Red Fox: An Execution Environment for Relational Query Processing on GPUs

Haicheng Wu
Georgia Institute of Technology
hwu36@gatech.edu

Gregory Diamos
NVIDIA
gdiamos@nvidia.com

Tim Sheard
Portland State University
sheard@cs.pdx.edu

Molham Aref
LogicBlox Inc.
molham.aref@logicblox.com

Sean Baxter Michael Garland
NVIDIA
{sbaxter,mgarland}@nvidia.com

Sudhakar Yalamanchili
Georgia Institute of Technology
sudha.yalamanchili@ece.gatech.edu

ABSTRACT

Modern enterprise applications represent an emergent application arena that requires the processing of queries and computations over massive amounts of data. Large-scale, multi-GPU cluster systems potentially present a vehicle for major improvements in throughput and consequently overall performance. However, throughput improvement using GPUs is challenged by the distinctive memory and computational characteristics of Relational Algebra (RA) operators that are central to queries for answering business questions.

This paper introduces the design, implementation, and evaluation of Red Fox, a compiler and runtime infrastructure for executing relational queries on GPUs. Red Fox is comprised of i) a language front-end for LogiQL which is a commercial query language, ii) an RA to GPU compiler, iii) optimized GPU implementation of RA operators, and iv) a supporting runtime. We report the performance on the full set of industry standard TPC-H queries on a single node GPU. Compared with a commercial LogiQL system implementation optimized for a state of art CPU machine, Red Fox on average is **6.48x** faster including PCIe transfer time. We point out key bottlenecks, propose potential solutions, and analyze the GPU implementation of these queries. To the best of our knowledge, this is the first reported end-to-end compilation and execution infrastructure that supports the full set of TPC-H queries on commodity GPUs.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing, relational databases*; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, run-time environments*

General Terms

Design

CGO '14, February 15 - 19 2014, Orlando, FL, USA

http://dx.doi.org/10.1145/2544137.2544166.

Keywords

Relational Query Processing, GPU

1. INTRODUCTION

The enterprise software stack is a collection of infrastructure technologies supporting bookkeeping, analytics, planning, and forecasting applications for enterprise data. The task of constructing these applications is challenged by the increasing sophistication of the analysis required and the enormous volumes of data being generated and subject to analysis. Consequently, there is a growing demand for increased productivity in developing applications for sophisticated data analysis tasks such as optimized search, probabilistic computation, and deductive reasoning. This demand is accompanied by the exponential growth in the target data sets and commensurate demands for increased throughput to keep pace with the growth in data volumes. While programming languages and environments have emerged to address productivity and algorithmic issues, the ability to harness modern high performance hardware accelerators such as Graphics Processing Units (GPUs) is nascent at best. This is impeded in large part by the semantic gap between programming languages, models, and environments designed for productivity, and GPU hardware optimized for massive parallelism, speed, and energy efficiency.

Towards this end we propose a system called Red Fox that combines the productivity of declarative languages with the throughput performance of modern high performance GPUs to accelerate relational queries. Queries and constraints in the system are expressed in a high-level logic programming language called LogiQL [18]. The relational data is stored as a key-value store [31] to support a range of workloads corresponding to queries over data sets. The target systems are cloud systems comprising high performance multicore processors accelerated with general-purpose graphics processing units (GPGPUs or simply GPUs). Our baseline implementation executes on stock multicore blades that employs a runtime system that parcels out work units (e.g., a relational query over a data partition) to cores and manages out-of-core datasets. The envisioned accelerated configuration employs GPUs attached to the node that can also execute such work units where a relational query is comprised of a mix of relational algebra, arithmetic, and logic operators. The challenge is that while such queries appear to exhibit significant data parallelism, this parallelism is generally more unstructured and irregular than encountered in traditional

scientific computations. The result is significant algorithmic and engineering challenges to harnessing the throughput capabilities of GPUs.

In Red Fox, an application is specified in LogiQL [18], a declarative programming model for database and business analytics applications. This development model for enterprise applications combines transactions with analytics, by using a single expressive, declarative language amenable to efficient evaluation schemes, automatic parallelization, and transactional semantics. The application then passes through a series of compilation stages that progressively lowers LogiQL into primitive operators, primitive operators into computation kernels, and finally kernels are translated into binaries that are executed on the GPU. A set of algorithm skeletons for each operator is provided as a CUDA template library to the compiler. During compilation, the skeletons are instantiated to match the data structure format, types, and low level operations used by a specific operator. The application is then serialized into a binary format that is executed on the GPU by a runtime implementation.

Two major components of the execution addressed by Red Fox are: i) the management of large data sets whose size exceeds available memory; ii) the efficient and effective compilation of relational queries operating over data partitions to make use of the throughput of GPUs. This paper describes our implementation of the latter. The system is evaluated on the full set of TPC-H benchmark queries executing on an NVIDIA GPU. The main contribution of this paper is a solution for effectively mapping full queries and query plans to GPUs and an implementation, demonstration, and evaluation of the solution. More specifically,

1. An extensible compilation and execution flow for executing queries expressed in declarative/query languages on processors implementing the bulk synchronous parallel execution model - specifically GPUs.

2. An engineering design that supports portability across multiple front-end languages and multiple back-end GPU architectures. This is achieved via two standardized interfaces.

   - Query Plan: A query plan format that decouples the language front-end from compiler optimizations making it possible to integrate other language front-ends, e.g., SQL or NoSQL.
   - Kernel IR [10]: An Internal Representation (IR) of operators implemented on the GPU that forms an interface with which to integrate i) machine-specific optimizers and ii) different GPU language implementations for operators, e.g., OpenCL or CUDA (we currently support CUDA).

3. Experiences and insights from the integration of an industrial strength declarative language front-end that can share program analysis information and query plans with an optimizing back-end. Specifically,

   - maintenance of control flow and dependency information for exploitation of parallelism in GPUs.
   - maintenance of functional dependency information for exploitation in the definition and use of keys and indices.
   - relationships between query plan implementations and microarchitectural features such as register files, shared memory, parallelism, etc.

4. The first infrastructure to the best of our knowledge that compiles and executes the complete TPC-H benchmark on GPUs. It is available as an open source backend (query plan to GPU executable).

The following section provides some background information . Section 3 provides an overview of the major modules while the detailed implementation challenges and solutions are provided in Section 4. Section 5 is a presentation and analysis of the results of the implementation of the TPC-H benchmarks on a state of the art NVIDIA GPU accelerator. The paper ends with a description of related work in Section 6 and some concluding remarks in Section 7.

2. BACKGROUND

2.1 Relational Algebra Primitives

Database programming languages are mainly derived from primitive operations common to first order logic. They are also declarative, in that they specify the expected result of the computation rather than a list of steps required to determine it. Due to their roots in first order logic, many database programming languages such as SQL and Datalog can be mostly or completely represented using Relational Algebra (RA) [6]. RA itself consists of a small number of fundamental operations, including PROJECT, SELECT, PRODUCT, SET operations (UNION, INTERSECT, and DIFFERENCE), and JOIN. These fundamental operators are themselves complex applications that are composed of multi-level algorithms and complex data structures. Given kernel-granularity implementations of these operations it is possible to compile many high level database applications into a Control-Flow Graph (CFG) of RA kernels.

