GPU-Accelerated Hypothesis Cover Set Testing for Learning in Logic

Eyard Algahtani and Dimitar Kazakov
University of York

28th ILP Conference, Ferrara, 4 Sep 2018
GPU-Accelerated Hypothesis Cover Set Testing for Learning in Logic

Eyas Algahtani and Dimitar Kazakov
University of York

28th ILP Conference, Ferrara, 4 Sep 2018
Overview

- Classic ILP algorithms combine cover set evaluation with a search algorithm using that result to find the best hypothesis.
Overview

• Classic ILP algorithms combine cover set evaluation with a search algorithm using that result to find the best hypothesis.
Overview (2)

• General-purpose GPU computation allows data-parallelism to be used in finding the cover set of logic hypotheses.

• Our long-term aim is: the efficient implementation of classic ILP-inspired algorithms for the Description Logic (DL) domain.
Description Logics

• Make use of unary predicates (concepts) and binary predicates, so called roles, e.g.: car1 in_front_of car2, car1 size short

• Several classes of DL exist depending on their expressivity, e.g. whether they have:
  ✦ Existential restriction: ∃drives.Ferrari
  ✦ Value restriction: ∀drives.Ferrari
  ✦ Number restrictions: ≥2 drives.Ferrari (cf. ‘2 Jags’ Prescott)
  ✦ Transitive roles, inverse roles, etc.
Description Logics (2)

- Expressivity of a given DL may affect decidability
- DL vs Horn clauses: neither subsumes the other. A good overlap, e.g. these definitions* are equivalent (for explicit types & non-transitive def. of `infront/2`):

```prolog
eastbound(X):-
    has_car(X,Y), shape(Y, rectangle), infront(Y,Z), load(Z, triangle).
```

```
Eastbound ≡ Train \in has_car.(∃ shape. Rectangle \in infront.(∃ load. Triangle))
```

```
Eastbound ≡ Train \in has_car.(∀ shape. Rectangle \in ∀ infront.(∀ load. Triangle))
```

*Horn clause example from:
Relational Learning for Description Logics

- DL-FOIL (Fanizzi, d’Amato, Esposito 2008)
- DL-Learner (Buehmann, Lehmann, Westphal 2016)
- APARELL: Learning ordinal relations (Qomariyah & Kazakov 2017)
Speeding up ILP Learners

• Avoid redundancy:
  ✦ Query packs (Blockeel et al. 2000)

• Parallelise computation (Fonseca et al. 2005):
  ✦ animal(X,fish) || animal(X,mammal) || animal(X,bird)
  ✦ divide the search space among different threads
  ✦ split the data, learn, merge results
Speeding up Hypothesis Evaluation

• Given a GPU
  ✦ The CPU can run the search algorithm while
  ✦ the GPU evaluates individual hypotheses.

• We’re using Nvidia GeForce GTX 1070 GPU running CUDA library
CPU ↔ GPU Interplay

- GPU-Accelerated program
  - CPU execution
  - Parallel intensive computation operation
  - GPU execution
GPU Hypothesis Evaluation for Propositional Data

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Result (C2 AND C3)

<table>
<thead>
<tr>
<th></th>
<th>Individual 1</th>
<th>Individual 2</th>
<th>Individual 3</th>
<th>Individual 4</th>
<th>Individual 5</th>
<th>Individual 6</th>
<th>Individual 7</th>
<th>Individual 8</th>
<th>Individual 9</th>
<th>Individual N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

cover set size = \( \sum Result_i \)
GPU Hypothesis Evaluation for Propositional Data

• Membership of unary predicates/concepts is represented through a 2D binary matrix in the ‘global’ GPU memory (shared by all threads). There are two such matrices for the ⊕, resp. ⊖ examples of the target concept.

• Data parallelism is used to compute conjunction, disjunction or negation of concepts: the matrix is split up, and a thread assigned to each part.

• A 1D array is used to record the membership of each individual in the hypothesis being tested, e.g. \( C_1 \cup C_2 \cup C_3 \). Lazy evaluation can be used. Total coverage is added up by the GPU using reduction-sum.

