Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Procesamiento Paralelo de Redes Lógicas de Markov en Unidades de Procesamiento Gráfico

Tesis que presenta

Carlos Alberto Martínez Angeles

para obtener el Grado de

Doctor en Ciencias en Computación

Directores de tesis:

Dra. Ana María Martínez Enríquez

Dr. Jorge Buenabad Chávez

México, DF **Marzo del 2018**

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Parallel Processing of Markov Logic Networks on Graphics Processing Units

Tesis que presenta

Carlos Alberto Martínez Angeles

para obtener el Grado de

Doctor en Ciencias en Computación

Directores de tesis:

Dra. Ana María Martínez Enríquez

Dr. Jorge Buenabad Chávez

México, DF **Marzo del 2018**

Resumen

La lógica de Markov es un lenguaje para representar conocimiento que combina lógica y probabilidades, creando un sistema eficiente de inferencia y aprendizaje. La *inferencia* implica *sustitución* y *búsqueda*. La *sustitución* transforma la representación lógica en clausulas proposicionales con pesos que codifican una Red Markov, formando la red lógica de Markov (MLN) sustituida. La *búsqueda* sobre una MLN sustituida provee de *inferencia* probabilística. El *aprendizaje* determina los pesos de las clausulas o su estructura, aplicando inferencia repetidamente. Como el espacio de soluciones de una MLN puede crecer rápidamente, su procesamiento se vuelve muy costoso, por lo que se ha buscado mejorarlo. El paralelismo en multinúcleos se utiliza en sistemas para MLNs como Tuffy y RockIt, pero no conocemos ningún otro trabajo basado en el paralelismo masivo que las Unidades de Procesamiento Gráfico (GPUs) proveen.

Esta tesis presenta el diseño, implementación y evaluación de plataformas paralelas para MLNs en GPUs. Diseñamos la paralelización de las etapas más lentas del procesamiento de MLNs, describiendo los problemas encontrados y nuestras propuestas para solucionarlos. Para evaluar nuestros diseños los integramos en Tuffy y RockIt, pues crear todo desde cero va más allá del enfoque de esta tesis. Nuestras contribuciones incluyen: *sustitución* procesada en paralelo utilizando nuestro sistema de programación lógica; *búsqueda* particionando y resolviendo un problema de satisfacibilidad booleana con nuestro algoritmo paralelo o un problema de programación lineal; *aprendizaje de parametros* con *sustitución* paralela y resolución de un problema de optimización, ajustado con nuestro algoritmo de muestreo paralelo; y *aprendizaje de la estructura* utilizando Programación Lógica Inductiva, cuya operación más intensiva es calculada con nuestro sistema de programación lógica.

Nuestras plataformas fueron probadas con aplicaciones de la literatura y funcionaron muy bien, pues algunas fueron procesadas en minutos mientras otros sistemas no terminaron después de horas. Además, los componentes base de nuestras plataformas también funcionaron bien y pueden usarse como sistemas autónomos.

Abstract

Markov Logic is a language for knowledge representation that combines logic and probabilities, providing a powerful framework for inference and learning tasks. *Inference* involves *grounding* and *search*. *Grounding* transforms logic representations into a set of weighted propositional clauses that encode a Markov network, the grounded Markov logic network (MLN). *Search* over the grounded MLN provides probabilistic *inference*. *Learning* can be used to find clause weights or structure, by applying inference repeatedly. As the solution space of MLNs can grow rather quickly, their processing can become very expensive, motivating research on their optimization. Multicore parallelism is already used in some well known MLN systems like Tuffy and RockIt, but we know of no other work based on bulk parallelism that Graphics Processing Units (GPUs) support.

This document presents the design, implementation and evaluation of parallel MLN platforms for GPUs. We have designed the parallelization of the most time consuming parts of the MLN process, describing the design issues encountered in both MLNs and GPUs, and our approaches to solving said issues. To evaluate our designs we integrated them into Tuffy and RockIt, as creating a whole system from scratch is beyond the scope of this work. Our contributions include: *grounding* processed in parallel using our logic programming system; *search* performed by partitioning and solving a satisfiability problem with our parallel algorithm or an integer linear programming problem; *parameter learning* through parallel *grounding* and solving an optimization problem, whose parameters are adjusted through our parallel sampling algorithm; and *structure learning* using Inductive Logic Programming, whose most compute intensive operation is also solved with our parallel logic programming system.

Our platforms were tested with applications from the literature and performed very well, with some applications being processed in minutes while other systems could not finish after several hours. Moreover, the core components of our platforms also performed well and can be used as stand-alone systems for general applications.

Acknowledgements

Analogous to my masters thesis, once again I would like to express my thanks to Dr. Jorge Buenabad Chávez for his valuable help, time, and guidance during the whole research work. Moreover, the new opportunities he provided me to travel allowed me to meet great people with wonderful ideas that surely improved this work.

I would also like to express my gratitude to Dr. Inês Dutra and Dr. Vítor Santos Costa from the University of Porto, Portugal, with whom I have been working for several years now. Their insightful comments and ideas led to many great papers and they have always treated me like family whenever I visit them.

Likewise, I am grateful of the suggestions and instruction provided by Dr. Ana María Martínez Enríquez during the writing of this document.

Additionally, I wish to acknowledge the perfect workplace and hardware provided by the Centre for Research & Postgraduate Studies of National Polytechnic Institute (Cinvestav-IPN) and the economic support provided by the Mexican Council for Science and Technology (Conacyt).

Finally, I wish to thank my parents for their support and encouragement throughout the years.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AMD. Advanced Micro Devices.

AI. Artificial Intelligence.

BP. Belief Propagation.

BUG. Bottom-up Grounding.

BUSL. Bottom-up Structure Learning.

CA. Computing Atoms.

CC. Computing Clauses.

CLL. Conditional Log-likelihood.

CPA. Cutting Planes Aggregation.

CPI. Cutting Planes Inference.

CSS-Tree. Cache Sensitive Search Tree.

CUDA. Compute Unified Device Architecture.

DN. Diagonal Newton.

ER. Entity Resolution.

FOL. First-Order Logic.

GA. Generic Algorithm.

GCC. GNU Compiler Collection.

GPU. Graphics Processing Unit.

IE. Information Extraction.

ILP. Inductive Logic Programming.

INLJ. Index Nested Loop Join.

KB. Knowledge Base.

LP. Link Prediction.

LBFGS. Limited-memory Broyden-Fletcher-Goldfarb-Shanno.

MaxSAT. Maximum Satisfiability.

MAP. Maximum a posteriori.

MB. Markov Blanket.

ML. Machine Learning.

MLN. Markov Logic Network.

MRC. Multiple Relational Clusterings.

MRF. Markov Random Field.

NLP. Natural Language Processing.

NR. Numeric Representation.

OpenCL. Open Computing Language.

PROLOG. PROgramming in LOGic.

PTX. Parallel Thread eXecution.

RA. Relational Algebra.

RC. Relational Classification.

RDBMS. Relational Database Management System.

SA. Simulated Annealing.

SAT. Satisfiability.

SIMD. Single Instruction, Multiple Data.

SM. Streaming Multiprocessor.

SQL. Structured Query Language.

SRL. Statistical Relational Learning.

TDSL. Top-down Structure Learning.

TPC-H. Transaction Processing Performance Council Benchmark H.

WPLL. Weighted Pseudolog-likelihood.

YAP. Yet Another Prolog.

Chapter 1

Introduction

This chapter provides a short explanation on Markov Logic Networks (MLNs), their advantages and the problems they currently face. We begin with the motivation and the statement of the problem that led to this thesis work. Then, we present our objectives and our contributions, followed by an overview of the following chapters.

1.1 Motivation

An MLN combines a first-order logic (FOL) language as relational representation and Markov networks as probabilistic representation. In practice, an MLN application is a Knowledge Base (KB), i.e., a database plus a set of FOL formulas with weights [24, 98], such that these weighted formulas establish *soft constraints* on the worlds (truth assignments to formulas): worlds that violate an MLN formula are less likely to exist, but still possible. In contrast, FOL formulas are hard constraints: a world that violates any formula is not possible.

Many of the shortcomings of logic like non-flexible constraints and inconsistencies when merging multiple KBs are solved in MLNs with the help of their probabilistic model. In addition, the probabilistic model benefits from the expressive power of FOL, as complex statistical problems can be represented with a simple, easy to understand syntax. MLNs form a simple but powerful language that is the basis of

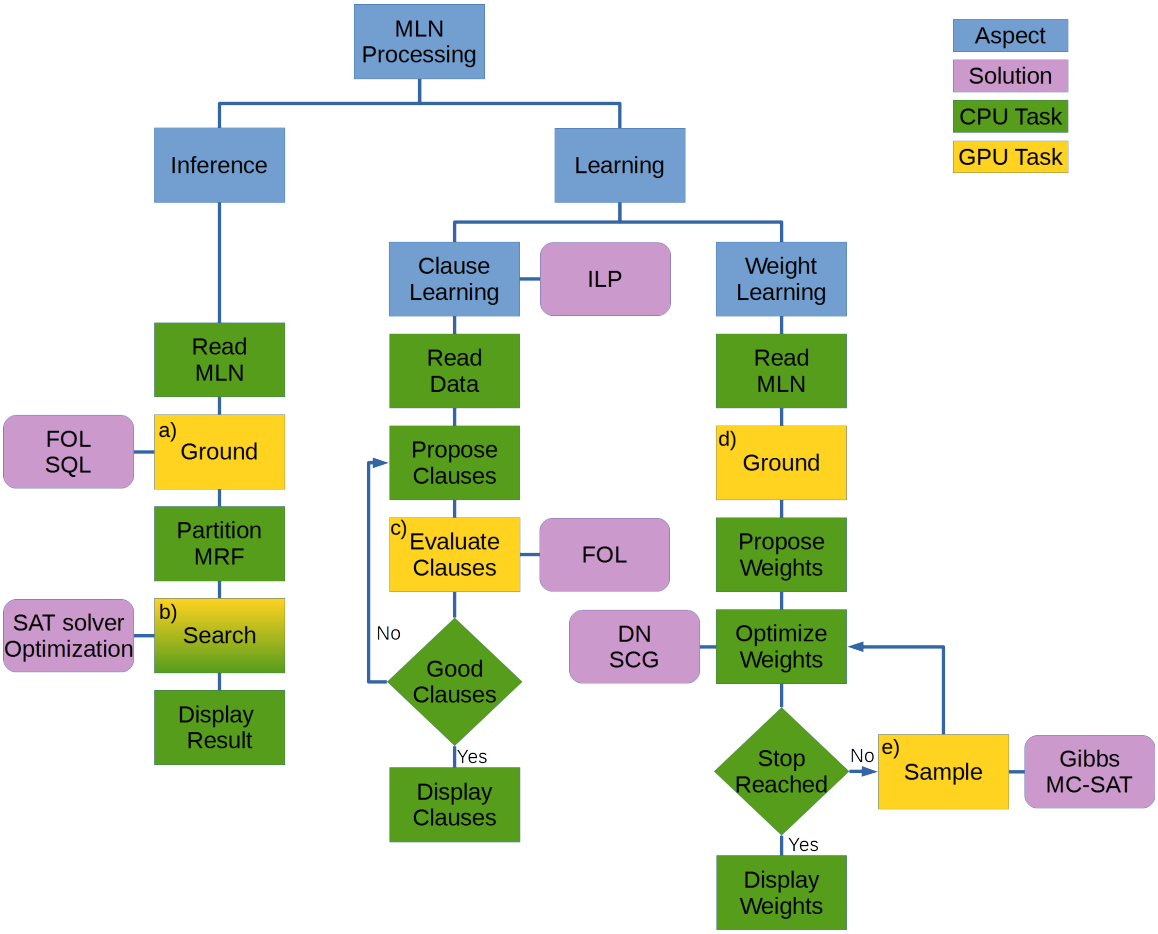


Figure 1.1: Phases of MLN processing and their steps with the most common solutions used.

various systems for classification [58], information extraction [59], Natural Language Processing (NLP) [100], among others.

As shown in Figure 1.1, MLN processing is divided in two independent phases: *inference* and *learning*. Inference in this thesis work (there are many kinds as described in Section 2.4.1) is called *Maximum a posteriori* and is the process of finding the most probable world of an MLN, requiring two compute intensive steps: a) *grounding* which involves assigning values to all variables in the formulas given some evidence data, by using a FOL system like Prolog or Datalog [116, 117], or a Relational Database Management System like MySQL or PostgreSQL; and, after partitioning the Markov Random Field [56] (MRF, an undirected graphical model also called Markov network) formed by the ground formulas, we proceed to b) *search* the MRF, which involves finding, for each ground formula, if said formula is true

or false by solving a weighted satisfiability problem (also called weighted MaxSAT) using algorithms like MaxWalkSAT [53] or by solving a mathematical optimization problem with Gurobi [149] or a similar solver. Note that the search is bicoloured because it can be performed by both CPU and GPU at the same time.

The weights of formulas can be learnt by d) *grounding* the formulas, and then solving an optimization problem (different from the one used in the search step) using iterative algorithms like Diagonal Newton, Scaled Conjugate Gradient, amongst others [24, p. 47]. The parameters of these algorithms are adapted at each iteration through e) *sampling*, using techniques like Gibbs sampling or MC-SAT [94]. Finally, formulas can also be learnt with several techniques like Inductive Logic Programming (ILP) [84] with some modifications on the way formulas are evaluated: a large number of candidate formulas are iteratively proposed and c) *evaluated* (in a process similar to the grounding) using some training data, stopping once good formulas are found.

1.2 Problem statement

As it can be seen, MLNs are a powerful and flexible framework with many applications. MLNs are the interface layer that artificial intelligence requires in order to easily bridge its basic tasks like inference and learning, with high-level applications like robotics, NLP, among others [24, p.13]. However, MLN processing is quite complex, with several components (e.g., algorithms and systems) working together. Moreover, some of these components have high worst-case time complexities like the grounders (Prolog, Datalog) which are EXPTIME-complete [17] (i.e., its worst-case scenario can be deterministically solved in exponential time), ILP which is NEXPTIME-complete [41] (i.e., same as EXPTIME-complete, but using stochastic algorithms) and MaxWalkSAT which is NP-Hard [12]. If even a single component delays, the whole MLN processing will suffer greatly, with even "small" MLNs (i.e., those with a small amount of evidence data and/or few formulas) taking hours or days to produce results, if any.

Several systems have tried, with varying degree of success, to efficiently compute MLNs by taking advantage of various techniques including multicore parallelism [88, 99, 90]. However, for many real world applications with evidence data not exceeding a few hundred MBs, the processing time still remains unacceptably high. Our proposed solution is to use Graphics Processing Units (GPUs) to process MLNs. GPUs are high-performance many-core processors capable of very high computation and data throughput [154]. GPUs are now used in a wide variety of applications [153], including gaming, data mining, bioinformatics, chemistry, finance, numerical analysis, imaging, weather, etc. Such applications are usually accelerated by at least an order of magnitude, but accelerations of 100x or more are common.

1.3 Objectives

The main objective of this thesis work is to design, implement and evaluate GPU-based platforms for MLN processing.

The specific objectives are:

- Create parallel algorithms for Datalog operators, grounding, satisfiability, sampling, and ILP.
- Describe our designs and approaches for MLNs on GPUs.
- Implement our approaches into two MLN platforms, using as basis existing MLN systems.
- Ground using our Datalog engine for GPUs, extended with additional operations.
- Assemble a GPU library to solve the satisfiability and sampling problems.
- Adapt an existing ILP system to handle parallel FOL clause learning.
- Test our platforms with several applications.

- Analyse our algorithms for performance improvements.

1.4 Contributions

Our contributions comprise GPU parallel algorithms and designs for the most time consuming phases and steps of MLN processing:

- *Grounding* using Datalog (a and d in Figure 1.1, and presented in Section 3.3).
- *Search* seen as a weighted MaxSAT problem (b, Section 3.4).
- *Weight learning* using Diagonal Newton with MC-SAT sampling (e, Section 3.5).
- *FOL clause learning* based on ILP (c, Section 3.5), which can be extended for MLN clause learning by changing the clause evaluation function.

These novel designs are important since they allow a better understanding of the parallel processing of MLNs, Datalog programs, satisfiability problems, and ILP. Moreover, they can be used to perform further, yet undiscovered efficiency optimizations or as the basis for other similar designs (e.g., the Datalog operations designs can be used in other FOL engines like Prolog, or the parallel MaxWalkSAT design can be used to parallelise other satisfiability solvers like ManySAT).

Our contributions also include several systems that have been thoroughly optimized and tested:

- *GPU-Datalog*, a parallel Datalog engine which can be used to compute general Datalog programs (and more thanks to its additional operators not part of the Datalog standard) with great speed-ups.
- A satisfiability and sampling library called GPUSATLIB, which includes the MaxWalkSAT and SampleSAT [119] algorithms for solving and sampling general MaxSAT problems.

- An ILP system based on GPU-Datalog and Aleph, which accelerates clause learning in FOL.
- The MLN platform called *GPU-Tuffy* which computes the inference problem as a weighted MaxSAT problem, using GPU-Datalog to perform the grounding and our GPUSATLIB library to solve the weighted MaxSAT problem. It can also compute clause weights in parallel using GPU-Datalog and GPUSATLIB.
- *GPU-RockIt*, which is another MLN platform that solves the inference problem as an optimization problem, using GPU-Datalog to perform the grounding.

We believe these contributions will significantly advance the state of the art of Datalog, ILP, and MLNs. Moreover, we propose in Chapter 6 an extensive future work that would further enhance the current state of the art. To the best of our knowledge, these contributions present the first extension of Datalog for GPUs (the original version is also our work [74]), the first use of Datalog for MLN grounding, the first MLN GPU-parallel grounding, the first GPU parallel *weighted* MaxWalkSAT algorithm, the first GPU-parallel satisfiability-based search on MLNs, the first GPU-parallel SampleSAT algorithm, the first GPU-parallel weight learning approach in MLNs, the first GPU-parallel ILP system, and the first GPU-parallel MLN platforms, GPU-Tuffy and GPU-RockIt.

1.5 Document Structure

This thesis is structured in six self-contained chapters. Next we present a small overview of the most important topic of each chapter.

1.5.1 Chapter 2. Background

Since MLNs are a complex framework whose processing combines several subjects, this chapter aims to familiarise the reader with all the necessary subjects, in order to fully understand MLNs by providing an overview of each subject and references to

further reading. The subjects covered include: *logic* with an emphasis on FOL and logic programming; *optimization* along with some of the most well-known algorithms for both constrained and unconstrained problems; the general MLN *framework* with a step-by-step example, a comparison with similar frameworks, and some interesting applications; all types of *inference* that can be applied to MLNs and the steps involved including *grounding*; and MLN weight and structure *learning* along with their algorithms that can be used to refine an existing MLN or create one from scratch.

1.5.2 Chapter 3. Parallel Processing of MLNs on GPUs

This chapter begins by presenting a brief introduction of the current MLN systems and explaining our choice of GPUs as the main hardware platform and their most relevant concepts. The chapter then continues with the design issues we faced with both GPUs and the MLN process including the GPU-CPU computation-to-communication ratio, the limited amount of GPU memory, and the serial nature of the MLN process. Finally, we describe our parallel approach and the state of the art for each MLN task: *grounding* based on FOL and Datalog; *search* based on satisfiability with the MaxWalkSAT algorithm in one proposal and based on integer linear programming on the other; and *learning* based on parallel grounding and optimization with sampling using MC-SAT for weight learning, and based on ILP with a modified evaluation function for clause learning.

1.5.3 Chapter 4. Experimental Platform

This chapter describes how we solved the design issues presented in Chapter 3 and the integration of our proposed approaches into functional systems. We begin by introducing the GPU parallel, core components of our MLN platforms: a Datalog engine with several new operations and optimizations used for grounding; a satisfiability solver based on the MaxWalkSAT algorithm used during the search step;

and a sampler of the MLN search space used as part of weight learning. Next, we show the integration of these components into two existing MLN systems, generating our MLN platforms called GPU-Tuffy and GPU-RockIt. Finally, we test the validity of our platforms and their components by executing several applications and comparing our results with those of other systems.

1.5.4 Chapter 5. Performance Evaluation

The results of testing our MLN platforms and their core components are discussed in this chapter. First, we present the evaluation of GPU-Datalog alone (the core component of the grounding) followed by the evaluation of our parallel ILP system using FOL clause learning. Next, we evaluate the general performance of our platforms with six applications from the literature. Finally, specific parts of our platforms were evaluated including the parallel search of GPU-Tuffy, the parallel grounding on our GPU-RockIt platform and our weight learning based on parallel sampling. Each evaluation presents the followed methodology, the applications and hardware used, and the obtained results along with their discussion. After all evaluations, the chapter ends with some further discussion of the performance of our MLN platforms and their GPU kernels.

1.5.5 Chapter 6. Conclusions and Future Work

In this final chapter we conclude by showing a positive outlook on our current designs, their implementation, and their evaluation. We also explain our contributions and possible future work to improve MLNs in general, the core components of our platforms (GPU-Datalog and GPUSATLIB), our parallel ILP system, and our MLN platforms themselves (GPU-Tuffy and GPU-RockIt).

Chapter 2

Background

Markov Logic Networks (MLNs) belong to the branch of Artificial Intelligence (AI) called Machine Learning (ML), but their processing is a complex task that combines several subjects and thus, this chapter is dedicated to all the background knowledge necessary for the full understanding of MLNs and our work to improve them. The reviewed subjects include: 2.1 *logic* which is the basis of the grounding step; 2.2 *optimization* used during the search step and during weight learning; 2.3 an overview of the most important concepts of *MLNs*; 2.4 *inference* described as both a satisfiability problem and an optimization problem; and 2.5 *learning* of weights and clauses.

A summary of the subjects covered in this chapter is presented in Figure 2.1. MLNs processing is divided in two *tasks*, which can be further divided into *critical steps* that can affect the performance and results of the MLN depending on the chosen *solution* and its *algorithms* or *systems*. For each subject, the following sections provide an explanation of the most relevant concepts and numerous references to further readings. Note that background knowledge for particular MLN systems is discussed in Chapter 4 and Graphics Processing Units are described in Appendix A.

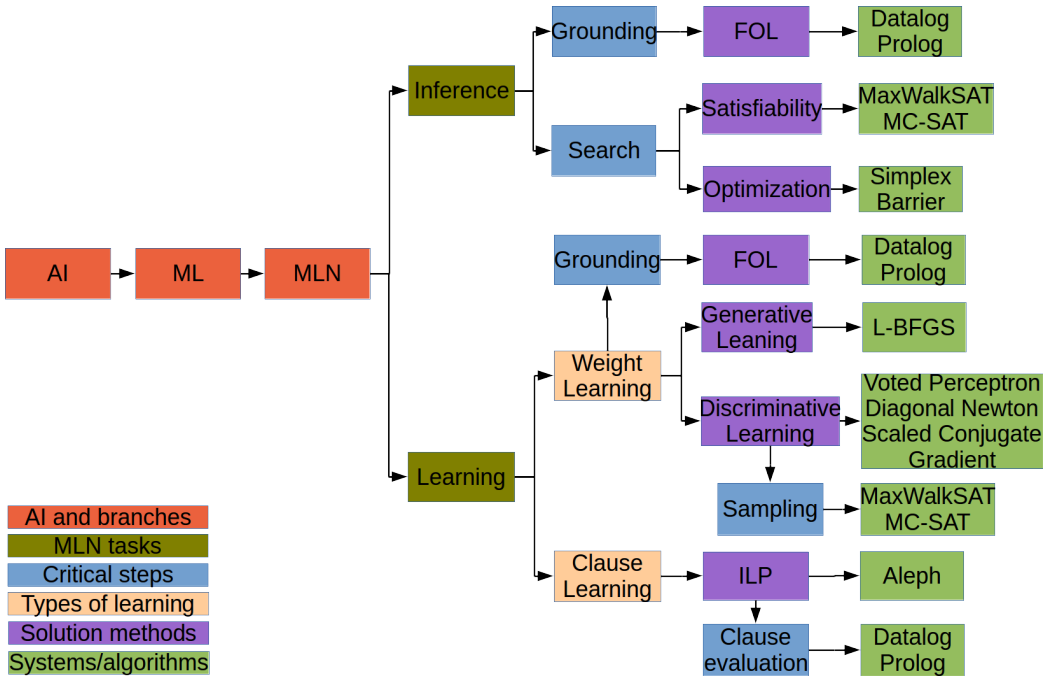


Figure 2.1: Subjects covered in this Chapter.

2.1 Logic

Logic is the analysis of the reasoning methods [78]. In Computer Science, logic is tightly related to the field of AI, where First-Order Logic (FOL) is used to formally represent knowledge and logic programming serves as the programming paradigm.

2.1.1 First-Order Logic

Propositional logic studies propositions, their interaction with other propositions by logical connectives (AND, OR, XOR, etc.), and their truth values (true or false). While useful for many tasks like electronic circuits representation and system requirements specifications, higher forms of reasoning including quantified expressions and expression with variables, are difficult or impossible to represent in this logic (e.g., to represent that all people are tall, we most write one proposition per person). Propositional logic can be extended to First-Order Logic [78], which includes a richer language based on an *alphabet* and *well-formed* formulas. The most relevant FOL definitions taken from [67] are:

Definition 1. *Alphabet.* An alphabet consists of variables (usually upper case letters), constants (lower case letters), function symbols (the letter `f` or Greek letters), predicate symbols (whole words like `father`), connectives, quantifiers and punctuation symbols. The connectives are negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$) and biconditional ($\Longleftrightarrow$). The punctuation symbols are "." and ",".

Definition 2. *Quantifier.* A quantifier is a construct that specifies the quantity of individuals in a domain. In FOL, the only quantifiers used are the universal quantifier ($\forall$) and the existential quantifier ($\exists$). As an example, the sentence "for all people X, there exists some person Y that is taller" can be written as $\forall X \exists Y taller(X, Y)$.

Definition 3. *Term.* A term is defined inductively as follows:

- A variable is a term.
- A constant is a term.
- If f is an n-ary function symbol and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

Definition 4. *Predicate.* A predicate is a boolean function that defines a relationship between its arguments. For example, $P(4)$ will be true if 4 belongs to the set P and $father(X, Y)$ will be true for those Xs who have Y as father.

Definition 5. *Atom.* If P is an n-ary predicate symbol and $t_1, ..., t_n$ are terms, then $P(t_1, ..., t_n)$ is a formula called atomic formula or, more simply, an atom.

Definition 6. *Well-defined formula.* It is inductively defined as:

- An atom is a formula.
- If F and G are formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$, and $F \Longleftrightarrow G$ are formulas.
- If F is a formula and x is a variable, then $\forall x F$ and $\exists x F$ are formulas.

Definition 7. *Literal.* A literal is an atom or the negation of an atom. A positive literal $L(t_1, ..., t_n)$ is an atom. A negative literal $\neg L(t_1, ..., t_n)$ is the negation of the atom $L(t_1, ..., t_n)$.

Definition 8. *Clause.* A clause is a formula of the form

$$\forall x_1, ..., \forall x_s(L_1 \vee, ..., \vee L_m) \tag{2.1}$$

where each L_i is a literal and $x_1, ..., x_s$ are all the variables occurring in $L_1 \vee, ..., \vee L_m$.

Definition 9. *Ground term.* A ground term is a term containing no variables. Similarly a *ground atom* is an atom containing no variables.

Definition 10. *Ground clause.* A clause is called ground whenever each of its members is a ground term.

Note that FOL differs from second-order or other higher-order logics in that quantifiers can only be defined over variables and predicates cannot include other predicates as arguments.

2.1.2 Logic Programming

As presented by Kowalski in [61], FOL has a procedural interpretation based on *unification* that can be seen as a programming language. This revolutionary concept is the basis of logic programming and was first implemented in the Prolog (PROgramming in LOGic) system [13].

Perhaps the most important aspect of logic programming is the separation between *logic* and *control* in an algorithm [67]. Logic specifies what the problem is, while control specifies how to solve the problem. Separating these aspects brings many benefits like the possibility of specifying only the logic, leaving the control to the logic programming system.

There are several variants and extension to logic programming including Constrained Logic Programming [31], Inductive Logic Programming (ILP) [84],

among others. Also, there are various systems (besides Prolog) for logic programming and their variants like Datalog [116, 117] and Progol [85].

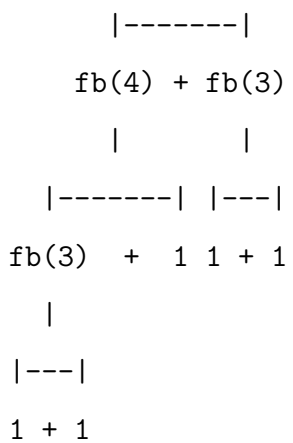
As an example of FOL and logic programming, consider the following Prolog program to compute the Fibonacci sequence (1, 1, 2, 3, 5, 8, 13, 21, 34, ...):

```
1 fib(1,1).
2 fib(2,1).
3 fib(N,R):- N > 2,
  N1 is N-1, fib(N1,R1),
  N2 is N-2, fib(N2,R2),
  R is R1+R2.
```

where the first two formulas are clauses with a single predicate and the last one is also a clause but with several predicates. Note that, unless otherwise specified, in both FOL and Prolog, variables are considered to be universally quantified and thus the $\forall$ symbol is omitted. In conjunction with a query say `?-fib(5,R).` (i.e., which is the 6th number of the Fibonacci sequence?), Prolog will try to answer the query using the most appropriate clause (3), processing each predicate in the clause from left to right: as `N = 5` then it is greater than 2 (`N > 2`); `N1` is set to `N-1` (4); the process continues with `fib(5,R1)` which once again requires the processing of the third clause now with `N = 4`, then `N = 3` and so forth until we reach `N = 2` which causes the predicate `N > 2` to be false and the clause to fail, with the second clause coming into effect instead and providing the first answer for `R` (1). The process would then continue with the next predicates of each instance of the third clause, until we have an answer for `R` by adding `R1+R2`.

As it can be seen, the third clause is recursive thanks to its third and fifth predicates, a common occurrence in many logic programs, and the recursion that has to follow to answer our query is:

```
fb(5)
  |
```



2.1.3 Summary

Logic is the basis of many ML application including MLNs. The type of logic used by MLNs is FOL, which is based on well-formed formulas constructed with an alphabet of variables, predicates, quantifiers, among others (e.g., A = [an overcast sky] implies that B = [the sun is not visible], is represented with the logical implication symbol $\Rightarrow$ as $A \Rightarrow B$). FOL has an interpretation that serves as a programming language and is the basis of logic programming. One of the most important logic programming systems is Prolog, whose syntax is very similar to that of FOL (e.g., "All humans are mammals" is represented as `mammal(X) :- human(X).`). The application of FOL for grounding in MLNs is presented in Sections 3.3, 4.1, and 4.4.

2.2 Optimization

Optimization is a branch of mathematics used in analysis and decision making [89]. It has a wide array of applications like maximizing profits in finance, minimizing costs in an industrial process, etc. The optimization process begins by constructing a model with the objective to minimize or maximize (e.g., cost, profit) that depends on certain variables (e.g., time, resources), which can be constrained (e.g., time cannot be negative). More formally, the optimization problem is written as:

$$\begin{gathered} min\, f(x) \\ \text{subject to} \\ c_i(x) = 0, i \in I \\ c_j(x) \geq 0, j \in J \end{gathered} \tag{2.2}$$

where f is the objective function, x is the vector of variables, I is the set of equality constraints (c_i) that must be satisfied and J is the set of inequality constraints. Note that most of the optimization literature assumes a minimization problem (hence the *min* in the equation) and that a maximization problem can be seen as a minimization one by using $-f(x)$.

Once the model is constructed, an optimization algorithm can be used to find its solution. However, there are several optimization algorithms, there is no universal optimization algorithm for all problems, and depending on the problem, certain algorithms are better than others. This rich variety of optimization problems and algorithms has led to the creation of several branches in the optimization field, which are usually mutually exclusive (e.g., a problem cannot be both constrained and unconstrained). The most important cases include:

- **Discrete versus continuous optimization.** Discrete optimization refers to those problems whose domain is restricted to a finite set of elements. Usually, said set is a set of integers and the problem is called an *integer linear programming* problem [93]. In contrast, continuous optimization normally deals with infinite sets of real numbers. For example, the problem $min\, x^2$ is discrete when $x \in Z$ and continuous when $x \in R$.

- **Unconstrained and constrained optimization.** When the sets I and J are empty, the problem is said to be unconstrained. This situation arises in many practical applications where there are no natural constraints to the variables or they can be disregarded. Constrained problems always include constraints

which, depending on the formula that represents them, can be simple bounds on the variables or complex relations between several variables. As an example, $min\, x^2$ is unconstrained, but can be constrained by adding that it is subject to $x > 1$.

- **Local and global optimization.** Many optimization problems have several local solutions, i.e., points where the value of the objective function is smaller compared to all other points in the neighbourhood. The best (smallest) of these local solutions is called the global solution. However, identifying and locating the global solution can be a very difficult and time consuming task. Thus, global optimization is usually reserved for small, simple problems and local optimization for large, complex problems. For example, $min\, 2x^4 - x^3 - 3x^2 + x$ has a local minimum that most algorithms would find at 1, but it is possible that only the global algorithms would find the global minimum at ~ -0.784.

- **Stochastic and deterministic optimization.** Sometimes the model of a problem cannot be completely specified since some values are missing or unknown. However, such values can be predicted or estimated using techniques from stochastic optimization. On the other hand, in deterministic optimization the model is always fully defined. For instance, $min\, x^2$ subject to $x > a$ is stochastic if we can only estimate values for a and deterministic if we know its exact value.

Finally, a fundamental concept in optimization is the *convexity* of the problem. A set $S \in R^n$ is said to be convex if the straight line connecting any two points in S is also inside S. Formally, using the line equation, S is convex if for any points $x, y \in S$, the result of $\alpha x + (1 - \alpha)y$ for all $\alpha \in [0, 1]$ is in S. Furthermore, f is a convex function if its domain is a convex set and if for any two points x, y in the domain, the following inequality is preserved:

$$f(\alpha x + (1-\alpha)y) \leq \alpha f(x) + (1-\alpha)f(y), \quad \text{for all} \quad \alpha \in [0,1] \tag{2.3}$$

Convex objective functions have an interesting property that is very helpful in optimization: if the optimization algorithm reaches a local minimum, then we have also reached the global minimum. In this thesis work, all the objective functions of our optimization problems are convex.

2.2.1 Optimization Methods for MLN learning

Next we present a general overview of the optimization methods and their algorithms used during weight learning in MLNs (see Section 3.5 for details of the implementation). The weight learning problem is continuous and unconstrained, and all of the following algorithms perform local deterministic optimization. The descriptions of these methods were mainly taken from [24, 89].

Line Search (Voted Perceptron)

Line search iterative algorithms usually begin from a random point (x_0) in the search space and move over this space finding new, hopefully better points ($x_{0,\dots,n}$). To move around said search space, a *direction* (ρ) and a *step size* (α) are required at each iteration (k). Formally, this movement is defined by the following equation:

$$x_{k+1} = x_k + \alpha_k \rho_k \tag{2.4}$$

The movement continues until a stopping condition is reached. Common stopping conditions include reaching a certain number of iterations or finding a good approximation of the solution, which usually occurs when the change between x_k and x_{k+1} is bellow a certain user-defined threshold (e.g., if the threshold is 0.01 and

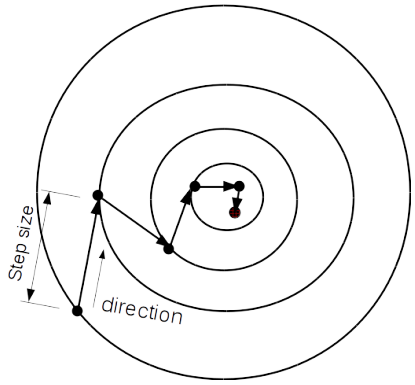


Figure 2.2: Example of the line search method for a certain function represented with contour lines.

$x_k = 1.333, x_{k+1} = 1.332$, then the algorithm finishes with x_{k+1} as solution).

In particular, the Voted Perceptron algorithm follows the direction called the *steepest-descent*, in which the objective function decreases most rapidly and is given as the negative of the derivative of the objective function in x_k (also called *gradient*). The step size is called *learning rate* and is initially defined by the user and then adapted by the algorithm.

Figure 2.2 shows an example of how the line search method would move around a certain function represented with *contour lines* (i.e., curves that represent points of constant value along the function), until it reaches the solution.

Newton's Method (Diagonal Newton)

Newton's method belongs to the family of line search methods, but it usually has a higher convergence rate (arrives at the solution in fewer iterations). In said method, the descent direction (ρ_k) is computed as:

$$\rho_k = - \bigtriangledown^2 f_k^{-1} \bigtriangledown f_k \tag{2.5}$$

where $- \bigtriangledown^2 f_k^{-1}$ is the inverse of the second derivative (also called Hessian) at point x_k and $\bigtriangledown f_k$ is the gradient at x_k. Computing and inverting the Hessian is usually a time consuming, error prone task. Fortunately, the Diagonal Newton (DN) algorithm

simplifies the Hessian by assuming that all off-diagonal entries are zero, yielding an easy to invert diagonal matrix. The step size of the DN algorithm is defined as:

$$\alpha_k = \frac{-\rho_k^T \bigtriangledown f_k}{\rho_k^T \bigtriangledown^2 f_k \rho_k + \lambda_k \rho_k^T \rho_k} \tag{2.6}$$

where λ_k limits the step size into a good search region and is the ratio between the actual change ($\triangle_a$) of the objective function value in an iteration and the predicted change ($\triangle_p$), which are computed as follows:

$$\begin{aligned} \triangle_a &= x_k - x_{k-1} \\ \triangle_p &= \rho_{k-1}^T \bigtriangledown f_{k-1} + \frac{1}{2\rho_{k-1}^T \bigtriangledown^2 f_{k-1} \rho_{k-1}} \end{aligned} \tag{2.7}$$

and then, depending on the ratio, λ_k is increased or decreased following a common method presented in [27]:

$$\text{if}(\triangle_a/\triangle_p > 0.75) \quad \text{then} \quad \lambda_{k+1} = \lambda_k/2 \tag{2.8}$$

$$\text{if}(\triangle_a/\triangle_p < 0.25) \quad \text{then} \quad \lambda_{k+1} = 4\lambda_k \tag{2.9}$$

Conjugate Gradient (Scaled Conjugate Gradient)

A problem with line search methods is that they often take several steps in the same direction. The conjugate gradient method attempts to solve this problem by computing the search direction and step size that warranty the best movement in said direction, ideally solving a problem in R^n using n steps.

In particular, MLNs use the Scaled Conjugate Gradient [82] algorithm that iteratively moves through the search space using Equation 2.4 and computes the step size (α) with Equation 2.6. The search direction (ρ) is computed using the

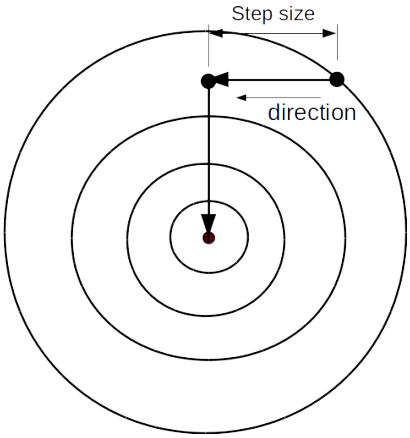


Figure 2.3: Example of the conjugate gradient method on a quadratic function.

Polak-Ribiere method which, given the objective function f, is formally defined as:

$$\begin{aligned} \rho_0 &= - \bigtriangledown f_0 \\ \beta_{k+1} &= \frac{\bigtriangledown f_{k+1}^T (\bigtriangledown f_{k+1} - \bigtriangledown f_k)}{\| \bigtriangledown f_k \|^2} \\ \rho_{k+1} &= - \bigtriangledown f_{k+1} + \beta_{k+1} \rho_k \end{aligned} \tag{2.10}$$

where $\bigtriangledown f_k$ is the gradient of f at the point x_k. An example of this method for a quadratic function is presented in Figure 2.3. The ideal step size is computed along the search directions and the result is found in two steps (since the problem is in R^2).

Quasi-Newton Method (L-BFGS)

Quasi-Newton methods have a faster convergence than the line search methods without having to compute second derivatives (a costly operation in Newton's method). The most popular Quasi-Newton method is called BFGS after its creators: Broyden-Fletcher-Goldfarb-Shanno.

Once again, Equation 2.4 is used in the BFGS algorithm with its own step size and search direction. For MLNs, a variant of the algorithm called Limited-memory BFGS (L-BFGS) [66] computes the search direction (ρ) using an approximation of the Hessian function based on the difference between solution points (s), between

their gradients (y), and other values (w, V, and A) in two consecutive iterations:

$$s_k = x_{k+1} - x_k \tag{2.11}$$

$$y_k = \bigtriangledown f_{k+1} - \bigtriangledown f_k \tag{2.12}$$

$$w_k = \frac{1}{y_k^T s_k} \tag{2.13}$$

$$V_k = I - w_k y_k s_k^T \tag{2.14}$$

$$A_0 = I \tag{2.15}$$

$$A_k = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} I \tag{2.16}$$

All the values of s and y from the first iteration (0) until the current iteration (k) are necessary to compute the approximation of the Hessian function (H) for the current iteration. However, as the algorithm progresses, storing all said values becomes impractical in terms of memory. Fortunately, good values for H can be obtained by storing only the last m values of s and y, where m is typically between 3 and 20. Formally, H is then defined as follows:

$$H_0 = A_0 \tag{2.17}$$

$$\begin{aligned} H_k = {} & (V_{k-1}^T \dots V_{k-m}^T) A_k (V_{k-m} \dots V_{k-1}) \\ & + w_{k-m} (V_{k-1}^T \dots V_{k-m+1}^T) s_{k-m} s_{k-m}^T (V_{k-m+1} \dots V_{k-1}) \\ & + w_{k-m+1} (V_{k-1}^T \dots V_{k-m+2}^T) s_{k-m+1} s_{k-m+1}^T (V_{k-m+2} \dots V_{k-1}) \\ & + \dots \\ & + w_{k-1} s_{k-1} s_{k-1}^T \end{aligned} \tag{2.18}$$

Note that values with negative indexes (like w_{k-m} where $k = 1$ and $m = 3$) are zero. Once obtained, the value of H can then be used to compute the search direction as:

$$\rho_k = -H_k \bigtriangledown f_k \tag{2.19}$$

Finally, the step size is initially chosen as $\alpha = 1$, but is adapted at each iteration if it violates the Wolfe conditions [121]:

$$f(x_k + \alpha_k \rho_k) \leq f(x_k) + c_1 \alpha_k \bigtriangledown f_k^T \rho_k \tag{2.20}$$

$$\bigtriangledown f(x_k + \alpha_k \rho_k)^T \rho_k \geq c_2 \bigtriangledown f_k^T \rho_k \tag{2.21}$$

where c_1 and c_2 are usually 10^{-4} and 0.9 respectively, since they provide good values for the step size.

2.2.2 Optimization Methods for MLN inference

The MLN inference problem can be seen as an integer linear programming problem, i.e., an optimization problem whose domain is the set of integer numbers and has a linear objective function with linear constraints (the implementation can be found in Section 4.5). Thus, the inference problem in MLNs is discrete and constrained, and all the solvers presented next perform local deterministic optimization. A more detailed description of these methods can also be found in [89].

Simplex method

In the simplex method, the optimization problem (Equation 2.2) is usually written in standard form:

$$min\, c^T x, \quad \text{subject to} \quad Ax \leq b, x \geq 0 \tag{2.22}$$

where x is a vector with the variables of the problem, c is a vector with the coefficients

of the objective function, A is a matrix of constants, and b is a vector of constants. Since inequalities are difficult to handle, this form can be simplified by transforming them into equalities by eliminating variables or introducing *slack* variables. For example, $x_1 \geq 3$ can be written as $y_1 = x_1 - 3$, thus eliminating x_1 and $2x_2 + 5x_3 \leq 7$ can be written as $2x_2 + 5x_3 + s_1 = 7$.

Due to the constrains on the problem, the feasible region forms a convex *polytope* where the global minimum is usually found in one of its vertices. Thus, the simplex method starts from a vertex and iteratively moves around adjacent vertices until the solution vertex is found. The method can determine if a solution has been found thanks to the Karush-Kuhn-Tucker conditions, which state that x is a solution if there exist vectors ϕ and s such that:

$$A^T\phi + s = c \quad (s \geq 0) \tag{2.23}$$

$$Ax = b \quad (x \geq 0) \tag{2.24}$$

$$x^T s = 0 \tag{2.25}$$

Finding the starting vertex is a non-trivial task that is usually solved by defining a simpler optimization problem whose result is a vertex in the original problem. Also, at each iteration, the decision of which vertex the method should move next can be performed with several heuristics like estimating the maximum advance the new vertex would provide and choosing the best. Figure 2.4 shows an example of a simple polytope created by the constraints of a problem and how the simplex method would move along these constraints until the optimum is reached.

Barrier method

The barrier method belongs to the family of interior-point methods (i.e., those methods that must strictly satisfy all constraints at each iteration) and seeks the solution of the optimization problem by replacing it with a series of unconstrained,

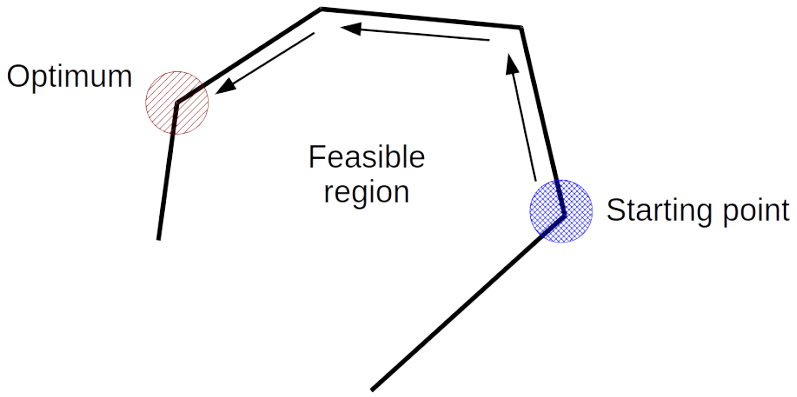


Figure 2.4: Example of a polytope created by the constraints of an optimization problem and the path followed by the simplex method to find the solution.

smaller problems. These smaller problems have a modified objective function called barrier function, that is infinite everywhere except on the feasible region, is smooth inside the feasible region and approaches infinity as one approaches the boundaries of the feasible region. The most popular barrier function is based on the natural logarithm and is defined as follows:

$$P(x;\mu) = f(x) - \mu \sum_{i \in I} ln\, c_i(x) \tag{2.26}$$

where $f(x)$ is the objective function of the original problem, μ is the barrier parameter, and $c_i(x)$ is the ith inequality constraint of the original problem.

With a starting point x_0^s, an initial value for the barrier parameter μ_0, and a certain tolerance τ, the barrier method begins its iterative process by minimizing Equation 2.26 with an unconstrained optimizer like Newton's method, until $\| \bigtriangledown P(x_k;\mu_k) \| \leq \tau$. Next, a smaller value for μ and a new starting point x^s are chosen and a new iteration begins. The process stops once we have a solution for $P(x;\mu)$ for a sufficiently small μ, as said solution is also a solution to the original problem [122]. The starting point at each iteration x_k^s can be computed by extrapolating the minimizers of the previous iteration $(x_{k-1}, ..., x_0)$ and the initial value μ_0 is usually ambitious (around 0.1 or 0.2). Afterwards, how much should μ_0

be reduced at each iteration will depend on the problem.

2.2.3 Summary

The optimization process begins by constructing a mathematical model of the problem based on an objective function subject to certain constraints. Then, the optimum (usually the minimum) of said function can be searched using a rich variety of algorithms. For MLN learning (a continuous unconstrained problem) the optimization algorithms (line search, Newton's method, conjugate gradient, and quasi-Newton's method) are based on moving from a certain starting point in the search space to the solution by iteratively taking steps in *descent* directions (i.e., those who reduce the value of the objective function). How to determine a descent direction and the size of the step in that direction are the fundamental differences between the algorithms. Their application in our work is discussed in Section 3.5. For MLN inference (a discrete constrained problem) there are two approaches: the simplex method that moves along the constraints until it reaches the solution (which is usually found in the vertex formed by two constraints); and the barrier method that transforms the problem into an unconstrained one and then solves it. These optimization methods are used in the MLN system RockIt as shown in Section 4.5.