RA consists of a set of fundamental transformations that are applied to sets of primitive elements. Primitive elements consist of n-ary tuples that map attributes to values. Each attribute consists of a finite set of possible values and an n-ary tuple is a list of n values, one for each attribute. Another way to think about tuples is as coordinates in an n-dimensional space. An unordered set of tuples of this type specifies a region in this n-dimensional space and is termed a "relation". Each transformation in RA performs an operation on a relation, producing a new relation. Many operators divide the tuple attributes into *key* attributes and *value* attributes. In these operations, the key attributes are considered by the operator and the value attributes are treated as payload data that are not considered by the operation.

2.2 LogiQL

LogiQL is a variant of Datalog [18], with extensions (aggregations, arithmetic, integrity constraints and active rules) to support the development of an entire enterprise application stack: from user-interface controls, to workflow involving complex business rules, to sophisticated analysis over data. The declarative nature of LogiQL is the key reason for its suitability for rapid application development. LogiQL is a logic programming language, where computations over data are expressed in terms of logical relations among sets of data: e.g., conjunctions, implications, etc. Internally, relational data are organized as a key-value store. Compared to

popular imperative programming languages such as Java or C++, LogiQL abstracts away much detail about the actual execution of a program from application developers: developers only specify logical relationships between data. Compared to emerging distributed programming languages for GPUs such as Map-Reduce [14], LogiQL expresses a richer set of relational and database operations that are cumbersome to implement in Map-Reduce.

2.3 General Purpose GPUs

The use of programmable GPUs has appeared as a potential vehicle for an order of magnitude or more performance improvement over traditional CPU-based implementations for large footprint relational query processing. This expectation is motivated by the fact that GPUs have demonstrated significant performance improvements for data intensive scientific applications and the recent emergence of GPU accelerated cloud infrastructures for small and medium enterprises such as Amazon's EC-2 with GPU instances [1]. Current EC2 Quadruple Extra Large GPU instances cost $2.10 per hour, and Quadruple Extra Large CPU instances cost $1.30 per hour. Nominally, it is expected that if GPU implementations can provide significant speedup in excess of 2.1/1.3 relative to CPU implementations, enterprises will have a strong motivation to move to GPU-accelerated clusters if the software stacks can accommodate mainstream development infrastructures - a major motivation for the contributions of this paper. A further motivation for the use of GPU accelerators is their energy efficiency. GPU-based systems occupy top 11 spots in the latest Green 500 list [33].

The current implementation targets NVIDIA GPUs and therefore we adopt the terminology of the bulk synchronous execution model [35] underlying NVIDIA's CUDA language. Figure 1 shows an abstraction of NVIDIA's GPU architecture and execution model. A CUDA application [25] is composed of a series of multi-threaded data parallel kernels. Data-parallel kernels are composed of a grid of parallel work-units called Cooperative Thread Arrays (CTA) which in turn consist of an array of threads that may periodically synchronize at CTA-wide barriers. In the processors, threads within a CTA are grouped together into logical units known as warps that are mapped to single instruction stream multiple data stream (SIMD) units called stream multiprocessors (SMs). Hardware warp and thread scheduling hides memory and pipeline latencies. Global memory is used to buffer data between CUDA kernels as well as to communicate between the CPU and GPU. Each SM has a shared scratch-pad memory with allocations for each CTA and can be used as a software controlled cache. Registers are privately owned by each thread to store immediately used values.

Performance is maximized when all of the threads in the warp take the same path through the program. However, when threads in a warp do diverge on a branch, i.e., different threads take different paths, performance suffers because the execution of two paths is serialized. This is referred to as branch divergence. Memory divergence occurs when threads in a single warp experience different memory-reference latencies and the entire warp has to wait until all memory references are satisfied.

Kernels are compiled to Parallel Thread eXecution (PTX), a virtual Instruction Set Architecture (ISA) that is realized on NVIDIA GPUs [26]. This PTX representation is a RISC virtual machine similar to LLVM [21] that is amenable to classical compiler optimization. Based on CUDA, NVIDIA also distributes an open source template library–Thrust [4] which is very similar to the C++ Standard Template Library (STL) library and provides high performance parallel primitives such as SET operations. While the current implementation is based on CUDA, the use of bulk synchronous parallel model permits relatively straightforward support for industry standard OpenCL [20].

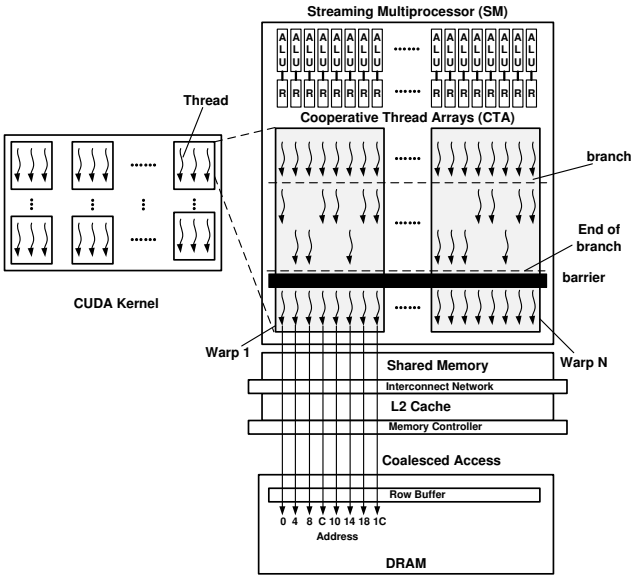

Figure 1: NVIDIA GPU Architecture and Execution Model.

3. SYSTEM OVERVIEW

The major software and hardware modules are described in the following.

3.1 Platform

The current non-accelerated software system executes on stock cloud platforms comprised of multicore blades, e.g., Amazon EC2 [1]. This paper targets such platforms accelerated by NVIDIA GPUs. Our current system compiles queries operating over a data partition (a work unit) and dispatches them for execution on the cores. The longer term vision is to extend this dispatch capability to use GPU accelerators in addition to host cores. This paper only deals with the GPU compilation of queries in the context of this execution model while a discussion of the impact of this capability in the larger system is provided in Section 5.

The results in this paper are reported for a standalone GPU implementation. The overall organization of Red Fox is illustrated in Figure 2. A LogiQL program is parsed and analyzed by the language front-end to produce an IR of the query plan that represents a plan of execution for a set of dependent and interrelated RA and arithmetic operators. The RA-Kernel compiler instantiates the query plan with executable CUDA implementations that are stored in the primitive library and converts it to the Kernel IR [10] which is serialized into the binary that is executed by the runtime.

The two IRs insulate the major components - the front-end, compiler and runtime. This is to support the longer term goal of easier migration to other language front-ends and GPU backends (e.g., OpenCL). Collectively, these com-

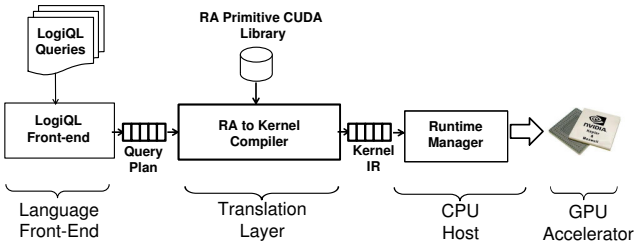

Figure 2: Red Fox Platform Diagram.

ponents implement a complete compilation chain from a LogiQL source file to a GPU binary.

