Accelerating SQL Database Operations on a GPU with CUDA: Extended Results

University of Virginia Department of Computer Science
Technical Report CS-2010-08

Peter Bakkum and Kevin Skadron
Department of Computer Science
University of Virginia, Charlottesville, VA 22904
{pbb7c, skadron}@virginia.edu

ABSTRACT

Prior work has shown dramatic acceleration for various database operations on GPUs, but only using primitives that are not part of conventional database languages such as SQL. This paper implements a subset of the SQLite virtual machine directly on the GPU, accelerating SQL queries by executing in parallel on GPU hardware. This dramatically reduces the effort required to achieve GPU acceleration by avoiding the need for database programmers to use new programming languages such as CUDA or modify their programs to use non-SQL libraries.

This paper focuses on accelerating SELECT queries and describes the considerations in an efficient GPU implementation of the SQLite command processor. Results on an NVIDIA Tesla C1060 achieve speedups of 20-70x depending on the size of the result set. This work is compared to the alternative query platforms developed recently, such as MapReduce implementations, and future avenues of research in GPU data processing are described.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; H.2.4 [**Database Management**]: Parallel Databases

Keywords

GPGPU, CUDA, Databases, SQL

1. INTRODUCTION

GPUs, known colloquially as video cards, are the means by which computers render graphical information on a screen. The modern GPU's parallel architecture gives it very high throughput on certain problems, and its near universal use in desktop computers means that it is a cheap and ubiquitous source of processing power. There is a growing interest in applying this power to more general non-graphical problems through frameworks such as NVIDIA's CUDA, an application programming interface developed to give programmers a simple and standard way to execute general purpose logic on NVIDIA GPUs. Programmers often use CUDA and similar interfaces to accelerate computationally intensive data processing operations, often executing them fifty times faster on the GPU [5]. Many of these operations have direct parallels to classic database queries [9, 17].

The GPU's complex architecture makes it difficult for unfamiliar programmers to fully exploit. An effective CUDA programmer must have an understanding of six different memory spaces, a model of how CUDA threads and thread blocks are mapped to GPU hardware, an understanding of CUDA interthread communication, etc. CUDA has brought GPU development closer to the mainstream but programmers must still write a low-level CUDA kernel for each data processing operation they perform on the GPU, a time-intensive task that frequently duplicates work.

At a high level, the GPU is a heavily parallelized processor. While CPUs generally have 1 to 4 heavily optimized and pipelined cores, GPUs have hundreds of simple and synchronized processing units. The programmer assigns work to threads, which are grouped into thread blocks which the GPU assigns to its streaming multiprocessors, which contain groups of CUDA cores. Each thread executes an identical program, and this execution remains synchronous within a thread block unless the execution path within a thread diverges based on its data. This is called the Single-Instruction Multiple Thread (SIMT) programming model. A thread can access data from the high-latency global memory, work within its own memory space, called the register space, and share data with other threads in the thread block using shared memory. Global memory accesses can be accelerated by coalescing accesses among threads or using the texture memory cache.

SQL is an industry-standard generic declarative language used to manipulate and query databases. Capable of performing very complex joins and aggregations of data sets, SQL is used as the bridge between procedural programs and structured tables of data. An acceleration of SQL queries would enable programmers to increase the speed of their data processing operations with little or no change to their

source code. Despite the demand for GPU program acceleration, no implementation of SQL is capable of automatically accessing a GPU, even though SQL queries have been closely emulated on the GPU to prove the parallel architecture's adaptability to such execution patterns [10, 11, 17].

There exist limitations to current GPU technology that affect the potential users of such a GPU SQL implementation. The two most relevant technical limitations are the GPU memory size and the host to GPU device memory transfer time. Though future graphics cards will almost certainly have greater memory, current NVIDIA cards have a maximum of 4 gigabytes, a fraction of the size of many databases. Transferring memory blocks between the CPU and the GPU remains costly. Consequently, staging data rows to the GPU and staging result rows back requires significant overhead. Despite these constraints, the actual query execution can be run concurrently over the GPU's highly parallel organization, thus outperforming CPU query execution.

There are a number of applications that fit into the domain of this project, despite the limitations described above. Many databases, such as those used for research, modify data infrequently and experience their heaviest loads during read queries. Another set of applications care much more about the latency of a particular query than strict adherence to presenting the latest data, an example being Internet search engines. Most high-traffic dynamic websites utilize multiple layers of caching, such that a query that occurs when a user accesses a certain page executes with a redundant and marginally stale set of data. Many queries over a large-size dataset only address a subset of the total data, such as a single column of a table, thus inviting staging this subset into GPU memory. Additionally, though the finite memory size of the GPU is a significant limitation, allocating just half of the 4 gigabytes of a Tesla C1060 to store a data set gives the user room for over 134 million rows of 4 integers.

The contribution of this paper is to implement and demonstrate a SQL interface for GPU data processing. This interface enables a subset of SQL SELECT queries on data that has been explicitly transferred in row-column form to GPU memory. SELECT queries were chosen since they are the most common SQL query, and their read-only characteristic exploits the throughput of the GPU to the highest extent. The project is built upon an existing open-source database, SQLite, enabling switching between CPU and GPU query execution and providing a direct comparison of serial and parallel execution. While previous research has used data processing primitives to approximate the actions of SQL database queries, this implementation is built from the ground up around the parsing of SQL queries, and thus executes with significant differences.

In this context, SQL allows the programmer to drastically change the data processing patterns executed on the GPU with the smallest possible development time, literally producing completely orthogonal queries with a few changes in SQL syntax. Not only does this simplify GPU data processing, but the results of this paper show that executing SQL queries on GPU hardware significantly outperforms serial CPU execution. Of the thirteen SQL queries tested in this paper, the smallest GPU speedup was 20x, with a mean of 35x. These results suggest this will be a very fruitful area for future research and development.

2. RELATED WORK

The preliminary results of this research project were published in and presented at the *Proceedings of the Third Workshop on General-Purpose Computation on Graphics Processing Units* (GPGPU-3), March 14, 2010, under the same title[2]. This workshop was part of the *Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2010).

2.1 GPU Data Mining

There has been extensive research in general data mining on GPUs, thoroughly proving its power and the advantages of offloading processing from the CPU. The research relevant to this paper focuses on demonstrating that certain database operations, (i.e. operations that are logically performed within a database during a query execution) can be sped up on GPUs. These projects are implemented using primitives such as Sort and Scatter, that can be combined and run in succession on the same data to produce the results of common database queries. One paper divides database queries into predicate evaluation, boolean combination, and aggregation functions [17]. Other primitives include binary searches, p-ary searches [25], tree operations, relational join operations [11], etc. An area where GPUs have proven particularly useful is with sort operations. GPUTeraSort, for example, is an algorithm developed to sort database rows based on keys, and demonstrated significant performance improvements over serial sorting methods [16]. One of the most general of the primitive-based implementations is GPUMiner, a program which implements several algorithms, including k-means, and provides tools to visualize the results [12]. Much of this research was performed on previous generations of GPU hardware, and recent advances can only improve the already impressive results.

One avenue of research directly related to production SQL databases is the development of database procedures that employ GPU hardware. These procedures are written by the user and called through the database to perform a specific function. It has been shown using stored and external procedures on Oracle [3] PostgreSQL databases [22] that GPU functionality can be exploited to accelerate certain operations. The novelty of this approach is that CUDA kernels are accessed through a database rather than explicitly called by a user program. These kernels have little coordination with the database query processor, and have no effect on database queries that do not explictly call them as a function.

The most closely related research is *Relational Query Coprocessing on Graphics Processors*, by Bingsheng He, et al. [21]. This is a culmination of much of the previous research performed on GPU-based data processing. Its authors design a database, called GDB, accessed through a plethora of individual operations. These operations are divided into operators, access methods, and primitives. The operators include ordering, grouping, and joining functionality. The access methods control how the data is located

in the database, and includes scanning, trees, and hashing. Finally the primitives are a set of functional programming operations such as map, reduce, scatter, gather, and split. GDB has a number of similarities to the implementation described in this paper, notably the read-only system and column-row data organization, but lacks direct SQL access. In the paper, several SQL queries are constructed with the primitives and benchmarked, but no parser exists to transform SQL queries to sequences of primitives.

This paper's implementation has similar results to the previous research, but approaches the querying of datasets from an opposing direction. Other research has built GPU computing primitives from the ground up, then built programs with these primitives to compare to other database operations. This paper's research begins with the codebase of a CPU-based database and adapts its computational elements to execute on a GPU. This approach allows a much more direct comparison with traditional databases, and most importantly, allows the computing power of the GPU to be accessed directly through SQL. SQL presents a uniform and standardized interface to the GPU, without knowledge of the specific primitives of a certain implementation, and with the option choosing between CPU and GPU execution. In other words, the marginal cost of designing data processing queries to be run on a GPU is significantly reduced with a SQL interface.

To our knowledge, no other published research provides this SQL interface to GPU execution. In practical terms, this approach means that a CUDA thread executes a set of SQLite opcodes on a single row before exiting, rather than a host function managing bundle of primitives as CUDA kernels. It is possible that a SQL interface to the primitives discussed in other research could be created through a parser, but this has not been done, and may or may not be more advantageous for GPU execution. Many primitives such as sort and group have direct analogs in SQL, future research may clarify how an optimal SQL query processor differs when targeting the GPU versus the CPU.

2.2 MapReduce

A new and active area of data mining research is in the MapReduce paradigm. Originally pioneered by Google, it gives the programmer a new paradigm for data mining based on the functional primitives `map` and `reduce` [8]. This paradigm has a fundamentally parallel nature, and is used extensively by Google and many other companies for large-scale distributed data processing. Though essentially just a name for using two of the primitives mentioned in the previous section, we give MapReduce special treatment because it has become a major topic in itself. Research in this area has shown that MapReduce frameworks can be accelerated on multicore machines [28] and on GPUs [20]. Notably, Thrust, a library of algorithms implemented in CUDA intended as a GPU-aware library similar to the C++ Standard Template Library, includes a MapReduce implementation [37].

In some cases, a MapReduce framework has become a replacement for a traditional SQL database, though its use remains limited. The advantage of one over the other remains a hotly debated topic, both are very general methods through which data can be processed. MapReduce requires the programmer to write a specific query procedurally, while SQL's power lies in its simple declarative syntax. Consequently, MapReduce most useful for handling unstructured data. A key difference is that the simplicity of the MapReduce paradigm makes it simple to implement in CUDA, while no such SQL implementation exists. Additionally the limited use of MapReduce restricts any GPU implementation to a small audience, particularly given that the memory ceilings of modern GPUs inhibit their use in the huge-scale data processing applications for which MapReduce is known.

