A Datalog Engine for GPUs

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

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

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

Abstract. We present the design and evaluation of a Datalog engine for execution in Graphics Processing Units (GPUs). The engine evaluates recursive and non-recursive Datalog queries using a bottom-up approach based on typical relational operators. It includes a memory management scheme that automatically swaps data between memory in the host platform (a multicore) and memory in the GPU in order to reduce the number of memory transfers. To evaluate the performance of the engine, four Datalog queries were run on the engine and on a single CPU in the multicore host. One query runs up to 200 times faster on the (GPU) engine than on the CPU.

Keywords: Logic programming · Datalog · Parallel computing · GPUs · Relational databases

1 Introduction

The traditional view of Datalog as a query language for deductive databases is changing as a result of the new applications where Datalog has been in use recently [18], including declarative networking [19], program analysis [9], information extraction [23] and security [20] — datalog recursive queries are at the core of these applications. This renewed interest in Datalog has in turn prompted new designs of Datalog targeting computing architectures such as GPUs, Field-programmable Gate Arrays (FPGAs) [18] and cloud computing based on Google's Mapreduce programming model [7]. This paper presents a Datalog engine for GPUs.

GPUs can substantially improve application performance and are thus now being used for general purpose computing in addition to game applications. GPUs are single-instruction-multiple-data (SIMD) [2] machines, particularly suitable for compute-intensive, highly parallel applications. They fit 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 corresponds to the modelling at different points in space.

Data-intensive, highly parallel applications such as database relational operations can also benefit from the SIMD model, substantially in many cases [11, 16, 17]. However, the communication-to-computation ratio must be relatively low for applications to show good performance, i.e.: the cost of moving data from host memory to GPU memory and vice versa must be low relative to the cost of the computation performed by the GPU on that data.

The Datalog engine presented here was designed considering various optimisations aimed to reduce the communication-to-computation ratio. Data is preprocessed in the host (a multicore) in order that: (i) data transfers between the host and the GPU take less time, and (ii) data can be processed more efficiently by the GPU. Also, a memory management scheme swaps data between host memory and GPU memory seeking to reduce the number of transfers.

Datalog queries, recursive and non-recursive, are evaluated using typical relational operators, *select, join* and *project*, which are also optimised in various ways in order to capitalise better on the GPU architecture.

Sections 2 and 3 present background material to the GPU architecture and the Datalog language. Section 4 presents the design and implementation of our Datalog Engine as a whole, and Sect. 5 of its relational operators. Section 6 presents an experimental evaluation of our Datalog engine. Section 7 presents related work. We conclude in Sect. 8.

2 GPU Architecture and Programming

GPUs are SIMD machines: they consist of many processing elements (PEs) that run the *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 *for* statements. However, a kernel is run in *bulk-synchronous parallelism* [28] by the GPU hardware, i.e.: each instruction within a kernel is executed across all PEs running the kernel. Thus, if a kernel compares strings, PEs that compare longer strings will take longer and the other PEs will wait for them.

Scheduling GPU work is usually as follows. A thread in the host platform (e.g., a multicore) first copies the data to be processed from host memory to GPU memory, and then invokes GPU threads to run the *kernel* to process the data. Each GPU thread has a unique id which is used by each thread to identify what part of the data set it will process. When all GPU threads finish their work, the GPU signals the host thread which will copy the results back from GPU memory to host memory and schedule new work.

GPU memory is organised hierarchically as shown in Fig. 1. Each (GPU) thread has its own *per-thread local* memory. Threads are grouped into *blocks*, each block having a memory *shared* by all threads in the block. Finally, thread blocks are grouped into a single *grid* to execute a kernel — different grids can be used to run different kernels. All grids share the *global memory*.

The global memory is the GPU "main memory". All data transfers between the host (CPU) and the GPU are made through reading and writing global memory. It is the slowest memory. A common technique to reducing the number

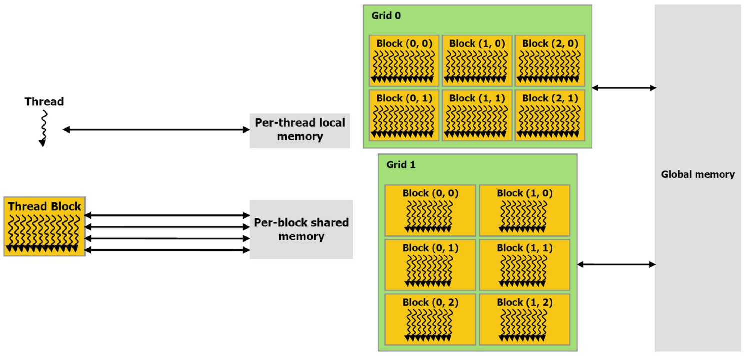

Fig. 1. GPU memory organization.

of global memory reads is *coalesced memory access*, which takes place when consecutive threads read consecutive memory locations allowing the hardware to coalesce the reads into a single one.