3.2 LogiQL Query

The abstractions and declarative nature of LogiQL enable the generation of high performance evaluation plans for LogiQL programs. A LogiQL program's data flow is made explicit through the logical relations used to define data. Furthermore, well-defined properties associated with logical relations, such as commutativity and associativity of conjunctions, readily expose data parallelism in LogiQL programs. Lastly, the limited number of control operators (e.g., one sequencing operator), makes it easy to construct a finite Data-Flow Graph (DFG) and CFG.

Listing 1 is a simple example of a LogiQL program that classifies even and odd numbers, which will be used throughout this paper. A LogiQL file contains a number of declarations (lines 1, 5, 10, and 14) stating the type of relations and definitions (lines 12 and 15) stating how they are computed. For example, line 1 states that data type of *number* is 32-bit integer. Line 12 states that if m is odd and the next number after m is n, then n is even; similarly, Line 15 expresses that a number next to an even number is itself odd. Together, they provide a **recursive definition** of the two relations. Line 2, 3, 6, 7, 11 explicitly assign initial data to *number*, *next*, and *even*. A LogiQL program starts with the initial data and iteratively derive facts for the other relations until it cannot derive any new facts. Note that if there are multiple rules having the same relation in the head, then the union of the derived data is computed. For example, *even* will contain 0 (as per Line 11) and other even numbers as of Line 12.

```
1 number(n) -> int32(n).
2 number(0).
3 number(1).
4 //other number facts elided for brevity
5 next(n,m) -> int32(n), int32(m).
6 next(0, 1).
7 next(1, 2).
8 //other next facts elided for brevity
9
10 even(n) -> int32(n).
11 even(0).
12 even(n) <- number(n),next(m,n),odd(m).
13
14 odd(n) -> int32(n).
15 odd(n) <- next(m,n), even(m).
```

Listing 1: LogiQL Query Example

The LogiQL front-end has two steps. In the first step, it parses a LogiQL source file into an Abstract Syntax Tree (AST) that stores information about relations and language clauses that operate on them. Clauses that contain multiple

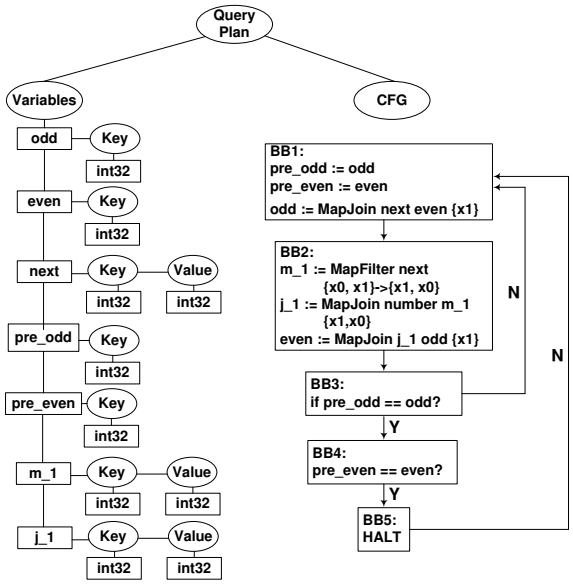

Figure 3: Example of a Query Plan (union of *even* and *odd* is omitted for brevity).

compound operations are broken down into atomic operations that can be executed individually. A list of all relations and their associated types is stored in this representation along with a DFG of operations. In the second step, the front-end translates from the AST into RA operators. It performs a simple mapping from each LogiQL atomic operation to a series of abstract RA operators. Relations are also translated into equivalent types. All of this information is stored in a query plan.

3.3 Query Plan

The query plan is generated by the LogiQL front-end and mainly contains two parts. The first part is the declarations of relations and the second part is a CFG of RA and other (e.g., arithmetic, aggregation, and string) operators.

Figure 3 shows the query plan of the above LogiQL query example. The *variables* part lists all the used tuples (not single scalars) and the data types of each associated attribute. The *CFG* part is comprised of basic blocks and each basic block has several commands. Commands include i) RA and aggregation operations; ii) data movement commands to make a duplicate of a variable; iii) conditional and unconditional jumps to select the next basic block to execute; iv) HALT command to terminate the execution. Most of the work is performed by the relational operations. The MapFilter command is a combination of SELECT, PROJECT and arithmetic/string functions that is used by LogiQL. Similarly, the MapJoin command is a combination of JOIN, PROJECTION and arithmetic/string function. MapFilter and MapJoin will be further decomposed into operators in a later compilation stage. Moreover, recursive definitions are translated into loops in the query plan.

3.4 Kernel IR

The Kernel IR is adapted from [10] and represents a program as a CFG of side-effect-free kernels that operate on managed variables. The compiler maps RA operations to kernels and intermediate data structures to kernel variables. A runtime system is responsible for scheduling kernels on processors subject to control and data dependencies. It is

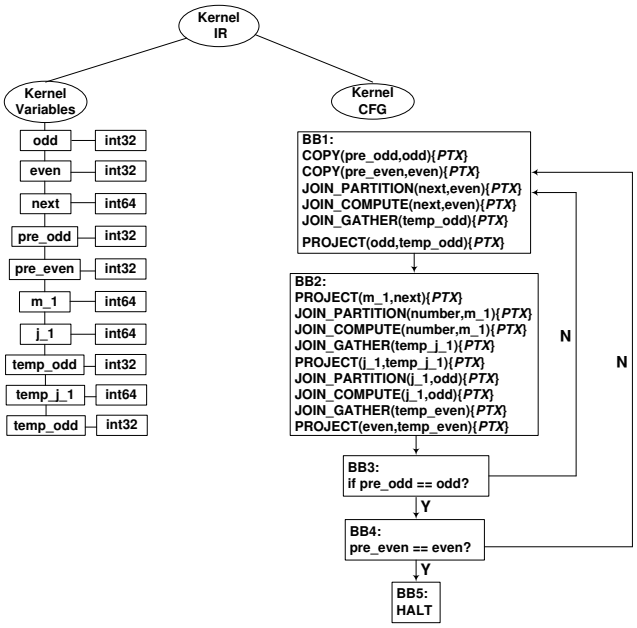

Figure 4: Example of Kernel IR (union of *even* and *odd* is omitted for brevity).

also responsible for materializing variables and moving them between address spaces in a heterogeneous system.

This high level representation lends itself to common program analysis and optimizations. For example, kernels can be scheduled statically subject to dependencies, and memory storage can be time multiplexed between variables using liveness information that directly falls out of the dataflow arcs between producing and consuming kernels.

Kernels can represent arbitrary operations. The compiler specializes skeleton implementations of RA algorithms first into templated CUDA source code, and then into a PTX kernel which is finally embedded in the Kernel IR. Kernels are executed on the GPU hardware by first using the NVIDIA compiler to lower PTX to the native GPU ISA, and then by using the NVIDIA driver to launch the kernel.