2.3 Programming Abstraction

Another notable vector of research is the effort to simplify the process of writing GPGPU applications, CUDA applications in particular. Writing optimal CUDA programs requires an understanding of the esoteric aspects of NVIDIA hardware, specifically the memory heirarchy. Research on this problem has focused on making the heirarchy transparent to the programmer, performing critical optimization during compilation. One such project has programmers write CUDA programs that exclusively use global memory, then chooses the best variables to move to register memory, shared memory, etc. during the compilation phase [29]. Other projects such as CUDA-lite and *hi*CUDA have the programmer annotate their code for the compiler, which chooses the best memory allocation based on these notes, an approach similar to the OpenMP model [19, 39]. Yet another project directly translates OpenMP code to CUDA, effectively making it possible to migrate parallel processor code to the GPU with no input from the programmer [27]. A common thread in this area is the tradeoff between the difficulty of program development and the optimality of the finished product. Ultimately, programming directly in CUDA remains the only way to ensure a program is taking full advantage of the GPU hardware.

Regardless of the specifics, there is clear interest in providing a simpler interface to GPGPU programming than those that currently exist. The ubiquity of SQL and its pervasive parallelism suggest that a SQL-based GPU interface would be easy for programmers to use and could significantly speed up many applications that have already been developed with databases. Such an interface would not be ideal for all applications, and would lack the fine-grained optimization of the previously discussed interfaces, but could be significantly simpler to use.

2.4 General Data Mining

Individually implemented data processing systems can be thought of as points in a multi-dimensional space, to which we will explicitly define three dimensions. The first dimension is the tradeoff between proceduralness and declarativeness in the query language used, which we will call *PD*, with higher values for more declarativeness. The second dimension is the tradeoff between storing unstructured versus structured data, which we will call *US*, with higher values for more structure. The final dimension is the tradeoff between serial and parallel query execution, which we will call *SP*, with higher values for increased parallelism.

Most modern relational database management systems (RDBMSs), such as SQLite or Oracle, can be described as systems with very high *PD* and *US* values. The *PD*

value derives from the declarativeness of SQL. Even though queries are necessarily translated into a language which can be executed on a CPU, SQL largely removes programmer understanding of the actual serial pattern described in a SQL query. On the opposite end of this dimension lies a query hand-written in serial code. For example, a simple SELECT query can be thought of as a `for` loop that checks rows in a table with an `if` statement. When written in SQL, this action becomes declarative, and thus has a higher *PD* value. The high *US* value is a consequence of the managed and structured way that databases store data, using a custom format to manage data types, and access methods such as hashing or trees to access it efficiently. Over the past few decades many databases have achieved higher *SP* values by executing queries on multiple CPU cores and on distributed systems. The contribution of this paper is to create a system with a very high *SP* value by executing on the GPU, while retaining the *PD* value of SQL access.

Query language declarativeness is an extremely useful way to handle both the structuredness of data in RDBMSs and the parallelism provided by modern hardware. The more general and declarative a query language is, the less the programmer needs to know about how the data is stored and how the query is executed. This paper shows that the parallelism of the GPU can be accessed surprisingly well through a declarative language: SQL.

The proliferation of large data-driven web applications such as the Google search engine and Facebook has led to demand for data systems with *PD*, *US*, and *SP* values not found in RDBMSs, hence the popularity of the MapReduce paradigm [7]. The Google search engine, for instance, must locate a phrase within billions of unstructured documents and rank the results, a difficult and inefficient operation on a relational database. The most popular MapReduce system, Hadoop, represents a point with much lower declarativeness and data structure, but very high parallelism [18]. There has been a significant amount of research, all within the past six years (The first paper on MapReduce was released in 2004[8]), on developing systems that lie between Hadoop and RDBMSs in terms of declarativeness, parallelism, and data structure.

One such research area currently under heavy development implements layers on top of Hadoop to provide a declarative interface for querying. Promoted by Facebook, the Hive project implements HiveQL, a declarative language that intersects with SQL, and provides ways to structure data [38]. Pig is a similar system developed by Yahoo! [7, 14, 32]. These query languages have syntaxes similar to SQL, but include keywords such as `MAPREDUCE` to bridge the gap to the Hadoop platform. A related but independent project, HadoopDB, extends Hive by creating local databases at each MapReduce node and managing them at a head node [1]. Under this model SQL queries are parsed to a Hadoop MapReduce format, then transformed back to SQL at each node. Yet another project called SQL/MapReduce creates a hybrid system by implementing MapReduce with SQL user defined functions [13]. This approach is expanded in a paper that points out the deficiencies of UDFs, and instead proposes Relation-Valued Functions, which would allow closer coordination between user-written code and the database through an API, and enable optimization such as user-defined caching [6].

This area has seen an unusual amount of proprietary corporate development, with many large organizations demanding solutions that exactly fit their needs. IBM has developed hybrid systems that rely on their System S stream processing platform. This research examined a similar hybrid query language, but more interesting is their code-generation query processor, which optimizes the compiled query output for the actual hardware and network architecture of their machines [15, 26]. Microsoft's entries in this field rely on their Dryad platform, which like System S, breaks jobs into an acyclic graph of pieces that can be mapped to hardware sequentially and/or in parallel [23]. Microsoft's SCOPE is a partially declarative language, like Pig, while the DryadLINQ language is focused more on parallelizing procedural programs for distributed graph execution [4, 24].

The diversity in this area demonstrates the wide range of data-processing systems and query languages that are possible. Platforms should not just be grouped into RDBMSs and MapReduce systems. Given the very heavy research and development this area has received in the past few years, we expect further major advances soon.

3. SQLITE

3.1 Overview

SQLite is a completely open source database developed by a small team supported by several major corporations [33]. Its development team claims that SQLite is the most widely deployed database in the world owing to its use in popular applications, such as Firefox, and on mobile devices, such as the iPhone [35]. SQLite is respected for its extreme simplicity and extensive testing. *Unlike most databases which operate as server, accessed by separate processes and usually accessed remotely, SQLite is written to be compiled directly into the source code of the client application.* SQLite is distributed as a single C source file, making it trivial to add a database with a full SQL implementation to a C/C++ application.

3.2 Architecture

SQLite's architecture is relatively simple, and a brief description is necessary for understanding the CUDA implementation described in this paper. The core of the SQLite infrastructure contains the user interface, the SQL command processor, and the virtual machine [34]. SQLite also contains extensive functionality for handling disk operations, memory allocation, testing, etc. but these areas are less relevant to this project. The user interface consists of a library of C functions and structures to handle operations such as initializing databases, executing queries, and looking at results. The interface is simple and intuitive: it is possible to open a database and execute a query in just two function calls. Function calls that execute SQL queries use the SQL command processor. The command processor functions exactly like a compiler: it contains a tokenizer, a parser, and a code generator. The parser is created with an LALR(1) parser generator called Lemon, very similar to YACC and Bison. The command processor outputs a program in an intermediate language similar to assembly. Essentially, the command

processor takes the complex syntax of a SQL query and outputs a set of discrete steps.

Each operation in this intermediate program contains an opcode and up to five arguments. Each opcode refers to a specific operation performed within the database. Opcodes perform operations such as opening a table, loading data from a cell into a register, performing a math operation on a register, and jumping to another opcode [36]. A simple SELECT query works by initializing access to a database table, looping over each row, then cleaning up and exiting. The loop includes opcodes such as `Column`, which loads data from a column of the current row and places it in a register, `ResultRow`, which moves the data in a set of registers to the result set of the query, and `Next`, which moves the program on to the next row.

This opcode program is executed by the SQLite virtual machine. The virtual machine manages the open database and table, and stores information in a set of "registers" [1]. When executing a program, the virtual machine directs control flow through a large switch statement, which jumps to a block of code based on the current opcode.

3.3 Usefulness

SQLite was chosen as a component of this project for a number of reasons. First, using elements of a well-developed database removes the burden of having to implement SQL query processing for the purposes of this project. SQLite was attractive primarily for its simplicity, having been developed from the ground up to be as simple and compact as possible. The source code is very readable, written in a clean style and commented heavily. The serverless design of SQLite also makes it ideal for research use. It is very easy to modify and add code and recompile quickly to test, and its functionality is much more accessible to someone interested in comparing native SQL query execution to execution on the GPU. Additionally, the SQLite source code is in the public domain, thus there are no licensing requirements or restrictions on use. Finally, the widespread adoption of SQLite makes this project relevant to the industry, demonstrating that many already-developed SQLite applications could improve their performance by investing in GPU hardware and changing a trivial amount of code.

From an architectural standpoint, SQLite is useful for its rigid compartmentalization. Its command processor is entirely separate from the virtual machine, which is entirely separate from the disk i/o code and the memory allocation code, such that any of these pieces can be swapped out for custom code. Critically, this makes it possible to reimplement the virtual machine to run the opcode program on GPU hardware.

A limitation of SQLite is that its serverless design means it is not implemented to take advantage of multiple cores. Because it exists solely as a part of another program's process, threading is controlled entirely outside SQLite, though it *has* been written to be thread-safe. This limitation means that there is no simple way to compare SQLite queries executed on a single core to SQLite queries optimized for multicore machines. This is an area for future work.

[1]SQLite registers are logical stores of data used in the intermediate opcode language. They should not be confused with CUDA registers, which are the physical banks used to store memory scoped to a single thread.

4. IMPLEMENTATION

4.1 Scope

Given the range of both database queries and database applications and the limitations of CUDA development, it is necessary to define the scope of of this project. We explicitly target applications that run SELECT queries multiple times on the same mid-size data set. The SELECT query qualification means that the GPU is used for read-only data. This enables the GPU to maximize its bandwidth for this case and predicates storing database rows in row-column form. The 'multiple times' qualification means that the project has been designed such that SQL queries are executed on data already resident on the card. A major bottleneck to GPU data processing is the cost of moving data between device and host memory. By moving a block of data into the GPU memory and executing multiple queries, the cost of loading data is effectively amortized as we execute more and more queries, thus the cost is mostly ignored. Finally, a 'mid-size data set' is enough data to ignore the overhead of setting up and calling a CUDA kernel but less than the ceiling of total GPU memory. In practice, this project was designed and tested using one and five million row data sets.

This project only implements support for numeric data types. Though string and blob types are certainly very useful elements of SQL, in practice serious data mining on unstructured data is often easier to implement with another paradigm. Strings also break the fixed-column width data arrangement used for this project, and transferring character pointers from the host to device is a tedious operation. The numeric data types supported include 32 bit integers, 32 bit IEEE 754 floating point values, 64 bit integers, and 64 bit IEEE 754 double precision values. Relaxing these restrictions is an area for future work.

