A Datalog Engine for GPUs

Carlos A. Martínez Angeles

camartinez@cinvestav.mx
Computer Science Department
Centre for Research & Postgraduate Studies
National Polytechnic Institute
Cinvestav-IPN

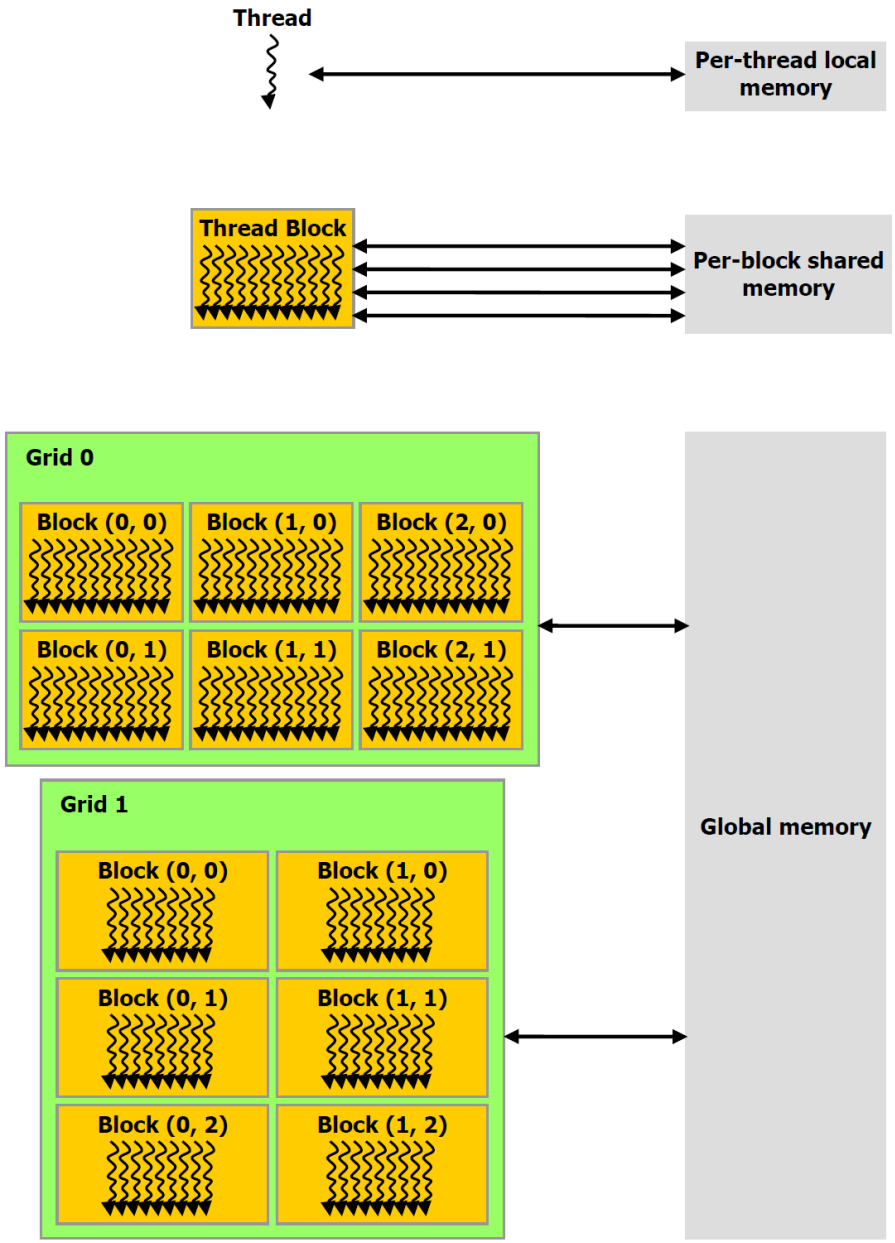
September 13, 2013

Contents

GPUs

- Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput.
- GPUs can be seen as SIMD machines: they consist of many processing elements that run all a *same function* but on distinct data items.
- GPUs are used in a wide array of applications, including gaming, bioinformatics, chemistry, finance, etc.
- CUDA a software platform used to program GPUs. It extends C by allowing the definition of functions, called *kernels*, that are executed in parallel.