Nvidia GPUs are mostly programmed using the CUDA toolkit, a set of developing tools and a compiler that allow programmers to develop GPU applications using a version of the `C` language extended with keywords to specify GPU code. CUDA also includes various libraries with algorithms for GPUs such as the Thrust library [5] which resembles the C++ Standard Template Library (STL) [21]. We use the functions in this library to perform sorting, prefix sums [15] and duplicate elimination as their implementation is very efficient.

CUDA provides the following reserved words, each with three components `x`,`y` and `z`, to identify each thread and each block running a kernel: **threadIdx** is the index of a thread in its block; **blockIdx** is the index of a block in its grid; **blockDim** is the size of a block in number of threads; and **gridDim** is the size of a grid in number of blocks. With 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 GID = threadIdx.x + threadIdx.y * blockDim.x +
                   threadIdx.z * blockDim.x * blockDim.z;
```

`x`,`y` and `z` are initialised by CUDA according to the *shape* with which a kernel is invoked, either as a 1D *Vector* (`y=z=0`), a 2D *Matrix* (`z=0`), or a 3D *Volume*.

3 Datalog

As is well known, Datalog is a language based on first order logic that has been used as a data model for relational databases [26,27]. A Datalog program consists of *facts* about a subject of interest and *rules* to deduce new facts. Facts can be seen as rows in a relational database table, while rules can be used to specify complex queries. Datalog recursive rules facilitate specifying (querying for) the transitive closure of relations, which is a key concept to many applications [18].

3.1 Datalog Programs

A Datalog program consists of a finite number of facts and rules. Facts and rules are specified using atomic formulas, which consist of predicate symbols with arguments [26], e.g.:

```
FACTS                                 father relational table
                                      -----------------------
father(harry, john).                  harry    john
father(john, david).                  john     david
...                                   ...

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

Traditionally, names beginning with lower case letters are used for predicate names and constants, while names beginning with upper case letters are used for variables; numbers are considered constants. Facts consist of a single atomic formula, and their arguments are constants; facts that have the same name must also have the same arity. Rules consist of two or more atomic formulas with the first one from left to right, the rule *head*, separated from the other atomic formulas by the implication symbol ':-'; the other atomic formulas are *subgoals* separated by ',', which means a logical AND. We will refer to all the subgoals of a rule as the *body* of the rule. Rules, in order to be general, are specified with variables as arguments, but can also have constants.

3.2 Evaluation of Datalog Programs

Datalog programs can be evaluated through a top-down approach or a bottom-up approach. The top-down approach (used by Prolog) starts with the goal which is reduced to subgoals, or simpler problems, until a trivial problem is reached. It is tuple-oriented: each tuple is processed through the goal and subgoals using all relevant facts. It is not suitable for GPU bulk-synchronous parallelism (BSP) because the processing time of distinct tuples may vary significantly.

The bottom-up approach first applies the rules to the given facts, thereby deriving new facts, and repeating this process with the new facts until no more facts are derived. The query is considered only at the end, to select the facts matching the query. Based on relational operations (as described shortly), this approach is suitable for GPU BSP because such operations are set-oriented and relatively simple overall; hence show similar processing time for distinct tuples. Also, rules can be evaluated in any order. This approach can be improved using the magic sets transformation [8] or the subsumptive tabling transformation [25]. Basically, with these transformations the set of facts that can be inferred contains only facts that would be inferred during a top-down evaluation.

