EXPLORING HIGH PERFORMANCE SQL DATABASES WITH GRAPHICS PROCESSING UNITS

Glen Hordemann

A Thesis

Submitted to the Graduate College of Bowling Green State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2013

Committee:

Dr. Jong Kwan Lee, Advisor

Dr. Joseph Chao

Dr. Andrew Schocket
ABSTRACT

Dr. Jong Kwan Lee, Advisor

This thesis introduces the development of a new GPU-based database to accelerate queries of Digital Humanities data to extract document texts that are then data-mined to produce visualizations of aspects of the humanities data. The goal is to advance the state-of-the-art in massively parallel database work by investigating methods for utilizing graphical processing units in database systems.

This thesis advances prior work done in the field of high-performance, massively-parallel databases. Some prior work focused on fixed length data types such as integers and doubles, often coupled with straight-forward single table queries. Other work focused on using primitives that are not a component of standard SQL databases.

This thesis introduces an efficient virtual database engine that executes the majority of database operations directly on the GPU. The GPU database executes a subset of SQLite’s SELECT queries, which are typically the most computationally expensive operations in a transactional database. This database engine extends existing research by exploring methods of string operations, multiple table joins, indexing, and varying length data types and sets.

This thesis discusses the design of a new GPU virtual database engine, which is developed using the CUDA extensions to C/C++, from loading data from the file on disk to processing the program on the GPU. This thesis focuses on the development of new improvements to deal with caching data for the GPU, processing and coalescing varying length data, and performing joins between multiple tables in the GPU database.

The GPU database is demonstrated in a real world application. The application wraps the database in a graphical user interface which facilitates data selection. The application performs data mining of Humanities data for common text mining algorithms. The mined data is processed into visualizations to illustrate the resulting data digests.
ACKNOWLEDGMENTS

I would like to gratefully acknowledge Catherine, who put up with a great deal while I finished this, and Dr. Jong Kwan Lee, who put me through a great deal to make me finish this. I would also like to thank Dr. Joseph Chao and Dr. Andrew Schocket for serving on my committee and for contributing their suggestions and advice for this thesis.
# Table of Contents

**CHAPTER 1: INTRODUCTION**

1.1 Organization ..................................................... 5

**CHAPTER 2: BACKGROUND**

2.1 SQLite .............................................................. 6
2.2 Compute Unified Device Architecture (CUDA) .................. 12
2.3 JSTOR ............................................................... 14

**CHAPTER 3: RELATED WORK**

3.1 GPU Databases ...................................................... 16
3.2 Visualization ......................................................... 20

**CHAPTER 4: NEW IMPROVEMENTS AND DESIGN**

4.1 Caching Improvements .............................................. 22
4.2 Varying Length Data ............................................... 28
   4.2.1 Coalescing Varying Length Data ............................ 28
   4.2.2 Data Loading .................................................. 29
   4.2.3 Coalescing Results .......................................... 30
4.3 Table Joins .......................................................... 33

**CHAPTER 5: EXPERIMENTAL RESULTS**

5.1 GPU Results ....................................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1 Caching Results</td>
<td>38</td>
</tr>
<tr>
<td>5.1.2 Varying Length Data</td>
<td>40</td>
</tr>
<tr>
<td>5.1.3 Join Performance</td>
<td>42</td>
</tr>
<tr>
<td>5.2 Toolset</td>
<td>44</td>
</tr>
</tbody>
</table>

### CHAPTER 6: CONCLUSION

6.1 Future Work 55

### BIBLIOGRAPHY 57
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SQLite Opcode Program Flow</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>Caching Data Flow</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Table Caching Decision Process</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Caching Process</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Coalescing Strings into a Superstring</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>Ship in a Bottle</td>
<td>29</td>
</tr>
<tr>
<td>4.6</td>
<td>Table Load Process</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>Atomic Coalescing of Results</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Division into Processing Groups</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>Next Opcode</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Table Select Tab View</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Visualization Tab View</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Options for Word Cloud Results</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Word Cloud Example</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Offset Viewpoint in Word Cloud</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>Twine Graph Visualization</td>
<td>50</td>
</tr>
</tbody>
</table>
## List of Tables

