Accelerating SQL Database Operations on a GPU with CUDA

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 command processor directly on the GPU. 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.

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 [2]. Many of these operations have direct parallels to classic database queries [4, 9].

The GPU's complex architecture makes it difficult for unfamiliar programmers to fully exploit. A productive CUDA programmer must have an understanding of six different memory spaces, a model of how CUDA threads and threadblocks 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.

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 [5, 6, 9].

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. Many queries over a large-size dataset only address a subset of the total data, thus inviting staging this subset into GPU memory. Additionally, though the finite memory size of the GPU is a significant limitation, allocat-

GPGPU-3 March 14, 2010, Pittsburg, PA, USA

ing 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

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 [9]. Other primitives include binary searches, p-ary searches [14], tree operations, relational join operations [6], 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 [8]. 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 [7]. 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 [1] PostgreSQL databases [13] 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.

The most closely related research is *Relational Query Coprocessing on Graphics Processors*, by Bingsheng He, et al. [12]. 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` [3]. 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, MapReduce has become a major topic itself. Research in this area has shown that MapReduce frameworks

can be accelerated on multicore machines [16] and on GPUs [11]. 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 [24].

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 [17]. 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 [10, 25]. 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 [15]. 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.

3. SQLITE

3.1 Overview

SQLite is a completely open source database developed by a small team supported by several major corporations [20]. 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 [22]. 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 [21]. 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 [23]. 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", which should not be confused with the register memory of CUDA. 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.

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.

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 [19]. 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[19]. 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. Note that 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.

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 different 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-threadblock 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 threadblocks 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-threadblock 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.

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. Note that 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.

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 threadblock output a result row, 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

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 streaming multiprocessors, 16 GB of global memory, and supports a maximum throughput of 102 GB/sec.

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.

- Pinned memory is not used in the comparison. Using pinned memory generally speeds transfers between the host and device by a factor of two. This means that the GPU timing results that include the memory transfer are worse than they would be if this feature was turned on.

- Results from the host query are not saved. In SQLite results are returned by passing a callback function along 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 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

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

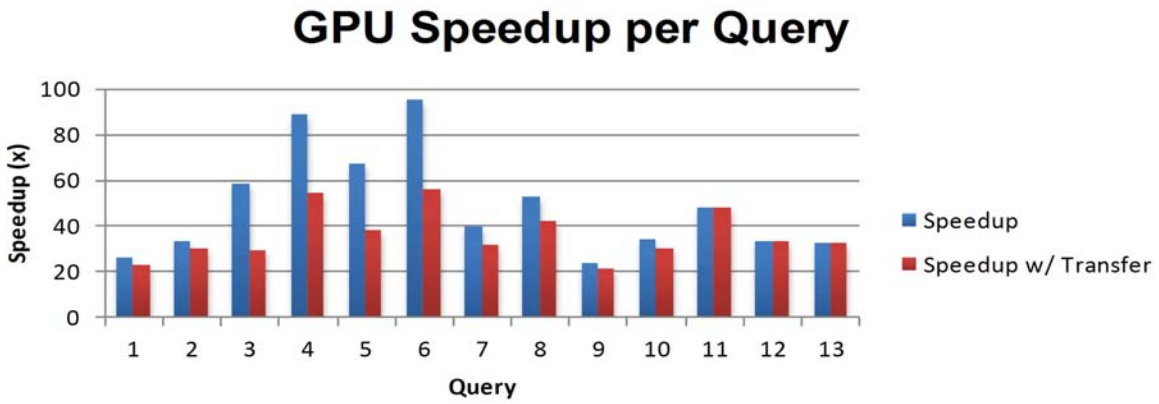

Figure 1: The speedup of query execution on the GPU for each of the 13 queries considered, both including and excluding the results transfer time

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[19]. 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.

There are several major factors that affect the 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 streaming multiprocessors 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 2. 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,

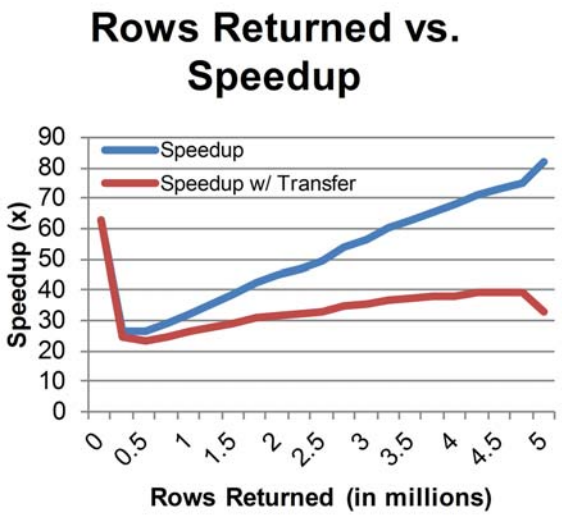

Figure 2: The effect of the result set size on the speedup of GPU query execution, including and excluding the results transfer time

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.

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 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 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 35 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 [19]. 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-threadblock 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 [18]. 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. 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.

With the capabilities of CUDA there are two ways around the memory limitation. First, data could be staged (or 'paged') between the host and the device during the execution of a query. For example, a query run on a 6 GB database could move 3 GB to the GPU, execute on this block, then move the 2nd half to the GPU and complete execution. The memory transfer time would create significant overhead and the entire database would have to fit into the host memory, since storing on disk would create huge bottleneck. It is possible that queries executed this way would still outperform CPU execution, but this scheme was not tested in this project. The second workaround for the memory limitation is to utilize CUDA's 'zero-copy' direct memory access functionality, but this is less feasible than the first option. Not only does this type of DMA have prohibitively low bandwidth, but it requires that the memory be declared as pinned[1][19]. In practice, both the GPU and the operating system are likely to have limits to pinned memory that are less than 4 gigabytes, thus undermining the basis of this approach.

6.3 Multi-GPU 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 streaming multiprocessors, 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.

[1]This type of memory is also called page-locked, and means that the operating system has relinquished the ability to swap out the page. Thus, once allocated, the memory is guaranteed to be in certain location.

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. We would also like to thank the anonymous reviewers for their helpful comments.

9. REFERENCES

[1] 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.

[2] 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.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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`.

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

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

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

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

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

[25] 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

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 normali5 * normali20 BETWEEN -5 AND 5`
8. `SELECT id, normalf5, normalf20 FROM test WHERE normalf5 * 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`