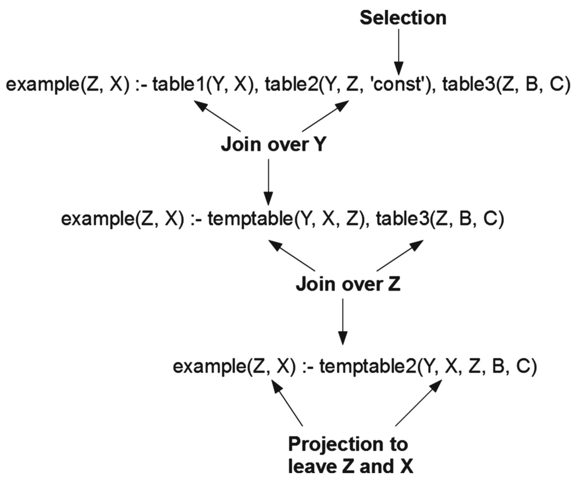
Selection
example(Z, X) :- table1(Y, X), table2(Y, Z, 'const'), table3(Z, B, C)
Join over Y
example(Z, X) :- temptable(Y, X, Z), table3(Z, B, C)
Join over Z
example(Z, X) :- temptable2(Y, X, Z, B, C)
Projection to leave Z and X

Fig. 2. Evaluation of a Datalog rule based on relational algebra operations.

3.3 Evaluation Based on Relational Algebra Operators

Evaluation of Datalog rules can be implemented using the typical relational algebra operators *select*, *join* and *projection*, as outlined in Fig. 2. *Selections* are made when constants appear in a rule body. Then a *join* is made between two or more subgoals in the rule body using the variables as reference. The result of a join becomes a temporary subgoal that must be joined to the other subgoals in the body. Finally, a *projection* is made over the variables in the rule head.

For recursive rules, fixed-point evaluation is used. The basic idea is to iterate through the rules in order to derive new facts, and using these new facts to derive even more new facts until no new facts are derived.

4 Our Datalog Engine for GPUs

This section presents the design of our Datalog engine for GPUs.

4.1 Architecture

Figure 3 shows the main components of our Datalog engine. There is a single *host* thread that runs in the host platform (a multi-core in our evaluation). The host thread schedules GPU work as outlined in Sect. 2, and also preprocesses the data to send to the GPU for efficiency, as described in Sect. 4.2.

The data sent to the GPU is organized into arrays that are stored in global memory. The results of rule evaluations are also stored in global memory.

Our Datalog (GPU) engine is organised into various GPU kernels. When evaluating rules, for each pair of subgoals in a rule, selection and selfjoin kernels are applied first in order to eliminate irrelevant tuples as soon as possible, followed

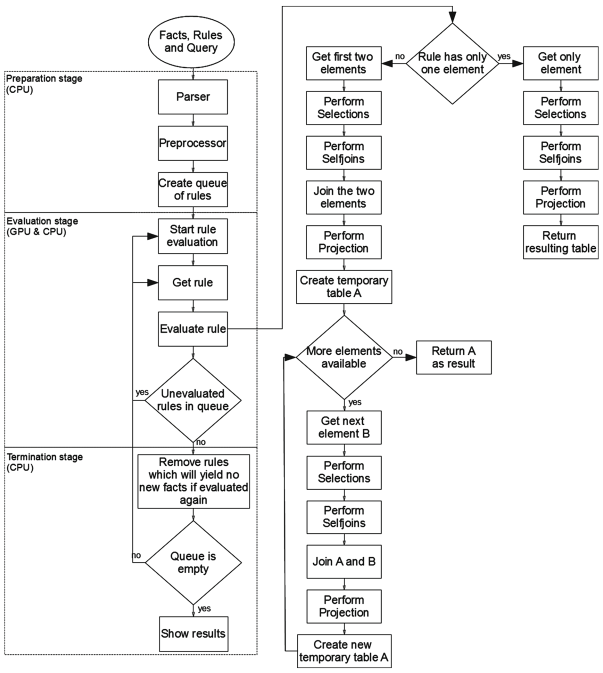

Fig. 3. GPU Datalog engine organisation.

by join and projection kernels. At the end of each rule evaluation, duplicate elimination kernels are applied. Figure 3 (on the right) shows these steps.

A memory management module helps identifying most recently used data by the GPU in order to keep it in global memory and to discard other data instead.

4.2 Host Thread Tasks

Parsing. To capitalise on the GPU capacity to process numbers and to have short and constant processing time for each tuple (the variable size of strings entails varying processing time), we identify and use facts and rules with/as 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 GPU thus works with numbers only; the dictionary is used at the very end when the final results are to be displayed.