2.3 Markov Logic Networks (MLNs)

Statistical Relational Learning (SRL) [36] is a field of ML that combines logic and probability to manage the uncertainty arising from noise and incomplete information which is typical of real-world ML applications. SRL has been successfully used in a variety of tasks including: collective classification (e.g., given a set of emails, determine which are spam), link prediction (e.g., given a social network, infer new, possible user interactions), link-based clustering (e.g., clustering object of multiple types like web pages with search queries and users, in order to offer better search suggestions), social network modelling (e.g., infer unobserved friendships in

Facebook), and entity resolution (e.g., determine whether an object on different photos is the same).

Various SRL frameworks have been proposed including Stochastic Logic Programs [16], Probabilistic Relational Models [30], PRISM [104], Bayesian Logic Programs [54], ProbLog [55], Constrained Logic Programming [31], Predicate Functor Logic [63], and MLNs [24]. MLNs are the main focus of this thesis, as they offer a simple and highly expressive language along with robust inference.

An MLN is a Knowledge Base (KB) composed of evidence and FOL formulas with a weight attached to each formula. FOL provides a rich and compact representation of the problem, while the weights provide probabilistic inference whose solutions cannot be found by the strict, deterministic inference typical of FOL alone. A detailed example of an MLN and its inference process is described next.

Markov Logic Networks Overview

In MLNs, FOL formulas are used to define objects/entities, their attributes and relationships among them. For example, the well-known *smokers* application taken from [24] (which will be used as running example through this thesis), determines the probability of people having cancer (`Ca`) based on who their friends (`Fr`) are and whether or not their friends smoke (`Sm`):

$$1.5 : \neg \texttt{Sm}(\texttt{x}) \vee \texttt{Ca}(\texttt{x})$$

$$1.7 : \neg \texttt{Fr}(\texttt{x}, \texttt{y}) \vee \neg \texttt{Sm}(\texttt{y}) \vee \texttt{Sm}(\texttt{x})$$

$$1.9 : \neg \texttt{Fr}(\texttt{x}, \texttt{y}) \vee \neg \texttt{Sm}(\texttt{x}) \vee \texttt{Sm}(\texttt{y})$$

where the weights of the formulas (1.5, 1.7, 1.9) transform the FOL program into a probabilistic model.

Given some evidence E (facts in logic, records in DBs), MLNs typically apply the structure and weights of their formulas to generate the *most probable world*, which is

the one that has the *lowest* solution cost, determined by the sum of the weights of the unsatisfied ground formulas. For instance, if E contains four facts, $\texttt{Fr}(\texttt{Frank}, \texttt{Bob})$, $\texttt{Fr}(\texttt{Gary}, \texttt{Bob})$, $\texttt{Sm}(\texttt{Frank})$ and $\neg\texttt{Sm}(\texttt{Gary})$, then we begin our search for the most probable world by *grounding* the formulas, i.e., assigning the values of the facts to the variables in the formulas:

$$1.5 : \neg\texttt{Sm}(\texttt{Bob}) \vee \texttt{Ca}(\texttt{Bob}) \tag{2.27}$$

$$1.5 : \neg\texttt{Sm}(\texttt{Frank}) \vee \texttt{Ca}(\texttt{Frank}) \tag{2.28}$$

$$1.5 : \neg\texttt{Sm}(\texttt{Gary}) \vee \texttt{Ca}(\texttt{Gary}) \tag{2.29}$$

$$1.7 : \neg\texttt{Fr}(\texttt{Gary}, \texttt{Bob}) \vee \neg\texttt{Sm}(\texttt{Bob}) \vee \texttt{Sm}(\texttt{Gary}) \tag{2.30}$$

$$1.7 : \neg\texttt{Fr}(\texttt{Gary}, \texttt{Bob}) \vee \neg\texttt{Sm}(\texttt{Gary}) \vee \texttt{Sm}(\texttt{Bob}) \tag{2.31}$$

$$1.9 : \neg\texttt{Fr}(\texttt{Frank}, \texttt{Bob}) \vee \neg\texttt{Sm}(\texttt{Bob}) \vee \texttt{Sm}(\texttt{Frank}) \tag{2.32}$$

$$1.9 : \neg\texttt{Fr}(\texttt{Frank}, \texttt{Bob}) \vee \neg\texttt{Sm}(\texttt{Frank}) \vee \texttt{Sm}(\texttt{Bob}) \tag{2.33}$$

Note that formulas are repeated as many times as necessary to generate all possible combinations. Once all formulas are grounded, truth values are assigned to each literal of each formula, starting with those which are true or false based on the evidence. These initial truth values may already *satisfy* some formulas (formulas with positive weights are satisfied when at least one literal is true, those with negative weights when all literals are false):

$$1.5 : \underline{\neg\texttt{Sm}(\texttt{Frank})} = \mathit{false} \vee \texttt{Ca}(\texttt{Frank}) = \mathit{unknown} \tag{2.28}$$

$$1.5 : \underline{\neg\texttt{Sm}(\texttt{Gary})} = \mathit{true} \vee \texttt{Ca}(\texttt{Gary}) = \mathit{unknown} \tag{2.29}$$

$$1.7 : \underline{\neg\texttt{Fr}(\texttt{Gary}, \texttt{Bob})} = \mathit{false} \vee \underline{\neg\texttt{Sm}(\texttt{Gary})} = \mathit{true} \vee \texttt{Sm}(\texttt{Bob}) = \mathit{unknown} \tag{2.31}$$

$$1.9 : \underline{\neg\texttt{Fr}(\texttt{Frank}, \texttt{Bob})} = \mathit{false} \vee \neg\texttt{Sm}(\texttt{Bob}) = \mathit{unknown} \vee \underline{\texttt{Sm}(\texttt{Frank})} = \mathit{true} \tag{2.32}$$

where the underlined literals are those whose truth values were obtained from the evidence and *unknown* means that we cannot determine the truth value of the literal based on that formula.

Unknown values, however, present a *conflict*: no matter which truth value we choose for the literals, some formulas may remain unsatisfied. In the example, if we set $\texttt{Sm}(\texttt{Bob}) = false$, then formula 2.33 cannot be satisfied; if we set $\texttt{Sm}(\texttt{Bob}) = true$, then formula 2.30 cannot be satisfied. This problem is resolved using the weights of each formula, as they reflect our confidence on the formula holding true when faced with conflicting formulas or evidence: formulas with higher weights will be *satisfied* at the expense of lower weighted formulas being ignored. Thus, we set $\texttt{Sm}(\texttt{Bob}) = true$ (since the weight of formula 2.33 is greater than that of formula 2.30) and obtain:

$$1.5 : \neg\texttt{Sm}(\texttt{Bob}) = false \vee \texttt{Ca}(\texttt{Bob}) = true \tag{2.27}$$

$$1.7 : \neg\texttt{Fr}(\texttt{Gary}, \texttt{Bob}) = false \vee \neg\texttt{Sm}(\texttt{Bob}) = false \vee \texttt{Sm}(\texttt{Gary}) = false \tag{2.30}$$

$$1.9 : \neg\texttt{Fr}(\texttt{Frank}, \texttt{Bob}) = false \vee \neg\texttt{Sm}(\texttt{Frank}) = false \vee \texttt{Sm}(\texttt{Bob}) = true \tag{2.33}$$

Finally, with all truth values assigned, we have the most probable world (whose solution cost is equal to the weight of the unsatisfied formula, i.e., 1.5): `Frank` has cancer since he smokes; `Bob` has cancer since he is `Frank`'s friend; and `Gary` does not have cancer since he does not smoke. Note that it is also possible to define *negative* weights, which mean that the formula should not hold (i.e., it should not be satisfied). Furthermore, weights can be *infinite* (in practice it is just a high value) and those formulas with infinite weights are called *hard* and should always be satisfied (or not satisfied in case of $-\infty$), as they are not probabilistic, they are a certainty like normal FOL formulas.

As illustrated by the smokers example, with MLNs it is possible to transform an existing FOL program into a probabilistic model by assigning weights to each formula. Many of the shortcomings of logic like non-flexible constraints and inconsistencies

when merging multiple KBs are solved with the help of probabilistic models. It also allows data and formulas from multiple programs to be used together without resolving their inconsistencies (e.g., when several related domains are used together to increase the available knowledge). On the other hand, probabilistic models benefit from the expressive power of FOL, as complex statistical models can be represented with a simple and easy to understand syntax.

MLNs have surpassed all other frameworks thanks to their ability to combine soft and hard constraints, since few systems are capable of processing both at the same time. They also handle a greater degree of uncertainty (i.e., they are able to produce good results even when the data is incomplete), are easier to specify thanks to their familiar Prolog-like syntax, and scale better with the data as they swiftly process small applications and have competitive performance with larger applications.

The result is a simple yet powerful language that is the basis of various systems including the Semantic Network Extractor [59] which is a scalable, unsupervised, and domain-independent system that extracts relations, concepts, and learns a semantic network [9] from the Web, producing better results than three other state-of-the-art systems. Also, the system by Wu *et al.* [123] which refines Wikipedia's infoboxes by restructuring their information into a clearly defined ontology. The resulting ontology enhances the infoboxes with improved query processing and other features. Finally, Riedel and Meza-Ruiz's system carries out collective semantic role labelling [100] by processing sentences in three stages that share information between them: predicate identification, argument identification, and argument classification. It is the second best out of five similar systems but, thanks to the flexibility provided by the MLNs, the authors propose several modifications to improve the overall performance.

There are several possible extensions to MLNs like hybrid domains [118] where both discrete and continuous terms are respectively handled by satisfiability or mathematical optimization algorithms (normal MLNs can only handle discrete terms). Another extension is allowing higher-order formulas (normal MLN formulas are FOL only). Second-order logic can be implemented by grounding with constant symbols

(as shown in the example above) and predicate symbols, and has been used for clustering [58].

2.3.1 Summary

An MLN combines FOL and probabilities in a compact and simple representation. In practice they are databases with weighted formulas that can be used to describe the *most probable world* of the database when the formulas are applied to it. MLNs have surpassed all other similar frameworks thanks to their ability to combine soft and hard constraints and handle a greater degree of uncertainty. MNLs are also easier to specify thanks to their familiar, Prolog-like syntax, and scale better with the data. They have been used as the basis of various systems including the Semantic Network Extractor which is a system that extracts relations, concepts, and learns a semantic network from the Web; a system which refines Wikipedia's infoboxes by restructuring their information into a clearly defined ontology; and a system to carry out collective semantic role labelling.

2.4 Inference

Inference is the process of reasoning to derive conclusions from evidence. However, many applications must reason with uncertain or incomplete evidence [87]. Since inference is very effective for reasoning tasks, many proposals have been created to handle the uncertainty using probabilistic reasoning models including fuzzy logic, probabilistic argumentation, evidential reasoning, among others. The basic idea behind these proposals is based on the *possible worlds* that can be created from a KB. For example, if our KB has a single clause, then there are two possible worlds: one where the clause is true and the other where it is false. The "right" world is either of these two, but we do not know which one. This uncertainty can be modelled with probabilities by assigning a probability p to one world and $1-p$ to the other. However, as our KB grows, our possible worlds grow exponentially and thus, the handling of

probabilities adds an additional layer of complexity to the inference problem. More formally, these models are based on *conditional* probabilities, i.e., the probability of an event A (query) given that another event B (evidence) occurred, which is represented as $P(A|B)$.

2.4.1 Inference in MLNs

Inference in MLNs is a combination of probabilistic methods with logical inference. The two approaches to inference in MLNs are marginal and *Maximum a posteriori* (MAP) [24, p.23]. Marginal inference uses the KB to answer the question *What is the probability of a set of predicates with unknown truth values (query) to be true, given a related set of predicates with known truth values (evidence)?* (e.g., in the smokers application, giving a probability to each person of having cancer based on their friends and their smoking habits). In MAP inference, the task is finding the *most probable world* given some evidence, a set of clauses and a query (e.g., in the same smokers application, finding the most likely configuration of boolean values that would determine those people who have cancer and those who do not). Thus, marginal inference gives a probability of being true to each query predicate, while MAP inference only determines if it is true or false. The processing of both approaches is divided in two phases: 1) *grounding* and 2) *search*.

In both inference approaches, the 1) *grounding* phase is the process of assigning values to all free variables in each clause using the evidence (e.g., clause $\neg\texttt{Fr}(\texttt{x},\texttt{y}) \vee \neg\texttt{Sm}(\texttt{y}) \vee \texttt{Sm}(\texttt{x})$ becomes $\neg\texttt{Fr}(\texttt{Gary},\texttt{Bob}) \vee \neg\texttt{Sm}(\texttt{Bob}) \vee \texttt{Sm}(\texttt{Gary})$). Using the ground clauses, marginal inference 2) *search* is based on sampling and MAP inference is based on either weighted satisfiability (weighted MaxSAT) or integer linear programming.

Markov Random Fields

After grounding, the grounded clauses form a Markov Random Field [56] (MRF, also called Markov network), which is a model for the joint probability distribution of a set of random variables $\overline{X} = (X_1, ..., X_n)$, that can be represented by an undirected

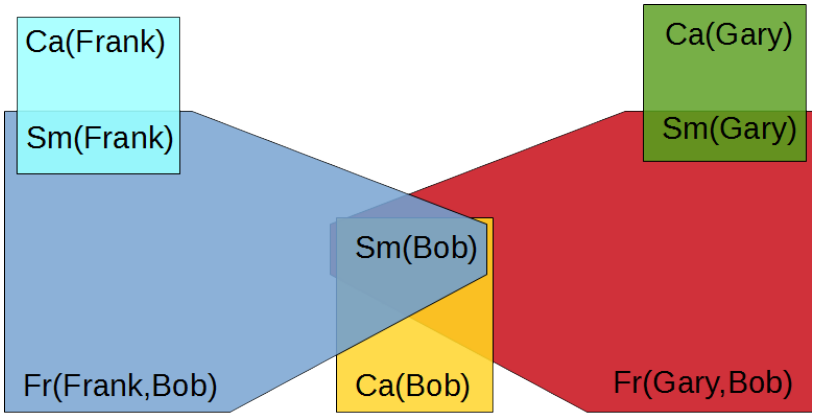


Figure 2.5: Markov Random Field representation of the smokers example.

hypergraph $G = (\overline{X}, E)$. In MLNs, each ground atom is a variable X (a node in the graph) and there is a hyperedge $e \in E$ for each ground clause. For example, Figure 2.5 shows the graphical representation of the MRF for the grounded smokers application, where each node is labelled with the atom's name and each hyperedge is a coloured polygon.

MRFs belong to the family of *Markov processes* which group all stochastic processes that share the *Markov property* including Markov Chains, Hidden Markov Models, etc. In general, the Markov property guaranties the memory-less of our process, i.e., that future states of the process will only depend on the current state and not on the past states that precede it. However, in MRFs, the Markov property is best described as a set of three properties based on the graphical model G and the conditional independence of its random variables $\overline{X}$:

Definition 11. *Pairwise Markov property.* Any two variables X_a and X_b not joined by an edge (non-adjacent) are conditionally independent given all other variables:

$$X_a \perp X_b \quad | \quad \overline{X} \setminus \{X_a, X_b\},\ \{X_a, X_b\} \notin E \tag{2.34}$$

Definition 12. *Local Markov property.* A variable X_a is conditionally independent of all other variables given those variables that share a vertex with it (i.e., all variables

adjacent to X_a, also called the neighbors $N(X_a)$ of said variable):

$$X_a \perp \overline{X} \setminus \{N(X_a), X_a\} \quad | \quad N(X_a) \tag{2.35}$$

Definition 13. *Global Markov property.* Any two subsets of variables $\overline{X}_A$ and $\overline{X}_B$ are conditionally independent given a third, separating subset $\overline{X}_C$ (i.e., a set whose variables have vertices to the variables in $\overline{X}_A$ and $\overline{X}_B$, and there are no direct vertices between any variable in $\overline{X}_A$ and $\overline{X}_B$):

$$\overline{X}_A \perp \overline{X}_B \quad | \quad \overline{X}_C \tag{2.36}$$

Formally, the joint probability distribution of the MRF can be easily established based on the hyperedges of the graph: given the set of random variables $\overline{X}$, $P(\overline{X} = \overline{x})$ (the probability of the whole MLN when all variables in $\overline{X}$ each takes a certain value $\overline{x}$) is defined as

$$P(\overline{X} = \overline{x}) = \frac{1}{Z} \prod_{e \in E} \phi_e(x_e) \tag{2.37}$$

$$Z = \sum_{\overline{X}} \prod_{e \in E} \phi_e(x_e) \tag{2.38}$$

$$\phi_e(x_e) = exp(w_e f_e(x_e)) \tag{2.39}$$

where Z is a normalizing constant (also called partitioning function) that involves summing over all possible combinations of the variables in the system, x_e denotes the truth values of the variables found in hyperedge e, $\phi_e(x_e)$ is a *potential function* or feature where w_e is the weight of the ground clause related to hyperedge e and f_e is a function that returns 1 if the ground clause of the hyperedge is satisfied and zero otherwise. As example, consider an MLN with only the first rule of the smokers application ($1.5 : \neg\texttt{Sm(X)}$ v $\texttt{Ca(X)}$) with `Bob` as the only person (i.e., the only value for `X`). There are four possible worlds for this MLN: 1) $\{\texttt{Sm(Bob)}, \texttt{Ca(Bob)}\}$,

2) $\{\neg\texttt{Sm}(\texttt{Bob}), \texttt{Ca}(\texttt{Bob})\}$, 3) $\{\texttt{Sm}(\texttt{Bob}), \neg\texttt{Ca}(\texttt{Bob})\}$, and 4) $\{\neg\texttt{Sm}(\texttt{Bob}), \neg\texttt{Ca}(\texttt{Bob})\}$. The probability of each world is given by:

$$P_1(\overline{X} = \overline{x}) = \frac{1}{Z} exp(1.5 * 1) \tag{2.40}$$

$$P_2(\overline{X} = \overline{x}) = \frac{1}{Z} exp(1.5 * 1) \tag{2.41}$$

$$P_3(\overline{X} = \overline{x}) = \frac{1}{Z} exp(1.5 * 0) \tag{2.42}$$

$$P_4(\overline{X} = \overline{x}) = \frac{1}{Z} exp(1.5 * 1) \tag{2.43}$$

$$Z = exp(1.5 * 1) + exp(1.5 * 1) + exp(1.5 * 0) + exp(1.5 * 1) \tag{2.44}$$

Note how the third world has a lower probability since it does not satisfy the clause and the other three have the same probability because they *do* satisfy the clause. On the other hand, if we assign a negative weight to the clause (i.e., we want the clause to be unsatisfied), then the third world would be the most probable as $exp(-1.5 * 0) > exp(-1.5 * 1)$.

The probability distribution $P(\overline{X} = \overline{x})$ can also be represented as a log-linear model as follows:

$$P(\overline{X} = \overline{x}) = \frac{1}{Z} exp(\sum_{i \in F} w_i tg_i(\overline{x})) \tag{2.45}$$

where the summation is over the set of all MLN formulas F, Z is the normalizing constant (Equation 2.38), w_i is the weight of the ith formula, and $tg_i(\overline{x})$ is the number of times formula i is true given the current truth values of its predicates ($\overline{x}$) for all its possible groundings. From the example above, it can be seen that the probability remains the same: $\frac{1}{Z} exp(1.5 * 1)$ for the three worlds that have one true grounding and $\frac{1}{Z} exp(1.5 * 0)$ for the world that has no true groundings.

Marginal Inference

In marginal inference, the probability of each atom can be computed based on its Markov blanket. The Markov blanket of an atom X_l is all other atoms that appear in all clauses where X_l appears (e.g., in the smokers example, the Markov blanket for $Sm(Gary)$ would be $Ca(Gary), Fr(Gary, Bob), Sm(Bob)$), and the state of the blanket corresponds to the truth values of the other atoms. Given the Markov blanket state ($MB(X_l)$), the probability of an atom X_l is the probability of the blanket (given by Equation 2.45) when X_l has a truth value x_l divided by the probability of the blanket when $X_l = false$ plus the same probability when $X_l = true$:

$$P(X_l = x_l | MB(X_l)) = \frac{exp(\sum_{f_i \in F_l} w_i f_i(X_l = x_l, MB(X_l))}{exp(\sum_{f_i \in F_l} w_i f_i(X_l = 1, MB(X_l)) + exp(\sum_{f_i \in F_l} w_i f_i(X_l = 0, MB(X_l))} \tag{2.46}$$

where F_l is the set of formulas where X_l appears, w_i is the weight of the ith formula, and $f_i(X_l = x_l, MB(X_l))$ is the truth value (0 or 1) of the ith formula when atom X_l has truth value x_l and its other atoms keep the values of the state of the Markov blanket.

The truth state of each atom given their Markov blanket can be computed with sampling techniques like Gibbs [34] or belief propagation [127]. However, for MLNs, the MC-SAT algorithm found in [94] and described in Section 3.5, produces much better results.

MAP Inference as a weighted MaxSAT problem

MAP inference search for the most probable world is based on determining the maximum sum of the weights of ground clauses that can be *satisfied* (made true) through assigning truth values to their atoms. For example, by setting the atom $\texttt{Ca}(\texttt{Frank}) = true$, the clause $\neg\texttt{Sm}(\texttt{Frank}) \vee \texttt{Ca}(\texttt{Frank})$ becomes satisfied. Also, the

solution to the MLN presented in Section 2.3 is based on MAP inference. From Equation 2.45, the probability of the world y given evidence x is:

$$argmax_y P(y|x) = argmax_y \sum_{i \in F} w_i tg_i(x, y) \tag{2.47}$$

where both Z and the exponential functions are constants and were removed since they do not affect the maximization.

Many MLN systems solve this problem using the MaxWalkSAT [53] algorithm, as it can solve hard problems with thousands of atoms in a short time. MaxWalkSAT works by iteratively selecting an unsatisfied clause and switching the truth value of one of its atoms until no unsatisfied clauses remain. Finally, the most probable world is determined by the truth values of the query atoms (e.g., Frank has cancer because $\texttt{Ca}(\texttt{Frank}) = true$).

MAP Inference as an Integer Linear Programming problem

The search on MAP inference seen as an integer linear programming problem consists of solving a binary optimization problem [99]. In this problem, each ground clause represents a variable and each atom inside the clause also represents a variable. With this representation, we can map the truth values of atoms and clauses to numbers (where $1 = true$ and $0 = false$) and define, for each clause, one positive (if the weight of the clause is positive, Equation 2.48) or negative (if the weight of the clause is negative, Equation 2.49) constraint as follows:

$$\sum_{p \in P} x_p + \sum_{n \in N} (1 - x_n) \geq z_g \tag{2.48}$$

$$\sum_{p \in P} x_p + \sum_{n \in N} (1 - x_n) \leq (|P| + |N|) z_g \tag{2.49}$$

where P is the set of positive atoms in the clause, N the set of negative ones, x_p are

those variables generated from positive atoms, x_n are the variables generated from negative atoms and z_g are those variables that represent a clause (the ones we seek to optimize). Finally, the objective function is defined as:

$$\max \sum_{g \in G} w_g z_g \tag{2.50}$$

where G is the set of ground clauses and w_g the weight of each clause. For example, consider the grounding of the smokers example (Equations 2.27-2.33) with the following variables for each atom and each clause:

$$1.5 : (\neg \texttt{Sm}(\texttt{Bob}) = x_1 \vee \texttt{Ca}(\texttt{Bob}) = x_2) = z_1$$

$$1.5 : (\neg \texttt{Sm}(\texttt{Frank}) = x_3 \vee \texttt{Ca}(\texttt{Frank}) = x_4) = z_2$$

$$1.5 : (\neg \texttt{Sm}(\texttt{Gary}) = x_5 \vee \texttt{Ca}(\texttt{Gary}) = x_6) = z_3$$

$$1.7 : (\neg \texttt{Fr}(\texttt{Gary}, \texttt{Bob}) = x_7 \vee \neg \texttt{Sm}(\texttt{Bob}) = x_1 \vee \texttt{Sm}(\texttt{Gary}) = x_5) = z_4$$

$$1.7 : (\neg \texttt{Fr}(\texttt{Gary}, \texttt{Bob}) = x_7 \vee \neg \texttt{Sm}(\texttt{Gary}) = x_5 \vee \texttt{Sm}(\texttt{Bob}) = x_1) = z_5$$

$$1.9 : (\neg \texttt{Fr}(\texttt{Frank}, \texttt{Bob}) = x_8 \vee \neg \texttt{Sm}(\texttt{Bob}) = x_1 \vee \texttt{Sm}(\texttt{Frank}) = x_3) = z_6$$

$$1.9 : (\neg \texttt{Fr}(\texttt{Frank}, \texttt{Bob}) = x_8 \vee \neg \texttt{Sm}(\texttt{Frank}) = x_3 \vee \texttt{Sm}(\texttt{Bob}) = x_1) = z_7$$

these clauses would generate the following problem:

$$\max(1.5z_1 + 1.5z_2 + 1.5z_3 + 1.7z_4 + 1.7z_5 + 1.9z_6 + 1.9z_7)$$

`Subject to` :

$$(1 - x_1) + x_2 \geq z_1$$

$$(1 - x_3) + x_4 \geq z_2$$

$$(1 - x_5) + x_6 \geq z_3$$

$$(1 - x_7) + (1 - x_1) + x_5 \geq z_4$$

$$(1 - x_7) + (1 - x_5) + x_1 \geq z_5$$

$$(1 - x_8) + (1 - x_1) + x_3 \geq z_6$$

$$(1 - x_8) + (1 - x_3) + x_1 \geq z_7$$

which can be solved with a wide variety of methods like interior-point methods or the simplex method, which are provided by many solvers like Gurobi [149].

2.4.2 Summary

Statistical inference is the process of reasoning with uncertain or incomplete information. In MLNs there are two approaches, marginal and MAP, which are based on sets of query and evidence atoms and whose processing is divided in two steps: grounding and search. *Grounding* is the process of assigning values to the variables in the formulas. The grounded formulas then form an MRF, an undirected graph where each atom is a node and each formula is a hyperedge, that is used during the search. The *search* step in marginal inference determines the probability of the query atoms being true given the evidence. This is performed by sampling the MRF with algorithms like Gibbs [34] or MC-SAT[94]. The search in MAP inference finds the most likely configuration of truth values for the query atoms. This task can be seen as a satisfiability problem that can be solved with algorithms like MaxWalkSAT [53] or an optimization problem to be solved with the simplex or barrier methods [89].

Further information can be found in Sections 3.4 and 4.2.1.

2.5 Learning

ML is a branch of AI focused on creating and refining algorithms that can perform predictions by learning from data [81]. Such learning is usually divided in two: unsupervised and supervised. Unsupervised learning involves working with unlabelled data and making predictions about its structure. It is used in a wide variety of tasks including clustering and neural networks. However, the unlabelled data makes the performance of these algorithms difficult to evaluate. A simple example of this kind of learning is the detection of anomalies in a data set (of bank operations or website logins for example), where the learner would first determine the normal instances (i.e., the majority of the data) and then the abnormal ones (outliers) without the need of any extra information or labels on the data.

In the supervised learning approach, a set of known examples of a problem are provided as training data and the task is to analyse this data in order to provide a function to map new unknown examples. Supervised learning is used in classification, regression, and ranking, among other tasks. As an example, consider the problem of classifying tree leaves where, given a set of known leaves and their characteristics, we can create a concept that determines, for instance, the type of leaf based on its shape, colour, size, etc. for all leaves known and unknown. Learning in MLNs is based on the supervised approach and is used to automatically create or refine weights and clauses.

2.5.1 Weight Learning

Weights in MLNs can be learnt or refined either *generatively* or *discriminatively*. The basic idea in both cases is to construct a function for the weights using the training data and then find the optimal values of said formula which correspond to the ideal values for the weights, starting the search from a "good" initial value. Also, both

approaches must first ground the MLN (similar to the inference grounding explained in Section 2.4.1).

Generative Learning

In generative weight learning, the evidence and clauses are used as training data and a close-world assumption [35] is made (i.e., all ground atoms not in the evidence are considered false). The function to compute the weights is based on the pseudo-log-likelihood [8] of the evidence X for the given weights w and is defined as follows:

$$logP_w^*(X = x) = \sum_{l=1}^{n} logP_w(X_l = x_l|MB(X_l)) \tag{2.51}$$

where n is the number of ground atoms, X_l is a ground atom and x_l its truth value, and $P_w(X_l = x_l|MB(X_l))$ is the probability of X_l taking value x_l for the current weight values w given its Markov blanket. Equation 2.46 shows how to calculate said probability, the only difference is that the weights w_i change at each iteration when learning. In order to optimize Equation 2.51 using the L-BFGS optimizer, the gradient with respect to the weight of each formula i in the MLN (w_i) must be computed using the following equation:

$$\frac{\partial}{\partial w_i} logP_w^*(X = x) = \tag{2.52}$$
$$\sum_{l=1}^{n} tg_i(x) - P_w(X_l = 0|MB(X_l))tg_i(x_{[X_l=0]}) - P_w(X_l = 1|MB(X_l))tg_i(x_{[X_l=1]})$$

where $tg_i(x)$ is the number of times the formula i is true for all its possible groundings. Likewise, $tg_i(x_{[X_l=1]})$ and $tg_i(x_{[X_l=0]})$ are the number of times formula i is true when we force an atom X_l to be true or false. For example, given $a(X) \vee \neg b(X)$ with evidence $a(1), b(1), b(2), b(3)$, the possible groundings of this formula and its truth values would be $a(1) = true \vee \neg b(1) = false, a(2) = false \vee \neg b(2) = false, a(3) =$

$false \vee \neg b(3) = false$. Thus, the formula has a single true grounding by default ($a(1) = true \vee \neg b(1) = false$) and $tg_i(x) = 1$, but if we force $b(X) = false$ then all groundings evaluate to true ($tg_i(x_{[b(X)=0]}) = 3$) and if we force $b(X) = true$, once again we get a single true grounding ($tg_i(x_{[b(X)=1]}) = 1$).

With the function and the gradient, the L-BFGS algorithm can iteratively compute better values for the weights until a good solution is reached. Moreover, the process is rather fast, with most of the processing time spent grounding and counting the number of true groundings ($tg_i(x), tg_i(x_{[X_l=1]}), tg_i(x_{[X_l=0]})$).

Discriminative Learning

Discriminative weight learning not only considers the evidence and the clauses but also a set of query atoms. Said additional consideration allows discriminative learning to outperform generative learning and other methods based solely on either logic or probabilities [107]. By grounding the MLN and then partitioning the ground atoms into X evidence atoms and Y query atoms, the main function of the discriminative learner called conditional log likelihood (CLL) can be defined as:

$$-logP_w(Y = y|X = x) = log(Z_x) - \sum_{i=1}^{n} w_i tg_i(x, y) \tag{2.53}$$

where Z_x is a normalizing constant for the evidence X, n is the number of formulas in the MLN, w_i is the current weight for the ith formula, and $tg_i(x, y)$ is the number of times the value of the ith formula is true. Note that we are using the negative of the CLL (hence the negative in the formula) because it allows us to express the problem as a weight minimization problem (instead of a maximization one), which is consistent with most of the optimization literature. Thus, the gradient is also negative and is computed as:

$$\frac{\partial}{\partial w_i}(-logP_w(Y = y|X = x)) = E_{w,y}(tg_i(x, y)) - tg_i(x, y) \tag{2.54}$$

where $E_{w,y}(tg_i(x, y))$ is the expected number of times the i formula is true, based on its current weight w. Since $E_{w,y}(tg_i(x, y))$ depends on the weights (unlike $tg_i(x, y)$) and the truth values of y cannot be determined using only the evidence, its computation cannot be performed using deterministic methods and is instead performed using inference (as presented in Section 2.4.1).

The optimization algorithms used to compute discriminative learning include Voted Perceptron (based on MAP inference), DN and Scaled Conjugate Gradient (both based on marginal inference). The last two methods also require information from the second derivative (Hessian), which is computed as:

$$\frac{\partial}{\partial w_i \partial w_j}(-logP_w(X = x|Y = y)) = \tag{2.55}$$
$$E_{w,y}(tg_i(x, y)tg_j(x, y)) - E_{w,y}(tg_i(x, y))E_{w,y}(tg_j(x, y))$$

2.5.2 Clause Learning

MLN clauses can also be learnt or refined using ILP-like methods. In normal ILP, we are given a set of positive examples E^+, a set of negative examples E^-, a description of the examples in FOL (background knowledge B), a language bias, and a set of constraints. The goal is to find a FOL hypothesis H that ideally covers (explains) all positive examples and none of the negative ones.

In general, there is no constructive algorithm to obtain the best rules. Instead, ILP enumerates legal rules and selects the best one found. The most popular algorithm to learn a rule is shown in Algorithm 1. This algorithm is based on Progol's [85]. Progol is one of the first ILP systems implemented. Aleph [139], the system used in this work, is based on Progol and, therefore, uses the same algorithm.

Algorithm 1 The ILP algorithm used by Aleph.

```
1: R_0 ← Init();
2: {P_0, N_0} ← Coverage(R_0);
3: while StopNotReached do
4:     R ← NextNode()
5:     {P, N} ← Coverage(R);
6:     if Better(P_0, N_0, P, N) then
7:         R_0 ← R; P_0 ← P; N_0 ← N;
8:         break
9:     end if
10: end while
```

The process starts from an initial rule R_0 (which is also considered the best rule found so far) and its coverage counters P_0 and N_0. Then, the algorithm searches for better rules until it finds a high-quality rule (one which covers all positive examples and no negative ones), all possible rule combinations are tried, or a certain number of iterations is reached. Three functions are critical to the algorithm. $NextNode()$ (line 4) creates a new rule R by using increasingly larger combinations of the best predicates (as shown in the example below). *Coverage* (line 5) obtains the number of examples that are explained by the rule R and stores them in P and N, for positive and negative examples, respectively. Finally, $Better()$ (line 6) compares the current best number of covered examples with the newly generated ones and changes the best rule found so far along with its coverage counters if the new one is better. Usually, to measure if a rule is better than other, a quality score is compared like the difference between P and N $(P - N)$ versus $P_0 - N_0$, selecting the one with the higher value.

As an example of this algorithm, consider the 'grandfathers' application where the task is to find a person's grandfather based on father relationships, with the following background knowledge, positive and negative examples:

```
Background Knowledge
  father(Bob, Sam).
  father(Sam, John).
  father(David, Greg).
```

```
  father(Greg, Harry).
  father(Ana, Gary).
  father(Helen, Frank).
Positive Examples
  grandfather(Bob, John).
  grandfather(David, Harry).
Negative Examples
  grandfather(Gary, Ana).
  grandfather(Frank, Helen).
```

Often we start with R_0 as the rule that is always true. In our example, this corresponds to saying that everyone is a grandfather:

```
grandfather(Y,X) :- true.
```

and the initial coverage would be $P = 2$ and $N = 2$, having an initial score of $P - N = 0$. To start searching, we include a body literal:

```
grandfather(Y,X) :- father(X,Y).
```

and we get $P = 0$, $N = 2$ and $P - N = -2$. Since the score was worse, we discard that rule and try another until we reach the rule:

```
grandfather(Y,X) :- father(Y,Z), father(Z,X).
```

which has the best possible score $P = 2$, $N = 0$ and $P - N = 2$.

Coverage computation (line 5 of Algorithm 1) is a key operation in this (and most) ILP algorithms, often dominating running time. It is usually implemented in Prolog using several optimization techniques like indexing [103] and coverage lists [139]. For MLN clause learning, coverage is replaced by a function that better reflects MLN applications called weighted pseudolog-likelihood, which is explained in detail in Section 3.5.1.

2.5.3 Summary

ML is used to perform predictions by learning data. In MLNs, weights can be learnt generatively or discriminatively by constructing a function for them based on some training data and then optimizing said function. In generative learning, all training data is considered evidence and the objective function and its gradient can be calculated by grounding the MLN and then counting the number of groundings whose truth value is true. The optimization algorithm used for generative learning is the L-BFGS algorithm. In discriminative learning the training data is divided in evidence and query data, and a much better objective function can be constructed with them. The values for said function, its gradient, and Hessian are computed by grounding and sampling (a process similar to marginal inference). These values are then used by optimization algorithms like Voted Perceptron and DN [24, 89]. Our work on weight learning is discussed in Sections 3.5 and 4.2.2. Clauses can also be learnt using ILP-like methods, where given a set of positive and negative examples, the task is to generate good clauses that cover all positive examples and none of the negative ones. ILP systems like Aleph [139] iteratively propose and evaluate new clauses until a good clause is found or a certain number of iteration have passed. More about clause learning is presented in Sections 3.5 and 4.3.

Chapter 3

Proposal: Parallel Processing of MLNs on GPUs

The use of Markov Logic Networks (MLNs) for various types of applications has fostered the development of several systems. *Alchemy* was the first MLN system implementation [24, 137]. It is written in C++ and is one of the most comprehensive systems, including various algorithms for inference and learning based on satisfiability. However, Alchemy does not include any kind of parallelism and does not cope well with large real-world applications. *theBeast* [99] (now called theppl) was the next system written in Scala and based on integer linear programming instead of satisfiability. Like Alchemy, theBeast is not a parallel system, however, its latest version (theppl) was designed with heavy emphasis on modularity and thus, users can create parallel modules adapted to their needs for grounding, search, etc. Despite the absence of parallelism, theBeast is faster than Alchemy for some problems, but lacks many of Alchemy's features and has a slightly different syntax. *Tuffy* [88, 136], written in Java and also based on satisfiability followed a few years after theBeast. Tuffy greatly improves upon Alchemy's approaches by relying on a relational database management system (RDBMS) and exploiting multicore parallelism during the search phase. Finally, *RockIt* is the latest system [90, 138] that, like theBeast, treats inference as an integer linear programming problem. It is written in Java and makes ample use

of parallelism for grounding, search, and learning, allowing it to outperform all other CPU-based systems.

Said systems have tried, with varying degree of success, to efficiently compute MLNs by taking advantage of various techniques, specially multicore parallelism. However, for many real world applications with data not exceeding a few hundred MBs, the processing time remains unacceptably high, ranging from several hours to days, as shown in our results in Section 5.3.2.

The purpose of our research is to investigate how to efficiently use Graphics Processing Units (GPUs) to process MLNs. In this chapter we first justify our use of GPUs to accelerate MLN processing (Section 3.1). We then discuss some of the design issues for MLN processing (Section 3.2). Next, we present our designs and the state of the art for processing each MLN phase, namely: inference, whose steps are grounding (Section 3.3) and search (Section 3.4), and learning (Section 3.5). Finally, we conclude with a summary of the whole chapter (Section 3.6).

3.1 Why use GPUs?

GPUs are high-performance many-core processors capable of very high computation and data throughput [154]. GPUs are now used in a wide array of applications [153], including gaming, data mining, bioinformatics, chemistry, finance, numerical analysis, imaging, weather forecast, etc. Such applications are usually accelerated by at least an order of magnitude, but accelerations of 10x or more are common. Moreover, there is a huge difference in the cost-benefit of using GPUs instead of CPUs on highly-parallel applications and we believe the difference will only continue to increase in the following years. For example, at the time of writing of this thesis, NVIDIA had just released the GeForce 1080Ti GPU, boasting 11.3 Tera Floating-Point Operations per Second (TFLOPS) with a power consumption of 250 Watts (W) and a retail price of $700 US dollars (USD). In contrast, Intel is about to release the Core i9-7980XE CPU processor, which has been promoted as the first commercial processor capable

of 1 TFLOPS with a power consumption of 165W and a retail price of $1,999 USD. As it can be seen, GPUs offer much greater performance at a lower cost and mildly higher energy requirements.

While various different brands of GPUs and programming models are available, our work is based exclusively on NVIDIA GPUs programmed using their Compute Unified Device Architecture (CUDA) [147] interface to the C language. However, it is important to mention the Open Computing Language (OpenCL) framework of the Khronos group [134], which seems to have a bright future since many companies are working on it and it is more flexible than CUDA as it can be executed on both NVIDIA and AMD GPUs, and on the CPU. Nevertheless, we chose the CUDA framework for our work, since OpenCL lacks many useful tools like an advanced debugger or profiler and, for some applications, CUDA has better performance. Hence, unless otherwise stated, any mention of GPUs in our work refer to NVIDIA GPUs and its programming model. In the following subsections, we present only the most important aspects of GPUs. A detailed presentation of GPUs is given in Appendix A.

GPU Architecture and Programming Model

GPUs are akin to single-instruction-multiple-data (SIMD) machines: many processing elements run the *same program* but on distinct data items. The program, referred to as the *kernel*, is similar to a C function and can be quite complex including control statements such as *if* and *while* statements. A kernel is executed by groups of threads called *warps*. Each warp has a single control unit that makes all its threads execute the same instruction at a time. When threads *diverge* in their execution path, the warp serially executes each branch path, disabling threads not on the path being executed, until all paths complete and the threads converge to the same execution path. Hence, if for example, 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 threads and memory are organized hierarchically as shown in Fig. 3.1. Each

thread has its own *per-thread local* memory. Threads are grouped into *blocks* of the same size (defined by the programmer), with each block having a memory *shared* by all threads in the block. Blocks are internally divided by the GPU driver into warps, based on the hardware capabilities of the GPU (see "Thread Hierarchy" in Appendix A.2.2). Finally, thread blocks (as many as the programmer defines) 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*, whose transfers are *coalesced* by the GPU into as few transactions as possible when good access patterns are used. The simplest and most commonly used pattern makes thread n access element n in an array.

At hardware level, GPU cores are grouped into Streaming Multiprocessors (SMs), with each SM handling the registers, caches, and other functions of its cores. When work is scheduled by the GPU driver, each SM receives a certain number of warps and executes them at the same time if possible.

3.2 Design Issues for Processing MLNs on GPUs

GPUs can substantially improve performance and are now being used for general purpose computing in addition to game applications. GPUs are particularly suitable for compute-intensive, highly parallel applications. They perform well in scientific applications that model physical phenomena over time and space, wherein the "compute-intensive" aspect corresponds to the modelling over time, while the "highly parallel" aspect to the modelling at different points in space. Data-intensive, highly parallel applications such as database relational operations and MLNs can also benefit from this model.

3.2.1 GPU Issues

As we explored the design space of parallel processing of MLNs on GPUs, we considered several aspects of GPU programming, in order to create efficient GPU

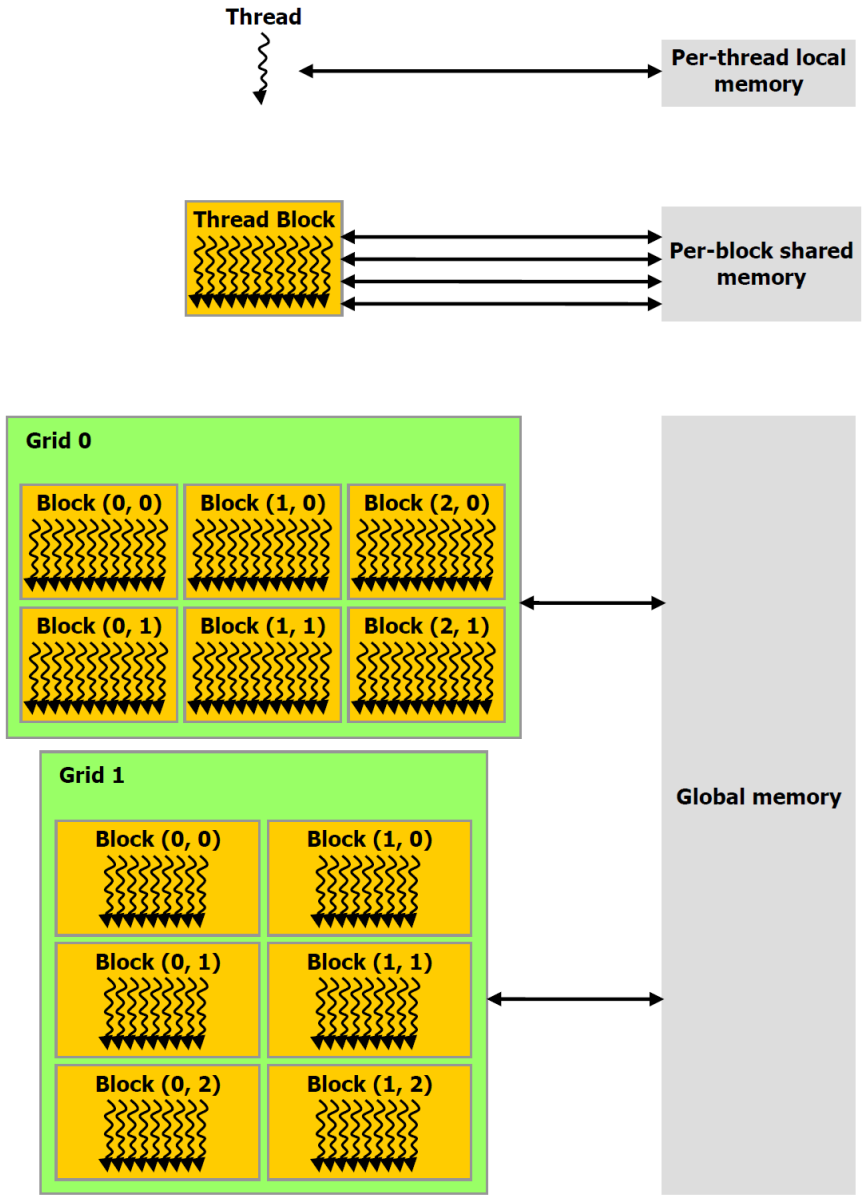


Figure 3.1: Thread and memory hierarchy on GPUs. Image taken from [147].

algorithms. One of the most important aspects of GPU programming is the handling (transferring and processing) of data in *bulk*, rather than element by element. How large a bulk of data should be depends on the *computation-to-communication ratio*, which reflects that a proper amount of data was selected to be transferred. This amount should be small enough to fit in the *limited* amount of GPU memory available, but large enough so that the number of transfers is minimized and the processing time is spent computing data on the GPU, rather than transferring it. For MLNs, a good ratio could be maintained by having a large amount of evidence data and by selecting the best hardware platform (CPU, GPU, or both) for each operation (e.g., sending a few rows instead of a whole evidence table to the GPU is a poor choice, as is using the GPU to control the flow of the MLN processing algorithm, instead of using it for key operations like sorting).

Other important efficiency consideration is maximizing *occupancy*, i.e., each time a kernel is launched, it should be processed by all available cores. Less than maximum occupancy occurs when some warps cannot be executed in parallel because they use too many registers, too much shared memory and/or an incorrect number of threads per block was defined (e.g., if one defines blocks of 28 threads for a GPU whose SMs have 32 cores, then 4 cores will go unused, thus not achieving maximum occupancy). This leads to slower kernels as the full computational resources of the GPU are not completely used. Fortunately, MLN operations do not require an excessive amount of registers or shared memory and understanding the number of threads that a certain operation requires is relatively easy (e.g., most relational operations like selections and joins use one thread per row to be processed).

As mentioned in Section 3.1, *thread divergence* is another issue that can adversely affect the kernels. However, it is very difficult to avoid as MLN operations are rather complex, requiring several *if* conditions and *while* cycles that may cause divergence. Furthermore, MLN evidence is usually encoded using strings of variable sizes, whose processing uses a variable number of cycles and is also a cause of divergence.

Also, the amount of *GPU memory* is limited and cannot be automatically

"extended" using paging like the CPU does. For MLNs, while the whole evidence in a single MLN may be larger than the total amount of GPU memory, it is usually divided in tables and not all tables are necessary to perform an operation. However, if several tables are loaded and then discarded to free up memory each time an operation is performed, we would create a poor computation-to-communication ratio. A control mechanism that decides when to load and discard tables is necessary (like the one presented in Section 4.1).

Finally, there are some *GPU optimizations* that can be easily applied when working with MLNs, since the data in MLN tables can be evaluated in any order: *coalesced memory access* occurs when threads within the same warp read contiguous memory locations (e.g., thread 1 reads location 1, thread 2 location 2, and so forth) and accelerates the execution by allowing the driver to perform a single read of the data for all threads, rather than one per thread; memory transfer operations can be performed *concurrently* with kernel executions, allowing the processing of a kernel and the data loading for the next kernel to be performed at the same time; and *shared memory* can be used instead of global memory to store data that threads constantly access, improving the performance of said accesses.