4.2 Data Set

As previously described, this project assumes data stays resident on the card across multiple queries and thus neglects the up-front cost of moving data to the GPU. Based on the read-only nature of the SQL queries in this project and the characteristics of the CUDA programming model, data is stored on the GPU in row-column form. SQLite stores its data in a B-Tree, thus an explicit translation step is required. For convenience, this process is performed with a SELECT query in SQLite to retrieve a subset of data from the currently open database.

The Tesla C1060 GPU used for development has 4 gigabytes of global memory, thus setting the upper limit of data set size without moving data on and off the card during query execution. Note that in addition to the data set loaded on the GPU, there must be another memory block allocated to store the result set. Both of these blocks are allocated during the initialization of the program. In addition to allocation, meta data such as the size of the block, the number of rows in the block, the stride of the block, and the size of each column must be explicitly managed.

During the initialization phase of our implementation

it is necessary to allocate 4 memory blocks: the CPU and GPU data blocks, and the CPU and GPU results blocks. The CPU and GPU blocks of both types are the same size, but the data and results blocks can be different sizes. This may be advantageous in circumstances where the data block is greater than 50% of the available GPU memory, but the result set is expected to be fairly small. For example, if the ratio of result records to data records is expected to be 1:4, then 80% of GPU memory could be allocated for data, with 20% allocated for results.

Since data must be transferred to and from the GPU, the memory transfer time significantly affects performance. The CUDA API provides the option to use pinned memory as a way to ensure memory transfers proceed as fast as possible [31]. This type of memory is also called page-locked, and means that the operating system has relinquished the ability to swap out pages of the allocated block. Thus, once allocated, the memory is guaranteed to be in certain location and can be directly accessed by the GPU without consulting the page table. Our implementation uses pinned memory for the data and result memory blocks. Pinned memory generally speeds memory transfers by 2x, and should thus be used whenever possible with large data blocks. Unfortunately, using significant amounts of pinned memory sometimes affects the performance of other processes on a machine. Many operating systems have a fairly small limit on the amount of memory that a single process can pin, since declaring that a large portion of a system's memory cannot be swapped can starve other processes and even the operating system itself of memory.

4.3 Memory Spaces

This project attempts to utilize the memory heirarchy of the CUDA programming model to its full extent, employing register, shared, constant, local, and global memory [31]. Register memory holds thread-specific memory such as offsets in the data and results blocks. Shared memory, memory shared among all threads in the thread block, is used to coordinate threads during the reduction phase of the kernel execution, in which each thread with a result row must emit that to a unique location in the result data set. Constant memory is particularly useful for this project since it is used to store the opcode program executed by every thread. It is also used to store data set meta information, including column types and widths. Since the program and this data set information is accessed very frequently across all threads, constant memory significantly reduces the overhead that would be incurred if this information was stored in global memory.

Global memory is necessarily used to store the data set on which the query is being performed. Global memory has significantly higher latency than register or constant memory, thus no information other than the entire data set is stored in global memory, with one esoteric exception. Local memory is an abstraction in the CUDA programming model that means memory within the scope of a single thread that is stored in the global memory space. Each CUDA thread block is limited to 16 kilobytes of register memory: when this limit broken the compiler automatically places variables in local memory. Local memory is also used for arrays that are accessed by variables not known at compile time. This is a significant limitation since the SQLite virtual machine registers are stored in an array. This limitation is discussed in further detail below.

Note that texture memory is not used for data set access. Texture memory acts as a one to three dimensional cache for accessing global memory and can significantly accelerate certain applications[31]. Experimentation determined that using texture memory had no effect on query performance. There are several reasons for this. First, the global data set is accessed relatively infrequently, data is loaded into SQLite registers before it is manipulated. Next, texture memory is optimized for two dimensional caching, while the data set is accessed as one dimensional data in a single block of memory. Finally, the row-column data format enables most global memory accesses to be coalesced, reducing the need for caching.

4.4 Parsed Queries

As discussed above, SQLite parses a SQL query into an opcode program that resembles assembly code. This project calls the SQLite command processor and extracts the results, removing data superfluous to the subset of SQL queries implemented in this project. A processing phase is also used to ready the opcode program for transfer to the GPU, including dereferencing pointers and storing the target directly in the opcode program. A sample program is printed below, output by the command processor for query 1 in Appendix A.

```
0:   Trace          0   0   0
1:   Integer        60  1   0
2:   Integer        0   2   0
3:   Goto           0   17  0
4:   OpenRead       0   2   0
5:   Rewind         0   15  0
6:   Column         0   1   3
7:   Le             1   14  3
8:   Column         0   2   3
9:   Ge             2   14  3
10:  Column         0   0   5
11:  Column         0   1   6
12:  Column         0   2   7
13:  ResultRow      5   3   0
14:  Next           0   6   0
15:  Close          0   0   0
16:  Halt           0   0   0
17:  Transaction    0   0   0
18:  VerifyCookie   0   1   0
19:  TableLock      0   2   0
20:  Goto           0   4   0
```

A virtual machine execution of this opcode procedure iterates sequentially over the entire table and emits result rows. Not all of the opcodes are relevant to this project's storage of a single table in GPU memory, and are thus not implemented. The key to this kind of procedure is that opcodes manipulate the program counter and jump to different locations, thus opcodes are not always executed in order. The `Next` opcode, for example, advances from one row to the next and jumps to the value of the second argument. An examination of the procedure thus reveals the block of opcodes 6 through 14 are executed for each row of the table. The procedure is thus inherently parallelizable

by assigning each row to a CUDA thread and executing the looped procedure until the `Next` opcode.

Nearly all opcodes manipulate the array of SQLite registers in some way. The registers are generic memory cells that can store any kind of data and are indexed in an array. The `Column` opcode is responsible for loading data from a column in the current row into a certain register. The `Integer` opcode sets the value of a register to a certain integer, for instance line 1 of the example code sets register 1 to a value of 60.

The code above utilizes common opcodes to execute a simple SELECT query, but also demonstrates the thread divergence discussed in detail in this paper. Line 6 uses the `Column` opcode to load the first column of data from a certain row into register 3. Line 7 then examines this row with the `Le` opcode, which checks to see whether or not the value in register 1 is less than or equal to the value in register 3. If so, we jump to line 14, which contains the `Next` opcode that advances the query exection to the next row, thus skipping the `ResultRow` opcode responsible for outputting values from this row in a result block. Thus, opcodes 8 through 14 only get executed for certain rows, since some rows will have a value in column 1 greater than 60, and some will have a value less than or equal to 60. In our implementation we examine each row concurrently in a separate thread, thus thread divergence occurs because some threads do not always execute the same opcodes.

Note the differences between a program of this kind and a procedure of primitives, as implemented in previous research. Primitives are individual CUDA kernels executed serially, while the entire opcode procedure is executed entirely within a kernel. As divergence is created based on the data content of each row, the kernels execute different opcodes. This type of divergence does not occur with a query-plan of primitives.

4.5 Virtual Machine Infrastructure

The crux of this project is the reimplementation of the SQLite virtual machine with CUDA. The virtual machine is implemented as a CUDA kernel that executes the opcode procedure. The project has implemented around 40 opcodes thus far which cover the comparison opcodes, such as `Ge` (greater than or equal), the mathematical opcodes, such as `Add`, the logical opcodes, such as `Or`, the bitwise opcodes, such as `BitAnd`, and several other critical opcodes such as `ResultRow`. The opcodes are stored in two switch statements.

The first switch statement of the virtual machine allows divergent opcode execution, while the second requires concurrent opcode execution. In other words, the first switch statement allows threads to execute different opcodes concurrently, and the second does not. When the `Next` opcode is encountered, signifying the end of the data-dependent parallelism, the virtual machine jumps from the divergent block to the concurrent block. The concurrent block is used for the aggregation functions, where coordination across all threads is essential.

A major piece of the CUDA kernel is the reduction when the `ResultRow` opcode is called by multiple threads to emit rows of results. Since not every thread emits a row, a reduction operation must be performed to ensure that the result block is a contiguous set of data. This reduction involves inter-thread and inter-thread block communication, as each thread that needs to emit a row must be assigned a unique area of the result set data block. Although the result set is contiguous, no order of results is guaranteed. This saves the major overhead of completely synchronizing when threads and thread blocks complete execution.

The reduction is implemented using the CUDA atomic operation `atomicAdd()`, called on two tiers. First, each thread with a result row calls `atomicAdd()` on a variable in shared memory, thus receiving an assignment within the thread block. The last thread in the block then calls this function on a separate global variable which determine's the thread block's position in the memory space, which each thread then uses to determine its exact target row based on the previous assignment within the thread block. Experimentation has found that this method of reduction is faster than others for this particular type of assigment, particularly with sparse result sets.

This project also supports SQL aggregation functions (i.e. COUNT, SUM, MIN, MAX, and AVG), though only for integer values. Significant effort has been made to adhere to the SQLite-parsed query plan without multiple kernel launches. Since inter-thread block coordination, such as that used for aggregation functions, is difficult without using a kernel launch as a global barrier, atomic functions are used for coordination, but these can only be used with integer values in CUDA. This limitation is expected to be removed in next-generation hardware, and the performance data for integer aggregates is likely a good approximation of future performance for other types.

As discussed, we assign a table row to each thread. Our implementation uses 192 threads per thread block based on experimentation. CUDA allows a maximum of 65,536 thread blocks in a single dimension, so we encounter a problem with more than 12,582,912 rows. We have implemented functionality in which a thread handles multiple rows if the data size exceeds this number, although this incurs a performance cost because of the increase in thread divergence.

4.6 Result Set

Once the virtual machine has been executed, the result set of a query still resides on the GPU. Though the speed of query execution can be measured simply by timing the virtual machine, in practice the results must be moved back to the CPU to be useful to the host process. This is implemented as a two-step process. First, the host transfers a block of information about the result set back from the GPU. This information contains the stride of a result row and the number of result rows. The CPU multiplies these values to determine the absolute size of the result block. If there are zero rows then no result memory copy is needed, otherwise a memory copy is used to transfer the result set. Because we know exactly how large the result set is, we do not have to transfer the entire block of memory allocated for the result set, saving significant time.

