Centro de Investigación y de Estudios Avanzados

del Instituto Politécnico Nacional

UNIDAD ZACATENCO

DEPARTAMENTO DE COMPUTACIÓN

Datalog para GPUs

Tesis que presenta

Carlos Alberto Martínez Angeles

para obtener el Grado de

Maestro en Ciencias en Computación

Director de tesis:

Dr. Jorge Buenabad Chávez

México, DF Octubre del 2013

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

ZACATENCO CAMPUS

COMPUTER SCIENCE DEPARTMENT

Datalog for GPUs

Submitted by

Carlos Alberto Martínez Angeles

as fulfillment of the requirement for the degree of

Master in Computer Science

Advisor:

Dr. Jorge Buenabad Chávez

Mexico, DF **October 2013**

Resumen

Datalog es un lenguaje basado en lógica de primer orden que fue desarrollado en los 80s como modelo de datos para bases de datos relacionales. Recientemente, ha sido utilizado en nuevas áreas de aplicación, por lo que se han hecho propuestas para ejecutar Datalog en nuevas plataformas tales como Unidades de Procesamiento Gráfico (GPUs en inglés) y MapReduce. En ese entonces como hoy en día, el interés en Datalog es el resultado de su habilidad para calcular el cierre transitivo de relaciones por medio de consultas recursivas que, en efecto, transforman las bases de datos relacionales en bases de datos deductivas o bases de conocimiento.

El tema de esta tesis es el diseño, implementación y evaluación de un motor paralelo del lenguaje Datalog para GPUs. A nuestro conocimiento, es el primer motor totalmente funcional de Datalog para GPUs. Consiste en: i) un compilador que traduce los programas de Datalog en operadores de álgebra relacional (selección, varios tipos de uniones y proyección); ii) un planificador que prepara y manda ejecutar estas operaciones en la GPU desde la plataforma anfitrión; iii) los algoritmos paralelos de dichas operaciones; y iv) un esquema de manejo de memoria que tiende a reducir el numero de transferencias de memoria entre el anfitrión y la GPU. También incluye varias optimizaciones que aprovechan las características del lenguaje Datalog y la arquitectura de las GPUs.

Nuestro motor de Datalog fue desarrollado en C utilizando la plataforma de software de Nvidia CUDA. La evaluación de nuestro motor utilizando varias consultas muestra un importante incremento en el rendimiento al compararla contra XSB y YAP, famosos motores de Prolog, y el motor de Datalog de la corporación Mitre. Para dos de las consultas, se obtuvo un incremento en el rendimiento de hasta 200 veces.

Abstract

Datalog is a language based on first order logic that was investigated as a data model for relational databases in the 1980s. It has recently been used in various new application areas, prompting proposals to run Datalog programs on new platforms such as Graphics Processing Units (GPUs) and MapReduce. Back then and nowadays, interest in Datalog has stemmed from its ability to compute the transitive closure of relations through recursive queries which, in effect, turns relational databases into deductive databases, or knowledge bases.

This thesis presents the design, implementation and evaluation of a Datalog engine for GPUs. It is the first fully functional Datalog engine for GPUs to the best of our knowledge. It consists of: i) a compiler that translates Datalog programs into relational algebra operations (select, various types of joins and project); ii) a scheduler that plans and launches such operations into the GPU from the host platform; iii) the GPU parallel algorithms of such operations; and iv) a memory management scheme that tends to reduce the number of memory transfers between the host and the GPU. It also includes various optimisations that capitalise on the characteristics of the Datalog language and the GPU architecture.

Our Datalog engine was developed in C with the Nvidia CUDA software platform. The evaluation of our engine using several queries shows a dramatic performance improvement when compared against the well known Prolog engines XSB and YAP, and the Datalog engine from Mitre Corporation. For two of the queries, a performance increase of up to 200 times was achieved.

Acknowledgements

I would like to express my very great appreciation to Dr. Jorge Buenabad Chávez for his valuable and constructive suggestions during the planning and development of this research work. The opportunities he provided me to travel and meet other researchers, and his willingness to give his time so generously has been very much appreciated.

I would also like to express my deep gratitude to Dr. Inês Dutra and Dr. Vítor Santos Costa, for being my surrogate family during the time I stayed in Porto and for their continued support there after. This work would not have been possible without their patient guidance, encouragement and useful critiques.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The subject of this thesis is the design, implementation and evaluation of a Datalog engine for Graphics Processing Units (GPUs). The specific objectives were:

- The design, implementation and evaluation of a new Datalog engine for GPUs, capable of evaluating standard Datalog programs faster than any other CPU Datalog engine.
- A compiler of Datalog programs that translates facts, rules and queries into numbers, which are easier to work with in GPUs due to their constant processing time (strings entail variable processing time due to their variable size).
- A memory management module that maintains data in GPU memory for as long as possible in order to reduce data transfers between CPU and GPU.
- Relational algebra algorithms tuned to exploit the architecture of the GPU thanks to the use of techniques like CSS-Trees, coalesced memory access, etc. They also capitalise on the distinctive features of Datalog rule evaluation which allows the use of simultaneous projections, operation fusing, etc.
- Publication of an international conference paper describing these contributions.

Datalog is a language based on first order logic that was widely investigated as a data model for relational databases [1, 2]. A Datalog program consist of a finite number of facts and rules. 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, for example 'If X is the father of Y and if Y is the father of Z, then X is the grandfather of Z'. To represent rules and facts, Datalog uses clauses. The left side of the clause is the head and the right size is the body, which can be empty. Clauses without body are facts; clauses with at least one element in the body are rules. For example:

```
edge(1,2).                                         <-- Fact
edge(2,3).                                         <-- Fact
path(X,Y) :- edge(X,Y).                            <-- Rule
path(X,Z) :- edge(X,Y), path(Y,Z).                 <-- Rule
```

Datalog programs can derive many new facts. Sometimes, only a subset of these facts is of importance. To derive only the necessary fact subsets from the rules, a query is used. Queries are specified as a single clause followed by a question mark. For example:

```
path(1,Y)?
```

Datalog can also use recursive rules which facilitate specifying (querying for) the transitive closure of relations, which is a key concept to many new applications including data integration [3, 4], declarative networking [5, 6], program analysis [7], information extraction [8, 9], network monitoring [10], security [11, 12], and cloud computing [13]. With these new applications, Datalog has gained much renewed interest, which include proposals to make use of GPUs and clusters composed of multicore-GPU nodes in order to make it more efficient in terms of performance. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput [14]. GPUs were designed for computer graphics and could only be programmed through 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.

CUDA (Compute Unified Device Architecture) is a software platform and programming model created by NVIDIA [15] to make use of the GPUs as a highly parallel general-purpose machine. CUDA extends C, C++ and Fortran [16] (other languages are supported but are not part of the standard) with its own functions and reserved words. It allows the definition of functions, called *kernels*, that are executed in parallel by CUDA threads.

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 CPU memory to GPU memory, and then invokes GPU threads to run the *kernel* to process the data. Each GPU thread has an 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 from GPU memory to host memory and schedule new work.

GPUs can profitably be used to evaluate Datalog programs both because Datalog programs can be expressed with the relational algebra operators *selection*, *join* and *projection*, and because these operators are suitable for computations using massive parallelism.

Selections are made when constants appear in the body of a rule. Then a *join* is made between two or more elements in the body of a rule using the variables as reference. The result of the join is then joined to other rest of the elements in the body, if any, an so on. Finally, a *projection* is made of the variables in the head of the rule. Figure 1.1 shows an example of the necessary operations to evaluate certain rule.

The approach to evaluate Datalog programs implemented in this thesis work is referred to as bottom-up. It consists, conceptually, in applying the rules to the given facts, thereby 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

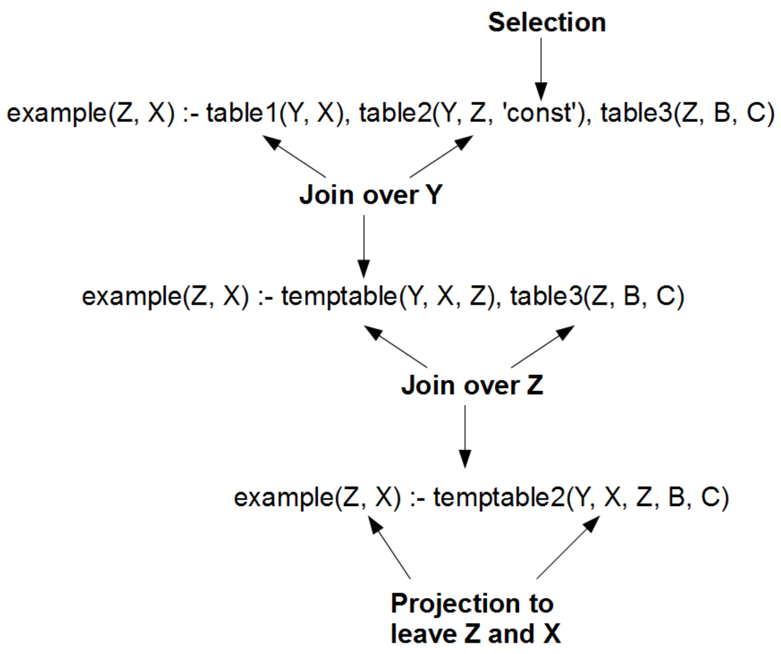
Selection
example(Z, X) :- table1(Y, X), table2(Y, Z, 'const'), table3(Z, B, C)
Join over Y
example(Z, X) :- temptable(Y, X, Z), table3(Z, B, C)
Join over Z
example(Z, X) :- temptable2(Y, X, Z, B, C)
Projection to leave Z and X

Figure 1.1: Rule evaluation.

the facts matching the query are selected. The benefits of this approach is that rules can be evaluated in any order and, as stated above, in a highly parallel manner based on equivalent relational operations. (This thesis also describes other approaches and techniques to evaluate Datalog programs and how to combine them in order to improve performance).

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

Our Datalog engine processes Datalog programs as follows:

Compiling. Datalog programs are compiled using Flex [17] and Bison [18]. To capitalise on the GPU capacity to process numbers and to have short and constant processing time for each tuple (strings variable size entails varying processing time), we identify and use facts and rules with/as numbers, keeping their corresponding strings in a hashed dictionary.

Preprocessing. Preprocessing data before sending it to GPU is a key factor for good performance. The most common form of preprocessing is the elimination of redundant calculations in GPU threads. The preprocessing module analyses the rules to determine both which relational operations to perform and on which columns should they be performed.

Evaluation. The required relational algebra operators were implemented for the GPU in the following way:

- **Selection.** Searches for constant values in determined columns, discarding the rows that do not have these values. Uses three different kernel executions. The first kernel marks all the rows that satisfy the selection predicate. The second kernel performs a prefix sum [19] 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.** Simply involves taking all the elements of each required column and store them in a new memory location. While it may seem pointless to use the GPU to move data items, 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.

- **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.: table1(X,Y) $\bowtie$ table2(Y,Z). A multijoin is used when more than two columns are to be joined: table1(X,Y) $\bowtie$ table2(X,Y). A selfjoin is used when two columns have the same variable in the same predicate: table1(X,X). The first two joins create and search for elements to join on a tree specially designed for GPUs. The Selfjoin is very similar to the selection operation, the main difference is that instead of checking a constant value, it checks if the values of the columns affected by the self join match.

To improve the performance of our engine, several optimizations were made:

- Additional projections are made to discard unnecessary columns earlier in the computation.
- Some 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. This is called fusion [20] and reduces the overall number of reads to data sets.
- Data transfers between GPU memory and host memory are costly. 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.

We tested our engine with computation intensive logic programming problems against well known Prolog and Datalog CPU engines like XSB [21] and YAP [22]. With all problems, our engine showed the best results, with a performance increase of up to 200x.

There is related work both on GPUs and Datalog. Regarding relational algebra operators on GPUs, the core operators of our engine, Bingsheng He et al. proposed GPUQP [23], an in-memory query co-processor focused on fully exploiting the architectural features of the GPUs. Also, Gregory Diamos et al. are working on Red Fox [24], an upcoming compilation and runtime environment for data warehousing applications on GPU clusters using an extended Datalog developed by LogicBlox [25].

1.1 Thesis layout

The thesis layout is as follows:

Chapter 2 describes the architecture of the GPUs, its programming model based on CUDA and the most important optimizations required for good performance in all GPU applications.

Chapter 3 describes the syntax of Datalog programs, its equivalence to relational algebra operations and the different approaches for their efficient evaluation.

Chapter 4 presents how we developed and optimized our Datalog engine, and which solutions we applied to the issues encountered.

Chapter 5 shows our experimental platform. It describes the hardware we used, the results we obtained with some common logic programming problems and compares the performance of our engine against other well known Prolog and Datalog engines.

Chapter 6 presents our conclusions and the ideas we have to further improve our Datalog engine.

Chapter 2

GPUs

Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput [14]. 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 [26], 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 work were Nvidia GPUs [27], so all future mention of GPUs refer to those of this particular brand. The examples and images used in this chapter were taken from [16].

This chapter presents an overview of the GPU architecture, its programming model and interface, and programming guidelines for good performance.

2.1 GPU Architecture

GPUs are 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. In contrast, an SPMD (single-program-multiple-data) program is synchronised through message passing and/or shared memory synchronisation primitives specified by the programmer.

GPUs usually have hundreds of processing units called CUDA cores, as shown in Figure 2.1, which execute one thread each. A CUDA core has the following elements:

- Floating point unit compliant with IEEE floating-point standard.
- Integer unit.

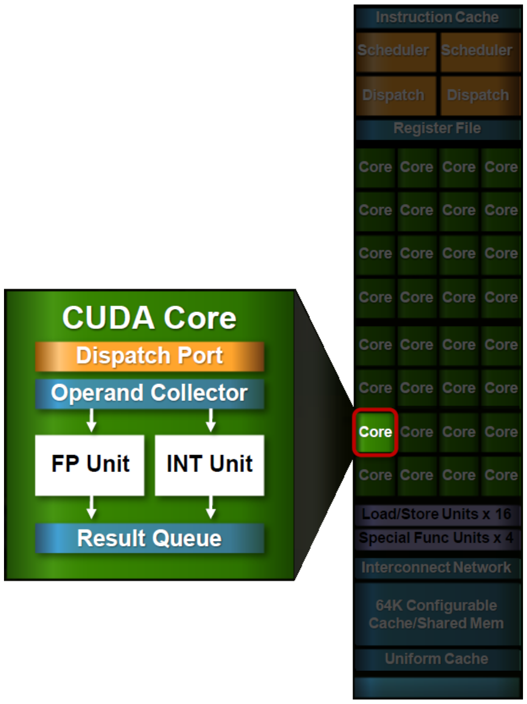

Figure 2.1: CUDA Core

- Logic unit.
- Move, compare unit.
- Branch unit.

CUDA cores are arranged in special hardware units called Streaming Multiprocessors(SM), each with 32 CUDA cores (low-end or old GPUs have 16 CUDA cores per SM). An SM schedules threads to be executed in *warps* of size equal to the number of CUDA cores it has (*warp size*). As shown in Figure 2.2, 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.

