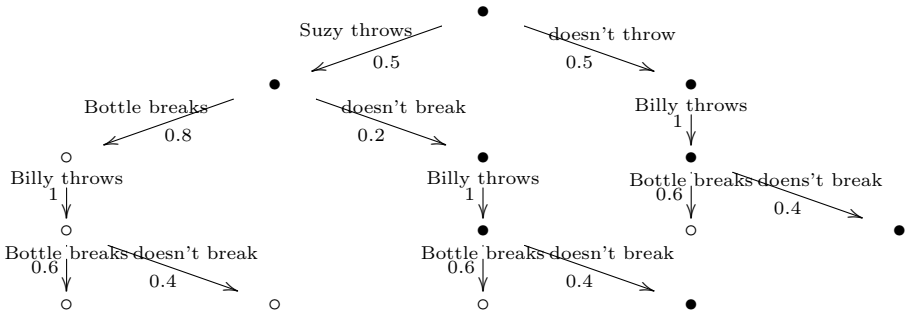
Example 1. Suzy and Billy may each decide to throw a rock at a bottle. Suzy throws with probability 0.5 and if she does, her rock breaks the bottle with

probability 0.8. Billy always throws and his rock hits with probability 0.6.

$$(Throws(Suzy) : 0.5). \qquad (Broken : 0.8) \leftarrow Throws(Suzy).$$
$$(Throws(Billy) : 1). \qquad (Broken : 0.6) \leftarrow Throws(Billy).$$

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 chosen atom by $\sigma(r)$, then the instance T^σ is the logic program that consists of the rules $\sigma(r) \leftarrow body(r)$ for all $r \in T$. To each such instance, we associate a probability π_σ, which is the product $\Pi_{r \in T} \alpha_r$, where each α_r is the probability of $\sigma(r)$ in $head(r)$. This probability distribution over instances is then mapped to a probability distribution over interpretation by applying the Well-Founded Model (WFM) semantics [15] to these logic programs. In other words, we define a probability distribution π_T over interpretation I as $\pi_T(I) = \sum_{WFM(T^\sigma)=I} \pi_\sigma$. This semantics is only defined for CP-theories which have the property that each instance T^σ has a two-valued well-founded model, which can be ensured by imposing a syntactic condition such as *stratification*.

In [17], an alternative—but equivalent—characterization of this semantics was developed. It makes use of the concept of an *execution model* of a CP-theory. This is a probability tree in which each node s is labeled with an interpretation $I(s)$. Such trees are constructed, starting from a root node in which all atoms are false, by "firing" rules whose body holds. The following is an execution model for Example 1. States s in which the bottle is broken (i.e., $I(s) \models Broken$) are represented by an empty circle, and those in which it is still whole by a full one.



To each branch of such a tree corresponds a set of instances of the theory, namely all those which make the same choices as the branch for all the rules that actually fire in the branch. For the rules that do not fire in the branch, it does not matter which choices the instances make. As shown in [17], the sum of the probabilities of all these instances is precisely the probability of the branch, and they all have the interpretation associated with the leaf 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 4.

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 π_T in the end [17].

3 A forward sampling method for positive CP-theories

The probability tree characterization 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 $\tilde{p}$ 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.

Algorithm 1 saturates a given interpretation I by applying rules from a set R

```
1: procedure SATURATE(T, I)                          ▷ Updates T and I
2:     Fired ← {}
3:     repeat
4:         ToFire ← {r ∈ T | I ⊨ body(r)}
5:         for all r ∈ ToFire do
6:             a ← RandomChoice(head(r))
7:             I(a) ← True
8:         T ← T \ ToFire
9:     until ToFire = {}
10: function RANDOMCHOICE((a_1, α_1), ..., (a_n, α_n))
11:     r ← random floating point number ∈ [0, 1]
12:     sum ← 0
13:     for all i ∈ [1, n] do        ▷ If Σ_{j<i} α_j ≤ r ≤ Σ_{j≤i} α_j, then return a_i
14:         sum ← sum + α_i
15:         if r ≤ sum then
16:             return a_i
```