GPU architecture



Datalog

- Datalog is a language based on first order logic that has been used as a data model for relational databases.
- A Datalog program consist of facts about a subject of interest and rules to deduce new facts.
- Facts and rules are specified using atomic formulas, which consist of predicate symbols with arguments, e.g.:

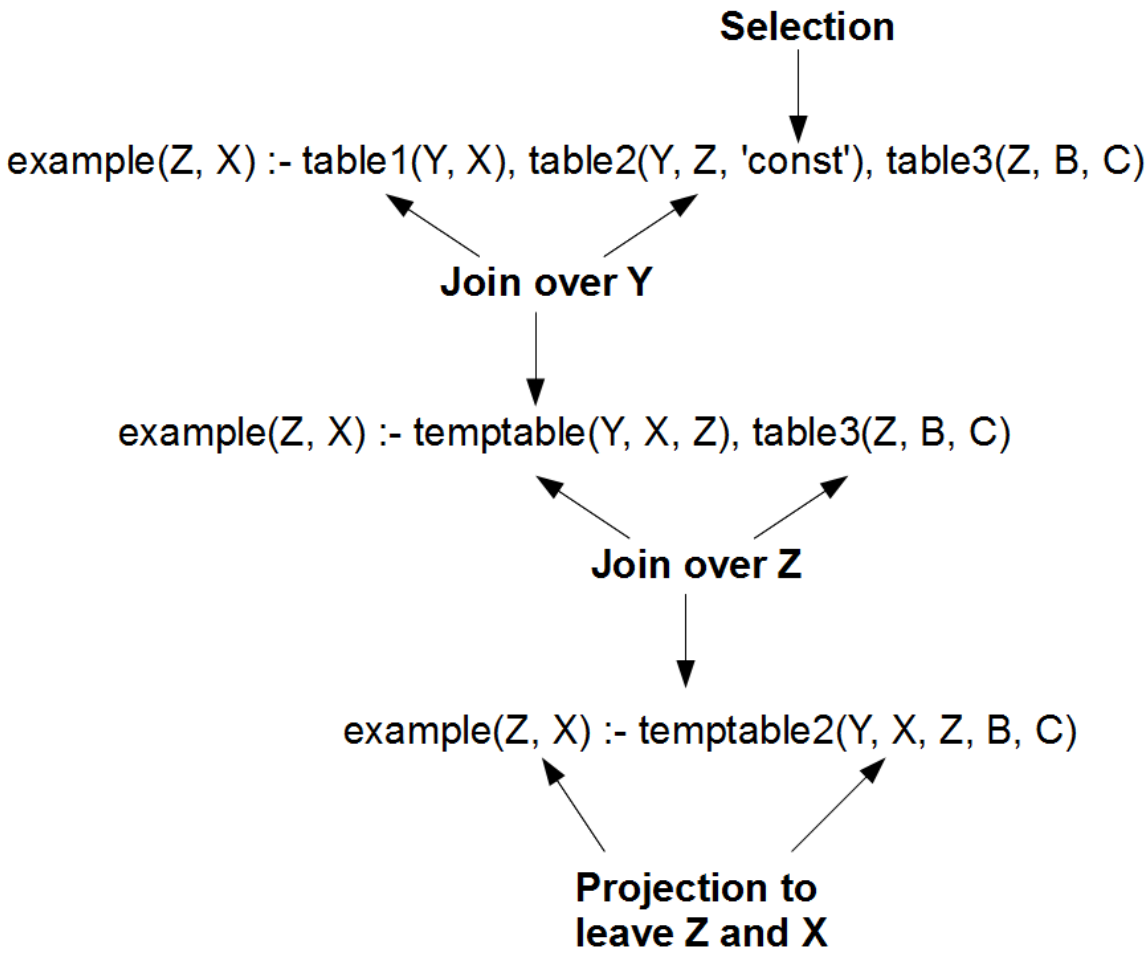
```
FACTS
father(harry, john).
father(john, david).

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

Evaluation of Datalog programs

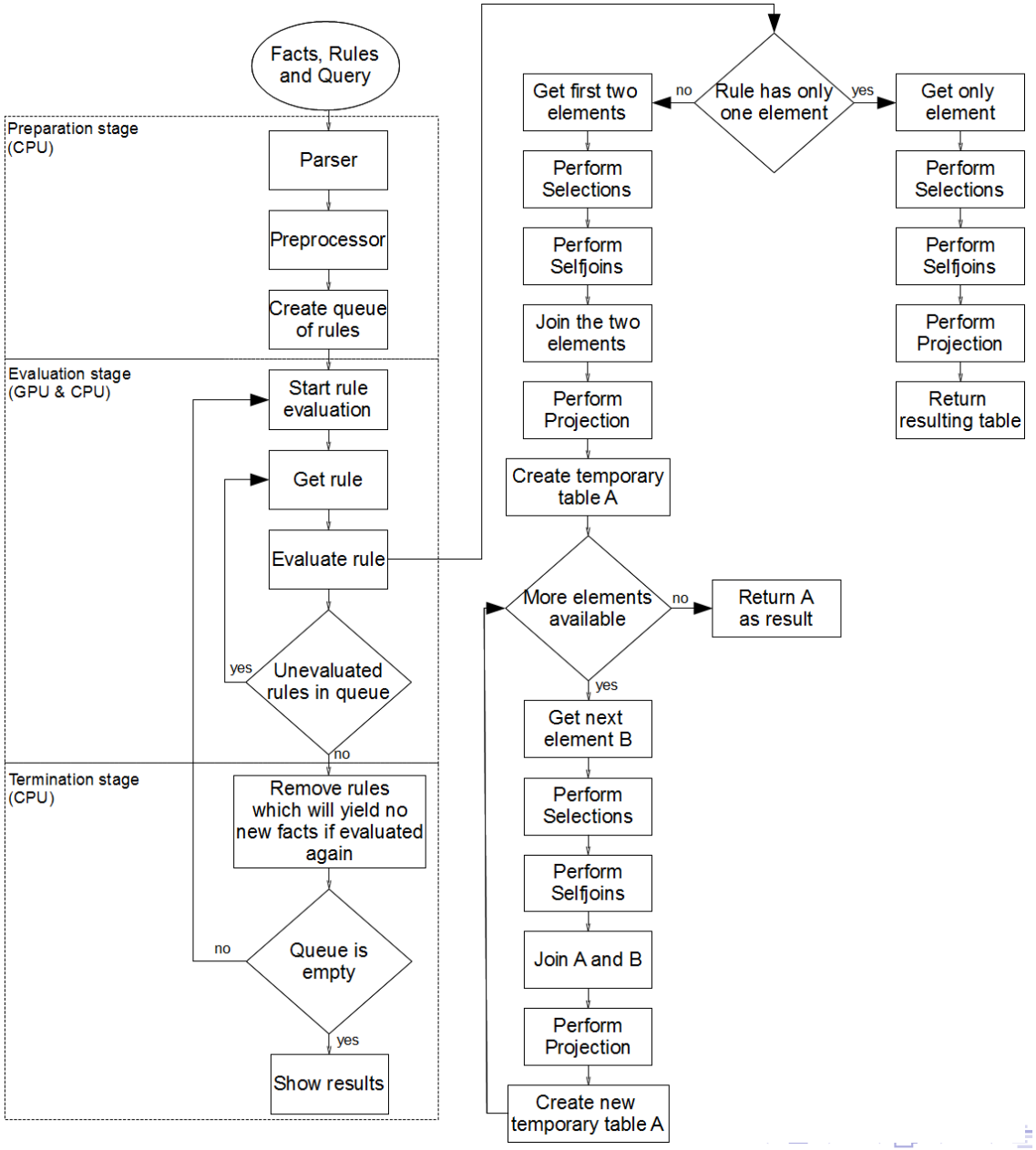
- Datalog programs can be evaluated through a top-down approach or a bottom-up approach.
- Bottom-up applies the rules to the given facts, deriving new facts, and repeating this process with the new facts until no more facts are derivable.
- The query is considered only at the end, when the facts matching the query are selected.
- Benefits of this approach include the fact that rules can be evaluated in any order and in a highly parallel manner.

Evaluation based on relational algebra operators

- Datalog rules can be evaluated using the typical relational algebra operators *select*, *join* and *projection*.



Our Datalog engine for GPUs



Parsing

- We translate facts and rules to numbers, keeping their corresponding strings in a hashed dictionary.
- Each unique string is assigned a unique id, equal strings are assigned the same id.
- The dictionary is used at the very end when the final results are to be displayed.
- The idea is to capitalise on the GPU capacity to process numbers and to have a constant processing time for each tuple.

Preprocessing

- For each rule, we specify what operations to perform and with which arguments should they be performed.
- To do so, we create small arrays for each operation, e.g.:

```
fact1(A,X,Y,Z), fact2(Z,X,B,C,Y).
-> [1, 1, 2, 4, 3, 0]
```

- These arrays are loaded in the *shared memory* of the GPU.
- The idea is to allow each thread to work with the correct arguments without having to calculate them.

Memory management

- We minimize transfers between GPU memory and host memory by maintaining facts and rule results in GPU memory for as long as possible.
- To do so, we maintain a list with information about each fact and rule result resident in GPU memory.
- Similar to Least Recently Used (LRU) page replacement algorithm.

Selection

- The size of the result in a selection is not known beforehand.
- Our selection uses three different kernel executions.
  - The first kernel marks all the rows that satisfy the selection arguments with a value 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.

Input numbers	1	1	0	0	1	0	...
Prefix sum	1	2	2	2	3	3	...

Join

- We use a modified version of the Indexed Nested Loop Join:
  - A CSS-Tree (Cache Sensitive Search Tree) is created with one of the columns to join.
  - CSS-Trees can be constructed in parallel and their tree traversal is performed via address arithmetic.
  - Using the tree, a preliminary join is made to obtain an array similar to that of the selection.
  - A second join writes the results.

Other operations

- **Projection** simply involves taking all the elements of each required column and storing them in a new memory location.
- **Multijoin** is similar to the join. The difference is that, when performing the first join, we compare any additional columns involved.
- **Selfjoin** is similar to the selection. The difference is that instead of checking constant values, it checks the values of the columns affected.

Optimisations

- **Duplicate Elimination.** When a rule iteration is finished, its result is sorted and the duplicates removed.
- **Optimising projections.** Executing a *projection* at the end of each join, allows us to discard unnecessary columns earlier in the computation.
- **Fusing operations.** Operations are applied together to a data set in a single read of the data set, as opposed to one operation per read of the data set.

Experimental Evaluation

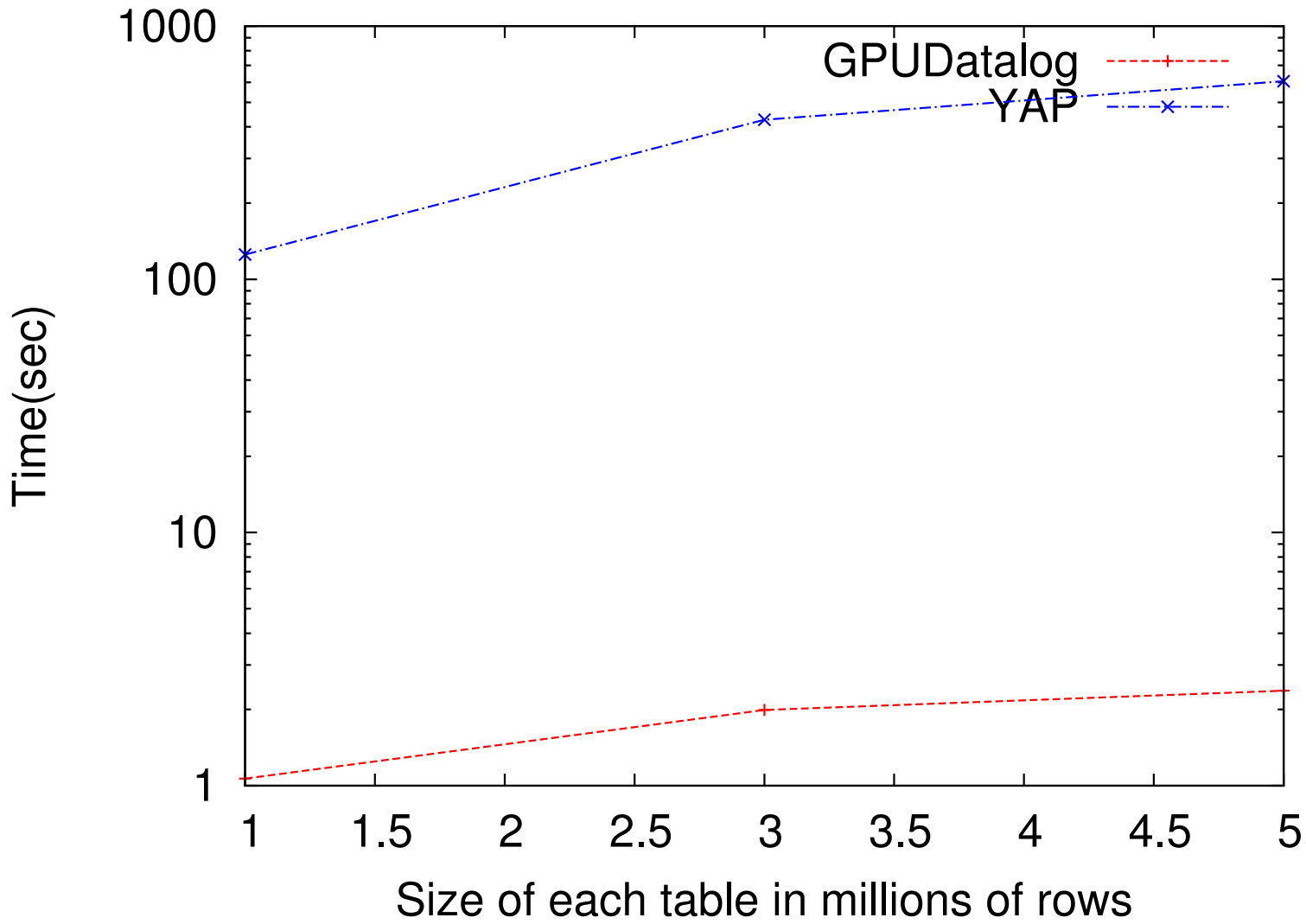
- **Hardware**. *Host platform*: Intel Core 2 Quad CPU Q9400 2.66GHz (4 cores in total), Kingston RAM DDR2 6GB 800 MHz. *GPU platform*: Fermi GeForce GTX 580 - 512 cores - 1536 MB GDDR5 memory.
- **Software.** Ubuntu 12.04.1 LTS 64bits. CUDA 5.0 Production Release, gcc 4.5, g++ 4.5. YAP 6.3.3 Development Version, Datalog 2.4, XSB 3.4.0.
- We compared our engine against XSB, YAP and MITRE Datalog.
- We are at this stage interested in the performance benefit of using GPUs for the evaluation of Datalog queries, as opposed to using a CPU only.

Join over four big tables