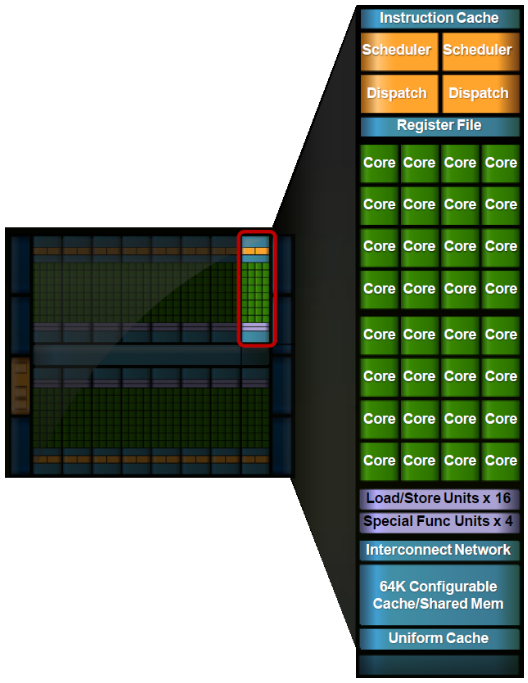

Figure 2.2: Streaming Multiprocessor

- 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:

- **Kepler.** The latest architecture; major revision number is 3.
- **Fermi.** The most widespread architecture; major revision number is 2.
- **Tesla.** 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.

2.1.1 CUDA

CUDA (Compute Unified Device Architecture) is a software platform and programming model created by Nvidia [15]. With CUDA, the GPU becomes a highly parallel general-purpose machine.

CUDA is an extension to the programming languages C, C++ and Fortran [28] (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 [29], a library of parallel algorithms and data structures based on the Standard Template Library(STL) library [30].

The current version of the CUDA SDK (5.5) is available for Microsoft Windows, Linux and Mac OS through the NVIDIA Developer Zone website [31]. CUDA works with all modern Nvidia GPUs. Programs developed for a particular GPU should also work on all GPUs of the same or better arquitectures without modifying the source code.

2.2 Programming model

This section describes the CUDA programming model for C, known as CUDA C [16]. The models for other languages are similar. We will refer to CUDA C as CUDA from now on.

Figure 2.3 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.

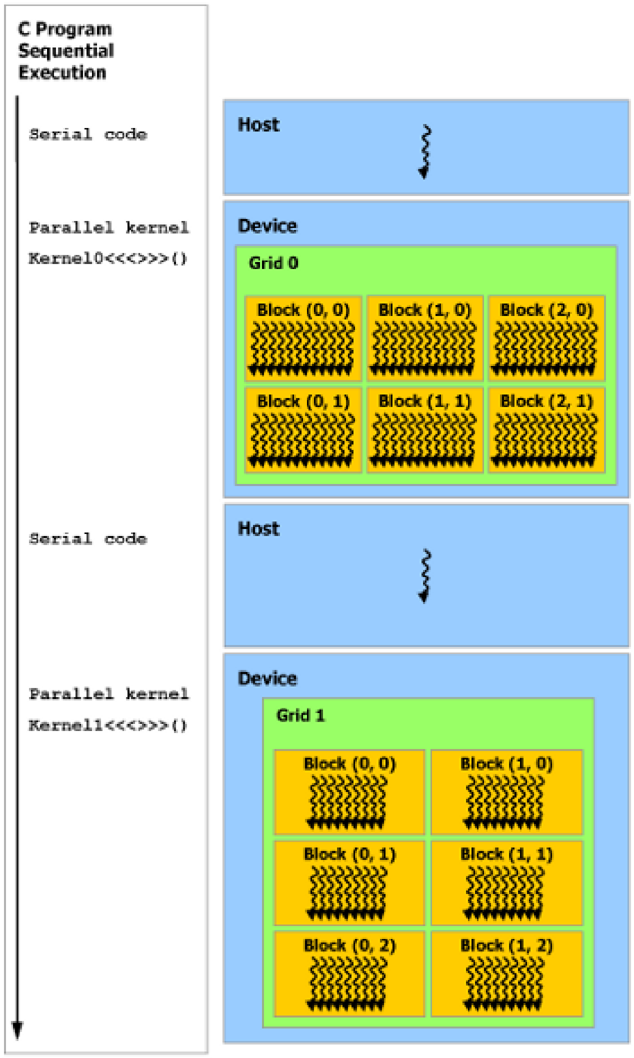

Figure 2.3: Heterogeneous Programming Model

2.2.1 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.

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.

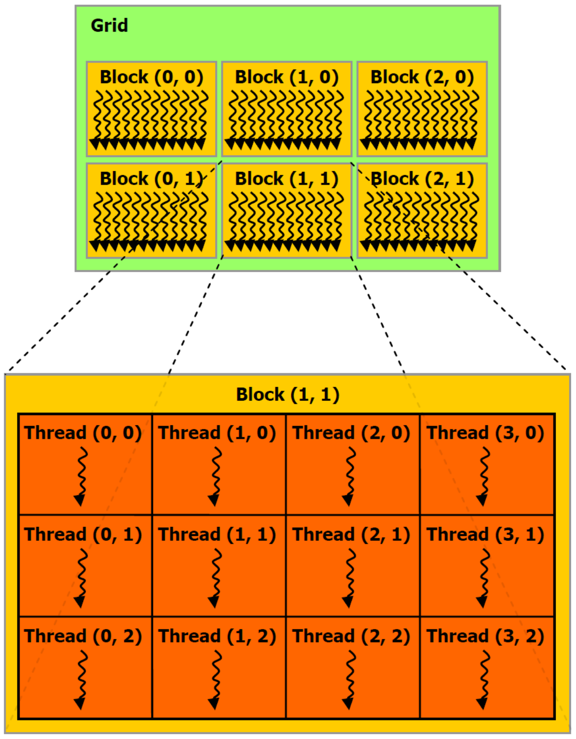

Figure 2.4: Thread Hierarchy

2.2.2 Thread Hierarchy

Threads are organized into *blocks*, and blocks into a *grid* as shown in Figure 2.4.

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).

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.

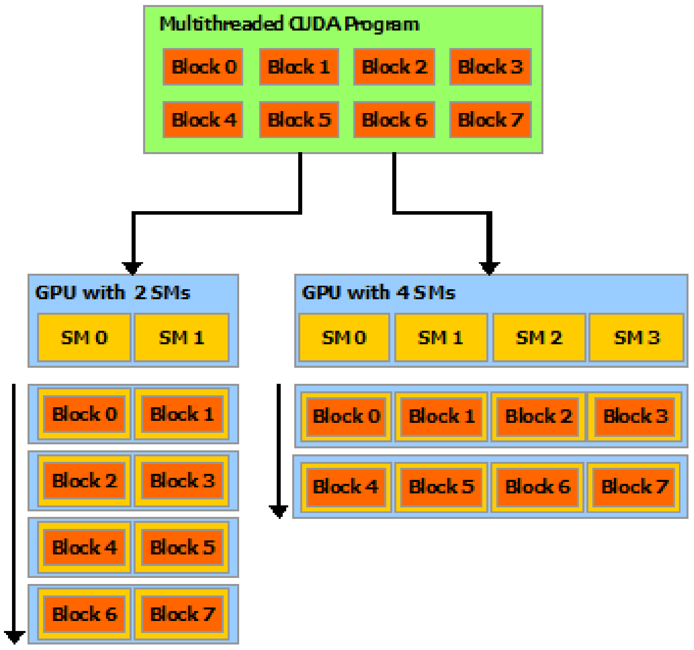

Figure 2.5: Automatic Scalability

However, recall that each Streaming Multiprocessor has a certain number of CUDA cores (usually 32), 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 2.5, 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.

2.2.3 Memory Hierarchy

CUDA threads have access to different memory types as shown in Figure 2.6. 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*.

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:

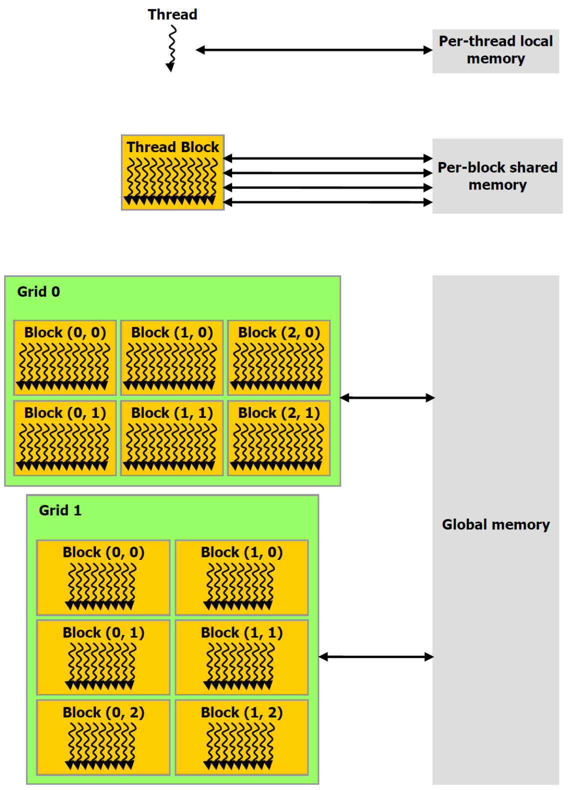

Figure 2.6: Memory Hierarchy

```
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.

2.3 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.

2.3.1 Compilation with nvcc

Nvcc is a compiler that simplifies the process of compiling CUDA code [32]. It uses command line options similar to those of GCC [33] 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 [34].
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

2.3.2 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.

2.3.3 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);
```

2.3.4 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);
```

2.3.5 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.

2.3.6 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.

2.4 Performance Guidelines

To maximize GPU performance, the CUDA Best Practices Guide [35] 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.

2.4.1 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.

2.4.2 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*.
- 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 the same all threads in a warp accesses consecutive global memory

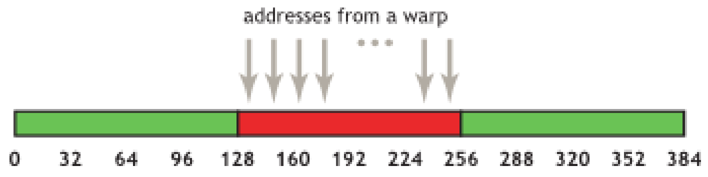

Figure 2.7: Coalesced access.

locations [36]. 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 2.7 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.

2.4.3 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.

2.5 Summary

Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. With CUDA, a software platform and programming model created by Nvidia, GPUs have become highly parallel general-purpose machines.

CUDA is an extension to the programming languages C, C++ and Fortran with its own functions and reserved words. It allows the definition of user functions, called *kernels*, that are executed in parallel by CUDA threads. These threads are organized in *blocks* and these blocks are, in turn, organized in a *grid*. To assign work to each thread and control their execution, threads and blocks are identified with indexes that determine their positions.

CUDA threads have access to different memory types. Each thread has a private *local memory* 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*.

CUDA applications are compiled using *nvcc* and can be optimized using several techniques like coalesced memory access or additional shared memory use.

Chapter 3

Datalog

Datalog is a language based on first order logic that has been used as a data model for relational databases [1, 2]; syntactically it is a subset of Prolog [37]. A Datalog program consist 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 as queries.

Datalog received its name from David Maier [38]. Datalog started in 1977 at a workshop on logic and databases with a simple but powerful idea: to add recursion to positive first order logic queries. In the 80's and early 90's, logic programming was a very active research domain and, as a result, Datalog flourished. However, industry useful applications were non-existent, as Hellerstein and Stonebraker wrote in 1998 [39]: "No practical applications of recursive query theory ... have been found to date". This caused Datalog research to be almost completely abandoned [40].

In recent years, Datalog has returned as part of new applications in the following domains: data integration, declarative networking, program analysis, information extraction, network monitoring, security, and cloud computing [40]. The interest in Datalog for these new applications, as in the past, is the ability of Datalog to compute the transitive closure of relations through recursive queries which, in effect, turns relational databases into deductive databases, or knowledge bases.

This renewed interest in Datalog has in turn prompted new designs of Datalog targeting computing architectures such as GPUs, Field-programmable Gate Arrays (FPGAs) [40] and cloud computing based on Google's Mapreduce programming model [41].

This chapter presents various aspects of the Datalog language: its syntax and semantics, its relation to and translation into relational algebra operations, the approaches to evaluate Datalog programs and optimisations. Finally, the chapter briefly describes some of the recent new applications where Datalog is being used.

3.1 Applications

Recently, Datalog has been used as part of new applications in the following domains:

Data Integration

Data integration is the combining of heterogeneous data sources into an unified query and view schema. In the work of Green et al. [3], Datalog is used to calculate provenance information when a datasource is to be modified by a query. Since provenance information may not be complete, they extend Datalog with Skolem functions [42] to represent unknown values.

In another work by Lenzerini [4], the power of Datalog to express queries and views of heterogeneous data is compared against other languages. These languages include conjunctive queries, positive queries and first-order queries.

Declarative Networking

Declarative networking is a programming methodology to specify network protocols and services using high-level declarative languages. These languages are, in turn, compiled into lower level languages that implement these protocols and/or services. Boon Thau Loo et al. [5] propose *NDlog*, an extension of Datalog, as the high-level language for declarative networking. It differs from traditional network protocol languages in the absence of communication primitives like "send" or "receive". It is also different from traditional Datalog because it considers networking specifics such as distribution, linklayer constraints, etc.

Boon Thau Loo et al. [6] extend their *NDlog* language with *Overlog*. With *Overlog*, it is possible to implement the soft-state approach common in network protocols. The idea is that data has a lifetime or time-to-live (TTL); data has to be refreshed every certain amount of time or it is deleted. *Overlog* accomplishes this by a special keyword at the beginning of each program that specifies the TTL of each predicate in seconds.

Program Analysis

Program analysis is the automatic analysis of computer programs. This analysis can be static (without executing the program) or dynamic (by executing the program). The applications of program analysis are program correctness (every input must return the correct output) and program optimization (to reduce resource utilization or increase efficiency).

Martin Bravenboer and Yannis Smaragdakis [7] implemented the *Doop* framework, a points-to (or pointer) analyser for Java programs [43] based on Datalog. Points-to analysis determines “What objects can a program variable point to?”. By using their highly optimized Datalog recursion, they are able to perform this analysis with a speedup of up to 15x when compared to other well-known analysers.

Information Extraction

Information extraction (IE) is the automatic extraction of structured information from documents, web pages, annotations, etc. Lixto is a web data extraction project by Gottlob et al. [8] based on *Elog* and XML [44]. *Elog* is an extension of monadic Datalog with conditions to detect “false positives” while extracting data, among other things. Monadic Datalog requires all rules to have arity one in their heads. The special properties of this particular Datalog over trees make it an efficient data extraction language.

Another Datalog IE tool was created by Shen et al. [9]. Compared to Perl [45] or

C++ information extraction programs, their Datalog extension called *XLog* provides smaller and easier to understand programs. An interesting addition to Datalog by *Xlog* are procedural predicates (note that Datalog is a truly declarative language). These predicates receive a set of tuples, perform some computations over the tuples using Java or C++ and return another set of tuples back to the Datalog rule.

Network Monitoring

Network monitoring is the continuous analysis of a computer network to obtain traffic information, component failure, etc. For peer-to-peer (P2P) applications, Abiteboul et al. [10] use an extension of Datalog called *dDatalog*. This Datalog distributes its rules over the peers in the network according to the information each of them possesses. To efficiently evaluate *dDatalog*, a distributed version of the query-subquery (QSQ) top-down evaluation strategy, called *dQSQ*, is used (SQS is described in Section 3.4.3).

Security

Marczak et al. [11] implemented *SecureBlox* a distributed query processor with security policies. *SecureBlox* is an enhancement of *LogicBlox* with additional predicates to define write permissions, cryptography, etc. *LogicBlox* is a platform based on an extension of Datalog called *Datalog*^{LB}. This Datalog extension allows the declaration of integrity constraints (e.g. functional dependencies). The difference between these constraints and Datalog rules is that a constraint ensures, for the data in its head, that its body is true (in contrast, a rule uses its body to derive data for the head).

Trevor Jim [12] created the Secure Dynamically Distributed Datalog (SD3) platform. It includes a trust manager system, a policy evaluator and a certificate retrieval system. SD3 extends Datalog's predicate names with an additional value that helps determine who is in control of the relation defined by the predicate. This means that a predicate will be true only if its controller (the one that has the relation) says it is true. The advantages of this platform over other trust management systems are its high-level language which abstracts many complex details and its ability to quickly create security policies from scratch or by modifing existing ones.

Cloud Computing

Cloud computing is the execution of programs over many computers connected in a network. Alvaro et al. [13] presented a distributed data analytic stack implemented using *Overlog*. Since *Overlog* was developed for networking, they implemented a new Java-based runtime called JOL. This runtime allows Java objects to be stored in tuples and Java functions to be called from *Overlog*. Their system was tested against Hadoop [46] showing a slightly worse but still competitive performance. While they attribute many of the benefits of their system to *Overlog*, they also note that *Overlog* has many bugs related to ambiguities in its semantics.

3.2 Datalog Syntax and Semantics

In this section we define the syntax of Datalog programs with some examples. We also describe the characteristics of facts and rules that allow their parsing and analysis.

3.2.1 Syntax

A Datalog program consist of a finite number of facts and rules. 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, for example 'If X is the father of Y and if Y is the father of Z, then X is the grandfather of Z'. To represent rules and facts, Datalog uses clauses which are a finite set of literals. These literals, also called predicates, are an atomic formulas (atoms) or their negations. An atom is the smallest unit in Datalog and has the following structure: $A(x_1, ..., x_n)$, where A is the name of the atom and x_i is either a variable or a constant.

The left side of a clause is called head and the right size is called body, which can be empty. Clauses without body are facts; clauses with at least one literal in the body are rules (literals in the body are also called subgoals). Datalog can also use recursive rules which facilitate specifying (querying for) the transitive closure of relations, which is a key concept to many applications [40].

For example, the facts 'John is Harry's father' and 'David is John's father', can be represented as:

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

The rule 'If X is the father of Y and if Y is the father of Z, then X is the grandfather of Z', is represented as:

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

Datalog programs can derive many new facts. Sometimes, only a subset of these facts is of importance. To derive only the necessary fact subsets from the rules, a query is used. Queries are specified as a single clause followed by a question mark. For example, the query 'Who is the grandfather of harry', is defined as:

```
grandfather(harry, X)?
```

3.2.2 Parsing

Datalog programs are usually read from a file and must be parsed into data and instructions that machines can process. While all Datalog engines follow a similar syntax based on Prolog, slight variations are possible. For example, in our engine, we use a question mark at the end of a clause to represent a query, while other engines use the question mark at the beginning of the clause.

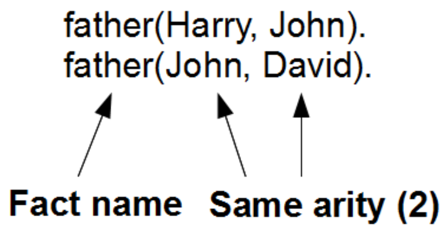

Figure 3.1: Fact structure.

Facts

To parse a fact, the following properties must be considered:

- All the characters before the parentheses compose the name of the fact.
- Two or more facts can have the same name but they must also have the same arity (number of subgoals).
- The name of a fact and a rule cannot be the same.
- Inside the parentheses all elements are separated by commas.
- After the closing parentheses, a dot is used to specify the end of the fact.

Figure 3.1 shows an example of these properties.

Rules

All rules have the following in common:

- Rules can have variables and constants.
- Variables start with a capital letter.
- Constants starting with capital letters should be within single quotes.
- Constants do not appear in the head of the rule.
- The head of the rule is separated by a colon followed by a hyphen.

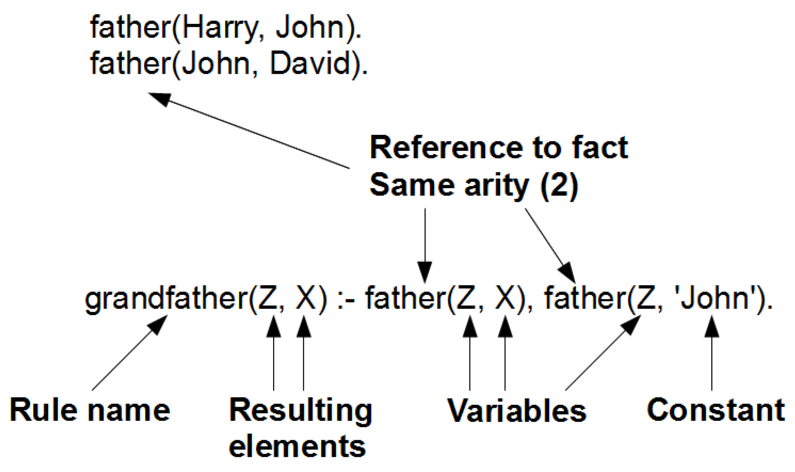

Figure 3.2: Rule structure.

- The name of the rule and the elements that compose the result of the rule are in the head.
- Two or more rules can have the same name and same or different arity.
- The name of a fact and a rule cannot be the same.
- Each clause of a rule has a name and a set of elements of its own.
- The name of each clause must refer to an existing fact or rule.
- The arity of each clause must match the arity of the fact or rule it refers to.
- After the last closing parentheses, a dot is used to specify the end of the rule.

As an example, consider the program of Figure 3.2.

Queries

When parsing a query, the following should be considered:

- Queries can have variables and constants.
- Two or more queries can have the same name.
- A query must have the same name of an existing fact or rule.

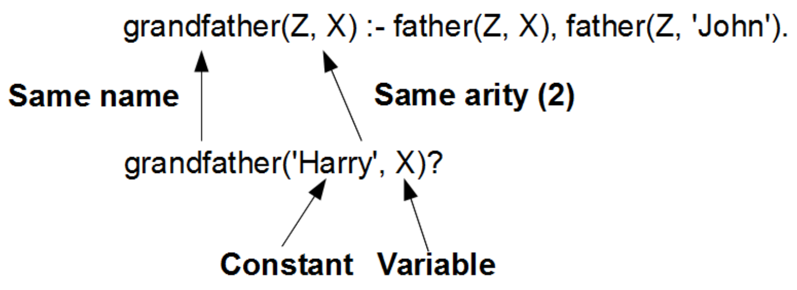

Figure 3.3: Query structure.

- A query must have the same arity of the clause it refers to.
- After the closing parentheses, a question mark is used to specify the end of the query.

Figure 3.3 gives an example of a query.

3.3 Datalog Programs and Relational Algebra

Every Datalog program can be translated into a series of positive relational algebra (RA^+) operations [47] (RA^+ is relational algebra without set difference). 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^+. Due to lack of negation (difference in relational algebra), Datalog cannot answer all the queries that classic relational algebra can.

3.3.1 Relational Algebra

Relational algebra [48] is the combination of first-order logic and set algebra that operates over finite relations (tables). It has many operations, but only the following are part of our Datalog engine:

- **Selection.** Selection is a unary operator that takes all the tuples that comply with a certain condition and discards the rest. It is represented as $\sigma_{condition}(R)$, where R is a relation and *condition* is usually a formula that includes comparison ($<, >, =, \neq$) and logical operators ($\wedge, \vee, \neg$) over attributes (columns) and constants (e.g. $c1 > c2 \wedge c2 \neq c3$ where $c1$ $c2$ and $c3$ are the attributes of a relation).

- **Projection.** Projection is a unary operator that leaves the required columns and discards the rest. It is written as $\Pi_{columns}(R)$, where *columns* are the name of the attributes to conserve.
- **Equijoin.** An equijoin is the combination of all tuples in two relations that have equal values over some defined attributes. It is represented as $R \bowtie_{columns} S$, where R and S are the two relations and *columns* are pairs of equalities over attributes joined together by *ands* (e.g. $c1 = c2 \wedge c4 = c5$, where cn are attributes). In this work, we call an equijoin over only one pair of values (e.g. $R \bowtie_{c1=c2} S$) a join or single join. We also call an equijoin over two or more pairs of values (e.g. $R \bowtie_{c1=c2 \wedge c3=c4...} S$) a multijoin. Finally, an equijoin over the same relation (e.g. $R \bowtie_{c1=c2 \wedge c3=c4...} R$) is called a selfjoin.

Section 3.3.2 has examples of these operations.

3.3.2 Translating Datalog Programs into Relational Algebra

To understand the transition from Datalog programs to relational algebra operations, we start by showing how each element of a Datalog program can be seen as an element in a relational database. First, each fact in a Datalog program can be seen as a row on a table: the head represents the name of the table and the body represents the elements of that row. For example, consider the following facts:

```
father(Harry, John).
father(John, David).
```

Their corresponding table would be Table 3.1.

Rules can be seen as virtual views [38], i.e., they represent operations over facts and rule results that are executed each time the rule is evaluated. Their results can

father	
Harry	John
John	David

Table 3.1: Datalog facts as a table.

be seen as tables that have the lifetime of the program. For example, consider the following rule:

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

This rule performs a join and a projection (as described shortly), generating the view represented by Table 3.2.

grandfather	
Harry	David

Table 3.2: Result of a Datalog rule as a view.

Now, consider a Datalog rule r. All predicates $p_1, ..., p_n$ in the body of rule r represent relations $P_1, ..., P_n$, where each P_i consists of all the tuples $t_1, ..., t_m$ that make predicate p_i true. This means that subgoals in the body of a rule are made true, by a certain set of tuples, based on the variables and constants in the body. If all subgoals in the body are true, then the head of the rule is also true. To obtain the set of tuples that make the head true, we must transform the Datalog program into relational algebra equations by following these translation rules:

- **Selection.** Selection is applied when constant values appear in a predicate. (e.g. $a(constant, X)$, Figure 3.5).

- **Join.** A join is made between two subgoals in the body of a rule using a pair of variables as reference. The result of the join can be seen as a temporary predicate that has to be joined in turn to the rest of the subgoals of the body. (e.g. $a(X, Y), b(Y, Z)$, Figure 3.6).

- **Multijoin.** Two or more pairs of common variables between two predicates represent multijoins (e.g. $a(X, Y), b(X, Y)$, Figure 3.9). Its result is also a temporary subgoal.

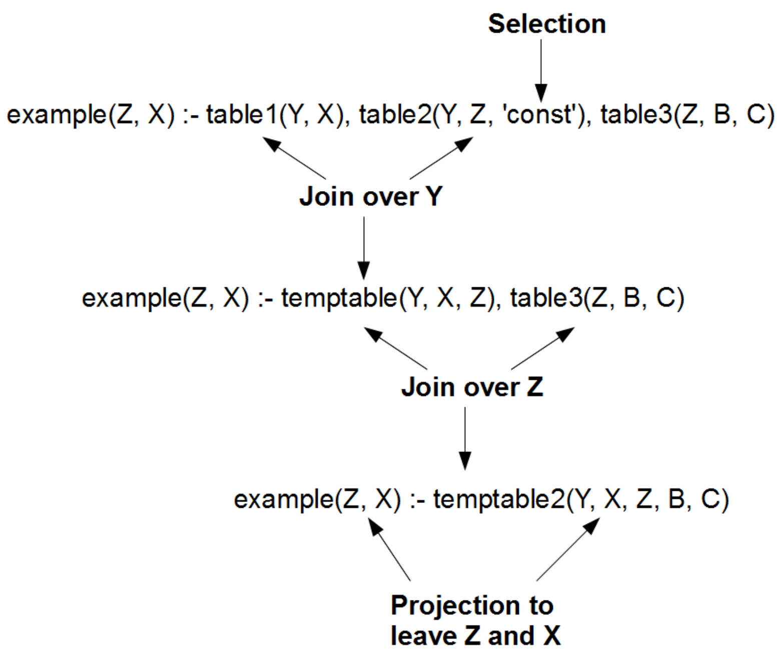
Selection
example(Z, X) :- table1(Y, X), table2(Y, Z, 'const'), table3(Z, B, C)
Join over Y
example(Z, X) :- temptable(Y, X, Z), table3(Z, B, C)
Join over Z
example(Z, X) :- temptable2(Y, X, Z, B, C)
Projection to leave Z and X

Figure 3.4: Rule evaluation based on relational algebra operations.

- **Selfjoin.** If two or more common variables are found in the same subgoal, a seljoin is applied (e.g. $a(X, X)$, Figure 3.8).

- **Projection.** Determined by the variables in the head of the rule (e.g. $r(X, Y) : -a(X, Y, Z)$, Figure 3.7).

As a general example, consider the rule of Figure 3.4 and the relational algebra operations required to solve it.

Rule translation

As a complete, step by step translation example, consider following Datalog program:

```
rule(X, Z) :- rel1(constant, X), rel2(X, Y, Y, Z), rule(Z).
rule(X, X) :- rel3(X).
```

To translate this program to relational algebra, first we translate each subgoal in the body of these rules as follows:

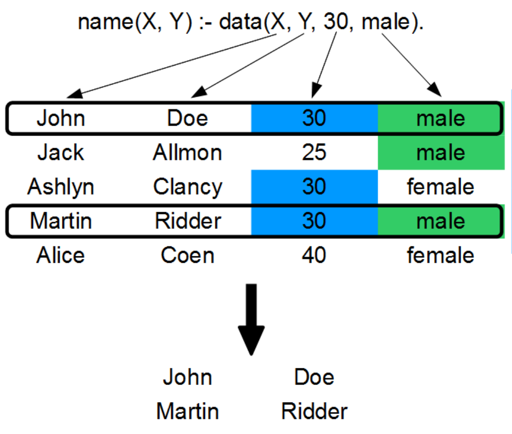

Figure 3.5: Selection.

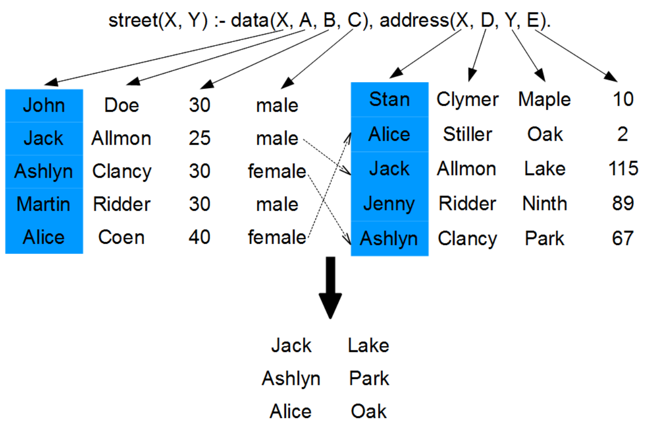

Figure 3.6: Single Join.

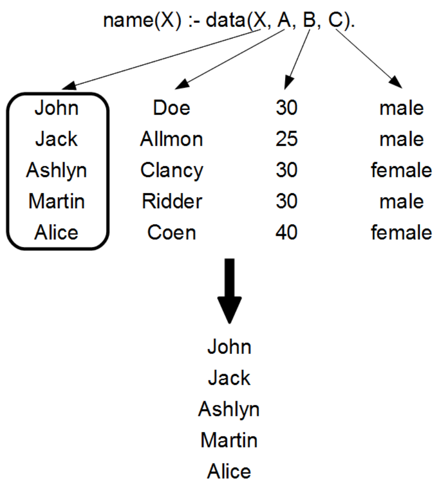

Figure 3.7: Projection.

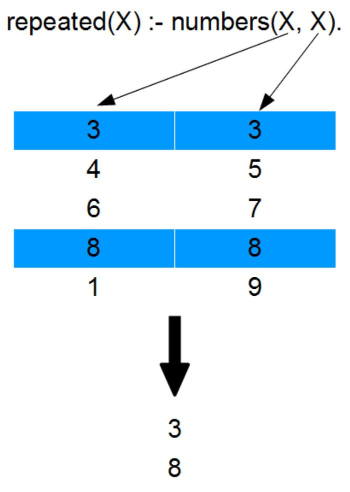

Figure 3.8: Selfjoin.

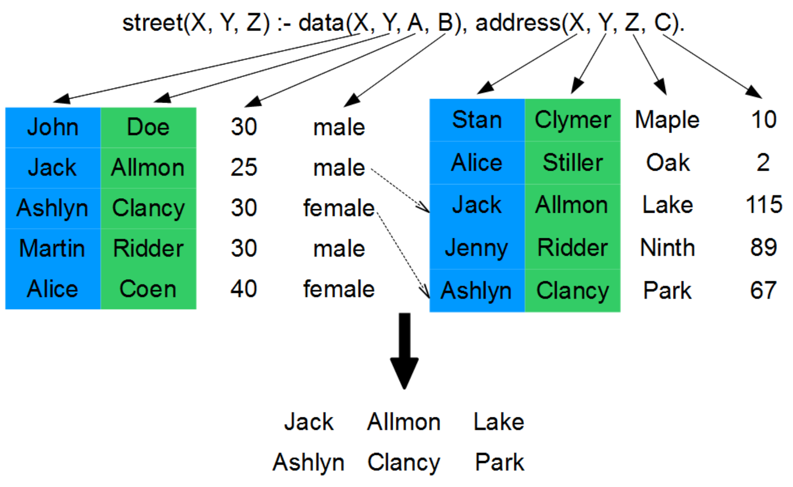

Figure 3.9: Multijoin.

$$\begin{aligned} A(X) &= \Pi_2(\sigma_{1=constant}(rel1)) && (3.1)\\ B(X,Z) &= \Pi_{1,4}(\sigma_{2=3}(rel2)) && (3.2)\\ C(X) &= \Pi_{1,1}(rel3) && (3.3)\end{aligned}$$

Note that the elements of each subgoal are changed to numbers depending on their position (e.g. variables X and Y in q(X,Y) would be 1 and 2 respectively). The first equation $A(X)$ is a selection on column 1 with value 'constant' and then a projection to get only the second column. $B(X,Z)$ is a selfjoin over columns 2 and 3 followed by a projection to get columns 1 and 4. $rule(Z)$ requires no operation and is left as it is. Finally, although $rel3(X)$ requires no additional operation per se, the variable in the head of the rule is repeated, so a projection on $rel3(X)$ is necessary to have the correct result.

Once we have each subgoal as a relational algebra equation, we build the translation of each rule as a series of joins over the subgoals:

$$A(X) \bowtie B(X,Z) \bowtie rule(Z) \subseteq rule \qquad (3.4)$$

$$C(X) \subseteq rule \qquad (3.5)$$

All rules with the same name are then fused together into a single equation having the rule name as left-hand side (LHS), and the union of all the left-hand sides of the rules as right-hand side (RHS). For instance, from the above equations we obtain the following equation:

$$rule = A(X) \bowtie B(X,Z) \bowtie rule(Z) \cup C(X) \qquad (3.6)$$

Query translation

Datalog queries are also translated into relational algebra queries using projections and selections. For example, the Datalog query $rule(a, X)$ to the program presented above, is equivalent to the algebraic query $\Pi_2(\sigma_{1=a}(rule))$.

3.4 Evaluation of Datalog Programs

Datalog programs can be evaluated through a top-down approach or a bottom-up approach. Which method is better has been object of much debate [49, 50], since both approaches have their advantages and disadvantages. Worse yet, there are programs that perform well with one approach but poorly on the other. Both evaluation strategies must satisfy three important properties:

- **Soundness.** The result of a program must not have tuples which do not belong to it.
- **Completeness.** All tuples of the result must be found.
- **Termination.** The program must be evaluated in finite time.

3.4.1 Datalog and Prolog

The syntax of Datalog is a subset of Prolog, so it can be parsed by a Prolog engine. However, there are several differences in the evaluation of Datalog and Prolog programs:

- Prolog respects the order of the rules and the subgoals in each rule. Datalog do not requires a certain order in neither rules nor subgoals.
- Prolog termination depends on proper rule and subgoals ordering. The termination of Datalog programs is unaffected by rule and subgoals ordering.

- Prolog works by retrieving data from memory one-tuple-at-a-time. Some Datalog evaluation strategies are set-oriented (i.e. take entire relations from memory). The reduced number of memory transactions make set-oriented methods a more efficient approach.

3.4.2 Bottom-Up Evaluation

The bottom-up approach works by applying the rules to the given facts, thereby deriving new facts, and repeating this process with until a fixed point is reached (i.e. no more new facts are derivable). This approach has several benefits:

- It can avoid infinite loops by correctly evaluating repeated or cyclic subgoals [49].

- Rules can be evaluated in any order and even in parallel.

- It works on sets of tuples, instead of one-tuple-at-a-time like most top-down implementations, thus decreasing the required number of memory transactions.

One disadvantage of this approach is that, at each iteration, facts we have already used in the computation of other facts are used again, deriving nothing new from this repeated use. This problem is solved by the semi-naive bottom-up approach which, at each iteration, considers only the newly derived facts.

Another disadvantages is that it is not goal-oriented. This means that rule evaluation generates many unnecessary tuples and performs additional computations because the queries are considered only at the end.

To improve the bottom-up approach, several methods have been proposed such as the magic sets transformation [51] or the subsumptive demand transformation [52]. Basically, these methods transform a set of rules and a query into a new set of rules such that the set of facts that can be inferred from the new set of rules contains only the facts that would be inferred during a top-down evaluation.

3.4.3 Top-Down Evaluation

The top-down approach, which is the one used by the Prolog language, starts with the goal which is reduced to subgoals, or simpler problems, until a trivial problem is reached. Then, the solution of larger problems is composed of the solutions of simpler problems until the solution of the original problem is obtained.

The advantage of top-down algorithms is that they are *goal-oriented*. That means that the query is considered early in the computation, thus ignoring facts that are not necessary to generate the result. This translates into a reduced number of unnecessary computations.

The disadvantage of these algorithms is that answers are computed a-tuple-at-a-time, i.e. only a small subset of the data is accessed each time a subgoal is to be answered. These small accesses are due to the simpler nature of subgoals. From the computational point of view, this means that additional memory transactions have to be performed. For massive parallelism this is an undesirable feature.

A top-down evaluation algorithm that solves the a-tuple-at-a-time problem is the query-subquery (QSQ) algorithm [53]. The idea is to consider both a goal and a Datalog program as a query. All predicates in the body of the rules that answer the goal are subgoals. Subgoals, together with the Datalog program, define subqueries. These subqueries are in turn expanded into more subqueries until the answer to each subquery requires only ground facts (facts that were not derived from rules).

The most common strategy to improve the performance of top-down methods is known as *tabling* [52]. The idea is to reuse the answers of subgoals to answer other subgoals, and thus, reduce the number of computations.

3.5 Extensions and Optimizations

In this section we present the most important extensions and optimizations to Datalog. These extensions increase Datalog's expressive power, allowing additional types of queries to be evaluated. Optimizations usually increase the performance of

Datalog programs.

3.5.1 Extensions

Datalog can be extended with built-in predicates, negation, complex objects, etc. These extensions have their own syntax and usually require changes to the entire implementation of rule evaluation.

Built-in predicates

Built-in predicates are special symbols such as $=, \neq, >, <$, that force variables to take certain values, e. g. the predicate $Y > 5$ would force Y to only take values greater than 5. They can appear in the right-hand side of a rule and are written in infix notation. Two important restrictions must be enforced when working with these predicates: a) to guarantee a finite output of the Datalog program (also known as safety), each variable involved in a built-in predicate must also appear in a nonbuilt-in predicate of the same rule body; and b) the evaluation of built-in predicates must be delayed until all variables involved are bound to constants, otherwise it is impossible to know which tuples are part of the computation of these predicates.

A detailed description of these predicates can be found in [54, 47]. In relational algebra, most of these predicates can be seen as join conditions. For example, consider the following rule:

```
rule(X) :- fact1(X, Y), fact2(X, Z), Y < Z.
```

This rule can be translated into the following relational algebra equation:

$$rule = \Pi_1(fact1 \bowtie_{1=1\, and\, 2<2} fact2) \tag{3.7}$$

where $1 = 1$ represents the join over X and $2 < 2$ is the built-in predicate transformed into a join condition. This means that the rows of $fact1$ will only be joined to the rows of $fact2$ if both rows are equal on the first column and if the second column of $fact1$ is less that the second column $fact2$.

Negation

To increase its expressive power, Datalog can be extended with the addition of negation in predicates [38, 55]. It is usually represented by the symbol ¬ and can appear in both the body and the head of a rule (e. g. `¬A(X) :- B(X, Y), ¬C(Y).`). In relational algebra, negation is equivalent to the difference operator (−). Negation with recursion, as is required by Datalog, is a difficult task with several possible implementations:

- **Stratification.** The idea is to compute all rules defining a negative predicate before it is used. This can be done by analysing the program and properly organizing the rules in it. This approach works only if all rules defining a negative predicate can be completely evaluated before the predicate is used. Programs that fulfil this condition are called stratified.

- **Well-founded Semantics.** Are based on the idea that a program may not derive every fact as true or false. It uses a three-value logic: true, false and unknown or undefined. It can be used for all Datalog programs but the answers are not guaranteed to provide the entire information. See [56] for details.

- **Inflationary Semantics.** Its name comes from the idea that once a fact has been inferred, it is always considered to be true. The evaluation is similar to the fixed-point method: all rules are used at each step to infer new facts until no new facts can be inferred. If a negative fact is not yet derived, it is considered to be true. The disadvantage of this method is that the answer is not minimal (e.g. under inflationary evaluation, a program may return (a, b, c) as answers, while the same program under other evaluations may return (a, b) or (a, c)).

- **Noninflationary Semantics.** Similar to inflationary semantics, rules are iterated until no new facts are inferred. However, two possible improvements make it noninflationary: the retaining of only new inferred facts at each iteration or the removal of an already inferred fact if found to be false. Its disadvantage is that termination is not guaranteed. Both inflationary and noninflationary semantics are well described in [38].

3.5.2 Optimizations

These optimizations described below can be implemented without compromising the semantics of the language. The examples presented in this section where taken from [57].

Magic Sets

Refer to a logical rewriting method used in many deductive database systems to improve the performance of bottom-up evaluation. It transforms a program by adding new rules which represent the query under consideration. The result of the new program is equivalent to that of the original one. By doing this transformation, the variables in the rules are restricted to take only certain values in a way similar to the top-down approach. This reduces the number of unnecessary facts, and thus the required amount of memory and computations. This method is well documented in logic programming literature [57, 51]. As an example, consider the following Datalog program to compute the same generation cousins:

```
sg(X, X).
sg(X, Y) :- par(X, X1), par(Y, Y1), sg(X1, Y1).
sg(a, W)?
```

Where *par* is a series of facts that define who is the parent of who, e.g. `par(A, B).` would mean that B is a parent of A. The magic sets method is applied by adding two new rules and rewriting the other two:

```
magic(a).
magic(U) :- magic(V), par(V, U).
sg(X, X) :- magic(X).
sg(X, Y) :- magic(X), par(X, X1), par(Y, Y1), sg(X1, Y1).
```

The first rule adds *a* to the *magic* relation. With *a* in this relation, the second rule will take the second column of all facts of relation *par* that have *a* in the first column (i.e., it will take all the parents of *a*). Since relation *magic* has derived facts (*a* and his parents), the *magic* predicate in the third and fourth rules will force them to start their recursive search with these derived facts. By doing this, all facts in relation *par* that are not related to *a* will not be considered.

Counting

Counting is an extension of the Magic Sets method that complements each element of the magic set with an index. This index represents the “distance” to the goal constants, allowing the evaluation to consider only those tuples that have the correct “distance”. While this method further reduces the number of computations, it can only be used in linear programs with acyclic databases, otherwise termination is not guaranteed. This method is described in [58, 59]. As an example, consider the counting transformation of the same program described in the magic sets method:

```
ancestor(a, 0).
ancestor(X, I) :- par(Y, X), ancestor(Y, J), I = J + 1.
cousin(X, I) :- ancestor(X, I).
cousin(X, I) :- par(X, Y), cousin(Y, J), I = J - 1, I >= 0.
sg(X) :- cousin(X, 0).
```

The first rule establishes *a* as the origin (hence the zero). The second rule recursively obtains all the ancestors of *a* and assigns them their corresponding distance to *a* (e.g. the parents of *a* would have a one, his grandparents would have a two and so forth). The third rule stores all the ancestors of *a* into relation cousin so they can be used recursively by the forth rule. Starting with the ancestors of *a*, the fourth rule recursively computes the cousins of *a*. The fifth rule returns the desired result by restricting the search to the same generation cousins of *a*.

3.6 Summary

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 a finite number of facts and rules. Facts are statements about something relevant. Rules allow the deduction of new facts from known facts.

Recently, Datalog has been used in new applications, including data integration, declarative networking, etc. This interest in Datalog has in turn prompted new designs of Datalog targeting computing architectures such as GPUs, FPGAs and cloud computing based on Google's Mapreduce programming model.

Every Datalog program can be translated into a series of relational algebra operations. The following relational algebra operations are used in Datalog: *selections* are made when constants appear in the body of a rule. Then a *join* is made between two or more elements in the body of a rule using the variables as reference. Finally, a *projection* is made according to the variables in the head of the rule.

Datalog evaluation can be performed bottom-up, starting from the existing facts and inferring new facts, or top-down, reducing the goal into simpler subgoals and solving them. Datalog can be extended to evaluate other types of queries, e.g. with negation, and optimize in various ways to improve performance.

Chapter 4

A Datalog engine for GPUs

In this chapter we present the design of our Datalog engine for GPUs.

As mentioned in Chapter 3, Datalog is being used in a wide array of applications other than as a data model of relational databases. This renewed interest in Datalog has in turn prompted new designs of Datalog targeting computing architectures such as GPUs, Field-programmable Gate Arrays (FPGAs) [40] and cloud computing based on Google's Mapreduce programming model [41].

Our work is one of those new Datalog designs targeted at GPUs. Yet it is the first fully functional Datalog engine for GPUs to the best of our knowledge.

GPUs can substantially improve performance of data-intensive, highly parallel applications such as database relational operations, substantially in many cases [60, 61, 62]. However, the communication-to-computation ratio may become a factor. This ratio must be relatively low for applications to show good performance, i.e.: the cost of moving data from the host memory to the 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 including various optimisations aimed to reduce the communication-to-computation ratio. Data is preprocessed in the host (a multicore) in order both for data transfers between the host and the GPU to take less time and for data to be processed more efficiently by the GPU. Also, a memory management scheme automatically swaps data between memory in the host platform multicore and memory in the GPU in order to reduce the number of such 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.

This chapter presents first an overview of the organisation and functioning of our Datalog engine, and then some detail of the functioning of its main modules: the compiler, the scheduler, the relational algebra operations and support operations. Finally, we present related work, after our design in order to more easily contrast it with our work.

4.1 Architecture

Figure 4.1 shows the main components of our Datalog engine. The engine represents a hybrid solution in that both GPU and CPU are used. Highly parallel code (i.e. relational algebra operators) is executed on the GPU while secuential code (i.e. input/output operations, control) is executed on the GPU. Our engine is organized into three stages with a single *host* thread executing the first and third stages, and scheduling work to the GPU during the second stage. These stages are called the Preparation stage, the Evaluation stage and the Termination stage, respectively, and will be outlined shortly.

All 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.

A memory management module, not shown in Figure 4.1 is always active during the evaluation and termination stages to help identify the most recently used data within the GPU in order to maintain it in global memory and discard sections of data that are no longer necessary.

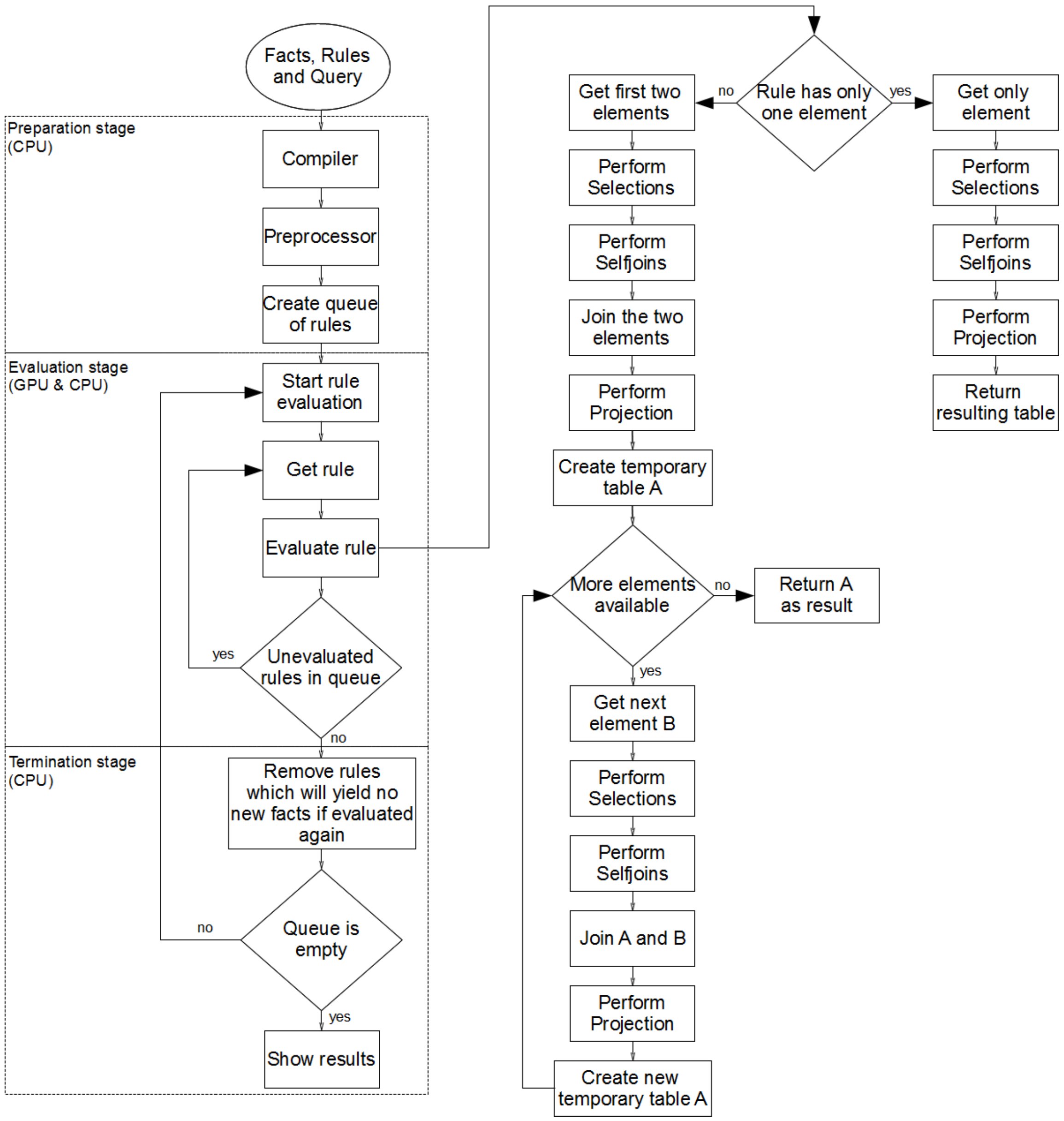

Figure 4.1: GPU Datalog engine organisation.

4.1.1 Preparation Stage

The preparation stage begins with the compiling of the Datalog program to store it in memory with an appropriate format. Once compiling is completed, the preprocessor analyses each rule to determine which operations to perform and over which tables and columns they will be performed. Finally, the query is analysed to determine which rules are to be evaluated. A queue is created with these rules.

Compiler

To capitalise on the GPU capacity to process numbers and to have short and constant processing time for each tuple (strings variable size 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.

As mention in Chapter 3, facts with the same name can be seen as a table. We store tables in contiguous linear arrays. This allows us to transfer arrays directly to the GPU without additional modification. To determine the beginning and end of each fact in the array, we use the number of columns of the table (arity of facts) and the total number of rows (total number of facts). For example, consider the following facts:

```
father(Harry, John).
father(John, David).
```

The strings of these facts are stored into the hashed dictionary as show in Table 4.1 and into the linear array as in Table 4.2. Notice that the number of rows is two (two facts with the same name) and that the number of columns is also two (the arity of both facts is two). Thus, we know that the first fact in the array starts at position 0 and ends at 1. The second fact starts at position 2, ends at 3 and the array ends at 3 too.

String	ID
father	1
Harry	2
John	3
David	4

Table 4.1: Datalog facts in hashed dictionary.

1	2	2	3

Table 4.2: Datalog facts as an array according to the hashed dictionary.

String	ID
grandfather	4
X	6
father	1
Y	7
Harry	2

Table 4.3: Datalog rule in hashed dictionary.

4	6	0	1	7	6	0	1	-2	7	0

Table 4.4: Datalog rule as an array.

Rules are also stored in a linear array. However, unlike facts, they cannot be grouped together by name and their size is variable. This requires that we use one array per rule and that we separate each element in the rule with a special character (0). This special character also helps us determine the name of each predicate in the rule. Finally, rules include both constants and variables. To distinguish between them, we use positive numbers for variables and negative numbers for constants. For example, consider the following rule:

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

The strings of the rule are stored into the hashed dictionary as show in Table 4.3 and into the linear array as in Table 4.4. A zero is placed at the end of each predicate as separator. The last zero also serves to indicate the end of the rule. Predicate names will always be at the beginning of the array and after each zero except the last one. Thus, the predicate names are at positions 0, 3 and 7. It can also be seen that variables and predicate names use positive numbers, while the constant (Harry), has the same number assigned by the hash dictionary but negative (-2).

To keep track of additional information regarding facts and rules (name, arity, number of subgoals, etc.), we store this information in a list of structures. The structure used is as follows:

```
struct predicates{
    int name;
    int num_rows;
    int num_columns;
    int is_fact;
    int *address_host_table;
}
```

Where *name* is the name of the predicate as assigned by the hashed dictionary. If a table is stored in the structure, *num_rows* represents the number of rows, otherwise it represents the number of predicates. *num_columns* represents the number of columns when storing table information, and the arity of the rule head when storing rule information. To determine what we are storing, we set *is_fact* to 1 when storing tables and 0 when storing rules. Finally, *address_host_table* stores a pointer to the arrays described above.

Preprocessor

A key factor for good performance is preprocessing data before sending it to the GPU. As mentioned in Chapter 3, 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 it. 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* with column 4 in *fact2*, and column 3 in *fact1* with column 0 in *fact2*.

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

- **Selfjoin** can be performed over two or more columns and more than one selfjoin can be performed in the same subgoal. Two or more values represent the column numbers required for a selfjoin, followed by a negative number (-1). The following example has two selfjoins: the first one is performed over columns 0, 1 and 2 (X), the second over columns 4 and 6 (Y). The first -1 separates the two selfjoins, and the second -1 indicates the end of the array.

```
fact1(X,X,X,A,Y,B,Y). -> [0, 1, 2, -1, 4, 6, -1]
```

- **Projection** is performed after each join as specified in Section 4.3. Because of this, we use two series of values, one for each subgoal to be joined, representing the columns that will remain after the projection. They are also separated by a negative number (-1). In the following example, the first series represents the columns of *fact1* and has column 0 (X) and column 2 (Y) as required by the rule head; the second series represents the columns of *fact2* and has only column 1 (Z). Separators (-1) are used at the end of each series.