At the end of this procedure, the interpretation I is saturated, in the sense that all rules $r \in R$, such that $I \models body(r)$, 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 $\tilde{P}_T$ be the probability distribution over all possible runs r of the probabilistic Algorithm 1 when called as* `Saturate`(T, F)*. For each possible run r, let n^r be the number of iterations of the inner for-loop (line 5), and, for each $1 \leq i \leq n^r$, let I_i^r be the value of I at the start of the ith iteration of this loop. Then there exists an execution model χ of T with a one-to-one mapping f between the branches of χ and the possible runs of the algorithm, such that, for each branch $b = (s_1, \ldots, s_n)$ of χ and the corresponding run $r = f(b)$, the probability of b is equal to $\tilde{P}_T(r)$ and $I(s_i) = I_i^r$ for all $1 \leq i \leq n^r$.*

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, 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 **False** to all atoms as the second. Algorithm 2 then estimates the probability of a query.

Algorithm 2 approximates the probability of a query Q according to π_T

```
1: function SAMPLE(T, Q)
2:     q ← 0, n ← 0          ▷ Nb of samples in which Q holds and total nb of samples
3:     I ← the interpretation such that all atoms are False
4:     repeat
5:         I ← Saturate(T, I)
6:         if I ⊨ Q then q ← q + 1
7:         n ← n + 1
8:     until desired accuracy reached                          ▷ See Section 5
9:     return q/n
```

4 A forward sampling method for general CP-theories

Negation in a CP-theory is interpreted by the WFM. In [17], the execution model semantics was also extended to this case. The WFM uses three-valued interpretations $\mathcal{I}$, in which atoms and formulas can be **Unknown** in addition to **True** and **False**. In our execution models, we will label each node s with a three-valued interpretation $\mathcal{I}(s)$, in addition to the two-valued $I(s)$. The true atoms of $\mathcal{I}(s)$ and $I(s)$ are always the same, but false atoms of $I(s)$ may be unknown in $\mathcal{I}(s)$. At the root r of the tree, $\mathcal{I}(r)$ assigns **Unknown** to all atoms.

The key difference with the positive case is that we now only allow rules to fire if their body is true in both $I(s)$ and in $\mathcal{I}(s)$ (so, in particular, rules which are true in $I(s)$ but unknown in $\mathcal{I}(s)$ 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 $\mathcal{I}(s)$. In this way, an atom that was originally unknown in some $\mathcal{I}(s)$ may become true in a child $\mathcal{I}(s')$. 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 when there is no longer any way in which they could still be caused. The details are as follows.

In each node s, we construct $\mathcal{I}(s)$ from $I(s)$ as the limit of a sequence $(\mathcal{I}'_i)_{i\geq 0}$ of three-valued interpretations. This sequence starts from the interpretation $\mathcal{I}'_0$ that coincides completely with $I(s)$ (and therefore has no unknown atoms). Given a $\mathcal{I}'_i$, we then select a rule r that has not yet fired and whose body is

either **True** or **Unknown** in $\mathcal{I}'_i$. We derive $\mathcal{I}'_{i+1}$ from $\mathcal{I}'_i$ by changing the truth value of all atoms from 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 $I(s)$ **Unknown**, but not (necessarily) all of them. The limit of this sequence is used as $\mathcal{I}(s)$.

