Processing Markov Logic Networks with GPUs: Accelerating Network Grounding

Carlos Alberto Martínez-Angeles[1(✉)], Inês Dutra[2], Vítor Santos Costa[2], and Jorge Buenabad-Chávez[1]

[1] Departamento de Computación, CINVESTAV-IPN, Av. Instituto Politécnico Nacional 2508, 07360 México D.F., Mexico
camartinez@cinvestav.mx, jbuenabad@cs.cinvestav.mx

[2] Departamento de Ciência de Computadores, CRACS INESC-TEC LA and Universidade do Porto, Rua do Campo Alegre 1021, 4169-007 Porto, Portugal
{ines,vsc}@dcc.fc.up.pt

Abstract. Markov Logic is an expressive and widely used knowledge representation formalism that combines logic and probabilities, providing a powerful framework for inference and learning tasks. Most Markov Logic implementations perform inference by transforming the logic representation into a set of weighted propositional formulae that encode a Markov network, the ground Markov network. Probabilistic inference is then performed over the grounded network.

Constructing, simplifying, and evaluating the network are the main steps of the inference phase. As the size of a Markov network can grow rather quickly, Markov Logic Network (MLN) inference can become very expensive, motivating a rich vein of research on the optimization of MLN performance. We claim that parallelism can have a large role on this task. Namely, we demonstrate that widely available Graphics Processing Units (GPUs) can be used to improve the performance of a state-of-the-art MLN system, Tuffy, with minimal changes. Indeed, comparing the performance of our GPU-based system, TuGPU, to that of the Alchemy, Tuffy and RockIt systems on three widely used applications shows that TuGPU is up to 15x times faster than the other systems.

Keywords: Statistical relational learning · Markov logic · Markov logic networks · Datalog · Parallel computing · GPUs

1 Introduction

Statistical relational learning (SRL) integrates statistical reasoning, machine learning and relational representations. SRL systems rely on a first-order logic language to represent the structure and relationships in the data, and on graphical models to address noisy and incomplete information. Various SRL frameworks have been proposed, Stochastic Logic Programs (SLP), Probabilistic Relational Models (PRM), PRISM, Bayesian Logic Programs, ProbLog, CLP($\mathcal{BN}$), PFL, and Markov Logic [13,30].

K. Inoue et al. (Eds.): ILP 2015, LNAI 9575, pp. 122–136, 2016.
DOI: 10.1007/978-3-319-40566-7_9

The last few years have seen significant progress in models that can represent and learn from complex data. One important such model is Markov logic, "a language that combines first-order logic and Markov networks. A knowledge base in Markov logic is a set of first-order [logic] formulas with weights" [11]. Markov Logic thus builds upon logic and probabilities. The logical foundation of Markov Logic provides the ability to use first-order logic formulas to establish *soft constraints* over *worlds*, or interpretations. Worlds that violate a formula are less likely to be true, but still possible. In contrast, formulas in standard first-order logic are *hard constraints*: a world that falsifies a formula is not possible. Worlds that violate a formula can be possible in Markov Logic because worlds are an assignment to a set of random variables, and follow a probability distribution. The distribution is obtained by identifying each ground atom as a random variable, and each grounded formula as a clique in a factor graph. This ground network thus forms a Markov Random Field (MRF) [17].

Markov logic systems address two major tasks [31]: inference and learning. In *inference*, we receive an MLN model M and a set of observations, or evidence E, and we want to ask questions about the unobserved variables. Typical queries are:

- *probability estimation queries*: one wants to find out the probability of an atom given the evidence E. A typical example would be "What is the probability of rain in Kobe and Kyoto, given that it is raining in Tokyo and Nagoya, but sunny in Fukuoka and Okinawa". Notice that MLNs naturally allow for collective inference, that is, we can ask for all the different cities in a single query.
- Maximum a posteriori (MAP) or *most likely world queries*: one wants to find out what is the most likely set of values for the variables of interest. From our example above, instead of outputting probabilities, the model would output the places where it is more likely to rain.

A large number of inference techniques have been developed for MLNs. Most of them operate on the ground network, that is, given the query and the observed data, they enumerate all relevant atoms and then use statistical inference on the resulting network. They then search for the set of grounded clauses that *m*aximize the sum of the satisfied clauses weights.

The second task, *l*earning, is about constructing the actual MLNs. Often the formulas of interest can be obtained from the experts, but it is still necessary to learn the weights. *Parameter learning* addresses this task. *Structure learning* goes further and tries to construct the actual model, by searching for relationships or important properties of the data.

Markov logic networks have been widely adopted. Applications include the Semantic Network Extractor (SNE) [19], a large scale system that can learn semantic networks from the Web; the work by We *et al.* to refine Wikipedia's Infobox Ontology [43]; and Riedel and Meza-Ruiz's work to carry out collective semantic role labelling [33], among others [11, p. 97].

Alchemy was the first widely available Markov Logic system [11]. It is still a reference in the field, as it includes a very large number of algorithms that address most MLN tasks. However, as it did not scale well to large real-world applications, several new implementations have been proposed [4,26,27,32].