Figure 4 is an example of translated Kernel IR. Operators in the query plan are broken into several Kernels. Some temporary variables are added to store the intermediate data.

To support devices other than NVIDIA GPUs, the kernels can be implemented in OpenCL. The back-end runtime can then be modified to launch kernels by calling the OpenCL runtime APIs instead of the CUDA driver APIs. These are engineering rather than conceptual challenges and we are working towards such OpenCL support in Red Fox.

4. IMPLEMENTATION

4.1 LogiQL Front-end

LogiQL front-end translates a LogiQL query into an GPU-executable query plan. Figure 5 shows the compilation flow of the front-end.

In the first stage of the compilation, the front-end parses a LogiQL query consisting of a set of clauses, declarations and constraints and builds an AST. In this process, types are strictly checked after parsing. The strong typing discipline of LogiQL guarantees that well typed LogiQL queries do not fail at runtime due to type errors. A complete set

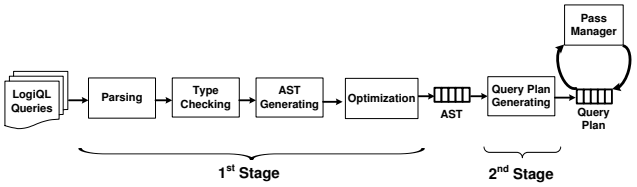

Figure 5: Compilation Flow of Red Fox Front-end.

of type annotations for all LogiQL predicates and variables appearing in the query is added to the original query as a part of this process. In the odd-even classification example, any attempts to derive strings or float data into the integer only predicates (e.g., *odd*) is statically rejected, allowing the query evaluator to optimize the storage representation and selection of primitive operators without the need for the runtime to check at every step that correct values are used. The type checker also performs sound type inference, minimizing the amount of type annotations and declarations needed in the source query while preserving safety.

After type checking, high-level syntactic features (including disjunctive formulas, recursive definitions, complex expressions, automatic primitive operator overloading and conversions) are de-sugared into a core logical language. This language consists of an ordered set of executable logical clauses in dependency/execution order thus specifying high-level control flow. Within clauses, arithmetic and string operators are checked to ensure that no iteration over potentially infinite tuple spaces can occur, guaranteeing termination. Further simplification and optimization steps include common subexpression elimination, clause and predicate inlining, and generation of alternate indexes. The temporary result of the first stage is an intermediate AST containing information about the types and binding sites of variables, and information about control flow and potential parallelism as illustrated in Listing 2.

```
1  Clauses1:
2    number(0). number(1). number(2).
3      number(3). number(4). number(5).
4    next(0,1). next(1,2). next(2,3).
5      next(3,4). next(4,5).
6    even(0).
7
8  Clauses2 {recursive}:
9    for all {int32(n)}
10     odd(n) <- exists {int32(m)}
11       next(m,n), even(m).
12   for all {int32(n)}
13     even (n) <- exists {int32(m)}
14       number(n),next(m,n),odd(m).
15
16 Dependencies:
17   Clauses2 <- Clauses1.
```

Listing 2: AST Example

There are two clauses in Listing 2. The first clause contains data initializations of *number*, *next*, and *even*. The second clause describes how to compute *even* and *odd*. The first clause can be done in parallel since the initializations are independent of each other. The second clause should be computed recursively because *odd* and *even* are mutually dependent. Clause 1 should be computed before Clause 2 because the computation of *even* and *odd* relies on *number* and *next*.

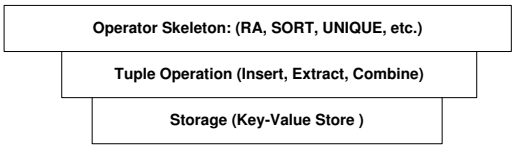

Figure 6: Three Layer Design of RA-Kernel Compiler.

The second stage is to i) map the predicates into tuple formats; ii) translate the clauses into a sequence of RA operations that can run on the GPU. The ordering of Clause1, Clause2, and the implicit control flow of Clause2 is made explicit in a static, single assignment style. Figure 3 is an example of translated query plan before any optimization. In this example, The data initialization part is omitted in the Figure. The recursive part is converted into loops over the section and checks for changes in the output relations after each iteration. Inside each basic block of Figure 3, the operators are ordered to respect the dependence requirements.

The end of the second stage is a pass manager which controls the transformation and analysis passes that run over the query plan IR. Currently, supporting passes include common (sub)expression elimination, several statistical passes and type inference passes which assign types and properties (e.g., uniqueness) to intermediate results. More relational and compiler optimizations will be added in the future.

4.2 RA-Kernel compiler

The RA-Kernel compiler translates a query plan to an executable GPU implementation exported in the Kernel IR format. The core part of the compiler is the primitive library. The job of the rest is to map the variables/operators in the query plan to data structure/CUDA implementation stored in the library. The primitive library, as shown in Figure 6 is comprised of three layers - i) the bottom layer deals with relation storage, ii) the middle corresponds to low level tuple operations that directly operate on tuple data, and iii) the top layer encompasses operator skeletons.

Relations are stored as key-value store. Both keys and values are represented by densely packed arrays of tuples. If the PROJECT operators change the key of a relation, the key tuple array and value tuple array will be reorganized to reflect the change. Figure 7 is an example of physical tuple data layout in the GPU memory. The padding zeros are used to pack the tuple data to the nearest 2^n byte boundary (n is the smallest integer necessary to store the tuple) to align the data storage and ease the system design. The tuple size is templated and the current system supports up to 1024-bit tuples which is very easy to be extended if needed. Strings are stored separately in string tables with several string tables used to store different length strings. Only the string starting addresses are stored in the tuples. For example, if attribute 3 of Figure 7 is a string less than 32 characters, all the string contents in attribute 3 are stored in a string table whose entry has 32 bytes. The pointers to each entry are stored in the tuples rather than the string contents. The entry size of the largest string table is 128 bytes. If the string length is larger than 128 bytes or unknown beforehand, strings are stored in the 128-byte string table and one string might occupy multiple entries. A helper kernel is designed to set up the string tables.

Low level tuple operators are called by the operator skeletons to manipulate tuples. These low level operations partially isolate the algorithm design and data storage and ease

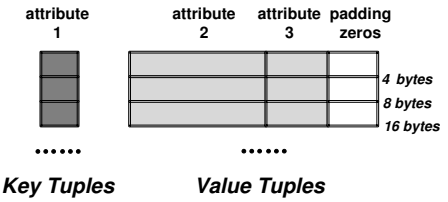

Figure 7: Example of Tuple Storage.

Diamos et al. [9]	SELECT, PROJECT, PRODUCT
ModernGPU [3]	JOIN, SORT
Thrust [4]	SET family, UNIQUE, AGGREGATION

Table 1: Algorithm Sources for Primitives.

modification and optimization. Currently, RA operators uses three tuple operations: i)Insert: insert an attribute into the tuple; ii)Extract: extract an attribute from the tuple; and iii) Combine: combine two tuples by concatenating their value attributes which is used by the JOIN operator.

