GPU-Accelerated Hypothesis Cover Set Testing for Learning in Logic

Eyad Algahtani and Dimitar Kazakov

University of York, Heslington, York YO10 5GH, UK,
ea922@york.ac.uk, kazakov@cs.york.ac.uk,
WWW home page: https://www-users.cs.york.ac.uk/kazakov/

Abstract. ILP learners are commonly implemented to consider sequentially each training example for each of the hypotheses tested. Computing the cover set of a hypothesis in this way is costly, and introduces a major bottleneck in the learning process. This computation can be implemented more efficiently through the use of data level parallelism. Here we propose a GPU-accelerated approach to this task for propositional logic and for a subset of first order logic. This approach can be used with one's strategy of choice for the exploration of the hypothesis space. At present, the hypothesis language is limited to logic formulae using unary and binary predicates, such as those covered by certain types of description logic. The approach is tested on a commodity GPU and datasets of up to 200 million training examples, achieving run times of below 30ms per cover set computation.

Keywords: Inductive Logic Programming, learning in logic, hypothesis cover set, data parallelism, GPU, GPGPU

1 Introduction

Classical ILP learners are implemented as sequential algorithms, which either consider training examples one at a time (Golem [1]) or they search through the hypothesis space computing the cover set of one hypothesis at a time (Foil [4]), or they combine elements of these two sequential strategies, e.g. they select a positive example, and then explore the hypothesis space implied by it (Progol [5], Aleph [6]). In all cases, computing the cover set of a hypothesis is a costly process that usually involves considering all negative examples[1] along with all relevant positive examples[2] in a sequential manner. This is a computation that is based on set membership tests. These tests can be parallelised efficiently through the use of data parallel computations. In particular, GPU-based data parallelism has proven very efficient in a number of application areas [15]. We propose a GPU-accelerated approach to the computation of the cover set for a given hypothesis

[1] These may not always be explicitly provided, but estimated [7] or inferred using an assumption, such as *output completeness* in Foidl [16].

[2] In the case of a cautious learner, that is all positive examples. For eager learners, only the positive examples not covered yet by another clause are considered.

for a subset of first order logic, which results in a speed up of several orders of magnitude. This approach could potentially be integrated with various strategies for the exploration of the hypothesis space. At present, the hypothesis language is limited to logic formulae using unary and binary predicates, such as those covered by certain types of description logic. This should be considered in the context of an increasing number of ILP algorithms targeting specifically description logic as a data and hypothesis language [17–19]. We test our approach on a commodity GPU, Nvidia GeForce GTX 1070, and data comprising datasets of size up to 200 million examples, achieving speeds of under 30ms on the full dataset.

The rest of this paper is structured as follows: Section 2 introduces related background information on GPU and GPGPU, Section 3 presents and evaluates our novel, parallel implementation of the cover set computation for propositional data, Section 4 extends our approach to hypotheses with binary predicates, and Section 5 discusses the results and future work on the implementation of an ILP algorithm making use of the GPU-accelerated cover set computation proposed here.

2 Related Work

Previous work on the efficient implementation of cover set evaluation in ILP has included efforts to avoid redundancy, e.g. through the use of query packs [9], and the use of concurrent computation [11]. Fonseca *et al* [8] provides a review and an evaluation of strategies for parallelising ILP, which are split in three groups. Firstly, when the target predicate amounts to a description of separate classes, each of these can be learned independently, in parallel, potentially from a separate copy of the whole dataset. Secondly, the search space can be explored in a parallel fashion, dividing the task among several processes. Thirdly, the data can be split and processed separately, in parallel, including for the task of testing hypothesis coverage, where the results of such mapping are merged in the final step.

Another way to categorise parallel evaluation approaches in ILP is according to whether they use shared or distributed memory [10]. Ohwada and Mizoguchi [11] used the former approach to combine branch-and-bound ILP search with a parallel logic programming language, while Graham, Page and Kamal [12] parallelised the hypothesis evaluation by assigning different clauses to different processors to be evaluated at the same time. In the distributed memory category, Matsui *et al* [13] studied the impact of parallel evaluation on FOIL [4], achieving linear growth in the speed-up for up to 4 processors, and lower relative gains for larger numbers as communication overheads started to become more prominent. In other work, Konstantopoulos [14] combined a parallel coverage test on multiple machines (using the MPI library) with the Aleph system [6].