```
rule(X,Y,Z) :- fact1(X,A,Y), fact2(A,Z). -> [0, 2, -1, 1, -1]
```

Along with each array, a counter to track the number of operations to perform is used. For instance, consider the array `[0, 'a', 2, 'b', 5, 'c']` in the selection example, the counter associated with this array will have its value set to 3, because three selections are to be performed. These counters also allow us to calculate the size of the array by multiplying its value times two (in the selection and join operations) or by counting the number of separators (in the selfjoin and projection operations).

4.1.2 Evaluation Stage

The evaluation stage takes the rule queue and evaluates each rule at a time. If the rule has only one subgoal, any selections required by this subgoal are performed first, followed by any selfjoins and, finally, a projection to obtain the result. If the rule has two or more subgoals, the first two subgoals are taken and any necessary selections and projections are performed. Next, the two subgoals are joined (using single or multijoin depending on the variables in the subgoals) and a projection is performed to create a temporary table with the result. If there are more subgoals in the rule, we take the next subgoal, perform any needed selections and selfjoins, and then it is joined with the temporary table. We then perform a projection to create a new temporary table that either has to be joined to the next subgoal or is returned as the result of the rule. Finally, once we have the result of the rule, duplicate elimination is performed to reduce memory and computation requirements.

All relational algebra operations are performed in the GPU and are described in detail in Section 4.2.

4.1.3 Termination Stage

Once 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. To remove rules, we consider the following:

- Rules with only facts as subgoals finish their evaluation in the first iteration.

- Rules with facts and other rules who have already finished as subgoals will finish their evaluation in the next iteration (we say in the next iteration because the result of one or more rule subgoals has yet to be considered by the main rule).
- Recursive rules finish when no new facts are generated by their evaluation.

Once the finished rules are removed, if the queue does not become empty, each rule in it is evaluated again. This process is repeated until the rules queue becomes empty. Once it is empty, we take all the results required by the query in order to answer it (i.e., we perform any selections and/or projections as required by the variables and constants in the query). Finally, any columns in the query result are returned to their original string form.

4.1.4 Memory Management

As mentioned in Chapter 2, global memory is limited by hardware and cannot be expanded using virtual memory. More often than not, it will be impossible for all tables to fit in global memory at the same time. Worse still, it is possible for one table to be so big that it cannot fit in global memory at all. In addition, operations like sort and join, among others, need global memory for temporary arrays.

Since data transfers, i.e. moving tables, between GPU and CPU memory is costly in all CUDA applications [35], an efficient memory management policy that addresses most of the above issues will tend to improve performance.

For this reason, 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. 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 its size, the fact or rule it represents and the pointer to its location in GPU memory.

When some 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 at 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.

Without our scheme, programming would be similar to that based on memory overlays before virtual memory was invented. Most data used or produced by an operation would have to be discarded from GPU memory unless it is known that it will be used by following operations. This not only complicates the coding, but is also dependant on the amount of memory available, which in turn depends on the amount of data being processed.

Our scheme works well for processing many rules that depend on each other or rules that use many different facts. It is very similar to the Demand Paging and Not Recently Used (NRU) Page Replacement algorithm described in [63]; but it works with memory sections as opposed to pages. The most important policies of our scheme are as follows:

- **Loading tables.** Tables are loaded into global memory as needed by an operation.

- **Unloading tables.** Tables are not unloaded until there is not enough space to load a table. When this happens, the table that has the longest time since it was last used will be unloaded, then the one with the second longest time. This process continues until there's enough space to load the required table.

- **Memory for temporary arrays.** Similar to 'unloading', tables with the longest time since they were last used will be unloaded until there's space for the temporary array.

- **Transfers between CPU and GPU.** Transfers are only needed when a table is not loaded or when results are sent to the CPU in order to free memory.

- **Control.** It's necessary to check if a table is loaded or not, counters to keep track of the last time a table was used are also needed and the amount of available memory must be monitored.

4.2 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.

Selection and join represent dynamic problems: the size of the solution depends on the input and cannot be known beforehand. For single threaded implementations, this problem can be solved with dynamic storage (i.e. vectors, lists, queues). The idea is to increase the resulting array each time a solution is found. However, for GPUs, dynamic storage is not possible (since there is no function to increase the size of an array), or very costly (since using pointers require additional global memory reads and may lead to memory fragmentation). Allocating a relatively big result array is also a bad idea due to the required coordination to have each thread write at a different location.

To avoid these issues altogether, we make use of additional kernels that calculate the exact size of the result before writing it on memory, thus leading to a static problem.

4.2.1 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 dynamically 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 kernels. 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.

4.2.2 Projection

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

4.2.3 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.: table1(X,Y) $\bowtie$ table2(Y,Z). A multijoin is used when more than two columns are to be joined: table1(X,Y) $\bowtie$ table2(X,Y). A selfjoin is used when two columns have the same variable in the same predicate: table1(X,X).

Single join.

We use a modified version of the Indexed Nested Loop Join described in [61], 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 [64] (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. The tree is adapted for GPUs by setting the size of each node to be equal to the warp size, thus maintaining some degree of coalesced memory access and reducing cache misses. To speed-up the initial stages of the search, the upper levels of the tree are stored in shared memory.

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 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 is applied this array. With the size and write locations known, a second join writes the results.

Multijoin.

To perform a join over more than two columns, for example table1(X,Y) $\bowtie$ table2(X,Y), we first take a pair of columns say (X,X) to create and search in the CSS-Tree as described in the single join algorithm. Then, in the first join, after performing the counting but before writing it 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 do not comply. In the second join, before writting the result, we check these columns again to decide if we write the element or not.

This requires additional reads to global memory but, when compared to the other common technique of hashing the columns to join into a single column (e.g., example

Z1 = hash(X,X), Z2 = hash(Y,Y) then Z1 $\bowtie$ Z2), it eliminates the cost of creating and storing the hash table.

Selfjoin.

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, it checks if the values of the columns affected by the self join match.

4.3 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.

4.3.1 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, fact1 $\bowtie_Y$ fact2, 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.

4.3.2 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. *fact1* is processed first: the selections required by *'const1'* and *'const2'* are performed at the same time — *fact1* does not require selfjoins. *fact2* is processed second: the selection required by *'const3'* is performed first, and then the selfjoins between Ys and Zs are performed at the same time. Finally, a join is performed between the third column of *fact1* and the first column of *fact2* 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 *fact1* and the fourth column of *fact2*.

4.4 Support Operators

Our relational algebra operations make use of the support operators sort, scan and duplicate elimination. These operations are performed on the GPU to reduce the number of GPU to CPU memory transfers. We use the efficient implementations of these functions provided by the Thrust library [29]. This library is a C++ template library for GPUs based on the Standard Template Library (STL) [30] and provided as part of the CUDA SDK (Software Development Kit).

In this section, we describe the support operators and how they are implemented in the Thrust library.

4.4.1 Sorts

Sorting is required for the single join, multijoin and duplicate elimination operations. Thrust implements two sorting sorting algorithms, merge sort and radix sort. Thrust automatically applies radix sort when sorting built-in types (char, int, float, etc.) with the comparison operator less (<). Otherwise, merge sort is used.

Radix Sort.

The Radix sort [65] works by grouping together integer keys by the individual digits which share the same position and value. The method iterates over digits from least-significant to most-significant. For each digit, it performs a distribution sort of the keys based upon the digit in order to create partitions of R distinct buckets.

The distribution sort (a.k.a. counting sort) is fundamental to the radix sorting method. This sorting algorithm works by having each processor gather its key, obtain the digit at the given position and work with other processors to determine where to place the key.

Merge Sort.

Merge sort [66] is a divide and conquer sorting algorithm invented by John von Neumann in 1945. It divides the unsorted array into n smaller arrays, and then divides these smaller arrays into n even smaller arrays. This process is repeated until each array has one element.

Once we have one element arrays, we merge them back into n bigger arrays by placing each element in its correct position according to its value. These bigger arrays are, in turn, merged together into n even bigger arrays. The process is repeated until we have a single array with all the elements in order.

Input numbers	1	2	3	4	5	6	...
Prefix sum	1	3	6	10	15	21	...

Table 4.5: Prefix sum of natural numbers.

4.4.2 Prefix sums

The prefix sum [19], also known as scan or cumulative sum, is a sequence of numbers (y, y1, y2, ...) created by adding the elements of an input sequence (x, x1, x2, ...):

$$\begin{aligned} y &= x \\ y1 &= x + x1 \\ y2 &= x + x1 + x2 \\ ... \end{aligned}$$

For example, the prefix sum of the natural numbers are the triangular numbers as shown in Table 4.5.

The Thrust library includes many different prefix sums, however, we use only *exclusive_scan* and *inclusive_scan*. Both functions receive the start and end positions of the array to scan and the start position of the array which will store the result. This output array can be the same as the input array to perform the prefix sum in-place. The only difference between these functions is that *exclusive_scan* starts the sum at zero, while *inclusive_scan* begins with the first element of the input array.

4.4.3 Duplicate Elimination

Join operations may generate duplicates, a great deal of them in some cases. These duplicates could, in turn, be used as input to other join operations, thus generating many more duplicates. To avoid wasting memory and computations on duplicates, duplicate elimination must be implemented in the GPU. However, it is not a simple task because the presence and location of duplicates is not known beforehand.

To implement it in our engine, we use 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.

4.5 Related Work

One of the most important aspects of our work is the efficient implementation of relational algebra operators on GPUs. Thus, in this section we present the approach followed by Bingsheng He et al. [67, 61, 62] in the implementation of GPUQP and Gregory Diamos et al. [60, 68, 20, 69] with Red Fox.

GPUQP: Query Co-Processing Using Graphics Processors

GPUQP [23] is an in-memory query co-processor focused on fully exploiting the architectural features of the GPUs. Their GPU engine is designed in four bottom up layers that allow one layer to be modified without altering the others. The four layers, from top to bottom are: operators, access methods, primitives and storage.

Storage refers to the way the relations to be processed are stored and how these relations are indexed.

Primitives are common operations aimed to exploit the GPU hardware features, especially thread parallelism and the fast local memory. They are also scalable to any number of processors because they are designed without locks and the synchronization cost is low. Their primitives include:

- **Map.** Applies a given function to every tuple in an array. Uses thread groups, each responsible for a segment of the relation. The access pattern is designed to exploit the coalesced memory access.

- **Scatter and Gather.** A scatter performs indexed writes to a relation, while a gather performs indexed reads. Using the multi-pass optimization scheme [70], in each pass, the scatter writes to a certain region in the output array; the gather reads the data from a certain region in the input array.

- **Prefix scan.** Applies a binary operator to the input relation (e.g. prefix sum). They adopt the implementation from CUDPP [71, 72].
- **Split.** Divides a relation into a number of partitions according to a given partitioning function. They use the lock-free implementation in [61], which uses histograms to compute the write location for each tuple. Write conflicts are avoided because each thread knows its target position to write.
- **Sort.** Transforms an unordered array into an ordered one. They adopt the quick sort from [61], which has two steps. First, given a set of random pivots, the relation is split into multiple chunks. The split process continues until each chunk is smaller than the shared memory. Then, multiple chunks are sorted in parallel using bitonic sort.
- **Reduce.** Computes a value based on the input relation (e.g. the sum of all key values in a relation). They implement it as a multi-pass algorithm. In each pass, the data is divided into multiple chunks and evaluated in parallel. Each chunk has the size of the shared memory. The result of each chunk forms the input for the next pass.
- **Filter.** Selects a subset of elements from a relation, and discards the rest. It works in three steps. First, the map primitive processes the input array returning a 0-1 result array. Then, a prefix sum is performed on this array. Finally, the results is scattered according to the map and prefix sum output arrays.

Their engine supports the following three access methods:

- **Table scan.** Each row of the table is read in a sequential order and the columns are checked for a condition. Implemented using the map primitive.

- **B+-trees.** They adopt a Cache Sensitive Search Tree (CSS-Tree) specially designed for GPUs described in detail in Section 4.2.3.

- **Hash indexes.** Consist of two arrays, headers and buckets [70]. The header array maintains the starting positions of the buckets. Each bucket stores the key values of the records that have the same hash value, and the pointers to the records. Hash search is performed in four steps. First, for each search key, the map primitive is used to compute its corresponding bucket and the gather primitive to fetch the start and end locations for the bucket. Second, it scans the bucket and determines the number of matching results. Third, based on the number of results for each key, a prefix sum is computed on these numbers to determine the start location at which the results are written. Fourth, a gather is performed to fetch the actual result records.

They implemented the following common query processing operators:

- **Selection.** Without indexes, selection is implemented using the filter primitive. With indexes, if selectivity is high, the filter primitive is also used. Otherwise, B+-tree index or hash index are used.

- **Projection.** Implemented using the gather primitive. If duplicate elimination is required, sorting is used to eliminate the duplicates.

- **Ordering.** Implemented using the sort primitive.

- **Grouping and aggregation.** The sort primitive performs grouping and the reduce primitive performs aggregation.

The join operation is implemented using the following four algorithms:

- **Non-indexed Nested Loop Join (NINLJ).** Both input relations are split into chunks. Each thread group takes two chunks and joins each tuple in one chunk to all tuples in the other.

- **Indexed Nested Loop Join (INLJ).** Creates a CSS-Tree in parallel using one input relation. The tuples of the other relation are then searched in the tree to determine join results. A detailed description can be found in Section 4.2.3.

- **Sort-Merge Join (SMJ).** The two relations are sorted and then merged together. The smaller relation is divided in chunks of size equal to the shared memory. Then, the first and last tuples of each chunk in shared memory are used to determine the corresponding chunks in the other relation. Finally, matching tuples are merged in parallel.

- **Hash joins (HJ).** They implemented a parallel version of the radix hash join. Both relations are split using $log_2(||S||/M)$ radix bits where M is the size of the shared memory. By doing this, the join is transformed into smaller joins over partitions of a relation with its correspondig partition in the other relation. The actual tuple matching is done by storing one partition in shared memory and searching it with binary or sequential searches.

In order to have an effective use of the hardware resources, they created a co-processing scheme that determines where an operator should be evaluated using a cost-based approach. An operator can be evalued in the CPU, in the GPU or in both. To use both CPU and GPU, data needs to be particioned, processed and then merged. In their benchmark, the performance of this co-processing scheme is similar to or better than both a GPU-only and a CPU-only schemes.

Their experiments also show that, while the GPU provides a good performance increase for most cases, simple queries are much slower in the GPU than in the CPU. This is mainly because data transfer time between host memory and device memory. The evaluation of the four types of join algorithms determined that the INLJ is the best in terms of performance.

Red Fox

Red Fox [24] is an upcoming compilation and runtime environment for data warehousing applications on GPU clusters. It is being developed in collaboration with NVIDIA and LogicBlox. Its main elements are a LB-Datalog front-end, the Harmony [73] run-time and the Ocelot [74] compiler.

While not yet completed, they have already implemented the relational algebra operators using algorithm skeletons. These skeletons allow easy adaptation of algorithms to different data types. They are executed in Cooperative Thread Arrays (CTAs), each of them mapped to a GPU's Streaming Multiprocessor. Input data is partitioned according to the number of CTAs available.

Relations are stored in arrays of compressed tuples which are sorted according to the tuple attributes. To compress tuples, the first bits are used to store the tuple key, the next bits store the value and the final bits are used as padding to have complete words. This allows good array partition and tuple search. The algorithm skeletons maintain this sorting through the execution. Their relational algebra skeletons are as follows:

- **Projection.** Decompresses the tuples, removes the unnecessary ones and recompresses them.
- **Product.** Since the size of the result is the product of the inputs, the product is implemented in a single pass. This pass combines the tuples and writes them to memory.
- **Selection.** It is divided in three kernels. The first kernel reads the input and marks with 1 all elements that comply with the selection condition. Once a CTA has finished marking, it performs a local prefix sum to determine the result size and to store the results in the correct positions. The second kernel performs a global prefix sum to, once again, determine total result size and result positions. The last kernel takes the partial results generated by each CTA and writes the final result.

- **Set Operators.** Include Set Intersection, Set Union and Set Difference. They are divided into three stages. The first stage partitions the data using a parallel implementation of the double-sided binary search by Baeza-Yates et al. [75]. The second stage implements a merge operation that loads two partitions and compares their elements according to the operation being performed. Once all elements are compared, a partition is swapped with a new one and the process repeated. Any resulting elements are stored in shared memory and offloaded to global memory when a threshold is reached. The last stage takes all partial results and writes them to a contiguous array.

Their join operation is similar to the set operations in the first and last stages. For the second stage (merge) they present the following strategies:

- **Binary-Search.** Each thread takes a tuple from a relation and computes its lower and upper bounds in the other relation. This approach presents bank conflicts and instruction replays.

- **Register Blocked Binary-Search.** Improves the binary-search by having each thread access a set of consecutive tuples to obtain their corresponding upper and lower bounds.

- **Brute-Force.** Compares each tuple in a relation against all the tuples in the other relation. It has terrible computational complexity but is highly parallel.

- **Hash Join.** Implements a hash table in shared memory with one relation. Then the tuples of the other relation are searched in the hash table. Any collisions during creation of the table are marked and solved by repeating the process without the entries that where correctly inserted. Once marking is finished, a local prefix sum is performed to determine the result size and to store the results in the correct positions. The second kernel performs a global prefix sum to, once again, determine total result size and result positions. The last kernel takes the partial results generated by each CTA and writes the final result.

- **Join-Network.** Implements a comparator network. In order to reduce the number of comparisons, side-exits are used. The idea is to remove groups of tuples from the network when the comparison of their pivot elements determines that they can be joined.

To improve the performance of relational algebra operators, they implemented kernel fusion and fission. These optimisations are implemented automatically by transforming the source code before sending it to the compiler. They can also be performed together: fission can be applied to a fused kernel or vice versa.

Kernel fusion is the transformation of two or more kernels into a single kernel. The idea is to reduce global memory access by reading input data only once in the fused kernel instead of one read per kernel. Other benefits include reduction in memory traffic and more available memory thanks to the reduction of intermediate data. However, not all kernels can be fused together and fusing too many kernels may downgrade performance due to the limited registers and shared memory of the GPU. Some of the commonly fused relational algebra kernels include: selections, joins, selection then projection, join then projection, etc.

Kernel fission is the partition of a single kernel into two or more smaller kernels. This approach hides memory transfer times by processing one kernel in the GPU while the results of another are being offloaded to the CPU. The main disadvantage of kernel fission is that two concurrent kernels have access to half the computational resources each. For operations over big input data, performance decreases when compared to the single kernel approach.

They evaluated their relational algebra operators against GPUQP. Their results show that selection is 3.54x faster than GPUQP and join is 1.69x faster. The fastest join algorithm was the Join-Network because all comparisons and move operations are determined statically, thus allowing many optimisations to be performed like loop unrolling and proper shared memory access. The evaluation of kernel fusion and fission show good performance increase using kernel fusion and a slightly better increase using both fusion and fission.

4.5.1 Comparison with our work

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; but 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* compared their join algorithm to that of GDB in [60], showing a 1.69x performance improvement. Since our join is based on that of GDB, similar results are likely. However, our algorithm works for joins over more than two columns. Selection is slightly different; our algorithm performs only one prefix sum instead of two, and the result is written once instead of written and then moved to eliminate gaps.

They discuss kernel fusion and fission in [69]. 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.

4.6 Summary

Our Datalog engine is implemented into three stages. In the first stage, a Datalog program is compiled and stored in memory with an appropriate numeric format, followed by a preprocessing of the rules to determine which operations to perform and over which columns. Finally, a queue of rules is created based on the query.

The second stage corresponds to the evaluation of the rules with a bottom-up approach using the following relational algebra operations:

- **Selection.** It is performed by three kernels, one marks the tuples that match the selection values, another performs a prefix sum to determine key positions, and the last one writes the result.

- **Joins.** Single and multijoins are performed by building a CSS-Tree with one of the tables to join and searching for join positions over this tree. Selfjoin is similar to the selection, with the difference that columns are compared one against the other (instead of comparing columns against given values).

- **Projection.** It is performed as part of the single and multijoin operations by writing in the result only the necessary columns and discarding the rest.

The third stage involves removing from the queue those rules which will not yield new facts if evaluated again. If the queue does not become empty, each rule in it is evaluated again. This process is repeated until the rules queue becomes empty. Once it is empty, we answer the query and return each column in the answer to its original string form.

To improve the performance of our engine, several optimisations where made like duplicate elimination, additional projections, etc.

Chapter 5

Experimental Evaluation

This chapter 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 the evaluation of Datalog queries, as opposed 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 the related work discussed in Section 4.5.1.

Performance is the main focus of our work. However, we have briefly considered other metrics:

- **Energy consumption.** Due to their higher bandwidth and number of processors, GPUs consume much more energy than CPUs. As an example, the GPU used in this work consumes a maximum of 244W, while the used processor consumes a maximum of 95W. However, for highly parallel applications, the shorter execution time on the GPU actually reduces the overall energy consumption when compared to CPU only implementations [76]. In our work, for 2 of the queries the execution time was greatly reduced and thus, executing this queries on the GPU is an energy efficient solution. The other 2 queries showed little or no performance gain, so using the GPU is not energy efficient. As a result, we suggest in Section 6.2 a rule analyser that evaluates the characteristics of each rule (input size, number of subgoals, etc.) and sends them to the appropriate processing unit.

- **Memory efficiency.** As mentioned in Section 2.4, memory efficiency is very important for a GPU application. We have applied, to the best of our knowledge, all possible memory optimizations techniques. For this reason, we believe that the poor performance on two of our queries is not a memory efficiency problem, but rather a computational one as discussed in Section 5.4.

On a single CPU in the host platform, 4 queries were run with the Prolog systems YAP [22] and XSB [21], and the Datalog system from the MITRE Corporation [77].

As the four queries showed the best performance with YAP, our results plots below 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) [21], extended with some optimizations to improve performance.

5.1 Experimental Platform

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.

Our Datalog engine is written in CUDA C with some data structures (lists and vectors) borrowed from the Standard Template Library [30] of C++. It also includes various functions from the CUDA Thrust library [29].

5.2 Applications

To evaluate our engine, we executed four Datalog programs that test different aspects of our engine.

5.2.1 Join over four big tables

This application is our design and its purpose is to test all the different operations of our 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)?
```

The operations are tested in the following parts of the rule:

- The join over `X` between `table1` and `table2` is a single join operation.
- The `4` in `table2` is a selection.
- The two `Z`s in `table3` represent a selfjoin.
- The join over `Y` and `Z` between `table3` and `table4` represents a multijoin.
- The head of the rule (`join`) represents a projection to remove `Y` and maintain `X` and `Z`.