Finally, as to mapping the operators, a compound operator in a query plan such as MapFilter or MapJoin is decomposed into SELECT or JOIN, PROJECT and arithmetic/string operators. Furthermore, SORT operators are added when some operators need sorted inputs. Currently, JOIN, AGGREGATION, and SET family require sorted inputs because of the chosen algorithm (introduced later in this subsection). Similarly, UNIQUE operators are added when the output data are required to retain uniqueness.

All operators are implemented using various algorithm skeletons that allow the same high level algorithm to be readily adapted to operations over complex data types. This approach is commonly used in compilers for high level domain specific languages such as Copperhead [5], Optix [28].

Currently every operator has one corresponding algorithm implementation. More algorithms will be added in the future. Red Fox uses the algorithms designed by Diamos et al. [9] for PROJECT, PRODUCT, and SELECT. SORT and JOIN algorithms are from ModernGPU library [3] which are both based on the merge path algorithm framework [13]. The JOIN implementation is a variant of sort-merge join optimized for GPUs. As far as we know, all of the above algorithms are one of the most efficient in their categories. They make trade-offs between computation complexity and memory access efficiency and scale well with high throughput. Operators such as SET family, UNIQUE, and AGGREGATION (thrust::reduce_by_key) use implementations from NVIDIA's Thrust library. The remaining operators such as arithmetic (including datetime support), string operations (e.g., string append and substring) are data parallel operations and are re-implemented. Table 1 summarizes the algorithms stored in the primitive library and the source of the implementation.

Algorithm skeletons are CUDA implementations of operators that are templated on the tuple type and possibly the lower level operation type as well (e.g., comparison in SELECT). Once a operator has been mapped to a skeleton, the skeleton is instantiated for the data types of the relation, and the low level operations performed in the case of SELECT and PROJECT. Operators from Diamos et al. and ModernGPU library use the same three stage design (partition, compute, and gather) in the algorithms. Some

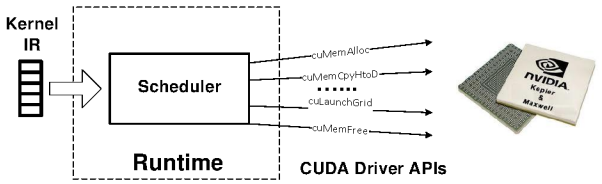

Figure 8: Red Fox Runtime.

operators such as JOIN involve more than one CUDA kernel. So, each operator may finally map to more than one CUDA kernel. The instantiated skeletons are then compiled into PTX format by *NVCC* and stored using the Kernel IR format. Similarly, Thrust library calls are also compiled into binaries by *NVCC* and stored in the Kernel IR. Similar to the pass manager in the front-end, different optimizations (e.g., kernel fusion/fission [36, 37]) can be applied over the Kernel IR.

4.3 Runtime

The runtime is responsible for executing Kernel binaries on a GPU device and data exchange between CPU and GPU. Since the program is represented abstractly in terms of kernels and variables, the runtime has a fair degree of freedom in how it schedules or optimizes kernels and in how it allocates variables. A few potential optimizations are possible, motivated by the observations described in [10]. The runtime is designed to support those optimizations while the current implementation is optimized for deterministic debugging.

Figure 8 shows the structure of the runtime. Operationally, the runtime first raises a kernel binary into a CFG of kernels. Kernels in each basic block in the graph are scheduled using a variant of list scheduling that attempts to minimize memory footprint by scheduling variable definitions and uses back-to-back. During scheduling, dataflow analysis is performed on the program to determine variable live ranges. Since variables represent complex data structures that may change size dynamically, explicit allocate and deallocate operations are inserted before definitions and after final uses respectively. The next step is to interpret the basic blocks in the CFG. When a basic block is selected for execution the first time, the PTX for the kernel is passed to the GPU driver for JIT compilation. The runtime maintains a database of previously compiled kernels and the driver's compiler is invoked only when executing a new kernel. Finally, kernels in the basic block are submitted in-order to the GPU driver for execution. When the block completes, branching code at the end of the block determines the next block to begin executing or to halt the program.

Future versions of the runtime may perform more aggressive optimizations such as concurrent execution of independent kernels, profile and feedback driven scheduling, runtime optimization based on input-data size/distribution, etc.

5. EXPERIMENTAL EVALUATION

The experimental evaluation is conducted by compiling and executing all 22 queries of the TPC-H industry benchmark. TPC-H [8] is a widely used benchmark for decision support systems. It is comprised of a set of business-oriented, complex, ad-hoc queries over large data sets. These queries answer important business questions by analyzing relations between customers, orders, suppliers and products using complex data types and multiple operators on large

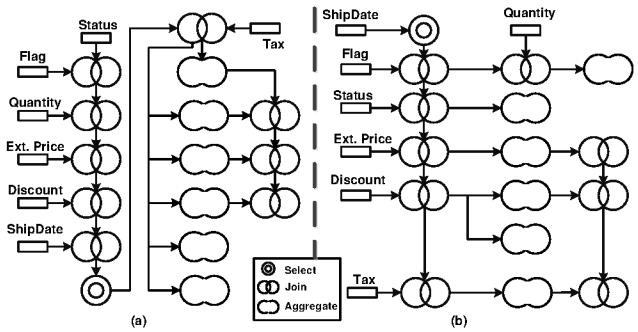

Figure 9: (a) Original Query Plan of Query 1; (b) Optimized Query Plan.

CPU	Intel i7-4771	GPU	GeForce GTX Titan
G++	4.6.3	NVCC	5.5
OS	Ubuntu 12.04	Thrust	1.7
CPU Mem	32GB	GPU Mem	6GB (288.4GB/s)
PCIe	3.0 x16		

Table 2: Red Fox Experimental Environment.

volumes of randomly generated data sets. Each query possesses unique features. For example, Figure 9(a) shows the generated query plan of Query 1 (Q1). Q1 provides a summary pricing report for all products shipped before a given date. Its query plan i) concatenates several variables (e.g., *Status* and *Flag*) into a big table; ii) finds products whose *ShipDate* is before a constant date; iii) groups by *Status* and *Flag* and get pricing statistics.

The input data set used by the following experiments are all generated by the standard TPC-H data generator. The data generator has a parameter, *scale factor*, to control the input data set size. A scale factor of 1 is roughly equal to a 1GB database.

Finally, we note that these queries are representative of industrial strength workloads. The complexity of the compiled queries ranges from 13 operators whose implementations include 56 CUDA kernel for query 13, to 150 operators whose implementations comprise 522 CUDA kernels for query 19.

Table 2 provides the definition of the experimental platform for Red Fox. The NVIDIA GeForce GPU is attached as a device on the host PCIe channel. The overall cost of the system is about 2,400 USD (GPU costs about 1,000 USD).