Resuming in Figure 3.2 the concepts covered in this subsection and in Section 3.1, there are several aspect to *consider* like designing GPU algorithms that handle the data in bulk, working with the limited amount of GPU memory available and ensuring that all threads are occupied and their execution does not diverge. GPU algorithm performance can be *optimized* by overlapping memory transfers with processing, by having consecutive threads read consecutive memory locations and by exploiting the speed of the shared memory. Also, CPU and GPU can *interact* in many ways: memory transfers between GPU and CPU should not take longer than the GPU processing time; working together at the same time; and deciding which hardware is best for a certain operation. Finally, recall that there are two main *languages* for GPU programming (but this thesis work is based only on CUDA), and that the substantial processing power of the GPUs has prompted their use in a rich variety of

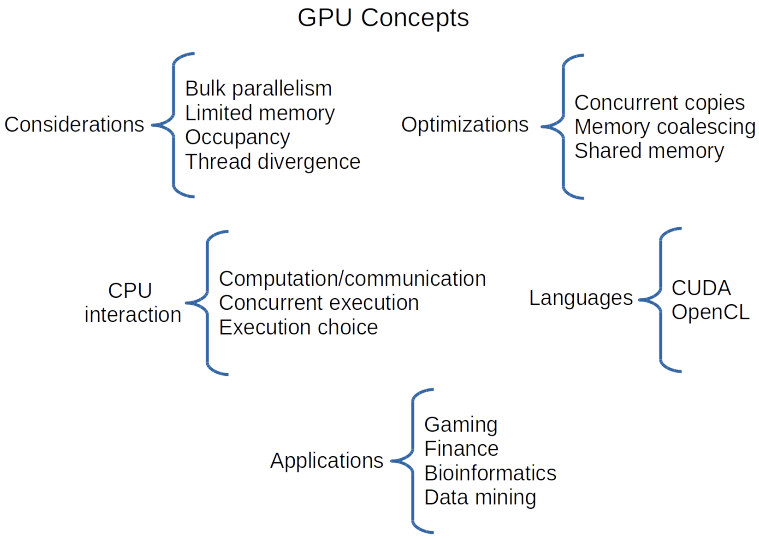


Figure 3.2: Summary of the most important GPU concepts.

applications [153] spanning several different fields.

3.2.2 MLN Issues

In MLN processing, *grounding* (the process of assigning values to the variables in the formulas) can be performed using a top-down approach (similar to Prolog), which reduces a problem to simpler problems and then composes the solution of the original problem as the solutions of simpler problems. However, top-down grounding offers little in terms of optimization or parallelisation as it process one element at a time. In contrast, bottom-up grounding (BUG) works by applying the rules to the given data, thereby deriving new data, and repeating this process until a fixed point is reached (i.e., no more new data is derived). As BUG works with sets of elements, it fits well with the GPU's bulk data handling. Moreover, BUG is an iterative processes that works over a set of formulas not related to each other in the same iteration. This means that formulas can be evaluated in parallel at each iteration. Finally, a formula can in turn be seen as a series of relational algebra (RA) operations (mainly selection, join, and projection) which can also be parallelised individually [23]. For example, if

an MLN has formulas $f1$ and $f2$, and $f1$ requires a join operation, then $f1$ and $f2$ can be independently evaluated from each other and the join in $f1$ can be performed in parallel.

As an example of the difference between top-down grounding and BUG, recall the last rule of the *smokers* example presented in Section 2.3 ($\neg\texttt{Fr}(\texttt{x},\texttt{y}) \vee \neg\texttt{Sm}(\texttt{x}) \vee \texttt{Sm}(\texttt{y})$) along with the facts `fr(frank, bob)`, `fr(gary, helen)`, `sm(frank)` and `sm(gary)`. The top-down approach can be exemplified using Prolog by rewriting the rule with Prolog's syntax (`sm(Y) :- fr(X,Y), sm(X).`) and executing the `trace` command. In YAP Prolog [15], the query to determine all smokers (`sm(Y).`) produces the following trace (the first part is omitted for brevity):

```
1  ?- sm(Y).
2  ...
3              (1)    redo:sm(gary) ?
4              (2)    call:fr(_131205,_131116) ?
5  ?           (2)    exit:fr(frank,bob) ?
6              (3)    call:sm(frank) ?
7  ?           (3)    exit:sm(frank) ?
8  ?           (1)    exit:sm(bob) ?
9  Y = bob ? ;
10             (1)    redo:sm(bob) ?
11             (3)    redo:sm(frank) ?
12             (4)    call:fr(_131395,frank) ?
13             (4)    fail:fr(_131395,frank) ?
14             (3)    fail:sm(frank) ?
15             (2)    redo:fr(frank,bob) ?
16 ?           (2)    exit:fr(gary,helen) ?
17             (5)    call:sm(gary) ?
18 ?           (5)    exit:sm(gary) ?
19 ?           (1)    exit:sm(helen) ?
```

```
20 Y = helen ? ;
21            (1)    redo:sm(helen) ?
22            (5)    redo:sm(gary) ?
23            (6)    call:fr(_131395,gary) ?
24            (6)    fail:fr(_131395,gary) ?
25            (5)    fail:sm(gary) ?
26            (2)    redo:fr(gary,helen) ?
27            (2)    fail:fr(_131205,_131116) ?
28            (1)    fail:sm(_131116) ?
29 false.
```

As it can be seen, top-down is a lengthy process that obtains the results one by one (note that the results are `frank`, `gary`, `bob`, and `helen`). For example, the process to find `bob` starts at line 3 by resuming from the last solution found and attempting to generate alternative solutions. Since there are no other solutions for `sm(gary)`, YAP calls the first predicate of the next possible rule (line 4, where the numbers starting with an underscore represent YAP's variable identifiers). The call then exits with the first value that satisfies the variables (line 5) and the last predicate is called (line 6, whose value is already known, hence no variable). With the result of the last predicate (line 7), the whole rule is true and `bob` is obtained as result (line 8). A similar process is repeated to find `helen` and then again to determine that all results have been found. Furthermore, each new addition to either `fr` or `sm` would imply many more steps to find all solutions.

In contrast, the bottom-up process of GPU-Datalog (our Datalog engine presented in Section 4.1) begins by finding all the rules that are relevant to the query. Since the only rule is relevant, it is processed by loading all facts with the same name as the first predicate (`fr(frank, bob)` and `fr(gary, helen)`) and then loading all the facts from the second predicate (`sm(frank)` and `sm(gary)`). The rule is then solved with the RA formula:

$$\Pi_{\mathtt{Y}}(\mathtt{fr}(\mathtt{X},\mathtt{Y}) \bowtie_{\mathtt{X}} \mathtt{sm}(\mathtt{X})) \cup \mathtt{sm}(\mathtt{X}) \tag{3.1}$$

that indicates that a join is performed between the first column of the first predicate and the only column of the second one. The result (`fr(frank, bob)` and `fr(gary, helen)`) is then projected to leave only `bob` and `helen`, and they are in turn appended to the already known values for `sm` (`frank` and `gary`). If more values are added to `fr` and/or `sm`, all solutions would still be found by using the same formula, the RA operations would simply process more elements.

The *search* phase as a satisfiability problem can also benefit from parallelism as the MaxWalkSAT algorithm [53], despite its extensive data dependencies, it can be parallelised by switching the truth value (flipping) of several atoms at the same time, trying to satisfy many clauses at each iteration (e.g., if formula $f1$ and formula $f2$ are unsatisfied and are conformed of different atoms, then we could use a thread to flip an atom in $f1$ and another thread to flip an atom in $f2$ during the same iteration, instead of taking two iterations). This idea is corroborated by the similar, parallel SAT algorithms that exist in the literature [92]. When solving an integer linear programming problem, the search phase can also be parallelised using finely-tuned, multicore solvers like Gurobi [149].

As *weight learning* (described in Section 2.5.1) requires grounding, it can also benefit from its parallelisation. Furthermore, the optimization algorithms used to find the best weight values have to do *sampling* using an algorithm similar to MaxWalkSAT called MC-SAT. This MC-SAT algorithm also switches the truth values of the atoms and thus, it can be parallelised like MaxWalkSAT.

Finally, in general *clause learning* (Section 2.5.2) with Inductive Logic Programming (ILP), a great number of candidate clauses can be created. To determine if said clauses are useful or not they need to be grounded and this grounding can also be done in parallel (e.g., if the ILP system proposes formula $f1$, how useful

$f1$ is can only be determined by grounding it and counting the number of positive and negative groundings). This idea can be extended to MLNs by using an MLN-specific formula (like Equation 3.11 of Section 3.5.1) instead of counting the groundings.

Based on our analysis of the design space, we propose two designs for parallel processing of MLNs on GPUs called GPU-Tuffy (Section 4.4) and GPU-RockIt (Section 4.5). The main motivation behind having two designs is their different approach to the inference problem, which results in one design outperforming the other for some applications and vice versa. However, before presenting our proposals (Chapter 4), the following sections detail our approach to grounding, search, weight learning, and clause learning in GPUs and the current state of the art for each one.

3.3 Our Approach to Grounding on GPUs

Grounding can be one of the most time-consuming tasks of MLN processing, thus we propose the use of a GPU-accelerated language to perform it. We consider grounding in MLNs as a First-Order Logic (FOL) program and the processing infrastructure for FOL in both our MLN platform designs (GPU-Tuffy and GPU-RockIt) is Datalog, a language that has been used as a data model for relational databases [116, 117]; syntactically it is a subset of Prolog. Similar to MLNs, Datalog programs can be processed top-down or bottom-up (as discussed in Section 3.2.2).

A Datalog program consists of a finite number of facts, rules, and queries. Facts are statements about something relevant, for example "John is Harry's father". Rules are sentences that allow the deduction of new facts from known facts, e.g., "If X is the father of Y and if Y is the father of Z, then X is the grandfather of Z". If we are only interested in a subset of all facts that can be derived from the rules, a query can be used (e.g., the query "Who is the grandfather of harry?"). Facts can be seen as rows in a relational database table, while rules and queries can be seen as SQL queries.

Every Datalog program can be translated into a series of RA operations [11].

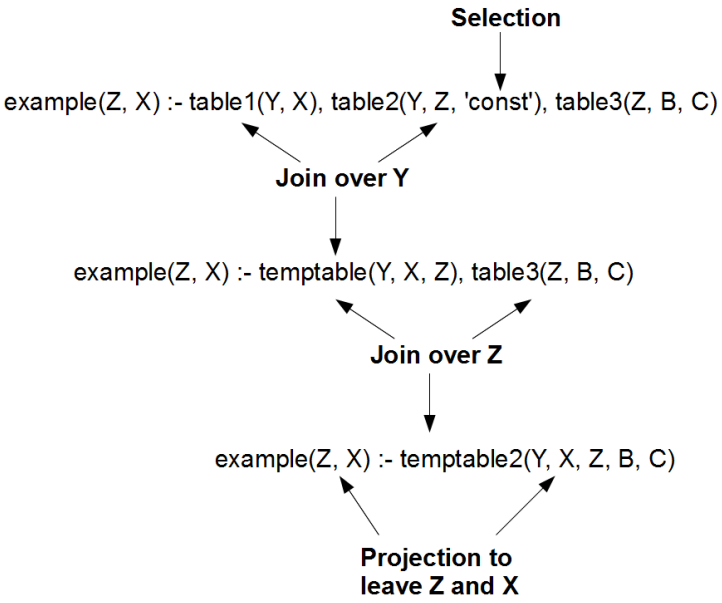


Figure 3.3: Equivalence of Datalog rules and relational operators.

Figure 3.3 shows this equivalence between rules and relational operators. *Selections* are made when constants appear in the body of a rule. Next, a *join* is made between two or more predicates in the body of a rule using the variables as reference. The result of a join can be seen as a temporary predicate (or table) that has to be joined in turn to the rest of the predicates in the body. Finally, a *projection* is made over the variables in the head of the rule. Any query that can be answered using RA can also be answered using a Datalog program. Thanks to recursion, Datalog may even evaluate queries which cannot be evaluated in RA (like the recursive query in the *smokers* example).

A *pure* Datalog program consists of *simple* and recursive clauses, and a single query. However, in order to allow Datalog to process MLNs, it has to be extended with: management of negation, built-in comparison predicates, and interfaces to communicate directly with Relational Database Management Systems. Furthermore, in order to be able to run inductive logic programs (the basis of clause learning), additional extension have to be included like: running several queries in sequence, arithmetic predicates $(+, -, *, /)$ like $\texttt{a(Z)}, \texttt{Z} + 3$, and aggregation (`GROUP BY` operator in SQL).

From the perspective of GPU programming, Datalog programs can be processed using GPU-based RA operations. However, this approach presents some design issues including: handling strings found in the data, as its processing on GPUs causes warp divergence; minimizing the number of data transfers between GPU and CPU, since each transfer implies some overhead; and efficiently processing RA and other operations. Also, to handle large amounts of data that exceed the memory capacity of the GPU and to allow multiple GPU to process an application simultaneously, a data partitioning scheme is necessary. Of great importance to this scheme is the size of the partition. If the partition is too small, performance is adversely affected as we need to perform many more iterations. If it is too large, the result of the operation may not fit and the whole process will fail.

Likewise, RA operations in GPUs present their own issues. Perhaps the most important one is the impossibility of knowing the result size beforehand (e.g., in the selection, we do not know how many tuples will be selected until we actually perform the operation). To further complicate matters, while the issue can be solved in the CPU using dynamically allocated containers (like those provided by stdlib in C++), in GPUs this method is inefficient as memory reallocation may require several GPU-CPU-GPU transfers.

Further narrowing the design issues by operation, the join is a complex operation that can be performed using many algorithms including nested-loop join with or without indexes, sort-merge join, and hash join [48]. The best algorithm has to be chosen and adapted based on the particularities of GPU programming. Also, the comparison predicates need to be evaluated as soon as their variables are bound, i.e., as soon as the variables appear in a positive predicate. Otherwise, many tuples that could be eliminated earlier in the evaluation are not, potentially increasing computation costs and memory usage.

For negation, in order to correctly map the variables to the data, a safety consideration has to be introduced: variables occurring in a negative literal must also occur in a positive literal of the same clause body. Thus, clauses

like `s(X) :- t(X), not r(X)` and `s(X) :- t(Y,X), not r(X)` are allowed, but `s(X) :- not r(X)` or `s(X) :- t(Y), not r(Y,X)` are not. This restriction is necessary because negation in RA is the *set difference* operator ($\setminus$) which, given some sets of tuples A and B, $A \setminus B$ removes all the common tuples between A and B, leaving only the odd ones in A. If there are no tuples to remove from (i.e., there is no set A), the operation cannot be performed. For example, if we have `t(1)`, `t(2)`, and `r(1)`, then the clause `s(X) :- t(X), not r(X)` can be performed by doing `t(X)` $\setminus$ `r(X)` ($\{1,2\} \setminus \{1\}$), getting `s(2) :- t(2)` as result. In contrast, the clause `s(X) :- not r(X)` cannot be performed because `r(X)` has no set to remove from ($? \setminus$ `r(X)`).

Another restriction when combining negation and recursion is that the program should never allow a predicate to be invoked recursively in its negated form. In other words, we should never allow clauses to be specified as `p(X) :- not p(X)`. Otherwise, recursion may never reach a fixed-point and the program would never finish.

MLN grounding can be seen as a Datalog program where the facts are the MLN evidence and the rules are the MLN formulas, as shown in Algorithm 2. The result of this grounding is the closure of the active atoms (`AA`): atoms whose truth value might change from true to false or vice versa during search, and the active clauses (`AC`): clauses that can be violated (i.e., their truth value becomes false) by flipping zero or more active atoms. The idea is to generate a queue of Datalog rules (`DR`) and apply the RA operators on said rules to find first the active atoms and then, the active clauses. Note that each recursive rule (`R`) is not removed from `DR` until it has reached a fixed-point that will generate no new data. Details on how this algorithm is implemented in our platforms can be found in Sections 4.4 and 4.5.

3.3.1 State of the Art for Grounding

The first grounding algorithms for MLNs were sequential and were written in C++ for the Alchemy [24, 137] system and in Scala for theBeast [99]. They were based on a top-down approach where MLN formulas are evaluated in a manner similar to

Algorithm 2 General algorithm for GPU grounding in MLNs based on Datalog.

```
1   Input: MLN clauses MC and evidence E
2   Output: grounded active atoms AA and active clauses AC
3   TR <- Translate MC to Datalog rules
4   DR <- TR
5   RF <- Read E from database into facts
6   While DR is not empty
7       For each rule R in DR
8           AA <- Perform the RA operations of R using RF
9           If R will not generate more data
10              Remove R from DR
11          End if
12      End for
13  End while
14  DR <- TR
15  While DR is not empty
16      For each rule R in DR
17          AC <- Perform the RA operations of R using AA and RF
18          If R will not generate more data
19              Remove R from DR
20          End if
21      End for
22  End while
23  Return AA, AC
```

Prolog. However, this type of grounding did not cope well with large applications. To overcome this problem, newer systems like Tuffy [88, 136] and RockIt [90, 138] follow a bottom-up approach based on a RDBMS. This approach translates the MLN formulas into SQL queries, allowing a much faster grounding thanks to indexing, query planification, parallelism, among other advantages. Furthermore, RockIt exploits multicore based parallelism by mapping each query to a CPU core.

Some applications can benefit from *lifted inference* [109], where the MLN is computed without having to materialize groundings, thus reducing the overall processing time. Lifted inference is based on *Belief propagation* (BP), where similar predicates are grouped together into *supernodes* and similar clauses into *superfeatures* of a *factor graph* [62], with vertices going from a supernode to superfeature if a predicate in the supernode appears in a clause of the superfeature.

The factor graph is initially constructed with three supernodes for each predicate, one with all true groundings, another with all false groundings, and the last with the unknown groundings (some supernodes can be empty). The superfeatures are initially constructed with all possible combinations of the supernodes with the clauses. As example, Figure 3.4 shows the initial graph for the smokers example with evidence `Sm(John), Fr(John,Bob)`. The coloured vertices represent which supernodes (sn_i) where used to create which superfeatures (sf_i). Note that the true groundings are represented without variables (like `Sm(John)`), the unknown groundings have at least a variable that should exclude all values already covered in the true and false groundings (`Sm(X). X != John`), and there are no false groundings because all the evidence is positive.

The initial factor graph can be prohibitively large to work with but can be iteratively compacted by counting the times and position of each atom in a superfeature and grouping together those atoms with the same counts into a new supernode. With the new finer supernodes, new finer superfeatures can be computed and the process is repeated until no new supernodes can be created. The compacted graph can then be used to search as presented in Section 3.4.1.

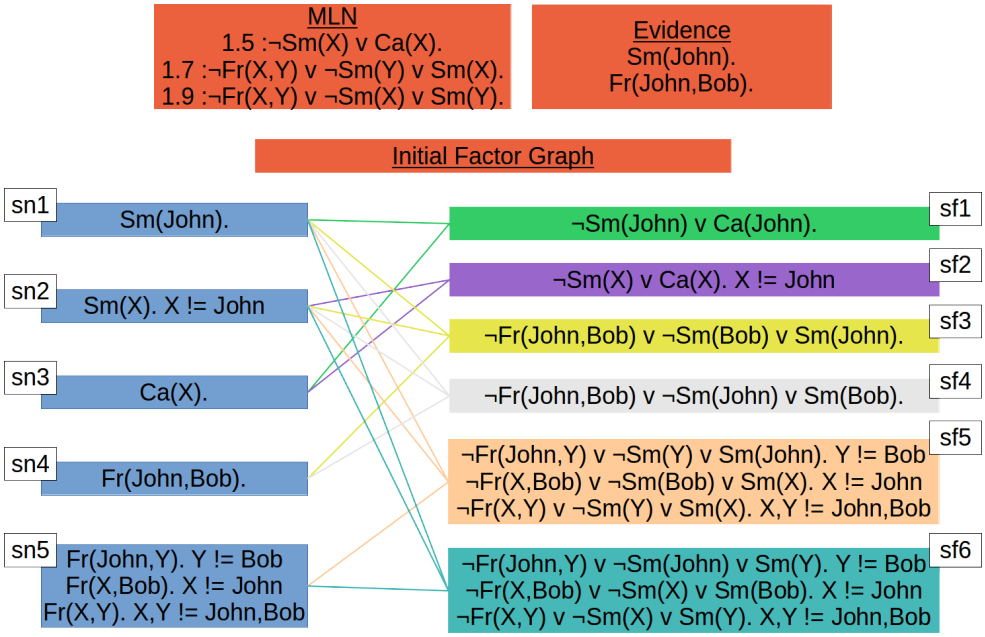


Figure 3.4: Initial factor graph for the smokers MLN, where sn_i are the supernodes and sf_i are the superfeatures.

Similar to lifted inference, after grounding the MLN, Shavlik and Natarajan [105] propose the use of a preprocessing algorithm called Fast Reduction Of Grounded networks (FROG) that can substantially reduce the effective size of the grounded MLNs by rapidly counting how often the evidence satisfies each clause, regardless of the truth values of the query atoms. Since these counts are constant through the following tasks, i.e., Markov Random Field (MRF) construction and search, these counts are computed once and their related ground clauses are safely ignored. The authors' comparison of several MLNs using Alchemy with and without said preprocessing algorithm shows that it can reduce both processing time and memory requirements, but its effectiveness depends on the MLN clauses, as clauses composed entirely of query and/or hidden literals (i.e., those literals that do not appear in the evidence) cannot benefit from the algorithm at all and offers little benefit those clauses with few negative literals.

3.4 Our Approach to Search

The search phase is also a time consuming process which is affected by suboptimal solutions, since current algorithms cannot cover the entire search space to find the best solution. Our proposal is to parallelise a satisfiability algorithm using GPUs, in order to improve both processing time and solution quality by covering more of the search space in less time.

Our first approach to search (used in GPU-Tuffy) is thus based on satisfiability where, once the active atoms and active clauses are computed by the grounding step, said search begins by creating and partitioning the corresponding MRF. The resulting partitions include active atoms and active clauses that are completely independent from each other and can be solved in any order. These properties make them easy to parallelise in multicores, with one thread solving one partition through running the MaxWalkSAT algorithm. Workload can be balanced by storing all partitions in a list and having each thread take a partition, solve it and take another partition until no partitions remain. While this scheme works well for partitions with uniform sizes, our first experimental results showed that this regularity rarely occurs in MRFs with hundreds of thousands of atoms and clauses (see Table 5.5 in Section 5.3.2). The culprit is usually a single partition that takes several orders of magnitude more time than the others to solve. The processing of such partition dominates the search time, forcing the single thread that processes it to finish much later than all other threads which, after having solved the rest of the MRF, keep idly waiting for that thread to finish.

To mitigate this problem, we devised a hybrid solution wherein the larger partition/s are processed in parallel by the GPU, while the remaining smaller partitions are processed in the multicore. This solution required the design and implementation of a GPU-based MaxWalkSAT solver. MaxWalkSAT [53] usually starts at a random point in the search space — if we consider all possible truth assignments of the atoms as our search space, a random starting point would be any

truth assignment. This initial assignment is an initial solution to the SAT problem that leaves many unsatisfied clauses. The sum of the weights of these unsatisfied clauses serves as a measure called the *cost* of the solution, which determines how good a solution is. Note that, in order to set the lowest possible cost at zero (when the number of unsatisfied clauses is also zero) the absolute values of the weights of negative clauses are used in the summation.

The MaxWalkSAT solver tries to find zero cost solutions by iteratively selecting an unsatisfied clause and flipping the truth value of one of its atoms (i.e., changing it from true to false or vice versa). The atom is chosen either randomly or to maximize the sum of the satisfied clause weights. Once the atom is flipped, any clause containing the atom is checked to determine if it is satisfied or not and a new iteration begins. The process is repeated for a number of iterations based on the total number of clauses or until no clauses are unsatisfied.

To perform a parallel search in the GPU, we considered the following two options: 1) each GPU thread starts at a random point in the search space and executes an instance of the MaxWalkSAT algorithm (i.e., each thread performs alone all the MaxWalkSAT steps); at the end, the best solution from all threads is chosen; or 2) all GPU threads start from the same random point and perform together a single instance of the MaxWalkSAT algorithm (i.e., each step is performed in parallel by all threads), with threads flipping different atoms of different clauses and returning the single solution found by these combined flips.

One of the issues of parallelising MaxWalkSAT is the inherently sequential nature of the algorithm, with many dependencies among the data. Another issue we found while testing the GPU design is that, as the algorithm approached a zero cost solution or a good solution with few unsatisfied clauses, the fixed number of flips per iteration may lead to overshoot the solution (too many atoms are being flipped, increasing the number of unsatisfied clauses rather than reducing it).

Our hybrid search approach is shown in Algorithm 3. It receives the active atoms `AA` and clauses `AC`, and returns the most probable world (i.e., the truth values for `AC`).

The algorithm begins by creating and partitioning the MRF and then, at the same time, the CPU processes the smaller partitions while the GPU is processing the larger ones. This processing is performed by MaxWalkSAT which is presented in a resumed version with simple stopping and flipping criteria (the full version for CPUs can be found in [53] and for GPUs in Algorithm 6) where initial truth values `TV` are assigned to the active clauses `AC` in each partition `P` and, while `P` has unsatisfied clauses `UC`, the algorithm flips the atoms `AA` of the unsatisfied clauses `UC`, removing all satisfied clauses and adding any new unsatisfied ones. Once there are no unsatisfied clauses `UC` in a partition `P`, the final truth values `TV` are stored and the hardware (CPU or GPU) moves to the next partition until no partitions remain and the final result is displayed.

Algorithm 3 General algorithm for hybrid search in MLNs based on MaxWalkSAT.

```
1   Input: grounded active atoms AA and active clauses AC
2   Output: truth values TV for AC
3   PS <- Create and partition MRF using AA and AC
4   For each small partition P on PS                    --- CPU
5       TV <- assign truth values to every AC in P
6       While P has unsatisfied clauses UC
7           Flip an atom AA of a clause in UC
8           Remove all satisfied clauses from UC
9           UC <- Add newly unsatisfied clauses if any
10      End while
11      TV <- final truth values of every AC in P
12  End for
13  For each large partition P on PS                    --- GPU
14      TV <- assign truth values to every AC in P
15      While P has unsatisfied clauses UC
16          Flip an atom AA of several clauses in UC
17          Remove all satisfied clauses from UC
18          UC <- Add newly unsatisfied clauses if any
19      End while
20      TV <- final truth values of every AC in P
21  End for
22  Return TV
```

While satisfiability is a very efficient approach, we found that mathematical

optimization using integer linear programming produces better results for some applications (as shown in Section 5.3.2). Thus, our second approach (used by GPU-RockIt) is called Cutting Plane Inference [99] (CPI) and is based on integer linear programming. We make use of CPI as defined in RockIt without any modifications, since it is quite efficient and designing competitive, parallel optimization algorithms is a considerable challenge in itself.

CPI is an iterative algorithm that works by dividing the large optimization problem created by the clauses and atoms of the MLN (such creation is described in detail in Section 2.4.1) into several, smaller optimization problems. Said smaller problems are further compacted with a technique called Cutting Plane Aggregation [90] (CPA), which groups similar groundings into a single constraint to the problem. At each iteration, CPI maintains a current, intermediate solution to the whole MLN and only those clauses that are not satisfied by the intermediate solution are compacted by the CPA algorithm and then used as constraints in the optimization problem, which creates a new, intermediate solution and so forth until convergence. The system in charge of solving these optimization problems is called Gurobi and uses both the simplex and the barrier methods presented in Section 2.2.2.

3.4.1 State of the Art for Searching

The search in the Alchemy system is performed over the whole MRF with different algorithms, depending on the type of inference being used [24, p.23]. For MAP inference, Alchemy uses a single CPU-core MaxWalkSAT implementation and for marginal inference, it uses the MC-SAT algorithm with sampling provided by the SampleSAT algorithm in a process similar to the weight learning described in 4.2.2, with the main difference being that no optimization algorithm is used. Alchemy's other inference algorithm is *lifted inference*, based on BP as described in Section 3.3.1. In this algorithm, once the factor graph has been built and compacted, the *marginal probability* of the supernodes with query atoms can be computed given the rest of the factor graph. Said computation is iteratively performed by passing *messages* from

supernodes to superfeatures and vice versa.

In order to formally introduce the passed messages, we must first define the concept of not-sum or *summary* [62], a non-standard summation notation that indicates which variables are *not* being summed over, instead of indicating those variables being summed as traditionally used in the summation notation. Formally, given a set of variables $x_1, ..., x_n$ each belonging to a corresponding domain $D_1, ..., D_n$ and a function of these variables $f(x_1, ..., x_n)$, the summary of f for a variable x_i is defined as the summation of all values of f given all possible combinations of the values of the variables except x_i, which is left at a fixed value:

$$\sum_{\sim\{x_i\}} f(x_1, ..., x_n) = \sum_{x_1 \in D_1} ... \sum_{x_{i-1} \in D_{i-1}} \sum_{x_{i+1} \in D_{i+1}} ... \sum_{x_n \in D_n} f(x_1, ..., x_n) \tag{3.2}$$

As example, consider a boolean function with 3 boolean variables $f(x_1, x_2, x_3)$ that returns 1 if at least one of its arguments is 1 and 0 otherwise. The summary for $x_2 = 0$ is:

$$\sum_{\sim\{x_2\}} f(x_1, x_2, x_3) = f(0,0,0) + f(1,0,0) + f(0,0,1) + f(1,0,1) = 0 + 1 + 1 + 1 = 3 \tag{3.3}$$

With the summary notation, the messages μ passed from a supernode sn to a superfeature sf ($\mu_{sn \to sf}$) and from sf to sn ($\mu_{sf \to sn}$) are defined as:

$$\mu_{sn \to sf} = \prod_{sg \in nbf(sn) \setminus \{sf\}} \mu_{sg \to sn}^{im(sg,sn)} \tag{3.4}$$

$$\mu_{sf \to sn} = \sum_{\sim\{sn\}} (f(nbn(sf)) \prod_{sm \in nbn(sf) \setminus \{sn\}} \mu_{sm \to sf}) \tag{3.5}$$

where $nbf(sn)$ is the set of superfeatures connected to sn (i.e., its neighbourhood),

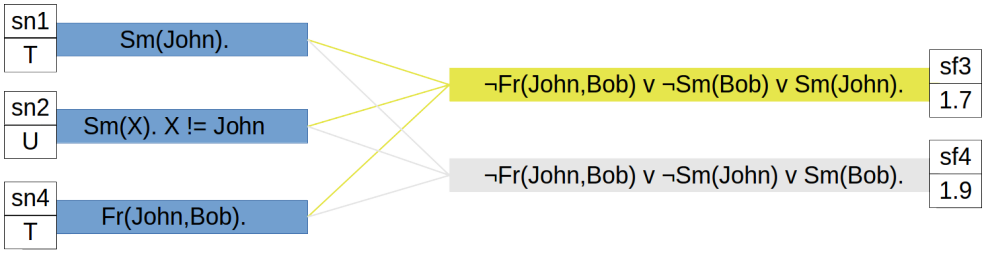


Figure 3.5: Part of the factor graph for the smokers MLN. The extra white boxes in the supernodes represent the truth value of their atoms (`T` for true and `U` for unknown) and the extra white boxes in the superfeatures contain the weight of the clauses inside them.

$nbn(sf)$ is the set of supernodes connected to sf, $im(sg, sn)$ is the number of identical messages that have to be sent to the superfeature sg, one for each time an atom of sn appears in a clause of sg, and $f(nbn(sf))$ is a function based on the clauses of the superfeature, whose arguments are the truth values of the atoms in the neighbouring supernodes, and over which the summary is applied. There are two possible return values for f: $e^{w_{sf}}$, where w_{sf} is the weight of the clauses in the superfeature, which occurs for those combinations of the truth values of the atoms in $nbn(sf)$ that make the clauses in the superfeature to be true; and 0 for those truth value combinations that make the clauses to be false or contradict the evidence.

The message passing iterations begin with the supernodes sending 1 as message to their corresponding superfeatures, the superfeatures calculate their message and send it to their corresponding supernodes, and so forth until convergence (a difficult topic as shown in [83], Alchemy usually executes BP for a fixed number of iterations). Once the algorithm has converged, the unnormalised marginal probability of all atoms in a supernode sn can be computed as:

$$\prod_{sg \in nbf(sn)} \mu_{sg \to sn}^{im(sg,sn)} \tag{3.6}$$

To summarize the process, consider the superfeatures $sf3$ and $sf4$ presented in Figure 3.5 (part of the factor graph presented in Figure 3.4). In the second iteration, after receiving 1 as message from all their neighbouring supernodes ($sn1$, $sn2$, and

$sn4$) during the first iteration, said superfeatures compute their message back to $sn4$ as follows:

$$\begin{aligned}
\mu_{sf3\to sn4} &= \sum_{\sim\{sn4\}} \mu_{sn1\to sf3}\mu_{sn2\to sf3} f(sn4, sn2, sn1) \qquad (3.7)\\
&= \mu_{sn1\to sf3}\mu_{sn2\to sf3}(f(T,F,F) + f(T,T,F) + f(T,F,T) + f(T,T,T))\\
&= 1 * 1 * (0 + 0 + e^{1.7} + e^{1.7})\\
&= 10.947\\
\mu_{sf4\to sn4} &= \sum_{\sim\{sn4\}} \mu_{sn1\to sf4}\mu_{sn2\to sf4} f(sn4, sn1, sn2) \qquad (3.8)\\
&= \mu_{sn1\to sf4}\mu_{sn2\to sf4}(f(T,F,F) + f(T,T,F) + f(T,F,T) + f(T,T,T))\\
&= 1 * 1 * (0 + 0 + 0 + e^{1.9})\\
&= 6.685
\end{aligned}$$

where T and F represent the case where the atoms in the corresponding supernode sn_i are true or false respectively. Note how the truth value for $sn4$ always remains true and how $f(T,F,T) = 0$ in Equation 3.8 even though ¬T v ¬F v T = T, because it contradicts the evidence by setting `Sm(John) = F`. With the messages $\mu_{sf3\to sn4}$ and $\mu_{sf4\to sn4}$, $sn4$ can then calculate its next messages as:

$$\begin{aligned}
\mu_{sn4\to sf3} &= \mu_{sf4\to sn4}^{im(sf4,sn4)} \qquad (3.9)\\
&= 6.685^1\\
\mu_{sn4\to sf4} &= \mu_{sf3\to sn4}^{im(sf3,sn4)} \qquad (3.10)\\
&= 10.947^1
\end{aligned}$$

where $im(sf4, sn4)$ and $im(sf3, sn4)$ are equal to one because only one message has to be sent. If there was a variable involved like X in $sf2$ (¬Sm(X) v Ca(X). X! = John),

then the number of messages for $sn3$ (`Ca(X)`) would be $n-1$, where n is all the possible values for X (i.e., all people in the evidence) and the -1 excludes `John`.

About the other systems, Tuffy implements the same algorithms as Alchemy for MAP and marginal inference, but does not include lifted inference. Tuffy was the first to improve MAP inference by using partitioning on the MRF and then processing the parts in parallel, as described above (Section 3.4). theBeast was the first to introduce the idea that MAP inference can be seen as integer linear programming problem and the first to use the CPI approach. CPI is also used by RockIt, which improved its performance thanks to the CPA preprocessing algorithm and the efficient Gurobi optimizer.

Finally, Beedkar *et al.* implemented fully parallel inference for MLNs [5]. Their system parallelises 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 parallelised using importance sampling together with Markov chain Monte Carlo [60]. Since the MLN is partitioned during grounding, no further partitioning of the resulting MRF is required before searching. The authors compared their approach against Tuffy's and found it to be more efficient since the partition is performed over a smaller data independent graph, while Tuffy must search for partition opportunities in the whole MRF. Experimental evaluation shows that this is faster and produces similar results when compared with Tuffy.

3.5 Our Approach to Learning

We chose to design and implement the most successful method for weight learning, called discriminative weight learning (presented in Section 2.5.1), on GPUs using the Diagonal Newton (DN) optimization method [70]. Said process begins (like inference) by finding the active atoms and active clauses through grounding the MLN. With the active atoms and active clauses, the MRF is created (but not partitioned like in MaxWalkSAT, as the whole search space has to be considered), an initial value for

the weights of each clause is defined and the DN method begins.

The DN method (described in Section 2.2.1) attempts to give the optimum value to each weight by starting from the initial value and iteratively computing new values, each better than the last, until the optimum is reached. This iterative "movement" towards the optimum requires the gradient and Hessian function values which are obtained by sampling over the truth values of the active atoms and active clauses using MC-SAT.

MC-SAT [94] is an iterative uniform slice sampler that uses SampleSAT starting from a different point in the search space at each iteration, in order to count the expected number of true groundings of the clauses. SampleSAT is a combination of the MaxWalkSAT algorithm with an optimization technique called Simulated Annealing (SA). The idea is to have MaxWalkSAT's ability to quickly converge towards solutions and SA's ability to uniformly sample solutions.

The problem of the DN method is that it has a rather high complexity $O(N*M*L)$, since each iteration N of the DN method has to execute M iterations of MC-SAT and each MC-SAT iteration requires L iterations of SampleSAT. Moreover, the solutions found by SampleSAT might be suboptimal as, like MaxWalkSAT, it cannot cover the entire search space. Thus, we propose to accelerate weight learning and improve its solutions by computing the grounding and the SampleSAT algorithm in the GPU (see the results in Section 5.3.2).

Algorithm 4 resumes GPU weight learning in MLNs based on DN and MC-SAT. It receives the MLN clauses `MC`, the evidence `E`, an error tolerance for the DN method `D`, and a number of iterations for MC-SAT `I` (which can also be computed automatically based on the number of active clauses). The algorithm returns the optimum weights `W` for the MLN. First, the MLN is grounded and initial value for the weights are proposed. Then, while our error in the weight values is above our threshold `D`, for `I` iterations, assign random truth values `TV` to the active clauses `AC`, apply SampleSAT (i.e., MaxWalkSAT with SA for atom flipping, which has also been simplified like in Algorithm 3) and, once SampleSAT finishes, count the number of true groundings of

each clause `AC` according to its truth value found on `TV`. When the `I` SampleSAT calls are over, use the total count of true groundings `TG` to compute the gradient `G` and the Hessian function `H`, which allow the DN method to calculate a better value for the weights `W` and move to the next iteration.

Algorithm 4 General algorithm for GPU weight learning in MLNs based on DN and MC-SAT.

```
Input: MLN clauses MC, evidence E, error torelance D,
       and MC-SAT iteration number I
Output: weights W for the clauses in MC
AA, AC <- Ground using MC and E
W <- Define an initial value to the weight of each clause in MC
While the error in W is more than D
    For I iterations
        TV <- assign random truth values to every AC
        While AC has unsatisfied clauses UC
            Flip atom AA of a clause in UC using SA or MaxWalkSAT
            Remove all satisfied clauses from UC
            UC <- Add newly unsatisfied clauses if any
        End while
        TG <- count the true grounding of AC acording to TV
    End for
    G, H <- Compute the gradient and the Hessian using TG
    W <- Update the weights using G and H
End while
Return W
```

For FOL clause learning on GPUs using ILP, the search space (i.e., the number of candidate formulas that can be proposed) is very large. Moreover, each formula has to be evaluated to compute its coverage (i.e., the number of positive and negative examples it covers). As the coverage problem can also be seen as a Datalog program where the training data is a series of Datalog facts and the proposed formula is a Datalog rule, we propose to accelerate this coverage using Datalog on GPUs. In MLN clause learning, changing the coverage function to an MLN-specific function would require grounding every proposed clause, a task which can also be solved by a Datalog program.

3.5.1 State of the Art for Learning

Alchemy is the most complete system in terms of learning as it includes generative learning and discriminative learning with several optimization algorithms (both described in Section 2.5.1) for weight learning, and top-down structure learning [57] (TDSL) and bottom-up structure learning [80] (BUSL) ILP-like algorithms for clause learning. All other systems are only capable of weight learning based on an algorithm first proposed by Alchemy like DN or conjugate gradient.

In its normal configuration, TDSL iteratively learns one literal at a time, starting from an MLN with the single atoms of the evidence as clauses (e.g., `Sm(X).` would be one of the first clauses in the smokers example). Clauses of length 2 (i.e., those with two literals) are then proposed from all possible combinations of literals, variables, and signs (e.g., the candidates to join with `Sm(X).` are `Fr(X,X).`, ¬`Fr`(X, X)., `Fr(X,Y).`, ¬`Fr`(X, Y)., `Fr(Y,X).`, ¬`Fr`(Y, X)., `Ca(X).`, and ¬`Ca`(X).). Next, the best combination is selected and clauses of length 3 are proposed and selected, repeating the process until a user-defined maximum length is reached. The best combination is selected by maximizing the weighted pseudolog-likelihood (WPLL) of the current MLN and selecting the combination with the highest value. WPLL is a function similar to the one presented in Equation 2.51 of Section 2.5.1 that better balances the contributions of each predicate regardless of its arity and is defined as:

$$logP_w^{\bullet}(X = x) = \sum_{a \in A} c_a \sum_{k=1}^{g_a} logP_w(X_{a,k} = x_{a,k} | MB(X_{a,k})) \tag{3.11}$$

where A is the set of atoms of the MLN, g_a is the number of groundings of atom a, $X_{a,k}$ is the kth grounding of a, with $x_{a,k}$ being its truth value, and $MB(X_{a,k})$ is the state of the Markov Blanket of $X_{a,k}$ (i.e., the truth values of all atoms that appear in the same clauses as $X_{a,k}$). For each proposed clause, the maximization of WPLL may create a possible bottleneck as optimization is a costly operation that has to be done several times. However, this bottleneck is avoided by employing the fast

L-BFGS optimization algorithm (explained in Section 2.2.1) with a relaxed converge goal (i.e., a larger margin of error is accepted in the solution) and with the result found in a former TDSL iteration as starting point, since said point should be close to the solution of the current problem. These two changes to the L-BFGS algorithm allow it to rapidly optimize the WPLL function, usually in just a few iterations.

One problem of the TDSL method is that a large search space must be covered with little guidance, often resulting in the creation of suboptimal clauses. In contrast, BUSL attempts to guide the search by construction templates based on the training evidence. These templates are conjunctions of predicates that are constructed by setting a head predicate for the template from an atom of the evidence and then including other atoms that share constants with the head predicate in the template, abstracting any similar constants into similar variables. For example, if we have `Sm(John).` and `Fr(John,Bob).` as evidence, the head predicate could be `Sm(X)` and since `John` is an argument in both predicates `Sm` and `Fr`, it can be included into the template as `Fr(X,Y)`. With the templates, the search for clauses can be limited to candidates that include the head predicate of the template and some or all other predicates in the template in positive or negative form and said candidates can also be evaluated with the L-BFGS method using the WPLL function.

Alchemy can also handle unsupervised learning [95], where the truth value of some predicates is not known in the training data (i.e., information is incomplete). Unsupervised weight learning can be done generatively using pseudo-log-likelihood (as presented in Equation 2.51 of Section 2.5.1), however the counts of true groundings $tg_i()$ for the gradient cannot be counted exactly for those predicates with unknown truth values and must instead be approximated through inference over the MLN. Unsupervised discriminative learning can be computed by minimizing the negative of the conditional log-likelihood (CLL) as shown by Equation 2.53 of Section 2.5.1, but the gradient is based on two expectation instead of one, the expected number of times the ith MLN formula is true based on the unknown query atoms Z given the evidence X ($E_{w,z}$) and the same expectation but this time based on both Z and the

known query atoms Y ($E_{w,y,z}$) also given the evidence:

$$\frac{\partial}{\partial w_i}(-logP_w(Y = y|X = x)) = E_{w,z}(tg_i(x, y, z)) - E_{w,y,z}(tg_i(x, y, z)) \tag{3.12}$$

Both the gradient and the Hessian function, which is now the difference of two matrices, one when only the unknown query atoms vary their truth values and the other when both query atoms vary said values, can be approximated using MC-SAT, but the optimization algorithm for the CLL must consider that the function may no longer be convex.

Predicates can also be learnt with unsupervised learning through statistical predicate invention based on the Multiple Relational Clusterings (MRC) algorithm [58], which attempts to discover new properties from the data based on current properties, using a probabilistic approach. These new properties can be used to cluster the data in multiple clusters that can be represented as MLN predicates. The basis of the MRC algorithm is a second-order MLN were variables can range over both predicate and constant symbols. Given a symbol x, the first set of rules of said MLN (many similar rules are used, one for each time a predicate variable requires quantification) state that x must belong to at least one cluster γ:

$$\forall x \exists \gamma \quad \texttt{such that} \quad x \in \gamma \tag{3.13}$$

Since the truth value of these rules cannot be false in the resulting world of the MLN (i.e., just like normal FOL formulas, they cannot be violated), they are assigned an infinite weight. The second set of rules, called the *mutual exclusion* rules, also have infinite weights and state that a symbol can only belong to a single cluster:

$$\forall x, \gamma, \gamma' \; x \in \gamma \wedge \gamma \neq \gamma' \Rightarrow x \notin \gamma' \tag{3.14}$$

If r is a predicate symbol belonging to cluster γ_r and $x_1, ..., x_n$ are r's argument symbols that belong to clusters $\gamma_1, ..., \gamma_n$, then the predicate $r(x_1, ..., x_n)$ is in the combination of clusters $(\gamma_1, ..., \gamma_n)$. The next rules with infinite weights state that each predicate belongs to *exactly one* combination of clusters:

$$\forall r, x_1, ..., x_n \exists (y_1, ..., y_n) \quad \texttt{such that} \quad r \in \gamma_r \wedge x_1 \in \gamma_1 \wedge ... \wedge x_n \in \gamma_n \tag{3.15}$$

The rules in the last set are called the *atom prediction* rules and state that the truth value of a predicate is determined by the single cluster combination it belongs to:

$$\forall r, x_1, ..., x_n, y_1, ..., y_n \; r \in \gamma_r \wedge x_1 \in \gamma_1 \wedge ... \wedge x_n \in \gamma_n \Rightarrow r(x_1, ..., x_n) \tag{3.16}$$

Equation 3.16 is repeated for each combination $(y_1, ..., y_n)$ and each repetition k has its own weight w_k based on the number of true t_k and false f_k atoms of the combination given the evidence:

$$w_k = log(\frac{t_k + \beta}{f_k + \beta}) \tag{3.17}$$

where β is a smoothing parameter. The atom prediction rules are the key of the MRC algorithm, since they allow the prediction of the probability of query atoms given the symbol clusterings that compose each atom. Said rules together with the mutual exclusion rules, allow the prediction of the clusterings for evidence atoms and the probability of the query atoms given the evidence.

The MRC algorithm is a recursive algorithm which is divided in two: a top level that finds possible clusterings for the atoms and a bottom level that constructs the clusters. MRC receives the evidence, an initial set of clusterings γ_r of predicates

grouped by arity and type, and a set of clusterings γ_i of constants by type. At each iteration, MRC creates a cluster for each clustering it received and then searches for better clusters and clusterings, first by creating or removing clusters, followed by moving symbols between clusters, merging or splitting clusters. Said operations are performed greedily and with restarts, always striving to increase the probability of the second-order MLN, computed using MAP inference. Once good clusters and clusterings are found, the algorithm calls itself again with each possible combination of the new formed clusters and clusterings, repeating the process until the probability of the second-order MLN does not increase with any new combination.

As an example of how MRC works, consider that our initial knowledge consists of a group of people (like `person(John)`, `person(Bob)`, and in general `person(X)`), along with their hobbies (like `hobby(John,reading)`, and in general `hobby(X,Y)`) and profession (`profession(X,Z)`). The initial predicate clusterings for the input of the MRC algorithm would be $\gamma_{r1} =$ `person`, $\gamma_{r2} =$ `hobby`, and $\gamma_{r3} =$ `profession`, and the symbol clusterings would be $\gamma_x =$ `X` (a cluster for all constants of type *person*), $\gamma_y =$ `Y` (constants of type *hobby*), and $\gamma_z =$ `Z` (constants of type *profession*). MRC would then construct six possible initial clusterings (3 for all γ_r, 1 for γ_x, γ_y, and γ_z) and start the search for better clusters and clusterings. Next, suppose that a good clustering has `hobby` in cluster γ_p, but has divided its variable `X` in clusters γ_{a1} and γ_{a2}, and its variable `Y` in clusters γ_{b1} and γ_{b2}, then the MRC algorithm must be recursively called with all combinations: $(\gamma_p, \gamma_{a1}, \gamma_{a2})$, $(\gamma_p, \gamma_{a1}, \gamma_{b1})$, $(\gamma_p, \gamma_{a1}, \gamma_{b2})$, and so forth (other atoms may also recursively call MRC, not just `hobby`). At the end, the algorithm should find some interesting clusterings like groups of people who are *friends* because they share the same hobby or groups of *coworkers* that share the same profession.