2.1 Example of SQLite Opcode Program ........................................ 8

4.1 Join of Three Tables .......................................................... 34

5.1 Table Sizes and Load Times .................................................... 38
5.2 CPU Cache Performance and Speedup ...................................... 39
5.3 GPU Cache Performance and Speedup ...................................... 39
5.4 CPU vs. GPU Comparison with Selection Criteria Performance ........ 40
5.5 Numeric vs. String Conditional Statement Performance ............... 41
5.6 Time of Virtual Database Engine Execution ............................... 42
5.7 Join Performance ............................................................... 43
CHAPTER 1

INTRODUCTION

The advent of general programming on a graphical processing unit (GPU) has the potential to dramatically alter the way computers are programmed. Choosing to utilize the GPU provides the ability to write programs that perform thousands of identical operations on data at once. This capability empowers developers to greatly increase the performance of existing systems that operate in serial order.

One common application with serial performance is the database, where data requests iterate over many data rows sequentially. Databases underlie many applications. (For example, the performance of the database has a direct impact on the performance of the application.) The utilization of a GPU to process database instructions has the potential to significantly speed up database operations. This increase in performance can reduce database application wait times from minutes to seconds, greatly enhancing the user experience.

In this thesis, we explore the GPU’s computing power in a database application for Humanities data. There are several obstacles to developing a database directly on a GPU and new improvements must be advanced to overcome these challenges. In addition to the challenges, the resulting solution must be integrable with existing CPU-centric applications. This integration ensures that future application developers do not need to have any specialized knowledge of GPU computing and removes a potential barrier to use of the system.
To address these challenges, this thesis devises improvements for developing a database on the GPU that supports the Structured Query Language (SQL) standard and integrates that database into another application. The SQL standard is an International Standards Organization standard for a structured query language to define and manipulate data in a database. The designed system supports complex SQL options such as querying multiple tables, processing strings and varying length data, caching tables for use on the GPU. This designed system is integrated into an application that provides data mining and visualizations of Digital Humanities data.

Developing a GPU database is feasible due to the considerable inherent parallelism in the way databases operate. A database query consists of a set of operations that are performed on the rows of one or more tables. Each of these operations is repeated, potentially once for each row in the table. Within a query, the execution of these operations is largely data independent; the execution of operations on one row usually does not effect the operations being executed on another row. This exposes a large amount of data parallelism. Each set of operations could be executed simultaneously on every row.

Traditional databases perform the steps of a query sequentially using the CPU. It is possible to achieve some parallelism by using multiple cores in the CPUs. CPUs have few cores, so attempting to exploit greater parallelism requires the addition of costly additional CPUs, and communication between large numbers of CPUs is problematic; the developer must use some mechanism to regulate the passing of data between CPU memory spaces and the rate at which data is transferred is determined by either inexpensive but slow hardware, such as Ethernet connections, or fast but expensive hardware, such as Cray’s Gemini interconnects.

The GPU provides a means to exploit a massive amount of parallelism with even a single GPU. The GPU has a significantly different design than a CPU. A GPU has many more processing cores than a CPU. Each processing core can juggle many threads at once. A thread is a sequence of programmed instructions or operations. The process of managing
threads is performed by a processing unit’s scheduling system. The scheduler on a GPU can handle millions of threads, each of which can be charged with processing a separate row from a table. Thousands of threads can then be scheduled to execute simultaneously. A GPU also has greater memory latency than a CPU. Memory latency is the amount of time taken to retrieve information from memory. Latency is hidden on the GPU by rapidly switching between execution of multiple threads, instead of relying on caching to mask memory latency as the CPU does.

In addition to the GPU having a different design than the CPU, the GPU is a separate ecosystem from the CPU that contains GPU-specific memory, scheduler, and processors. This separation imposes restrictions on controlling devices such as the hard disk, accessing memory, transferring data, and even executing program instructions. The GPU is generally unable to directly access the CPU resources. It can not access the hard disk directly, nor can it directly reference the CPU memory or assign work to the CPU. Conversely, the CPU can not directly access the GPU memory. In order to access CPU resources, the GPU must work through the CPU, which requires extra steps for translation, instruction, and control. The same limitation applies to the CPU attempting to access GPU resources.