4.7 Streaming

Although we have targeted our implementation towards applications that amortize moving data to the GPU by executing multiple queries, we have also implemented a feature which allows some of this transfer time to be overlapped with kernel execution, thus making *a priori* data staging less important. CUDA includes a feature called streams [2] which allows the programmer to explicitly define which memory copies and kernel launches are dependent upon one another and accelerate applications by concurrently executing independent steps [31]. By breaking query execution into multiple kernel launches, for instance by executing 4 kernels which each process 25% of the total dataset, we can use streams to assign a memory copy operation to each of these kernels and overlap them with other kernels. Thus, we first perform a memory copy for kernel 1, then we are able to start memory copies for kernels 2, 3, and 4 while kernel 1 executes, thus reducing the number of rows transferred before kernel execution by 75%.

This process of streaming is complicated by the overhead of initializing memory copies. Using the previous example, reducing the number of rows transferred before kernel execution by 75% will not reduce the total memory transfer time before kernel execution by 75%, since setting up the memory copy takes a significant amount of time. In our implementation the memory copy to the GPU is a much less expensive step than transforming the data from SQLite's format to the GPU-optimized row-column format. Thus, we have implemented this feature more for research purposes than performance improvement, as it provides insight into the performance that could be achieved if the data format was identical between the CPU and the GPU.

This type of streaming necessarily makes use of CUDA's asynchronous memory copy functionality. During most memory copy operations, the code executed on the host waits for a memory copy to return, but this is not the case with the asynchronous API. However, it is possible even without streams that kernels will return before execution has completed. Thus, a call to `cudaThreadSynchronize()` is a necessary part of kernel launches with and without streams.

Streaming has not been implemented for transferring results rows during kernel execution, only data rows. Although all of its result rows have been output once a query execution kernel finishes, there is no way to know exactly how many rows have been output without checking the meta-variables of the results block. Thus, after each of the streaming kernels, the current size of the results block would have to be checked before transferring rows. Because the results block and the meta-information are being frequently updated by subsequent streaming kernels, there are a number of concurrency concerns in implementing such a system. Even though such an implementation would almost certainly increase performance for certain queries, the time constraints of this project mean that it is a topic for future research. The result records are almost always a strict subset of the data records, thus the acceleration for implementing streaming with result row transfers is smaller than the acceleration we gain by implementing streaming for the data block.

A huge advantage to streaming is the possibility of executing queries on data blocks that are larger than GPU memory, possibly of an arbitrarily large size. Though we have not implemented such a system, overlapping data transfers with kernel execution means that data could reside on the GPU only as long as necessary. Such an implementation would need to stream the results off of the GPU as well, since these would accumulate as more and more data is processed. A table of data and results blocks would also need to be kept, since memory would have to be cleared to be re-used during query execution. Under such a system, the disk i/o rate and network speed becomes much more of a bottleneck than the GPU memory transfer time.

[2]Streaming in this context refers to a specific feature of the CUDA API, and should not be confused with the streaming multiprocessors of NVIDIA GPUs, or the more general concept of stream processing.

4.8 Interface

The project has been implemented with the intention of making the code as simple an accessible as possible. Given the implementation as a layer above SQLite, effort has been made to conform to the conventions of the SQLite C interface. The full application programming interface (API) is documented in Appendices B through G.

5. PERFORMANCE

5.1 Data Set

The data used for performance testing has five million rows with an id column, three integer columns, and three floating point columns. The data has been generated using the GNU Scientific Library's random number generation functionality. One column of each data type has a uniform distribution in the range [-99.0, 99.0], one column has a normal distribution with a sigma of 5, and the last column has a normal distribution with a sigma of 20. Integer and floating point data types were tested. The random distributions provide unpredictable data processing results and mean that the size of the result set varies based on the criteria of the SELECT query.

To test the performance of the implementation, 13 queries were written, displayed in Appendix A. Five of the thirteen query integer values, five query floating point values, and the final 3 test the aggregation functions. The queries were executed through the CPU SQLite virtual machine, then through the GPU virtual machine, and the running times were compared. Also considered was the time required to transfer the GPU result set from the device to the host. The size of the result set in rows for each query is shown, as this significantly affects query performance. The queries were chosen to demonstrate the flexibility of currently implemented query capabilities and to provide a wide range of computational intensity and result set size.

We have no reason to believe results would change significantly with realistic data sets, since all rows are checked in a select operation, and the performance is strongly correlated with the number of rows returned. The implemented reductions all function such that strange selection patterns, such as selecting every even row, or selecting rows such that only the first threads in a thread block output a result row,

Table 1: Performance Data by Query Type

Queries	Speedup	Speedup w/ Transfer	CPU time (s)	GPU time (s)	Transfer Time (s)	Rows Returned
Int	42.11	28.89	2.3843	0.0566	0.0259148	1950104.4
Float	59.16	43.68	3.5273	0.0596	0.0211238	1951015.8
Aggregation	36.22	36.19	1.0569	0.0292	0.0000237	1
All	50.85	36.20	2.2737	0.0447	0.0180920	1500431.08

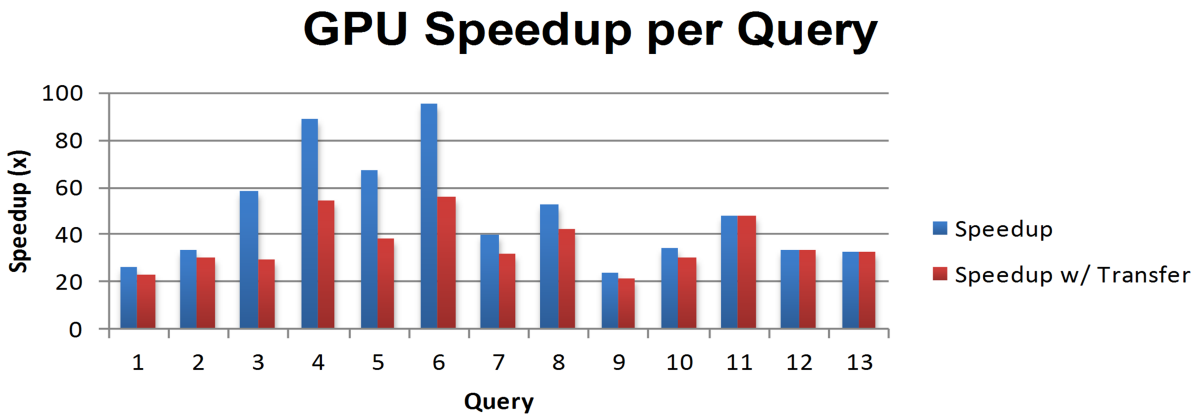

Figure 1: The speedup of query execution on the GPU for each of the 13 queries considered, both including and excluding the results transfer time.

make no difference in performance. Unfortunately, we have not yet been able to set up real data sets to validate this hypothesis, and this is something left for future work, but there is little reason to expect different performance results.

5.2 Hardware and Software

The performance results were gathered from an Intel Xeon X5550 machine running Linux 2.6.24. The processor is a 2.66 GHz 64 bit quad-core, supporting eight hardware threads with maximum throughput of 32 GB/sec. The machine has 5 gigabytes of memory. The graphics card used is an NVIDIA Tesla C1060. The Tesla has 240 CUDA cores, 4 GB of global memory, and supports a maximum throughput of 102 GB/sec.

The machine used the CUDA 2.2 drivers during testing. SQLite 3.6.22 was used for the performance comparison, and compiled using the Intel C Compiler 11.1, with heavy optimization enabled. The NVIDIA CUDA compiler, `nvcc`, uses the GNU C Compiler to handle the C code in CUDA files. Thus, some of the code which directly calls the CUDA kernel was compiled with GCC 4.2.4.

5.3 Fairness of Comparison

Every effort has been made to produce comparison results that are as conservative as possible.

- Data on the CPU side has been explicitly loaded into memory, thus eliminating mid-query disk accesses. SQLite has functionality to declare a temporary database that exists only in memory. Once initalized, the data set is attached and named. Without this step the GPU implementation is closer to 200x faster, but it makes for a fairer comparison: it means the data is loaded completely into memory for both the CPU and the GPU.

- SQLite has been compiled with the Intel C Compiler version 11.1. It is optimized with the flags `-O2`, the familiar basic optimization flag, `-xHost`, which enables processor-specific optimization, and `-ipo`, which enables optimization across source files. This forces SQLite to be as fast as possible: without optimization SQLite performs significantly worse.

- Directives are issued to SQLite at compile time to omit all thread protection and store all temporary files in memory rather than on disk. These directives reduce overhead on SQLite queries.

- Results from the host query are not saved. In SQLite results are returned by passing a callback function with the SQL query. This is set to null, which means that host query results are thrown away while device query results are explicitly saved to memory. This makes the the SQLite execution faster.

5.4 General Results

Table 1 shows the mean results for the five integer queries, the five floating point queries, the three aggregation queries, and all of the queries. The rows column gives the average number of rows output to the result set during a query, which is 1 for the aggregate functions data, because the functions implemented reduce down to a single value across all rows of the data set. The mean speedup across all queries was 50x, which was reduced to 36x when the results transfer

time was included. This means that on average, running the queries on the dataset already loaded on to the GPU and transferring the result set back was 36x faster than executing the query on the CPU through SQLite. The numbers for the all row are calculated with the summation of the time columns, and are thus time-weighted.

Figure 1 graphically shows the speedup and speedup with transfer time of the tested queries. Odd numbered queries are integer queries, even numbered queries are floating point queries, and the final 3 queries are aggregation calls. The graph shows the significant deviations in speedup values depending on the specific query. The pairing of the two speedup measurements also demonstrates the significant amount of time that some queries, such as query 6, spend transferring the result set. In other queries, such as query 2, there is very little difference. The aggregation queries all had fairly average results but trivial results transfer time, since the aggregation functions used all reduced to a single result. These functions were run over the entire dataset, thus the speedup represents the time it takes to reduce five million rows to a single value.

The time to transfer the data set from the host memory of SQLite to the device memory is around 2.8 seconds. This operation is so expensive because the data is retrieved from SQLite through a query and placed into row-column form, thus it is copied several times. This is necessary because SQLite stores data in B-Tree form, while this project's GPU virtual machine expects data in row-column form. If these two forms were identical, data could be transferred directly from the host to the device with a time comparable to the result transfer time. Note that if this were the case, many GPU queries would be faster than CPU queries even including the data transfer time, query execution time, and the results transfer time. As discussed above, we assume that multiple queries are being performed on the same data set and ignore this overhead, much as we ignore the overhead of loading the database file from disk into SQLite memory.