3.6 Summary

This chapter described the motivation behind our proposal of processing MLNs in GPUs by giving an overview of the current MLNs systems, a synopsis on GPUs, the possible design issues of our proposal, and our approach to solving them. GPUs are many-core processors with high data-throughput that have a wide array of applications in several fields like bioinformatics, finance, etc., and are programmed using C-like functions called kernels that are written in the CUDA language and executed by many threads.

GPU performance can be affected by several issues, programmers should always strive to: handle all data transfers in *bulk* and minimize the number of such transfers; ensure that threads are always busy and do not process to much conditional code (if, while, etc.); and use good memory access patterns like having consecutive threads performing reads on consecutive memory locations, without exceeding the total amount of GPU memory. In MLNs, the main issue of the grounding step is deciding the approach to follow: *top-down*, which is similar to Prolog's and finds solutions one element at a time or *bottom-up*, which works on whole groups of elements at a time. For our proposal, we consider bottom-up to be a better approach. The main issue of the other MLNs steps is whether they can be parallelised or not. We prove that they can indeed be parallelised by proposing two MLN platform designs called GPU-Tuffy and GPU-RockIt.

Our approach to grounding in MLNs is based on Datalog [116, 117], a FOL language similar to Prolog. Datalog processing is performed bottom-up and requires the computation of RA operations, extended with negations, comparisons, among others. Said operations can be parallelised using GPUs but there are several issues that most be addressed like string handling, data transfers, and more. The first MLN systems solved the grounding problem using an inefficient, top-down, Prolog-like approach. Modern systems consider the grounding as a set of SQL queries and answer them using a RDBMS, producing the results much faster. For some MLNs,

grounding can be avoided by using lifted inference [109].

Our search approach in GPU-Tuffy is based on solving a satisfiability problem using MaxWalkSAT. MaxWalkSAT [53] is an iterative algorithm that tries to satisfy (i.e., make true) as many clauses as possible by selecting an atom of an unsatisfied clause and flipping its truth value (i.e., changing its value from true to false or vice versa). The CPU version flips one atom at each iteration, requiring millions of iterations to solve a large MLN. We propose the use of GPU threads to flip multiple atoms at the same time and note the issues of doing so. Our approach in GPU-RockIt is based on solving an optimization problem by splitting it into smaller problems and then solving them using the Gurobi [149] solver. The Alchemy and Tuffy systems are also based in satisfiability, with Tuffy being faster thanks to its partitioning of the problem. theBeast and RockIt are based on integer linear programming, but RockIt further simplifies the optimization problems and its solver is better. Some MLNs can benefit from lifted inference, which solves the search step using BP [109].

We propose a parallel weight learning approach using the DN method [24, p. 49]. DN computes the optimum of each weight by starting from the initial value and iteratively computing better values. To adapt the DN parameters, the MLN must be grounded in parallel as explained in Section 3.3 and then, at each DN iteration, the grounding must be sampled with the MC-SAT and the SampleSAT algorithms [94]. Since SampleSAT is essentially an extension of the MaxWalkSAT algorithm, it was also parallelised in GPUs. For FOL clause learning, we propose the use of an ILP system where formula coverage, the most demanding task, is parallelised in the GPU. While Tuffy, RockIt and theBeast can only learn weights, Alchemy includes several algorithms for both weight and clause learning. MLN clause learning in Alchemy is based on TDSL [57] and BUSL [80], which are similar to ILP, but the evaluation is based on likelihood. Learning can also be performed with incomplete information, using modified functions for weight learning and statistical predicate invention [58] for clause learning.

Table 3.1 resumes some of the characteristics of the four most relevant, CPU-based

Table 3.1: Characteristics of the most relevant MLN systems.

System	Language	Grounding	Search	Parallelism	Particulars
Alchemy (2012)	C++	Prolog-like	SAT	None	The only capable of clause learning.
theBeast/theppl (2013)	Scala	Prolog-like	Optimization	None/Modularity	Different syntax compared to the other systems.
Tuffy (2014)	Java	PostgreSQL	SAT	Multicores (search only)	Solves many of Alchemy's shortcomings.
RockIt (2015)	Java	MySQL	Optimization	Multicores	Currently the fastest CPU system.
GPU-Tuffy (2016)	Java & CUDA	Datalog	SAT	GPUs Multicores (search only)	Our system based on Tuffy.
GPU-RockIt (2016)	Java & CUDA	Datalog	Optimization	GPUs (grounding only) Multicores	Our system based on RockIt.

MLN systems (Alchemy, theBeast/theppl, Tuffy, and RockIt) and our two proposed, GPU-based systems (GPU-Tuffy and GPU-RockIt), including: the *year* of publication for the latest version; the programming *language* which is built upon; the type of *grounding* used, which can be based on top-down Prolog, on SQL with a RDBMS, or on Datalog queries; the *search* approach adopted, where some systems prefer to formulate the search problem as a satisfiability problem and others prefer to formulate it as an integer linear programming problem; the *parallelism* used, where Alchemy and theBeast use none, theppl allows programmers to design their own parallel modules with any type of parallelism, Tuffy parallelises only the search, RockIt is fully parallel, GPU-Tuffy is fully GPU-parallel and uses GPU and CPU together during the search, and GPU-RockIt parallelises grounding on the GPU and the rest on multicores; and finally, some of the *particulars* of each system, like Alchemy being the only system capable of learning both weights and clauses (other systems can only learn weights), theBeast whose MLN syntax specification is different from the rest of the systems, Tuffy was created following Alchemy's approach, but includes many upgrades that speed-up MLN processing, RockIt is currently the fastest CPU system thanks to its optimizations and multicore parallelism, and GPU-Tuffy along with GPU-RockIt are our systems described in detail in Chapter 4.

Chapter 4

Experimental Platforms

In Chapter 3 we presented our approach to process on GPUs the main phases of MLN processing: inference and learning. However, in order to have a fully functional platform, our designs must be integrated with an input/output system, a First-Order Logic (FOL) language compiler, among other components. Although it is possible to develop all the components of an MLN platform, this endeavour is beyond the scope of our research. Hence, we set out to design two fully functional, GPU parallel MLN platform using existing CPU platforms as their basis.

Our platforms are thus the integration of several components working together to process MLNs: with Tuffy [88] and RockIt [90] serving as the base, the core components of said platforms are *GPU-Datalog* [74] (Section 4.1), our parallel Datalog engine, and *GPUSATLIB* (Section 4.2), our parallel library of satisfiability solvers. Our GPU parallel ILP system (Section 4.3) works well for FOL clause learning and will be a future component for MLN clause learning on our platforms.

We named our platforms *GPU-Tuffy* (Section 4.4) and *GPU-RockIt* (Section 4.5) after their base systems and validated them, along with their core components, using several tests (Section 4.6). We chose to have two platforms in order to test the flexibility of our grounding design, which is included in both platforms with minimal changes between them, and because some MLN applications are better modelled as a satisfiability problem like Tuffy does, while others are better modelled as optimization

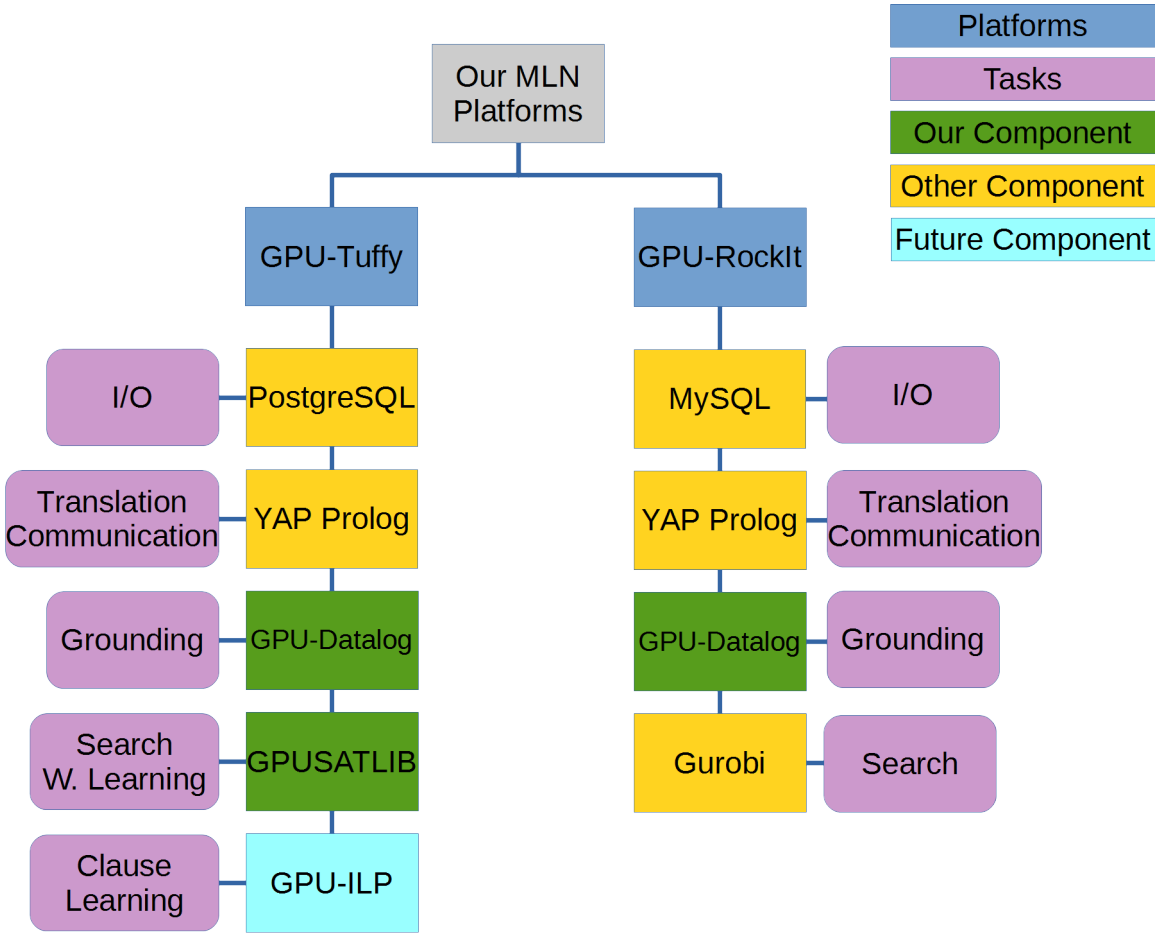


Figure 4.1: Our GPU-based platforms and their components along with their functions.

problems like RockIt does (refer to Section 5.4 for more details). Figure 4.1 shows the core components of our platforms, distinguishing between those we created, those we simply use, and our ILP system for GPUs which we hope to include for MLN clause learning. Also, each component indicates the tasks that it performs (whose details are in their corresponding sections for our components and on the description of GPU-Tuffy and GPU-RockIt for the other components).

It is worth to mention that the complex integration of all elements was possible thanks to the modular well designed APIs found in both the base systems and the core components. Also, all components involved (except the Gurobi [149] solver) and our platforms are free, open software[1].

[1]GPU-Datalog is available at `https://github.com/vscosta`, other systems coming soon.

4.1 GPU-Datalog

GPU-Datalog [74] is our parallel, bottom-up engine for the Datalog language based on GPUs. Its latest version currently has over 7,000 lines of C and CUDA code and its main components are shown in Figure 4.2. GPU-Datalog was integrated as a module into the YAP Prolog system [15] in order to bridge the database model to the GPU world and represents a hybrid solution where both GPU and CPU are used. Highly parallel code (i.e., relational algebra operators) is executed on the GPU while sequential code (i.e., input/output operations, control) is executed on the CPU. Our engine is organized into three stages (*P*, *E*, and *T* in Figure 4.2) with a single *host* thread executing the first and third stages, and scheduling work to the GPU during the second stage.

The *Preparation* stage begins with YAP reading and compiling the Datalog program into a numerical representation (NR) suitable for GPU processing where each unique string is assigned a unique integer Id. 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. Once compiling is completed, YAP sends the NR to GPU-Datalog and its *Preprocessor* analyses each rule to determine which operations to perform and over which tables and columns said operations will be performed. Finally, the query is analysed to determine which rules are to be evaluated. A queue is created with these rules.

The *Evaluation* stage takes the rule queue and evaluates each rule one at a time. If the rule has only one normal predicate (*a1*), any reductions (i.e., selections and comparison predicates) required are performed first (*b1*), followed by any selfjoins (*c1*) and a single projection based on the variables in the head of the rule (*d1*). Finally, any extra operations (i.e., arithmetic or aggregation predicates) are performed (*e1*) and the result is returned (*f1*).

When the rule has two or more normal predicates, the first two predicates are taken (*a2*) and any necessary reductions (*b2*) and selfjoins (*c2*) are performed. Next,

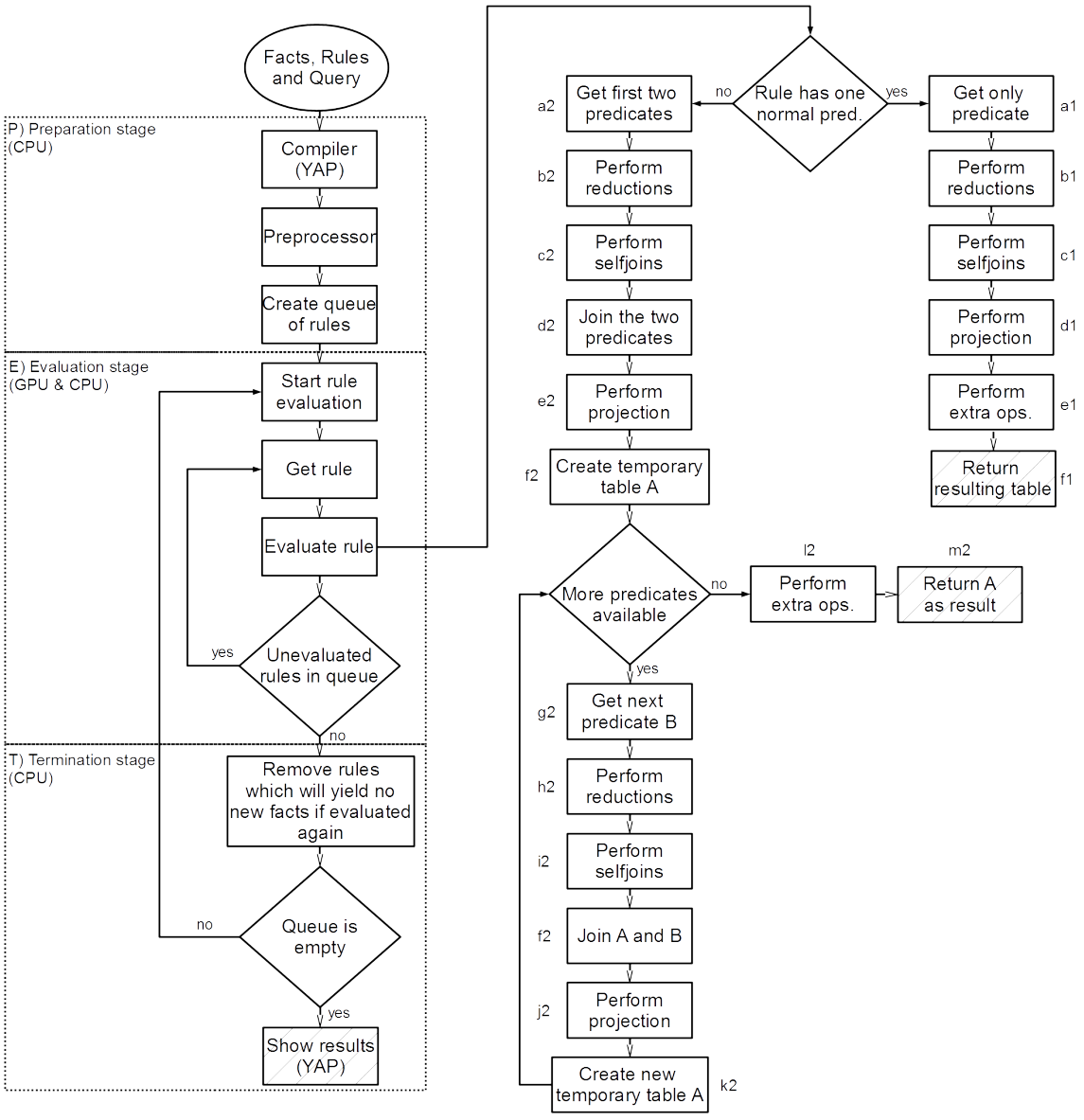


Figure 4.2: GPU-Datalog engine organisation.

the two predicates are joined (*d2*, using single or multijoin depending on the variables in the predicate) and a projection is performed (*e2*) to discard columns that are not going to be used in further operations and to create a temporary table A with the result (*f2*). When there are more normal predicates in the rule, we take the next predicate B (*g2*), perform any needed reductions (*h2*) and selfjoins (*i2*), and then it is joined with the temporary table A (*f2*). We then perform a projection to create a new temporary table A (*k2*) that either has to be joined to the next predicate B (*g2* again) or it is used to compute any extra operations (*l2*) before being returned as result (*m2*). Finally, once we have the result of the rule evaluation, duplicate elimination is performed to reduce memory and computation requirements.

When all the rules in the queue have been evaluated, we proceed to the *Termination* stage. This stage removes from the queue those rules which will not yield new facts if evaluated again. After the finished rules are removed, if the queue does not become empty, each rule still in the queue is evaluated again. This process is repeated until the rule queue becomes empty. Once the queue is empty, the queries are answered by taking all the rule results required by the queries and performing any selections and/or projections specified by the variables and constants in said queries. Finally, the query results are returned to YAP to be translated back into strings and displayed.

About rule removal from the queue, rules composed only of fact predicates finish their evaluation in the first iteration (and are thus removed), while those that include other rule predicates are removed after the first rule representing one of its predicates has finished. For example, if we have the facts `f1(1,2,3).` and `f2(2).`, along with rules `r1(X,Y) :- f1(X,Y,Z).`, `r2(Y) :- r1(X,Y), f2(Y).`, and query `r2(Y)?`, then `r1` finishes in the first iteration because all that needs to be done is project `f1` to leave `r1(1,2)` as the result of the rule, while `r2` does nothing on this iteration because the value of `r1` is still empty. On the second iteration the queue has only one rule `r2`, said rule can now calculate its result which would be `r2(2).`, and at the end, it would also be removed from the queue because one of its predicates

(`r1`) will not produce more results, thus finishing the evaluation. Special cases like self-recursive rules (e.g., `r(X) :- f(X,Y), r(Y)`) are removed when they reach a fixed point or an orbit of at most 5 points (i.e., they return the same result in two consecutive iterations or a series of results that repeat after at most 5 iterations).

In order to minimize the number of data transfers between GPU and CPU memory, each time a table is necessary to perform an operation, the request is handled through our memory management module. Its purpose is to maintain facts and rule results in GPU memory for as long as possible. To do so, it keeps track of GPU memory available and GPU memory used, and maintains a list with information about each fact and rule result that is resident in GPU memory. Such information includes the fact or rule result represented, its size and the pointer to its location in GPU memory.

When some data (facts or rule results) is requested to be loaded into GPU memory, its entry is first looked up in that list. If found, said entry in the list is moved at the beginning of the list; otherwise, memory is allocated and a new list entry is created at the beginning of the list for the data. In either case, the address of the data in GPU memory is returned. If allocating memory for the data requires deallocating other facts and rule results, those at the end of the list are deallocated first until enough memory is obtained — rule results are written to CPU memory before deallocating them. By doing so, most recently used fact and rule results are kept in GPU memory. This approach is very similar to the Demand Paging and Not Recently Used Page Replacement algorithm described in [114]; but our approach works with memory sections as opposed to pages.

As an example, consider the rule `a(X):- b(X,Y), c(Y,Z), d(Z).`, where `b` and `c` are joined first so they are loaded in order into GPU memory and registered into our memory management module. The result of said first join (we will call it `r1`) remains in GPU memory and is also registered by our module (leaving 3 memory chunks `b, c, and r1` registered in the module). Next, the join between `r1` and `d` is to be performed, but our GPU does not have enough memory to load `d`. Our memory management module detects this insufficiency problem and removes `b` from both GPU

memory and the module's registry (note that `b` is deleted and not returned to CPU memory because it is not a rule result). If there is enough GPU memory, then `d` is loaded, otherwise `c` is also removed before loading `d`. Finally, the join is performed and the result of the whole rule is obtained.

4.1.1 GPU Operators

Next, we present the operators of the original GPU-Datalog [74], followed by the new additions required for MLN processing like negation and aggregation. These operators use some support functions like sort, scan (also called prefix sum), and duplicate elimination, which is based on sorting and then removing all but one duplicate element with the unique function. We use the efficient implementations of these functions provided by the Thrust library [151]. This library is a C++ template library for GPUs based on the Standard Template Library [86] and provided as part of CUDA. Also, our operators include several optimizations not part of the Datalog standard, aimed at improving performance. The most important optimizations are executing a *projection* at the end of each join (as shown in *j2* of Figure 4.2), allowing us to discard unnecessary columns earlier in the computation of a rule, and *fusing* operators by applying two or more operators to a data set in a single read of the data set, as opposed to applying only one operator, which involves as many reads of the data set as the number of operators to be applied.

Note that our operators are implemented as one (e.g., all arithmetic operators) or three (e.g., selections and joins) kernel calls depending on whether the size of the solution is known beforehand or not.

Original GPU-Datalog operators

The first original operator, *selection*, uses three different kernels. The first kernel marks in a new array all the rows in a table that satisfy the selection predicate with a value of one. The second kernel performs a prefix sum on the marks to determine the size of the results buffer and the location where each GPU thread must write the

results. The last kernel writes the results.

Projection requires little computation, as it simply involves one kernel taking all the elements of each required column and storing said elements in a new memory location. While it may seem pointless to use the GPU to move memory, the higher memory bandwidth of the GPU, compared to that of the host CPU/s, and the fact that the results remain in GPU memory for further processing, make *Projection* a suitable operation for GPU processing.

Our Datalog engine uses three types of *Joins*: *Single join*, *Multijoin* and *Selfjoin*. A *Single join* is used when only two columns are to be joined, e.g., table1(X,Y) $\bowtie_Y$ table2(Y,Z). A *Multijoin* is used when more than two columns are to be joined: table1(X,Y) $\bowtie_{(X,Y)}$ table2(X,Y). A *Selfjoin* is used when two columns have the same variable in the same predicate: table1(X,X).

Algorithm 5 Our modified version of the INLJ for single joins.

```
Create an array with the elements of one of the columns to be joined
Sort the array
Create a CSS-Tree using the sorted array
Search the tree to determine the join positions
Do a preliminary join to determine the size of the result
Do a second join to write the result
```

Single join uses a modified version of the Indexed Nested Loop Join (INLJ) described in [48], which is presented in Algorithm 5. The CSS-Tree [97] (Cache Sensitive Search Tree) is a special B+-Tree that is very adequate for the GPU because it can be quickly constructed in parallel and because tree traversal is performed via address arithmetic instead of the traditional memory pointers. While the tree allows us to know the location of an element, it does not tell us how many times each element is going to be joined with other elements nor in which memory location must each thread write the result. Hence, we must perform a "preliminary" join. This preliminary join counts the number of times each element has to be joined and returns an array that, as in the select operation, allows us to determine the size of the result and the write locations when a prefix sum operation is applied to this array.

With the size and write locations known, a second join writes the results.

To perform a *Multijoin* operation, for example table1(X,Y) $\bowtie_{(X,Y)}$ table2(X,Y), we first take any column to be joined (say X in table1) and create and search in the CSS-Tree as described above. Then, in the first join, after performing the counting but before writing the result to global memory, we check if the values of the remaining columns are equal (in our example we check if Y = Y) and reduce the count accordingly to discard the rows that are not equal. In the second join, since we only know the count of the number of times a row has to be joined, but not to which other rows it must be joined, we check the additional join columns again to decide if we write the row join or not.

The *Selfjoin* operation is very similar to the *Selection* operation. The main difference is that, instead of each thread checking a constant value on its corresponding row, each thread checks if the values match on the columns affected by the *Selfjoin*.

Built-in Comparison Predicates

Built-in comparison predicates ($<, >, <>, =, >=, <=$) are similar to the *Selection* and *Selfjoin* operations, i.e., they use a pipeline of three kernel executions. The first kernel marks all the rows that satisfy the comparison predicate. The second kernel performs a prefix sum on these marks to determine the size of the result buffer and to use them as the indexes where each GPU thread must write its result. The third kernel writes the results. The difference is that comparison predicates use the given operator instead of always testing for equality. Also, comparison predicates can compare columns against a constant value ($Y > 3$) or against another column ($Y > Z$).

In the first version of GPU-Datalog, these predicates used be to performed at the end of each clause evaluation [75]. It was a simple, restriction-free evaluation, but inefficient, as many tuples that could be eliminated earlier in the evaluation were removed only at the end.

Comparison predicates are now evaluated as soon as their variables are bound,

```
ORIGINAL RULE:   r(Z, X) :- f1(Y, X),f2(Z, X),f3(Z, W),X > 2,f4(W).
OLD GPU-Datalog: r(Z, X) :- f1(Y, X),f2(Z, X),f3(Z, W),f4(W),X > 2.
NEW GPU-Datalog: r(Z, X) :- f1(Y, X),X > 2,f2(Z, X),f3(Z, W),f4(W).
-----------------------------------------------------------------------
SQL:          SELECT C1 FROM t WHERE C1 > 3 OR C2 < 1;
CONJUNCTIONS: res(C1) :- t(C1, C2), C1 > 3.
              res(C1) :- t(C1, C2), C2 < 1.
DISJUNCTIONS: res(C1) :- t(C1, C2), C1 > 3, C2 < 1.
```

Figure 4.3: Improved processing of comparison predicates in GPU-Datalog. The upper part represents how comparisons are now performed as soon as possible and the lower part how handling disjunctions simplifies translating SQL queries to Datalog.

i.e., as soon as the variables appear in a positive predicate, allowing us to eliminate the computational cost and memory usage of tuples that are not to be used in the rest of the rule. This was achieved by evaluating any possible comparison predicate before performing each *join* in a clause (similarly to how *selections* and *self-joins* are performed before a join). The upper part of Figure 4.3 shows an example of how we used to evaluate comparison predicates and how the new version of GPU-Datalog evaluates them.

Also, it is now possible to specify if a clause should evaluate comparison predicates as it normally would using conjunctions or as disjunctions. Disjunctions simplify specifying some SQL queries and greatly improve their processing by reducing the number of Datalog clauses required to represent said queries. The lower part of Figure 4.3 illustrates this simplification: as conjunctions the SQL query requires two clauses and their results may include many repeated tuples (those where both $C1 > 3$ and $C2 < 1$ are true).

Arithmetic predicates

These predicates $(+, -, *, /)$ are performed after all joins and comparison predicates are completed (*e1* and *l2* in Figure 4.2). Since the result size is always equal to the input size, arithmetic predicates can be executed in a single kernel instead of the 3 kernels common in other operations. Thanks to YAP's syntactic analysis of rules, these predicates can be written in the usual infix notation and are automatically

translated to postfix notation for easier evaluation. As an example on how to write arithmetic predicates, consider the rule:

```
result(Z,W) :- fact1(X,Y), Z is X + 3, W is Y / 5.
```

where the operations $X + 3$ and $Y/5$ will be internally translated to $X\,3+$ and $Y\,5\,/$, solved, and the results stored in Z and W respectively.

Aggregation and related operations

Aggregation (GROUP BY) is commonly used with other operations like *summation* (SUM), *count* (COUNT), *average* (AVG), among others. Aggregation indicates a series of columns whose combined row values will determine the result of the related operations. They were implemented by adding a dummy predicate called *aggregation* that simply indicates the required variables. The related operations are represented by predicates with two values: the variable representing the input column and the one representing the output column.

The related operations are performed by sorting the data according to the aggregation variables and then using Thrust's *reduce_by_key* operation with the sorted variables as keys. Note that *reduce_by_key* always performs a summation over the data. However, *count* can be implemented by creating an additional column filled with 1's and performing the summation on said column. The last operation, *average*, can be implemented by performing the summation on the required columns and then dividing the result by the result of the *count*. As an example, consider the rule:

```
result(X,Y,W) :- fact1(X,Y,Z), aggregation(X,Y), sum(Z,W).
```

with *fact1* having the following values:

```
X       Y       Z
1       2       3
1       1       5
1       2       7
```

the result of the rule after performing a *summation* over column `Z` and aggregating over `X` and `Y` is in column `W`:

```
X       Y       W
1       2       10
1       1       5
```

Negation

One of the most important new additions to GPU-Datalog is the ability to evaluate negated literals in the body of a clause. This allows the processing of new programs that would have been impossible otherwise. At source level, negation is similar to our *join* operators: we know which columns must be compared based on their same variables and the result is a temporary table with all the rows of the positive data whose columns were not matched in the negative data. For example, consider the clause `t(Y) :- r(Y), not s(X,Y)` with `r(1), r(2), s(3,2), s(1,4)`, which will compare column one of the positive data r and column two of the negative data s, removing `r(2)` and leaving `r(1)` as result.

Evaluation begins by taking an appropriate column from the positive data. The column is sorted and a CSS-Tree constructed with it. Using the CSS-Tree, we launch a kernel that can efficiently traverse through the data in order to mark which tuples are to be removed. Next, we perform a prefix sum on these marks to determine the size of the result buffer and to use them as the indexes where each GPU thread must write its result. The final kernel writes the results.

4.1.2 Data partitioning

To handle large amounts of data that exceed the memory capacity of the GPU and to allow multiple GPU to process an application simultaneously, we implemented a first version of a data partitioning scheme adapted for our operators. For all operations except joins, the partitioning scheme loads into the GPU the first n rows of data,

performs the operation, unloads the result to CPU memory, removes the processed rows from GPU memory and loads the next n rows. These steps are repeated until there is no data left to process.

The join operations follow a similar procedure where both tables to be joined are partitioned. However, the join operations require more processing steps as each partition of a table needs to be joined with all partitions of the other table (e.g., if table1 has partitions A and B, and table2 has partitions C and D, we need to perform 4 joins A with C, A with D, B with C, and B with D). Thus, for joining two tables say X and Y, the partitioning scheme loads the first m rows of both tables, performs the join, unloads the result, removes the processed partition of X and loads the next m rows of X. When all rows of X are processed, the steps are repeated with the next partition of Y and once again, all partitions of X. The join finally ends when all partitions of Y are processed.

Of great importance to the scheme are the choices of n and m. If they are too small, performance is adversely affected as we need to perform many more iterations. If they are too large, the result of the operation may not fit and the whole process will fail. A safe choice would be to consider the worst-case scenario: not a single row is removed in selections, comparisons, and other operations, thus the size of the result is equal to the size of the input; and, in the case of join operations, each row in one table joins with all rows of the other table (a Cartesian join). Hence, a safe value for n would be approximately (we say approximately because some memory is necessary for auxiliary arrays) half the amount of GPU memory available, leaving the other half for the result. Considering two tables with an equal number of columns and that the result will include all rows from both tables, the safe value for m can be computed by adding m twice (because we are loading two tables) and the size of the worst-case result which is m^2. This addition should be equal to the total amount of GPU memory t and that gives us a quadratic equation $m^2 + 2m = t$, whose positive root $\sqrt{t+1} - 1$ equals the approximate safe value for m.

However, worst-case scenarios rarely occur and most of the time we would be

performing more iterations than necessary. Thus, we devised a scheme where partition sizes can change at runtime and each operator can write its result in batches if necessary. The idea is to start with partitions sizes larger than the worst-case and then, once an operator has calculated the size of the result but before it is written (after the second kernel as explained above), we check if the result would fit in the available amount of GPU memory. If the result fits, the operator continues normally with the writing and the partition size is maintained if the result fit with little memory left unused or it is increased if plenty of memory was still left. When the size of the result does not fit, then the operator must write its result in two or more steps. At each step, the operator writes the maximum possible number of elements in the available amount of GPU memory and said write result is then flushed to CPU memory, liberating GPU memory for the next step. This process is repeated until the whole result is written and the operator finishes by indicating to the system's control flow that the result is in CPU memory and that the partition size should be decreased.

4.1.3 The GPU-Datalog Architecture on Multicores

Our GPU-Datalog engine was ported to multicores using OpenMP [143]. The CPU version of GPU-Datalog has some 1,000 lines of code, uses the same YAP interface, and it is based on the same relational algebra operations, with one major difference: instead of using 3 function calls (kernels) for selections and joins, we decided to capitalize on the CPU's flexible memory scheme by using size-changing arrays (i.e., *vectors* and *lists* found in the Standard Template Library [140]). With these arrays, each thread can write a resulting tuple as soon as it is found, and the final result is given by joining all the arrays. This reduces the number of function calls to a single call, and therefore reduces processing time.

While this CPU version is suitable for Datalog and ILP programs, it lacks some of the latest operations recently included in GPU-Datalog like negation and improved comparison predicates and thus, said CPU version is currently not being used in MLN

processing.

4.1.4 State of the Art for Datalog

As presented in [72], Datalog has a wide array of applications [50] and many of those applications extend the language with new ideas like special predicates or values over predicates, showing the flexibility of the language. Some of the most interesting applications include:

- **Data Integration** is the combining of heterogeneous data sources into an unified query and view schema. In the work of Green et al. [45], Datalog is used to calculate provenance information of datasources. In another work by Lenzerini [65], the power of Datalog to express queries and views of heterogeneous data is compared against other languages with very good results.

- **Declarative Networking** is a programming methodology to specify network protocols and services using high-level declarative languages. Boon Thau Loo et al. [68] proposed *NDlog*, an extension of Datalog with networking specifics such as distribution, linklayer constraints. *Overlog* [69] is another Datalog extension that implements the soft-state approach common in network protocols. The idea is that data, represented as predicates, has a lifetime in seconds attached to each predicate and has to be refreshed when the lifetime expires.

- **Program Analysis** is the automatic analysis of computer programs to ensure correctness or find potential optimizations. Martin Bravenboer and Yannis Smaragdakis [10] implemented the *Doop* framework, a points-to analyser for Java based on Datalog that determines "What objects can a program variable point to?". By exploiting Datalog recursion, their analyser was 15 times faster compared to other well-known analysers.

- **Information Extraction** is the automatic extraction of structured information from documents, web pages, etc. Lixto is a web data extraction project by

Gottlob et al. [40] based on *Elog*, an extension of Datalog with conditions to detect false positives while extracting data, among other things, making it an efficient data extraction language. Also, Shen et al. [106] created a Datalog extension called *XLog*, whose information extraction rules are smaller and easier to understand compared to those of other systems. They also introduced procedural predicates that perform some computations over the arguments using Java or C++ and return the result back to Datalog.

- **Network Monitoring** is the continuous analysis of a computer network to obtain traffic information, component failure, etc. To this end, Abiteboul et al. [1] created a distributed extension of Datalog called *dDatalog*, which distributes its rules over the peers in the network according to the information each of them possesses.

- **Security.** Marczak et al. [71] implemented *SecureBlox* a distributed query processor with security policies. This Datalog extension allows, among other things, the declaration of integrity constraints like functional dependencies. Trevor Jim [51] created the Secure Dynamically Distributed Datalog (SD3) platform. SD3 extends Datalog's predicates with an additional value that determines who is in control of the predicate. Thus, a predicate will be true only if its controller says it is true. The advantages of this system over others are its high-level language and its ability to quickly create security policies from scratch or by modifying existing ones.

- **Cloud Computing** is the execution of programs over many computers connected in a network. Alvaro et al. [4] presented a distributed data analytic stack implemented using Overlog. It allows Java objects to be stored in tuples and Java functions to be called from Datalog. Their system was tested against Hadoop [135] showing a slightly worse but still competitive performance.

Similar to our GPU-Datalog engine, He *et al.* [47] have designed, implemented and evaluated GDB, an in-memory relational query co-processing system for execution on

both CPUs and GPUs. GDB consists of various primitive operations (scan, sort, prefix sum, etc.) and relational algebra operators built upon those primitives.

We modified the INLJ of GDB for our single join and multijoin (presented in Section 4.1.1), so that more than two columns can be joined (i.e., the multijoin case), and a projection performed, at the same time. Their selection operation and ours are conceptually similar too; but ours were created from the beggining to takes advantage of GPU shared memory and uses the prefix sum of the Thrust Library. Our projection is fused into the join and does not perform duplicate elimination (that task is handled by another function), while they have optional duplicate elimination and do not use fusion at all.

Diamos *et al.* [22, 23], Wu *et al.* [124, 126] and Young *et al.* [128] have also developed relational operators for GPUs which integrated into the Red Fox [150] platform based on LogiQL, an extended version of Datalog developed by LogicBlox [44]. Their join algorithm, compared to the INLJ of GDB, shows 1.69 performance improvement [22], we believe a similar performance difference exists with our modified version of the GDB's INLJ because the core of the algorithm was not modified. Their selection performs two prefix sums and the result is written and then moved to eliminate gaps; our selection performs only one prefix sum and writes the result once. They discuss kernel fusion and fission in [126]. We applied fusion (e.g., simultaneous selections, selection then join, etc.) at source code, while they implement it automatically through the compiler. Kernel fission, the parallel execution of kernels and memory transfers, is not yet adopted in our work.

4.1.5 Summary

A summary of GPU-Datalog is presented in Figure 4.4. GPU-Datalog is based in three *main* RA operators: selection (σ), join ($\bowtie$) and projection (π). GPU-Datalog also includes *secondary* operators like negation ($\neg$), comparisons ($<, >, <>, =, >=, <=$), aggregation (GROUP BY) and its related operations (SUM,COUNT,AVG), and arithmetics ($+, -, *, /$). Many of said operators are implemented with help from

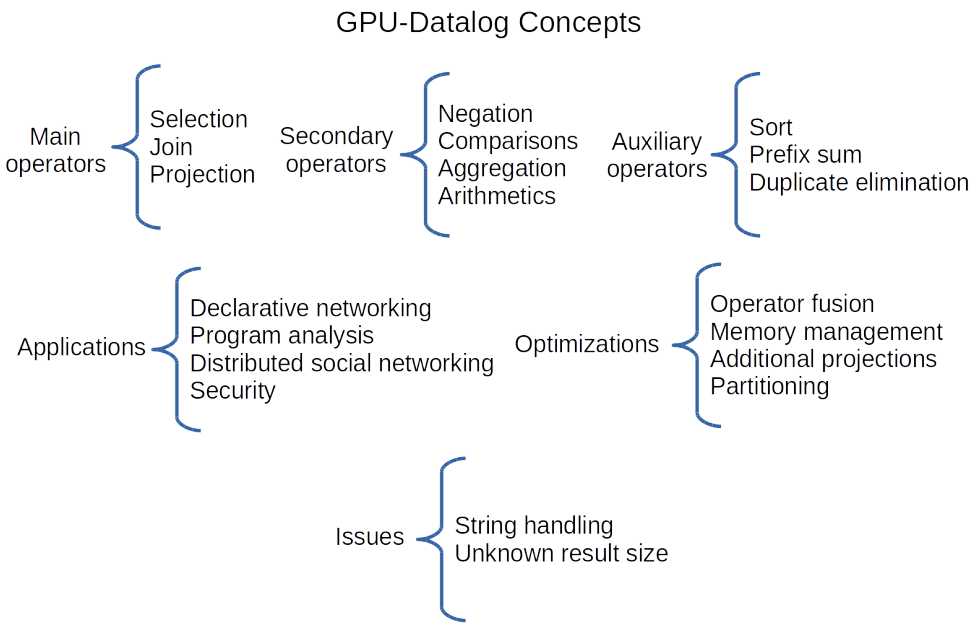


Figure 4.4: Most important concepts for GPU-Datalog.

auxiliary operators including sorting, prefix sums (also called scans) and duplicate elements elimination. Two main *issues* had to be addressed in order to parallelise the RA operators: the variable processing time of strings, which was solved by transforming said strings to a numeric representation and the unknown size of the result for many operators, which was addressed by counting the elements in the result before writing it to memory. Several *optimizations* were implemented like executing more than one operator in a single kernel call, an intelligent memory management that minimizes the number of transfers necessary to ground an MLN, additional projections to eliminate unnecessary columns earlier in the computation and data partitioning to process big data and use multiple GPUs. Finally, the Datalog language has many *applications* [50] which we believe can be accelerated with GPU-Datalog.

4.2 GPUSATLIB

Our GPU-based MaxWalkSAT [53] and SampleSAT [119] algorithms compose a library called GPUSATLIB. This library has around 800 lines of C and CUDA code, and not only allows to perform the search step of MLN inference (MaxWalkSAT) or weight learning (SampleSAT), but also includes an interface to allow it to be used for solving or sampling general weighted MaxSAT problems. Note that it is also possible for our library to handle unweighted MaxSAT problems (i.e., where all clauses are equally important) by assigning 1 as weight to each clause.

At code level, the main function of our library is called `walkSAT` and can be called normally from CUDA C or through Java using JavaCPP [130]. Said function receives several arguments including: a flag value to indicate if MaxWalkSAT or SampleSAT will be used; values for the stopping conditions, which can be a maximum number of iterations, a minimum number of unsatisfied clauses, and/or a minimum solution cost (a -1 indicates that the condition will not be used); linear GPU arrays with the size, weight, and atoms of each clause; linear arrays with the truth value, expected reduction (or increase) to the solution cost if flipped (if the truth value is changed from true to false or vice versa), and clauses that include each atom along with their number; and arrays with the current best solution and the unsatisfied clauses. Note that both arrays that measure sizes must be transformed with a prefix sum, to set the right positions where the elements are to be found. At the end, the function returns the solution cost and the solution itself is found in the argument array with the current best solution. As an example of how these array are created, consider a problem with clauses `1: a v b, 2: a, 3:b v c v d` with `a = f, b = t, c = t, d = f`, the arrays whose first position has index (0) will then be:

```
CLAUSE ARRAYS
sizes 2, 1, 3 -> 0, 2, 3, 6
weights 1, 2, 3
atoms in clauses 0, 1, 0, 1, 2, 3
```

```
ATOM ARRAYS
truth value f t t f
gain -2, 1, 0, 0
clauses that include the atoms 0, 1, 0, 2, 2, 2
sizes for clauses that include the atoms 2, 2, 1, 1 -> 0, 2, 4, 5, 6
EXTRA ARRAYS
current solution f t t f
unsatisfied clauses 1
```

To better understand these arrays consider the first clause `1: a v b`: its size is 2 and is written as the first element (0) of `sizes` because it is the first clause (note that the second part of `sizes` is the prefix sum); its weight is 1 and is also written at the first position of `weights`; and it has two atoms `a` and `b` which are represented by the first two elements of `atoms in clauses`, 0 which is the first atom and 1 which is the second one. Now, consider the atoms of said clause (`a` and `b`): their truth values are false and true, and are the first two elements of `truth value`; the reduction for flipping `a` is -2 because it would satisfy the second clause, while the increase for `b` is 1 because flipping it would no longer satisfy the first clause; `a` is included in the first and second clauses, while `b` is included in the first and third clauses, thus `clauses that include the atoms` has elements (0, 1) for `a`, followed by (0, 2) for `b`; and the prefix sum of `sizes for clauses that include the atoms` allows to access the clauses that include said atoms in a straight-forward manner, where the clauses that include `a` start at position 0 of `clauses that include the atoms` and finish at position 1, while those for `b` start at position 2 and finish at 3. Finally, the `current solution` has the truth values for each atom in order (`a,b,c,d`) and the only unsatisfied clause is the second clause (`2: a`).

4.2.1 GPU MaxWalkSAT

As mentioned in Section 3.4, we propose two options to parallelise the MaxWalkSAT algorithm on GPUs: having each thread execute the algorithm and find its own solution to the satisfiability problem or using all threads together to find a single solution to the problem. Since MaxWalkSAT is an approximate, stochastic solver, it may not be able to find the best solution. However, the probability of finding good solutions increases in the first option as different starting points may lead to different (and hopefully better) solutions. Unfortunately, we had to implement the second option, as our analysis of the MaxWalkSAT algorithm revealed that the memory requirements for the first one would be too high, since each GPU thread has to store its own truth values, unsatisfied clauses and so on. This complication was corroborated during the evaluation of the second option (Section 5.3.2), as the single MaxWalkSAT instance used required around 100-160MB of GPU memory. If we were to execute one instance per thread in our GPU (Tesla K40c with 12GB of memory and 2880 CUDA Cores), we would require 281-450GB, much more than the total amount of GPU memory available. We propose an interesting third intermediate solution in Section 6.2.3.

The problem of overshooting (too many atoms are being flipped, increasing the number of unsatisfied clauses rather than reducing it) when close to a good solution, was solved by monitoring how the system performs. As soon as the number of unsatisfied clauses is less than the number of cores or a certain number of iterations have elapsed, we transfer the computation to the CPU solver (which flips only one atom per iteration), starting from the best solution found by the GPU MaxWalkSAT solver.

The GPU MaxWalkSAT Algorithm 6 is based on the second option and was implemented as a loop of four GPU kernels running in sequence. Note that the start of each kernel and of the CPU instructions are tagged and recall that in a kernel all GPU threads execute the same instructions. The algorithm's input includes the set of active atoms AA in the Markov Random Field (MRF) partition, the set of active

Algorithm 6 MaxWalkSAT algorithm for GPUs using a Host (CPU) thread and four GPU kernels.

```
1   Input:  active atoms AA, active clauses AC, iteration limit IL,
2           number of GPU cores NC, probability P
3   Output: best truth values for active atoms BTV
4   TV <- Generate random truth values for AA       --- Host
5   UC <- Add unsatisfied clauses from AC
6   SC <- Calculate cost of TV
7   BTV <- TV
8   BC <- SC
9   Prepare metadata
10  While size of UC >= NC AND iteration < IL
11      For each GPU thread                          --- Kernel 1
12          RC <- Select random clause from UC
13          With probability P
14              A <- Select random atom in RC
15          else
16              A <- Select best atom in RC
17          Ensure no other thread has A
18          Flip truth value of A in TV
19          SC <- Add A's flip cost
20          ADJC <- Add all adjacent clauses of A
21      Remove duplicates from ADJC                 --- Thrust Library
22      If SC < BC                                  --- Host
23          BTV <- TV
24          BC <- SC
25      For each clause C in ADJC                   --- Kernel 2
26          Calculate satisfied atoms in C
27      For each clause C in ADJC                   --- Kernel 3
28          If C is not satisfied
29              UC2 <- Add C
30          For each atom A in C
31              Ensure no other thread has A
32              Update flip cost of A
33      For each clause C in UC                     --- Kernel 4
34          If C is not satisfied
35              UC2 <- Add C
36      UC <- UC2                                   --- Host
37  Return BTV
```

clauses AC (where each clause also belongs to the MRF partition and is made of one or more active atoms AA), an iteration limit IL, the number of GPU cores NC, and a probability P used during atom flipping. The output is an array with the best truth value (BTV) found for each active atom AA. Note that, each time a (GPU) kernel finishes, control is first returned to the Host thread (running in one CPU-core in the multicore host), which either performs an operation or launches another kernel.

First the Host thread prepares an initial random solution (lines 4-8) by: creating an initial truth value assignment to all active atoms (AA) which is stored in the truth values array TV; adding all unsatisfied clauses from all active clauses (AC) to array UC; computing the solution cost (SC) of our initial random solution TV by summing the weights of all unsatisfied clauses; and setting the best cost (BC) and best truth values (BTV) found so far as SC and TV respectively. It also creates some metadata (line 9) including *clause adjacency* (for each atom, we keep a record of all clauses that include it), *flip cost* (each atom has a negative cost if flipping it reduces the number of unsatisfied clauses and a positive cost if flipping it increases that number), number of *satisfied atoms* in each clause, among others.

Kernel 1 (lines 11-20) begins by having each thread select a random clause (RC) from the array UC of unsatisfied clauses, and then an atom in the clause. The atom selection has a probability P (usually 0.5) of choosing a random atom A or choosing the best atom A (i.e., the one with the lowest *flip cost*). Then we ensure, through locks and atomic operations, that no other thread has the same atom A. If two or more threads chose the same atom, all but one stop their execution. Then, for each atom A, its truth value stored in TV is flipped, its *flip cost* added to the solution cost SC and its *adjacent clauses* are appended to array $ADJC$. Kernel 1 uses a fixed number of threads based on the capabilities of the GPU (maximum number of threads per block times available Streaming Multiprocessors), seeking to use all available GPU processing capacity.

The Host thread then calls the GPU-based sort and unique functions of the Thrust Library to remove duplicates from array $ADJC$ (line 21). Said Host thread then

compares the solution cost SC produced by Kernel 1 to the best solution cost BC (from a previous loop or the initial one computed by the Host in line 8), storing it as the new best cost BC if it is smaller and replacing the best truth values BTV with those in TV (lines 22-24).