Preprocessing. A key factor for good performance is preprocessing data before sending it to the GPU. As mentioned before, Datalog rules are evaluated through a series of relational algebra operations: selections, joins and projections. For the evaluation of each rule, the specification of what operations to perform, including constants, variables, facts and other rules involved, is carried out in the host (as opposed to be carried out in the GPU by each kernel thread), and sent to the GPU for all GPU threads to use. Examples:

- **Selection** is specified with two values, column number to search and the constant value to search; the two values are sent as an array which can include more than one selection (more than one pair of values), as in the following example, where columns 0, 2, and 5 will be searched for the constants a, b and c, respectively:

```
fact1('a',X,'b',Y,Z,'c'). -> [0, 'a', 2, 'b', 5, 'c']
```

- **Join** is specified with two values, column number in the first relation to join and column number in the second relation to join; the two values are sent as an array which can include more than one join, as in the following example where the following columns are joined in pairs: column 1 in fact1 (X) with column 1 in fact2, column 2 in fact1 (Y) with column 4 in fact2, and column 3 in fact1 (Z) with column 0 in fact2.

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

Other operations are specified similarly with arrays of numbers. These arrays are stored in GPU shared memory (as opposed to global memory) because they are small and the shared memory is faster.

4.3 Memory Management

Data transfers between GPU memory and host memory are costly in all CUDA applications [1]. We designed a memory management scheme that tries to minimize the number of such transfers. Its purpose is to maintain facts and rule results in GPU memory for as long as possible so that, if they are used more than once, they may often be reused from GPU memory. To do so, we keep track of GPU memory available and GPU memory used, and maintain a list with information about each fact and rule result that is resident in GPU memory. When data (facts or rule results) is requested to be loaded into GPU memory, it is first looked up in that list. If found, its entry in the list is moved to the beginning of the list; otherwise, memory is allocated for the data and a list entry is created at the beginning of the list for it. In either case, its address in 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 so doing, most recently used fact and rule results are kept in GPU memory.

5 GPU Relational Algebra Operators

This section presents the design decisions we made for the relational algebra operations we use in our Datalog engine: select, join and project operations for GPUs. The GPU kernels that implement these operations access (read/write) tables from GPU global memory.

5.1 Selection

Selection has two main issues when designed for running in GPUs. The first issue is that the size of the result is not known beforehand, and increasing the size of the results buffer is not convenient performance-wise because it may involve reallocating its contents. The other issue is that, for efficiency, each GPU thread must know onto which global memory location it will write its result without communicating with other GPU threads.

To avoid those issues, our selection uses three different kernel executions. The first kernel marks all the rows that satisfy the selection predicate 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.

5.2 Projection

Projection requires little computation, as it simply involves taking all the elements of each required column and storing them 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 CPUs, and the fact that the results remain in GPU memory for further processing, make projection a suitable operation for GPU processing.

5.3 Join

Our Datalog engine uses these types of join: Single join, Multijoin and Selfjoin. A single join is used when only two columns are to be joined, e.g.: $table_1(X,Y) \bowtie table_2(Y,Z)$. A multijoin is used when more than two columns are to be joined: $table_1(X,Y) \bowtie table_2(X,Y)$. A selfjoin is used when two columns have the same variable in the same predicate: $table_1(X,X)$.

Single join. We use a modified version of the Indexed Nested Loop Join described in [16], which is as follows:

```
Make an array for each of the two columns to be joined
Sort one of them
Create a CSS-Tree for the sorted column
Search the tree to determine the join positions
Do a first join~to determine the size of the result
Do a second join~to write the result
```

The CSS-Tree [22] (Cache Sensitive Search Tree) is very adequate for GPUs 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, so we must perform a “preliminary” join. This 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 write locations when a prefix sum is applied to it. With the size and write locations known, a second join writes the results.

Multijoin. To perform a join over more than two columns, e.g., $table_1(X, Y) \bowtie table_2(X, Y)$, first we take a pair of columns, say (X, X), to create and search on the CSS-Tree as described in the single join algorithm. Then, as we are doing the first join, we also check if the values of the remaining columns are equal (in our example we check if $Y = Y$) and discard the rows that do not comply.

Selfjoin. The selfjoin operation is similar to the selection operation. The main difference is that, instead of checking for a constant value on the corresponding row, it checks if the values of the columns involved by the self join match.

5.4 Optimisations

Our relational algebra operations make use of the following optimisations in order to improve performance. The purpose of these optimisations is to reduce memory use and in principle processing time — the cost of the optimisations themselves is not yet evaluated.