5.1 Performance Breakdown

Table 3 summarizes the performance of all 22 TPC-H queries for scale factor 1 (SF 1) whose data set size fits the GPU memory. The execution time in column 2 *includes* the PCIe transfer time and the GPU computation time. Column 3 reports the pure GPU computation time. The PCIe transfers only occur at the beginning and the end of the query computation to move the input and output data sets between the CPU and the GPU. The allocated host memory used to buffer the data are pinned memory which can be read/written by GPU at higher bandwidth than pageable host memory. The query results are verified against a CPU implementation. For SF 1, queries take from 0.02 seconds to 0.68 milliseconds to execute. The 22 queries cumulatively take 4.5 seconds including PCIe time with 4.0 seconds spent in GPU computation. Across the 22 queries, PCIe takes

Query #	Execution Time (seconds) GPU (w/ PCIe)	GPU (w/o PCIe)	CPU Parallel	CPU Serial
Q1	0.33	0.28	2.76	18.60
Q2	0.03	0.03	0.41	2.35
Q3	0.19	0.16	2.88	4.74
Q4	0.11	0.09	0.34	2.59
Q5	0.17	0.15	1.19	19.68
Q6	0.17	0.14	0.91	11.50
Q7	0.12	0.09	0.62	4.87
Q8	0.35	0.32	1.17	12.25
Q9	0.27	0.23	2.00	132.7
Q10	0.44	0.42	0.75	9.35
Q11	0.03	0.03	0.27	2.76
Q12	0.25	0.22	1.31	10.54
Q13	0.08	0.06	0.60	2.38
Q14	0.70	0.68	0.82	3.22
Q15	0.09	0.06	0.59	2.11
Q16	0.02	0.02	1.21	4.65
Q17	0.17	0.15	0.19	43.12
Q18	0.04	0.03	0.51	4.86
Q19	0.68	0.63	1.80	40.67
Q20	0.06	0.03	0.27	21.57
Q21	0.18	0.16	2.25	18.32
Q22	0.02	0.02	0.78	2.97
Total	**4.49**	**3.96**	**23.65**	**375.89**

Table 3: TPC-H Performance (SF 1).

11.7% of the total execution time. Several queries (Q15, Q18, and Q20) spend more than 33% of their time in PCIe transfers - motivating pipelined execution of PCIe transfers and GPU computation.

The Power metric and Price/Performance metric are two standard reporting conventions [8] required by the TPC-H organization. The TPC-H power metric [1] (the higher, the better) measures the raw performance. It reports how many queries the system can execute back to back in one hour, i.e. the reverse of the query execution time geometric mean. The value of the power metric for Red Fox is **28,371 QphH@1GB** (w/ PCIe), or **34,402 QphH@1GB** (w/o PCIe). The TPC-H Price/Performance metric (the lower, the better) is the unit cost for performance. For Red Fox, the number is about **0.08 USD/QphH@1GB**, (w/ PCIe), or **0.07 USD/QphH@1GB** (w/o PCIe). It is difficult to directly compare these values to commercial benchmark records [7] since i) they report aggregate values across full systems (min 100GB) and ii) this paper is about the effectiveness of mapping relational queries to utilize the compute throughput GPUs. Therefore we report speedups relative to a single node implementation of the commercial LogiQL system in Section 5.2, i.e., the accelerated system. The last two columns of Table 3 lists the TPC-H performance of a CPU-based system which will be discussed in Section 5.2.

Due to space limitations, we cannot provide a full characterization for each query. Instead, we provide the overall performance analysis and use some queries as examples. The performance breakdown of each query will be listed in our website together with their query plans.

Figure 10 shows the frequency of occurrence of each primitive (top part) and execution time breakdown (bottom part) across all the queries. The bar "others" includes arithmetic operation, string operation, string table setting up, etc. SORT is split into two parts: SORT_JOIN and SORT_AGG. The former is the sorting before the JOIN and the latter is the

[1]TPC-H Power@Size = 3600 * SF / $\left(\prod_{i=1}^{22} T_{Qi}\right)^{1/22}$ [8].

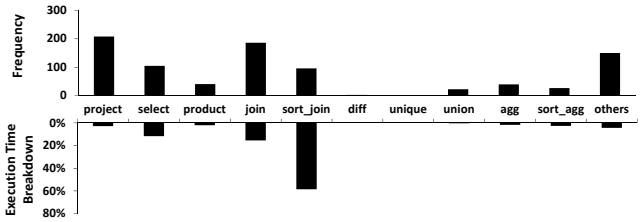

Figure 10: Operator Frequency (top) and Performance Break Down (bottom).

sorting before AGGREGATION. It should be noted that Q14 and Q19 have more operators and longer execution time than the other queries so that they take a higher portion of impact in Figure 10 than the other queries. Overall, the most widely used operator is PROJECT, followed by JOIN, and SELECT. SORT_JOIN is called about half as frequent as JOIN because one or both inputs of the JOINs are already sorted. This also happens to AGGREGATION and SORT_AGG. The least occurring operators are SET DIFFERENCE and UNIQUE since only a few queries use them. SET INTERSECTION is never called.

Each bar in the bottom part of Figure 10 represents the percentage of time spent in an operator across all 22 queries. Overall, most of the execution time is spent in SORT_JOIN (59%), JOIN (15%) and SELECT(12%). Therefore these operators deserve more attention in algorithm optimization on GPUs. Especially for the sort-merge join algorithm used in the paper, the sorting part takes much longer time than the merging part. Furthermore, although PROJECT and SELECT are used frequently, the percentage of execution time is relatively small. They are not computationally intensive. Consequently early ordering of SELECT and PROJECT operators in a query plan can significantly reduce the run time of downstream operators like JOIN and SORT by pruning data set sizes while also being computationally simple and embarrassingly parallel (across tuples).

5.2 Performance Comparison

The execution performance on GPUs is compared to that of the commercial LogiQL implementation, LogicBlox 4.0 on CPUs [18]. We use stock compilation without specific optimizations manually or otherwise targeted to TPC-H. We execute all 22 queries in one Amazon EC2 instance cr1.8xlarge [1] (2× Intel Xeon E5-2670, 16 cores in total) with 32 threads to parallelize the query processing. The overall cost of this instance is about 6,000 USD excluding the network and software cost, which is 2.5x as expensive as the tested GPU system. Two Xeon E5-2670 CPUs cost about 3,000 USD which is three times of the price of the GTX Geforce Titan card. The theoretical memory bandwidth of the Xeon CPU is 51.2 GB/s which is about 17% of that of the GPU device used in the evaluation.

The fourth column of Table 3 lists the absolute execution time of the commercial system. Figure 11 shows the relative speedup of individual queries achieved by Red Fox compared with this system for SF 1 databases. It should be noted that the baseline commercial system employs novel optimizations that produces very efficient and often optimal or near optimal query plans for CPUs. The query plans produced by GPU are *not* fully optimized and do not employ such industrial strength query plan optimizations. Thus, we view the speedup estimates as conservative. Section 5.3 discusses

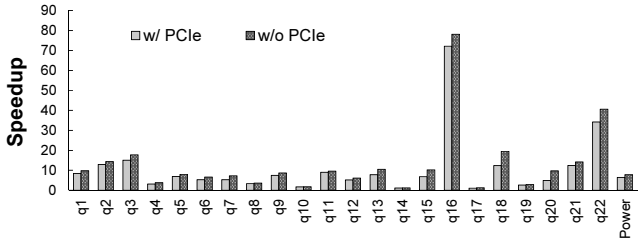

Figure 11: Comparison between Red Fox and parallel build of LogiQL (SF 1).