Grounding is, arguably, the step that mostly affects performance of MLNs, preventing them from scaling to large applications. For large domains, we may need to ground a very large number of atoms, which can be quite time and space-consuming. Often (but not always) the solver algorithm converges in few iterations and grounding will dominate running time [26]. We claim that GPU processing can significantly expedite grounding, and that this can be done effectively with few changes to the state-of-the-art systems. To verify our hypothesis, we designed TuGPU, a Markov Logic system based on: Tuffy [26], YAP Prolog [35], and GPU-Datalog, a GPU-based engine that evaluates Datalog programs [22]. We compare the performance of TuGPU to that of Alchemy [11], Tuffy and RockIt [27], with three applications of different types: information extraction, entity resolution and relational classification. The performance of TuGPU is on par or better than the other systems for most applications.

This paper is organized as follows. Section 2 presents background on Markov logic and its implementation, Tuffy, Datalog, and GPUs. Section 3 presents the design and implementation of our TuGPU platform for Markov logic networks. Section 4 presents an experimental evaluation of our platform. In Sect. 5, we discuss about our system and other related systems. We conclude in Sect. 6.

2 Markov Logic, Tuffy, Datalog and GPUs

First-order (predicate) logic is widely used for knowledge representation and inference tasks. Datalog is a language based on first-order logic that was initially investigated as a data model for relational databases in the 80s [41,42]; recent applications include declarative networking, program analysis, and security [15]. Interest in Datalog has always stemmed from its ability to compute the transitive closure of relations through recursive queries which, in effect, turns relational databases into deductive databases. Relational Learning is the task of learning from databases, modelling relationships among data items from multiple tables (relations); Inductive Logic Programming (ILP) [9] is a popular relational learning approach that employs logic-based formalisms, often based on subsets of first-order logic such as Horn clauses.

Statistical Relational Learning (SRL), in the form of probabilistic inductive logic programming, extends logic-based approaches by combining relational learning and probabilistic models (e.g., graphical models such as Bayesian networks and Markov networks), in order to manage the uncertainty arising from noise and incomplete information which is typical of real-world applications. Markov logic networks (MLNs) are a very popular approach that combines first-order logic and Markov networks in a simple manner: a weight is attached to each first-order logic formula that represents how strong the formula is as a constraint in all possible worlds. MLNs use inference to answer queries of the form: "What is the probability that formula F_1 holds given that formula F_2 does?" [31].

2.1 Inference in Markov Logic

A Markov Logic network is a set of formulas with attached weights. Internally, the program is stored as a conjunction of clauses, where each clause is a

disjunction of positive and negative atoms, as shown in the well-known *s*mokers example, which determines the probability of people having cancer (`Ca`) based on who their friends (`Fr`) are and whether or not their friends smoke (`Sm`):

$$1.5 : \neg\texttt{Sm(x)} \vee \texttt{Ca(x)}$$
$$1.1 : \neg\texttt{Fr(x,y)} \vee \neg\texttt{Sm(y)} \vee \texttt{Sm(x)}$$
$$0.7 : \neg\texttt{Fr(x,y)} \vee \neg\texttt{Fr(y,z)} \vee \texttt{Fr(x,z)}$$

Each ground instance of a literal, say `Ca(Anna)` can be seen as a boolean random variable (RV). RVs in a clause form a clique, and the set of all cliques a hypergraph. Assuming the network includes N ground atoms and R rules or cliques, such that clique i has size k_i, the Markov property says that the joint probability over the hypergraph is a normalized sum of products:

$$P(a_1, \ldots, a_N) = \frac{1}{\mathcal{Z}} \prod_R e^{w_i \phi(a_{i1}, \ldots, a_{ik_i})}$$

Each RV can take two values (0 or 1), hence we have 2^N disjoint configurations. The partition function $\mathcal{Z} = \sum_{(a_1=0 \ldots a_N=0)}^{(a_1=1 \ldots a_N=1)} \prod_R e^{w_i \phi(a_{i1} \ldots a_{ik_i})}$ sums up all the different values and ensures that the total probabilities add up to one (1). Usually there is no closed form for $\mathcal{Z}$.

The boolean function ϕ is 1 if the clause i is true under this grounding, 0 otherwise. Thus, a false grounding contributes $e^0 = 1$ to the product, and a true grounding e^w: in other words, if $w = 1.5$, a world with that grounding is $e^{1.5}$, which is approximately 5 times more likely than a world whose grounding is false. As $wF \equiv -w\neg F$ (where F is a a clique i), we can always ensure that weights are positive or zero, hence the probability of a world where all constraints are soft is $0 < \frac{1}{\mathcal{Z}} < \frac{\prod e^{w_i}}{\mathcal{Z}} < 1$: strictly larger than zero and always less than one.

Inference is most often divided in two phases: *grounding* and *search*. Grounding is the process of assigning values to all free variables in each clause. While we can ground a clause by assigning all possible values to its variables, it is impractical even for small domains. There are several, more efficient alternatives that discard unnecessary groundings, such as lazy closure grounding and inference [28]. In a number of cases, one can obtain even better results by using *lifted inference*, that avoids grounding the program [38].