Duplicate Elimination. Duplicate elimination uses the *unique* function of the Thrust library. It takes an array and a function to compare two elements in the array, and returns the same array with the unique elements at the beginning. We apply duplicate elimination to the result of each rule: when a rule is finished, its result is sorted and the *unique* function is applied.

Optimising Projections. Running a *projection* at the end of each join, as described below, allows us to discard unnecessary columns earlier in the computation of a rule. For example, consider the following rule:

```
rule1(Y, W) :- fact1(X, Y), fact2(Y, Z), fact3(Z,W).
```

The evaluation of the first join, $fact_1 \bowtie_Y fact_2$, generates a temporary table with columns (X, Y, Y, Z), not all of which are necessary. One of the two Y columns can be discarded; and column X can also be discarded because it is not used again in the body nor in the head of the rule.

Fusing Operations. Fusing operations consists of applying two or more operations to a data set in a single read of the data set, as opposed to applying only one operation, which involves as many reads of the data set as the number of operations to be applied. We fuse the following operations.

- All selections required by constant arguments in a subgoal of a rule are performed at the same time.
- All selfjoins are also performed at the same time.

– Join and projection are always performed together at the same time.

To illustrate these fusings consider the following rule:

```
rule1(X,Z):- fact1(X,'const1',Y,'const2'),fact2(Y,'const3',Y,Z,Z).
```

This rule will be evaluated as follows. $fact_1$ is processed first: the selections required by $const_1$ and $const_2$ are performed at the same time — $fact_1$ does not require selfjoins. $fact_2$ is processed second: (a) the selection required by $const_3$ is performed, and then (b) the selfjoins between Ys and Zs are performed at the same time. Finally, a join is performed between the third column of $fact_1$ and the first column of $fact_2$ and, at the same time, a projection is made (as required by the arguments in the rule head) to leave only the first column of $fact_1$ and the fourth column of $fact_2$.

6 Experimental Evaluation

This section describes our platform, applications and experiments to evaluate the performance of our Datalog engine. We are at this stage interested in the performance benefit of using GPUs for evaluating Datalog queries, compared to using a CPU only. Hence we present results that show the performance of 4 Datalog queries running on our engine compared to the performance of the same queries running on a single CPU in the host platform. (We plan to compare our Datalog engine to similar GPU work discussed in Sect. 7, Related Work, in another paper).

On a single CPU in the host platform, the 4 queries were run with the Prolog systems YAP [10] and XSB [24], and the Datalog system from the MITRE Corporation [3]. As the 4 queries showed the best performance with YAP, our results show the performance of the queries with YAP and with our Datalog engine only. YAP is a high-performance Prolog compiler developed at LIACC/Universidade do Porto and at COPPE Sistemas/UFRJ. Its Prolog engine is based on the WAM (Warren Abstract Machine) [10], extended with some optimizations to improve performance. The queries were run on this platform:

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.

For each query, in each subsection below, we describe first the query, and then discuss the results. Our results show the evaluation of each query once all data has been preprocessed and in CPU memory, i.e.: I/O, parsing and preprocessing costs are not included in the evaluation.

6.1 Join over Four Big Tables

Four tables, all with the same number of rows filled with random numbers, are joined together to test all the different operations of our Datalog engine. The rule and query used are:

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

Figure 4 shows the performance of the join with YAP and our engine, in both normal and logarithmic scales to better appreciate details. Our engine is clearly faster, roughly 200 times. Both YAP and our engine take proportionally more time as the size of tables grows. Our engine took just above two seconds to process tables with five million rows each, while YAP took about two minutes to process tables with one million rows each. Joins were the most costly operations with multijoin alone taking more than 70 % of the total time; duplicate elimination and sorting were also time consuming but within acceptable values; prefix sums and selections were the fastest operations.

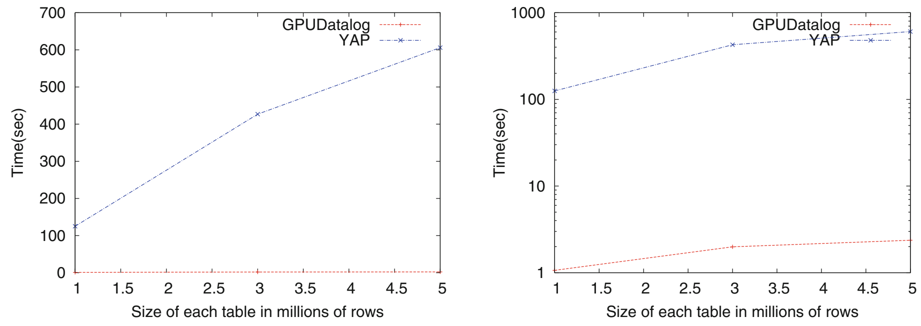

Fig. 4. Performance of join over four big tables (NB: log. scale on the right).