Our approach shares similarities with the last two approaches [13, 14]. However, it uses the GPU in a shared memory architecture. The GPU provides much greater computational powers for the same cost when compared to a typical multi-core CPU; thus, efficient and more effective use of the available hardware resources is achieved. Also, in our approach, the entire data is copied (gigabytes of data are transferred within few seconds) from the main memory to the GPU's own memory once, before the learning process starts. The data remain in the GPU's memory until the learning process is completed. Our approach has a very low communication overhead compared to that of Konstantopoulos [14]; this reduces the learning time dramatically (especially for millions of examples) by allowing more rules to be evaluated per second. Moreover, during learning, as the CPU sends the rule for evaluation (asynchronously) to the GPU, it is then free to focus on the remaining parts of the learning process while waiting for the result.

3 GPU and GPGPU

The graphics processing unit (GPU) is a specialised hardware that has many processing cores intended to accelerate graphics-related computations. In recent years, there has been a growing interest in using the massive computation capabilities of GPUs to accelerate general purpose computations, especially in the scientific community. This has evolved into the concept of General-Purpose GPU (GPGPU) that enables the use of GPUs for the acceleration of general purpose computations [15]. The GPU is built upon the Single Instruction Multiple Data (SIMD) architecture, i.e. applying the same operation to every data item in parallel. The GPU is essentially assigning each thread to process a certain area of the input data and output its results. The number of threads depends on the GPU used, and it is not uncommon to have more than 1000 threads.

Figure 1 demonstrates the differences between the CPU and GPU on the problem of summing up two arrays. One can observe that in the CPU, the result for each row is computed sequentially using a single thread. However, in the GPU, there are many threads each of which will process a group of array rows (indices) simultaneously, resulting in a massive performance gain, especially for large data. A sequence diagram of a typical GPU-accelerated program can be seen in Figure 2.

In that figure, the program is executed mostly by the CPU. However, when the program contains a computationally intensive operation, it sends it to the GPU which calculates the results and returns them back to the CPU. After sending the computation task to the GPU, the control is returned immediately to the CPU, which can either wait for the results or do something else while waiting. When developing GPU-accelerated programs, the code for both GPU and CPU is written using the same high level programming language, residing in the same file(s) and compiled at the same time with all other code.

There are many programming platforms available that enables the writing, debugging and execution of general purpose GPU programs. Some of them are brand-specific, such as CUDA [2], others are brand-agnostic, e.g. OpenCL [3]. Since our work revolves around the CUDA platform, we will discuss it in more detail. *Compute Unified Device Architecture* (CUDA) is the Nvidia brand-specific platform for writing general purpose programs for its GPUs. CUDA divides the

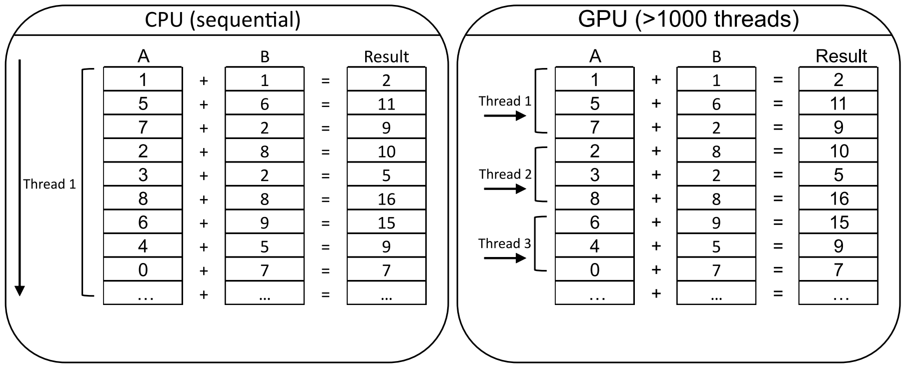


Fig. 1. Adding two arrays: CPU vs GPU

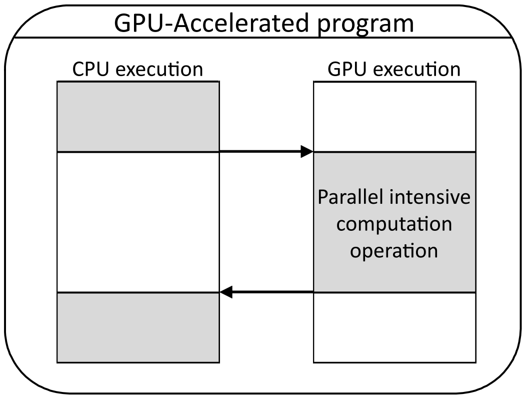


Fig. 2. Interplay between CPU and GPU of a typical GPU-accelerated program.