Next we focus on the most common inference task, Maximum a Posteriori (MAP), where we search for the most probable state of the world given the observed data or *evidence* E; that is, we search for an assignment of $a_1 \ldots a_N$ that maximizes $P(E|a_1 \ldots a_N) \propto P(a_1 \ldots a_N|E) = \frac{P(a_1 \ldots a_N, E)}{P(E)}$. Thus, we have:

$$\begin{aligned}
\text{argmax}_{a_1 \ldots a_N} P(a_1 \ldots a_N|E) &= \text{argmax}_{a_1 \ldots a_N} \frac{1}{P(E)\mathcal{Z}} \prod_R e^{w_i \phi(a_{i1} \ldots a_{ik_i})} \\
&= \text{argmax}_{a_1 \ldots a_N} \prod_R e^{w_i \phi(a_{i1} \ldots a_{ik_i})} \\
&= \text{argmax}_{a_1 \ldots a_N} log \prod_R e^{w_i \phi(a_{i1} \ldots a_{ik_i})} \\
&= \text{argmax}_{a_1 \ldots a_N} \sum_R w_i \phi(a_{i1} \ldots a_{ik_i})
\end{aligned}$$

$\mathcal{Z}$ and $P(E)$ are the same for every world, so they do not affect the optimization problem. Moreover, applying a monotonic function such as the logarithm will

preserve the maximum, but enable us to work on a sum. Observing closely, the problem reduces to finding the maximal value of discrete function of boolean variables. Notice that if the coefficients w_i are positive, and the underlying boolean formula is satisfiable, an assignment that satisfies the model will be optimal. Thus, finding a solution to this problem requires solving the satisfiability problem with weights.

Several Markov logic systems use MaxWalkSAT [16] for its ability to solve hard problems with thousands of variables in a short time. MaxWalkSAT works by selecting an unsatisfied clause and switching the truth value of one of its atoms. The atom is chosen either randomly or to maximize the sum of the satisfied clause weights.

2.2 Optimizations

A ground MLN may quickly have thousands of boolean variables, making it hard to find even an approximate solution. Thus, it is important to start by simplifications of the system. Typically, one applies a combination of two techniques:

- *Elimination*: Consider a clause $\neg\texttt{Sm(j)} \vee \neg\texttt{Sm(k)}$, with evidence $\neg\texttt{Sm(j)}$ and query variable $\texttt{Sm(k)}$. The clause is always true, hence it does not affect the total score, and can be dropped.
- *Partitioning*: Consider $c_1 \equiv a \vee \neg b$ and $c_2 \equiv c \vee d$. If $(\texttt{a}, \texttt{b})$ is a solution to c_1, and $(\texttt{c}, \texttt{d})$ is a solution to c_2, then we have that $(\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d})$ is a solution to the joint network. In practice this means the two sub-problems can be solved independently.

Most MLN systems apply these principles to reduce the search space. To speed up inference in large relational problems, *lazy grounding* takes the idea further and grounds as late as possible. The idea is to take advantage of the fact that most of their groundings are known to be trivial or false beforehand. The other approach, lifted inference, groups indistinguishable atoms together and treats them as a single unit, thus reducing the size of the network.

2.3 Learning

Learning is used to automatically create or refine weights and to create clauses in an MLN. Weights can be learned generatively or discriminatively; clauses are learned using Inductive Logic Programming (ILP) [9]. The learning process makes repeated use of the inference phase, using one of the methods described below. However, it is common of many applications to use only the inference phase with an already configured knowledge base (KB) and a number of facts in relational tables, as is the case of the applications we use in our experiments in Sect. 4. Learning is not considered any further in the paper after this subsection.

In *generative* weight learning, the idea is to maximize the likelihood (a function of the parameters of our statistical model) of our training evidence following the closed-world assumption [12]. i.e.: all ground atoms not in the database are

false. However, computing the likelihood requires all true groundings of each clause, a difficult task even for a single clause [34]. MLNs use pseudo-log likelihood instead [6], which consists of a logarithmic approximation of the likelihood. Combined with a good optimizer like L-BFGS [7], pseudo-log likelihood can create weights for domains with millions of groundings.

Discriminative weight learning is used to predict query atoms given that we know the value of other atoms. This is achieved by maximizing the conditional log-likelihood (CLL: a constrained version of the log-likelihood), instead of the pseudo-log likelihood [37]. Maximizing CLL can be performed by optimizer algorithms like Voted Perceptron, Diagonal Newton, Scaled Conjugate Gradient, among others [11].

Clauses can be learned using ILP algorithms. The most important difference is the use of an evaluation function based on pseudo-likelihood, rather than accuracy or coverage. These modified methods include top-down structure learning [18] (TDSL) and bottom-up structure learning [24] (BUSL). On real-world application against famous ILP systems like CLAUDIEN [10], FOIL [29] and Aleph [39], both TDLS and BUSL find better MLN clauses.

2.4 Tuffy

Tuffy [26] is an MLN system that employs a bottom-up approach to grounding that allows for a more efficient procedure, in contrast to the top-down approach used by other systems. It also performs an efficient local search using an RDBMS. Inference is performed using the MaxWalkSat algorithm mentioned in Sect. 2.1. Tuffy can also perform parameter learning, but it does not implement structure learning (creating new clauses). In order to speedup execution, it partitions the MRF formed by the grounded clauses so as to perform random walks in parallel for each partition.