6.2 Transitive Closure of a Graph

The transitive closure of a graph (TCG) is a recursive query. We use a table with two columns filled with random numbers that represent the edges of a graph [13]. The idea is to find all the nodes that can be reached if we start from a particular node. This query is very demanding because recursive queries involve various iterations over the relational operations that solve the query. The rules and the query are:

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

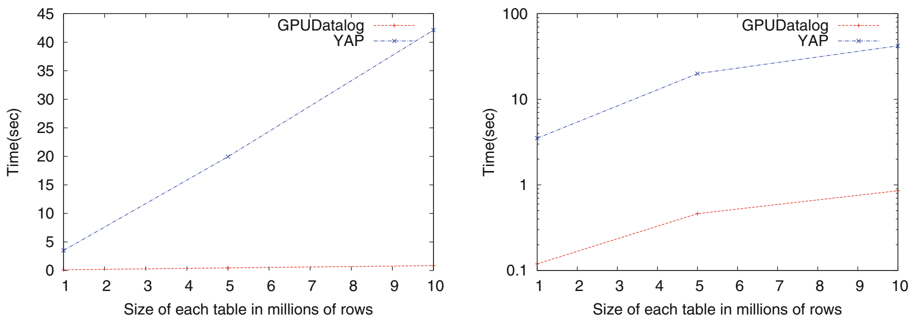

Fig. 5. Performance of transitive closure of a graph (NB: log. scale on the right).

Figure 5 shows the performance of TCG with YAP and our engine. Similar observations can be made as for the previous experiment. Our engine is 40x times faster than YAP for TCG. Our engine took less than a second to process a table of 10 million rows while YAP took 3.5 s to process 1 million rows.

For the first few iterations, duplicate elimination was the most costly operation of each iteration, and the join second but closely. As the number of rows to process in each iteration decreased, the join became by far the most costly operation.

6.3 Same-Generation Program

This is a well-known program in the Datalog literature, and there are various versions. We use the version described in [6]. Because of the initial tables and the way the rules are written, it generates lots of new tuples in each iteration. The three required tables are created with the following equations:

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

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

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

Where a and f are two known numbers and b, c, d and e are series of n random numbers. The rules and query are as follows:

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

The results show (Fig. 6) very little gain in performance, with our engine taking an average of 827 ms and YAP 1600 ms for $n = 75$. Furthermore, our engine cannot process this application for n > 90 due to lack of memory.

The analysis of each operation revealed that duplicate elimination takes more than 80 % of the total time and is also the cause of the memory problem. The

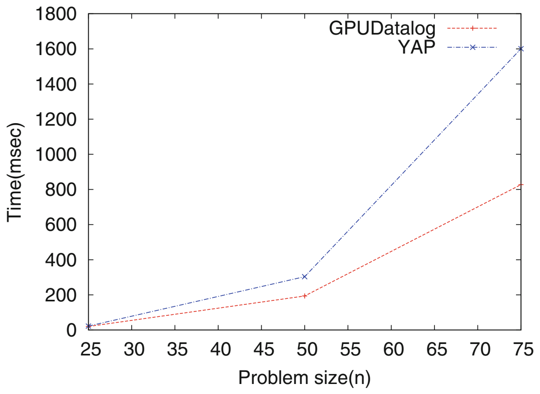

Fig. 6. Same-generation program.

reason of this behaviour is that the join creates far too many new tuples, but most of these tuples are duplicates (as an example, for $n = 75$ the first join creates some 30 million rows and, after duplicate elimination, less than 10 thousand rows remain).