GPU into grids, each of which can run a separate general purpose GPU program, known as *kernel*, and contains multiple blocks of threads (thread blocks); the number of thread blocks within a grid is known as grid size. The execution of thread blocks is done through Streaming Multiprocessors (SMs). A GPU has one or multiple SMs, where each SM can execute certain number of threads blocks in parallel (independently of other SMs). The GPU has a hardware scheduler that constantly distribute thread blocks to SMs. Whenever a SM is idling (or finished computing), the scheduler feeds another thread blocks. In other words, all the SMs are computing results (in parallel) independent of each other while the hardware scheduler keeps them busy at all times by continuously feeding thread blocks to the available SM (s). When all SMs in the GPU are actively computing, the GPU can easily (depending on the hardware) execute 10,000s of threads simultaneously. A kernel needs at minimum two parameters: the grid size (number of blocks) and block size (number of threads per block). There are standard, commonly used approaches to computing the number of blocks as a function of the dataset size, and of N, the user-specified number of threads per block. For the purposes of this study, which focuses on the development of a suitable representation of the data and the development of the corresponding algorithms, we will only be concerned with the choice of N, from which all other relevant parameters of the computation will be derived.

There are many types of memory in the GPU. The first type is the *registers*, which are the fastest memory on the GPU and the smallest in capacity. The second type is *global memory.* It is the largest memory on the GPU and the slowest in terms of speed of access. The third type is the *local memory*, which is an abstraction of the global memory with smaller capacity and as slow as the global memory. The fourth type is the *shared memory*, which is a memory shared among the threads within the same block. It can be very fast (comparable to the registers) or slow (caused by multiple threads trying to access the same memory bank at the same time). The fifth type is the *constant memory*, which is a read-only memory that can have high access speeds (very close to register access speed). The last memory type is the *texture memory*, which is a read only memory suitable for 2D spatial locality needs. Texture memory has higher access speed than local and global memory. See Figure 3 for the CUDA memory model and Table 1 for a summary of CUDA memory types.

Table 1. Summary of CUDA memory types

Memory type	**Access scope**	**Access type**	**Access speed**
Register	Thread	Read/Write	Fastest
Shared	Block	Read/Write	Slow-to-fast
Constant	Application	Read only	Slow-to-fast
Texture	Application	Read only	Medium
Local	Thread	Read/Write	Slowest
Global	Application	Read/Write	Slowest

After discussing CUDA and its memory model, we will discuss the typical flow of GPU-accelerated program using CUDA. For example, assume we want to calculate the sum of two arrays, A and B, using CUDA. First, we will allocate memory to store three arrays in the GPU's memory, namely, A, B and R (in order to store the result). Second, we will copy the contents of A and B from the main memory (RAM) into the GPU allocated memory. Next, we run the GPU code to sum the two arrays in parallel (just like in Figure 1) and store the result in R. After the GPU finishes the computations, R has the result, so we copy it from the GPU's memory into the CPU. The flow of the program is identical to Figure 2. However, in Figure 4 we provide more technical details.

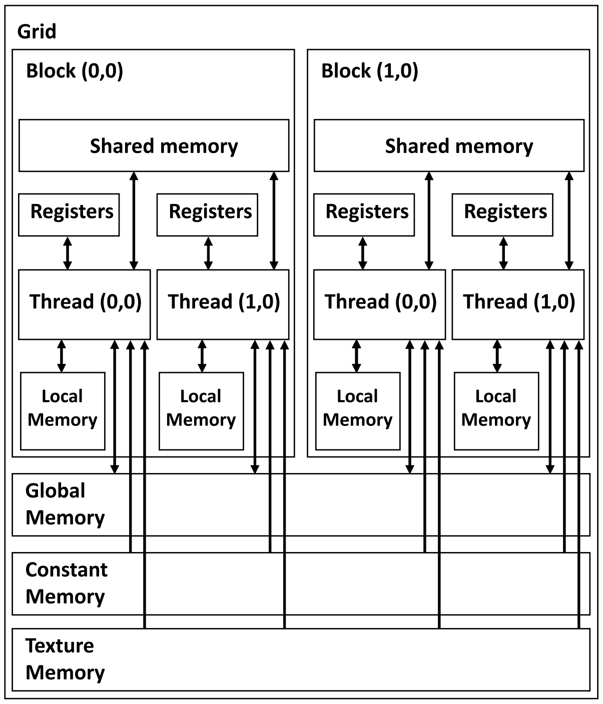


Fig. 3. CUDA memory model [20].