For a child s' of a node s, the three-valued interpretation $\mathcal{I}(s')$ is therefore constructed from $I(s')$, and not from $\mathcal{I}(s)$ directly. Nevertheless, it can be shown [17] that $\mathcal{I}(s')$ is always more precise than $\mathcal{I}(s)$, in the sense that its true and false atoms, respectively, are a superset of those of $\mathcal{I}(s)$. This allows us to postpone the expensive operation of actually computing $\mathcal{I}(s')$: if a rule body is true in $\mathcal{I}(s)$ for some ancestor s of s', we can be sure that it is still true in $\mathcal{I}(s')$ itself. Moreover, if we construct a different $\mathcal{I}'$ from $\mathcal{I}(s)$ by making some additional atoms true that are also true in $I(s')$, then this $\mathcal{I}'$ is still less precise than $\mathcal{I}(s)$, and therefore all rule bodies that are true in $\mathcal{I}'$ must also be true in $\mathcal{I}(s')$.

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 $I \models \mathit{body}(r)$ in Line 11 of Algorithm 1 has to be interpreted as that the truth value of $\mathit{body}(r)$ has to be true according to I (i.e., unknown is not enough).

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

```
 1: function SingleSample(T)
 2:     I ← interpretation that maps each a appearing in T to Unknown
 3:     repeat
 4:         Saturate(T, I)                    ▷ Atoms go from Unknown to True
 5:         Shrink(I, T)                      ▷ Atoms go from Unknown to False
 6:     until fixpoint reached
 7:     return I
 8: function Shrink(T, I)                    ▷ computes I(s), given the true atoms of I
 9:     I' ← interpretation with the same true atoms as I and all other atoms false
10:     repeat
11:         ToFire ← {r ∈ T | body(r)^I' ≠ False}
12:         for all atoms a in some head(r) with r ∈ ToFire do
13:             I'(a) ← True
14:         T ← T \ ToFire
15:     until ToFire = {}
16:     return I'
```

5 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 independent, they 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.

An OpenCL program consists of two parts: there are *kernels*, which are executed in parallel on OpenCL-capable devices such 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 *work groups*. 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.

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, initialized to **Unknown**; an array `rules` in which they can store the truth value of each rule body (**True**, **False**, **Unknown** or "already fired"), also initialized to unknown; and a sequence of random numbers for the required random decisions. When a work item finishes, it 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 $(a_1^{w_i}, \ldots, a_m^{w_n})$ in which to store the truth value of the atoms $(a_1, \ldots, a_m)$. Instead of storing these arrays as $\big((a_1^{w_1}, \ldots, a_m^{w_1}), (a_1^{w_2}, \ldots, a_m^{w_2}), \ldots, (a_1^{w_n}, \ldots, a_m^{w_n})\big)$, it is more efficient to store them as $\big((a_1^{w_1}, \ldots, a_1^{w_n}), (a_2^{w_1}, \ldots, a_2^{w_n}), \ldots, (a_m^{w_1}, \ldots, a_m^{w_n})\big)$. 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: Initializing 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.

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.[1]

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.

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 $m \ll n$ 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 $i$the 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.

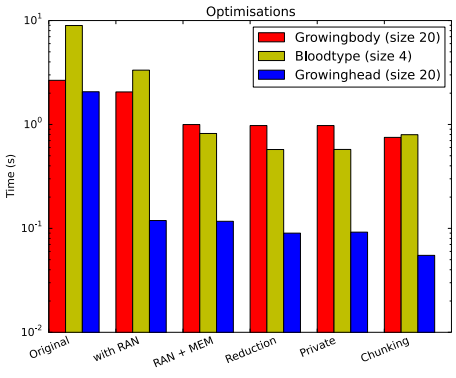


Fig. 1: Effect of different optimisations.

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 [4] uses a criterion based on the central limit theorem: $2z_{1-\alpha/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{k}} \leq \delta$, 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 δ 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. We will use the same criterion as Problog Monte

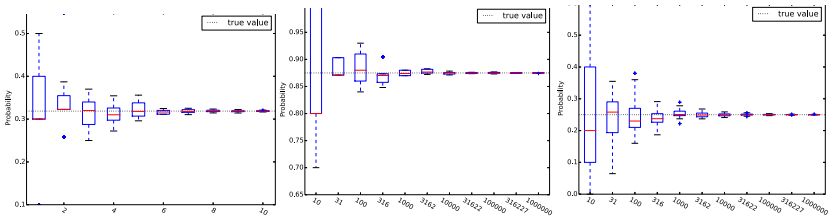
[1] `http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html`

Carlo and MCINTYRE. The worst case value for the number of samples k that may need to be computed is $k = 38416$.