6.4 Tumour Detection

Correctly determining whether or not a tumour is malignant requires analysing and comparing a great amount of information from medical studies. Considering each characteristic of a tumour as a fact, the rules and query below can be used to determine, for each patient, if his/her tumour is malignant or not:

```
is_malignant(A):-
   same_study(A,B),                  'HO_BreastCA'(B,hxDCorLC),
   'MassPAO'(B,present),             'ArchDistortion'(A,notPresent),
   'Calc_Round'(A,notPresent),       'Sp_AsymmetricDensity'(A,notPresent),
   'SkinRetraction'(B,notPresent), 'Calc_Popcorn'(A,notPresent),
   'FH_DCNOS'(B,none).
same_study(Id,OldId) :-
   'IDnum'(Id,X),                    'MammoStudyDate'(Id,D0),
   'IDnum'(OldId,X),                 'MammoStudyDate'(OldId,D0),
   OldId \= Id.
is_malignant(A)?
```

The query asks for those studies which detect a malignant tumour. Some tumour characteristics are taken from the most recent study, while others must be taken from past studies. This restriction requires defining an additional rule (`same_study`) to determine if two studies belong to the same person and if they have different dates. The last subgoal of `same_study` (`OldId \= Id`) prevents a study from referencing to itself, thus avoiding incorrect results. We evaluated this program with 65800 studies, i.e., each table is composed of 65800 rows.

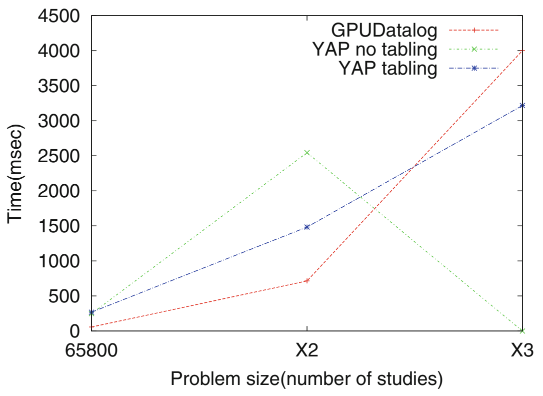

Fig. 7. Performance of tumour detection.

Figure 7 shows the performance of tumour detection with YAP and our engine. We used different sizes of input data through duplicating and triplicating each table, i.e.: each table had 65800 rows for the first test, and 131600 and 197400 rows for the second and third tests. We could thus increase processing time while obtaining the same results thanks to duplicate elimination.

Our engine performs best for the first and second tests, but is surpassed in the third test by YAP with tabling [25]. A detailed analysis showed that the multijoin required by `same_study` consumed almost 90 % of the total execution time, and mostly in duplicate elimination, as follows. In the third test, `same_study` generated 4095036 rows, which were reduced to 50556 after duplicate elimination. For `is_malignant` the results were similar: 4881384 rows were generated but only 550 remained after duplicate elimination.

YAP with no tabling in the third test is so affected by duplicates that it simply terminates after throwing an error — shown in the figure as zero execution time. In contrast, YAP with tabling avoids performing duplicate work and thus performs rather well.

7 Related Work

He *et al.* [17] have designed, implemented and evaluated GDB, an in-memory relational query coprocessing 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 Indexed Nested Loop Join (INLJ) of GDB for our single join and multijoin, so that more than two columns can be joined, and a projection performed, at the same time. Their selection operation and ours are similar too; ours 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, while they do not use fusion at all.

Diamos *et al.* [11,12,29–31] have also developed relational operators for GPUs for the Red Fox [4] platform, an extended Datalog developed by LogicBlox [14] for multiple-GPU systems [31]. Their operators partition and process data in blocks using algorithmic skeletons. Their join algorithm is 1.69 times faster than that of GDB [11]. 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 [30]. We applied fusion (e.g., simultaneous selections, selection then join, etc.) at source code; they implemented it automatically through the compiler. Kernel fission, the parallel execution of kernels and memory transfers, is not yet adopted in our work. We plan to compare our relational operators to those of GDB and Red Fox, and extend them to run on multiple-GPU systems too.

8 Conclusions

Our Datalog engine for GPUs evaluates queries based on the relational operators select, join and projection. Our evaluation using 4 queries shows a dramatic performance improvement for two queries, up to 200 times for one of them. The other two queries did not perform that well, but we are working on the following extensions to our engine in order to improve its performance further.

- Evaluation based on tabling [25] or magic sets [8] methods.
- Managing tables larger than the total amount of GPU memory.
- Mixed processing of rules both on the GPU and on the host multicore.
- Improved join operations to eliminate duplicates before writing final results.
- Extended syntax to accept built-in predicates and negation [6].