```
join(X,Z) :- table1(X), table2(X,4,Y), table3(Y,Z,Z),
             table4(Y,Z).
join(X,Z)?
```

- Four tables, all with the same number of rows filled with random numbers, are joined together.
- It was used to test all the different operations of our engine.
- Our engine is roughly 200 times faster than YAP.
- Joins were the most costly operations with the Multijoin alone taking more than 70% of the total execution time.

Join over four big tables results

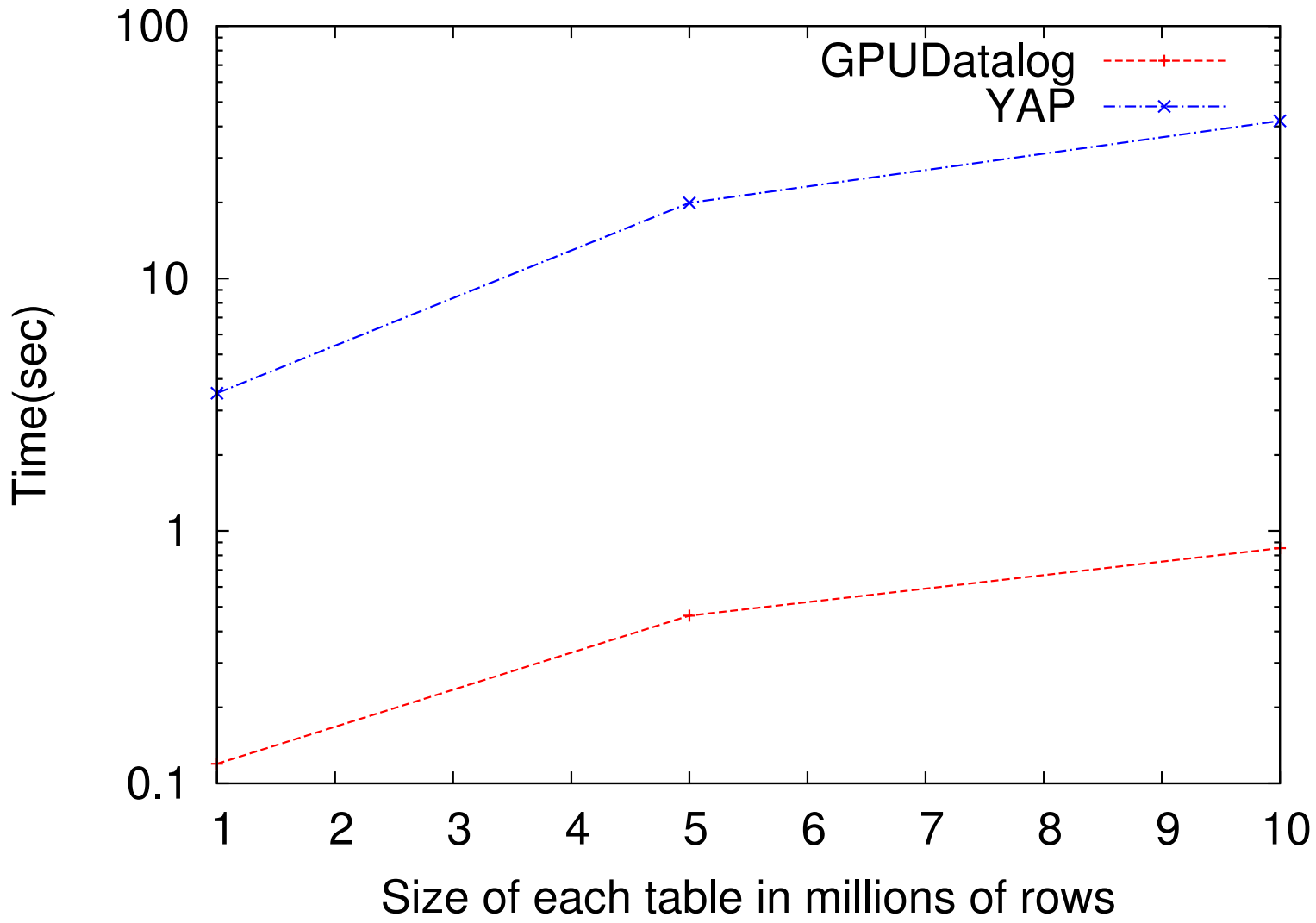


Transitive closure of a graph

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- edge(X,Y), path(Y,Z).
path(X,Y)?
```