6 Experimental results

The experiments presented in this section were run on a Linux machine with an Intel Core i7 965 CPU (3.20GHz) 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 1242MHz. 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, GROWINGHEAD and GROWINGBODY benchmarks from [6]. The latter two benchmarks consist of ground programs. However, the BLOODTYPE benchmark first needs to be grounded before we can apply our algorithm. Source code and graphs of our experiments can be downloaded at: `http://www.cs.kuleuven.be/~joost/PLP/`

Figure 1 investigates the effect the different optimisations. The variant *Reduction* contains the optimisations RAN+MEM+RED; *Private* extends this further with PRI; and *Chunking* is RAN+MEM+PRI+CHU. For each version, we have run a single instance of the three benchmarks. The smallest gain is observed in the GROWINGBODY benchmark, where the fully optimized version is only about 4× faster. For BLOODTYPE and GROWINGHEAD, the speedup is 15× and 37×, respectively. Our remaining experiments will therefore be conducted using the fully optimized version (*overlap*).



(a) BLOODTYPE (b) GROWINGHEAD (c) GROWINGBODY

Fig. 2: Convergence of the algorithm.

The convergence behavior of our algoritm is examined in Figures 2a, 2b, 2c. 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 5 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.

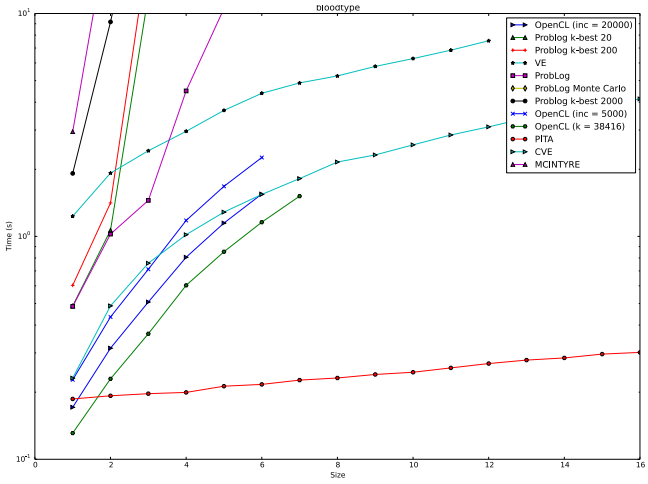


Fig. 3: Results for the Bloodtype benchmark.

Finally, we compare our algorithm to the approach of [5] which transforms CP-logic into a Bayesian network and then applies the Variable Elimination (VE) algorithm, to Contextual Variable Elimination (CVE) [6], to the default and Monte Carlo inference algorithms of the ProbLog system [4], to the PITA algorithm [12] that is built into XSB Prolog, and to the MCINTYRE backwards sampling algorithms [9]. For the k-best [4] and k-optimal [8] 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 Figures 3, 4 and 5. 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: *inc = 20000*, *inc = 5000* and $k = 38416$. The first two variants use the MCINTYRE stopping criterion, where *inc* is the chunk size (i.e., the number of samples that are run at once). A smaller *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 $k = 38416$ as a fixed number of samples. For k-best, we have included experiments with $k = 2000$, $k = 200$ and $k = 20$. As the results below show, the precise value of k has only a limited impact on the way in which this algorithm compares to the others.