Kernel 2 (lines 25-26) calculates the number of *satisfied atoms* in each clause C in array $ADJC$ — positive atoms are satisfied if their truth value is true and negative ones are satisfied if their value is false. Kernel 2 uses one thread for each clause on array $ADJC$ (thus, its number of threads changes at each iteration).

Kernel 3 (lines 27-32) also uses array $ADJC$ and a thread for each clause in $ADJC$. The kernel begins by testing each clause C to determine if C is satisfied or not, adding unsatisfied clauses to array $UC2$. Then, the *flip cost* of each atom A in clause C is updated based on its new truth value stored in TV (obtained in the first Kernel) and on the number of *satisfied atoms*. Since it is possible for an atom A to be repeated in the clauses of array $ADJC$, we must once again use atomic operations to ensure that only one thread updates the *flip cost* of a single atom A.

Kernel 4 (lines 33-35) tests if the clauses found in array UC are now satisfied or not, using one thread per clause. Unsatisfied clauses are added to array $UC2$. Finally, array $UC2$ becomes array UC (i.e., the input array of unsatisfied clauses that is passed to the first Kernel) and a new iteration begins.

This process is repeated until the number of unsatisfied clauses (size of UC) is less than the number of GPU cores NC or until the iteration limit IL is reached. In either case, the MRF partition given to the GPU is further processed by the CPU solver, using the best truth values (in array BTV) returned by the GPU solver. The idea is to further improve the solution of the GPU solver, which is very likely to happen if the GPU solver stopped due to the number of unsatisfied clauses UC and unlikely if the iteration limit IL was reached.

4.2.2 GPU SampleSAT

To compute clause weights, we use the Diagonal Newton (DN) [24, p. 49] method, an iterative method that adapts its parameters at each iteration using MC-SAT. MC-SAT [94] is an iterative uniform slice sampler that uses SampleSAT [119] to sample around the solution of a weighted MaxSAT problem. Finally, SampleSAT is a weighted MaxSAT sampler that combines the MaxWalkSAT algorithm with a search technique called Simulated Annealing (SA) [52].

MaxWalkSAT and SA work together as follows: at each SampleSAT iteration, an atom is chosen with probability p according to the MaxWalkSAT algorithm (a random or the best atom A of an unsatisfied clause UC, as described above) and with probability $(1 - p)$ the choice is based on SA (which is essentially any atom). Each SampleSAT iteration starts with random truth values for the active atoms and, at the end, returns the best found truth values BTV along with the corresponding solution cost SC. MC-SAT then uses these values to adapt the parameters of the DN method.

Our MLN platform, called GPU-Tuffy and presented in Section 4.4, attempts to accelerate weight learning by computing the grounding and the SampleSAT algorithm in the GPU, as shown in Figure 4.5. The grounding process is the same as the one described further in Section 4.4 and has been reduced to a single call in Figure 4.5 for clarity. GPU SampleSAT is a modified version of GPU MaxWalkSAT where the first Kernel includes, for each thread, an extra stochastic condition to determine if it should select the atom according to MaxWalkSAT or SA. All other kernels are reused without changes and its input is usually larger as it includes the active atoms AA and active clauses AC that form the whole MRF (not just a partition of it). The output is also larger (truth values for all active atoms AA), but the stopping criteria is the same (few unsatisfied clauses UC or an iteration limit IL) and, once the GPU solver finishes, the CPU solver is likewise used to improve the solution.

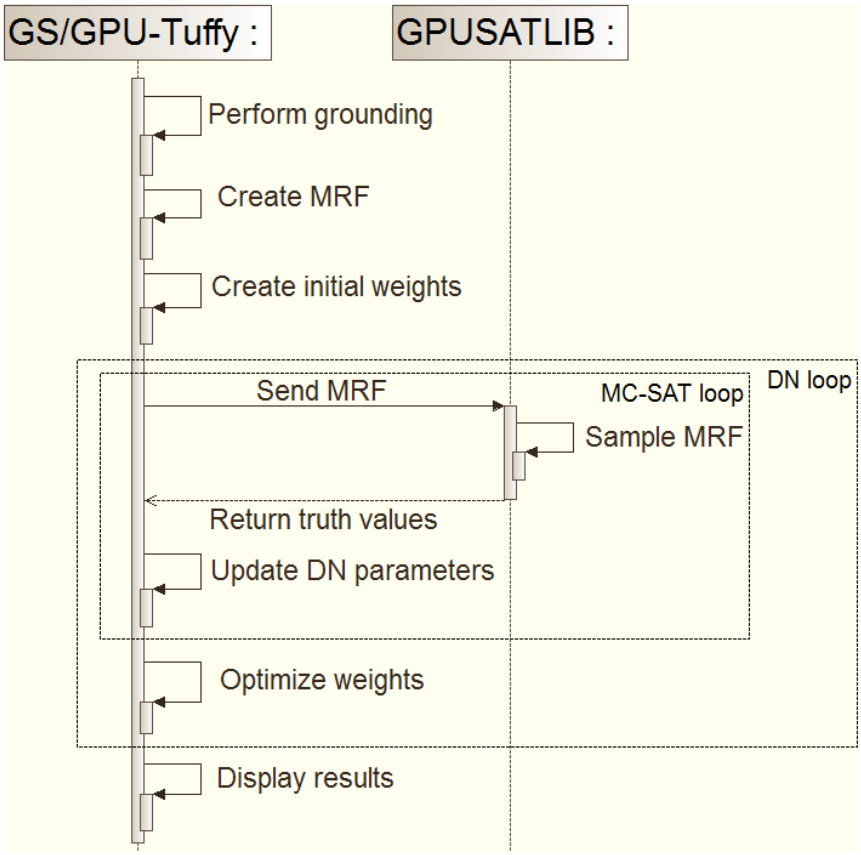


Figure 4.5: GPU-Tuffy running our GPU SampleSAT algorithm for weight learning.

4.2.3 State of the Art for Satisfiability on GPUs

The boolean satisfiability problem has been extensively studied and there are several GPU systems based on different approaches. One of such systems is called GPU4SAT [21], which is based on the multiplication of two matrices. The first matrix indicates the problem with each clause representing the rows, each variable representing two columns (one for the positive truth value of the variable and one for the negative), and each position has one if the variable satisfies the clause and zero otherwise. The second matrix represents the possible solutions to the problem, where each row is a variable or its negation, each column is a possible solution and each position has one if the variable should be true in the solution and zero otherwise. Each column of the resulting matrix whose values are all greater than zero indicates that the possible solution in the same column index of the second matrix is actually a solution to the problem. As an example consider the following satisfiability problem solved by multiplying:

```
                                                  Possible solutions
                                                    I1 I2 I3 I4
                                                  | 1  0  1  0| x1
                Satisfiability problem            | 1  0  0  1| x2      Solutions
              x1 x2 x3 x4 -x1 -x2 -x3 -x4         | 1  0  1  0| x3     I1 I2 I3 I4
x1 v x2   | 1  1  0  0   0   0   0   0| | 1  0  0  0| x4    | 2  0  1  1|
-x2 v -x3 | 0  0  0  0   0   1   1   0|*| 0  1  0  1|-x1 = | 0  2  1  1|
x3 v -x4  | 0  0  1  0   0   0   0   1| | 0  1  1  0|-x2    | 1  1  2  1|
                                         | 0  1  0  1|-x3
                                         | 0  1  1  1|-x4
```

Note how the first row of the first matrix has ones in `x1` and `x2` because they are the only variables that can satisfy the clause and that same logic is applied to all the other clauses. Also, the first column in the second matrix (`I1`) indicates that a possible solution would be to make all variables true, `I2` indicates that another

solution could have all variables false, and so forth until all valid combinations are enumerated. A valid combination cannot have a variable true and false at the same time, like `x1 = 1` and `-x1 = 1` (for simplicity, not all valid combinations appear in our example). Finally, the solution matrix indicates that only `I3` and `I4` are solutions to the problem and one can easily see that it is indeed the case, since the second clause is not satisfied in `I1` (`-x2 = 0 v -x3 = 0`) and the first is not satisfied in `I2` (`x1 = 0 v x2 = 0`), while `I3` and `I4` satisfy all.

Matrix multiplication in GPUs is a very efficient operation, however, in larger problems, enumerating the possible combinations for the problem and its solutions results in matrices of intractable sizes. Fortunately, as shown by the example, not all combinations need to be specified for the second matrix. GPU4SAT exploits said property by dividing the processing in two steps: a local search step that proposes possible solutions to the problem by changing the truth values of variables in unsatisfied clauses (similar to what MaxWalkSAT does) and a multiplication step that evaluates the proposed solutions. These steps are iteratively repeated until the solution is reached or a certain number of iterations have elapsed.

In another work, Fujii and Fujimoto [32] proposed a CPU-GPU hybrid algorithm based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [19] for 3-SAT, a variation of the general satisfiability problem where clauses have three literals at most. The DPLL algorithm considers the satisfiability problem as a large, unknown valued formula where all clauses are joined by conjunctions and works by iteratively assigning truth values to the variables one by one, until the formula becomes true (if the problem is satisfiable) and the problem is solved. If at some point, a value assignment causes the formula to be false, then the currently assigned values will not satisfy the problem and the algorithm must backtrack. The algorithm backtracks by changing the assigned value of the last considered variable and testing whether the formula is still false, in which case it once again backtracks to a previous variable, or it is no longer false and the algorithm can continue assigning values to the next variables. DPLL can be optimized with several techniques including the Boolean

Constraint Propagation (BCP), which periodically checks if certain variables *must* take certain values in order to satisfy the formula.

As an example of this algorithm, consider a problem with clauses `x1` v ¬`x2` v `x3`, ¬`x1`, and `x2`, which is equivalent to the formula `x1` v ¬`x2` v `x3` ∧ ¬`x1` ∧ `x2`. The algorithm starts from a variable say `x1` and sets its value to `true`, then evaluates the formula and concludes that it is false (because ¬`x1` is false). As the formula is now false, the algorithm backtracks and now tries `x1 = false`, which is an acceptable value for the formula (whose value is still unknown but not false). Finally, suppose that the algorithm sets `x2 = true`, which is an acceptable value and BCP determines that `x3` has to be `true`, otherwise the clause `x1` v ¬`x2` v `x3` would not be satisfied. The found values for the variables make the formula true and satisfy the problem.

DPLL was parallelised by having the CPU assign the values to the variables and perform backtracking if necessary. The GPU receives the clauses of the problem in groups and maps each group to a thread block. Each thread block then checks if a clause is *conflicting* (i.e., all its literals evaluate to false, causing the whole problem to be unsatisfiable for the current truth values) and performs BCP. The performance of their parallel algorithm was compared against a sequential version in the CPU, with both randomly generated and competition problems. The results show an speed-up of up to 6.7 for problems of more than 1000 variables.

Also relevant is the work by McDonald [77], who parallelised the WalkSAT algorithm, which is similar to the MaxWalkSAT algorithm except that clauses are not weighted, all are equally important. His approach (which was implemented for multicores but only outlined for GPUs) is based on having each thread compute the WalkSAT algorithm, starting from a different point in the search space. To prevent threads from visiting less promising regions of the search space, an optimization technique for the DPLL algorithm called conflict driven clause learning (not to be confused with clause learning in MLNs, they are two unrelated concepts) was adapted into WalkSAT. The idea is to create a shared database of clauses that indicate the dependencies among the variables. Said database is incrementally constructed by all

threads when working with multicores and by a single CPU thread when using the GPU.

The basis of this clause learning is to construct conflict graphs at certain iterations of the algorithm. Said graph construction starts from two clauses with a common conflicting variable (i.e., a variable who has positive instances in some clauses and negative instances in others) and then adds the positive and negative instances of the variable as leaf nodes. Next, all other variables in the clauses are added as parent nodes with the opposite sign and another clause that includes any of the variables already in the graph is chosen to add its variables as parents of the parent nodes. This process is repeated until an arbitrary stop criterion is reached, like the depth of the graph. Once the graph is finished, clauses are learnt by "cutting" the graph at certain intervals and joining the negation of all variables in the cut with disjunctions. These clauses represent dependencies among the variables.

As an example of said learning, consider the clauses and their conflict graph presented in Figure 4.6. Construction begins from the first two clauses using variables `a` and ¬`a` as leafs and then adding the negation of `b` from the first clause and the negation of `c` from the second. From there, since ¬`c` has no other occurrence, it becomes a root node and the construction continues with ¬`b`, which adds `d` and `f` to the graph. Finally, these last two variables add `g` to the graph and the graph is complete. Then, the graph is cut wherever new variables appear and the negation of the variables in the cut are joined with disjunctions, forming the learnt clauses. These learnt clauses indicate dependencies like: if `c = false`, then `g` must be `false`, `b` must be `true` and ¬`d` `v` ¬`f` must be satisfied. It can easily be seen that it is indeed the case, since `c = false` forces `a = false` in order to satisfy the second clause, `a = false` in turn forces `b = true` to satisfy the first clause, and that forces `g = false` to satisfy the last two clauses.

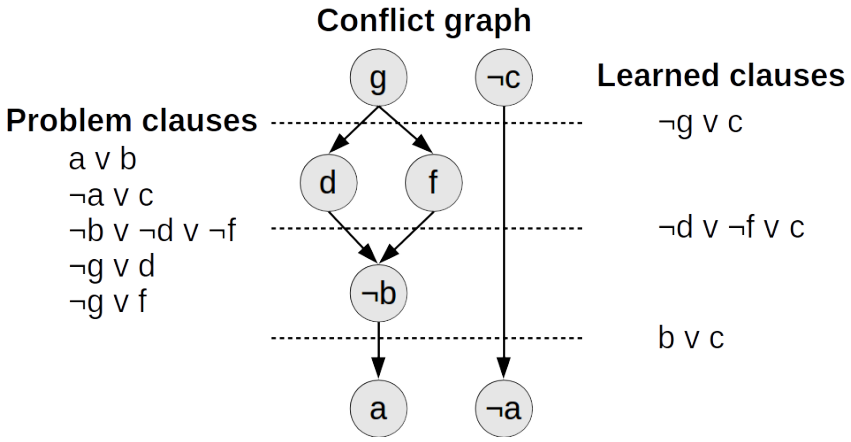


Figure 4.6: Conflict graph created from the first two problem clauses using conflicting variable `a`.

4.3 GPU parallel ILP

In order to understand how to better parallelise clause learning in MLN, we started by creating a parallel platform for general, FOL clause learning based on Inductive Logic Programming (ILP). Recall that ILP is usually based on Algorithm 1 of Section 2.5.2, which searches for rules that, when applied to some training data, their results must cover a set of given positive examples and must not cover any example found in a set of negative ones. The main components of our platform are: the YAP Prolog system, the Aleph ILP system [139], written in Prolog, and GPU-Datalog. Coordinating all these systems and setting GPU-Datalog to compute coverage required roughly 200 lines of C and Prolog code.

Figure 4.7 shows our GPU-based platform for relational learning from a high level perspective. The left block runs in a single host CPU core. First, the YAP Prolog system reads/compiles the ILP program and translates its strings into a numeric representation (NR), in a similar process to the one described in Section 4.1. Aleph then performs Algorithm 1 by simply calling its internal predicate `induce`. For each new candidate rule found, YAP compiles the candidate rule into a Datalog coverage computing rule by modifying the head, adding an extra predicate and translating it to GPU-Datalog's NR. The extra predicate includes, for each example, an identifier, if its positive or negative (polarity) and its variables. The modified head generates

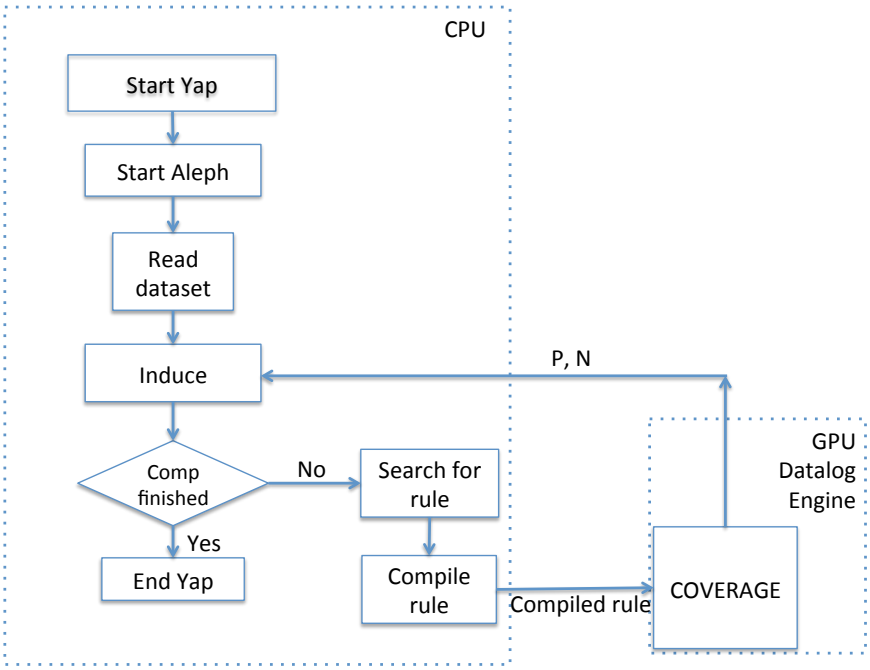


Figure 4.7: Our GPU-based ILP platform organization.

a result that includes only the identifier and the polarity of each example, as shown in the following Aleph sample rule (the last rule from the example presented in Section 2.5.2):

```
grandfather(Y, X) :-
  father(Y, Z),
  father(Z, X).
```

which is compiled into the coverage computing rule:

```
covers(Id, Pol) :-
  example(Id, Pol, Y, X),
  father(Y, Z),
  father(Z, X).
```

Evaluating this rule using GPU-Datalog materializes the `covers` table with the set of covered example identifiers and their respective polarities, and we simply have to count using the polarity to obtain the number of positive and negative examples (P

and N respectively in Figure 4.7). The process continues searching and evaluating rules until it finds a high-quality rule (one which covers all positive examples and no negative ones), all possible rule combinations are tried, or a certain number of iterations is reached. As an example, consider the sample coverage computing rule with tables containing the following elements:

```
example(1, positive, Bob, John).
example(2, positive, David, Harry).
example(3, negative, Gary, Ana).
example(4, negative, Frank, Helen).
father(Bob, Sam).
father(Sam, John).
father(David, Greg).
father(Greg, Harry).
father(Ana, Gary).
father(Helen, Frank).
```

joining and projecting `father(Y, Z)` and `father(Z, X)` would set the variables `(Y,X)` with `(Bob, John)` and `(David, Harry)`. The next join and projection between `(Y,X)` and `example(Id, Pol, Y, X)` would materialize the `covers` table with `covers(1, positive)` and `covers(2, positive)`. Finally, we count the number of positive ($P = 2$) and negative ($N = 0$) examples covered and since we found a high-quality rule, the computation finishes.

Using the same approach, we also created a multicore ILP implementation based on the OpenMP version of GPU-Datalog (described in Section 4.1.3). While these implementations proved to be very effective for general clause learning (as shown in [75]), for MLN clause learning, instead of using the coverage of the examples, a different evaluation function like Equation 3.11 found on Section 3.5.1 has to be used.

4.3.1 State of the Art for parallel ILP

With respect to running ILP algorithms in GPUs, to the best of our knowledge, this work presents the first attempt. Regarding parallelisation of the coverage algorithm using a multicore implementation and top-down evaluation, Côrte-Real *et al.* [14] managed to achieve linear speed-ups using a multi-threaded implementation based on Prolog and Aleph parallelised with *MapReduce* [20]. MapReduce is a programming model based on two key operations: *map* which applies a mapping function like sorting or classification to some given data, producing an equally large set of mapped values; and *reduce* which, given the mapped values, applies a reducing function like summation or aggregation. The most commonly used implementation of MapReduce is on a cluster of master-slave nodes, where the master node partitions the data, distributes it among the slave nodes which execute the map and reduce operations, and collects the results. Note that the execution times reported in [14] are only for the coverage step and ignore all other Aleph operations. In contrast, we report the full execution time in the experiments with our system in Section 5.2.

A. Srinivasan *et al.* also parallelised ILP with Aleph and MapReduce [112]. The authors investigated task parallelism, where each mapper node proposes a clause that should cover at least one positive example and the reducer nodes determine the best clause proposed. They also investigated data parallelism, where the mapper nodes compute a partial clause coverage given a subset of the examples and the reducer nodes calculate the total coverage of the clause. Their experiments show that data parallelism is better for their ILP configuration, but only if the data size is above a certain threshold (around 500,000 tuples), otherwise slowdowns are incurred due to the MapReduce overhead.

Finally, Fonseca *et al.* [29] evaluated several parallel ILP implementations using LAM MPI [111]. They considered 4 strategies: a search strategy where all nodes cooperate to generate a clause; a data strategy where each node is given a subset of the examples and computes the best clause for them; another data strategy where the nodes are also given a subset of the examples, but propose a set of good clauses

that are reevaluated at the end to obtain the best; and an evaluation strategy where coverage is computed using all nodes (similar to our implementation). Their results show that the search strategy is better, followed by the data strategies. However, the author's suggest that data and evaluation strategies might be better when working with a large background knowledge, as computing the coverage becomes much more time consuming. This is indeed the case in our experiments, were about 91% of the total execution time is spent on coverage.

4.4 GPU-Tuffy

Our first design to accelerate MLN performance is based on the Tuffy system and it is called GPU-Tuffy [73]. The UML diagram in Figure 4.8 shows the interaction between the main modules of GPU-Tuffy running MAP inference on an MLN, including the grounding and search steps. Coordinating these modules and translating the clauses to Datalog required the addition of about 600 lines of Java and C code. During the inference process, Tuffy runs first (top-left side of Figure 4.8), receiving three input files: i) the evidence (facts) file; ii) the MLN program; and iii) the queries file. Tuffy continues by creating a temporary database in PostgreSQL to store the evidence data and partial results. 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 grounding phase then starts. Grounding performs two steps: Computing Atoms (CA) and Computing Clauses (CC). CA computes the closure of the active atoms: atoms whose truth value might change from true to false or vice versa during search. CC computes the active clauses: clauses that can be violated (i.e., their truth value becomes false) by flipping zero or more active atoms. While CC uses all clauses, CA tends to use a smaller subset.

CA and CC are run by GPU-Datalog on the GPU after some preprocessing, as shown in Figure 4.8 and described next. For CA (top box in Figure 4.8), as Tuffy

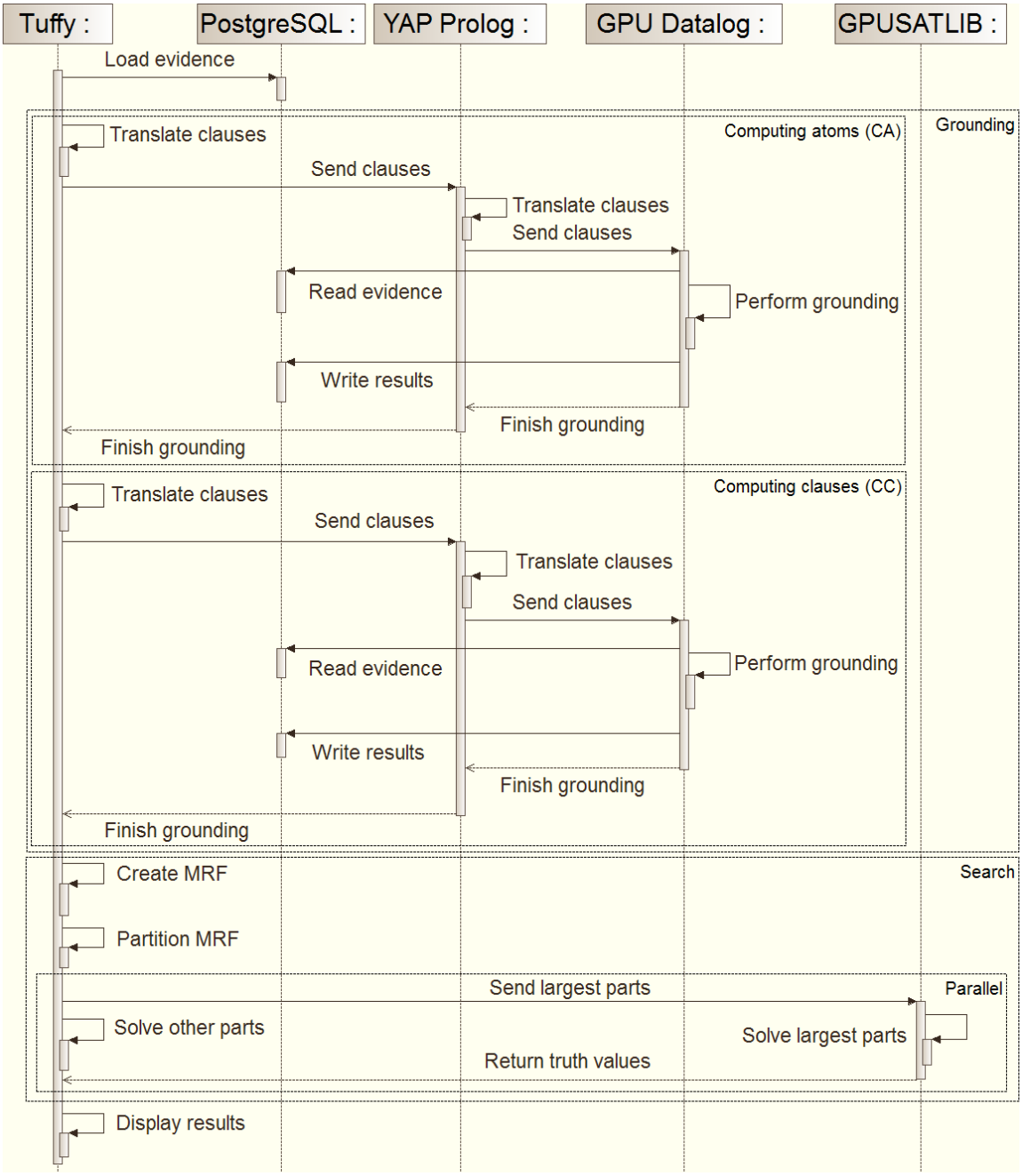


Figure 4.8: GPU-Tuffy modules running an MLN.

uses disjunctions to specify the MLN clauses, these clauses are first translated to Datalog's conjunctions (first translate clauses in Figure 4.8). The Datalog clauses are then sent to YAP (using a Java-Prolog interface) which compiles them into a numerical representation (NR) where each unique string is assigned a unique integer Id (as mentioned in Section 4.1 and represented in the second translate clauses in Figure 4.8). YAP then sends the clauses' NR to GPU-Datalog to process the actual grounding of active atoms. Tuffy also uses this NR for evidence loaded in the database; this simplified extending it with GPU processing.

For CC (second box in Figure 4.8), a similar preprocessing is carried out (third and fourth translate clauses in Figure 4.8). However, this time the clauses translated into Datalog are not the MLN disjunctions, but the SQL queries generated by Tuffy based on such disjunctions. This is because the output (one active clause per row and each clause represented by a set of active atoms along with its weight) is easier to generate.

Both CA and CC follow the same steps: read from the database to retrieve the evidence, compute the closure of the clauses using its relational algebra kernels and write the results back to the database.

Once the active atoms and active clauses are computed, the search step begins by creating and partitioning the corresponding MRF. The large partitions are then processed in the GPU using our GPUSATLIB and, at the same time, the smaller partitions are processed in parallel (with one partition per thread) using Tuffy's CPU MaxWalkSAT solver. Once all partitions of the MRF are processed, Tuffy displays the final result based on the found truth values of the atoms in the MRF.

4.5 GPU-RockIt

Our next design is based on RockIt [90] which (as mentioned in Section 3.4.1) uses a different approach to MLN MAP inference based on integer linear programming and Cutting Plane Inference (CPI), which partitions the linear programming problem into

smaller problems. Since the search phase in RockIt is already quite efficient, only the grounding process is parallelised in the GPU by the addition of of approximately 500 lines of Java code into RockIt (for clause translations into Datalog and coordination between modules).

As grounding in RockIt is similar to that of Tuffy, in this section we present the design of GPU-based grounding for, and its integration into, RockIt. The resulting system, GPU-RockIt, consists of RockIt, YAP Prolog and GPU-Datalog. Figure 4.9 shows the components of GPU-RockIt and their interactions. RockIt begins by reading the input files and storing the evidence using MySQL (instead of Tuffy's PostgreSQL). Strings are translated and stored in the DB as key, value pairs. We modified this translation to allow only numbers in the key, as required by GPU-Datalog. RockIt then calls our translation functions to create the Datalog clauses (first and only *translate clauses* in RockIt's timeline) needed to perform the grounding. These clauses are translated from RockIt SQL clauses which are, in turn, a translation of the MLN clauses. Note that all clauses are translated in this single step (which should not be confused with YAP's numerical translations shown later in Figure 4.9), while GPU-Tuffy uses two steps.

The translated clauses are divided into two groups: before-CPI and during-CPI. The first group includes the evidence and those clauses with a single atom and weight greater than zero. The second group includes all remaining clauses.

The before-CPI clauses are then grounded (top box in Figure 4.9). The grounding process in GPU-RockIt follows that of GPU-Tuffy (the only relevant difference is the organisation they use of input and output database tables). RockIt sends the Datalog clauses to YAP in order to transform them to the GPU-friendly numeric representation (NR), which will be the input for GPU-Datalog to compute the grounding. Then GPU-Datalog retrieves the evidence from the database, grounds each clause and writes back the result. The results of this first grounding will be used as the constraints of the first optimization problem, after being compacted by the Cutting Plane Aggregation (CPA) algorithm, which groups similar clause groundings

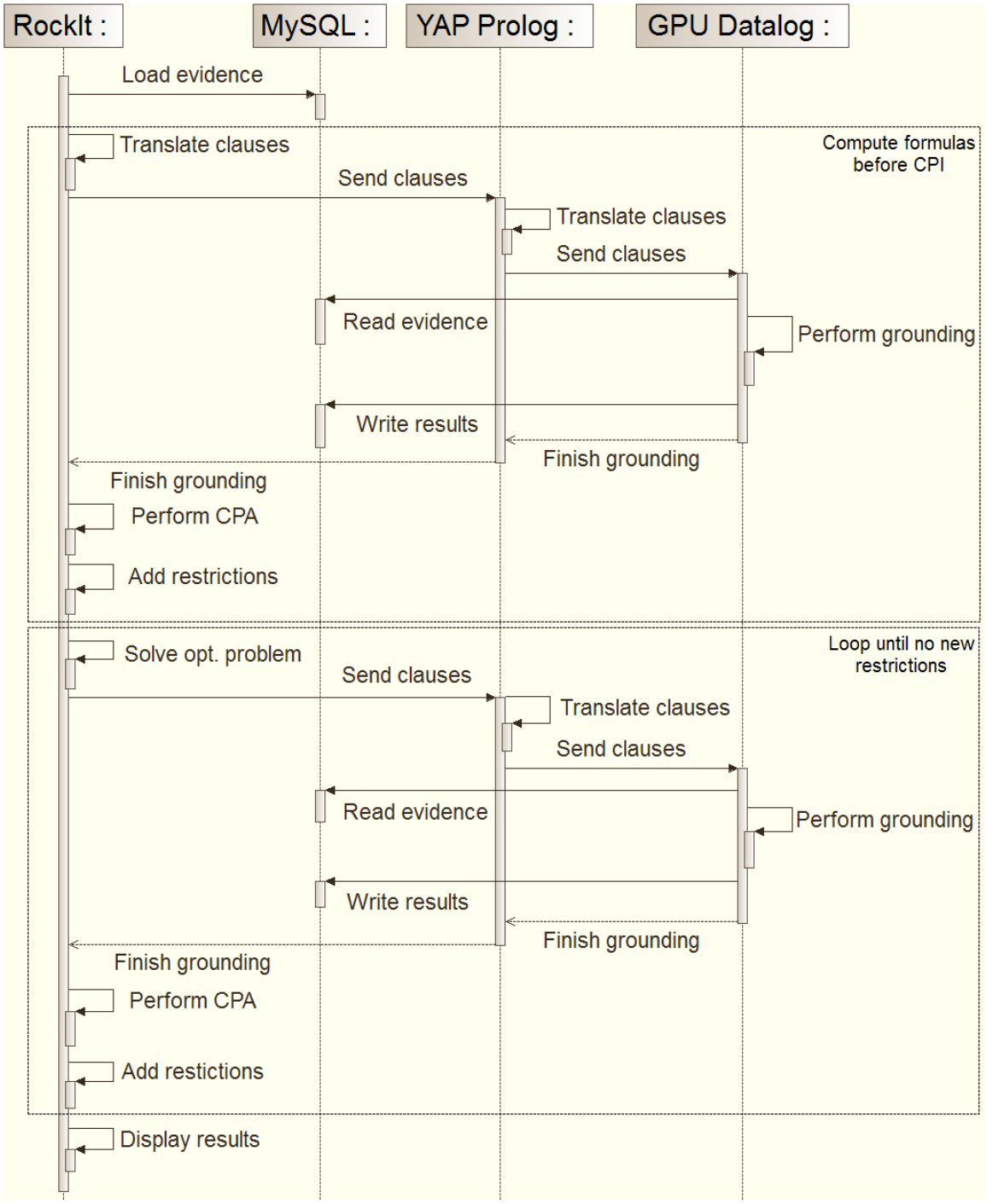


Figure 4.9: Main components of the GPU-RockIt system and their interactions.

into a single constraint.

With the constraints in place, the core part of CPI begins (bottom box in Figure 4.9): a loop that solves the current optimization problem, generating new evidence; computes new groundings using the during-CPI clauses and the new evidence; and then, with the newly generated groundings compacted by CPA, posts constraints to the next optimization problem. This process is repeated until it is not possible to find constraints for another optimization problem. At the end of the loop, RockIt displays the final result. The optimization problems are solved by the Gurobi multicore optimizer [149], which controls its execution through a bound on the error between the current and expected solutions called *gap*. Users can define this *gap* value or use RockIt's default *gap*. The optimizer will continue to search for a better solution to the problem until the error is lower than the *gap*.

In order to speed-up the grounding, GPU-Datalog preserves RockIt's duplicate removal procedure: keeping the results of each clause in a table and through a left join with such table, ensuring that any new row does not appear in older results. We take this idea further by maintaining the results in GPU memory for as long as possible with the help of GPU-Datalog's memory management module (described in Section 4.1).

4.6 Validation of our Platforms with Running Examples

To validate the proper functioning of our platforms and the correctness of their results, we executed several tests aimed at each platform as a whole and at the most important components: grounding with GPU-Datalog and SAT solving with GPU MaxWalkSAT. Note that performance gains were not measured in these tests (they can be found in the next chapter).

4.6.1 Validation of GPU-Datalog

The first validation of the current version of GPU-Datalog was performed against its first version presented in [74] and against YAP. The idea is to ensure that the new additions and modifications did not change the correctness achieved by the first version. The two applications used were the first two presented in [74]: a *join over four big tables*, where four tables, all with the same number of rows filled with random numbers, are joined together and the *transitive closure of a graph*, which is a recursive query based on a table with two columns filled with random numbers that represent the edges of a graph and the task is to find all the nodes that can be reached if we start from a particular node. The results where exactly the same in both applications for all 3 systems (YAP's results were in a different order, but that does not affect the tests), thus the latest version of GPU-Datalog maintained the correctness of the original version.

The second validation tested the whole operations of GPU-Datalog, including the new operation in the latest version like negation and comparisons, and was performed against the SQL grounding queries of PostgreSQL and MySQL. This validation warrants that the results of the groundings created by GPU-Datalog (for GPU-Tuffy and GPU-RockIt) were equal to those created by PostgreSQL (for Tuffy) and MySQL (for RockIt). The two applications used were *smokers* presented in Section 2.3 and *relational classification*, which classifies papers into categories based on several attributes. Both were set to their original configurations as shown in Section 5.3.1. For PostgreSQL, the test involved creating a barrier to pause the computation in both Tuffy's and GPU-Tuffy's code right after the second grounding is finished (CC, described in Section 4.4). Once the systems reached the barrier, we would manually access the database and dump all tables involved in the computation (ordered by the first column) to text files using the `COPY` command. The text files obtained from GPU-Datalog's groundings were then compared against those of PostgreSQL in a terminal using the `diff` command and, since not a single difference was found, we can be certain that GPU-Datalog is grounding correctly in GPU-Tuffy. For MySQL,

a similar process was performed except that, since RockIt and GPU-RockIt ground at each iteration (see Section 4.5 for details), the execution had to be paused, the tables dumped, and the files compared at each iteration. Similar to PostgreSQL, no differences in the groundings where found between GPU-Datalog and MySQL.

4.6.2 Validation of GPU MaxWalkSAT

GPU MaxWalkSAT was validated against the results of several CPU-based SAT solvers, both complete (i.e., those based on exhaustive searching) and incomplete (i.e., those based on local, stochastic search). This comparison is possible thanks to the SAT problems and their results published in the seventh evaluation of Max-SAT solvers (Max-SAT-2012) [133]. We chose the following problems based on their large size (compared to the rest of the problems in the set) and based on the fact that most of the solvers managed to find the best solution for them: `s2v140c1600_3.wcnf` (with 140 variables and 1600 clauses) and `file_rwms_wcnf_L3_V70_C900_7.wcnf` (70, 900) from the *random* set and `frb20-11-5.wcnf` (220, 5821) and `ram_k3_n12.ra1.wcnf` (66, 715) from the *crafted* set. The test consisted in creating a simple interface for GPU-Tuffy to directly feed the SAT problem to GPU MaxWalkSAT, skipping the grounding and other unnecessary tasks. Also, we had to reduce the number of threads per block used for the GPU kernels, as the number of variables in each problem is less than the number of cores in the GPU where the tests were made (a GeForce GTX 765M with 768 cores). However, this change did not affect the results of our algorithm and the best result for all four problems was found.

4.6.3 Validation of our MLN Platforms

Having validated all critical components, we sought to validate our whole platforms, GPU-Tuffy and GPU-RockIt. The validating comparison was made against Tuffy and RockIt using the smokers application, as it is rather simple to create a large amount of random evidence which has a unique, zero-cost solution (i.e., a solution

where all clauses are satisfied). The tests were carried out with 3 different sizes for the applications: 5000, 10000, and 20000 *friend* relationships, along with 500, 1000, and 2000 *smokers*. For each size, the application was executed 10 times in all four systems, ordering and comparing the results of every system in each execution. In the end, all systems managed to find the same, zero-cost solutions at each execution.

Since all tests presented in this section generated the same results in both our systems and in the state of the art systems, and since no execution errors where found in neither these tests nor the ones presented in the following chapter, we believe that our systems have been successfully validated and can be used for any Datalog, MaxSAT, or MLN application.

4.7 Summary

We have presented our experimental platforms to process MLNs using GPUs, *GPU-Tuffy* [73] and *GPU-RockIt*, their core components, and our parallel ILP system. GPU-Datalog [74], the FOL infrastructure of our MLN platforms and our ILP system, was greatly extended from its original version with several new operators and optimizations in order to handle MLN grounding and FOL clause learning. To handle large amounts of data that exceed the memory capacity of the GPU and to allow multiple GPU to process an application simultaneously, GPU-Datalog was also extended with a data partitioning scheme.

The other core component is a novel GPU-based library to solve and sample MaxSAT problems called GPUSATLIB, which is based on the efficient MaxWalkSAT [53] and SampleSAT [119] algorithms. The idea is to try to satisfy as many clauses as possible (prioritising those with higher weights) by changing (flipping) the truth value from true to false or vice versa of an atom in the unsatisfied clauses for a certain number of iterations or until no unsatisfied clauses remain. Sampling follows a similar process but allows any atom (not just those from unsatisfied clauses) to be flipped. Similar to this library, there are other satisfiability solvers for GPUs based

on other algorithms [21, 77]. Also, it is important to mention that both of these core components are stand-alone and can be used in several applications beyond MLNs.

Our ILP system parallelises clause learning in FOL using GPUs. This system is based on YAP Prolog which translates strings to numbers for easier processing and serves as a bridge between Aleph [139], which is in charge of proposing new clauses, and GPU-Datalog, whose job is to compute the coverage of each clause to determine its worth (i.e., based on some positive and negative examples, determine how many of each are represented by the clause, with the best clauses representing all positive and none of the negative). The idea of parallelising ILP in GPUs comes from the fact that several systems have parallelised it in distributed systems with good results [14, 29].

GPU-Tuffy, our first MLN platform, solves the MAP inference problem as a weighted MaxSAT problem. Its main components are: the Tuffy MLN system, which performs all preprocessing (data loading, MRF partitioning, among others) and displays the final results; GPU-Datalog which is used to compute the clause groundings; YAP which serves as the bridge between GPU-Datalog and Tuffy; PostgreSQL which stores the initial evidence and grounding results; and our GPUSATLIB which solves the weighted MaxSAT problem of the search step.

GPU-RockIt is our second MLN platform based on formulating the MAP inference process as an integer linear programming problem (i.e., an optimization problem). Its main components are: the RockIt MLN system, which performs all preprocessing, solves the optimization problem and displays the final results; GPU-Datalog and YAP Prolog which are used to compute the groundings similar to GPU-Tuffy; and MySQL (instead of PostgreSQL) to handle data I/O.

Finally, both platforms and their critical components were properly validated for correctness (performance is discussed in the next Chapter) in their results using several tests. All these tests were successfully completed by all our systems and the results were equal to those of the state of the art systems currently in use.

Chapter 5

Performance Evaluation

This chapter presents a detailed performance evaluation of our MLN platforms (GPU-Tuffy and GPU-RockIt), their core components (GPU-Datalog and GPUSATLIB), and our parallel Inductive Logic Programming (ILP) system.

First, we introduce an evaluation of the performance of GPU-Datalog (Section 5.1), the logic component of our MLN platforms, against Red Fox [150], currently the best state of the art system, showing that GPU-Datalog has good, competitive performance. Then, we present the evaluation of our ILP system based on Aleph [139] and GPU-Datalog (Section 5.2) using general first-order logic applications, which show the potential of our system as the basis of future MLN clause learning systems. Next, we present the general performance evaluation of our GPU platforms and of our GPU designs (Section 5.3) for: 1) the MaxWalkSAT algorithm part of GPUSATLIB for the search step on Tuffy, which concludes that the search as a satisfiability problem can also benefit from GPU parallelisation; 2) the grounding also based on GPU-Datalog for the RockIt MLN system (which forms our GPU-RockIt platform), which proves the flexibility our grounding design; and 3) the MC-SAT algorithm based on parallel SampleSAT provided by GPUSATLIB under weight learning (part of GPU-Tuffy), which indicates that parallelism can also improve weight learning.

In each section, we present our methodology, experimental setting, and the

obtained results. At the end, we conclude with an exhaustive discussion of the results obtained by our MLN platforms (Section 5.4) and a summary of the Chapter (Section 5.5).

5.1 GPU-Datalog Base Performance

Our first version of GPU-Datalog was presented in [74], showing great results when performing relational algebra (RA) operations (especially *joins*), easily outperforming non-GPU based systems. As mention in Section 4.1.4, Diamos *et al.* also developed a GPU platform for RA called Red Fox [150], which is based on an extended version of Datalog. Their experiments show that Red Fox is currently the best performing GPU system for RA and thus, we decided to compare our system with theirs.

5.1.1 Applications

We used 8 of the 22 TPC-H [152] queries (a famous decision support benchmark with scalable queries presented in Appendix B) to evaluate the performance of our system: 1, 3, 4, 5, 10, 18, 19, and 21. Since these queries are written in SQL, first we had to manually translate them from SQL to Datalog. Queries 2, 7, 9, 13, 14, 16, 20, and 22 were not used as they had sub-string operators (like, extract, substring). Since our system transforms whole strings into numbers, it was not possible for us to use these operators. Queries 6 and 17 had floating-point constants and were not of interest to our work. Queries 8, 11, 12, and 15 had operators that are hard to directly translate from SQL to Datalog (case, having, views). The evaluation used the same scale factor of 1 for TPC-H data which is roughly 1 GB. The tables were populated with random data using the TPC-H data generator, and that same scale factor.

Hardware-software Platform

The evaluation was performed in the same architecture described in [125], which includes the following components:

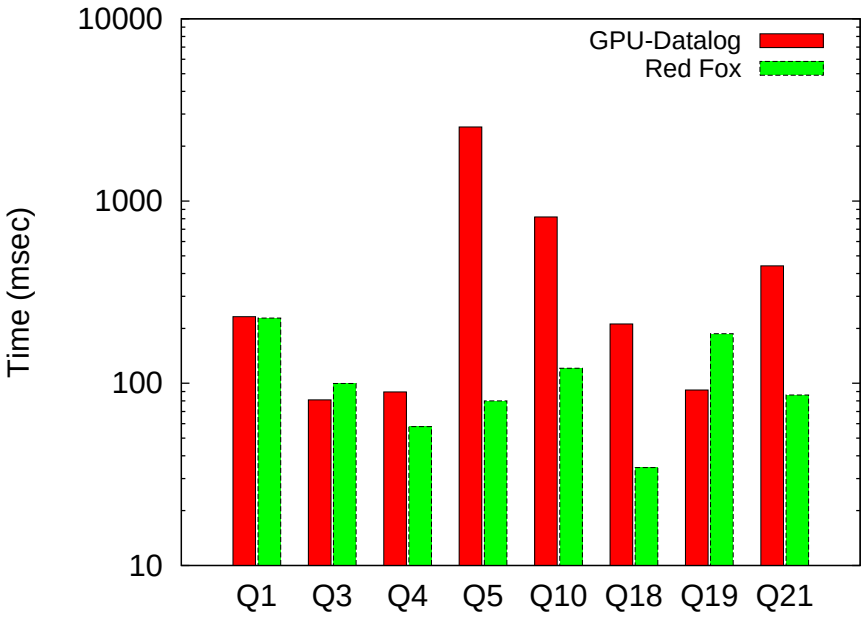


Figure 5.1: Comparison between GPU-Datalog and Red Fox using 8 TPC-H queries.

Host hardware: Intel i7-4771 with 4 cores at 3.5GHz, 4x32KB L1 instruction caches, 4x32KB L1 data caches, 4x256KB L2 associative caches, 8MB L3 cache, and 32GB of RAM. **Host software**: Ubuntu 12.04 and G++ 4.6.3.

GPU hardware: GeForce GTX Titan with 6GB of memory and 288.4 GB/s memory bandwidth, connected to a PCIe 3.0 x16. **GPU software**: NVCC 5.5 and Thrust library 1.7.

5.1.2 Results

Figure 5.1 shows the results of our system against the latest Red Fox results (2014/03/11) [150] with PCIe (GPU-bus) transfer time (i.e., time taken by memory transfers between the GPU and the host) — time spent on loading tables from disk to main memory is not included. Q1 shows roughly the same execution time for both Red Fox and our Datalog engine. Our engine is faster for Q3 and Q19 and Red Fox is faster for the remaining five queries. Red Fox uses primitives of the ModernGPU library [144], which are well-tuned for coalesced memory accesses, while we used the Thrust library [151] implementations. ModernGPU also uses the Merge-

Path framework [43, 91], which partitions the data and balances the workload among groups of threads. Still, we chose to keep Thrust instead of moving to ModernGPU as Thrust is a core part of the CUDA standard that is constantly evolving and because the latest performance comparisons between these libraries and other similar ones [79] show that no library is better for everything and that a new library called CUB [132] actually outperforms both Thrust and ModernGPU in several tasks.

Added to the library question, our system was optimized for logical queries, not SQL and our manual query translation from SQL to logic queries may also be sup-optimal. We believe that the main factor that affected the performance of our engine was query rewriting, particularly the use of auxiliary rules. For example, Q21 required two additional rules to represent two SQL *EXISTS* operators (boolean operator that return true if a subquery returns at least one record), which are not implemented per-se in our system. When these additional rules are evaluated, all possible solutions are returned (instead of true or false as *EXISTS* does in SQL), thus creating large intermediate tables which need to be joined to the main rule, increasing execution time.

Finally, the following factors adversely contributed to the performance results of our system:

- **Comparison predicates**. These predicates were performed after all join operations were finished in order to ensure that their variables are assigned to a value. By correctly rearranging these predicates, their evaluation is performed sooner, thus eliminating many unnecessary tuples earlier in the computation. For example, consider rule `a(X) :- b(X,Y), c(X,Z), Y < 3.`, where `b` and `c` have 100 elements each and only 5 of the elements in `b` comply with the conditional `Y < 3`. If the join is performed first, then 100*100 elements will be generated in the worst case and then reduced to 500 by the conditional. On the other hand, by doing the conditional first, the size of `b` becomes 5 and the worst case for the join becomes 5*100.