Interestingly, the floating point queries had a slightly higher speedup than the integer queries. This is likely a result of the GPU's treatment of integers. While the GPU supports IEEE 754 compliant floating point operations, integer math is done with a 24-bit unit, thus 32-bit integer operations are essentially emulated[31]. The resulting difference in performance is nontrivial but not big enough to change the magnitude of the speedup. Next generation NVIDIA hardware is expected to support true 32-bit integer operations.

5.5 Data Size Results

An examination of query speedup as a function of the data set size yields interesting results. Figure 2 displays the speedups for a set of queries in which the query is held constant but the table size changed. A SELECT query was run that selected every member of the uniform integer column with a value less than 0, roughly half of the set. Testing data set set sizes at 5000 row increments in the range of 0 to 500,000 yields the displayed results. Below 50,000 rows the speedup increases rapidly as the GPU becomes more saturated with work and is able to schedule threads more efficiently. Interestingly, speedup then increases slowly until

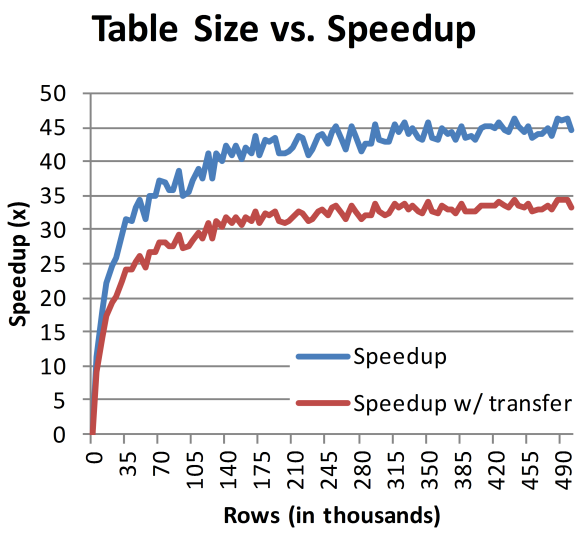

Figure 2: The effect of the data set size on speedup of GPU query execution, including and excluding the results transfer time.

around 200,000 rows, where it becomes constant. Further testing has shown this constant trend to continue to data sets up to the thread block barrier around 12.5 million rows, where it decreases slightly. Although only two columns are selected in our query, it was performed on the data set described previously, which contains 4 integer columns and 3 floating point columns, for a total width of 28 bytes. Thus, for this table specific query, we do not reach optimal performance for data sizes less than 5 MB. Note, however, that the break even point at which queries execute on the GPU is very low, above the resolution of this graph.

It is surprising just how low the break-even point of GPU query execution was in terms of data size. Testing the same query to select every member of the uniform integer column less than 0 with increasing data sets of 10 row increments, GPU execution matched the speed of CPU execution with a data set of around 350 rows. These results are shown in Figure 3, which graphs the speed of GPU execution relative to CPU execution at a much higher resolution than Figure 2. This means the break-even point occurs where speedup is equal to 1. Obviously this point is slightly different when the results transfer time is included. The break-even of 350 is only slightly larger than the 240 multiprocessors of the Tesla used for testing, and indicates that GPU query execution is faster than CPU execution even for very small data sets, and even when considering the time needed to transfer results back. Although every query has a slightly different GPU speedup, this low value demonstrates the expected break-even point for even the hardest query is still fairly low. There are very few data sets for which query speed is an issue that are less than 350 rows, suggesting that GPU is a good approach for almost all SELECT queries other than equality searches.

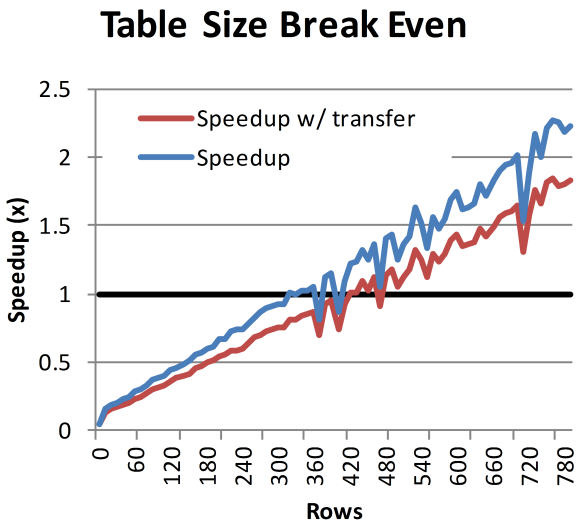

Figure 3: The break-even point in rows for query execution, including and excluding the results transfer time.

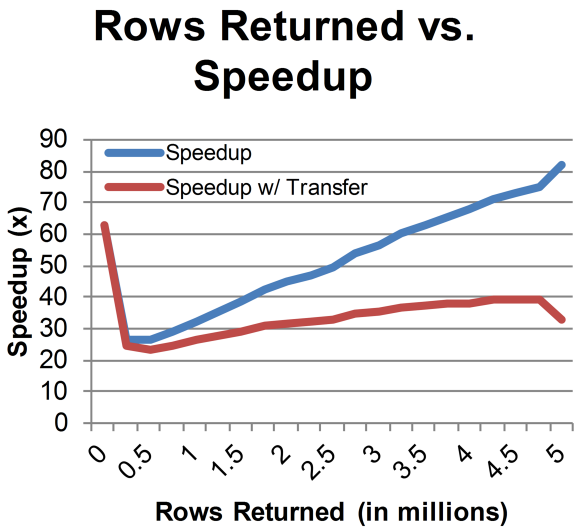

Figure 4: The effect of the result set size on the speedup of GPU query execution, including and excluding the results transfer time.

5.6 Result Size as a Factor

There are several major factors that affect the speedup results of individual queries, including the difficulty of each operation and output size. Though modern CPUs run at clock speeds in excess of 2 GHz and utilize extremely optimized and deeply pipelined ALUs, the fact that these operations are parallelized over 240 CUDA cores means that the GPU should outperform in this area, despite the fact that the SMs are much less optimized on an individual level. Unfortunately, it is difficult to measure the computational intensity of a query, but it should be noted that queries 7 and 8, which involve multiplication operations, performed on par with the other queries, despite the fact that multiplication is a fairly expensive operation.

A more significant determinant of query speedup was the size of the result set, in other words, the number of rows that a query returned. This matters because a bigger result set increases the overhead of the reduction step since each thread must call `atomicAdd()`. It also directly affects how long it takes to copy the result set from device memory to host memory. These factors are illuminated with Figure 4. A set of 21 queries were executed in which rows of data were returned when the `uniformi` column was less than x, where x was a value in the range [-100, 100] incremented by 10 for each subsequent query. Since the `uniformi` column contains a uniform distribution of integers between -99 and 99, the expected size of the result set increased by 25,000 for each query, ranging from 0 to 5,000,000.

The most striking trend of this graph is that the speedup of GPU query execution increased along with the size of the result set, despite the reduction overhead. This indicates that the GPU implementation is more efficient at handling a result row than the CPU implementation, probably because of the sheer throughput of the device. The overhead of transferring the result set back is demonstrated in the second line, which gradually diverges from the first but still trends up, showing that the GPU implementation is still more efficient when the time to transfer a row back is considered. For these tests, the unweighted average time to transfer a single 16 byte row (including meta information and memory copy setup overhead) was 7.67 ns. Note that the data point for 0 returned rows is an outlier. This is because transferring results back is a two step process, as described in the implementation section, and the second step is not needed when there are no result rows. This point thus shows how high the overhead is for using atomic operations in the reduction phase and initiating a memory copy operation in the results transfer phase.

5.7 Streaming Results

The streaming feature described in the implementation section significantly increased query speedup when the data transfer time is included in the timing. A significant variable used for this feature is the stream width, or the number of independent streams of asynchronous data transfers and kernel executions. For example, with a stream width of only two blocks, then there are just two memory copies and kernel executions. This may not be entirely optimal, since the first kernel must wait for 50% of the data to transfer before beginning execution.

Figure 5 shows experimentation with several different stream widths and demonstrates clearly how streaming affects speedup. A single query selecting all values in the uniform integer column less than 50, or about 75% of the data, was run for data tables of sizes ranging from 0 to 15,000,000 at 100,000 row increments. The results show timing that includes the data transfer time and the kernel execution time, but not the results transfer time. Note that this is different than the other figures, which show result transfer time but not data transfer time. The graph demonstrates that

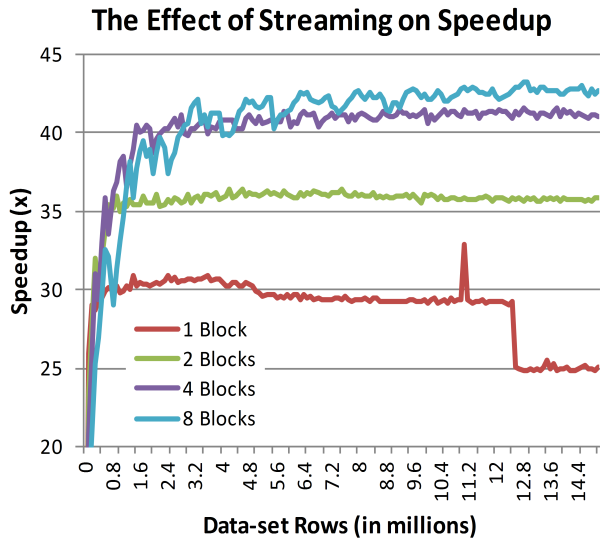

Figure 5: A query executed over a varying number of rows with the streaming feature disabled and enabled with 2, 4, and 8 blocks.

streaming can significantly increase query speedup. The differing blocks provides a good intuitive understanding of the decreasing marginal speedup returns of increasing the stream width. With 1 block 100% of the data must be transferred prior to the single kernel execution, but this decreases to 50%, 25%, and 12.5% with 2, 4, and 8 blocks. With 8 blocks it appears that the transfer initialization overhead becomes more of a facter relative to the total transfer time, and tests show that there is little to no additional speedup with 16 blocks.

Figure 5 also demonstrates query speedup for table sizes above 12.5 million. As discussed in the Virtual Machine Infrastructure section, at this point the kernel begins processing multiple threads with the same thread. Clearly this has a deleterious affect on performance, and further research may provide insight into better approaches to handling huge data sets. An advantage of streaming is that this point is multiplied by the number of streaming blocks, and thus is much larger for streaming execution.

5.8 Multicore Extrapolation

We have not yet implemented a parallel version of the same SQLite functionality for multicore CPUs. This is an important aspect of future work. In the meantime, the maximum potential speedup with multiple cores must be kept in mind when interpreting the GPU speedups we report. Speedup with multicore would have an upper bound of the number of hardware threads supported, 8 on the Xeon X5550 used for testing, and would be reduced by the overhead of coordination, resulting in a speedup less than 8x. The speedups we observed with the GPU substantially exceed these numbers, showing that the GPU has a clear architectural advantage.