Acknowledgments. CMA thanks the support during his MSc studies from: the University of Porto, the Centre for Research and Postgraduate Studies of the National Polytechnic Institute (CINVESTAV-IPN) of Mexico, and the Mexican Council of Science and Technology (CONACyT). ICD and VSC were partially supported by: the European Regional Development Fund (ERDF), COMPETE Programme; the Portuguese Foundation for Science and Technology (FCT), project LEAP FCOMP-01-0124-FEDER-015008; and project ABLe PTDC/EEI-SII/2094/2012.

References

1. CUDA, C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
2. CUDA, C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
3. Datalog by the MITRE Corporation. http://datalog.sourceforge.net/
4. Red Fox: A Compilation Environment for Data Warehousing. http://gpuocelot.gatech.edu/projects/red-fox-a-compilation-environment-for-data-warehousing/

5. Thrust: A Parallel Template Library. http://thrust.github.io/
6. Abiteboul, S., et al.: Foundations of Databases. Addison-Wesley, Boston (1995)
7. Afrati, F.N., Borkar, V., Carey, M., Polyzotis, N., Ullman, J.D.: Cluster Computing, Recursion and Datalog. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 120–144. Springer, Heidelberg (2011)
8. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program. **10**(3–4), 255–299 (1991)
9. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA, pp. 243–262 (2009)
10. Costa, V.S., et al.: The YAP prolog system. TPLP **12**(1–2), 5–34 (2012)
11. Diamos, G., et al.: Efficient relational algebra algorithms and data structures for GPU. Technical report, Georgia Institute of Technology (2012)
12. Diamos G. et al.: Relational algorithms for multi-bulk-synchronous processors. In: 18th Symposium on Principles and Practice of Parallel Programming (2013)
13. Dong, G., Jianwen, S., Topor, R.W.: Nonrecursive incremental evaluation of datalog queries. Ann. Math. Artif. Intell. **14**(2–4), 187–223 (1995)
14. Green, T.J., Aref, M., Karvounarakis, G.: LogicBlox, Platform and Language: A Tutorial. In: Barceló, P., Pichler, R. (eds.) Datalog 2.0 2012. LNCS, vol. 7494, pp. 1–8. Springer, Heidelberg (2012)
15. Harris, M., et al.: Parallel prefix sum (scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, pp. 851–876. Addison Wesley, Boston (2007)
16. He, B., et al.: Relational joins on graphics processors. In: SIGMOD Conference, pp. 511–524 (2008)
17. He, B., et al.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. (TODS) **34**(4), 21:1–21:39 (2009)
18. Huang, S.S., et al.: Datalog and emerging applications: an interactive tutorial. In: SIGMOD Conference. pp. 1213–1216 (2011)
19. Loo, B.T., et al.: Declarative networking: language, execution and optimization. In: SIGMOD Conference, pp. 97–108 (2006)
20. Marczak W.R., et al.: Secureblox: customizable secure distributed data processing. In: SIGMOD Conference, pp. 723–734 (2010)
21. Musser, D.R., Derge, G.J., Saini, A.: STL Tutorial and Reference Guide: C++ Programming With The Standard Template Library, 2nd edn. Addison-Wesley Longman Publishing Co. Inc., Boston (2001)
22. Rao, J., Ross, K.A.: Cache conscious indexing for decision-support in main memory. In: 25th VLDB Conference, San Francisco., CA, USA, pp. 78–89 (1999)
23. Shen, W., et al.: Declarative information extraction using datalog with embedded extraction predicates. In: VLDB, pp. 1033–1044 (2007)
24. Swift, T., Warren, D.S.: Xsb: Extending prolog with tabled logic programming. TPLP **12**(1–2), 157–187 (2012)
25. Tekle, K.T., Liu, Y.A.: More efficient datalog queries: subsumptive tabling beats magic sets. In: SIGMOD Conference, pp. 661–672 (2011)
26. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. 1. Computer Science Press, Beijing (1988)
27. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. 2. Computer Science Press, Beijing (1989)
28. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)

29. Wu, H., et al.: Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In: 45th International Symposium on Microarchitecture (2012)
30. Wu, H., et al.: Optimizing data warehousing applications for GPUs using kernel fusion/fission. In: IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum (2012)
31. Young, J., et al.: Satisfying data-intensive queries using GPU clusters. In: 2nd Annual Workshop on High-Performance Computing meets Databases (2012)