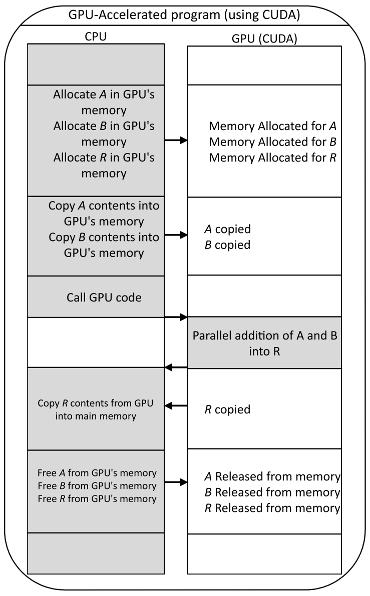

GPU-Accelerated program (using CUDA)
CPU
GPU (CUDA)
Allocate A in GPU's memory
Allocate B in GPU's memory
Allocate R in GPU's memory
Memory Allocated for A
Memory Allocated for B
Memory Allocated for R
Copy A contents into GPU's memory
Copy B contents into GPU's memory
A copied
B copied
Call GPU code
Parallel addition of A and B into R
Copy R contents from GPU into main memory
R copied
Free A from GPU's memory
Free B from GPU's memory
Free R from GPU's memory
A Released from memory
B Released from memory
R Released from memory

Fig. 4. Typical CUDA-accelerated application

4 Computing Propositional Hypothesis Cover Sets

4.1 Proposed Method

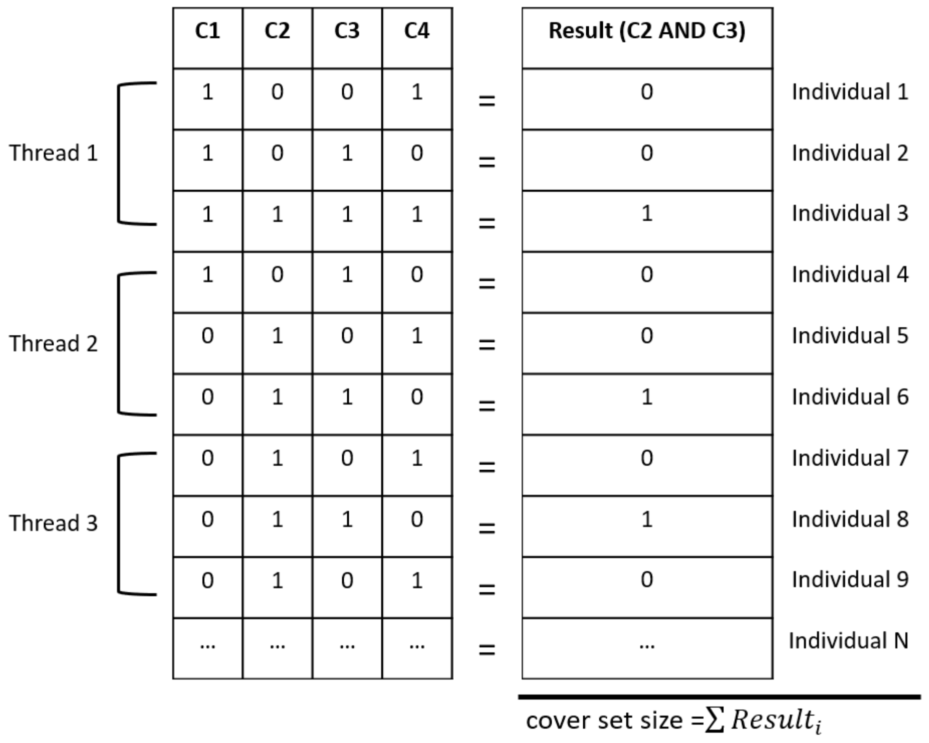
A GPU can manipulate matrices very efficiently. Here this is first used to show how to speed up the calculation of the cover set of propositional hypotheses. We adopt the terminology used in Description Logic, and talk about *concepts*, defined as sets over a universe of *individuals* in the same way in which unary predicates can be defined for a domain consisting of a set of terms. We can represent a propositional data set as a Boolean 2D matrix M with each column corresponding to a concept, and each row to an individual (see Figure 5). The membership of an individual to a concept (i.e. whether the unary predicate is true when it takes this term as argument) is represented as a Boolean value in the matrix. To make it possible to process a large amount of data (of the order of gigabytes), this array is stored in the global memory of the GPU.