We used four tables with 1, 3 and 5 million rows each filled with random numbers.

5.2.2 Path Finder

Path finder is a recursive query and a very good example of the power of Datalog to compute the transitive closure of a graph (TCG) [78] and, in general, the transitive closure of relations.

Given the set of edges in a directed graph, each connecting two (adjacent) nodes, the application finds 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)?
```

We use a table with two columns and 1, 5 and 10 million rows filled with random numbers that represent the edges of a graph.

5.2.3 Same-Generation program

This is a well-known program in the Datalog literature to compute the transitive closure of a relation. Among the various versions of this application, we use the one described in [38]. 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:

$$up = \{(a, b_i)|i\epsilon[1,n]\} \cup \{(b_i, c_j)|i,j\epsilon[1,n]\}. \tag{5.1}$$

$$flat = \{(c_i, d_j)|i,j\epsilon[1,n]\}. \tag{5.2}$$

$$down = \{(d_i, e_j)|i,j\epsilon[1,n]\} \cup \{(e_i, f)|i\epsilon[1,n]\}. \tag{5.3}$$

Where a and f are two known numbers and b, c, d and e are series of n random numbers. Figure 5.1 taken from [38] shows an example of the input for $n = 2$.

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).
```

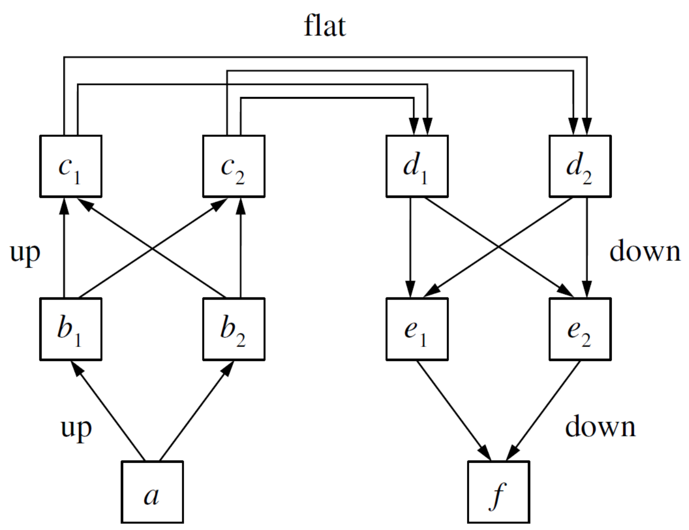

Figure 5.1: Same-Generation input for $n = 2$.

```
sg(a,Y)?
```

The three tables *up*, *flat* and *down* were generated with random numbers for $n =$ 25, 50 and 75. This means 650, 2550 and 5700 rows for tables *up* and *down*; 625, 2500 and 5625 for table *flat*.