- **Join algorithm**. We based our join algorithm on the Indexed Nested Loop Join of the GDB system [47]. Red Fox's join algorithm was compared against GDB, report a 1.69 performance improvement over it [22]. We believe a similar difference exists between our algorithm and Red Fox's.

Note that this evaluation was performed with an earlier version of GPU-Datalog than the one described in Section 4.1. Some optimizations like faster comparisons and partitioning were missing from this early version. Furthermore, ModernGPU, Thrust, and Red Fox have also suffered changes. Nevertheless, we consider this evaluation as a good measure of the base performance of our system.

5.2 ILP with GPU-Datalog

As mentioned in Section 4.3, we combined GPU-Datalog with the ILP system called Aleph in order to speed-up the most compute intensive task, rule *coverage*. To evaluate its performance, we experimented with three different versions of Aleph: 1) the original Prolog code that runs relational learning applications in YAP [15], where coverage-lists are used to minimize the number of examples to be tested against a new rule (**Aleph-cov**), 2) a modified version of the original Aleph, where the whole set of examples is always passed to the coverage procedure (**Aleph-all**), and 3) our modified version of Aleph that calls GPU-Datalog to execute the coverage step, also using the whole set of examples. Our version uses two different libraries: 3.1) that runs the coverage step on the GPU (**Aleph-cuda**) and 3.2) that runs the coverage step on the host multicore with several threads using GPU-Datalog's multicore support presented in Section 4.1.3 (**Aleph-multi**). The two first versions **Aleph-cov** and **Aleph-all** use a top-down approach to evaluate the queries. The other two, **Aleph-cuda** and **Aleph-multi** use a bottom-up approach based on Datalog where rules are modified to return the identifier of each example and its polarity (i.e., its truth value) as result, which are then counted to compute the coverage (details of this process can be found in Section 4.3).

The first order models generated by all versions are the same as well as the counters of positive and negative examples. Each run was performed ten times. The reported execution times are averages of ten runs and are expressed in seconds.

5.2.1 Applications

- *carcino*: this is a well-known application that focuses on whether drugs may be carcinogenic in rodents. The ILP system has information on the drug's 2-D structure (atoms and bonds) and on major chemical properties such as the structural group (methyl groups, benzene rings, etc.), the genotoxicity (i.e., agents that damage the genetic information), and mutagenicity (i.e., an agent that changes the genetic material increasing the frequency of mutations) [113].

- *hiv*: this dataset is based on the DTP AIDS anti-viral screen, that checks tens of thousands of compounds for evidence of anti-HIV activity. The ILP system has information on the drug compound's 2-D structure [148, 120].

- *omop*: this dataset consists of simulated medical records, namely diagnosis and prescription data. The task is to find drugs that can cause adverse side-effects, and it relies on temporal relationships between prescriptions and diagnosis [101].

- *blog*: this is an application that contains collected blog postings about solar energy solutions taken from two Italian blog's discussions/threads. blogs are stored as a collection of Part-Of-Speech (POS) tokens. Further data on authorship, threading and title is also included in the analysis. The ILP task is to model blog's author's opinions [33].

These applications are characterized according to Table 5.1. Notice that *carcino* is the smallest dataset. The other three applications have data bases with similar sizes, millions of tuples and tens of thousands of examples.

Table 5.1: Applications Characteristics. Background Knowledge is given as number of facts (tuples) in the database.

Application	Background knowledge	Number of examples
carcino	21,303	297
hiv	2,310,575	48,766
omop	4,802,317	125,000
blog	5,124,092	50,000

Hardware-software Platform

We used the following platform to run our experiments:

Hardware: Core 2 Quad Processor Q9400 (4 cores in total) at 2.66GHz, with 4 x 32KB L1 instruction caches, 4 x 32KB L1 data caches, 2 x 3MB L2 caches (each cache shared between 2 cores), 6 GB DRAM, and a GeForce GTX 580, 1.54 GHz 512 CUDA Cores (16 Multiprocessors x 32 CUDA Cores/MPK), 1535 MB GDDR5 memory, 768KB L2 cache, CUDA Capability Major/Minor version number: 2.0.

Software: Ubuntu 12.04.1 LTS, gcc version 4.6.3, NVIDIA Corporation Cuda compilation tools, release 5.0, V0.2.1221, CUDA Driver Version/Runtime Version 5.0/5.0.

5.2.2 Results

In discussing our experimental results below, **Aleph-cov** is used as a baseline comparison algorithm, because it is the sequential version that is normally used for relational learning tasks. Also, **Aleph-all** shows the great labour that is covering the whole search space (which **Aleph-cuda** also does) and the benefit of using coverage-lists to guide said search (like **Aleph-cov** does).

First we compare the versions that run entirely on the host, in order to evaluate how the top-down and bottom-up evaluation behave on our relational learning applications. Thus, **Aleph-cov** and **Aleph-all**, both of which use the top-down approach, along with **Aleph-multi(1)** (i.e., **Aleph-multi** executed with only one thread) which is based on the bottom-up approach are presented first. Next, we

Table 5.2: Execution times for Aleph-cov, Aleph-all and Aleph-multi (1 thread), in seconds.

Application	Aleph-cov	Aleph-all	Aleph-multi(1)
carcino	14.02	139.50	238.81
hiv	211.79	272.99	391.77
blog	3201.76	13343.27	7177.20
omop	656.55	4528.78	1514.50

compare the results of our **Aleph-cuda** implementation with **Aleph-cov**, **Aleph-all**, and the best results obtained with **Aleph-multi**. We then evaluate the scalability of the **Aleph-multi** version with varying number of threads.

5.2.3 Aleph-cov, Aleph-all and Aleph-multi(1)

Table 5.2 shows the average execution times (in seconds) of all applications using three versions of Aleph (-cov, -all, -multi(1)). As expected, the execution times vary hugely from one application to another, since they present different characteristics (use of constants or built-in comparison predicates, compound terms, etc.) and stress different parts of YAP and Datalog.

Since Aleph-cov does not use the whole set of examples to perform the coverage, its execution time is much lower than Aleph-all and Aleph-multi with 1 thread, but the difference in times vary according to the characteristics of each application. For example, *carcino*, *omop* and *blog* benefit greatly from the coverage list optimization used by Aleph-cov.

5.2.4 Aleph-cuda, Aleph-all, Aleph-cov and Aleph-multi(best)

Figure 5.2 shows the execution times of all applications using the four versions of Aleph, Aleph-cuda, Aleph-all, Aleph-cov and Aleph-multi. Results shown for Aleph-multi are the best for each application, being *carcino*'s best time obtained with 1 thread, *hiv*'s best time obtained with 8 threads, and *omop*'s and *blog*'s best times obtained with 4 threads.

From the four applications, three of them greatly benefit from the CUDA

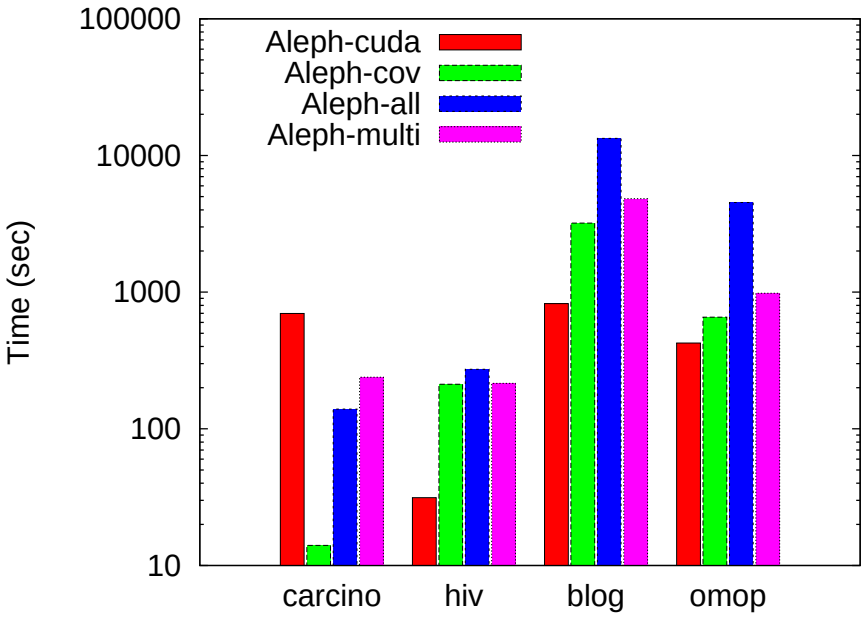


Figure 5.2: Total execution time for each application using all Aleph versions. Notice that the best times in the three larger applications are obtained by our version based on GPU-Datalog (Aleph-cuda).

implementation with speedups varying from 5 (*blog*) to 8.36 (*omop*), when comparing Aleph-cuda with Aleph-cov, the best CPU version. The application *carcino* was the only one that did not benefit from the CUDA implementation. Its small size caused the cost of memory transfers to dominate the total execution time.

One interesting observation is that Aleph-all, which imitates the same coverage behaviour of Aleph-cuda, by not using the coverage-list, is still much slower than the CUDA and multicore versions. Note that the plots use logarithmic scale. Also, we instrumented our code to report the total amount of time spent in the different operations of the ILP process. Our results show that *coverage* is indeed the most compute intensive task, with an average of 91% of the total running time in all applications spent on it.

Figure 5.3 shows the time taken by Aleph-cuda in each GPU-Datalog operation. There are six different steps. The first two perform all selections and self-joins required by the two sub-goals involved in each join. Next, an array is sorted to prepare for the join operation, then the join is performed. Finally, built-in comparison predicates

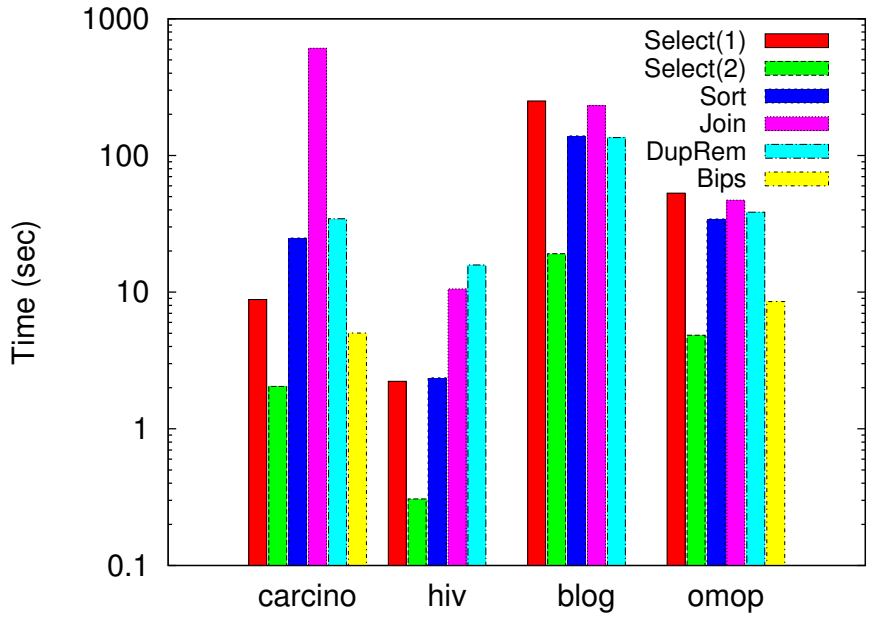


Figure 5.3: GPU-Datalog execution time breakdown.

are evaluated (if any), followed by duplicate removal (both operations detailed in Section 4.1.1).

Each application has rather different characteristics that affected its execution time. The *carcino* dataset, which did not benefit from the CUDA implementation spent most of its time in the join operation. In fact, join is one of the dominant operations for all applications together with selection. This is expected, since join is usually the most expensive operation in database processing. The *hiv* application is from a similar domain, chemo-informatics, but it is much less about joins: in fact the main operation is duplicate removal. A detailed analysis shows that the two applications use different representations for a molecule, with *hiv* fusing bonds and atoms into a single table. This technique was developed to take advantage of Prolog indexing, but seems to also result in smaller intermediate tables over which the joins are applied.

The *blog* and *omop* applications have the most well-balanced execution. Both datasets use many constants in the rules which imply selections, and also include self-joins. The joins themselves are noticeably fast, as they do not take much more

time than sorting or duplicate removal. This is probably because the main tables have strong functional dependencies in both cases. Last, *omop* uses comparisons to establish temporal precedence and is thus the only dataset where built-in comparisons play a significant role.

5.2.5 Aleph-multi(all)

Finally, we present a detailed measure of the performance of Aleph-multi. Table 5.3 shows the average execution times of Aleph-multi for a varying number of threads in the four applications. Speed-ups (if the number is above 1.00) or slowdowns (below 1.00) related to the Aleph-all (right) and Aleph-cov (left) versions, respectively, are between parentheses.

Table 5.3: Aleph-multi execution time with speed-ups (> 1.00) and slowdowns (< 1.00) for 2, 4 and 8 threads. The first number in the parentheses represents the speedup/slowdown of Aleph-multi when compared to Aleph-all. The second number represents the comparison against Aleph-cov.

Application	2	4	8
carcino	252.26 (0.55, 0.05)	264.36 (0.53, 0.05)	301.83 (0.46, 0.05)
hiv	271.51 (1.00, 0.78)	233.21 (1.17, 0.91)	215.10 (1.27, 0.98)
blog	5452.07 (2.45, 0.59)	4827.10 (2.76, 0.66)	4854.19 (2.75, 0.66)
omop	1109.33 (4.08, 0.59)	982.27 (4.61, 0.67)	1013.02 (4.47, 0.65)

Comparing the results of Table 5.3 with the results shown in Table 5.2, we can observe that the Prolog top-down execution with coverage lists (Aleph-cov) is the best for all applications. Bottom-up multicore Datalog (Aleph-multi) outperforms normal Prolog (Aleph-all) in all applications except *carcino*, where it also suffers from *carcino*'s small size. These results show that coverage lists are more important than the language (Datalog vs Prolog) or the evaluation method (top-down vs. bottom-up).

Also, despite both being based on the same Datalog engine, Aleph-cuda performs much better than Aleph-multi because our engine is finely tuned for GPUs. Thus, Aleph-cuda exploits more efficiently the GPU resources than Aleph-multi exploits the multicore resources. Moreover, because the coverage step presents a tendency

Table 5.4: Applications Characteristics.

Application	Inference formulas	Evidence relations	Tuples in relations
SM	3	3	310,000
RC	15	4	156,998
CS	6	13	2,458,317
HL	11	18	7,206,390
LP	24	21	2,924
ER	1265	10	12,304

to produce finer-grained tasks, the results are clearly better on the GPU, with a maximum speed-up of 8 compared to all other versions, while our multicore implementation has a maximum speed-up of 4 compared to Aleph-all.

Note that this evaluation was also performed with an earlier version of GPU-Datalog (similar to the one used in Section 5.1). We believe that the optimizations in the latest version of GPU-Datalog and in the Thrust library should lead to greater speed-ups for Aleph-cuda, especially since Aleph has not received significant optimizations to its code.

5.3 GPU-Tuffy and GPU-RockIt

This section presents the performance evaluation of our GPU-Tuffy platform with parallel grounding and search, of our RockIt-based platform with parallel grounding (GPU-RockIt), and of our parallel MC-SAT algorithm used during weight learning (included as part of GPU-Tuffy).

5.3.1 Applications

Our performance evaluation used four applications included in Tuffy's installation [136] and two of our own creation. Some applications were extended with additional, randomly generated data. Table 5.4 shows their general characteristics.

Smokers (SM) is a 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 [98]). The original number of tuples for SM is only eight. We created another dataset 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.

Relational Classification (RC) classifies papers into 10 categories based on authorship and on the categories of other papers it references (Cora dataset [76]). The randomly generated dataset uses a fixed number of papers and authors, the same categories found in the original data and constructs the dataset as follows: 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 labelled as belonging to a particular category.

Census (CS) was created by us and determines, given a 1% random sample of the 1999 US census available at [142], if a certain person would be drafted in case of war (the US ended conscription in 1973 but the "Selective Service System" remains as a contingency plan) based on ancestry (Hispanic, Korean or other immigrant); income (occupation, scholarship and work hours); family members (brothers, spouse and children) and physical attributes (women, young, elderly and disabled would not be drafted). The distinctive characteristic of this application is that most of its data is stored in a single, very large table with several attributes.

Hospitals (HL), is another application created by us using the US hospitals database found in [141]. The task is to find the best hospitals based on the rate of complications, readmissions and infections; an overall score given to each hospital by its patients; if care was timely and effective; and if the payment correspond to the value of the care received. Furthermore, in order to create a recursive application, each hospital has a certain number of nearby clinics and pharmacies that are considered good if the hospital is good, and if a hospital is near good clinics or pharmacies, then it is also considered good. The size of the HL database was increased through

adding additional hospitals with random attributes and all clinics and pharmacies. The clinics and pharmacies have only two attributes: an Id and the Id of the hospital near them.

Link Prediction (LP) determines student-advisor relationship from a database of several university departments [98]. The database includes information such as taught courses (for professors), attended courses (for students), publications (for both), among others. The size of the original database was quadruplicated by appending numbers to each element of the database (e.g., for each `publication (Title,Person)`, we created `publication(Title1,Person1)`, `publication(Title2, Person2)` and so forth), seeking to preserve the structure of the data.

Entity Resolution (ER) removes duplicates from citation records (title, author, venue, etc.) based on similar words [108]. The original size of the data was unaltered as its large number of formulas proves to be quite challenging.

Hardware-software Platform

We ran our experiments in the following hardware and software:

Host hardware. AMD Opteron 6344, 12 cores CPU, with 64 GBs of RAM.

Host software. CentOS 7, MySQL 5.6, PostgreSQL 9.5, YAP Prolog 6.3.

GPU hardware. Tesla K40c, 2880 CUDA Cores, with 12 GB GDDR5 memory and CUDA Capability 3.5.

GPU software. CUDA Toolkit 7.0.

5.3.2 Results

Next we present the results of the evaluation of our GPU-based designs. Note that all graphs (except Figure 5.15 and Figure 5.16 left) are in logarithmic scale.

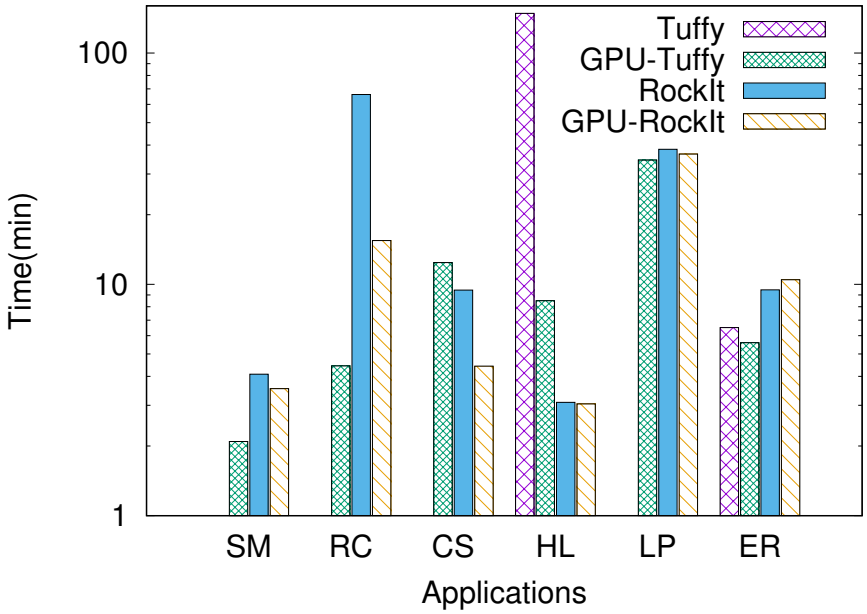


Figure 5.4: Performance of the GPU and CPU platforms on six applications.

5.3.3 General Performance Evaluation

Figure 5.4 shows the performance of our GPU platforms against their base systems, all of them using their default configurations except for RockIt and GPU-RockIt in LP and ER, where their default error bound had to be increased (thus lowering the expected quality of the solution), otherwise their execution would not finish after more than 5 hours. Also, note that Tuffy is not present in all applications as it could not finish grounding after more than 5 hours.

In general, our platforms performed very well, with either or both of them outperforming the CPU-based platforms in all applications. In RC, their best performing application, GPU-Tuffy is 15 times and GPU-RockIt is more than 4 times faster than the fastest CPU platform (RockIt). 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 like LP and ER.

The results also confirm that no single approach is best for all applications, as GPU-Tuffy which is based on two large groundings and satisfiability performed better

in SM, RC, LP, and ER, while GPU-RockIt which is based on several small groundings and optimization did better in CS and HL. However, how to determine which approach is better for a certain application beforehand is not an easy task that might require a careful analysis of the characteristics of said application like the depth of its recursive clauses and the amount of contradicting evidence.

5.3.4 Performance of GPU Search with MaxWalkSAT

To evaluate the performance of our GPU MaxWalkSAT algorithm in context, we integrated said algorithm into our previous version of Tuffy with GPU grounding only, G/GPU-Tuffy. To the new version with the GPU MaxWalkSAT algorithm used during search, we refer to as GS/GPU-Tuffy. Figure 5.5 shows the performance of G/GPU-Tuffy and GS/GPU-Tuffy with all applications described above in minutes. GS/GPU-Tuffy is always faster (up to 35 times faster for LP) than G/GPU-Tuffy for all applications, with a large enough Markov Random Field (MRF) partition (recall that Tuffy partitions the MRF to process it in parallel, but such a process usually leaves a partition that is several orders of magnitude bigger than the rest). In general, GS/GPU-Tuffy provides great speed-ups in those applications that have large, recursive groundings (like SM and RC) and/or a large MRF partition with thousands or millions of active clauses and active atoms (like SM, RC and LP).

A comparison on those applications with a large MRF partition between our GPU MaxWalkSAT solver and Tuffy's default MaxWalkSAT solver is shown in Table 5.5. For each application, Table 5.5 shows the number of atoms and clauses of that large partition, the time to process it, and its solution cost. The solution cost is computed as the sum of the weights of all unsatisfied clauses, which means that *the lower the cost, the better a solution will be.* Also, depending on the problem, it is not possible to know what the lowest solution cost is because satisfying some clauses may require breaking others. Nevertheless, our solver provides better or similar solutions in a lower amount of time.

About the performance of our GPU algorithm, the frequent, random memory

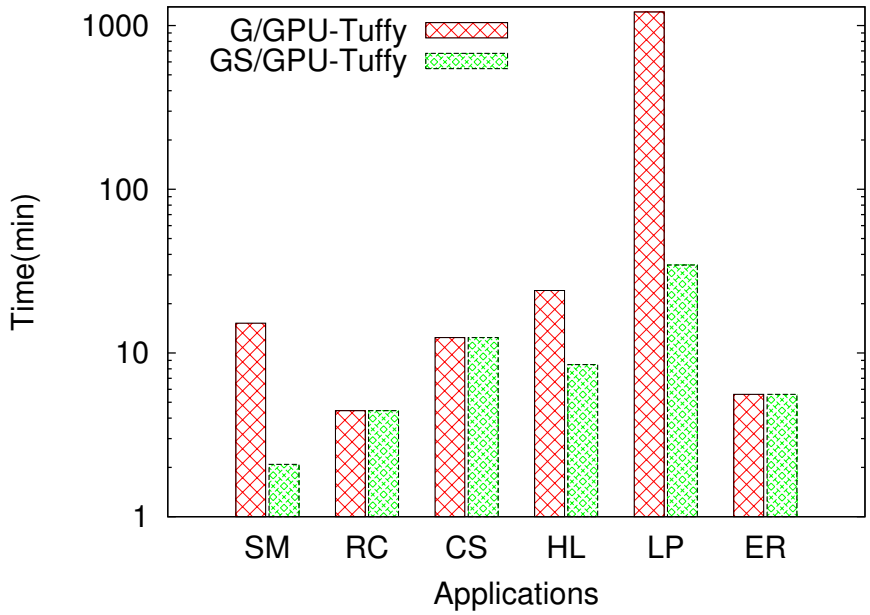


Figure 5.5: Total running time in minutes of G/GPU-Tuffy and GS/GPU-Tuffy in six applications.

Table 5.5: Time and cost comparison between the GPU and the CPU based MaxWalkSAT solvers for SM, HL and LP. The lower the cost the better the solution is.

	Largest partition		GPU solver		CPU solver	
Application	Atoms	Clauses	Time	Cost	Time	Cost
SM	451,888	774,038	1.21	0.80	14.33	6387.30
HL	662,101	1,337,439	2.02	0.0	17.21	0.0
LP	68,905	9,236,664	24.23	49224.80	1198.5	49160.72

accesses of the original MaxWalkSAT coupled with the multiple atom flipping of our algorithm, made it very difficult to adapt for GPUs: poorly coalesced memory accesses, atomic operations, and conditional statements which cause great divergence amongst threads, are some of the issues adversely affecting performance. However, we found that flipping multiple atoms at the same time has several benefits like reducing the number of times the flip effect is computed (i.e., determining which clauses are still unsatisfied and recalculating the flip cost metadata), as it is done once for many atoms, rather than once for each atom. Also, when the solver is far from the solution (i.e. there are many unsatisfied clauses), multiple flips force the number of unsatisfied clauses to decrease faster as many poor, intermediate solutions are skipped. The algorithm could also benefit from a better heuristic to pick multiple atoms for flipping, based perhaps on the number of times an atom appears in the clauses.

5.3.5 Performance of GPU Grounding for RockIt

To test the flexibility of our GPU-based grounding, we integrated it into the RockIt system, creating GPU-RockIt. The performance of GPU-RockIt was evaluated and compared against RockIt using six applications. Figure 5.6 shows the total running time of both systems on said applications in minutes. GPU-RockIt performs well in the first three applications (particularly RC) but does not provide a significant speed-up in the other three.

To understand these results, recall that RockIt's inference is composed of several iterations and each iteration requires grounding over different evidence. While some of these groundings (specially the first one) are performed over a large amount of evidence and greatly benefit from GPU processing, others are performed over such a small amount of evidence that the database and GPU I/O times greatly reduce any benefit of GPU processing.

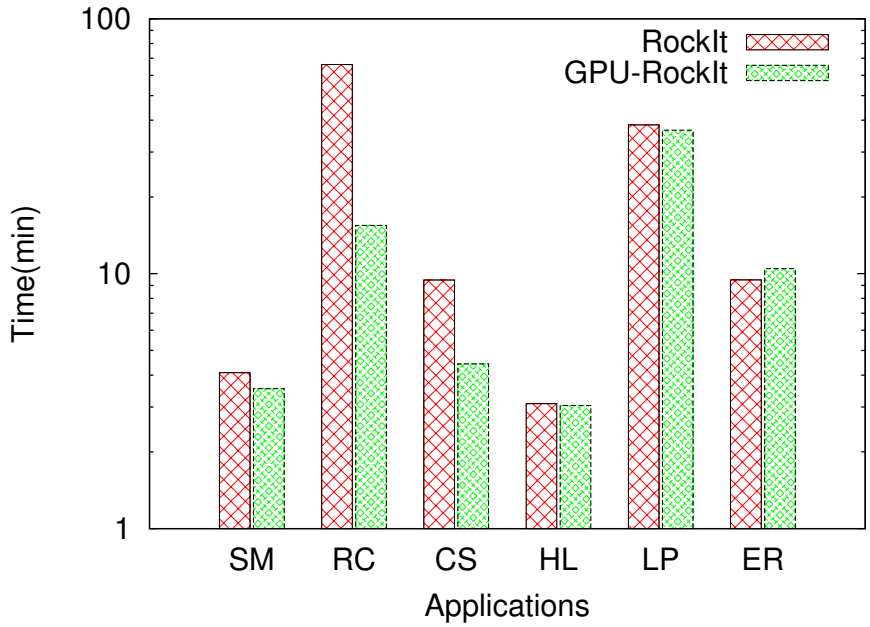


Figure 5.6: Total running time in minutes of RockIt and GPU-RockIt in six applications.

5.3.6 Performance of GPU MC-SAT under Weight Learning

Our next contribution is the integration of a GPU-based MC-SAT solver for weight learning into GS/GPU-Tuffy. As weight learning also involves grounding, to have a fair comparison where the grounding time is the same, we compared our GPU solver against G/GPU-Tuffy using Tuffy's default CPU MC-SAT solver. The total running time for five applications using 10 MC-SAT iterations, each one calling SampleSAT 10 times, is shown in Figure 5.7. Our GPU MC-SAT solver is faster in the first three applications but lags behind in the other two. Note that CS was not considered as its MRF is too small to benefit from GPU processing.

To understand these results, we measured the number of active atoms and active clauses found in the MRF, along with the average time and solution cost of the SampleSAT iterations. As shown in Table 5.6, our GPU SampleSAT solver is the reason the overall GPU MC-SAT algorithm is faster in the first three applications and also produce much better results when no zero cost solution can be found. However, for the last two applications, GPU SampleSAT performance is severely affected by the relatively low number of atoms and the Simulated Annealing (SA)

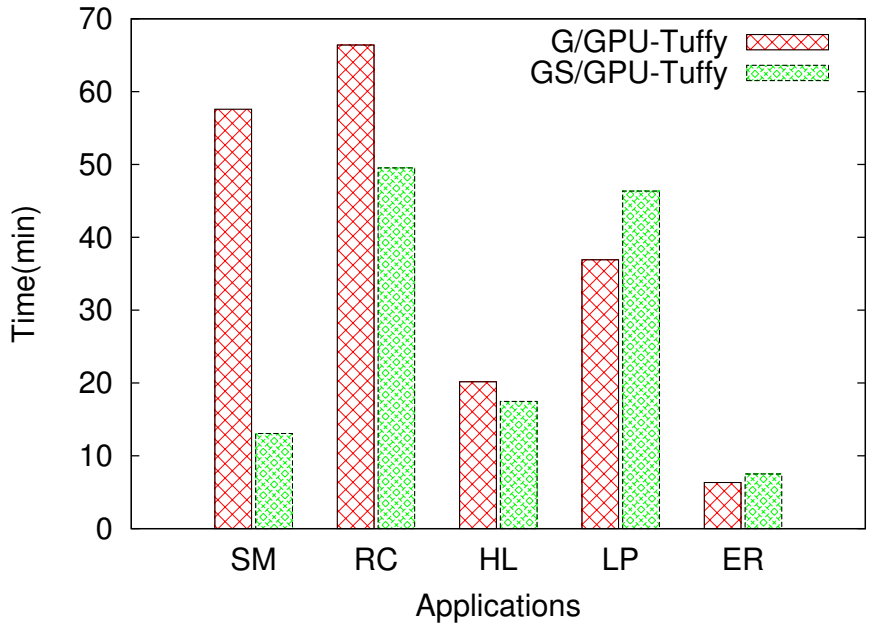


Figure 5.7: Total running time in minutes of G/GPU-Tuffy using Tuffy's default CPU MC-SAT and GS/GPU-Tuffy using our GPU MC-SAT for weight learning in five applications.

on multiple flips, which is performed during each iteration. This happens because, when the number of unsatisfied clauses is greatly reduced, but it is not enough to stop the GPU SampleSAT solver and transfer the computation to the CPU SampleSAT solver, the GPU solver slows down on a series of "halfway solutions" still far from the solution. These halfway solutions occur more often on applications with a low number of atoms because, while some threads (those using MaxWalkSAT steps) push to a better solution, others (those using SA steps) have a greater probability of incorrectly flipping a critical atom that may worsen the solution (recall that SA can flip any atom), thus leaving SA in a halfway point. Possible solutions to this problem include changing the computation to the CPU at an earlier point or having all threads perform either a MaxWalkSAT step or a SA step at each iteration.

5.4 Discussion

In order to gain a deeper understanding of our systems, we measured the time spent at each phase: loading, grounding and searching by each system. Loading refers to

Table 5.6: Time and cost comparison between the GPU and the CPU based SampleSAT solvers for all applications except CS. The lower the cost the better the solution is.

	MRF size		GPU solver		CPU solver	
Application	Atoms	Clauses	Time	Cost	Time	Cost
SM	473,613	816,859	1.46	18463.61	7.44	26567.36
RC	500,000	4,633,040	4.51	106370.70	5.43	278220.63
HL	735,923	1,411,261	0.59	0.0	1.33	0.0
LP	72,361	9,877,479	3.33	0.0	2.36	0.0
ER	68,286	357,374	0.41	0.0	0.32	0.0

the time spent reading the evidence and MLN files, creating and filling the tables in the RDBMS and translating the MLN to Datalog. Figure 5.8 shows the execution time of each phase for each system under each application. Note that *GT* stands for GPU-Tuffy, *GR* for GPU-RockIt and *RIT* for RockIt.

For SM, grounding times for all systems are similarly small compared to the inference time. However, inference is faster in GPU-Tuffy thanks to the GPU MaxWalkSAT implementation. In RC, its recursive grounding is far more time consuming, with RockIt's time suffering greatly as a result and GPU-RockIt proving to be very effective. In CS, while grounding time is similar for both GPU-based systems, inference in GPU-Tuffy takes longer as it runs over a large number of small partitions that do not benefit from GPU acceleration. For HL, the RockIt-based systems beat GPU-Tuffy in both grounding and inference times.

The last two applications, LP and ER, are characterized by their relatively small groundings and complex clauses with several contradictions. These contradictions adversely affect the stopping criteria (number of unsatisfied clauses in GPU-Tuffy and the *gap* between expected and obtained solutions in both RockIt-based systems) of the search algorithms, causing the search time to dominate the execution time (particularly in ER). Furthermore, while GPU-Tuffy can automatically stop after an iteration limit based (by default) on the size of the application, the RockIt-based systems require manual adjustments to the *gap*, else the search may take an exceedingly high amount of time. To have a fair evaluation, we picked the lowest

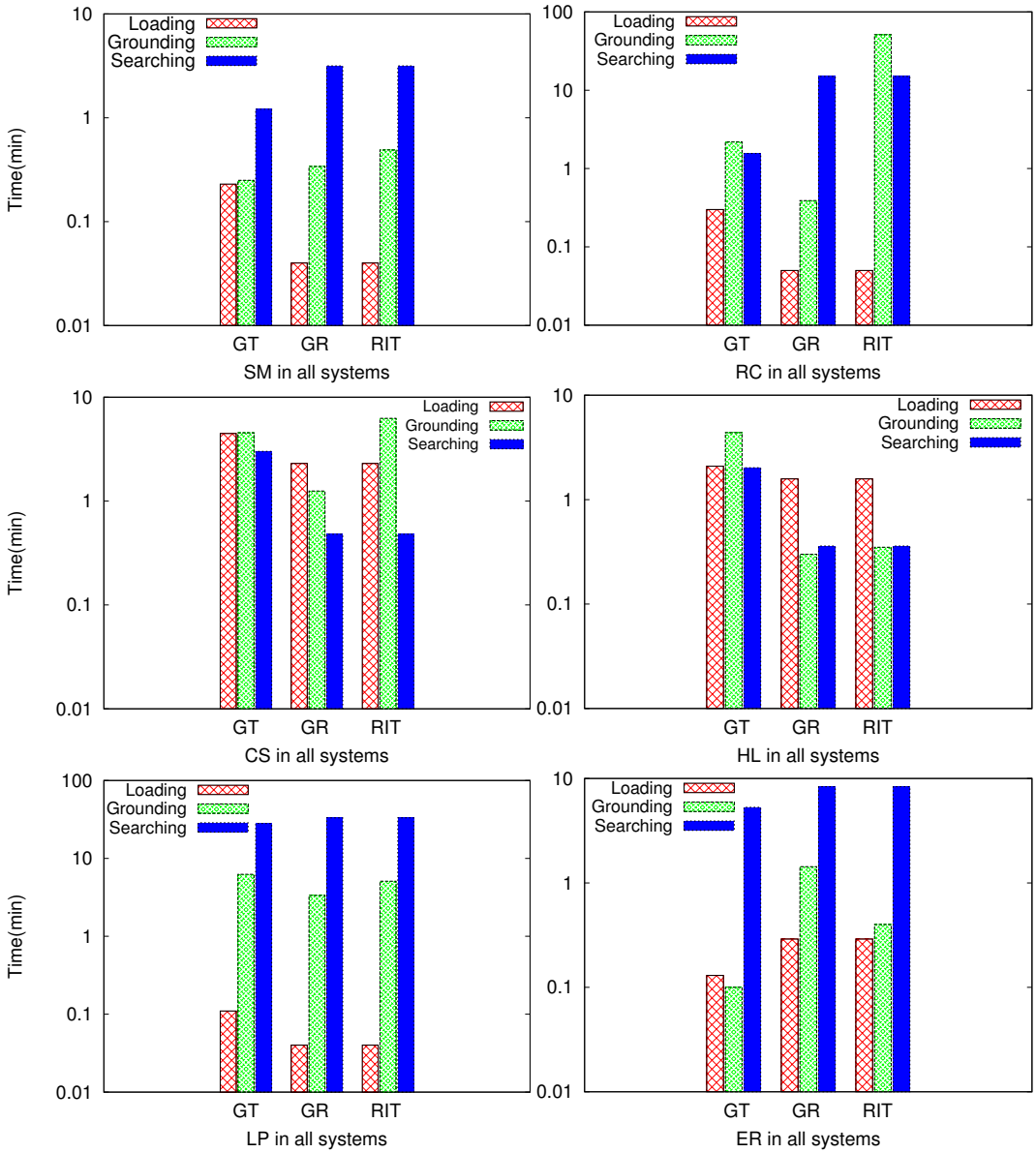


Figure 5.8: Time spent in loading, grounding and searching by each application in each system: GPU-Tuffy (GT), GPU-RockIt (GR) and RockIt (RIT). Note that RockIt and GPU-RockIt times for loading and inference are equal, only the grounding is different.

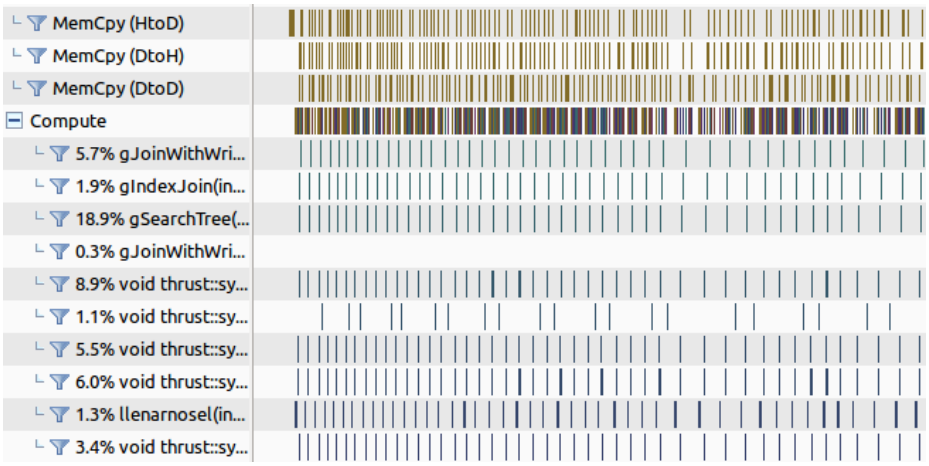


Figure 5.9: Nsight profile for SM.

Figure 5.10: Nsight profile for RC.

possible *gap* that would not cause RockIt to get stuck searching. Otherwise RockIt's and GPU-RockIt's times for these applications would have been much higher.

5.4.1 GPU Kernel Profiling

To determine the performance of our GPU kernels, we obtained the GPU usage for each kernel with the CUDA profiler tool for the grounding on GPU-Tuffy, part of which is presented in Figures 5.9 to 5.14. The first part of the figures shows memory transfers from the host to the GPU (HtoD), from the GPU to the host (DtoH), and within the GPU (DtoD). The next part is the timeline of all kernel calls and the percentage of GPU time each of them occupied. Kernels starting with `thrust` belong

Figure 5.11: Nsight profile for CS.

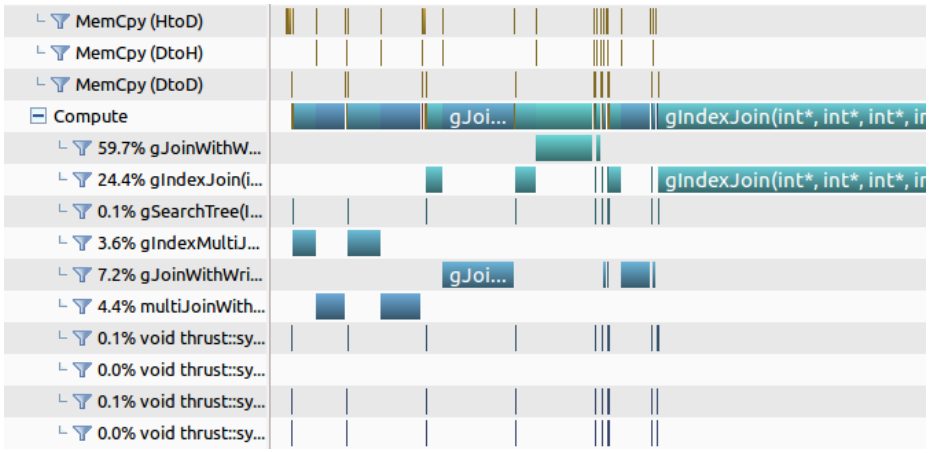


Figure 5.12: Nsight profile for HL.

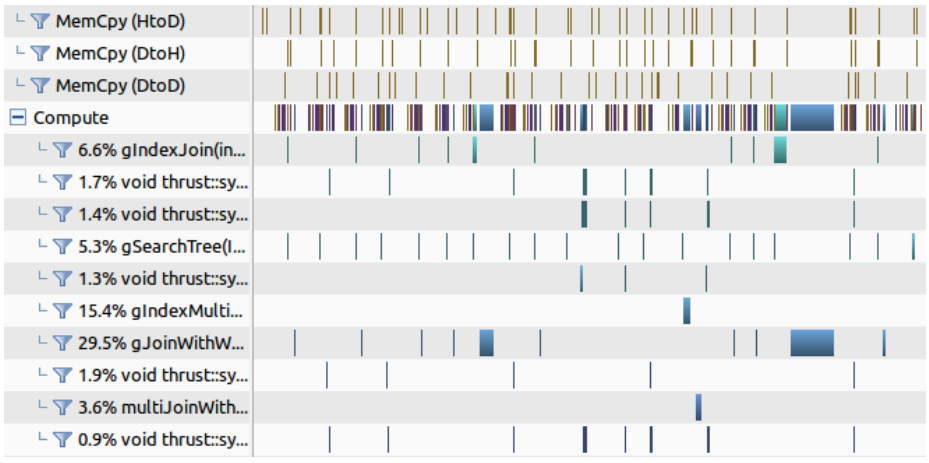


Figure 5.13: Nsight profile for LP.

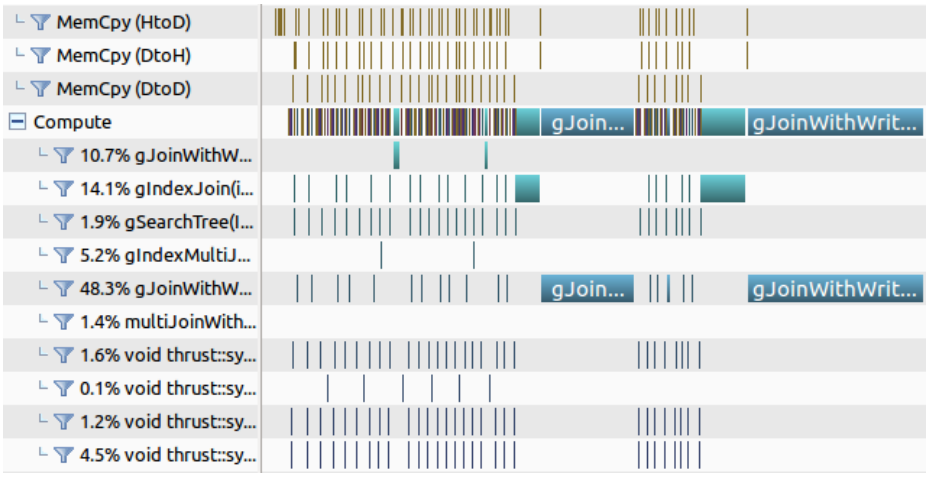


Figure 5.14: Nsight profile for ER.

to the Thrust library and include sorting, prefix sums and duplicate removal, the other kernels are RA operations (especially joins) or auxiliary kernels. Note that the graph indicates the relative size of the kernels where each bar represents a kernel call and most of the kernels have a relatively small size compared to the compute intensive join kernels.

The full GPU usage of all inference kernels in both platforms (including GPU-RockIt's grounding and GPU-Tuffy's MaxWalkSAT, whose profile graphs are not presented for brevity) is resumed in Figure 5.15 by dividing them in three categories: *Joins* including single, multiple, and negative joins along with their auxiliary kernels; *Thrust* which includes all operations of the Thrust library; and *Other* kernels used for selections, comparisons, etc. As shown in Figure 5.15, for most of the applications the GPU is mainly used by the *Join* kernels. While these results are not surprising as joins are costly operations [2, p. 105], we decided to further analyse the most compute intensive join among them.

The single most compute intensive join kernel in our systems writes and projects a multiple join (a join over more than one column that projects the result by eliminating unnecessary columns, as described in Section 4.1.1), taking around 60% of the total GPU usage. Table 5.7 shows the most important performance aspects of this kernel. The low compute and load/store throughput (recommended values are around 60%),

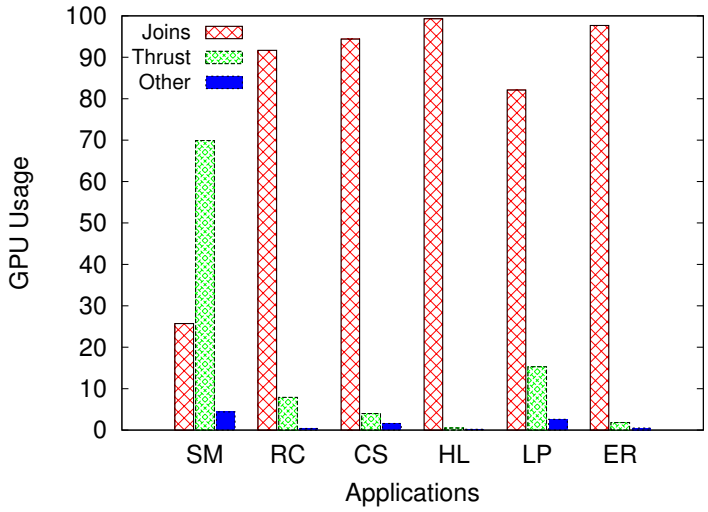


Figure 5.15: GPU usage of all kernels grouped into three categories for each application.

suggests possible improvement by increasing occupancy and/or reducing instruction stalls. Occupancy is measured as the number of active warps (groups of threads) through the kernel lifetime and is not limiting our kernel performance since it is close to the theoretical maximum. The global memory bandwidth is also not a limiting factor as it is not being overused. While warp efficiency is reduced due to divergent operations (i.e., loops, conditions, etc.) and slightly impacts performance, the most important improvement would be to reduce instruction stalls caused mainly by memory throttle (i.e., too many memory transactions that cannot be executed in a timely manner). Unfortunately, we believe it would be hard to reduce this memory throttle as the kernel's main function is to read and write memory locations, and the compute throughput is simply low because few arithmetic and control operations are required.

5.4.2 GPU Memory Usage

We also measured the maximum amount of GPU and CPU memory required in our experiments, as we are interested in allowing our systems to run on commodity hardware which typically has 2GB of GPU memory. Figure 5.16, left, shows maximum GPU memory usage by each system for each application, including grounding (GPU-

Table 5.7: Performance results of the multi-join kernel, the most demanding kernel in our systems. Note that memory throttle is a cause for instruction stall and a lower percentage is better for the overall performance of the kernel.