Using the above representation, it is possible to employ data parallelism (Single-Instruction-Multiple-Data) for the application of logic operations to any subset of concepts. The rows of the table are divided among multiple threads. For maximum efficiency, the number of threads per block (block size) is always chosen to be equal to a power of 2, $N = 2^i$, where the upper limit of i is imposed by the GPU hardware. Since this article does not focus on performance optimisation, we only report results for $N = 32$ (threads/block), a value that resulted in good performance across the board that is representative for the problem and GPU at hand. The grid size for the kernel is then calculated from the number of individuals and the block size. For 200,000,000 individuals and $N = 32$, this results in the use of 6,250,000 blocks.

Now the cover set of a conjunction $\cap_{i=1}^{m} C_i$ of any set of concepts can be calculated through the parallel execution of all threads T_j, each of which processing sequentially the individuals I_k it has been allocated (see Algorithm 1). Lazy evaluation is used to calculate whether each individual is a member of the conjunction of concepts: if it is not covered by one of them, the rest are ignored and the result is returned immediately (see the `if...break` line of pseudocode).

The cover set of a disjunction $\cup_{i=1}^{m} C_i$ of a set of concepts is computed in an analogous way, starting for each individual with `result` set to 0, and applying the logic `OR` operator until the temporary result is set to 1 or all concepts in `S` are considered. Similarly, the negation operator divides all individuals among multiple threads, and returns a vector `result(1..numberOfIndividuals)` with all Boolean values negated.

In all cases, the size of the cover set is computed by summing up the elements of the vector `result(1..numberOfIndividuals)`. This is done with the help of the Nvidia Thrust library in a process that adds a very small overhead, which is included in all reported execution times. For all logic operators, the vector representing the cover set can be left in the global memory of the GPU to implement *memoization* [21]. When given two sets of positive and negative examples of a target concept, computing the cover set for each is controlled by a separate CPU process, and both are run in parallel. The CPU threads communicate directly



	C1	C2	C3	C4		Result (C2 AND C3)	
Thread 1	1	0	0	1	=	0	Individual 1
	1	0	1	0	=	0	Individual 2
	1	1	1	1	=	1	Individual 3
Thread 2	1	0	1	0	=	0	Individual 4
	0	1	0	1	=	0	Individual 5
	0	1	1	0	=	1	Individual 6
Thread 3	0	1	0	1	=	0	Individual 7
	0	1	1	0	=	1	Individual 8
	0	1	0	1	=	0	Individual 9
	...	...	...	...	=	...	Individual N

cover set size $= \sum Result_i$

Fig. 5. Concepts $\times$ Individuals matrix representation

with the GPU to conduct the coverage test (in the GPU) as well as to retrieve back the results. All results here report the overall run time of executing both parallel CPU threads.

It should be noted that the 2D matrix shown in Figure 5 and discussed above is represented internally as a 1D array using Row-major order in order to minimise the time for memory allocation/access. E.g. allocating memory for a 10×10 array would require $1 + 10 = 11$ memory allocation calls, whereas a 1D array of size 100 would only need one such call. Allocating memory sequentially also facilitates *coalesced* global memory access [20].