6. FURTHER IMPROVEMENT

6.1 Unimplemented Features

By only implementing a subset of SELECT queries on the GPU, the programmer is limited to read-only operations. As discussed, this approach applies speed to the most useful and frequently used area of data processing. Further research could examine the power of the GPU in adding and removing data from the memory-resident data set. Though it is likely that the GPU would outperform the CPU in this area as well, it would be subject to a number of constraints, most importantly the host to device memory transfer bottleneck, that would reduce the usefulness of such an implementation.

The subset of possible SELECT queries implemented thus far precludes several important and frequently used features. First and foremost, this project does not implement the JOIN command, used to join multiple database tables together as part of a SELECT query. The project was designed to give performance improvement for multiple queries run on data that has been moved to the GPU, thus encouraging running an expensive JOIN operation before the data is primed. Indeed, since data is transferred to the GPU with a SELECT query in this implementation, such an operation is trivial. GROUP BY operations are also ignored. Though not as complex as join operations, they are a commonly implemented feature that may be included in future implementations. The SQL standard includes many other operators, both commonly used and largely unimplemented, and this discussion of missing features is far from comprehensive.

Further testing should include a multicore implementation of SQLite for better comparison against the GPU results presented. Such an implementation would be able to achieve a maximum of only n times faster execution on an n-core machine, but a comparison with the overhead of the shared memory model versus the CUDA model would be interesting and valuable. Additionally, further testing should compare these results against other open source and commercial databases that do utilize multiple cores. Anecdotal evidence suggests that SQLite performance is roughly equivalent to other databases on a single core, but further testing would prove this equivalence.

6.2 Hardware Limitations

There exist major limitations of current GPU hardware that significantly limit this project's performance, but may be reduced in the near future. First, indirect jumps are not allowed. This is significant because each of the roughly 40 SQLite opcodes implemented in the virtual machine exist in a switch block. Since this block is used for every thread for every opcode, comparing the switch argument to the opcode values creates nontrivial overhead. The opcode values are arbitrary, and must only be unique, thus they could be set to the location of the appropriate code, allowing the program to jump immediately for each opcode and effectively removing this overhead. Without indirect jumps, this optimization is impossible.

The next limitation is that dynamically accessed arrays are stored in local memory rather than register memory in CUDA. Local memory is an abstraction that refers to memory in the scope of a single thread that is stored in the global

memory of the GPU. Since it has the same latency as global memory, local memory is 100 to 150 times slower than register memory [31]. In CUDA, arrays that are accessed with an an index that is unknown at compile time are automatically placed in local memory. In fact it is impossible to store them in register memory. The database virtual machine is abstract enough that array accesses of this nature are required and very frequent, in this case with the SQLite register array. Even the simplest SQL queries such as query 1 (shown in Appendix A) require around 25 SQLite register accesses, thus not being able to use register memory here is a huge restriction.

Finally, atomic functions in CUDA, such as `atomicAdd()` are implemented only for integer values. Implementation for other data types would be extremely useful for inter-thread block communication, particularly given the architecture of this project, and would make implementation of the aggregate functions much simpler.

All three of these limitations are expected to disappear with Fermi, the next generation of NVIDIA's architecture [30]. Significant efforts are being made to bring the CUDA development environment in line with what the average programmer is accustomed to, such as a unified address space for the memory heirarchy that makes it possible to run true C++ on Fermi GPUs. It is likely that this unified address space will enable dynamic arrays in register memory, drastically reducing the number of global memory accesses required in our implementation. Combined with the general performance improvements of Fermi, it is possible that a slightly modified implementation will be significantly faster on this new architecture.

The most important hardware limitation from the standpoint of a database is the relatively small amount of global memory on current generation NVIDIA GPUs. The current top of the line GPGPU, the NVIDIA Tesla C1060, has four gigabytes of memory. Though this is large enough for literally hundreds of millions of rows of data, in practice many databases are in the terabyte or even petabyte range. This restriction hampers database research on the GPU, and makes any enterprise application limited. Fermi will employ a 40-bit address space, making it possible to address up to a terabyte of memory, though it remains to be seen how much of this space Fermi-based products will actually use.

The warp size of GPU hardware is a major consideration for query performance. NVIDIA GPU hardware divides thread blocks into so-called warps, which are groups of threads that are concurrently scheduled for execution. Warps are SIMD synchronous, so every thread within a warp either executes the same command or waits until it has a chance to do so, giving NVIDIA hardware its SIMT characteristic. Thus, overall execution is slower if threads disagree, or diverge, on what command to execute, as they do in our implementation when different opcodes are executed by threads within the same warp. Current hardware has a warp size of 32 threads, which means that a performance penalty is incurred if any of 32 threads in a warp diverge. We believe that this type of divergence is much more prevalent with the general data processing we describe in this paper than most other CUDA applications. Future hardware is expected to include more advanced warp scheduling that allows 2 warps execute concurrently, significantly altering how divergence is handled [30]. We do not consider this a "limitation," because reducing the warp size essentially optimizes the hardware towards applications with more divergence, however, there may be significant acceleration of our implementation with this new scheduler.

6.3 Heterogeneous Configuration

A topic left unexamined in this paper is the possibility of breaking up a data set and running a query concurrently on multiple GPUs. Though there would certainly be coordination overhead, it is very likely that SQL queries could be further accelerated with such a configuration. Consider the NVIDIA Tesla S1070, a server product which contains 4 Tesla GPUs. This machine has a combined GPU throughput of 408 GB/sec, 960 CUDA cores, and a total of 16 GB of GPU memory. Further research could implement a query mechanism that takes advantage of multiple GPUs resident on a single host and across multiple hosts.

A similar topic is the possibility of executing queries concurrently on the CPU and the GPU. With our current implementation the programmer explicitly chooses between CPU and GPU execution, and during GPU execution the CPU remains idle until the kernel finishes. Other research, such as the GDB implementation, has examined having the query compiler choose between the CPU and the GPU based on expected query time [21]. Since these systems break queries into a directed graph of steps, some of these steps can be executed on one processing unit, before moving the data and executing on the other. There may be significant performance gains from advancing this system to the next level by executing a step partially on both the GPU and CPU. For instance, idle CPU cycles could be utilized by allocating 10% of the table data for CPU execution. This approach is especially feasible with the platform-independent system of opcodes currently used by SQLite, but requires more complex synchronization between the CPU and GPU. Further research could examine these topics in detail and develop a method for finding the optimal division of query execution based on the characteristics of individual queries.

6.4 SQL Compilation

As described, our current implementation utilizes the SQLite command processor (or compiler), which includes the query parser and code generator that outputs the query in the SQLite opcode form. Consequently, the opcodes and opcode patterns are identical for the SQLite virtual machine and our GPU virtual machine. It is certainly possible to implement additional features such as JOIN using this form, but a closer examination reveals that these opcode patterns were intented for serial execution. Parallel execution can only be achieved by altering the order of certain opcodes and delaying execution of certain opcode effects until later in the query, hence our implementation of one set of opcodes for divergent execution and one set for synchronized execution. It is clear that an intermediate code-generator specifically targeted at parallel execution would be better suited for GPU execution, but this may not be an entirely optimal solution.

An optimal query code generator should be aware of the

structure of GPU hardware. A major concern with GPU execution, and GPU programming in general, is the use of complete thread synchronization to reach optimal solutions for certain problems. A reduction to find the average of a million values in GPU memory, for instance, performs optimally when all threads and thread blocks execute this task concurrently. In CUDA, thread block synchronization can be performed with kernel launches, because all thread blocks synchronize at the beginning an end of a kernel launch, or with atomic functions, which is a less efficient and less elegant approach. The drawback of using kernel launch synchronization is that register memory is thrown away between kernel launches. The SQLite opcode structure relies on using SQLite registers to store intermediate values, between when these values are accessed from a table and when they are output as a result. This step is essential, because these values are often manipulated and compared before being output as results, and having them lost at the end of a kernel launch is disruptive to this process. We could save these intermediate values to global memory between kernel launches, but this is inefficient and impractical, since every thread may have 10 or 20 intermediate variables that must be written. Thus, the gains from kernel launch thread block synchronization must be balanced against the gains from using register memory to manage intermediate values during query execution. An optimal query compiler for GPU hardware would have to be aware of this tradeoff. A major area for future research is the design of these optimal compilers.

Another approach to SQL query compilation is to use code generation specifically targeted at GPU hardware, similar to the code generation described for specific machine hardware and network architecture described in other research [15]. Such an approach would output code written directly in CUDA or even PTX, the assembly language to which CUDA is compiled, thus completely eliminating the opcode layer. Once generated, this code would be compiled with NVIDIA's software before execution. Implementation of this step would require significant effort, and is outside the scope of this research, but we *can* speculate as to its effectiveness. In the current implementation, a significant amount of time is spent switching to the next opcode, and switching to data type specific code within each opcode. This overhead could be avoided with all code generated for a specific query, which we alone estimate could result in a 2x speedup. Additionally, this approach would avoid the issue we have on current hardware that arrays are stored in local rather than register memory, since the generated code would be aware of exactly where data is stored. The drawback of this approach is that query compilation would probably take longer, since code would have to both be generated with our software and separately compiled with NVIDIA software. Thus, the break-even point at which this approach is faster than the current opcode scheme may be fairly high, possibly only being useful for queries executed over many millions of rows.

A useful feature in future implementations would be the ability to batch queries. This would enable multiple queries to be compiled together before concurrent execution, with each query outputting its results to a separate block of results. This approach could yield significant performance gains for several reasons. The first relates to the locking of the GPU. If the GPU platform was implemented more as a database component and less as a query executor, as in our current implementation, there would need to be significant CPU-side locking. Locking would ensure that queries are not run while data is updated on the GPU and that multiple threads do not access the GPU simultaneously, as these situations could lead to inconsistent results and GPU-memory data corruption. Batched queries would mean that once a lock is obtained by a thread, more work could be performed with a single access before the lock is relinquished. The next advantage is that global memory accesses over all queries would be significantly reduced. If two different queries needed to access the same column of data, this costly global memory access could be performed just once, after which each query would perform its own task with the data. We believe this batching approach is more useful on the GPU than most other hardware because of the lack of caching for global memory accesses.