2.5 Evaluation of Datalog Programs

Datalog programs can be evaluated through a top-down approach or a bottom-up approach. The top-down approach (used by Prolog) starts with the goal that is reduced to subgoals, or simpler problems, until a trivial problem is reached. It is tuple-oriented: each tuple is processed through the goal and subgoals using all relevant facts. Because evaluating each goal can give rise to very different computations, the top-down approach is not easily adapted to GPUs *bulk* parallelism — more on this below and in Sect. 2.6.

The bottom-up approach first applies the rules to the given facts, thereby deriving new facts, and repeats this process with the new facts until no more facts are derived. The query is considered only at the end, to select the facts matching the query. Based on relational operations (as described shortly), this approach is suitable for GPUs because such operations are set-oriented and relatively simple overall. Also, rules can be evaluated in any order. This approach can be improved using the magic sets transformation [5] or the subsumptive tabling

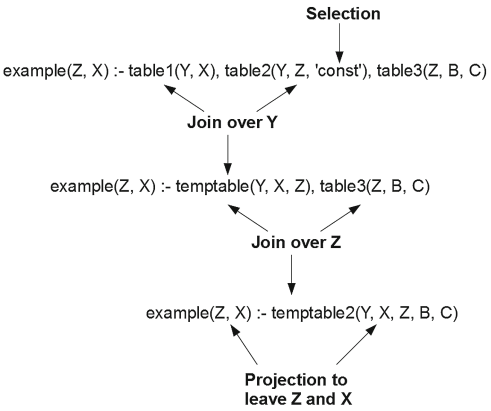

Fig. 1. Evaluation of a Datalog rule based on relational algebra operations.

transformation [40], through which the set of facts that can be inferred tends to contain only facts that would be inferred during a top-down evaluation.

Bottom-up evaluation of Datalog rules can be implemented with the relational algebra operators *selection*, *join* and *projection*, as outlined in Fig. 1. *Selections* are made when constants appear in the body of a rule. Next, a *join* is made between two or more subgoals in the body of a rule using the variables as reference. The result of a join can be seen as a temporary subgoal (or table) that has to be joined in turn to the rest of the subgoals in the body. Finally, a *projection* is made over the variables in the head of the rule.

We use fixed-point evaluation to compute recursive rules [41]. The basic idea is to iterate through the rules in order to derive new facts, and using these new facts to derive even more new facts until no new facts are derived.

2.6 GPU Architecture and Programming

GPUs are high-performance many-core processors capable of very high computation and data throughput [2]. They are used in a wide variety of applications [3]: games, data mining, bioinformatics, chemistry, finance, imaging, weather forecast, etc. Applications are usually accelerated by at least one order of magnitude, but accelerations of 10 times or more are common.

GPUs are akin to single-instruction-multiple-data (SIMD) machines: they consist of many processing elements that run the *same program* but on distinct data items. This program, referred to as the *kernel*, can be quite complex including control statements such as *if* and *while* statements. A kernel is executed by groups of threads called *warps* [1]. These warps execute one common instruction at a time, so all threads of a warp must have the same execution path in order to obtain maximum efficiency. If some threads diverge, the warp serially executes each branch path, disabling threads not on that path, until all paths complete and the threads converge to the same execution path. Hence, if a kernel has to

compare strings, processing elements that compare longer strings will take longer and other processing elements that compare shorter strings will have to wait.

GPU memory is organized hierarchically. Each (GPU) thread has its own *per-thread local* memory. Threads are grouped into *blocks*, each block having a memory *shared* by all threads in the block. Finally, thread blocks are grouped into a single *grid* to execute a kernel — different grids can be used to run different kernels. All grids share the *global memory*.

3 Our GPU-Based Markov Logic Platform

Our platform TuGPU was designed to accelerate the grounding step, as this is often the most time consuming. Its main components are: the Tuffy Markov logic system [26], the YAP Prolog system [35] and GPU-Datalog [22]. The latter evaluates Datalog programs with a bottom-up approach using GPU kernels that implement the relational algebra operations *selection, join* and *projection*. For GPU-Datalog to be able to run Markov logic networks, its original version was extended with: management of stratified negation; improved processing of built-in comparison predicates; processing of disjunctions, in addition to conjunctions (to simplify specifying SQL queries as Datalog queries and to improve their processing); and an interface to communicate directly with PostgreSQL.

Figure 2 shows the interaction between the main modules of our platform in running a Markov logic network. Tuffy is called first, receiving three input files: (i) the evidence (facts) file; (ii) the MLN file; and the queries. Tuffy starts by creating a temporary database in PostgreSQL to store the evidence data and partial results (left side of Fig. 2). It then parses the program and query files in order to determine predicates and to create a (relational) table for each predicate found. Tables are then loaded with the evidence data.