4.2 Results and Evaluation

Due to the lazy evaluation strategy used, the Worst Case Execution Time (WCET) for evaluating the cover set of a conjunction of concepts is when the

Algorithm 1 For a Boolean matrix M (individuals × concepts)

procedure PARALLELCONCEPTCONJUNCTIONCOVERSET

```
set S := list of concepts in conjunction
parallel_foreach thread T_i
|  foreach individual I_j in thread T_i
|  |  set result(row(I_j)) := 1
|  |  foreach concept C_k in set S
|  |  | set result(row(I_j)) := result(row(I_j)) && M(row(I_j),column(C_k))
|  |  | if (result(row(I_j)) == 0) break
|  |  endfor
|  endfor
endfor
```

return Boolean array: result(1..numberOfIndividuals)

membership matrix M contains all 1s. This also computationally equivalent to the WCET for evaluating the cover set a disjunction of concepts with M only made of zeros. Here we list the parallel (GPU) and Serial (GPU and CPU) execution times for data sets in which the number of individuals (Table 2, Table 3 and Figure 6) or the number of concepts (Table 4 and Figure 7) is varied. The counting of examples (final step) for both CPU (Serial) and GPU (Serial and Parallel) are computed in parallel (in the GPU) using Nvidia's Thrust library. Also, due to CPU caching, there are fluctuations in the CPU execution times, i.e. in the serial CPU experiment.

Table 2. Conjunction/Disjunction of concepts cover set run times in parallel (GPU)

Instances (pos+neg)	4 concepts			
	Time (all 1s)		Time (all 0s)	
	conj	disj	conj	disj
2×100	10.69 μs	9.09 μs	10.69 μs	11.20 μs
$2 \times 1,000$	12.83 μs	16.38 μs	11.14 μs	11.81 μs
$2 \times 10,000$	20.35 μs	29.95 μs	17.34 μs	20.35 μs
$2 \times 100,000$	44.16 μs	31.97 μs	31.55 μs	45.02 μs
$2 \times 1,000,000$	315.65 μs	224.74 μs	223.30 μs	314.08 μs
$2 \times 10,000,000$	2.76 ms	2.05 ms	2.06 ms	2.75 ms
$2 \times 100,000,000$	27.55 ms	20.63 ms	19.10 ms	25.45 ms

The results show that both the worst case and best case execution times have linear time complexity with respect to the number of individuals. Clearly, memoization can be used as a trade-off between time and space complexity, e.g. by using the existing result for $\cap_{i=1}^{m} C_i$ in order to compute the cover set of the conjunction of any superset of these concepts. When the number of concepts is

Table 3. Conjunction of Concepts: Serial CPU vs Serial GPU

Instances (pos+neg)	4 concepts			
	Serial GPU (Single thread)		Serial CPU (Single thread)	
	All 1s	All 0s	All 1s	All 0s
2×100	222.7 μs	31.6 μs	$\sim 439.32\mu s$	$\sim 624\mu s$
$2 \times 1,000$	2.1 ms	109.9 μs	$\sim 994.59\mu s$	$\sim 1.44ms$
$2 \times 10,000$	21.2 ms	1 ms	$\sim 1.79ms$	$\sim 1.63ms$
$2 \times 100,000$	211.6 ms	10 ms	$\sim 3.68ms$	$\sim 1.94ms$
$2 \times 1,000,000$	1.8 S	99.9 ms	$\sim 12.12ms$	$\sim 4.2ms$
$2 \times 10,000,000$	-	-	$\sim 103.02ms$	$\sim 29.23ms$
$2 \times 100,000,000$	-	-	$\sim 969.66ms$	$\sim 262.72ms$

Table 4. Computing conjunction of N concepts for varying N in parallel (GPU)

Instances (pos+neg)	4 concepts		8 concepts		16 concepts		32 concepts	
	Time (all 1s)	Time (all 0s)	Time (all 1s)	Time (all 0s)	Time (all 1s)	Time (all 0s)	Time (all 1s)	Time (all 0s)
$2 \times 10,000,000$	2.77 ms	2.06 ms	4.38 ms	1.95 ms	7.41 ms	2.34 ms	18.66 ms	3.88 ms

varied, the slope of the WCET plot appears to suggest an approximately linear growth.