Finally, further research could examine using more general query languages, such as Pig or Hive, for GPU query execution [14, 32, 38]. The advantage of this approach is that specific GPU primitives, such as map or scan, could be explicitly codified in the query language. Thus, a programmer could both perform a SQL SELECT and a scan operation from the same query interface. This would also allow operations on more unstructured data to be easily performed, for instance with matrix multiplication, an extremely efficient operation on GPUs. A huge advantage of this approach is the possibility of moving between heterogenous computing platforms with no change in query languages. For instance, if a `MATRIX-MULTIPLY` keyword were implemented with backend support for both GPUs and multi-core CPUs, a programmer could access proven optimal algorithms written in CUDA and a library such as pthreads with no development time, thus allowing him to move between optimal CPU and GPU execution effortlessly. In summary, this approach could duplicate the declarativeness of SQL but allow access to very complex data operations that cannot be performed with SQL with heavily optimized algorithms, thus making GPUs an even more powerful query execution platform.

7. CONCLUSIONS

This project simultaneously demonstrates the power of using a generic interface to drive GPU data processing and provides further evidence of the effectiveness of accelerating database operations by offloading queries to a GPU. Though only a subset of all possible SQL queries can be used, the results are promising and there is reason to believe that a full implementation of all possible SELECT queries would achieve similar results. SQL is an excellent interface through which the GPU can be accessed: it is much simpler and more widely used than many alternatives. Using SQL represents a break from the paradigm of previous research which drove GPU queries through the use of operational primitives, such as map, reduce, or sort. Additionally, it dramatically reduces the effort required to employ GPUs for database acceleration. The results of this paper suggest that implementing databases on GPU hardware is a fertile area for future research and commercial development.

The SQLite database was used as a platform for the project, enabling the use of an existing SQL parsing mechanism

and switching between CPU and GPU execution. Execution on the GPU was supported by reimplementing the SQLite virtual machine as a CUDA kernel. The queries executed on the GPU were an average of 35x faster than those executed through the serial SQLite virtual machine. The characteristics of each query, the type of data being queried, and the size of the result set were all significant factors in how CPU and GPU execution compared. Despite this variation, the minimum speedup for the 13 queries considered was 20x. Additionally, the results of this paper are expected to improve with the release of the next generation of NVIDIA GPU hardware. Though further research is needed, clearly native SQL query processing can be significantly accelerated with GPU hardware.

8. ACKNOWLEDGEMENTS

This work was supported in part by NSF grant no. IIS-0612049 and SRC grant no. 1607.001.

9. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.

[2] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, New York, NY, USA, 2010. ACM.

[3] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration in commercial databases: a case study of spatial operations. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1021–1032. VLDB Endowment, 2004.

[4] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[6] Q. Chen, A. Therber, M. Hsu, H. Zeller, B. Zhang, and R. Wu. Efficiently support mapreduce-like computation models inside parallel dbms. In *IDEAS '09: Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 43–53, New York, NY, USA, 2009. ACM.

[7] Y. Chen, D. Pavlov, P. Berkhin, A. Seetharaman, and A. Meltzer. Practical lessons of data mining at yahoo! In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 1047–1056, New York, NY, USA, 2009. ACM.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention*, Dec 2004.

[9] A. di Blas and T. Kaldeway. Data monster: Why graphics processors will transform database processing. *IEEE Spectrum*, September 2009.

[10] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *WWW '09: Proceedings of the 18th international conference on World wide web*, pages 421–430, New York, NY, USA, 2009. ACM.

[11] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. GPUQP: query co-processing using graphics processors. In *ACM SIGMOD International Conference on Management of Data*, pages 1061–1063, New York, NY, USA, 2007. ACM.

[12] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. Hel, Q. Luo, P. V. Sander, and K. Yang. Parallel data mining on graphics processors. Technical report, Hong Kong University of Science and Technology, 2008.

[13] E. Friedman, P. Pawlowski, and J. Cieslewicz. Sql/mapreduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2(2):1402–1413, 2009.

[14] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, 2009.

[15] B. Gedik, H. Andrade, and K.-L. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 847–856, New York, NY, USA, 2009. ACM.

[16] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data*, pages 325–336, New York, NY, USA, 2006. ACM.

[17] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 206, New York, NY, USA, 2005. ACM.

[18] Hadoop. Welcome to apache hadoop! `http://hadoop.com`.

[19] T. D. Han and T. S. Abdelrahman. hicuda: a high-level directive-based language for gpu programming. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61, New York, NY, USA, 2009. ACM.

[20] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[21] 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):1–39, 2009.

[22] T. Hoff. Scaling postgresql using cuda, May 2009. http://highscalability.com/scaling-postgresql-using-cuda.

[23] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, New York, NY, USA, 2007. ACM.

[24] M. Isard and Y. Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM.

[25] T. Kaldeway, J. Hagen, A. Di Blas, and E. Sedlar. Parallel search on video cards. Technical report, Oracle, 2008.

[26] V. Kumar, H. Andrade, B. Gedik, and K.-L. Wu. Deduce: at the intersection of mapreduce and stream processing. In *EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology*, pages 657–662, New York, NY, USA, 2010. ACM.

[27] S. Lee, S.-J. Min, and R. Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–110, New York, NY, USA, 2009. ACM.

[28] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 287–296, New York, NY, USA, 2008. ACM.

[29] W. Ma and G. Agrawal. A translation system for enabling data mining applications on gpus. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 400–409, New York, NY, USA, 2009. ACM.

[30] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[31] NVIDIA. *NVIDIA CUDA Programming Guide*, 2.3.1 edition, August 2009. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[32] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[33] SQLite. About sqlite. http://sqlite.org/about.html.

[34] SQLite. The architecture of sqlite. http://sqlite.org/arch.html.

[35] SQLite. Most widely deployed sql database. http://sqlite.org/mostdeployed.html.

[36] SQLite. Sqlite virtual machine opcodes. http://sqlite.org/opcode.html.

[37] Thrust. Thrust homepage. http://code.google.com/p/thrust/.

[38] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[39] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu. Cuda-lite: Reducing gpu programming complexity. In *LCPC*, pages 1–15, 2008.

APPENDIX

A. QUERIES USED FOR TESTING

Below are the ten queries used in the performance measurements. Note that `uniformi`, `normali5`, and `normali20` are integer values, while `uniformf`, `normalf5`, and `normalf20` are floating point values.

1. ```
   SELECT id, uniformi, normali5
   FROM test
   WHERE uniformi > 60 AND normali5 < 0
   ```
2. ```
   SELECT id, uniformf, normalf5
   FROM test
   WHERE uniformf > 60 AND normalf5 < 0
   ```
3. ```
   SELECT id, uniformi, normali5
   FROM test
   WHERE uniformi > -60 AND normali5 < 5
   ```
4. ```
   SELECT id, uniformf, normalf5
   FROM test
   WHERE uniformf > -60 AND normalf5 < 5
   ```
5. ```
   SELECT id, normali5, normali20
   FROM test
   WHERE (normali20 + 40) > (uniformi - 10)
   ```
6. ```
   SELECT id, normalf5, normalf20
   FROM test
   WHERE (normalf20 + 40) > (uniformf - 10)
   ```
7. ```
   SELECT id, normali5, normali20
   FROM test
   WHERE n̄ormali5 * normali20
   BETWEEN -5 AND 5
   ```
8. ```
   SELECT id, normalf5, normalf20
   FROM test
   WHERE n̄ormalf5 * normalf20
   BETWEEN -5 AND 5
   ```
9. ```
   SELECT id, uniformi, normali5, normali20
   FROM test
   WHERE NOT uniformi
   OR NOT normali5
   OR NOT normali20
   ```
10. ```
    SELECT id, uniformf, normalf5, normalf20
    FROM test
    WHERE NOT uniformf
    OR NOT normalf5
    OR NOT normalf20
    ```
11. ```
    SELECT SUM(normalf20)
    FROM test
    ```
12. ```
    SELECT AVG(uniformi)
    FROM test
    WHERE uniformi > 0
    ```
13. ```
    SELECT MAX(normali5), MIN(normali5)
    FROM test
    ```

B. DEVELOPMENT API OVERVIEW

The implementation of the research described in this paper is named Sphyraena, a name which is used frequently in the API because it is intended to be compiled into the same namespace as client applications, like SQLite. The Sphyraena API is built on top of SQLite, and we have attempted to conform to the style of C development used in SQLite. The state of the library is held in the sphyraena data struct, and passed to each API function, as in SQLite. SQLite should be initialized and cleaned up independently of Sphyraena. An example program using the API is provided below, with detailed documentation about all the functions and data structures you will need, and the error codes you may encounter. The most important compile-time variables are documented, and can be tweaked to improve performance for specific queries. There are a number of undocumented aspects of this program that represent decisions made during implementation and may affect performance for a certain application in some way, so further manipulation of Sphyraena can be done at a level below the API, though this will certainly be more difficult.

The API is intended for research purposes rather than production database use. As discussed, it only performs read-only queries once data has been explicitly loaded on to the GPU. However, if your application involves multiple queries over large data sets in SQLite, it should be very easy to add the API to your code and achieve significant acceleration.

C. API EXAMPLE

This example demonstrates the C code necessary to open a SQLite database and execute a simple query on the GPU. It is documented with comments but more information on each step can be found in the API documentation. This example assumes that the sphyraena.h header file has been included. Understanding it also assumes some knowledge of the SQLite API, which is documented on the SQLite website.

```
sqlite3 *db;
sphyraena sphy;

// initialize SQLite using dbfile.db as the
// database file
sqlite3_open("dbfile.db", &db);

// initialize Sphyraena with data and results
// blocks of 10 MB and using pinned memory
sphyraena_init(&sphy, db, 10485760, 10485760, 1);

// transform the data in test_table to row-column
// format so that it can be quickly moved to
// the GPU
sphyraena_prepare_data(&sphy,
"SELECT * FROM test_table");

// move the data from main memory to GPU memory
sphyraena_transfer_data(&sphy);

// execute a query on the GPU, without using
// streaming
sphyraena_select(&sphy,
"SELECT column1, column2 FROM test_table
WHERE column1 < column2", 0);

// transfer the results of the query from the
```
```

```
// GPU to main memory
sphyraena_transfer_results(&sphy);

// get data from the tenth row and second column
// of the results.
sphyraena_results *res = sphy->results_cpu;
int value = ((int*)(res->r + 10 * res->stride +
res->offsets[1]))[0];
```

D. API FUNCTIONS

This is a list of all the functions needed to execute queries on the GPU, in no particular order. The function declaration is given, followed by a description of the functions action and a list describing each argument and the return value of each function.

```
int sphyraena_init(
        sphyraena *s,
        sqlite3 *db,
        size_t data_size,
        size_t results_size,
        int pinned_memory );
}
```

This function is used to initialize the Sphyraena library, and needs to be called before any other functions. This should be called with a valid `sqlite3*` variable, meaning after `sqlite3_open()` has been called.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`db` A pointer to the sqlite3 object associated with the SQLite instance initialized.