The limitations on CPU and GPU interaction impose special complications for database processing on the GPU. The lack of direct memory access requires foreknowledge of all characteristics of any data to be transferred, which may not be available with varying sized data items. The exact size of the item may not be known until after the item is generated. Complex processes that contain steps that can not be run independently on separate threads due to data dependencies must be transformed to remove the data dependencies. Data must be transferred to and from the GPU and the data resident on the GPU must be managed.

The contribution of this thesis is development a GPU-empowered SQL database that introduces new improvements for overcoming data processing obstacles on the GPU and the development of a Humanities visualization system powered by this database. This system can process complex SQL queries and creates and implements visualizations which enable
the Digital Humanities researcher to explore and understand very large data sets.

This thesis expands beyond previous work in the implementation of databases on the GPU. Previous research into GPU databases has used data processing primitives in some cases, and simplified implementations of SQL in others. This thesis adds the ability to do complex joins, which are critical to the utility of a database in large real-world applications. Additionally, this thesis investigates and demonstrates the use of strings and non-fixed length data values, where other work has focused on fixed-length values. The fixed length values are excellent for certain data, such as map data that contains coordinate values, but do not provide the utility and flexibility needed for more complex data such as Humanities data. This thesis also introduces a caching system to address the issue of data transfer, providing a solution to the issues of data transfer times and limited GPU memory space. It examines a method for throttling the workload assigned to the GPU based on dividing the required pool of threads into chunks to avoid overloading the GPU.

This thesis supports data mining and two visualizations in a real-world application of the GPU database. It provides a traditional word cloud visualization for displaying word frequencies and a three-dimensional scatter graph that shows the result of data mining of humanities texts. This thesis is also a tool set for the Digital Humanities researcher to use. The end product is a demonstration of the effectiveness of concepts explored in the research and provides the basis for a research tool that may be used to gain new insights in the Humanities field.

Visualizing Humanities data can reveal previously hidden correlations within large sets of data that are not apparent without special data processing. Visualization allows users to use their highly-developed visual pattern recognition facilities to process data, which allows complex data to be easily understood by users. Humanities data sets are extremely large, which makes direct consumption by a user impractical. Data mining must be performed to summarize and analyze the data to make the data more accessible to a researcher. Converting this refined data into a visualization further enhances the ability of a researcher to understand
and utilize the information.

1.1 Organization

This thesis is composed of six chapters. The first chapter provides a brief overview of the challenges of this thesis and the contributions made by the research done for this thesis. The second chapter covers the background information needed to understand the thesis. The third chapter discusses related work. The fourth chapter describes the new work done to address and overcome each challenge. The fifth chapter contains the results of the thesis. The sixth chapter contains conclusions and discusses future work.
CHAPTER 2

BACKGROUND

In this chapter, the background information needed to understand the contents of the thesis, in particular the working of the SQLite database, the implementation of CUDA platform, and the origin of the Humanities data, is discussed. The GPU-enabled SQLite database system is discussed in context of the original CPU-based SQLite database implementation, which is referred to as the vanilla implementation. Understanding the vanilla implementation allows for more knowledgeable comparison with the GPU-enabled SQLite database and a better understanding of the design choices made for the GPU implementation. The CUDA architecture and the GPU architecture of the underlying GPU are also discussed. Lastly, the source of the data used in this thesis is discussed.

2.1 SQLite

SQLite [37] is an open-source database system that is widely used for embedding a database in applications on computers, smartphones, and tablets. Unlike other databases such as Oracle, SQLite can be compiled directly into the source code and run as part of a program, instead of being called as a service that runs as a separate process. SQLite is public domain software and is one of the most commonly used database systems in the world. SQLite supports the standard SQL syntax. SQL is an easy-to-use language that is ubiquitous in
industry. It is commonly used not just by programmers but by business professionals of all types. Supporting such a common interface ensures that the barrier to entry to use the GPU database system is trivial.

SQLite allows the use of complex logic in queries of the database, allowing disparate tables to be joined together by common values and a selection of results to be chosen by both simple and complex criteria. The joins are of critical interest, as the process of finding matching values between rows of data in two tables can be very repetitive and computationally expensive. Likewise, the task of finding rows that meet certain criterion is also a repetitive and computationally taxing task. Both of these tasks are ideal for GPU assistance.