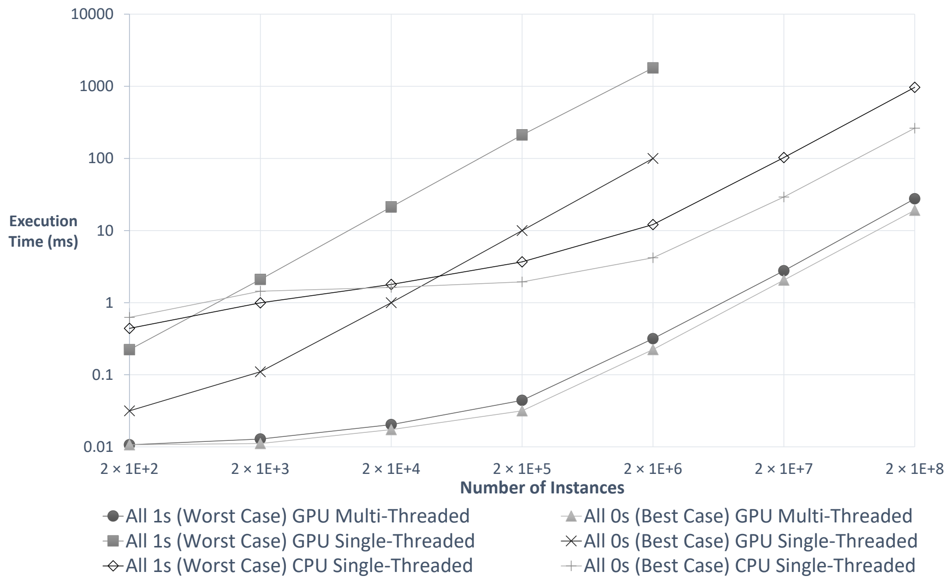


Fig. 6. Computing conjunction of N concepts, $N = 4$ (data from Table 2 & Table 3)

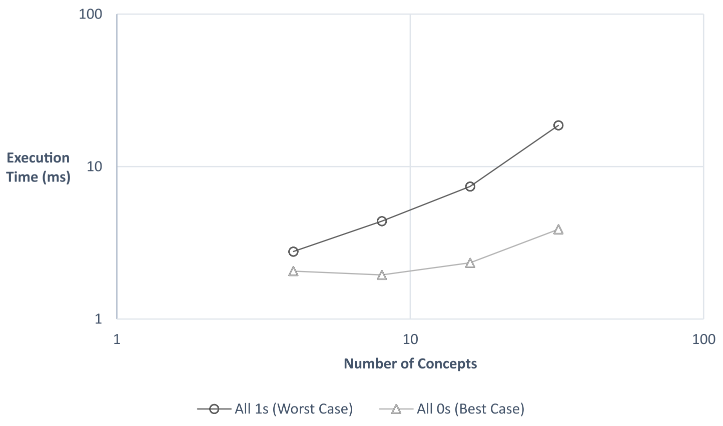


Fig. 7. Computing conjunction of N concepts for 2×10^7 individuals and varying N (data from Table 4)

5 Computing Cover Sets for Hypotheses with Binary Predicates

We shall now consider how the data and hypothesis languages can be extended to include *roles*, i.e. binary predicates. We adopt a representation for this type of data as shown in Figure 8. We use an array with three columns in which the middle contains the name of the role/predicate, here represented with an integer. The pair of individuals related through a given role are listed in Column 1 and Column 3, respectively.

With this representation in mind, we can, for instance, compute a hypothesis of the type $\exists role.(C_1 \sqcap \cdots \sqcap C_m)$, e.g. $\exists hasChild.(Female \sqcap Musician)$ using Algorithm 1 to compute the conjunction and Algorithm 2 to apply the existential restriction operator. In Alg. 2, $setAllElementsInResultToVal\_inParallel(value)$ is a utility function used to fill all the elements of the result array (in parallel) with either 0 or 1. Algorithms for other DL operators, such as value restriction or number restriction can be defined in a similar way.

Algorithm 2 For an individual-role-individual matrix R and Boolean matrix M (individuals × concepts)

procedure PARALLELEXISTENTIALRESTRICTION

```
Call setAllElementsInResultToVal_inParallel(0)
var Concept := the concept in the restriction
set IndA := list of individualA (individualA values) from R matrix (same role)
set IndB := list of individualB (individualB values) from R matrix (same role)
parallel_foreach thread T_i
| foreach individualA I_A in set IndA
| | | set result(row(I_A)) := M(row(I_B),column(Concept))
| endfor
endfor
```

return Boolean array: result(1..numberOfIndividuals)

Individual A	Role	Individual B
6	1	46
6	1	5
9	2	14
1079	3	78
749	3	43
465	4	89
89	4	700
658	4	133
98	5	1079
...	...	...