`data_size` The size in bytes of the data block used to store data selected from SQLite for GPU execution. This block is allocated in both main and GPU memory, and should be larger than the size of the data you want to select.

`results_size` The size of the results storage block, allocated in both main and GPU memory.

`pinned_memory` An integer value indicating whether or not to use pinned memory when allocating the data and results blocks. Pinned memory enables 2x faster memory transfers to and from the GPU, but comes with some restrictions. For example: if you allocate 80% of your memory as pinned, you will probably lock your machine, assuming your OS even allows the allocation.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
int sphyraena_cleanup(
      sphyraena *s );
```

This function frees the memory allocations made by the `sphyraena_init()` function. It has no effect on the state of SQLite.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
int sphyraena_prepare_data(
      sphyraena *s,
      const char* sql_stmt );
```

This function selects data from SQLite and loads it into the Sphyraena main memory data block. Since SQLite stores this data in B+Tree format, this step is the most expensive part of staging data to the GPU, where it is stored in row-column format. The data is selected with a SQL statement, so to prepare an entire table for GPU execution use "`SELECT * FROM tablename`". You must call this before `sphyraena_transfer_data()`.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`sql_stmt` The SELECT statement used to select records from a SQLite table which is prepared by moving it into Sphyraena memory.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
int sphyraena_transfer_data(
      sphyraena *s );
```

Transfers data from from the main memory data block to the GPU data block. This is relatively quick, especially with pinned memory. Note that if `sphyraena_select()`, is called with streaming set to 1, then this step does not need to be performed.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
int sphyraena_transfer_results(
      sphyraena *s );
```

Transfers data from the results block of GPU memory to the results block of main memory. As with `sphyraena_transfer_data()`, this step is fairly quick, especially with pinned memory. This function still needs to be called with a streaming select of data, however. Note that this is a two-step procedure, we first contact the GPU to discover the size of the results block, since this varies with the query and data-set. Next we perform the actual transfer. This means that transfers for smaller results blocks will be faster, particularly for results of size 0, in which case only 1 transfer is needed.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
int sphyraena_select(
      sphyraena *s,
      const char* sql_stmt,
      int streaming );
```

This function performs GPU query execution. Only a subset of SELECT queries can be used. This subset includes specifying the columns selected, math operations such as equality and inequality and operations, logical operations, and several aggregate operations (i.e. `COUNT`, `MIN`, `MAX`, `SUM`, `AVG`). Joins and groups are not supported, and you may encounter problems with more complex combinations of these, such as using multiple aggregate operations in the same query, because of how thread block synchronization is performed. There are a number of arbitrary limits to queries and data blocks, such as the number of columns that can be used, and the complexity of the query, because there are limited allocations for these on the GPU. Many of these limitations can be changed by tweaking the compile-time variables in sphyraena.h, but these may affect performance.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`sql_stmt` The SQL statement being executed.

`streaming` An int value with 1 or 0 to turn streaming on or off, respectively. Streaming overlaps data transfer with query execution, and means that sphyraena_transfer_data() does not need to be called before this function. Streaming is probably only faster for very large data sets.

`return value` In int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
void sphyraena_print_results(
      sphyraena *s,
      int n );
```

This function prints the first n rows of the results set. This should only be called after a query has been executed and the result set has been transferred back from the GPU.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`n` The maxixmum number of rows to print. This function will stop if the result set contains less than this number.

```
int sphyraena_vm(
      sphyraena *s );
```

This function is called by `sphyraena_select()` and directly calls the kernel. It is located in vm.cu, and consequently compiled by `gcc` through `nvcc`. This function should not be directly called under normal API use.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

```
int sphyraena_vm_streaming(
      sphyraena *s );
```

This function is exactly like `sphyraena_vm()`, except it implements streaming. It is also called by `sphyraena_select()`, based on whether or not its streaming argument is set to 1.

`s` A pointer to a `sphyraena` struct, used to store the state of the library.

`return value` An int with a value of `SPHYRAENA_SUCCESS` indicating a successful execution or any of `SPHYRAENA_ERR_*` for an unsuccessful execution.

E. API DATA STRUCTURES

This is a short list of the data structures that you may want or need to manipulate to run queries. There are a number of other data structure used in the library, but they are not necessary to directly use if an application only runs queries through the API.

```
struct sphyraena {
      sqlite3 *db;
      sphyraena_data *data_cpu;
      char *data_gpu;
      sphyraena_results *results_cpu;
      sphyraena_results *results_gpu;
      sphyraena_stmt *stmt_cpu;
      size_t data_size;
      size_t results_size;
      int pinned_memory;
      int threads_per_block;
      int stream_width;
};
```

This is the primary data structure used to store the state of Sphyraena. All of the variables in this struct are set by the `sphyraena_init()` function, and other than the last two, should probably not be altered at run time. This reference is provided because of these last two variables, and because you may want to build the individual components yourself for more advanced query manipulation outside the scope of the API. Note that this can be declared directly as `sphyraena varname`, without the struct keyword.

`db` A pointer to the state struct used by SQLite. This is initialized with the `sqlite3_open()` function.

`data_cpu` A pointer to the structure used to manage the data block in main memory. The `sphyraena_data` struct is very similar to the `sphyraena_results` struct, and is initialized with data when the `sphyraena_prepare_data()` function is called.

`data_gpu` A pointer to the data block on the GPU. The `sphyraena_transfer_data()` function moves data from the block in the `data_cpu` struct to this data block, moving the meta data to the GPU's constant memory rather than storing it next to the data block. Note that this pointer has no meaning on the CPU, it points to device memory.

`results_cpu` A pointer to the struct that holds query results in main memory. This struct is not initialized until a call to `sphyraena_transfer_results()`.

`results_gpu` A pointer to the struct that holds query results in GPU memory. As with the `data_gpu` pointer, this pointer has no meaning on the CPU.

`data_size` This variable stores the total amount of memory allocated for the data block. This memory has been allocated on both the CPU and the GPU.

`results_size` Like data_size but for the results block. Note that these blocks can have arbitrarily different sizes.

`pinned_memory` A variable with a value of 1 if the data and results blocks have been allocated using pinned memory and 0 otherwise.

`threads_per_block` The number of threads run per block when the query execution kernel is called. This can be tweaked at run-time.

`stream_width` The number of blocks that the memory transfer and kernel launch should be broken into when using the streaming feature.

```
struct sphyraena_results {
      int rows;
      int columns;
      int stride;
      int types[];
      int offsets[];
      char r[];
};
```

This struct is how the results block is stored, and using it is necessary to access the returned results. This is accessed through the sphyraena object, and will not be initialized until after `sphyraena_transfer_results()` has been called. The data is stored in a single block, which the `r[]` array begins. Note that because data types of different sizes are used, you need to use the stride and offsets variables to find the location of a particular piece of data. For example, to access the third column in the fifth row of results, you would look at `r[5 * stride + offsets[2]]`.

`rows` An integer representing the number of rows in the result set.

`columns` An integer denoting the number of columns in the result set.

`stride` The width in bytes of a single row of results.

`types[]` An array of integers which represent the type of data in each column. These values are described in the Data Types section.

`offset[]` The difference in bytes between the beginning of the row and each column.

`r[]` The beginning of the actual results data block.

F. API ERROR CODES

These are compile-time integer values that are returned from certain functions in the event of normal execution or an error. Note that certain error codes can be returned only by certain functions.

`SPHYRAENA_SUCCESS`

This is the standard return values, and indicates successful execution.

`SPHYRAENA_ERR_MEM`

This indicates a failed memory allocation on the host-side.

`SPHYRAENA_ERR_STMT`

This indicates that there was a problem with the SQL statement used to select data in `sphyraena_prepare_data()`. This will get returned if the query failed or if it returned zero rows.

`SPHYRAENA_ERR_CUDAMEMCPY`

Indicates a failed memory copy in either `sphyraena_transfer_data()` or `sphyraena_transfer_results()`. Check that cleanup has not been called and that the memory has not been freed for some other reason.

`SPHYRAENA_ERR_CUDAMALLOC`

This indicates a failed memory allocation on the GPU. To troubleshoot this, check both the sizes of allocations you have chosen and if other programs are using the GPU.

`SPHYRAENA_ERR_DEVICE`

This is returned when a CUDA capable device can not be found and selected by `sphyraena_init()`. Check that the device is installed properly and that Sphyraena knows where the drivers are.

`SPHYRAENA_ERR_CUDAFREE`

This is returned when GPU-side memory can not be freed in `sphyraena_cleanup()`.

`SPHYRAENA_ERR_CUDAFREEHOST`

This is returned when pinned memory on the host side can not be freed in `sphyraena_cleanup()`.

G. DEVELOPMENT COMPILE-TIME VARIABLES

Sphyraena has a number of variables that are used at compile time to define certain operational parameters. Most have been loosely optimized through experimentation, but you may be able to increase performance on specific queries by tweaking these variables. They are all found in the `sphyraena.h` header file.

`SPHYRAENA_MAX_OPS`

Defines the maximum number of SQLite opcodes that can be included in a program. This variable controls the size of the program information block that is sent to the GPU and stored in constant memory. Simple and medium complexity queries should have no trouble fitting in this block.

`SPHYRAENA_MAX_COLUMNS`

Defines the maximum number of table columns that can be included in the table loaded onto the GPU. This is important because each column data type and byte width must be tracked. This information is also stored in the constant memory block of the GPU.

`SPHYRAENA_REGISTERS`

Defines the maximum number of SQLite registers that can be used in a query. This is similar to the max ops variable in that simple queries should have no trouble fitting into this many registers.

`SPHYRAENA_GLOBAL_REGISTERS`

A set of separate SQLite registers used for global synchronization among thread blocks. These are used primarily for the aggregation functions, which require extensive coordination among thread blocks.

`SPHYRAENA_THREADSPERBLOCK`

The standard CUDA number of threads per thread block. The default is 192 threads, but there are probably other optima depending upon the characteristics of the query, including the processing time per row and the number of result rows. Remember that this number should be a multiple of a larger power of 2. This variable is loaded into the sphyraena state struct during `sphyraena_init())` and can thus be changed at run time.

`SPHYRAENA_STREAMWIDTH`

The number of sections that the program should be split into when using streaming. Like the threads per block variable, this variable can be tweaked at run time, and probably has separate optima for specific queries.