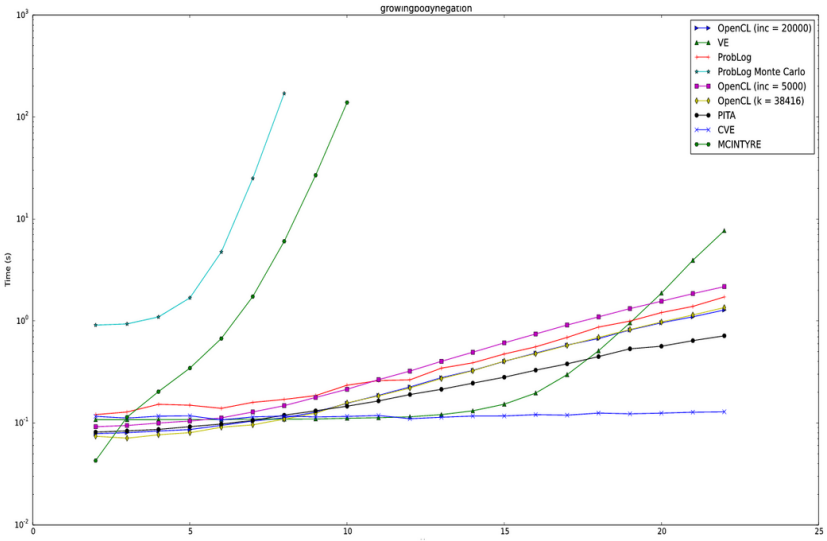


Fig. 4: Results for the GrowingBody benchmark.

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 10s), Problog Monte Carlo and the three version of k-best already require more than $100s$ 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 the more expensive ($\pm$ \$ 1500) 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 algoritm, 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 ($\sim 0.01s$–$0.1s$).

A similar phenomenon can be seen for the GrowingBody benchmark. Again, CVE and PITA perform best, comfortably handling problem sizes ≥ 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

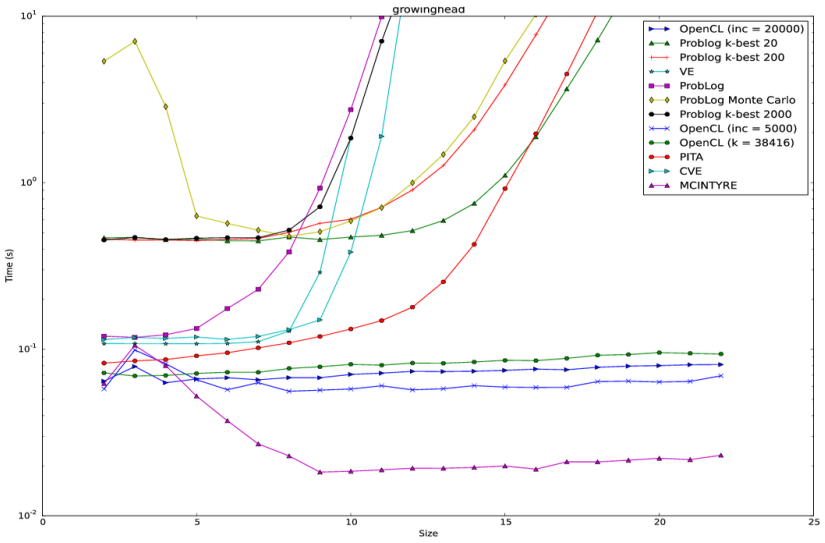


Fig. 5: Results for the GrowingHead benchmark.

sizes ≥ 20 in under a second. Finally, the GrowingHead benchmark is clearly better suited for the sampling methods. Problog MC is the worst of the sampling methods, being (slightly) outperformed by the exact inference of PITA, but with the gap narrowing for the larger problem sizes. For $k = 20$, 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 ≥ 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 $0.1s$) for this benchmark.

7 Discussion

We have tried to compare an approximative algorithm on GPU with exact algorithms, and with algorithms that run on CPU. To do this in a fair way, we have used comparatively priced processors (at the time of writing, both our CPU and GPU are worth around \$100-\$200) and have used the MCINTYRE stopping criterion to automatically determine the number of samples.