Fig. 8. Individual-Role-Individual matrix representation

6 Conclusions

We have demonstrated here how GPGPU can be used to accelerate the computation of the cover set for a hypothesis expressed in propositional logic. The results show that even the use of a commodity GPU can provide the ability to process data sets of size well beyond what can be expected from a CPU-based sequential algorithm of the same type, and within a time that makes the evaluation of many thousands of hypotheses on a dataset with 10^8 training examples a viable proposition. We have also indicated how our approach can be extended to handle binary predicates. The goal in sight is the ability to evaluate hypotheses on DL databases, e.g. linked open data ontologies. The approach adopted here fits well with the open world assumption that is commonly made in DL. An example of the potential benefits is, for instance, the ability to learn about user preferences in e-commerce [19]. A case study with real-world data would need to identify the type of DL needed and potentially extend appropriately the algorithms for the computation of the cover set, and add an appropriate search strategy.

References

1. Muggleton, S. and Feng, C.: Efficient Induction of Logic Programs. Turing Institute, pp. 368–381 (1990).
2. CUDA Zone. URL: https://developer.nvidia.com/cuda-zone. Retrieved: June 2018.
3. OpenCL Overview. URL: https://www.khronos.org/opencl/. Retrieved: June 2018.
4. Quinlan, R.: Learning Logical Definitions from Relations. *Machine Learning* 5:239–266 (1990).
5. Muggleton, S.: Inverse Entailment and Progol. *New Generation Computing* 13(3): 245–286 (1995).
6. Srinivasan, A.: The Aleph Manual (2001).
URL: http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph.html
7. Muggleton, S.: *Learning from Positive Data.* Selected Papers from the 6th International Workshop on Inductive Logic Programming, pp. 358–376 (1996).
8. Fonseca, N. A., Silva, F. and Camacho, R.: *Strategies to Parallelize ILP Systems.* Inductive Logic Programming: Proc. of the 15th Intl Conf. (2005).
9. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J. and Vandecasteele, H.: *Executing Query Packs in ILP.* International Conference on Inductive Logic Programming ILP 2000, pp. 60–77 (2000).
10. Fonseca, N. A., Srinivasan, A., Silva, F. and Camacho, R.: Parallel ILP for Distributed-memory Architectures. *Machine Learning* 74(3):257–279 (2009).
11. Ohwada, H., Mizoguchi, F.: *Parallel Execution for Speeding up Inductive Logic Programming Systems.* Proceedings of the 9th Intl Workshop on Inductive Logic Programming, pp. 277–286 (1999).
12. Graham, J., Page, D., Kamal, A.: *Accelerating the Drug Design Process through Parallel Inductive Logic Programming Data Mining.* Proceedings of the 2003 IEEE Bioinformatics Conference CSB2003 (2003).
13. Matsui, T., Inuzuka, N., Seki, H., Itoh, H.: *Comparison of Three Parallel Implementations of an Induction Algorithm.* Proceedings of the 8th Intl Parallel Computing Workshop, pp. 181–188 (1998).
14. Konstantopoulos, S. K..: *A Data-parallel Version of Aleph.* Proceedings of the Workshop on Parallel and Distributed Computing for Machine Learning (2003).
15. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T.: A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113 (2007).
16. Califf, M.E. and Mooney, R.J.: Advantages of Decision Lists and Implicit Negatives in Inductive Logic Programming. *New Generation Computing* 16, 263–281 (1998).
17. Bühmann, L., Lehmann, J. and Westphal, P.: DL-Learner – A Framework for Inductive Learning on the Semantic Web. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web* 39, 15–24 (2016).
18. Fanizzi, N., d'Amato, C. and Esposito, F.: DL-FOIL: Concept Learning in Description Logic. *Proc. of the 18th Conf. on Inductive Logic Programming* (2008).
19. Qomariyah, N. and Kazakov, D.: *Learning from Ordinal Data with Inductive Logic Programming in Description Logic.* Proc. of the Late Breaking Papers of the 27th Intl Conf. on Inductive Logic Programming, 38–50 (2017).
20. NVIDIA CUDA Programming Guide (2007).
URL: http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf. Retrieved: May 2018.
21. Michie, D.: Memo Functions and Machine Learning, *Nature* 218:19–22 (1968).