SQLite’s internal architecture breaks the execution of an SQL statement into several steps. First, SQL statements are checked for validity. This ensures that the statement obeys the syntactic rules of SQL and refers to tables and rows that exist in the database. Then valid SQL statements are transformed from standard SQL notation into a series of opcodes, where each opcode can have up to five parameters, that form a program. This opcode program is then executed on a virtual machine, also known as the Virtual Database Engine (VDBE). Results from the virtual machine are returned as rows of data to the calling program.

This sequence of steps contains significant complexity; SQLite does caching of data, pre-processes SQL statements to achieve a highly efficient ordering of the opcode program, including building indexes on the fly to improve performance, handles disk operations, and transforms SQL statements into opcode programs. Processing the opcodes themselves also requires significant complexity.

There are several aspects of SQLite’s internal workings that prepare the SQL statement for execution on the virtual database engine the can be retained directly in the GPU implementation. The tokenizer, which converts the SQL statement into a list of discrete words, can be reused as written in the original code. The SQL pre-processor, which attempts to modify the SQL statement into a more efficient form, can be reused as well. The translator, which converts the list of tokens into a sequence of opcodes, is also usable and can be reused.
Table 2.1: Example of SQLite Opcode Program

without modification. All of these stages are inherently serial and are used to create an efficient opcode program to run on the virtual database engine.

Opcode processing is the most central concept to the SQLite database system. Opcodes are processed as virtual machine instructions. For example, given the query “SELECT id FROM authors”, this query will access the table named “authors” in the database, traverse every row to find the value stored in the column named “id”, and return all of those values as the result of the query. The opcode program generated by this query are shown in Table 2.1.

This program opens the database file, finds the relevant table, extracts the values stored in the correct field, and returns the result. The virtual database engine processes this program by means of a program counter (PC), which keeps track of the line whose opcode the engine is currently processing. When not otherwise modified, the program counter always advances to the following line. Opcodes have five operands, which are labeled P1 through P5. These operands provide extra information that the opcode requires. Most opcodes only use operands P1 through P3, with P4 and P5 being blank, so some future examples ignore those operands.

When executing this program, the program counter is set to 0. The program flow for
Figure 2.1: SQLite Opcode Program Flow
this program is shown in Figure 2.1. Line 0 is the Trace opcode, which performs no actions when tracing is not enabled. The program counter then is incremented to 1, telling the virtual machine to process the opcode on that line. At the next line the virtual machine encounters a Goto opcode. The Goto opcode jumps to the line stored in P2 by setting the program counter to that value, which in this case is Line 9. The program counter is set to 9, telling the virtual machine to process the opcode on that line. With the program counter at Line 9, the Transaction opcode begins a transaction. This opcode performs no actions; a transaction is not needed in a SELECT statement. The program counter then advances to Line 10. Line 10 contains the VerifyCookie opcode, which verifies that the database layout has not changed. This opcode also performs no actions as SELECT statements do not alter the database layout. The program counter advances to Line 11. The TableLock opcode at Line 11 accesses the database file and attempts to gain access to the “authors” table. It does this by finding the data page whose number is stored in operand P2. A data page is a unit of measurement of the size of the database file. In the case of this implementation, a data page is a chunk of the database file 1024 bytes in length. The database file is divided into data pages with the first page starting at the first byte, the second page starting at the 1025th byte, et cetera. The data page in this case is the sixth data page in the database file. It is the root, or starting, page for the “authors” table and will serve as the starting point for traversing the table. The program counter then advances to Line 12. Line 12 contains a Goto operand, which changes the program counter. The program counter is set to the value stored in P2, which is Line 2.

All the steps shown above are the same in every opcode program that processes a SELECT statement. The specific lines that the opcodes are on may change, and the number, name, and data page of tables may differ, but the order of operations is the same for all SELECT statements. The remaining statements will vary from program to program.