The main conclusion of our experiments is that our algorithm is a useful addition to the algorithms that are currently available in the literature.

First, our implementation proved quite robust. While all of the other algorithms had at least one benchmark in which their runtimes explode, our method was always able to keep reasonable track with the best performing algorithms. In one of the benchmarks, however, we did run into an inherent limitation of our approach: once theories no longer fit into the memory of the GPU, our current implementation cannot do anything. However, even for this benchmark, we

could still handle significantly larger theories than standard Problog inference, the k-best method and MCINTYRE, especially when using a higher-end GPU.

Second, our algorithm differs from the existing algorithms 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 could be useful in applications where the goal is to find the most probable of a set of atoms, or to compute their entire joint probability distribution. As such, it is a useful addition to the existing set of algorithms, which are all query-based.

Third, our algorithm may also be easier to extend to new language features, because it 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 probabilities 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 [17].) 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 [16] has investigated the possibility of extending CP-logic with negated atoms in the head. Integrating such features into a forward sampling algorithm seems much easier than extending one of the proof-based methods.

8 Conclusion and future work

We have presented a forward sampling method for the expressive PLP language of CP-logic and have shown how OpenCL can be used to execute this algorithm in parallel on GPU hardware. As demonstrated by our experiments, it is important to properly optimize this code in order to achieve efficient results. Once this is done, the implementation is competitive with existing algorithms. In particular, our experiments show that, while it is never the fastest method, ours has the most consistent performance throughout. Additional advantages are that it can compute the probabilities of a set of queries almost as quickly as that of a single query, and that it seems easier to adapt to extensions of the CP-logic language, in particular those that allow more expressive heads of rules.

One limitation of the 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.

References

1. L. A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. High Dimensional Pricing of Exotic European Contracts on a GPU Cluster, and Comparison to a CPU Cluster. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.
2. C. Baral, M. Gelfond, and N. Rushton. Probabilistic reasoning with answer sets. *Theory and Practice of Logic Programming*, 2008.
3. D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in probabilistic logic programs using weighted cnf's. In *Proc. UAI*, 2011.
4. A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming*, 11:235–262, 2011.
5. W. Meert, J. Struyf, and H. Blockeel. Learning ground CP-logic theories by leveraging Bayesian network learning techniques. *Fundamenta Informaticae* 89(1), 2008.
6. W. Meert, J. Struyf, and H. Blockeel. CP-logic theory inference with contextual variable elimination and comparison to BDD based inference methods. In *Proc. ILP*, 2009.
7. D. Poole. Abducing through negation as failure: Stable models within the independent choice logic. *Journal of Logic Programming* 44, 2000.
8. J. Renkens, G. Van den Broeck, and S. Nijssen. k-optimal: a novel approximate inference algorithm for problog. In *Proc. ILP*, 2011.
9. F. Riguzzi. MCINTYRE: A monte carlo system for probabilistic logic programming. *Fundamenta Informaticae*, 124(4):521–541, 2013.
10. F. Riguzzi and T. Swift. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *TPLP* 13:279–302, 2013.
11. Fabrizio Riguzzi. SLGAD resolution for inference on Logic Programs with Annotated Disjunctions. *Fundamenta Informaticae* 102(3-4):429–466, 2010.
12. F. Riguzzi and T. Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *TPLP* 11(4-5):433–449, 2011.
13. T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *Proceedings of IJCAI*, 1997.
14. D. Van Antwerpen. Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU. In *High Performance Graphics*, 41–50, 2011.
15. A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM* 38(3):620–650, 1991.
16. J. Vennekens. Negation in the head of CP-logic rules. In *Proc. ASPOCP*, 2013.
17. J. Vennekens, M. Denecker, and M. Bruynooghe. CP-logic: A language of causal probabilistic events and its relation to logic programming. *TPLP* 9(3), 2009.