The original Tuffy would then start the grounding phase. In TuGPU, this phase is performed by GPU-Datalog (center of Fig. 2), but, as Tuffy uses conjunctions to

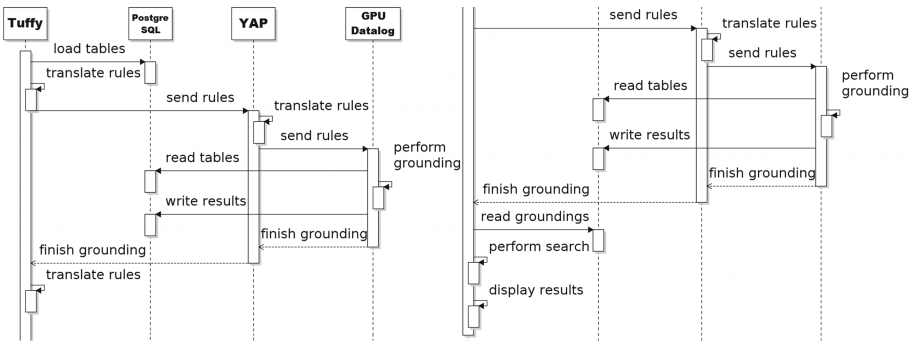

Fig. 2. TuGPU-Datalog modules running a Markov logic network. The left part corresponds to the active atoms grounding, while the right corresponds to the active clauses grounding.

specify a program, we first translate it to Datalog disjunctions. Then the Datalog program is sent to YAP, using a Java-Prolog interface, to compile it into a numerical representation (NR) where each unique string is assigned a unique integer id. YAP then sends the program's NR to GPU-Datalog to process the grounding. By using an NR, our GPU kernels show relatively short and *constant* processing time because all tuples in a table, being managed as sets of integers, can be processed in the same amount of time. Tuffy also uses an NR for evidence loaded in the database; this simplified extending it with GPU processing. Weights are not used in this phase, since the search for the most probable world will be performed by the host after the grounding is done.

To speed-up the grounding, Tuffy and TuGPU use the Knowledge-Based Model Construction [26] (KBMC) algorithm to determine those atoms and clauses that are relevant to the query. Then, GPU-Datalog reads the evidence from the database and performs the first step (of two) of the grounding process: computing the closure of the *active atoms* (i.e., those atoms whose truth value might change from true to false or vice versa, during search). The second step determines the *active clauses*, clauses that can be violated (i.e., their truth value becomes false) by flipping zero or more active atoms. For this step, TuGPU translates the program rules from the SQL that Tuffy generates into Datalog, and then YAP translates it into the NR used by GPU-Datalog.

When GPU-Datalog finishes each grounding step, it writes the found *active atoms* or *clauses* to the database. At the end of both grounding steps, Tuffy searches for the most likely world of the MLN. The search begins by using the ground active atoms and clauses to construct the MRF and then partition it into components. Each component has a subset of the active atoms and clauses, so that if an atom is flipped, it affects only those clauses found in the component.

The partitioned MRF is processed in parallel by the CPU with one thread per core and one component per thread, using the MaxWalkSAT algorithm mentioned in Sect. 2.1. The algorithm stops after a certain number of iterations or after an error threshold is reached. Finally, the results are displayed by TuGPU.

4 Experimental Evaluation

This section describes our experimental evaluation of the performance of TuGPU compared to that of the systems Alchemy [11], Tuffy [26] and RockIt [27].

4.1 Applications and Hardware-Software Platform

We used the following applications available with the Tuffy package. Table 1 shows some of their characteristics. For two of them (ER and RC), more tuples were randomly generated to test the systems with bigger data (right column).

– *Entity Resolution (ER)*: a simple, recursive MLN to determine if a person has cancer based on who his/her friends are and their smoking habits (this is an example from [31]).

Table 1. Applications characteristics.

Application	Inference rules	Evidence relations	Tuples in relations	
			Original	Random
ER	3	3	8	(310,000)
RC	15	4	82,684	(441,074)
IE	1024	18	255,532	(na)

- *Relational Classification (RC)*: classifies papers into 10 categories based on authorship and on the categories of other papers it references (Cora dataset [23]).
- *Information Extraction (IE)*: given a set of Citeseer citations, divided in tokens, rules with constants are used to extract structured records.

The original number of tuples of ER is only 8. We created another data set with a larger number of tuples, 310,000, with randomly generated data: creating a fixed number of people, assigning a small random number of friends to each person, and labelling a fixed number of people as smokers.

For RC we also created a larger, randomly generated data set with 441,074 tuples. We used a fixed number of papers and authors, and the same categories found in the original data: each author has a small random number of written papers, each paper is referred to by a small random number of other papers, and a small fixed number of papers are already labeled as belonging to a particular category.

We ran our experiments in the following hardware-software platform. **Host**: an AMD Opteron 6344, 12 cores CPU, with 64 GB DRAM. **GPU**: a Tesla K40c, 2880 CUDA Cores, with 12 GB GDDR5 memory and CUDA Capability 3.5. **Software**: CentOS 7, PostgreSQL 9.5 and CUDA Toolkit 7.0.