- The edges of a graph are represented by a table with two columns filled with random numbers.
- The idea is to find all the nodes that can be reached if we start from a particular node.
- Our engine is 40 times faster than YAP.
- At first duplicate elimination was the most costly operation, but as the rows to process in each iteration decreased, the join became the most costly operation.

Transitive closure of a graph results



Same-Generation program

```
sg(X,Y) :- flat(X,Y).
sg(X,Y) :- up(X,X1), sg(X1,Y1), down(Y1,Y).
sg(a,Y)?
```

- Three tables are created with the following equations:

$$up = \{(a, b_i)|i\epsilon[1, n]\} \cup \{(b_i, c_j)|i, j\epsilon[1, n]\}. \quad (1)$$

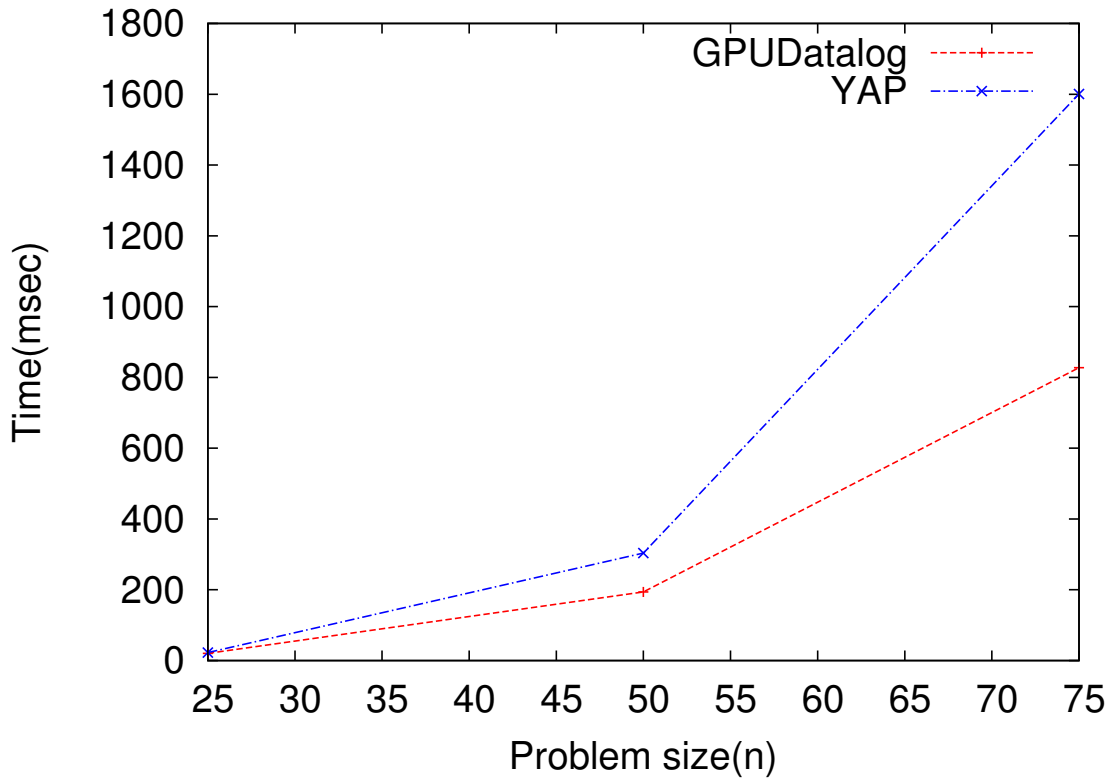
$$flat = \{(c_i, d_j)|i, j\epsilon[1, n]\}. \quad (2)$$

$$down = \{(d_i, e_j)|i, j\epsilon[1, n]\} \cup \{(e_i, f)|i\epsilon[1, n]\}. \quad (3)$$

- Very little gain in performance and our engine fails for $n > 90$ due to lack of memory.

Same-Generation program results.

- Duplicate elimination takes more than 80% of the total time and is the cause of the memory problem.
- The reason is that the rule creates too many tuples, but most are duplicates.



Related Work

- He *et. al* created GDB, a relational processing system for both CPUs and GPUs. GDB has primitive operations and the RA operators are built upon them.
- We modified the INLJ of GDB for our joins. We added multijoin and fused joins and projections.
- Diamos *et. al* developed relational operators for GPUs which partition and process data in blocks using algorithmic skeletons.
- Their join algorithm was compared to that of GDB, showing 1.69 performance improvement.

Conclusions

- Our engine performed very well but can be further improved by:
- Extended syntax to accept built-in predicates and negation.
- Optimisations based on tabling or magic sets methods.
- Mixed processing of rules both on the GPU and on the host multicore.
- Improved join operations to eliminate duplicates before writing final results.

Thank you