5.2.4 Tumour detection

Correctly determining if a tumour is malignant or not requires the analysis and comparison of a great deal of information provided by medical studies. If we consider each characteristic of a tumour as a fact, then it is possible define a rule to determine for each patient, if his or her tumour is malignant or not. The rules and query are defined as follows:

```
is_malignant(A):-
  same_study(A,B),
  'HO_BreastCA'(B,hxDCorLC),
  'MassPAO'(B,present),
  'ArchDistortion'(A,notPresent),
  'Sp_AsymmetricDensity'(A,notPresent),
  'Calc_Round'(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 tumor. However, some of these characteristics are taken from the most recent study, while others must be taken from past studies. This restriction requires the definition of 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`) is used to prevent an study from referencing to itself, thus avoiding incorrect results in the query.

We evaluated this program with 65800 studies. This means that each table is composed of 65800 rows.

5.3 Results

This section presents the results of our engine and compares them against YAP. Both results show the evaluation of each query once all data is in CPU memory —I/O cost is not considered.

5.3.1 Join over four big tables

Figure 5.2 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

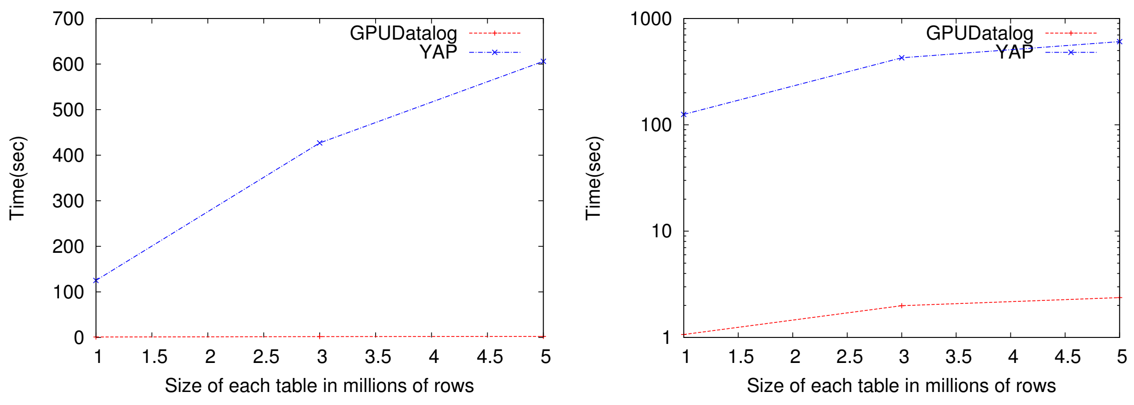

Figure 5.2: Performance of join over four big tables.

faster, roughly 200x times. Note that due to this sheer performance difference, the line representing our engine in the left plot can barely be seen at the bottom.

Both YAP and our engine take proportionally more time as the size of the tables grow. Our engine took just above two seconds to process tables with five million rows each, while YAP took about two minutes process tables with one million rows each.

The time taken by each operation was as follows: joins were the most costly operations with the Multijoin alone taking more than 70% of the total time; the duplicate elimination and the sorting operations were also time consuming but within acceptable values; prefix sums and selections were the fastest operations.

5.3.2 Path Finder

Figure 5.3 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 seconds 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.

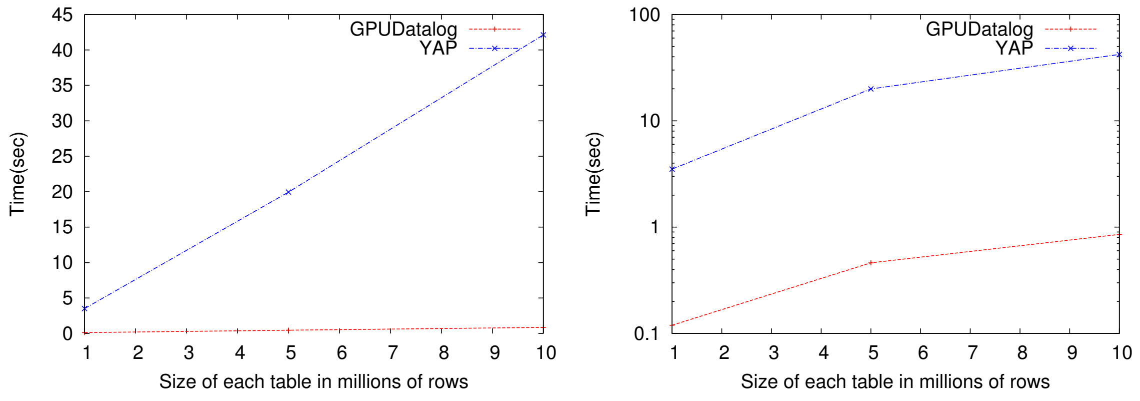

Figure 5.3: Performance of the path finder application.

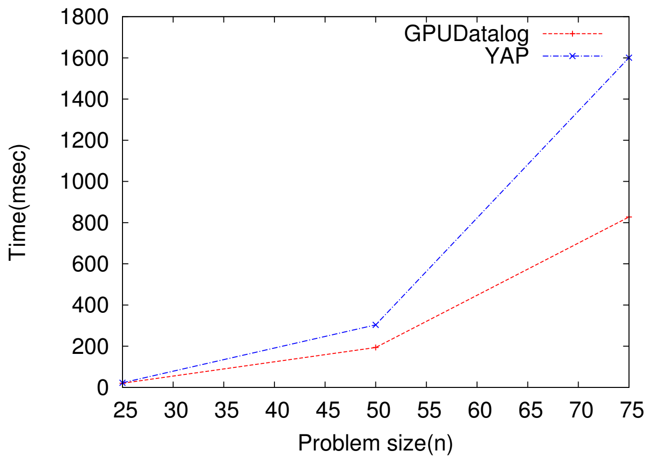

Figure 5.4: Same-Generation program performance.

5.3.3 Same-Generation program

The results show (Figure 5.4) very little gain in performance, with our engine taking an average of 827ms and YAP 1600ms 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 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).

5.3.4 Tumour detection

The performance of YAP and our engine for this program is shown in Figure 5.5. To evaluate this application with different sizes of input data, we duplicated and triplicated each table (e.g. table *'IDnum'* had 65800 rows for the first test, 131600 for the second and 197400 for the last one). This increase in input data allows us to increase processing time while maintaining the same results thanks to duplicate elimination.

Our engine performs best for the first and second evaluations; however, it is surpassed in the final evaluation by YAP with tabling. A detailed analysis of the times of each operation show that the multijoin required by *same_study* consumed almost 90% of the total execution time.

Once again, we have concluded that the main reason for this behaviour is due to duplicates. For example, the problem with thrice the number of tuples (197400 rows in each table) generates 4095036 duplicated rows in the `same_study` function and after duplicate elimination only 50556 rows remain. For the `is_malignant` function the result is similar: 4881384 rows are generated and only 550 remain after duplicate elimination.

As can be seen, duplicates create far too many rows that have to be joined using a multijoin, the slowest of the join operations (it is the slowest because of the additional columns it has to consider). YAP without tabling is so affected by these duplicates that it simply terminates after throwing an error —shown in the plot as zero execution time. In contrast, YAP with tabling avoids performing duplicate work and thus, performs very well.

5.4 Discussion

Our engine performed very well for the first two applications thanks to the following factors:

- The great difference in the number of cores and memory bandwidth between

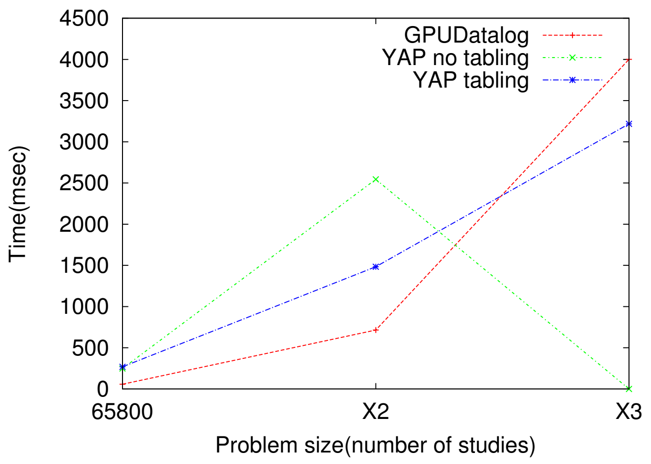

Figure 5.5: Performance of tumour detection.

CPU and GPU.

- The parallel nature of both the relational algebra operators and the support operators, and their finelly tuned implementations.

- The memory management module that reduces memory traffic.

- The translation of strings into numbers and the preprocessing of each rule.

However, the performance of the other two applications was adversely affected by the creation of duplicated tuples each time a join operation was carried out. Worse yet, these duplicates may, in turn, generate many more duplicates, eventually filling all available memory and forcing the application to an erroneous termination. These duplicates also require additional computations to evaluate and store them that ultimately cause a very noticeable performance decrease.

Our approach of eliminating duplicates at the end of the evaluation of a rule is clearly not enough. Duplicates should be eliminated after each join or they should not be generated at all. However, these approaches are not easy to implement or require additional computation that may slow down applications with few or no duplicates. Suggestions for the implementation of these approaches can be found in Section 6.2.

5.5 Summary

We evaluated our engine using four queries and compared our results against the Prolog systems YAP and XSB, and the Datalog system from the MITRE Corporation. All these systems run on a single CPU, while our engine runs in both a single CPU and a GPU.

For the first two queries, our engine shows a dramatic performance increase (between 200x and 40x). However, the other two queries show only a small performance increase. We have determined that duplicates are the cause of these small increases. While our engine eliminates duplicates after each rule evaluation, it is clearly not enough as the number of duplicates greatly increases before the end of the rule is reached. YAP uses a method called tabling to avoid processing duplicates, thus eliminating many unnecessary computations.

A detailed analysis of each operation shows that the joins, specially the multijoins, are the slowest operations, sometimes taking more than 50% of the total execution time. Sorting and duplicate elimination are also very time consuming.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have presented the design and evaluation of a Datalog engine for GPUs. GPUs are high-performance many-core processors capable of very high computation and data throughput. GPUs are programmed using CUDA which extends C to allow the definition of *kernels*, code procedures that are executed in parallel by hundreds of threads within the GPU.

Datalog is a language based on first order logic that has been used as a data model for relational databases; syntactically it is a subset of Prolog. A Datalog program consist 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 as queries.

Our engine evaluates Datalog rules using the relational algebra operations selection, projection and join, with the help of the support operators sort, prefix sum, and duplicate elimination. Rule evaluation is carried out using a bottom-up semi-naive approach, which means that rules are applied to the given facts in order to derive new facts, repeating this process with the new facts until no more facts are derived.

The engine includes a compiler that reads Datalog programs and translates them into a numeric representation easier to work with on GPUs. Once rule evaluation is completed, these numbers are returned to their corresponding strings with the help of a dictionary created during the compiling. After the compiling is done, a preprocessing module analyses the rules to determine which relational operations to perform and on which data elements should they be performed.

The engine also includes a memory management scheme whose purpose is to maintain facts and rule results in GPU memory for as long as possible in order to reduce memory transfers between host and device. When memory allocation is necessary but there is not enough available memory, the module automatically takes the least recently used data and swaps it to the CPU or discards it, depending on the necessity of the data for further calculations. Without our memory management scheme, input data would have to be transferred each time an operation is to be performed and results would have to be sent to the CPU immediately after an operation is completed.

To improve the performance of rule evaluation, various optimisations were performed to the engine. A projection and a duplicate elimination are carried out after each join to eliminate unnecessary columns and tuples. Some operations, e. g. selections and joins with projections, are fused together (performed at the same time) to read the input data only once, instead of reading it for each operation.

The evaluation of our engine using 4 queries shows a dramatic performance improvement when compared against the well known Prolog engines XSB and YAP, and the Datalog engine from the Mitre Corporation. For two of the queries, a performance increase of up to 200 times was achieved. The performance of the same-generation problem is improved twice only, but we believe it can be further improved.

The contributions of this thesis are as follows:

- The design, implementation and evaluation of a new Datalog engine for GPUs, capable of evaluating standard Datalog programs faster than any other CPU Datalog engine.

- A compiler of Datalog programs that translates facts, rules and queries into numbers, which are easier to work with in GPUs due to their constant processing time (strings entail variable processing time due to their variable size).
- A memory management module that maintains data in GPU memory for as long as possible in order to reduce data transfers between CPU and GPU.
- Relational algebra algorithms tuned to exploit the architecture of the GPU thanks to the use of techniques like CSS-Trees, coalesced memory access, etc. They also capitalise on the distinctive features of Datalog rule evaluation which allows the use of simultaneous projections, operation fusing, etc., as described in Section 4.3.
- Publication of an international conference paper describing these contributions.

Clearly, there is room for improvement in most parts of our engine as described in the following section.

6.2 Future Work

Our engine can be extended with the following additions:

Magic Sets

Refer to a logical rewriting method used in many deductive database systems to improve the performance of bottom-up evaluation. It transforms a program by adding new rules which represent the query under consideration. The result of the new program is equivalent to that of the original one. By doing this transformation, the variables in the rules are restricted to take only certain values in a way similar to the top-down approach. This reduces the number of unnecessary facts, and thus the required amount of memory and computations.

This method could be implemented in our engine as part of the preprocessor by doing the rewriting before the analysis of the rules. Since this method changes the way rules are written, and not the way they are evaluated, no additional modifications to the engine should be needed.

Counting

Counting is an extension of the Magic Sets method that complements each element of the magic set with an index. This index represents the "distance" to the goal constants, allowing the evaluation to consider only those tuples that have the correct "distance". While this method further reduces the number of computations, it can only be used in linear programs with acyclic databases, otherwise termination is not guaranteed.

Since it is an extension of the Magic Sets method, it could be implemented into our engine in a similar way.

Built-in predicates

Built-in predicates are special symbols such as $=, \neq, >, <, +$, that force variables to take certain values, e. g. the predicate $Y > 5$ would force Y to only take values greater than 5. They can appear in the right-hand side of a rule and are written in infix notation. Two important restrictions must be enforced when working with these predicates: a) to guarantee a finite output of the Datalog program (also known as safety), each variable involved in a built-in predicate must also appear in a nonbuilt-in predicate of the same rule body; and b) the evaluation of built-in predicates must be delayed until all variables involved are bound to constants, otherwise it is impossible to know which tuples are part of the computation of these predicates.

In our engine, comparison built-in predicates ($=, \neq, >, <, \leq, \geq$) can only be evaluated if they appear at the end of the rule and are performed together using three kernels. Like the selection operator, the first kernel marks the rows that satisfy all the built-in predicates 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. Other built-in predicates ($+, -, *$, etc.) are not implemented.

To better evaluate built-in predicates, they should be implemented as extra conditions for the single and multijoin operations. They could be considered in the joins as follows:

- **Single join.** The extra conditions should be considered in both join kernels, similar to the way the multijoin checks for additional columns. In the first join, they should be considered after counting the number of times an element has to be joined but before writing this result to global memory. The idea is to subtract from this number each time an element fails to meet the extra conditions. In the second join, conditions should be checked before writing an element into the result array. If the conditions are not satisfied, the element is not written.
- **Multijoin.** In the first join kernel, these conditions could be considered along with the additional columns comparison, also reducing the count for element that do not comply. In the second join kernel, they could also be considered at the column comparison to determine if an element is written or not.
- **Selfjoin.** Since the purpose of the selfjoin is to check for equality between columns, the extra conditions could also be checked at the same time the equality is considered.

Any built-in predicate that cannot be seen as a join condition ($+, -$, etc.) would have to be implemented as a complete operator, similar to the selection and join operators.

Negation

To increase its expressive power, Datalog can be extended with the addition of negation in predicates [38, 55]. It is usually represented by the symbol $\neg$ and can appear in both the body and the head of a rule (e. g. `¬A(X) :- B(X, Y), ¬C(Y).`).

In relational algebra, negation is equivalent to the difference operator ($-$). Negation with recursion, as is required by Datalog, is a difficult task with several possible implementations.

To add negation to our engine, our compiler must first be modified to accept the negation symbol $\neg$ and to mark predicates as negative. Stratification analysis could be implemented at preprocessing and rule evaluation could be extended to accept negative predicates. Inflationary and noninflationary semantics could be implemented as part of rule evaluation.

Memory management to handle tables larger than the total amount of GPU memory

To handle tables larger than the total amount of GPU memory, we propose splitting each table into smaller chunks and then applying the necessary operations on these chunks. Basically, with this addition our memory management scheme would become an all-software paged virtual memory.

Selection can be easily split because the size of the results of all three kernels involved are known beforehand, and because each chunk has to be processed only once. Joins and projections (projections are part of the joins in our engine) are harder to split because they involve two tables and each chunk of one table has to be compared against all chunks of the other table. One alternative is to split GPU memory in three, using two sections to store table chunks and the other to store join results, alternating the table chunks and sending the results to CPU memory at the end of each pass. To illustrate this, consider the following example:

Suppose we have *table1* split into chunks (a, b), *table2* split into chunks (x, y, z) and memory sections (*m1*, *m2*, *m3*). We store chunk a in *m1*, chunk x in *m2* and then we perform the join over these chunks, storing the result in *m3*. Next, we send *m3* to CPU memory, store chunk b in *m1*, perform the join and store the result in *m3*. Once again, we send *m3* to CPU memory, store chunk y in *m2*, join and store in *m3*. This process continues until all chunks are processed.

The disadvantage of this approach is the high memory traffic between CPU and GPU. To reduce memory transfer times, streams and page-locked host memory [16] could be used. Streams are sequences of instructions that are executed in order. Instructions from different streams may be executed concurrently depending on the type of instruction and the resources it requires. Page-locked host memory is host memory that will not be paged out by the operating system. It can be used to extend the amount of global memory available to the device, to increase the speed of device to host transfers, among others. To reduce memory traffic, the idea is to create two streams and a section of page-locked host memory, and overlap kernel executions in one stream with memory transfers to page-locked host memory in the other stream.

Mixed processing of rules both on the GPU and the host multicore

Thanks to our bottom-up approach, rules can be evaluated in any order and even in parallel. To extend our engine to capitalize on this, relational algebra operators for the CPU must be implemented. These implementations should take into account the different architecture of the CPU (plenty of memory, fewer processing units, different memory model, etc.) and try to fully capitalise on it.

Once these operators are implemented, we suggest the implementation of a rule analyser that considers the approximate number of input and output tuples and the number of predicates in each rule. The idea is that the GPU processes rules with lots of tuples and the CPU processes rules with few tuples and/or predicates. Since it is possible for a rule on the CPU to require the results of a rule on the GPU or vice versa, the analyser should also try to assign related rules to the same processing unit. Unfortunately, it may not always be possible to do this assignment, so the memory management module should also be extended to keep copies of these results in both memories and to hide memory transfers using streams and page-locked host memory.

Improved join operations to eliminate duplicates

As shown by the evaluation of the third Datalog program in Chapter 6, many duplicate tuples may be inferred with each join operation. To improve performance, these duplicate tuples should be removed as early as possible in the computation. One approach would be to use the duplicate elimination operation after each join. However, duplicate elimination is a costly operation because it involves sorting the table. For queries with few duplicates this approach would actually increase the computation time and does not guarantee an important improvement even in queries with lots of duplicates.

Another alternative would be to perform the duplicate elimination as part of the join operation. However, determining if an element is duplicated would require, to our knowledge, a great deal of comparisons. Therefore, it is recommended to start by investigating efficient implementation of duplicate elimination.

In summary, our work provides the design, implementation and evaluation of a Datalog engine for GPUs. It includes efficient relational algebra operators optimised to capitalize on both the architecture of the GPU and the opportunities the Datalog language allows. It also includes a Datalog compiler, a rule analyser and preprocessing module and a memory management module. Finally, for the four Datalog programs considered in this work, our engine showed a performance increase of up to 200 times when compared to other Prolog and Datalog engines.

Bibliography

[1] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[2] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.

[3] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val 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.

[4] Maurizio 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.

[5] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion 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.

[6] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, New York, NY, USA, October 2005.

[7] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.*, 44(10):243–262, New York, NY, USA, October 2009.

[8] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio 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.

[9] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu 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.

[10] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova 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.

[11] William R. Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham 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.

[12] Trevor Jim. Sd3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 106–, Washington, DC, USA, 2001. IEEE Computer Society.

[13] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell 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.

[14] General-Purpose Computation on Graphics Hardware. `http://gpgpu.org/`, October, 2013.

[15] CUDA. `http://www.nvidia.com/object/cuda_home_new.html`, October, 2013.

[16] CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`, October, 2013.

[17] flex: The Fast Lexical Analyzer. `http://flex.sourceforge.net/`, October, 2013.

[18] Bison - GNU parser generator. `http://www.gnu.org/software/bison/`, October, 2013.

[19] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[20] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Micro-12: 45th International Symposium on Microarchitecture*, Vancouver, BC, Canada, 2012.

[21] XSB. `http://xsb.sourceforge.net/`, October, 2013.

[22] YAP: Yet Another Prolog. `http://www.dcc.fc.up.pt/~vsc/Yap/`, October, 2013.

[23] GPUQP: Query Co-Processing Using Graphics Processors. `http://www.cse.ust.hk/gpuqp/`, October, 2013.

[24] Red Fox: A Compilation Environment for Data Warehousing. `http://gpuocelot.gatech.edu/projects/red-fox-a-compilation-environment-for-data-warehousing/`, October, 2013.

[25] Todd J. Green, Molham Aref, and Grigoris 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.

[26] GPU Applications. `http://www.nvidia.com/object/gpu-applications-domain.html`, October, 2013.

[27] NVIDIA Corporation. `http://www.nvidia.com/page/home.html`, October, 2013.

[28] The Fortran Company. `http://www.fortran.com/`, October, 2013.

[29] Thrust: A Parallel Template Library. `http://thrust.github.io/`, October, 2013.

[30] David R. Musser, Gilmer J. Derge, and Atul 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.

[31] CUDA Downloads. `https://developer.nvidia.com/cuda-downloads`, October, 2013.

[32] NVIDIA CUDA Compiler Driver NVCC. `http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html`, October, 2013.

[33] GCC, the GNU Compiler Collection. `http://gcc.gnu.org/`, October, 2013.

[34] PTX: Parallel Thread Execution. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html, October, 2013.

[35] CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, October, 2013.

[36] Advanced CUDA Webinar: Memory Optimizations. http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf, October, 2013.

[37] U. Nilsson and J. Małuszyński. *Logic, programming, and Prolog.* John Wiley, 1995.

[38] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[39] Michael Stonebraker and Joseph M. Hellerstein, editors. *Readings in Database Systems, Third Edition.* Morgan Kaufmann, 1998.

[40] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD Conference*, pages 1213–1216, Athens, Greece, 2011.

[41] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Cluster computing, recursion and datalog. In *Datalog*, pages 120–144, 2010.

[42] Lucian Popa, Yannis Velegrakis, Mauricio A. Hernández, Renée J. Miller, and Ronald Fagin. Translating web data. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 598–609, Hong Kong, China, 2002. VLDB Endowment.

[43] Monica Pawlan. *Essentials of the Java Programming Language.* Addison-Wesley Professional, 2000.

[44] XML: Extensible Markup Language. `http://www.w3.org/TR/xml/`, October, 2013.

[45] The Perl Programming Language. `http://www.perl.org/`, October, 2013.

[46] Apache Hadoop. `http://hadoop.apache.org/`, October, 2013.

[47] 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, March 1989.

[48] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts.* Connect, learn, succeed. McGraw-Hill Education, 2010.

[49] Yurek K. Hinz. Datalog bottom-up is the trend in the deductive database evaluation strategy. Technical Report INSS 690, University of Maryland, 2002.

[50] Stefan Brass, Dem Fachbereich Mathematik, Fur Das Fachgebiet Informatik, and Vorgelegte Habilitationsschrift. *Bottom-Up Query Evaluation in Extended Deductive Databases.* PhD thesis, Universitat Hannover, 1996.

[51] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3&4):255–299, 1991.

[52] K. Tuncay Tekle and Yanhong A. Liu. More efficient datalog queries: subsumptive tabling beats magic sets. In *SIGMOD Conference*, pages 661–672, Athens, Greece, 2011.

[53] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, Charleston, South Carolina, 1986.

[54] Stefano Ceri, Georg Gottlob, and Letizia Tanca. Extensions of pure datalog. In *Logic Programming and Databases*, Surveys in Computer Science, pages 208–245. Springer Berlin Heidelberg, 1990.

[55] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. In *In Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, Nashville, Tennessee, USA, 1990.

[56] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, July 1991.

[57] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[58] Domenico Saccà and Carlo Zaniolo. The generalized counting method for recursive logic queries. In Giorgio Ausiello and Paolo Atzeni, editors, *ICDT '86*, volume 243 of *Lecture Notes in Computer Science*, pages 31–53. Springer Berlin Heidelberg, 1986.

[59] Domenico Saccà and Carlo Zaniolo. On the implementation of a simple class of logic queries for databases. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, pages 16–23, New York, NY, USA, 1986. ACM.

[60] Gregory F. Diamos, Haicheng Wu, Ashwin Lele, Jin Wang, and Sudhakar Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical report, Georgia Institute of Technology, 2012.

[61] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational joins on graphics processors. In *SIGMOD Conference*, pages 511–524, Vancouver, BC, Canada, 2008.

[62] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst. (TODS)*, 34(4), New York, NY, USA, December 2009.

[63] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems - design and implementation (3. ed.)*. Pearson Education, 2006.

[64] Jun Rao and Kenneth 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.

[65] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, Singapore, June 2011.

[66] Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2001.

[67] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Gpuqp: query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1061–1063, New York, NY, USA, 2007. ACM.

[68] Gregory Diamos, Haicheng Wu, Jin Wang, Ashwin Lele, and Sudhaka Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. In *PPoPP-13: The 18th Symposium on Principles and Practice of Parallel Programming*, Shenzhen, China, 2013.

[69] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications

for GPUs using kernel fusion/fission. In *IPDPSW-12: IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.

[70] B. He, N.K. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing-Volume 00*, pages 1–12, Reno, Nevada, 2007. ACM.

[71] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[72] CUDPP: CUDA Data Parallel Primitives Library. `https://code.google.com/p/cudpp/`, October, 2013.

[73] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 197–200, New York, NY, USA, 2008. ACM.

[74] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[75] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In SuleymanCenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusoz, editors, *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 400–408. Springer Berlin Heidelberg, 2004.

[76] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[77] Datalog by the MITRE Corporation. `http://datalog.sourceforge.net/`, October, 2013.

[78] Guozhu Dong, Jianwen Su, and Rodney W. Topor. Nonrecursive incremental evaluation of datalog queries. *Ann. Math. Artif. Intell.*, 14(2-4):187–223, 1995.