4.2 Results

Figure 3 shows the performance of the systems using the original datasets available in the Tuffy package and on our extended versions of these datasets. The left side shows our system to be the fastest in 2 out of the 3 original datasets, but only by a few seconds relative to standard Tuffy. Alchemy was the fastest in ER because the dataset is small and does not incur overhead setting up a database. We were unable to execute IE in RockIt, hence the empty space in the graph. Figure 3 (right) shows the performance of the systems with the extended datasets. For ER, our system was 15 times faster than RockIt and 77 times faster than Alchemy. Tuffy did not finish the grounding after more than 3 h. RockIt was 2.5 times faster than our system for RC. Both Tuffy and Alchemy did not finish after more than 5 h.

We performed a detailed analysis to determine why our system performed so well in ER and so poorly in RC. For ER, our random data, combined with

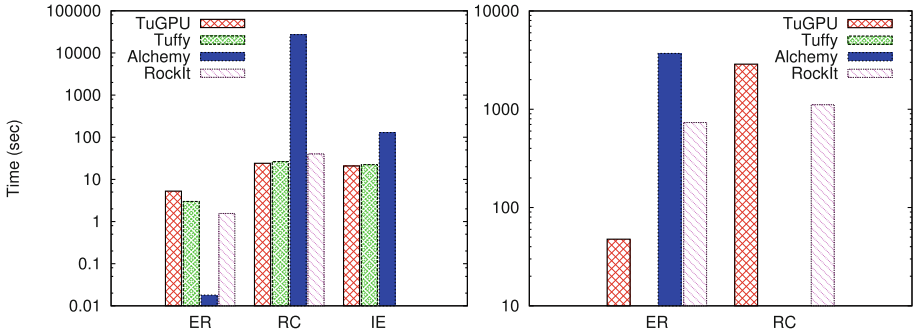

Fig. 3. Performance of the systems with original (left) and random (right) datasets. Note that the graphs are in log. scale.

its recursive clauses, generates many more recursive steps, 24 vs 2 in the original data. Each recursive step creates new tuples that need to be evaluated again. In our system, approximately 1,000,000 new tuples were generated in each iteration, most of them to be later discarded by our duplicate elimination kernel. Since our system was designed around these recursive applications, grounding was finished rather quickly while other systems struggled with costly joins that do not capitalize on parallel processing.

In RC, the number of recursive steps was 2 for both normal and random datasets. We hence analyzed the execution times of each part of our system. Using our random data, both atom and clause grounding take about 2 min to complete, loading data and other tasks take 30 s, but the search phase takes an astounding 43 min. In contrast, the times for ER are about 8 s for both groundings, 21 s for data loading and other tasks, and 16 s for the search.

Also in the search phase, ER, despite generating many more intermediate tuples during grounding, uses only 252,249 active clauses, while RC uses 5,586,900. Furthermore, when partitioning the resulting MRF, we get a single component with approximately 4,000,000 active clauses. Since each component is assigned to a thread (and one thread to each CPU-core), smaller components finish quickly and we are left with a very large component being processed by a single core, thus dominating the execution time. In contrast, RockIt creates a large optimization problem but its parallel resolution has a much better balanced workload.

Overall these results are promising since they mean that the benefit of performing the grounding phase on the GPU outweighs the overhead associated with the database and GPU I/O, even for rather small datasets.

5 Related Work

The wide adoption of Markov logic for various types of applications has fostered the development of various systems and research on improvements. Alchemy was

the first Markov logic system implementation [11]. It is one of the most complete systems, including various algorithms for inference following a top-down approach, various techniques for learning weights and structure and more. The original Alchemy always performs inference by first grounding the program and then using approximated methods either based in MCMC (Markov Chain Monte Carlo), such as MC-SAT and Gibbs sampling, or variants of belief propagation. Alchemy supports both total probability and most likely explanation queries, and also provides a large number of learning algorithms. However, Alchemy does not cope well with large real-world applications.

Tuffy was developed by Feng Niu *et al.* [26]. It relies on PostgreSQL relational database management system (RDBMS) to perform inference. Tuffy follows a bottom-up approach to solve the grounding step. This allows the grounding to be expressed as SQL queries which, combined with query optimization by the RDBMS, allows Tuffy to complete the grounding faster than Alchemy.

Several other systems are available. theBeast, developed by Riedel [32], uses Cutting Planes Inference (CPI) optimization, which instantiates and solves small parts of a complex MLN network. This takes advantage of the observation that inference can be seen as either a MAX-SAT problem or as an integer linear programming problem (i.e. a mathematical optimization problem where the variables are restricted to integers). While theBeast is faster than Alchemy for some problems, it lacks many of Alchemy's features such as structure learning and MPE (Most Probable Explanation) inference [21].

RockIt is a recent system by Noessner *et al.* [27]. It treats the inference problem as an integer linear programming problem and includes a new technique called cutting plane aggregation (CPA) which, coupled with shared-memory multi-core parallelism during most of the inference, allows RockIt to outperform all other systems.

Beedkar *et al.* implemented fully parallel inference for MLNs [4]. Their system parallelizes grounding by considering each clause as a set of joins and partitioning them according to a single *join graph*. The search step of inference is also parallelized using importance sampling together with MCMC [20]. Since the MLN is partitioned during grounding, no further partitioning is required before searching. This approach is more efficient than Tuffy's since the partition is performed over a smaller, data independent graph. Experimental evaluation shows that this is faster and produces similar results when compared with Tuffy.