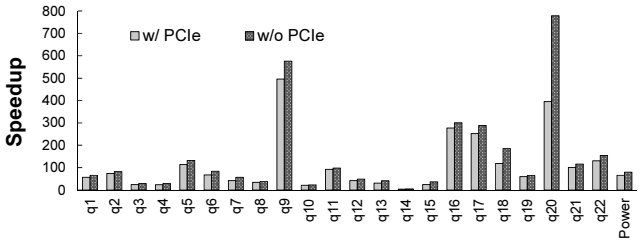

Figure 12: Comparison between Red Fox and sequential build of LogiQL (SF 1).

in greater detail aspects of improvements for GPUs that we have learned as a consequence, and which can produce additional factors of performance improvement.

In Figure 11, GPU performs better than CPU for all queries. Q16 and Q22 have higher speedup in GPU than the other queries. One common feature of these two queries that distinguishes them from the rest is that they concentrate on string processing upon a large number of different short strings (less than or equal to 128 bytes). Q16 has three SELECTs doing regular expression searches (string notlike). Q22 focuses on substring and string matching. The CPU system utilizes the STL and BOOST library to perform the string operation. In GPU, tuples are mapped to threads so that each thread performs the required string operation on one string. For example, in the case of regular expression search each thread performs the same search pattern upon its own string. Thus, the GPU threads may follow different code path depends on the string contents which vary a lot. Severe branch and memory divergence are expected. The throughput of normal SELECTs for integer comparison is larger than 100GB/s. However, the throughputs of three SELECTs of Q16 are much smaller, i.e. 22GB/s, 17GB/s, and 5GB/s (the difference is caused by search pattern, string length, string content, etc.). Even then, Red Fox still outperforms the CPU system which is also limited by low branch prediction accuracy and CPU cache misses. Q10, Q14 and Q17 have relatively smaller speedup which is less than 2x. The unoptimized query plan is the main reason. Section 5.3 will analyze the impact in more detail and discuss future improvements.

The last two bars in Figure 11 compare the TPC-H power metric between different execution configurations. The power metric for the parallel CPU system is 4,380 QphH@1GB. Red Fox is **6.45x** faster if including PCIe time or **7.86x** faster if just comparing the processor computation time. As to the TPC-H Price/Performance metric, the difference between the CPU and GPU would be 15.48x or 18.86x including or excluding PCIe. If only considering the cost of the processors, GPU is 19.35x or 23.58x more cost efficient.

For completeness, Figure 12 includes the performance comparison between Red Fox and the *sequential* build of LogiQL 4.0 which runs one query on a single core of CPU because often database systems map one query to one thread when concurrently processing queries, i.e., in throughput optimized CPU designs. The last column of Table 3 is the raw performance data of the sequential CPU system. The CPU configuration of the sequential experiment is an Intel i7-920 with 12GB memory. Please note that the CPU is not as powerful as the server class CPU used in the parallel experiment. Overall on average, Red Fox is 65.92x faster with PCIe or 79.94x faster without PCIe.

5.3 Analysis and Future Improvement

In terms of opportunities for improvement, we first note that Red Fox currently does not include some standard optimizations utilized in the database community. Simply reordering the positions of operators in the query plan can significantly improve the performance. In Figure 9(b), we manually i) move the SELECT operator to the beginning of the query; ii) perform the aggregations as soon as all the data are ready and discard these data immediately after aggregation. The result is that the memory footprint of Q1 was reduced by half (due to reduction in size of intermediate data) and the execution time excluding PCIe transfer becomes 0.16 seconds which is 1.8X faster than the original.

Moreover, SORT is the most time consuming operator that needs to be further optimized. First, grouping the JOIN operations by key attribute can minimize the number of intervening SORTs required. In the three queries that have the lowest speedup against the CPU (Q10, Q14, and Q17), these unnecessary SORTs occupies 40%, 26%, and 44% respectively of their total GPU execution time. Second, some primitive implementations do not require pre-sorting such as the hash join operator [19]. Third, for different data distributions or key value sizes, other SORT algorithms may perform better. For example, radix sort [30] may perform better when sorting small key size arrays.

More broadly, we identify several hardware and software issues that are related to performance.

GPU DRAM System: Simple operators such as SELECT and PROJECT are already memory bound [9]. Complex operators such as JOIN [3] are not completely memory bound yet, but most of the computations address calculations and shared memory accesses, not floating point. Improving the SM performance (e.g., shared memory bandwidth, integer instruction throughput) for these operators can improve performance up to the point of saturating memory bandwidth. As to DRAM latency, it is less important than capacity or bandwidth because the contiguous tuple storage and the large amount of data make it relatively easy to hide.

SM Microarchitecture: The instruction mix in database primitives is comprised mainly of integer and load/store instructions with relatively smaller percentage of floating point operations. Control and memory divergence occurs when searching or comparing data but their cost is not as significant as moving data. The goal of increasing the occupancy is to saturate SM utilization or DRAM bandwidth since the basic strategy of RA primitive design is to increase core utilization until SMs are saturated or the operations are memory bound. Thus, the shared memory and register files should be large enough to buffer the computation data especially in the cases when intermediate tuple sizes become large. Consequently, some emphasis on reducing tuple size

when feasible is important for achieving high occupancy for given shared memory and register file sizes.

Data movement: Consider Q1 that reads/writes 20GB from/into GPU memory. Suppose the memory bandwidth is 200GB/s, transferring these data would take 0.1 seconds which is about 1/3 of the total GPU computation time. In relational queries, data movement cost is amplified by the relatively fine grained nature of RA operators. Employing variants of classical loop fusion optimizations to kernels, e.g., multi-predicate join operations, can improve performance.

In this accelerator configuration, the GPU performs operations over partitioned data sets and logically appears as a faster core to the host runtime. However, the runtime must account for the cost of data transfer. Thus, intelligent workload partitioning schemes (CPU vs. GPU) will have to make the decisions as to when accelerator usage is productive as a function of query plan and input data set characteristics. With appropriate changes in cost, the model applies to fused parts where the CPU and GPU share the same memory hierarchy such as in AMD Fusion or Intel's Haswell. Finally, another approach to addressing the memory limitation of discrete parts is the use of global virtual address spaces (e.g., CUDA UVA). In this case, data movement is not managed by the application programmer, but rather by the system software/compiler.

6. RELATED WORK

Previous GPU-related database research [12, 16, 34, 19, 17, 9, 3] focused on implementations of primitives. RA primitives have been difficult to implement on GPUs because of i) irregular structure in the memory access patterns; ii) fine grained parallelism with low arithmetic density (ops/memory access), and iii) non-obvious or low data locality. These previous works achieved several factors of speedup in comparison with their CPU counterparts and some are used in Red Fox.