• Coverage of ⊕ and ⊖ ex.s of the target concept is counted separately.
Each result (e.g. $C_1 \cup C_2 \cup C_3$) can be directly memoized by simply adding it as another column to the $individuals \times concepts$ matrix. The memory for it needs to be preallocated though to make the process efficient.

- I.e. if we have $N$ concepts, we need to allocate a combined matrix of size $individuals \times (N+M)$. 
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)
<table>
<thead>
<tr>
<th>Instances (pos+neg)</th>
<th>4 concepts</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>conj</td>
<td>disj</td>
</tr>
<tr>
<td></td>
<td>(all 1s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 × 100</td>
<td>10.69 μs</td>
<td>9.09 μs</td>
<td></td>
</tr>
<tr>
<td>2 × 1,000</td>
<td>12.83 μs</td>
<td>16.38 μs</td>
<td></td>
</tr>
<tr>
<td>2 × 10,000</td>
<td>20.35 μs</td>
<td>29.95 μs</td>
<td></td>
</tr>
<tr>
<td>2 × 100,000</td>
<td>44.16 μs</td>
<td>31.97 μs</td>
<td></td>
</tr>
<tr>
<td>2 × 1,000,000</td>
<td>315.65 μs</td>
<td>224.74 μs</td>
<td></td>
</tr>
<tr>
<td>2 × 10,000,000</td>
<td>2.76 ms</td>
<td>2.05 ms</td>
<td></td>
</tr>
<tr>
<td>2 × 100,000,000</td>
<td>27.55 ms</td>
<td>20.63 ms</td>
<td></td>
</tr>
</tbody>
</table>
CPU (1 thread) v GPU (1, 32 threads/block)  
4 attributes/concepts/unary predicates
GPU vs single-thread CPU

• For 2,000,000 individuals and 4 concepts, and the worst case w.r.t. lazy evaluation:
  ✦ 1 thread/block GPU is \(~150\) times slower
  ✦ 64-thread/block GPU is \(~38\) times faster
\[ t = f(\#\text{concepts}) \]
Due to lazy evaluation, increasing the number of concepts does not necessarily increase execution time.

The worst case time complexity increases approx. linearly as hoped for.
Representing Roles on the GPU

<table>
<thead>
<tr>
<th>Individual A</th>
<th>Role</th>
<th>Individual B</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>1079</td>
<td>3</td>
<td>78</td>
</tr>
<tr>
<td>749</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>465</td>
<td>4</td>
<td>89</td>
</tr>
<tr>
<td>89</td>
<td>4</td>
<td>700</td>
</tr>
<tr>
<td>658</td>
<td>4</td>
<td>133</td>
</tr>
<tr>
<td>98</td>
<td>5</td>
<td>1079</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Algorithm 4 Compute Existential Restriction (∃)

procedure ParallelExistentialRestrictionOnConcepts(role, role_start_ind, role_end_ind, S)

Call setAllElementsInResultToVal_inParallel(0)
set S := list of concepts in conjunction
set IndA := list of individualA (individualA values) in the same Role
set IndB := list of individualB (individualB values) in the same Role
parallel foreach thread T i
  | foreach individualA I_A in set IndA
  | | set result(row(I_A)) := 1
  | | foreach concept C_k in set S
  | | | set result(row(I_A)) := result(row(I_A)) && M(row(I_B), column(C_k))
  | | if (result(row(I_A)) == 0) break
  | endfor
endfor

return Boolean array: result(1..numberOfIndividuals)
Algorithm 5 Compute Value Restriction (\forall)

procedure ParallelUniversalRestrictionOnConcepts(role, role_start_ind, role_end_ind, S)

Call setAllElementsInResultToVal_inParallel(1)
set S := list of concepts in conjunction
set IndA := list of individualA (individualA values) in the same Role
set IndB := list of individualB (individualB values) in the same Role
parallel foreach thread T i
  | foreach individualA I_A in set IndA
  |   | foreach concept C_k in set S
  |   |   | set result(row(I_A)) := result(row(I_A)) && M(row(I_B), column(C_k))
  |   | if (result(row(I_A)) == 0) break
  | endfor
endfor

return Boolean array: result(1..numberOfIndividuals)
The end