Other works speedup inference and learning with MLNs. Shavlik and Natarajan [36] propose ways of speeding up inference by using a preprocessing algorithm that can substantially reduce the effective size of MLNs by rapidly counting how often the evidence satisfies each formula, regardless of the truth values of the query atoms. Mihalkova and Mooney [24] and Davis and Domingos [8] have proposed bottom-up methods that can improve structure learning time and accuracy over existing top-down approaches. Mihalkova and Richardson [25] proposes to cluster query atoms and then perform full inference for only one representative from each cluster.

Our system is the first one to run Markov logic networks using GPUs. Since Datalog and MLNs share an equivalent syntax, a modified version of our

GPU-Datalog engine was used. Like Tuffy, our system uses a bottom-up approach based on relational operators to process one of the most time consuming parts of the inference step, but in a GPU.

Similar to our work on GPU-Datalog (described in Sect. 3), Wu *et al.* created Red Fox [44], a system that parallelizes relational algebra and other operations in the GPU, in order to solve programs based on a variant of Datalog called LogiQL. Comparison with GPU-Datalog using the famous TCP-H queries can be found in [22]. Other similar systems that execute SQL queries in parallel using the GPU include [14,45].

6 Conclusions

We have presented a system that accelerates the grounding step in MLNs by combining Tuffy with our GPU-Datalog engine. Its performance is on par or better than other well-known MLN systems. Our results show that the benefit of performing the grounding phase on the GPU outweighs the overhead of using a database and of GPU I/O, even for rather small datasets. Our system can be greatly improved by also performing the search step of the inference phase in the GPU. This would require the parallelization of a SAT solver. There are several available in the literature and we expect to benefit from extensive work in parallelization of SAT and ILP solvers.

Our GPU-Datalog system could benefit from data partitioning algorithms. This would allow tables bigger than the amount of GPU memory available to be processed.

Since GPU-Datalog has been successfully used to improve ILP [22], we believe that clause learning in MLNs could also be improved by our system. We also plan to research the parallelization of generative and discriminative weight learning.

Acknowledgments. This work is partially financed by the ERDF - European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project <<POCI-01-0145-FEDER-006961>>, and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project UID/EEA/50014/2013. Also, Martínez-Angeles gratefully acknowledges grants from CONACYT and Cinvestav-IPN. Finally, we gratefully acknowledge the major contribution of the referees, whose recommendations have led us to significantly improve the paper.

References

1. Cuda C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
2. General-Purpose Computation on Graphics Hardware, March 2015. http://gpgpu.org/
3. GPU Applications, March 2015. http://www.nvidia.com/object/gpu-applications-domain.html

4. Beedkar, K., Del Corro, L., Gemulla, R.: Fully parallel inference in markov logic networks. In: 15th GI-Symposium Database Systems for Business, Technology and Web, BTW 2013. Bonner Kllen, Magdeburg (2013)
5. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program. **10**(3–4), 255–299 (1991)
6. Besag, J.: Statistical analysis of non-lattice data. J. R. Stat. Soc. Ser. D (Stat.) **24**(3), 179–195 (1975)
7. Byrd, R.H., Lu, P., Nocedal, J., Zhu, C.: A limited memory algorithm for bound constrained optimization. SIAM J. Sci. Comput. **16**(5), 1190–1208 (1995)
8. Davis, J., Domingos, P.: Bottom-up learning of markov network structure. In: Proceedings of the Twenty-Seventh International Conference on Machine Learning, pp. 271–280. ACM Press (2010)
9. De Raedt, L.: Logical and Relational Learning. Springer, Heidelberg (2008)
10. De Raedt, L., Dehaspe, L.: Clausal discovery. Mach. Learn. **26**(2–3), 99–146 (1997)
11. Domingos, P., Lowd, D.: Markov Logic: An Interface Layer for Artificial Intelligence, 1st edn. Morgan and Claypool Publishers, San Rafael (2009)
12. Genesereth, M.R., Nilsson, N.J.: Logical Foundations of Artificial Intelligence. Morgan Kaufmann Publishers Inc., San Francisco (1987)
13. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)
14. He, B., et al.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. (TODS) **34**(4), 21 (2009)
15. Huang, S.S., et al.: Datalog and emerging applications: an interactive tutorial. In: SIGMOD Conference, pp. 1213–1216 (2011)
16. Kautz, H., Selman, B., Jiang, Y.: A general stochastic approach to solving problems with hard and soft constraints. In: The Satisfiability Problem: Theory and Applications, pp. 573–586. American Mathematical Society (1996)
17. Kindermann, R., Snell, J.L.: Markov Random Fields and their Applications, 1st edn. American Mathematical Society, Providence (1980)
18. Kok, S., Domingos, P.: Learning the structure of markov logic networks. In: Proceedings of the 22Nd International Conference on Machine Learning, ICML 2005, pp. 441–448. ACM, New York (2005)
19. Kok, S., Domingos, P.: Extracting semantic networks from text via relational clustering. In: Daelemans, W., Goethals, B., Morik, K. (eds.) ECML PKDD 2008, Part I. LNCS (LNAI), vol. 5211, pp. 624–639. Springer, Heidelberg (2008)
20. Koller, D., Friedman, N.: Probabilistic Graphical Models. MIT Press, Cambridge (2009)
21. Kwisthout, J.: Most probable explanations in bayesian networks: complexity and tractability. Int. J. Approx. Reason. **52**(9), 1452–1469 (2011)
22. Martínez-Angeles, C.A., Wu, H., Dutra, I., Santos-Costa, V., Buenabad-Chávez, J.: Relational learning with GPUs: accelerating rule coverage. Intl. J. Parallel Programm. **44**(3), 663–685 (2016). doi:10.1007/s10766-015-0364-7. http://dx.doi.org/10.1007/s10766-015-0364-7
23. McCallum, A., Nigam, K., Rennie, J., Seymore, K.: Automating the construction of internet portals with machine learning. Inf. Retrieval **3**(2), 127–163 (2000)
24. Mihalkova, L., Mooney, R.J.: Bottom-up learning of markov logic network structure. In: Proceedings of the Twenty-Fourth International Conference on Machine Learning, pp. 625–632. ACM Press (2007)
25. Mihalkova, L., Richardson, M.: Speeding up inference in statistical relational learning by clustering similar query literals. In: Raedt, L. (ed.) ILP 2009. LNCS, vol. 5989, pp. 110–122. Springer, Heidelberg (2010)