Variable	Achieved	Maximum
Compute throughput	18%	100%
Load/store throughput	55%	100%
Active warps	62.93	64
Occupancy	98.3%	100%
Memory bandwidth	86.853GB/s	288.384GB/s
Warp efficiency	78.9%	96.1%
Memory throttle (stall)	75%	-

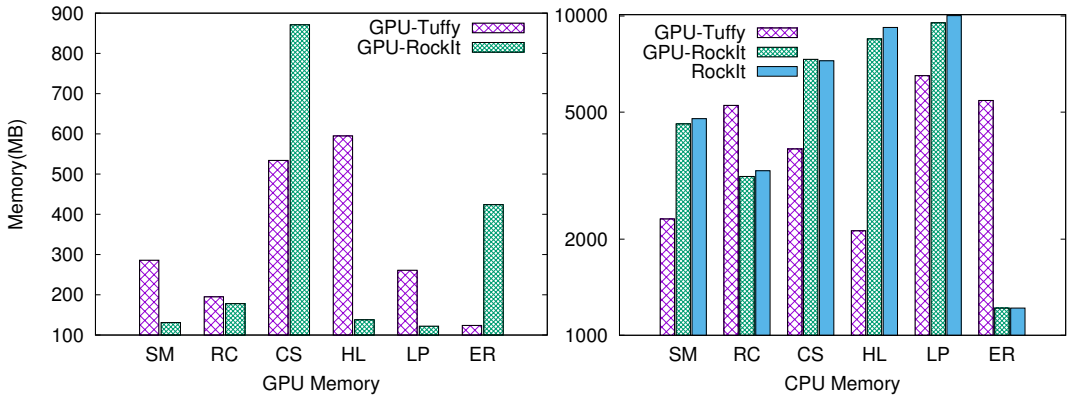


Figure 5.16: GPU and CPU memory used by each system for each application.

Datalog) and search (GPU MaxWalkSAT). Memory usage is higher for GPU-Tuffy except for CS, but neither system requires more than 1GB. Thus it is possible to use even low-end GPUs from the 600 series and up.

Figure 5.16, on the right, also shows CPU memory usage by each system under each application. GPU-Tuffy requires the least CPU memory for all applications except RC, for which it needs about 5GB. GPU-RockIt and RockIt have similar CPU memory usage, meaning that our GPU-based grounding does not affect much RockIt's CPU memory requirements. Overall, our systems can be used in CPU commodity hardware with 8GB, although RockIt and GPU-RockIt may need more memory for the larger applications.

5.4.3 Scalability

The memory results shown above suggest that our systems should be able to handle applications with larger amounts of data. However, GPU memory is the limiting factor for the scalability of our systems, as once an application runs out of GPU memory, it simply fails (there is no virtual memory or anything similar for GPUs). Possible solutions to this problem include: using CUDA's mapped memory which maps CPU memory as an extension of GPU memory or partitioning the data before transferring it to the GPU. Since mapped memory usually incurs large performance penalties, we believe data partitioning is the way to go.

Also, regarding the execution of our platforms in other GPUs, the main limiting factor besides the amount of GPU memory is the compute capability. The compute capability (explained in Section A.1) is a number that currently goes from 1 to 6 and determines some important characteristics of the GPU like the amount of shared memory and the operations it can perform (e.g., some atomic operations are only available for devices with high compute capabilities). Our platforms cannot be executed on GPUs with compute capability 1, as it corresponds to very old GPUs that are no longer supported by the latest CUDA toolkit. Initial version of our platforms where tested on a GPU with compute capability 2, but we recommend the use of GPUs with compute capabilities 3 and up.

5.5 Summary

We evaluated the performance of our GPU platforms, GPU-Tuffy and GPU-RockIt, and their core components: GPU-Datalog for grounding, GPU MaxWalkSAT for search, and GPU MC-SAT for weight learning (the last two are part of our GPUSATLIB). We also evaluated our parallel ILP system based on Aleph and GPU-Datalog. Our evaluations were performed over a wide variety of hardware and used several applications both from literature and real-life. For most of these applications, their original data was extended with additional random tuples.

GPU-Datalog was evaluated against a similar system called Red Fox, providing mixed results. However, we note some problems in the evaluation, while also providing possible performance improvements to our system. Our ILP system performed much better, improving upon its highly optimized CPU counterparts by providing a speed-up of up to 8 times.

Our parallel MaxWalkSAT solver used in GPU-Tuffy performed very well, providing a speed-up of up to 35 times compared to the CPU version and finding equal or better results. Parallel grounding in GPU-RockIt provides good speed-ups in three applications, but fails to provide a significant performance gain in the other three. The reason is the different sizes of the groundings performed at each iteration of the inference algorithm, with small groundings adversely affecting performance due to database and GPU I/O overheads. Finally, our parallel SampleSAT algorithm used during weight learning in GPU-Tuffy provided great speed-ups in three applications, but lagged behind in the other two (the CS application was not used). The problem with these two applications lies in their small number of active atoms and the parallelisation of the SampleSAT algorithm. We get stuck in "halfway solutions" as some threads try to push to a better solution while other try to sample around the current solution.

In order to gain a deeper understanding of our systems, we measured the time spent in each phase: loading, grounding and searching by each system. While loading never dominates the total execution time, either grounding or search can dominate depending on the application and the system that executed it. Also, to determine the performance of our GPU kernels, we obtained the GPU usage for each kernel with the CUDA profiler tool. Profiling shows that joins are the most compute intensive operations, taking up to 90% of the total GPU processing time, however, their conditional and memory operations make it difficult to improve them. Our last considerations were the amount of CPU and GPU memory used by our systems and their scalability. While GPU memory is the main limiting factor for scalability, in the tested applications none of our system requires more than 1GB of GPU memory and

only some applications require more than 8GB of CPU memory, thus our systems are well suited to be executed in commodity hardware.

Chapter 6

Conclusions and Future Work

This chapter presents our conclusions on our designs and their performance (Section 6.1), followed by our contributions and what we consider the most promising future work along with possible implementations for MLNs in general, the core components of our MLN platforms, and our MLN platforms themselves (Section 6.2).

In order to better understand this chapter, recall that MLNs are a powerful framework but their processing time is unacceptably high and the solution quality is very poor on larger applications. Our proposed solution was to accelerate in GPUs the most compute intensive tasks of the MLN process: *grounding* which involves assigning values to all variables in the formulas given some evidence data; *search* which finds, for each ground formula, if said formula is true or false by solving a weighted satisfiability problem or an integer linear programming problem; *weight learning* where the formulas are first grounded and then the optimum value of the weights is determined by solving an optimization problem whose parameters are adapted through sampling; and finally, *clause learning* uses Inductive Logic Programming (ILP) were the worth of candidate formulas is determined in a process similar to grounding.

6.1 Conclusions

We achieved the main objective of this work, which was the design, implementation and evaluation of two MLN platforms (GPU-Tuffy and GPU-RockIt) and their core components based on GPUs. Results (Section 5.3.2) have showed that inference in some applications (like those with deep recursive clauses) is better formulated as a *satisfiability* problem, while other are better formulated as mathematical *optimization* problems. The designs for each platform and their core components are defined with several algorithms and examples, ample insight on the issues faced (particularly those related to parallel processing on GPUs), and our proposed solutions. The implementation of said designs was performed following important practices including: *modularity*, which the various interchangeable components preserve; *portability*, provided by the use of common languages and libraries that can be easily compiled; *validity*, that was demonstrated through rigorous testing; and *usability*, by maintaining the same interfaces of the base platforms and components.

The performance of our platforms is on par or significantly better than that of RockIt, currently the fastest CPU-based MLN system. The results in Section 5.3 show that the benefit of performing the grounding phase on the GPU outweighs the overhead of GPU and database I/O, even for rather small datasets. Said results also show that for large weighted SAT problems, GPU processing delivers better results in less time. Furthermore, the analysis of said results, along with some additional testing, proved to be useful as they: confirmed that grounding and search are the most time consuming tasks of the MLN process; assured us that our platforms can run on commodity hardware; provided insight on the scalability of our platforms; and prompted us to formulate various possible improvements to our platforms and their components, which are discussed in detail in the following section.

The designs of our platforms and the platforms themselves constitute, to the best of our knowledge, the *first* GPU-based MLN processing infrastructure. Their core components are also a *first* GPU-based solution for their specific applications.

Moreover, these core components can be used as stand-alone systems for other applications besides MLNs.

Finally, GPUs are nowadays widely used for traditional machine learning algorithms [7]. Our results show that GPUs are also a natural fit for Statistical Relational Learning (SRL) tasks, and should be considered for other SRL computationally intensive tasks.

6.2 Contributions and Future work

To better explain our contributions, we present in Figure 6.1 (similar to Figure 1.1) the phases of MLN processing and their steps, with our used solutions and the hardware that executes them. We believe these contributions have advanced the state of the art of MLNs, Datalog, ILP, satisfiability, and GPUs, and will be useful for current and future real-world applications.

6.2.1 MLNs

Contributions

Our contributions to MLNs include GPU designs for the most time consuming parts of MLN processing:

- *grounding* based on the Datalog language (a language similar to Prolog) which was extended with comparisons, negations, arithmetics, and aggregations (a) and d) in Figure 6.1).

- *search* as a partitioned, weighted MaxSAT problem where the smaller partitions are solved by the MaxWalkSAT algorithm on the CPU cores and, at the same time, the larger partitions are solved on the GPU by a modified version of the MaxWalkSAT algorithm where several atoms are flipped (their truth values changed from true to false or vice versa) at the same time (b) in Figure 6.1).

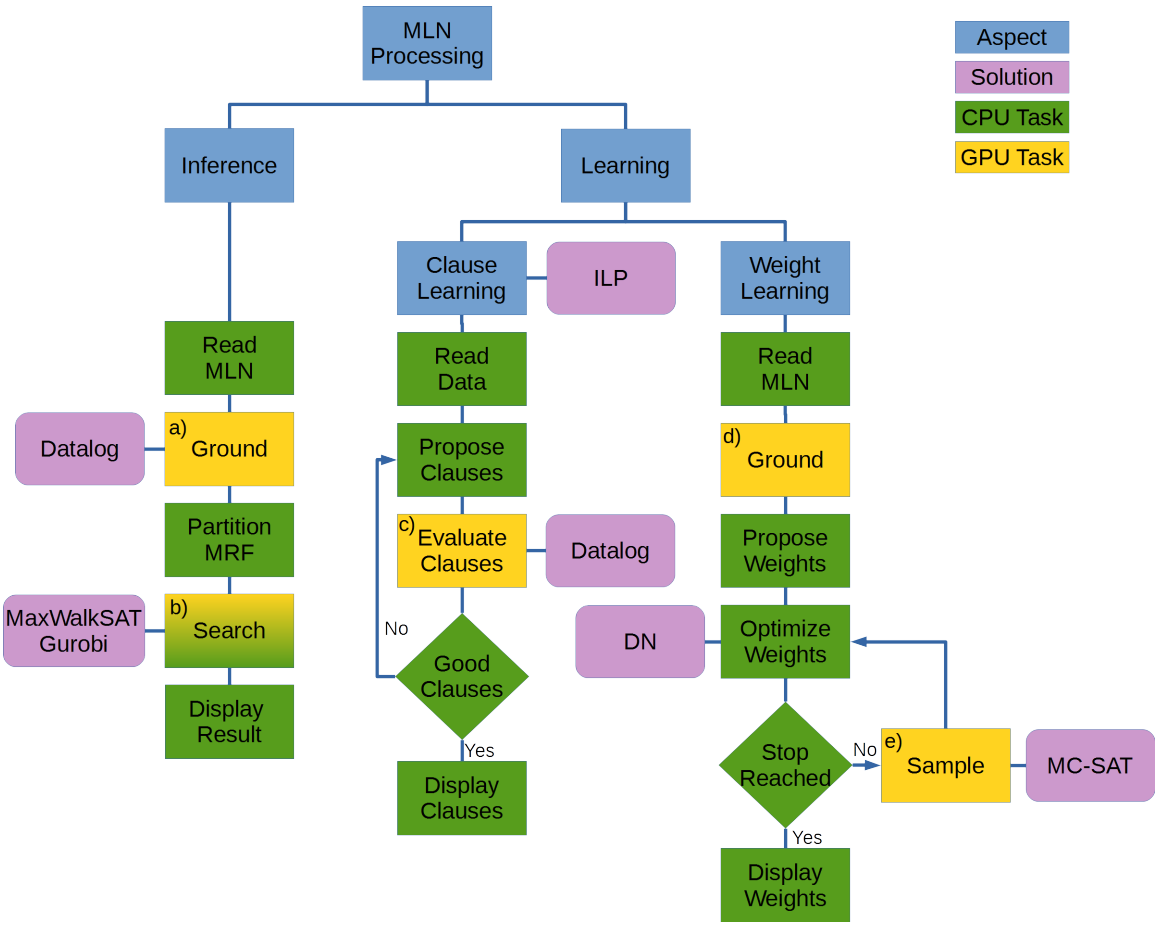


Figure 6.1: Phases and steps of MLN processing with our solutions and hardware used.

- *weight learning* using the Diagonal Newton optimization method with MC-SAT sampling which is based on MaxWalkSAT with Simulated Annealing (e) in Figure 6.1).

- *FOL clause learning* with Inductive Logic Programming (ILP) where coverage (the number of right and wrong examples that a proposed clause outputs as its result) is computed with Datalog (c) in Figure 6.1) and could be extended to MLNs by changing the clause evaluation function.

Future Work

The current test suite available for MLNs is adequate to show their potential and to evaluate new ideas and platforms. However, we consider that, as more and more high-quality data becomes available, the current suite may appear outdated. Thus, we propose the creation of a larger collection of test applications including big data applications. Some of such applications should represent real-work problems, while others should stress particular aspects of the MLN process like using lots of clauses with infinite weights to test the satisfiability solver, or clauses with lots of predicates to test the grounder. There are several different domains that could be considered for said applications like natural language processing, e-commerce, and social networks. As an example of this task, consider the datasets in [131] which could be used to generate MLNs that predict if a person would like a certain restaurant or the nationality of the person who posted a certain comment. The rules for these datasets should be hand-crafted first, and then refined using weight and clause learning.

MLNs are related to many other logic and probabilistic approaches [24, p.19] like the Knowledge-based model construction (a combination of logic programming and Bayesian networks) and Stochastic logic programs (logic programming with log-linear models). The program representations used by some of these approaches can be easily translated into MLNs. Furthermore, the applications of said approaches are quite useful and would nicely extend the MLN repository. For these reasons, it would be convenient to create a tool to automatically translate their programs and data into

MLNs. Said tool would also allow interesting combinations between approaches, as different parts of an application could be performed by different approaches. Since an MLN is essentially a weighted Conjunctive Normal Form (CNF, a conjunction of one or more clauses, where a clause is a disjunction of literals), the work on Problog (a probabilistic logic programming language similar to MLNs) by Daan Fierens et al. [26, 25] could be used as starting point for the translation tool, as the author describes the Problog to CNF conversion and inference, while also suggesting that said conversion can be used for MLNs and other similar languages.

6.2.2 GPU-Datalog

Contributions

GPU-Datalog (Section 4.1) is our parallel Datalog engine based on bottom-up relational algebra (RA) operators. It was extended with additional operators like negation and comparison in order to accelerate the grounding on MLNs (a) and d) in Figure 6.1) and the evaluation on clauses in ILP (c) in Figure 6.1). It can also be used for general and extended Datalog programs.

Future Work

GPU-Datalog could benefit from the *magic sets* [6] and *counting* [102] transformations. Said transformations rewrite any Datalog program with additional rules that eliminate the computation of unnecessary rows by considering the query at all times, similar to the top-down approach (in contrast, GPU-Datalog only considers the query at the end) while maintaining the benefit of bulk operations from the bottom-up approach. In GPU-Datalog, these transformations could be implemented as described in the papers and incorporated into the *Preprocessor* of the Preparation Stage (see Section 4.1). These transformations would not require modifications to the GPU kernels as they only add rules or rewrite them with additional variables and/or predicates.

About the RA operators, most of them would greatly benefit from good algorithms for sorting, prefix sums, and duplicate elimination. Currently, several open libraries provide these algorithms, being Thrust [151], CUB [132], and ModernGPU [144] the most prominent. The latest comparisons between these libraries [79] show that no library is better for all cases (although CUB's performance seem to be the overall best). Thus, we propose a new comparison between these libraries by integrating them into GPU-Datalog and checking which one is better for each operation, as it might be possible that one library is better than the other in some operations and vice versa. Such integration might require compiling the libraries, linking them to GPU-Datalog, and modifying GPU-Datalog's code to use their functions.

Another possible improvement to RA would be the removal of duplicates earlier in the computation, as they generate many unnecessary tuples and increase processing time. Said removal could be performed after each join operation by calling our already implemented duplicate elimination function. However, duplicate elimination is a costly operation and thus, better results might be obtained with a heuristic that decides when to apply duplicate removal based perhaps on sampling the results of the join.

Since bottom-up processing allows GPU-Datalog to process rules independently and in any order, another performance improvement could be the mixed processing of rules by both the CPU and GPU (e.g., have the CPU process rule 1 in a program while the GPU is processing rule 2 at the same time). Currently, we have an outdated version of GPU-Datalog for multicores (described in Section 4.1.3) that only includes the basic operations (joins, selections, and comparison predicates). This multicore version should be updated with negations and arithmetic predicates, and should be integrated into the GPU version though an algorithm that determines which hardware should evaluate a certain rule, based perhaps on the size of the input, an approximation of the size of the output and whether the rule is recursive or not. Large, recursive rules are to be assigned to the GPU and it should be possible to reassign rules to the opposite hardware (i.e., from the CPU to the GPU and vice

versa) if necessary, always considering that such reassignment is convenient in some cases: when a rule in the CPU is taking too long (in which case it should be moved to the GPU) or when a small recursive rule is not fully utilising the GPU and should be moved to the CPU. However, note that reassignment is a costly operation that requires moving all data (facts) related to the rule from the memory of one hardware to the other.

Also, the implementation of our partitioning scheme has been completed, but needs to be extensively tested to eliminate any possible bugs, ensure the validity of its results, and quantify its performance increase. Further improvements to this scheme should be based around estimating the right size of the partition for each operation. Such estimation could consider the result of performing the operation over a small subset of the data, reducing the size if the result is large and increasing it otherwise. For recursive applications, the results of earlier iterations of the same rule could also improve the estimation as the same operations are always performed each time the rule is iterated (only the data over which said operations are performed changes).

6.2.3 GPUSATLIB

Contributions

GPUSATLIB is our GPU library for satisfiability based on the MaxWalkSAT algorithm (Section 4.2.1) and sampling based on SampleSAT (Section 4.2.2). These algorithms improve both the processing time and the solution quality of inference and weight learning in MLNs (b) and e) in Figure 6.1). They can also be used for general satisfiability and sampling problems.

Future Work

GPUSATLIB could benefit from the work of McDonald [77] (described in more detail in Section 4.2.3 and described in the context of SAT solvers in [38]). The idea is

to incorporate a technique called *conflict driven clause learning* that constructs a shared database of clauses, which is used to learn dependencies among the variables. The knowledge of such dependencies would allow our algorithms to set the correct truth values for certain variables at each iteration, thus reducing the search space for the remaining ones. One important aspect to consider in order to incorporate this technique is the handling of the clause weights, as McDonald's work does not consider them. We believe that the truth value of a variable would have to be set based on said clause weights.

Also, we considered two approaches for parallelising our algorithms (presented in Sections 3.4 and 4.2.1): 1) to have all threads work together towards a single solution or 2) have each thread work on its own to find a solution and then use the best one. Our algorithms are based on the first approach because the second one has almost impossible memory requirements (as each thread needs a large memory space), but it should have a better coverage of the search space. Thus, we propose an intermediate solution: have groups of threads work together on a solution and then take the best solution. Furthermore, the solution of a thread group could be used to direct the search by warning other groups about regions of poor solutions. How many threads each group should be composed of would depend on the number of cores on the GPU and the amount of memory, as each thread group must have its own memory space. Ideally, there should be enough groups to fully utilize the memory and enough threads to fully utilize all cores, with said threads evenly divided among the groups. This new approach would require the use of GPU streams (CUDA functions that allow the programmer to control simultaneous kernel launches and/or memory transfers) as each thread group should have its own kernels and groups should communicate between each other every few iterations.

Finally, regarding our GPU MC-SAT algorithm for sampling based on Simulated Annealing (SA) over MaxWalkSAT (presented in Section 4.2.2), the results in Section 5.3.2 show that SA might not be the best technique for GPUs, as it got "stuck" in middle solutions. Fortunately, many related methods exist in literature

like Tabu search [37], which could be used to mark certain good and bad solutions to avoid visiting them again, or genetic algorithms [96] where a neighbourhood or population could be used to decide which atom should be flipped. One of these techniques is sure to be better for GPUs, but it must be carefully studied and adapted, considering the problems our current algorithm shows and that the solution marks or the neighbourhoods will increase the GPU memory requirements.

6.2.4 GPU parallel ILP

Contributions

Our ILP system (Section 4.3) performs clause learning in FOL by proposing and evaluating possible clauses. Since the evaluation process (called coverage and described in detail in Section 2.5.2) is a time-consuming task that can be seen as a Datalog program, we parallelised it using GPU-Datalog. Our results of testing said system show a performance improvement for three of the four relational learning applications, varying between 5 and almost 8.5 times.

Future Work

The general performance of our ILP system would improve with the addition of coverage lists or other techniques [28] that keep track of clause coverage. These techniques reduce the processing time by avoiding the recomputation of certain clause results when proposing a new, similar clause. For example, if the result of clause `h(X) :- p1(X,Y), p2(Y,Z).` has already been computed and a new clause `h(X) :- p1(X,Y), p2(Y,Z), p3(Z,W).` is proposed, then it is not necessary to compute again the join `p1(X,Y), p2(Y,Z)`, its result is simply used and only one join and a projection (which are represented as `h(X) :- know_result(X,Y,Z), p3(Z,W)`), are necessary to obtain the result of the new clause. How to adapt the data structures used by these techniques into GPU-Datalog and how to deal with the limited amount of GPU memory (a GPU-CPU-GPU

swamping algorithm might be necessary), are the two main aspects to consider for their successful implementation on GPUs.

From the work of Fonseca *et al.* [29] (resumed in Section 4.3.1), some of their described strategies could be adapted to GPUs. In particular, we believe that the *data strategies* could work well on the GPU (the search strategy might require too much control or communication amongst threads), as groups of threads could work together on subsets of examples, in order to generate the best clause for them. Said data strategies could be better than our system's current strategy when the background knowledge is small (and thus coverage computation is fast), and it is necessary to generate many clauses and/or clauses with lots of predicates. Implementing these strategies in our system might require the use of streams to launch a kernel for each subset of examples and then communication between these kernels would be necessary.

Our ILP system could also be used for MLN *clause learning*, since it follows an ILP-like process as shown by Alchemy system (currently, the only system capable of clause learning and whose algorithms are presented in Section 3.5.1). We consider that the most significant change required is to replace the traditional ILP coverage function with an MLN specific evaluation function, like the weighted pseudolog-likelihood (WPLL, Equation 3.11). Moreover, while Alchemy and our system propose new clauses using different approaches, with the former's approach being better for MLNs [18], the parallel processing of our system should make-up for the suboptimal clause proposition. However, it may also be interesting to parallelise Alchemy's approach instead. In this case, GPU-Datalog could be adapted into Alchemy (like it was adapted into Tuffy and RockIt) in order to compute the groundings of the proposed clauses.

6.2.5 GPU-Tuffy

Contributions

Our first platform, called GPU-Tuffy (Section 4.4), performs *inference* by integrating the Tuffy MLN system with GPU-Datalog, in order to compute the *grounding* step (a) in Figure 6.1), and our parallel version of the MaxWalkSAT algorithm, in order to solve the satisfiability problem of the *search* step (b) in Figure 6.1). GPU-Tuffy is also capable of *learning* clause weights by *grounding* (d) in Figure 6.1) with GPU-Datalog and *sampling* (e) in Figure 6.1) with our MC-SAT algorithm for GPUs.

Future Work

The overall performance of GPU-Tuffy might benefit from the removal of its Relational Database Management System (RDBMS), PostgreSQL. In normal Tuffy, PostgreSQL was an important component in charge of the grounding process. In GPU-Tuffy, PostgreSQL is mostly used to store data and as a bridge to communicate between GPU-Datalog and Tuffy (as shown in Figure 4.8 of Section 4.4). Removing the RDBMS would eliminate the overhead of interacting with an additional component, but would also require the implementation of directives for data storage and communication between the remaining components. These directives could be implemented in Java (as part of Tuffy, which is also written in Java) to benefit from its extensive libraries or in C (as part of GPU-Datalog) to benefit from its speed.

Improving the Markov random field (MRF) partitioning (presented in Section 4.4) might also be favourable, as our results in Section 5.3.2 show it tends to produce a very large partition and many smaller ones. Such behaviour might be related to the applications and/or the algorithm, so the first task would be to perform a careful analysis of the algorithm using several applications. Ideally, partitions should be more balanced in order to have a similar execution time, as the overall execution time would be based on the time it takes to process the slowest partition. Another improvement (specially for big data applications) would be to accelerate the partitioning using the

GPU, as it is a rather hard problem with no polynomial-time approximation [88, p.11]. Graph partitioning on GPUs (recall that the MRF is a graph) seems to be a new area of research, but some of the existing algorithms [39] could be adapted for this task.

While we consider that the MaxWalkSAT algorithm used in the search step (b) in Figure 6.1) works well, the processing of smaller partitions on the CPU could be solved by an exact algorithm like branch and bound, instead of a stochastic one like MaxWalkSAT. While such approach might not improve processing time, it may increase the quality of the solutions, as we would warranty that the smaller partitions have the lowest possible solution cost (the lower the solution cost the better). There are many exact solvers in literature like ahmaxsat [3], that can be adapted into Tuffy for this task by passing the satisfiability problem to the solver using the DIMACS CNF format [129], which indicates the number of variables and clauses, along with the clauses themselves with positive variables represented by a positive number and their negation with the same number negated.

Some applications may greatly benefit from *lifted inference* (detailed in Section 3.4.1), which Tuffy and GPU-Tuffy currently lack. Lifted inference is based on belief propagation (BP) and thus, two aspects are critical for its successful parallel implementation: a good BP algorithm for GPUs and adequate data structures for supernodes and superfeatures. Fortunately, BP parallelism has been well studied [42] and one of the existing algorithms for GPUs could be adapted into GPU-Tuffy. However, the data structures are a more complex problem, as the best structure for sequential lifted inference is still a topic of active research [24, p.37], and the best parallel data structure has not been studied yet.

Weight learning could be improved by changing the Diagonal Newton (DN) method for the Scaled Conjugate Gradient method (both described in Section 2.2.1), which is known to be more efficient [70]. Such change would require modifying the code that deals with the mathematical computations of the method and the sampling would still be performed by the GPU with no changes. Furthermore, the

mathematical computations on either of these methods could also be parallelised using GPUs by adapting an existing algorithm like those presented in [115] and [49]. Unsupervised weight learning (i.e., learning with predicates which have some unknown groundings in the training data) could also be added to GPU-Tuffy, following Alchemy's approach based on a different function to optimize (shown in Section 3.5.1). Sampling to compute the gradient and the Hessian matrix of said function would still be performed in the GPU and the DN method would perform the mathematical optimization. The only additional aspect to consider would be the handling of unknown atoms caused by the missing data.

6.2.6 GPU-RockIt

Contributions

Our second platform, GPU-RockIt (Section 4.5) is based on the RockIt MLN system and shows the flexibility of our *grounding* design with GPU-Datalog (a) in Figure 6.1), as it was originally planned for Tuffy, but was easily adapted into RockIt and produces great performance gains. Search is performed as an integer linear programming problem using the efficient Gurobi solver.

Future Work

The results of the grounding on GPU-RockIt 5.3.2, where applications with several small groundings (like LP) adversely affected the performance, suggest that a *hybrid* approach might be a better alternative. In said hybrid approach, the grounding of simpler clauses (e.g., a single join between two predicates with 10 elements each) would be left to the RDBMS (MySQL), while more complex clauses (e.g., a single join between two predicates with 10,000 elements each whose result is expected to be close to the Cartesian product, approximately 100,000,000 elements) would be processed by the GPU. Determining if a clause is simple or complex is no trivial task, but it could be performed using a heuristic based on the size of the input, the number

of predicates, and perhaps a sampling of its operations (e.g., a join between two tables could be sampled by joining 10 random elements from both tables and if the result is close to 100, then the clause might be complex, if it is close to zero, then it might be simple).

About solving the integer linear programming problem, the current solver Gurobi is quite efficient, perhaps the best alternative for multicores. However, it is not open source (although academic licenses are provided) and new, efficient optimization algorithms for GPUs are constantly being proposed, as GPUs work very well in mathematical applications. Thus, we consider an important improvement in both processing time and solution quality may result from changing the Gurobi solver to a GPU solver. Such GPU solver could be adapted from an existing one based on the same methods that Gurobi uses (explained in Section 2.2.1): the simplex method [64] and the barrier method [110] (part of the interior point methods). Adapting these GPU solvers into RockIt requires changing the data structures of RockIt or of the solvers (whichever is most efficient) into a common data structure that allows RockIt to give the optimization problem to the solver and retrieve the result. Also, the proposed solvers work on general linear programming problems while RockIt problems are restricted to integers, thus the solvers should be modified to benefit from integer GPU computations which are faster and exact (unlike those performed on doubles).

Another interesting option for solving the integer linear programming problem would be to change the exact methods currently in use for *metaheuristics* like genetic algorithms [46] (GAs). While metaheuristics have no guaranty of finding a result, in practice they often do with good solution quality and processing time. In the particular case of GAs, a GA could be created from scratch for multicores, GPUs, or even a hybrid solution, using the traditional process of generating an initial population (solution) and iteratively crossing, mutating, and selecting the best individuals until a good population is found. Possible specific optimizations for MLNs could be investigated, exploiting information like the sparse connections between atoms in the MRF to partition the population.

Appendix A

GPUs

This Appendix is an updated version from Chapter 2 of [72] and presents an overview of the GPU architecture, its programming model and interface, and programming guidelines for good performance.

Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput [154]. They were designed for computer graphics and could only be programmed through relatively complex APIs like DirectX and OpenGL. Nowadays, GPUs are general-purpose processors with specially designed APIs like CUDA and OpenCL. Applications may obtain great speed-ups even when compared against finely tuned CPU implementations.

GPUs are now used in a wide array of applications [153], including gaming, data mining, bioinformatics, chemistry, finance, numerical analysis, imaging, weather, etc. Such applications are usually accelerated by at least an order of magnitude, but accelerations of 10x or more are common.

Numerical applications are typical of science and engineering, wherein vast amounts of integer and floating point operations are carried out in order to simulate physical phenomena as close to reality as possible. It was for numerical applications that GPUs were originally targeted, as the game industry has been pushing for games to look the most real possible. Numerical applications are typically developed in the high-level languages Fortran and C. In clusters composed of multicore-GPU nodes, numerical applications use both OpenMP code and MPI (Message Passing Interface) code in order to capitalise from both intra-node and inter-node parallelism respectively.

Symbolic applications are typical of artificial intelligence, which itself includes the following areas: expert systems, automated reasoning, knowledge representation, natural language processing, problem solving, planning, machine learning and data mining. The main characteristic of these applications is that they perform vast amounts of search and pattern matching operations. Work to use GPUs for these applications is just beginning.

The GPUs used in this thesis work were Nvidia GPUs [146], so all future mention of GPUs refer to those of this particular brand. The examples and images used in this Appendix were taken from [147].

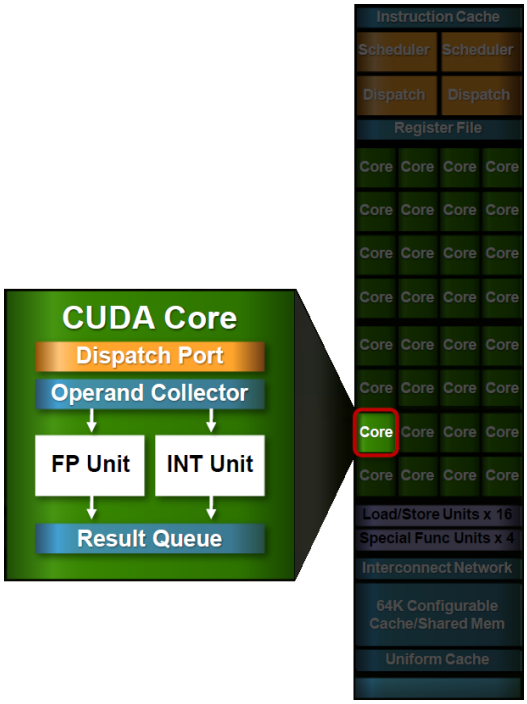


Figure A.1: Elements of a CUDA Core.

A.1 GPU Architecture

GPUs are akin to SIMD machines: they consist of many processing elements that run all a *same program* but on distinct data items. This same program, referred to as the *kernel*, can be quite complex including control statements such as *if* and *while* statements. However, a kernel is *synchronised by hardware*, i.e.: each instruction within the kernel is executed across all the active processing elements running the kernel. Thus, if the kernel involves comparing strings, the processing elements that compare longer strings will take longer, making other processing elements to wait for them.

GPUs usually have hundreds of processing units called CUDA cores, as shown in Figure A.1, which execute one thread each. A CUDA core has the following elements:

- Floating point unit compliant with IEEE floating-point standard.
- Integer unit.
- Logic unit.
- Move, compare unit.
- Branch unit.

CUDA cores are arranged in special hardware units called Streaming Multiprocessors (SM), each with several CUDA cores. An SM schedules threads to be executed

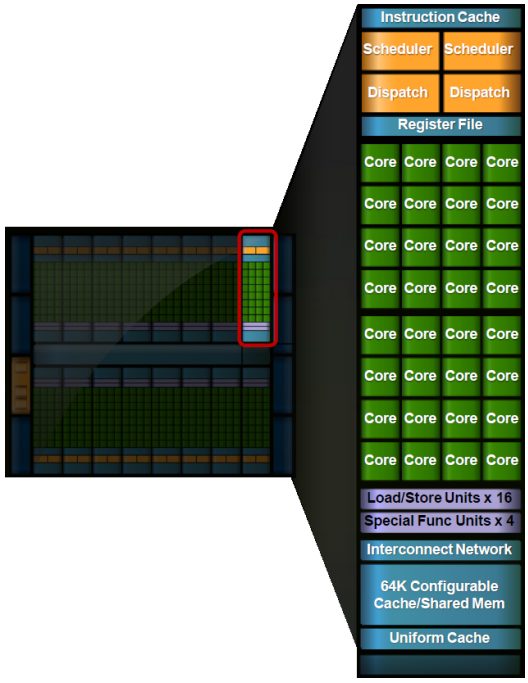


Figure A.2: Elements of a Streaming Multiprocessor.

in *warps* of size equal to the number of CUDA cores it has (*warp size*). As shown in Figure A.2, each SM has the following components:

- Warp schedulers to handle thread concurrency.
- Instruction dispatchers that, ideally, issue the same instruction to all threads.
- Registers to store thread level variables and arrays.
- Load/Store units to handle memory reads/writes.
- Special-function units designed for high speed execution of transcendental instructions such as sin, cosine, square root, etc.
- L1 cache/shared memory whose size can be changed by the programmer to adapt to his needs.

The compute capability of a GPU determines various characteristics like maximum number of threads, amount of shared memory, etc. It is defined by a major revision number and a minor revision number. The architectures corresponding to the major revision numbers are:

- **Pascal.** The latest architecture; major revision number is 6.
- **Maxwell.** Still in use for many applications as Pascal is fairly recent; major revision number is 5.

- **Kepler.** The most widespread architecture; major revision number is 3.
- **Fermi.** An aging architecture that may not be supported in future version of CUDA; major revision number is 2.
- **Tesla.** Though no longer supported, it was the first architecture to support CUDA; major revision number is 1.

The minor revision number is a small improvement over the architecture, like increasing the number of processing cores or the number of registers.

A.1.1 CUDA

CUDA (Compute Unified Device Architecture) is a software platform and programming model created by Nvidia [147]. With CUDA, the GPU becomes a highly parallel general-purpose machine.

CUDA is an extension to the programming languages C, C++ and Fortran (other languages are supported but are not part of the standard). It also includes highly tuned libraries for a wide variety of applications like Thrust [151], a library of parallel algorithms and data structures based on the Standard Template Library (STL) library [86].

The current version of the CUDA SDK (8.0) is available for Microsoft Windows, Linux and Mac OS through the NVIDIA website [146]. CUDA works with all modern Nvidia GPUs. Programs developed for a particular GPU should also work on all GPUs of the same or better architectures without modifying the source code.

A.2 Programming model

This section describes the CUDA programming model for C, known as CUDA C. The models for other languages are similar. We will refer to CUDA C as CUDA from now on.

Figure A.3 shows that CUDA threads are executed on a different *device* (GPU) that serves as a coprocessor to the *host* (CPU). A host thread executes all serial code (in the host), including memory management and work scheduling functions, while the device executes parallel work using the most appropriate configuration of threads. Both host and device maintain their own memory, called host memory and device memory. GPUs usually have their own high speed on-chip memory, however, low-end GPUs use a reserved portion of the host's RAM.

A.2.1 Kernels

CUDA extends C with its own functions and reserved words. It also allows the definition of user functions, called *kernels*, that are executed in parallel by CUDA threads.

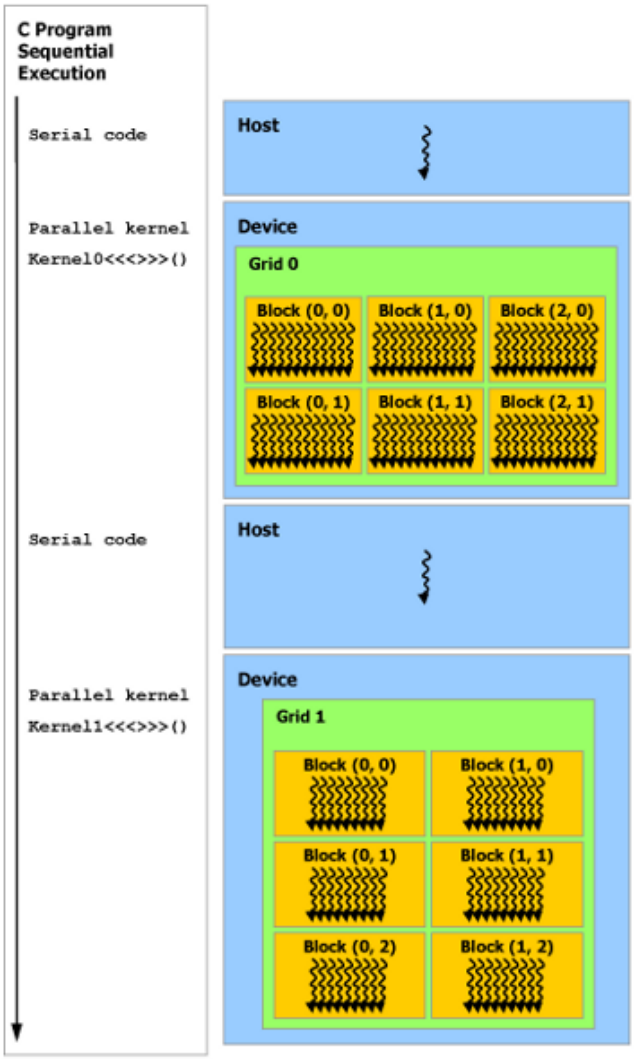


Figure A.3: The heterogeneous programming model.

Kernels are defined using the `__global__` identifier before the return type of a function. For example, consider the following sample code adds two vectors, A and B, and stores the result into vector C:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

The host thread invokes a kernel specifying the number of CUDA threads that will execute the kernel using <<< ... >>>. For example, to call the kernel *VecAdd* we do the following:

```
int main()
{
  ...
  // Kernel invocation with N threads
  VecAdd<<<1, N>>>(A, B, C);
  ...
}
```

In this example, we invoke N threads with global identifiers from 0 to N-1. The number 1 inside the <<< ... >>> refers to the number of blocks that will be invoked to process the kernel. The following subsection explains more about thread identifiers and blocks.

A.2.2 Thread Hierarchy

Threads are organized into *blocks*, and blocks into a *grid* as shown in Figure A.4.

To assign work to each thread and control their execution, threads are identified with indexes that determine their position in a block. A thread may have the following indexes depending on the "shape" of the block:

- **Vector.** The block has only one dimension and the thread is identified by one index (x).
- **Matrix.** The block has two dimensions and the thread is identified by two indexes (x, y).
- **Volume.** The block has three dimensions and the thread is identified by three indexes (x, y, z).

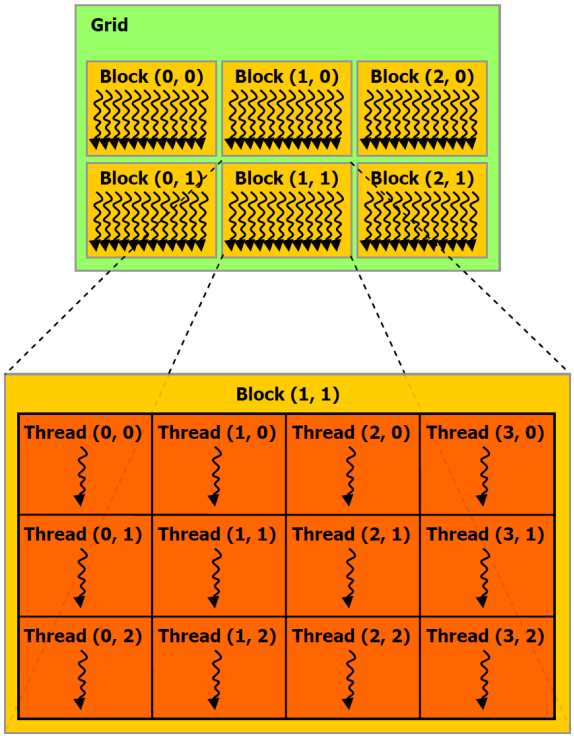


Figure A.4: Thread hierarchy.

Blocks also have their own indexes to identify them inside a grid. Grids, like blocks, may have up to three dimensions, and thus, block indexes may have up to three values (x, y, z). To identify each of the threads and blocks running a kernel, CUDA provides the programmer with the following reserved words as identifiers, each with three components (x, y and z):

- **threadIdx** is the index of the thread in his block.
- **blockIdx** is the index of the block in the grid.
- **blockDim** is the size, in number of threads, of the block.
- **gridDim** is the size, in number of blocks, of the grid.

Using these identifiers, new identifiers can be derived with simple arithmetic operations. For example, the global identifier of a thread in a three-dimensional block would be:

```
unsigned int ID = threadIdx.x + threadIdx.y * blockDim.x +
                  threadIdx.z * blockDim.x * blockDim.z;
```

The number of threads per block and the number of blocks per grid are specified using *int* or *dim3* types. *dim3* is a structure of three unsigned integers with components x, y and z. An important characteristic of this structure is that any

unspecified component is initialized to one. Using the <<< ... >>> syntax, the number of threads is specified as follows:

```
dim3 numBlocks(A, B, C);
dim3 threadsPerBlock(X, Y, Z);
kernel<<<numBlocks, threadsPerBlock>>>();
```

The total number of threads to be executed is equal to the number of threads per block times the number of blocks. Because of that, there are many possible combinations that yield the same total number of threads, for example, 32 blocks of 10 threads each would yield 320 threads in total and, apparently, it would be the same as having 10 blocks of 32 threads each.

However, recall that each Streaming Multiprocessor has a certain number of CUDA cores, and schedules threads to be executed in *warps* of size equal to this number of cores (*warp size*). Hence, if a block has less threads than the warp size, some cores will be idle. On the other hand, if the block has more threads than the warp size, some threads will have to wait their turn. This means that, for each block, we should try to avoid using less threads than the warp size. However, it does not mean that we should always use a number of threads equal to the warp size because switching threads in a block is faster than switching entire blocks. There is also a limit to the number of threads that can be specified for a block (1024 for current GPUs, less for others), since all threads of a block are scheduled to the same SM and must share registers and shared memory.

As shown in Figure A.5, at hardware level, the GPU automatically assigns thread blocks to SMs depending on the number of available SMs. This allows GPUs to execute kernels according to their capabilities. This scheduling policy should be considered when determining the number of blocks. If this number is less than the number of available SMs, the computational power will not be fully exploited.

To coordinate threads in the same block, the function *synchthreads* can be used as a barrier. This function makes all the threads in a block to wait until all of them have reached the function. Example:

```
if(threadIdx.x == 0)
  a[0] = 5;
__syncthreads();
```

In this example, all the threads in the block will wait until thread 0 finishes writing to memory and only then will they continue.

A.2.3 Memory Hierarchy

CUDA threads have access to different memory types as shown in Figure A.6. Each thread has a private *local memory* (registers) for stack and variables. Each thread block has *shared memory* visible to all threads in the block. All threads have access to the same *global memory*.

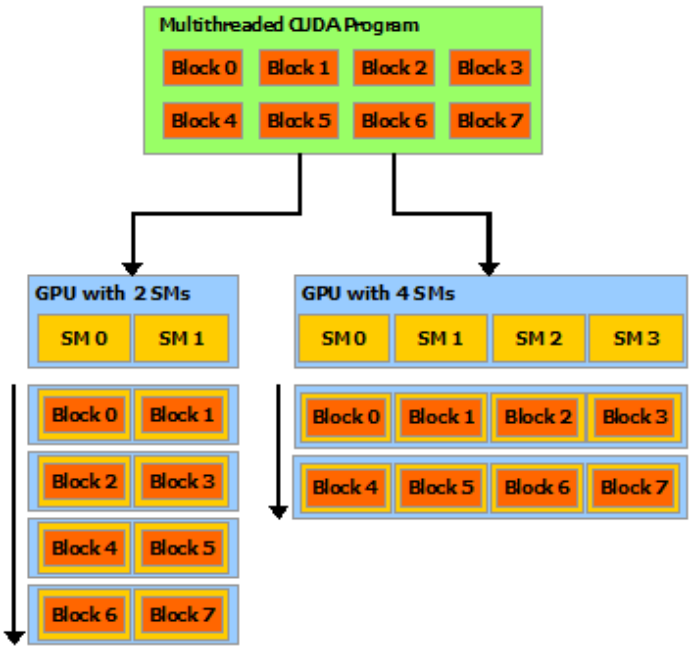


Figure A.5: Automatic scalability based on the characteristics of the GPU.

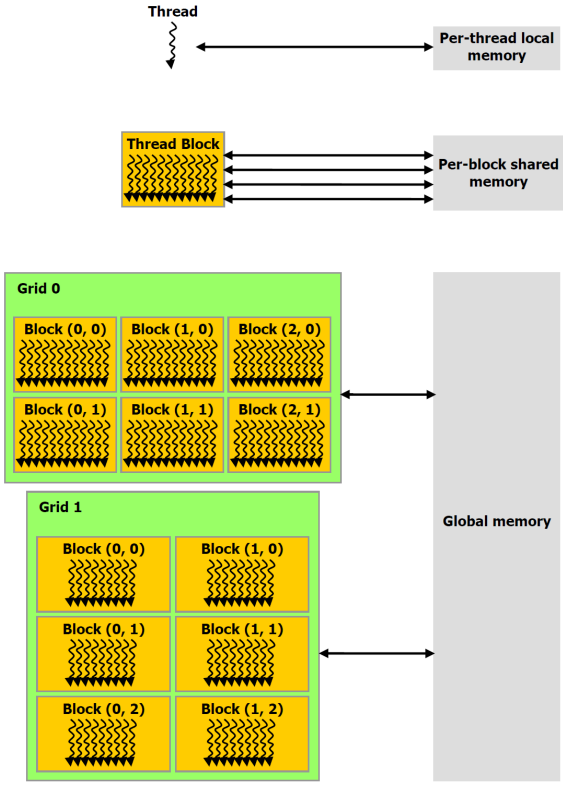


Figure A.6: Memory hierarchy.

Global memory

Global memory is the medium of communication between host and device. Usually, the host transfers to this memory the elements to be processed in the device and obtains the result from this same memory.

Global memory is allocated with *cudaMalloc* which requires the address of a pointer and the number of bytes to allocate. Example:

```
int *ptr;
/*Allocate memory for ten integers*/
cudaMalloc(&ptr, 10 * sizeof(int));
```

Once memory has been allocated, data can be transferred with *cudaMemcpy* which requires a destination address, a source address, the number of bytes to transfer and the "direction" of the transfer. For example:

```
int *ptr, i = 5;
cudaMalloc(&ptr, sizeof(int));
/*Copy one integer from host to device*/
cudaMemcpy(ptr, &i, sizeof(int), cudaMemcpyHostToDevice);
```

There are four possible directions which indicate from where to where the data transfer is to be made:

- **cudaMemcpyHostToDevice.** From the CPU to the GPU.
- **cudaMemcpyDeviceToHost.**: From the GPU to the CPU.
- **cudaMemcpyHostToHost.** Between two CPU addresses.
- **cudaMemcpyDeviceToDevice.** Between two GPU addresses. No CPU interaction is required.