However, the system level research of executing queries on GPUs is still in the early stage. When executing a complete query, the above problems are amplified because many of primitives are involved. Pioneering early work in GPU database systems was GPUQP, developed by He et al [15] which has started to migrate to OpenCL [40]. However, they were not addressing runtime issues necessary to manage the execution of full scale queries and its interaction with compilation, reporting results for two TPC-H queries. Subsequently, Fang et al. designed several data compression schemes based on GPUQP to reduce the PCIe data transfer for database queries [11]. Bakkum et al. also tried to run full queries on GPUs but with a very different approach [2]. They modified the virtual machine infrastructure of SQLite to use GPUs to execute SQLite opcodes (not RA primitives). While novel, it had not yet matured to complex operators (e.g., not supporting JOIN). Recently, Yuan et al. [39] built a column-store GPU query engine for data warehouse workloads and tested it with Star Schema Benchmark [27]. Their system lacks a runtime system and a flexible infrastructure (e.g., two IRs in Red Fox). Their optimization suggestions are also valuable for improving Red Fox and similar infrastructures. Martinez [22] et al. designed a Datalog engine for GPUs and could run it with simple queries. Compared with their work, LogiQL is more expressive and Red Fox uses more efficient primitive design and can support more complex queries. Rauhe [29] et al. designed a multi-level parallelism framework to execute relational queries in GPUs and tested with only seven TPC-H queries due to the lack of support of functions such as string operations. GPU threads execute the programs adopted from their earlier CPU implementation upon a portion of the data. The results computed by the GPU threads are aggregated inside the CTAs and then across the CTAs by another kernel. Their method did not optimize for GPUs and had problems such as load balance. Sitaridi et al. [32] proposed a method to create optimum query plans for executing selection having compound conditions in GPUs and planed to extend it to other primitives in the future. Their work can be integrated to Red Fox to optimize the front-end. Mostak et al. [23] built a GPU accelerated database system, MapD, specialized for geography map queries. As to using multiple GPU devices running over large amount of data, Young et al. proposed to use a runtime and user-level library, Oncilla, to manage multiple GPU devices' memory for large data warehousing applications [38].

Ngamsuriyaroj et al. [24] reported small scale TPC-H performance on MySQL cluster which is one to two orders of magnitude slower than this work (it is using older processor technology). Finally, while the TPC-H website [7] lists and ranks reported large scale performance, none of the reference platforms use GPUs.

7. CONCLUSION

This paper presents the design of a GPU compiler/runtime framework for a high level declarative language commonly used for database and business analytics applications. The context is that of a cloud system where individual nodes are accelerated with GPUs and where the runtime system is targeted to multicore execution of queries over partitioned data sets and uses the GPU logically as a high speed accelerator core. This paper focuses on the compilation of industrial strength queries represented by the full TPC-H benchmark onto GPUs. Comparison with multithreaded host implementations demonstrates significant computing speedup is feasible for a declarative programming model for database and business analytics. The language is progressively parsed and lowered through a series of RA representations that eventually are mapped to PTX kernels embedded in a Kernel IR that is executed by an implementation of the runtime on a high-end discrete GPUs. We report its performance on the full set of TPC-H queries which to the best of our knowledge is the first such implementation for GPUs. When compared with a CPU-based commercial parallel system, GPU system is also **6.48x** faster if including PCIe or **7.86x** faster if excluding PCIe. We also provide analysis of the performance, the lessons learned and future directions.

8. ACKNOWLEDGMENTS

This research was supported by the National Science Foundation under grant CCF 0905459, by LogicBlox Corporation, by an NVIDIA Graduate Fellowship, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). We would also like to acknowledge the detailed and constructive comments of the reviewers.

9. REFERENCES

[1] Amazon. Amazon elastic compute cloud, 2013.

[2] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. GPGPU '10, pages 94–103. ACM, 2010.

[3] S. Baxter. Modern gpu. `http://nvlabs.github.io/moderngpu/`, 2013.

[4] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. pages 359–372. Morgan Kaufmann Publishers, 2011.

[5] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. PPoPP '11, pages 47–56. ACM, 2011.

[6] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, Jan. 1983.

[7] Council, T.P.P. Tpc-h - top ten performance results - non-clustered. `http://www.tpc.org/tpch/results/tpch_perf_results.asp?resulttype=noncluster`.

[8] Council, T.P.P. Tpc benchmark h, standard specification revision 2.16.0, 2013.

[9] G. Diamos, H. Wu, J. Wang, A. Lele, and S. Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. PPoPP '13, pages 301–302. ACM, 2013.

[10] G. F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. HPDC '08, pages 197–200. ACM, 2008.

[11] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, Sept. 2010.

[12] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. SIGGRAPH '05, 2005.

[13] O. Green, R. McColl, and D. A. Bader. Gpu merge path: a gpu merging algorithm. ICS '12, pages 331–340. ACM, 2012.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. PACT '08, pages 260–269. ACM, 2008.

[15] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[16] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. SIGMOD '08, pages 511–524. ACM, 2008.

[17] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[18] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. SIGMOD '11, pages 1213–1216. ACM, 2011.

[19] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. DaMoN '12, pages 55–62, 2012.

[20] Khronos Group. The OpenCL Specification, version 2.0. November 2013.

[21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04, pages 75–. IEEE Computer Society, 2004.

[22] C. A. Martınez-Angeles, I. Dutra, V. S. Costa, and J. Buenabad-Chávez. A datalog engine for gpus. *CHRISTIAN-ALBRECHTS-UNIVERSITAT ZU KIEL*, page 239, 2013.

[23] T. Mostak. An overview of mapd (massively parallel database). 2013.

[24] S. Ngamsuriyaroj and R. Pornpattana. Performance evaluation of tpc-h queries on mysql cluster. WAINA '10, pages 1035–1040. IEEE Computer Society, 2010.

[25] NVIDIA. Cuda c programming guide, 2012.

[26] NVIDIA. Parallel thread execution isa, 2013.

[27] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2009.

[28] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29:66:1–66:13, July 2010.

[29] H. Rauhe, J. Dees, K.-U. Sattler, and F. Faerber. Multi-level parallel query execution framework for cpu and gpu. In *Advances in Databases and Information Systems*, volume 8133 of *Lecture Notes in Computer Science*, pages 330–343. Springer Berlin Heidelberg, 2013.

[30] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. IPDPS '09, pages 1–10. IEEE Computer Society, 2009.

[31] M. Seeger and S. Ultra-Large-Sites. Key-value stores: a practical overview. *Computer Science and Media*, 2009.

[32] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on gpus. DaMoN '13, pages 4:1–4:8. ACM, 2013.

[33] The Green 500. The green500 list - nov 2013, 2013.

[34] P. Trancoso, D. Othonos, and A. Artemiou. Data parallel acceleration of decision support queries using cell/be and gpus. CF '09, pages 117–126. ACM, 2009.

[35] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[36] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. MICRO '12, pages 107–118. IEEE Computer Society, 2012.

[37] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. IPDPSW '12, pages 2433–2442. IEEE Computer Society, 2012.

[38] J. Young, H. Wu, and S. Yalamanchili. Satisfying data-intensive queries using gpu clusters. SCC '12, pages 1314–. IEEE Computer Society, 2012.

[39] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.

[40] S. Zhang, J. He, B. He, and M. Lu. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment*, 6(12), 2013.