26. Niu, F., Ré, C., Doan, A., Shavlik, J.: Tuffy: scaling up statistical inference in markov logic networks using an rdbms. Proc. VLDB Endow. **4**(6), 373–384 (2011)
27. Noessner, J., Niepert, M., Stuckenschmidt, H.: Rockit: exploiting parallelism and symmetry for MAP inference in statistical relational models. CoRR, abs/1304.4379 (2013)
28. Poon, H., Domingos, P., Sumner, M.: A general method for reducing the complexity of relational inference and its application to mcmc. In: Proceedings of the 23rd National Conference on Artificial Intelligence, AAAI 2008, vol. 2, pp. 1075–1080. AAAI Press (2008)
29. Quinlan, J.R.: Learning logical definitions from relations. Mach. Learn. **5**(3), 239–266 (1990)
30. De Raedt, L., Kersting, K.: Probabilistic inductive logic programming. In: Raedt, L., Frasconi, P., Kersting, K., Muggleton, S.H. (eds.) Probabilistic Inductive Logic Programming. LNCS (LNAI), vol. 4911, pp. 1–27. Springer, Heidelberg (2008)
31. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. **62**(1–2), 107–136 (2006)
32. Riedel, S.: Improving the accuracy and efficiency of map inference for markov logic. In: Proceedings of the 24th Annual Conference on Uncertainty in AI, UAI 2008, pp. 468–475 (2008)
33. Riedel, S., Meza-Ruiz, I.: Collective semantic role labelling with markov logic. In: Proceedings of the Twelfth Conference on Computational Natural Language Learning, CoNLL 2008, pp. 193–197. Association for Computational Linguistics, Stroudsburg (2008)
34. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1–2), 273–302 (1996)
35. Santos-Costa, V., et al.: The YAP Prolog system. TPLP **12**(1–2), 5–34 (2012)
36. Shavlik, J., Natarajan, S.: Speeding up inference in markov logic networks by preprocessing to reduce the size of the resulting grounded network. In: Proceedings of the 21st International Jont Conference on Artifical Intelligence, IJCAI 2009, pp. 1951–1956. Morgan Kaufmann Publishers Inc., San Francisco (2009)
37. Singla, P., Domingos, P.: Discriminative training of markov logic networks. In: Proceedings of the 20th National Conference on Artificial Intelligence, AAAI 2005, vol. 2, pp. 868–873. AAAI Press (2005)
38. Singla, P., Domingos, P.: Lifted first-order belief propagation. In: Proceedings of the 23rd National Conference on Artificial Intelligence, AAAI 2008, vol. 2, pp. 1094–1099. AAAI Press (2008)
39. Srinivasan, A.: The Aleph Manual (2001)
40. Tekle, K.T., Liu, Y.A.: More efficient datalog queries: subsumptive tabling beats magic sets. In: SIGMOD Conference, pp. 661–672 (2011)
41. Ullman, J.: Principles of Database and Knowledge-Base Systems, vol. I. Computer Science Press, Rockville (1988)
42. Ullman, J.: Principles of Database and Knowledge-Base Systems, vol. II. Computer Science Press, Rockville (1989)
43. Wu, F., Weld, D.S.: Automatically refining the wikipedia infobox ontology. In: Proceedings of the 17th International Conference on World Wide Web, WWW 2008, pp. 635–644. ACM, New York (2008)
44. Wu, H., Diamos, G., Sheard, T., Aref, M., Baxter, S., Garland, M., Yalamanchili, S.: Red fox: an execution environment for relational query processing on gpus. In: International Symposium on Code Generation and Optimization (CGO) (2014)
45. Yuan, Y., Lee, R., Zhang, X.: The yin and yang of processing data warehousing queries on gpu devices. Proc. VLDB Endow. **6**(10), 817–828 (2013)