Memory can be freed with *cudaFree* which requires the address to be freed. Example:

```
int *ptr;
cudaMalloc(&ptr, sizeof(int));
cudaFree(ptr);
```

Shared Memory

Shared memory is declared in kernels by using the `__shared__` reserved word before the type of the desired memory. It is usually initialized by the first threads of each block. Example:

```
__shared__ int a;
if(threadIdx.x == 0)
  a = 5;
```

A variable sized array of shared memory can be allocated by creating a shared pointer in the kernel and using the third argument of the kernel call to specify the size in bytes. Example:

```
//Host code to create an array of ten integers in shared memory
kernel<<<numBlocks, threadsPerBlock, 10 * sizeof(int)>>>();
/*Device code to have the first ten threads of each block initialize
the array with their thread ID*/
__shared__ int array[];
if(threadIdx.x < 10)
  array[threadIdx.x] = threadIdx.x;
```

Shared memory is much faster than global memory. If an element in global memory has to be read or written more than once, it is a good idea to transfer it to registers or shared memory if possible.

A.3 Programming Interface

CUDA provides functions that execute on the host to perform tasks like timing, error checking, device handling, etc. To compile CUDA programs, a compiler tool called *nvcc* is also provided.

A.3.1 Compilation with nvcc

Nvcc is a compiler that simplifies the process of compiling CUDA code. It uses command line options similar to those of GCC [145] and automatically calls the necessary programs for each compilation stage.

CUDA programs usually include kernels and C code for input/output and memory management operations. The compilation stages for these programs are as follows:

1. Kernels (device code) are separated from the C host code.

2. Device code is compiled by nvcc into the assembly language for GPUs called PTX.

3. Device code can then be left in assembly form or compiled into binary form by the graphics driver.

4. Host code is modified by changing kernel calls into the appropriate CUDA functions that prepare and launch kernels.

5. Host code is then compiled into object code by the designated C compiler (usually gcc).

6. Both codes are linked to produce the executable program

A.3.2 Concurrent Execution between Host and Device

Some CUDA function calls are asynchronous. It means that the host thread calls one such function and then continues its work, instead of waiting for the function to return. The following functions are asynchronous:

- Kernel launches.
- Memory copies between two addresses in device memory.
- Memory copies of 64 KB or less from host to device.
- All functions whose name starts with *async*.
- Memory set functions (this function is equivalent to Unix function *memset* which sets the bytes of a block of memory to an specific value).

These functions are asynchronous to the host because they are performed by the device. However, their execution in the device is serialized. For example:

```
int *ptr, var;
//Allocate memory for ptr
cudaMalloc(&ptr, sizeof(int));
//Call of a kernel that will store its result in ptr
kernel<<<numBlocks, threadsPerBlock>>>(ptr);
//Copy the result to var in host memory from device memory
cudaMemcpy(&var, ptr, sizeof(int), cudaMemcpyDeviceToHost);
//Print the result
printf("%d", var);
```

Here the call to the kernel will immediately return control to the host and the host will execute a synchronous *cudaMemcpy* — the host will block waiting for the result of the copy. The device will execute the kernel and, once finished, will execute the memory copy the host is waiting for.

A.3.3 Events

Events allow programmers to monitor the device and perform accurate timing. Events can be asynchronously started and ended at any point in the host code. An event is completed when all host and device tasks between its starting and ending positions are completed. At this point, it is possible to check the elapsed time. The following code sample shows how to measure the elapsed time of a code section using events:

```
//Event creation
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
//Start timer
cudaEventRecord(start, 0);
...
//Code to measure
...
//Stop timer
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
//Show elapsed time
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("%f", elapsedTime);
//Event destruction
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

A.3.4 Device handling

A host system can have more than one GPU. Host threads can set the *current device* at any time by using *cudaSetDevice*. Any device memory management functions, kernel launches and events are executed only for the current device. By default, the current device is always device 0. The following code sample shows how to enumerate these devices, query their compute capability, and change the current device:

```
//Get the number of devices
int deviceCount;
cudaGetDeviceCount(&deviceCount);
//For each device
int device;
for(device = 0; device < deviceCount; device++)
{
  //Show the device properties
  cudaDeviceProp deviceProp;
  cudaGetDeviceProperties(&deviceProp, device);
  printf("Device %d has compute capability %d.%d.\n",
  device, deviceProp.major, deviceProp.minor);
}
//Set device 0 as current
cudaSetDevice(0);
```

A.3.5 Error Checking

All runtime functions return an error code. However, for asynchronous functions, this error cannot be retrieved by the return value of the function (as control is returned to the host before the device finishes executing the function). When an error happens in an asynchronous function, the next runtime function, asynchronous or not, will return this error.

When it is necessary to immediately check for errors in an asynchronous function, the host must be blocked until the device finishes executing the function. The function *cudaDeviceSynchronize* blocks the host until the device finishes executing the last function invoked; its return value has any error associated with the last CUDA function execution.

Since kernels do not return anything, the runtime environment has an error variable initialized to *cudaSuccess* which is overwritten with an error code when an error occurs. *CudaPeekAtLastError* and *cudaGetLastError* return this variable. Then, to get kernel errors, the kernel has to be launched, the host has to be blocked with *cudaDeviceSynchronize*, and *cudaPeekAtLastError* or *cudaGetLastError* have to be called to obtain any kernel errors.

A.3.6 Compatibility

While newer GPUs support all the instructions of older GPUs, instructions introduced for newer architectures cannot possibly be supported by older architectures. For example, double-precision is only available on devices of compute capability 1.3 and above. To compile CUDA code for a certain compute capability, the *-arch* compiler flag can be used. This option can be specified regardless of the current hardware in the machine doing the compiling (it can even be a machine with no GPUs). For example, code with double-precision instructions must be compiled with *-arch=sm_13* (or higher), otherwise any double-precision instructions will automatically be transformed into single-precision instructions by the compiler.

There are two versions of the nvcc compiler, for 64-bit and 32-bit host architectures. Any version can be installed, regardless of the host architecture. However, device code compiled for 64-bit can only work with 64-bit host code, and 32-bit device code can only works with 32-bit host code. By default, nvcc compiles code for 64-bit if the 64-bit version is installed, but it can also compile in 32-bit mode with the *-m32* compiler flag if the 32-bit CUDA libraries are installed. The 32-bit version can compile to 64-bit mode with the *-m64* flag if the necessary libraries are installed.

A.4 Performance Guidelines

To maximize GPU performance, the CUDA Best Practices Guide [147] suggests the following strategies:

- Maximize parallel execution to achieve maximum device utilization.
- Optimize memory usage to achieve maximum memory throughput.
- Optimize instruction usage to achieve maximum instruction throughput.

It is important to correctly choose which strategies to pursue depending on how much they improve the code. For example, optimizing instruction usage for a kernel with memory access problems will not show great performance increase.

A.4.1 Maximize Utilization

To maximize utilization, programmers must be familiar with the massive parallelism the GPUs provide and try to make full use of it.

Application Level

Thanks to the asynchronous nature of kernels calls, programmers should try not to leave the host idle while it waits for the result of a kernel. Simple or non-parallelizable tasks should be executed by the host, while highly parallel tasks should be sent to the device.

Device Level

Kernels should be executed with at least as many threads per block as there are cores in each SM. The number of blocks should at least be equal to the number of SMs in the GPU. If a kernel requires less blocks than the number of available SMs, two or more small kernels should be run at the same time (using streams), thus fully utilizing the GPUs capabilities.

A.4.2 Maximize Memory Throughput

One of the most important optimizations to any CUDA program is to minimize data transfers between the host and the device. These transfers are done through the PCIe bridge and have the lowest bandwidth when compared to other types of transfers. Excessive use of these transfers may even cause applications to be slower than their CPU-only counterpart versions.

Minimizing access (reads and writes) to global memory by kernels with the help of shared memory and registers also improves performance — although it tends to complicate programming. To use shared memory for this purpose, each thread in a block has to do the following:

- Move its corresponding data from global memory to shared memory.
- If this data is to be accessed by other threads, then we must synchronize with all the other threads of the block using the function *synchthreads*.

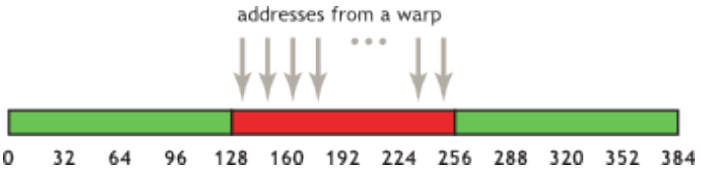


Figure A.7: Coalesced memory access.

- Process the data in shared memory.
- Synchronize again if data was used by other threads to allow them to finish processing.
- Write the results back to global memory.

Data Transfer between Host and Device

To minimize data transfers between host and device, code that is executed in the host could be executed in the device. Even if such code is not very parallelizable, performance may increase due to the reduced number of memory transfers. Joining small data transfers into a single, large transfer also increases performance.

Device Memory Accesses

When all threads in a warp execute a load instruction, the best global memory access occurs when all threads in a warp accesses consecutive global memory locations. When this happens, the hardware *coalesces* (combines) all memory accesses into a single access to consecutive locations. For example:

If thread 0 accesses location n, thread 1 accesses location $n + 1$, ..., thread 31 accesses location $n + 31$, then all these accesses are coalesced. Figure A.7 shows an example of coalesced access.

Global Memory

When global memory is accessed by an instruction in a warp, one or more memory transactions are issued. This depends on which memory locations are to be accessed by each thread. More transactions means less performance. The worst case would be a number of transactions equal to the warp size.

For devices of compute capability 1.0 and 1.1, access has to be completely coalesced, else the number of transactions will be equal to the warp size (the worst case scenario). For devices of higher compute capability, memory transactions are cached (using L1 or L2 cache), so a single transaction might be issued even if accessing non-contiguous memory locations.

Size and Alignment Requirement

Global memory instructions read or write words of 1, 2, 4, 8, or 16 bytes. Coalesced access to global memory also requires the data to have one of these sizes and to be naturally aligned (i.e., its address is a multiple of its size). The alignment is automatically fulfilled for most built-in types.

Local Memory

Local memory is a section of global memory automatically reserved by the compiler. It is used to store the following variables found inside a kernel:

- Large structures or arrays that would consume too much register space.
- Any variable if the kernel uses more registers than available (known as *register spilling*).

Since local memory resides in global memory, it has the same disadvantages (i.e. slow reads and writes, slow transfers, etc.). Use of this memory should be avoided by splitting structures or arrays into smaller ones and by using less registers or launching fewer threads per block.

A.4.3 Maximize Instruction Throughput

To maximize instruction throughput the following strategies are suggested:

- Use single-precision instead of double-precision if this change does not affect the required result.
- Avoid any control flow instructions.
- Remove synchronization points wherever possible.

Control Flow Instructions

Control flow instructions (if, switch, do, for, while) tend to make threads of the same warp to diverge (i.e., to follow different execution paths). The different executions paths are serialized and instructions for each of them have to be issued, thus increasing the total number of instructions. When all execution paths are completed, threads converge back to the same execution path.

Appendix B

The Transaction Processing Performance Council Benchmark H

The Transaction Processing Performance Council Benchmark H (TCP-H) [152] is a famous decision support benchmark based on SQL that was used to compare the performance of the logical component of our MLN platforms, namely GPU-Datalog (described in Section 4.1), against the state of the art system Red Fox [150]. This appendix is an adapted version of the 2.17.2 specification of the benchmark and presents an explanation of TCP-H, the structure of its tables, and the queries used in our comparison.

TPC-H is comprised of a set of business queries designed to exercise system functionalities in a manner representative of complex business analysis applications. These queries have been given a realistic context, portraying the activity of a wholesale supplier to help the reader relate intuitively to the components of the benchmark. TPC-H does not represent the activity of any particular business segment, but rather any industry which must manage sell, or distribute a product worldwide (e.g., car rental, food distribution, parts, suppliers, etc.). The queries that have been selected for this benchmark exhibit the following characteristics:

- They have a high degree of complexity.
- They use a variety of access methods.
- They are of an ad hoc nature.
- They examine a large percentage of the available data.
- They all differ from each other.
- They contain query parameters that change across query executions.

Table B.1: PART Table Layout.

Column Name	Datatype Requirements	Comment
P_PARTKEY	identifier	Primary Key
P_NAME	text, size 55	
P_MFGR	text, size 25	
P_BRAND	text, size 10	
P_TYPE	text, size 25	
P_SIZE	integer	
P_CONTAINER	text, size 10	
P_RETAILPRICE	text, size 55	
P_RETAILPRICE	text, size 55	

Table B.2: SUPPLIER Table Layout.

Column Name	Datatype Requirements	Comment
S_SUPPKEY	identifier	Primary Key
S_NAME	text, size 25	
S_ADDRESS	text, size 40	
S_NATIONKEY	identifier	Foreign Key to N_NATIONKEY
S_PHONE	text, size 15	
S_ACCTBAL	decimal	
S_COMMENT	text, size 101	

B.1 Database structure

The components of the TPC-H database are defined to consist of eight separate and individual tables. These tables can be generated and filled with random data using the TPC-H tools [152]. The relationships between columns of these tables are illustrated in Figure B.1 where the parentheses following each table name contain the prefix of the column names for that table, the arrows point in the direction of the one to many relationships between tables, and the number/formula below each table name represents the cardinality (number of rows) of the table.

Some tables are factored by a Scale Factor (SF), to obtain the chosen database size (in our comparison against Red Fox, SF is equal to 1). The cardinality for the LINEITEM table is approximate since the number of LINEITEMs in an ORDER is chosen at random with an average of four. Also, orders are not present for all customers. In fact, one-third of the customers do not have any order in the database. The orders are assigned at random to two-thirds of the customers. The purpose of this is to exercise the capabilities of the DBMS to handle "dead data" when joining two or more tables.

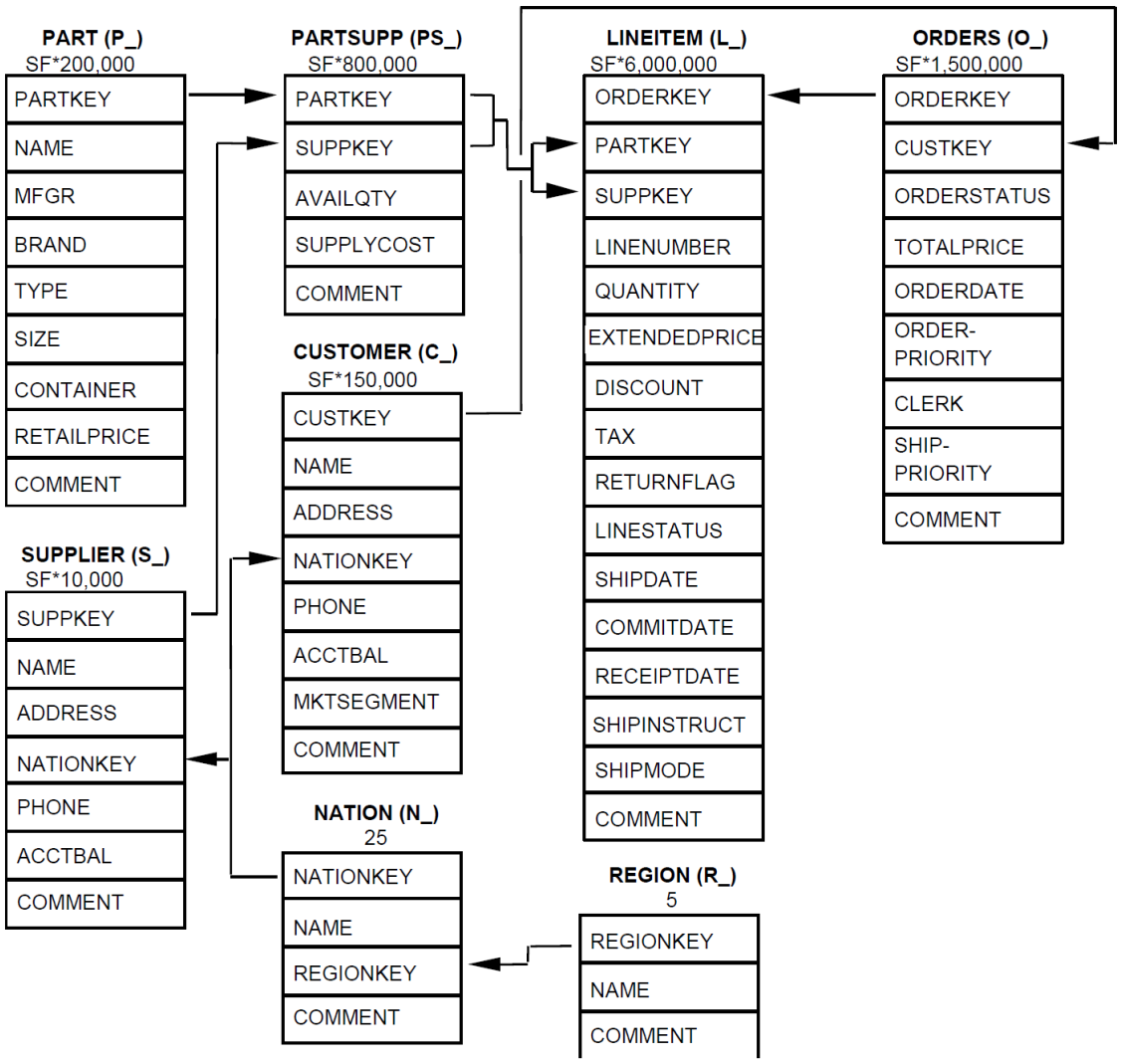


Figure B.1: The TPC-H database schema.

Table B.3: PARTSUPP Table Layout.

Column Name	Datatype Requirements	Comment
PS_PARTKEY	identifier	Primary Key Foreign Key to P_PARTKEY
PS_SUPPKEY	identifier	Primary Key Foreign Key to S_SUPPKEY
PS_AVAILQTY	integer	
PS_SUPPLYCOST	decimal	
PS_COMMENT	text, size 199	

Table B.4: CUSTOMER Table Layout.

Column Name	Datatype Requirements	Comment
C_CUSTKEY	identifier	Primary Key
C_NAME	text, size 25	
C_ADDRESS	text, size 40	
C_NATIONKEY	identifier	Foreign Key to N_NATIONKEY
C_PHONE	text, size 15	
C_ACCTBAL	decimal	
C_MKTSEGMENT	text, size 10	
C_COMMENT	text, size 117	

Table B.5: ORDERS Table Layout.

Column Name	Datatype Requirements	Comment
O_ORDERKEY	identifier	Primary Key
O_CUSTKEY	identifier	Foreign Key to C_CUSTKEY
O_ORDERSTATUS	text, size 1	
O_TOTALPRICE	decimal	
O_ORDERDATE	date	
O_ORDERPRIORITY	text, size 15	
O_CLERK	text, size 15	
O_SHIPPRIORITY	integer	
O_COMMENT	text, size 79	

Table B.6: LINEITEM Table Layout.

Column Name	Datatype Requirements	Comment
L_ORDERKEY	identifier	Primary Key Foreign Key to O_ORDERKEY
L_PARTKEY	identifier	Foreign key to P_PARTKEY
L_SUPPKEY	identifier	Foreign key to S_SUPPKEY
L_LINENUMBER	integer	Primary Key
L_QUANTITY	decimal	
L_EXTENDEDPRICE	decimal	
L_DISCOUNT	decimal	
L_TAX	decimal	
L_RETURNFLAG	text, size 1	
L_LINESTATUS	text, size 1	
L_SHIPDATE	date	
L_COMMITDATE	date	
L_RECEIPTDATE	date	
L_SHIPMODE	text, size 10	
L_COMMENT	text size 44	

Table B.7: NATION Table Layout.

Column Name	Datatype Requirements	Comment
N_NATIONKEY	identifier	Primary Key
N_NAME	text, size 25	
N_REGIONKEY	identifier	Foreign Key to R_REGIONKEY
N_COMMENT	text, size 152	

Table B.8: REGION Table Layout.

Column Name	Datatype Requirements	Comment
R_REGIONKEY	identifier	Primary Key
R_NAME	text, size 25	
R_COMMENT	text, size 152	

B.2 Query Definitions

Pricing Summary Report Query (Q1)

The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60 - 120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included.

```
select
  l_returnflag,
  l_linestatus,
  sum(l_quantity) as sum_qty,
  sum(l_extendedprice) as sum_base_price,
  sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
  avg(l_extendedprice) as avg_price,
  avg(l_discount) as avg_disc,
  count(*) as count_order
from
  lineitem
where
  l_shipdate <= date '1998-12-01' - interval '[DELTA]' day (3)
group by
  l_returnflag,
  l_linestatus
```

```
order by
  l_returnflag,
  l_linestatus;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: DELTA = 90. It must produce the following sample output data:

```
L_RETURNFLAG L_LINESTATUS     SUM_QTY SUM_BASE_PRICE SUM_DISC_PRICE
           A             F 37734107.00 56586554400.73 53758257134.87
    SUM_CHARGE AVG_QTY AVG_PRICE AVG_DISC COUNT_ORDER
55909065222.83   25.52  38273.13      .05     1478493
```

Shipping Priority Query (Q3)

The Shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of l_extendedprice * (1-l_discount), of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more than 10 unshipped orders exist, only the 10 orders with the largest revenue are listed.

```
Return the first 10 selected rows
select
  l_orderkey,
  sum(l_extendedprice*(1-l_discount)) as revenue,
  o_orderdate,
  o_shippriority
from
  customer,
  orders,
  lineitem
where
  c_mktsegment = '[SEGMENT]'
  and c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate < date '[DATE]'
  and l_shipdate > date '[DATE]'
group by
  l_orderkey,
  o_orderdate,
  o_shippriority
order by
  revenue desc,
  o_orderdate;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: SEGMENT = BUILDING; DATE = 1995-03-15. It must produce the following sample output data:

```
L_ORDERKEY   REVENUE O_ORDERDATE O_SHIPPRIORITY
   2456423 406181.01  1995-03-05              0
```

Order Priority Checking Query (Q4)

The Order Priority Checking Query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

```
select
  o_orderpriority,
  count(*) as order_count
from
  orders
where
  o_orderdate >= date '[DATE]'
  and o_orderdate < date '[DATE]' + interval '3' month
  and exists (
    select
      *
    from
      lineitem
    where
      l_orderkey = o_orderkey
      and l_commitdate < l_receiptdate
  )
group by
  o_orderpriority
  order by
  o_orderpriority;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: DATE = 1993-07-01. It must produce the following sample output data:

```
O_ORDERPRIORITY ORDER_COUNT
       1-URGENT       10594
```

Local Supplier Volume Query (Q5)

The Local Supplier Volume Query lists for each nation in a region the revenue volume that resulted from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. The query is run in order to determine whether to institute local distribution centers in a given region. The query considers only parts ordered in a given year. The query displays the nations and revenue volume in descending order by revenue. Revenue volume for all qualifying lineitems in a particular nation is defined as sum(l_extendedprice * (1 - l_discount)).

```
select
  n_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue
from
  customer,
  orders,
  lineitem,
  supplier,
  nation,
  region
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and l_suppkey = s_suppkey
  and c_nationkey = s_nationkey
  and s_nationkey = n_nationkey
  and n_regionkey = r_regionkey
  and r_name = '[REGION]'
  and o_orderdate >= date '[DATE]'
  and o_orderdate < date '[DATE]' + interval '1' year
group by
  n_name
order by
  revenue desc;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: REGION = ASIA; DATE = 1994-01-01. It must produce the following sample output data:

```
   N_NAME     REVENUE
INDONESIA 55502041.17
```

Returned Item Reporting Query (Q10)

The Returned Item Reporting Query finds the top 20 customers, in terms of their effect on lost revenue for a given quarter, who have returned parts. The query

considers only parts that were ordered in the specified quarter. The query lists the customer's name, address, nation, phone number, account balance, comment information and revenue lost. The customers are listed in descending order of lost revenue. Revenue lost is defined as sum(l_extendedprice*(1-l_discount)) for all qualifying lineitems.

```
Return the first 20 selected rows
select
  c_custkey,
  c_name,
  sum(l_extendedprice * (1 - l_discount)) as revenue,
  c_acctbal,
  n_name,
  c_address,
  c_phone,
  c_comment
from
  customer,
  orders,
  lineitem,
  nation
where
  c_custkey = o_custkey
  and l_orderkey = o_orderkey
  and o_orderdate >= date '[DATE]'
  and o_orderdate < date '[DATE]' + interval '3' month
  and l_returnflag = 'R'
  and c_nationkey = n_nationkey
group by
  c_custkey,
  c_name,
  c_acctbal,
  c_phone,
  n_name,
  c_address,
  c_comment
order by
  revenue desc;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: DATE = 1993-10-01. It must produce the following sample output data:

```
C_CUSTKEY             C_NAME   REVENUE C_ACCTBAL N_NAME
```

```
    57040 Customer#000057040 734235.24    632.87  JAPAN
 C_ADDRESS         C_PHONE
Eioyzjf4pp 22-895-641-3466
                                                        C_COMMENT
sits. slyly regular requests sleep alongside of the regular inst
```

Large Volume Customer Query (Q18)

The Large Volume Customer Query finds a list of the top 100 customers who have ever placed large quantity orders. The query lists the customer name, customer key, the order key, date and total price and the quantity for the order.

```
Return the first 100 selected rows
select
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice,
  sum(l_quantity)
from
  customer,
  orders,
  lineitem
where
  o_orderkey in (
    select
      l_orderkey
    from
      lineitem
    group by
      l_orderkey having
        sum(l_quantity) > [QUANTITY]
  )
  and c_custkey = o_custkey
  and o_orderkey = l_orderkey
group by
  c_name,
  c_custkey,
  o_orderkey,
  o_orderdate,
  o_totalprice
order by
  o_totalprice desc,
```

```
  o_orderdate;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: QUANTITY = 300. It must produce the following sample output data:

```
            C_NAME C_CUSTKEY O_ORDERKEY O_ORDERDATE
Customer#000128120    128120    4722021  1994-04-07
O_TOTALPRICE Sum(L_QUANTITY)
   544089.09          323.00
```

Discounted Revenue Query (Q19)

The Discounted Revenue query finds the gross discounted revenue for all orders for three different types of parts that were shipped by air and delivered in person. Parts are selected based on the combination of specific brands, a list of containers, and a range of sizes.

```
select
  sum(l_extendedprice * (1 - l_discount) ) as revenue
from
  lineitem,
  part
where
(
  p_partkey = l_partkey
  and p_brand = '[BRAND1]'
  and p_container in ( 'SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
  and l_quantity >= [QUANTITY1] and l_quantity <= [QUANTITY1] + 10
  and p_size between 1 and 5
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON'
)
or
(
  p_partkey = l_partkey
  and p_brand = '[BRAND2]'
  and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
  and l_quantity >= [QUANTITY2] and l_quantity <= [QUANTITY2] + 10
  and p_size between 1 and 10
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON'
)
or
```

```
(
  p_partkey = l_partkey
  and p_brand = '[BRAND3]'
  and p_container in ( 'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
  and l_quantity >= [QUANTITY3] and l_quantity <= [QUANTITY3] + 10
  and p_size between 1 and 15
  and l_shipmode in ('AIR', 'AIR REG')
  and l_shipinstruct = 'DELIVER IN PERSON'
);
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: QUANTITY1 = 1; QUANTITY2 = 10; QUANTITY3 = 20; BRAND1 = Brand#12; BRAND2 = Brand#23; BRAND3 = Brand#34. It must produce the following sample output data:

```
   REVENUE
3083843.05
```

Suppliers Who Kept Orders Waiting Query (Q21)

The Suppliers Who Kept Orders Waiting query identifies suppliers, for a given nation, whose product was part of a multi-supplier order (with current status of 'F') where they were the only supplier who failed to meet the committed delivery date.

```
Return the first 100 selected rows
select
  s_name,
  count(*) as numwait
from
  supplier,
  lineitem l1,
  orders,
  nation
where
  s_suppkey = l1.l_suppkey
  and o_orderkey = l1.l_orderkey
  and o_orderstatus = 'F'
  and l1.l_receiptdate > l1.l_commitdate
  and exists (
    select
      *
    from
      lineitem l2
    where
```

```
      l2.l_orderkey = l1.l_orderkey
      and l2.l_suppkey <> l1.l_suppkey
  )
  and not exists (
    select
      *
    from
      lineitem l3
    where
      l3.l_orderkey = l1.l_orderkey
      and l3.l_suppkey <> l1.l_suppkey
      and l3.l_receiptdate > l3.l_commitdate
    )
  and s_nationkey = n_nationkey
  and n_name = '[NATION]'
group by
  s_name
order by
  numwait desc,
  s_name;
```

For validation against the qualification database the query must be executed using the following values for substitution parameters: NATION = SAUDI ARABIA. It must produce the following sample output data:

```
            S_NAME NUMWAIT
Supplier#000002829      20
```

Bibliography

[1] S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '05, pages 358–367, New York, NY, USA, 2005. ACM.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] A. Abramé and D. Habet. Efficient Application of Max-SAT Resolution on Inconsistent Subsets. In B. O'Sullivan, editor, *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 92–107. Springer International Publishing, Cham, 2014.

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 223–236, New York, NY, USA, 2010. ACM.

[5] K. Beedkar, L. Del Corro, and R. Gemulla. Fully parallel inference in Markov logic networks. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI)*, pages 205–224, 01 2013.

[6] C. Beeri and R. Ramakrishnan. On the Power of Magic. *J. Log. Program.*, 10(3&4):255–299, 1991.

[7] R. Bekkerman, M. Bilenko, and J. Langford, editors. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2011.

[8] J. Besag. Statistical Analysis of Non-Lattice Data. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 24(3):pp. 179–195, 1975.

[9] A. Borgida and J. F. Sowa. *Principles of semantic networks: explorations in the representation of knowledge*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1991.

[10] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, New York, NY, USA, October 2009.

[11] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, Mar. 1989.

[12] D. Cohen, M. Cooper, and P. Jeavons. A Complete Characterization of Complexity for Boolean Constraint Optimization Problems. In M. Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 212–226. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[13] A. Colmerauer and P. Roussel. The Birth of Prolog. In T. J. Bergin, Jr. and R. G. Gibson, Jr., editors, *History of Programming languages—II*, pages 331–367. ACM, New York, NY, USA, 1996.

[14] J. Côrte-Real, I. de Castro Dutra, and R. Rocha. Prolog programming with a map-reduce parallel construct. In R. Peña and T. Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 285–296. ACM, 2013.

[15] V. S. Costa, L. Damas, and R. Rocha. The YAP Prolog system. *TPLP*, 12(1-2):5–34, 2012.

[16] J. Cussens. Stochastic Logic Programs: Sampling, Inference and Applications. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, UAI'00, pages 115–122, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[17] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 33(3):374–425, Sept. 2001.

[18] J. Davis and P. Domingos. Bottom-up learning of Markov Network structure. In *Proceedings of the Twenty-Seventh International Conference on Machine Learning*, pages 271–280. ACM Press, 2010.

[19] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[20] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Operating Systems Design and Implementation*, pages 137–149, 2004.

[21] H. Deleau, J. Christophe, and M. Krajecki. GPU4SAT: solving the SAT problem on GPU. In *PARA 2008, 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, Trondheim, Norway, 2008.

[22] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical Report GIT-CERCS-12-01, Georgia Institute of Technology, 2012.

[23] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili. Relational Algorithms for Multi-bulk-synchronous Processors. In *Proceedings of the*

18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, pages 301–302, New York, NY, USA, 2013. ACM.

[24] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Morgan and Claypool Publishers, 1st edition, 2009.

[25] D. Fierens, G. V. den Broeck, J. Renkens, D. S. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. D. Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *TPLP*, 15(3):358–401, 2015.

[26] D. Fierens, G. V. den Broeck, I. Thon, B. Gutmann, and L. D. Raedt. Inference in Probabilistic Logic Programs using Weighted CNF's. *Technical Report*, abs/1202.3719, 2012.

[27] R. Fletcher. *Practical Methods of Optimization; (2Nd Ed.)*. Wiley-Interscience, New York, NY, USA, 1987.

[28] N. Fonseca, R. Rocha, R. Camacho, and F. Silva. Efficient Data Structures for Inductive Logic Programming. In T. Horváth and A. Yamamoto, editors, *Inductive Logic Programming: 13th International Conference, ILP 2003, Szeged, Hungary, September 29 - October 1, 2003. Proceedings*, pages 130–145. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[29] N. A. Fonseca, A. Srinivasan, F. M. A. Silva, and R. Camacho. Parallel ILP for distributed-memory architectures. *Machine Learning*, 74(3):257–279, 2009.

[30] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *In IJCAI*, pages 1300–1309. Springer-Verlag, 1999.

[31] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming. In *Logic Programming in Action*, pages 3–35. Springer Berlin Heidelberg, 1992.

[32] H. Fujii and N. Fujimoto. GPU acceleration of BCP procedure for SAT algorithms. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1–7. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012.

[33] M. Gavanelli, F. Riguzzi, M. Milano, and P. Cagnoli. Constraint and Optimization techniques for supporting Policy Making. In T. Yu, N. Chawla, and S. Simoff, editors, *Computational Intelligent Data Analysis for Sustainable Development*, Data Mining and Knowledge Discovery Series, chapter 12, pages 361–382. Chapman & Hall/CRC, Abingdon, UK, 2013.

[34] S. Geman and D. Geman. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, Nov. 1984.

[35] M. R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[36] L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[37] F. Glover. Tabu Search-Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[38] W. Gong and X. Zhou. A survey of SAT solver. *AIP Conference Proceedings*, 1836(1):020059:1–10, 2017.

[39] B. Goodarzi, M. Burtscher, and D. Goswami. Parallel Graph Partitioning on a CPU-GPU Architecture. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 58–66, May 2016.

[40] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The Lixto data extraction project: back and forth between theory and practice. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, pages 1–12, New York, NY, USA, 2004. ACM.

[41] G. Gottlob, N. Leone, and F. Scarcello. On the complexity of some inductive logic programming problems. *New Generation Computing*, 17(1):53–75, 1999.

[42] S. Grauer-Gray and J. Cavazos. Optimizing and Auto-tuning Belief Propagation on the GPU. In K. Cooper, J. Mellor-Crummey, and V. Sarkar, editors, *Languages and Compilers for Parallel Computing: 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*, pages 121–135. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[43] O. Green, R. McColl, and D. A. Bader. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340, New York, NY, USA, 2012. ACM.

[44] T. J. Green, M. Aref, and G. Karvounarakis. LogicBlox, Platform and Language: A Tutorial. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, pages 1–8, Berlin, Heidelberg, 2012. Springer-Verlag.

[45] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 675–686, Vienna, Austria, 2007. VLDB Endowment.

[46] R. Haupt and S. Haupt. *Practical Genetic Algorithms*. Wiley InterScience electronic collection. Wiley, 2004.

[47] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[48] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[49] R. Helfenstein and J. Koko. Parallel Preconditioned Conjugate Gradient Algorithm on GPU. *J. Comput. Appl. Math.*, 236(15):3584–3590, Sept. 2012.

[50] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD Conference*, pages 1213–1216, 2011.

[51] T. Jim. SD3: A Trust Management System with Certified Evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 106–115, Washington, DC, USA, 2001. IEEE Computer Society.

[52] W. L. Jorgensen. Perspective on "Equation of state calculations by fast computing machines". *Theoretical Chemistry Accounts*, 103(3):225–227, 2000.

[53] H. Kautz, B. Selman, and Y. Jiang. A General Stochastic Approach to Solving Problems with Hard and Soft Constraints. In *The Satisfiability Problem: Theory and Applications*, pages 573–586. American Mathematical Society, 1996.

[54] K. Kersting and L. D. Raedt. Bayesian Logic Programs. *Technical Report*, cs.AI/0111058, 2001.

[55] A. Kimming, B. Demoen, L. De Raedt, V. S. Costa, and R. Rocha. On the implementation of the probabilistic logic programming language ProbLog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

[56] R. Kindermann and J. L. Snell. *Markov random fields and their applications*. American Mathematical Society, 1st edition, 1980.

[57] S. Kok and P. Domingos. Learning the Structure of Markov Logic Networks. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 441–448, New York, NY, USA, 2005. ACM.

[58] S. Kok and P. Domingos. Statistical Predicate Invention. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 433–440, New York, NY, USA, 2007. ACM.

[59] S. Kok and P. Domingos. Extracting Semantic Networks from Text Via Relational Clustering. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I*, ECML PKDD '08, pages 624–639, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] D. Koller and N. Friedman. *Probabilistic Graphical Models*. MIT Press, 2009.

[61] R. Kowalski and T. Frühwirth. *Logic for Problem Solving, Revisited*. Computer science essentials. Books on Demand, 2014.

[62] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, Feb 2001.

[63] S. T. Kuhn. An axiomatization of predicate functor logic. *Notre Dame J. Formal Logic*, 24(2):233–241, 04 1983.

[64] M. E. Lalami, V. Boyer, and D. El-Baz. Efficient Implementation of the Simplex Method on a CPU-GPU System. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1999–2006, Washington, DC, USA, 2011. IEEE Computer Society.

[65] M. Lenzerini. Data integration: a theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 233–246, New York, NY, USA, 2002. ACM.

[66] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1):503–528, 1989.

[67] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[68] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 97–108, New York, NY, USA, 2006. ACM.

[69] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, New York, NY, USA, October 2005.

[70] D. Lowd and P. Domingos. Efficient Weight Learning for Markov Logic Networks. In *Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD 2007, pages 200–211, Berlin, Heidelberg, 2007. Springer-Verlag.

[71] W. R. Marczak, S. S. Huang, M. Bravenboer, M. Sherr, B. T. Loo, and M. Aref. Secureblox: customizable secure distributed data processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 723–734, New York, NY, USA, 2010. ACM.

[72] C. A. Martínez-Angeles. Datalog for GPUs. Master's thesis, Cinvestav-IPN, Mexico City, Mexico, 2013.

[73] C. A. Martínez-Angeles, I. Dutra, V. S. Costa, and J. Buenabad-Chávez. Processing Markov Logic Networks with GPUs: Accelerating Network Grounding. In K. Inoue, H. Ohwada, and A. Yamamoto, editors, *Inductive Logic Programming: 25th International Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers*, pages 122–136. Springer International Publishing, Cham, 2016.

[74] C. A. Martínez-Angeles, I. Dutra, V. Santos-Costa, and J. Buenabad-Chávez. A Datalog Engine for GPUs. In *Declarative Programming and Knowledge Management: Declarative Programming Days, KDPD 2013, Unifying INAP, WFLP, and WLP, Kiel, Germany, September 11-13, 2013, Revised Selected Papers*, Lecture Notes in Computer Science, pages 239–253. Springer International Publishing, 2014.

[75] C. A. Martínez-Angeles, H. Wu, I. Dutra, V. S. Costa, and J. Buenabad-Chávez. Relational Learning with GPUs: Accelerating Rule Coverage. *Int. J. Parallel Program.*, 44(3):663–685, June 2016.

[76] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the Construction of Internet Portals with Machine Learning. *Inf. Retr.*, 3(2):127–163, 2000.

[77] A. McDonald. Parallel WalkSAT with clause learning. In *Data analysis project papers*, 2009.

[78] E. Mendelson. *Introduction to Mathematical Logic, Fourth Edition*. Discrete Mathematics and Its Applications. Taylor & Francis, 1997.

[79] B. Merry. A Performance Comparison of Sort and Scan Libraries for GPUs. *Parallel Processing Letters*, 25:1550007, 12 2015.

[80] L. Mihalkova and R. J. Mooney. Bottom-up learning of Markov logic network structure. In *In Proceedings of the Twenty-Fourth International Conference on Machine Learning*, pages 625–632. ACM Press, 2007.

[81] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[82] M. F. Møller. A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning. *Neural Netw.*, 6(4):525–533, Apr. 1993.

[83] J. M. Mooij and H. J. Kappen. Sufficient Conditions for Convergence of the Sum-Product Algorithm. *IEEE Transactions on Information Theory*, 53(12):4422–4437, Dec 2007.

[84] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[85] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[86] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide: C++ programming with the standard template library, 2nd Ed.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[87] N. J. Nilsson. Probabilistic Logic. *Artif. Intell.*, 28(1):71–88, Feb. 1986.

[88] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling Up Statistical Inference in Markov Logic Networks Using an RDBMS. *Proc. VLDB Endow.*, 4(6):373–384, Mar. 2011.

[89] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2000.

[90] J. Noessner, M. Niepert, and H. Stuckenschmidt. RockIt: Exploiting Parallelism and Symmetry for MAP Inference in Statistical Relational Models. In *Proceedings of the 16th AAAI Conference on Statistical Relational Artificial Intelligence*, AAAIWS'13-16, pages 37–42. AAAI Press, 2013.

[91] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge Path - Parallel Merging Made Simple. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1611–1618, Washington, DC, USA, 2012. IEEE Computer Society.

[92] A. D. Palú, A. Dovier, A. Formisano, and E. Pontelli. CUD@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3):293–316, 2015.

[93] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[94] H. Poon and P. Domingos. Sound and Efficient Inference with Probabilistic and Deterministic Dependencies. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 458–463. AAAI Press, 2006.

[95] H. Poon, P. Domingos, and M. Sumner. 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 - Volume 2*, AAAI'08, pages 1075–1080. AAAI Press, 2008.

[96] P. Pospichal, J. Jaros, and J. Schwarz. Parallel Genetic Algorithm on the CUDA Architecture. In *Proceedings of the 2010 International Conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 442–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[97] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[98] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006.

[99] S. Riedel. Improving the accuracy and Efficiency of MAP Inference for Markov Logic. In *Proceedings of the 24th Annual Conference on Uncertainty in AI (UAI '08)*, pages 468–475, 2008.

[100] S. Riedel and I. Meza-Ruiz. Collective Semantic Role Labelling with Markov Logic. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, CoNLL '08, pages 193–197, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[101] P. B. Ryan and M. J. Schuemie. Evaluating Performance of Risk Identification Methods Through a Large-Scale Simulation of Observational Data. *Drug Safety*, 36(1):171–180, 2013.

[102] D. Saccà and C. Zaniolo. The generalized counting method for recursive logic queries. In G. Ausiello and P. Atzeni, editors, *ICDT '86*, volume 243 of *Lecture Notes in Computer Science*, pages 31–53. Springer Berlin Heidelberg, 1986.

[103] V. Santos Costa, K. Sagonas, and R. Lopes. Demand-Driven Indexing of Prolog Clauses. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 305–409. Springer, 2007.

[104] T. Sato and Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *In Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAIâ97*, pages 1330–1335, 1997.

[105] J. Shavlik and S. Natarajan. 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'09, pages 1951–1956, San Francisco, CA, 2009. Morgan Kaufmann Publishers Inc.

[106] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In

Proceedings of the 33rd international conference on Very large data bases, VLDB '07, pages 1033–1044, Vienna, Austria, 2007. VLDB Endowment.

[107] P. Singla and P. Domingos. Discriminative Training of Markov Logic Networks. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, AAAI'05, pages 868–873. AAAI Press, 2005.

[108] P. Singla and P. Domingos. Entity Resolution with Markov Logic. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 572–582, Washington, DC, USA, 2006. IEEE Computer Society.

[109] P. Singla and P. Domingos. Lifted First-order Belief Propagation. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 1094–1099. AAAI Press, 2008.

[110] E. Smith, J. Gondzio, and J. Hall. GPU Acceleration of the Matrix-Free Interior Point Method. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, editors, *Parallel Processing and Applied Mathematics: 9th International Conference, PPAM 2011, Torun, Poland, September 11-14, 2011. Revised Selected Papers, Part I*, pages 681–689. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[111] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[112] A. Srinivasan, T. A. Faruquie, and S. Joshi. Data and task parallelism in ILP using MapReduce. *Machine Learning*, 86(1):141–168, 2012.

[113] A. Srinivasan, R. King, S. Muggleton, and M. Sternberg. Carcinogenesis predictions using ILP. In N. Lavrac and S. Dszeroski, editors, *Inductive Logic Programming*, volume 1297 of *Lecture Notes in Computer Science*, pages 273–287. Springer Berlin Heidelberg, 1997.

[114] A. S. Tanenbaum and A. S. Woodhull. *Operating systems - design and implementation (3. ed.)*. Pearson Education, 2006.

[115] V. M. Teslyuk and Y. I. Lukomskyi. Parallel implementation of Newton's method. In *2013 XVIIIth International Seminar/Workshop on Direct and Inverse Problems of Electromagnetic and Acoustic Wave Theory (DIPED)*, pages 179–181, Sept 2013.

[116] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

[117] J. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II*. Computer Science Press, 1989.

[118] J. Wang and P. Domingos. Hybrid Markov Logic Networks. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 1106–1111. AAAI Press, 2008.

[119] W. Wei, J. Erenrich, and B. Selman. Towards Efficient Sampling: Exploiting Random Walk Strategies. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 670–676. AAAI Press, 2004.

[120] O. S. Weislow, R. Kiser, D. L. Fine, J. Bader, R. H. Shoemaker, and M. R. Boyd. New Soluble-Formazan Assay for HIV-1 Cytopathic Effects: Application to High-Flux Screening of Synthetic and Natural Products for AIDS-Antiviral Activity. *Journal of the National Cancer Institute*, 81(8):577–586, 1989.

[121] P. Wolfe. Convergence Conditions for Ascent Methods. *SIAM Review*, 11(2):226–235, 1969.

[122] M. H. Wright. Interior methods for constrained optimization. *Acta Numerica*, 1:341–407, 1992.

[123] F. Wu and D. S. Weld. Automatically Refining the Wikipedia Infobox Ontology. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 635–644, New York, NY, USA, 2008. ACM.

[124] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 107–118, Washington, DC, USA, 2012. IEEE Computer Society.

[125] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, New York, NY, USA, 2014. ACM.

[126] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/-Fission. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 2433–2442, Washington, DC, USA, 2012. IEEE Computer Society.

[127] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Generalized Belief Propagation. In *IN NIPS 13*, pages 689–695. MIT Press, 2000.

[128] J. Young, H. Wu, and S. Yalamanchili. Satisfying data-intensive queries using GPU clusters. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1314–1314, Nov 2012.

URL References (consulted February, 2018)

[129] Max-SAT 2016. Eleventh Max-SAT Evaluation. Benchmarks and Solver Requirements. http://maxsat.ia.udl.cat/requirements/.

[130] Bytedeco. JavaCPP. https://github.com/bytedeco/javacpp.

[131] Nicolas Iderhoff. nlp-datasets. https://github.com/niderhoff/nlp-datasets.

[132] Duane Merrill. CUB. https://nvlabs.github.io/cub/.

[133] Fifthteen International Conference on Theory and Applications of Satisfiability Testing (SAT-2012). The Seventh Evaluation of Max-SAT Solvers (Max-SAT-2012). http://www.maxsat.udl.cat/12/introduction/index.html.

[134] Khronos Group. OpenCL. https://www.khronos.org/opencl/.

[135] Apache Hadoop. http://hadoop.apache.org/.

[136] Christopher Ré. Project Tuffy. http://i.stanford.edu/hazy/tuffy/.

[137] Alchemy: Open Source AI. https://alchemy.cs.washington.edu/.

[138] RockIt: A query engine for Markov logic. https://github.com/jnoessner/rockIt.

[139] Ashwin Srinivasan. The Aleph Manual. http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph.

[140] Silicon Graphics International. Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/.

[141] Medicare. Medicare Hospital Compare datasets. https://data.medicare.gov/data/hospital-compare.

[142] M. Lichman. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml.

[143] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. http://openmp.org/wp/.

[144] Sean Baxter. Modern GPU library — Tutorial. http://nvlabs.github.io/moderngpu/index.html.

[145] Free Software Foundation. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[146] NVIDIA Corporation. NVIDIA Website. http://www.nvidia.com/page/home.html.

[147] NVIDIA Corporation. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[148] J. M Collins. The DTP AIDS antiviral screen program.

[149] Gurobi Optimizer. http://www.gurobi.com/index.

[150] Red Fox: A Compilation Environment for Data Warehousing. http://gpuocelot.gatech.edu/projects/red-fox-a-compilation-environment-for-data-warehousing/.

[151] Thrust: A Parallel Template Library. http://thrust.github.io/.

[152] TPC-H Transaction Processing Performance Council Benchmark H. http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp.

[153] GPU Applications. http://www.nvidia.com/object/gpu-applications-domain.html.

[154] General-Purpose Computation on Graphics Hardware. http://gpgpu.org/.