With the program counter now at Line 2, the OpenRead opcode is used to associate a cursor with a table. A cursor is a pointer to a table that indicates not only a specific table
but also a row index in a table is currently being processed. The OpenRead opcode accesses the database file and assigns the cursor in P1, in this case cursor 0, to the table whose root page is found on the page whose number is stored in operand P2, in this case data page 6, of the database file. This means that the “authors” table is now indicated by cursor 0. This cursor will be used in all later cases to refer to this table. The value 1 stored in operand P4 indicates that only one column from this table will be used in this opcode program. The program counter then advances to Line 3. At Line 3 the Rewind opcode resets the row index, or row counter, in cursor P1, in this case cursor 0, to point to the first entry in the table. If the table was empty the program counter is to the value shown in operand P2. The program counter is not set to the value in P2, so it is incremented to 4. Next, on Line 4, the Column opcode is used to copy a value from a row in a table to a register. The value stored in column P2 of the table that is pointed to by cursor P1 is placed in register P3. In this case, we take column 0, which happens to be the column “id”, of the current row of the table pointed to by cursor 0, which is the “authors” table, and store it in register 1. The program counter is then incremented to 5. On Line 5, the ResultRow opcode creates a result. This opcode takes a range of registers, assembles the contents of those registers into a row, and returns that row as a result. Operand P1 is the first register whose contents make up the result row and P2 lists how many total registers are used to make the result row. In this case, starting from register 1, a single register (register 1) is used to make up the result row. This results in a row one entry wide. All values that are to be returned by the virtual machine must first be stored in registers. The program counter is then incremented to 6. The Next opcode on Line 6 advances the cursor to the next row in a specific table. The cursor whose value is stored in P4 has its row counter advanced to the next row. If the row counter in the cursor is set to point to the next row, the program counter is set to the value stored in P2. If there is not a next row, then the entire table has been traversed. In this case, the program counter is incremented to the next row. If the table has additional rows, the opcodes on Lines 4-6 are repeated multiple times, once for each row. Once the entire table has been processed,
the program is moved past the Next opcode. When the program counter is set to Line 7, the Close opcode is reached. This opcode discards the cursor for a table and releases that table. The cursor is the one whose value is stored in operand P1. The program counter is then incremented to Line 8. At Line 8 the Halt opcode is executed. This opcode signifies the end of the opcode program and terminates the virtual machine that was running the opcode program.

A complex SQL statement will result in a longer opcode program. For instance, adding a WHERE clause, which is used to add criteria to select only specific rows, or joining multiple tables together introduces new opcodes and increased complexity. Multiple tables result in nested loops in the opcode program. We will discuss these additions when we discuss their development on the GPU.

2.2 Compute Unified Device Architecture (CUDA)

Compute Unified Device Architecture (CUDA) is the name given to the architecture NVIDIA created for GPU computing. It defines a general computing instruction set that runs on the GPU and a set of language extensions that allow a programmer to perform execution of this instruction set using standard C/C++ syntax. CUDA supports several languages and this work uses the CUDA extensions to C/C++. The developed code is then compiled using the NVIDIA CUDA compiler.

GPUs contain many cores which are organized together into larger processing units. A core is the smallest processing unit on a GPU. It is capable of processing a single thread. A group of cores is collected into a multiprocessor, which contains a fixed number of cores. These multiprocessors act as the arbiter of activity on the cores they contain, scheduling threads on specific cores, arbitrating global memory access, and managing shared memory. Shared memory is a set of memory space that is available to all cores in a multiprocessor, but not to cores in a different multiprocessor. The specifics of a GPU depend on the architecture
of the GPU. The number of cores in a multiprocessor, the number of multiprocessors on the
GPU, and the amount of shared memory available to each multiprocessor all depend on the
specific architecture used.

GPUs have three types of memory; *global memory*, *shared memory*, and *local memory*. Global memory is the main memory of the GPU and is the largest pool of memory. It
is accessible by any core in any multiprocessor. It is not directly accessible by the CPU,
however, and special calls must be made to pass data to the GPU’s main memory from the
CPU. Shared memory is much smaller than global memory in size. It is accessible by any
core within a multiprocessor, but cores on separate multiprocessors can not access the same
shared memory. Local memory is available only to a core and can not be accessed by any
other core. This memory is faster than the other types of memory.

Threads are assigned to cores for execution. Threads are not assigned individually, but
instead are scheduled in groups called *warps*. A warp is the smallest collection of threads
possible. If a single thread is executed, it is assigned to a warp by itself and that warp
is assigned for execution. Warps are the scheduling unit used by multiprocessors to con-
trol execution of threads on cores. Warp size can potentially vary by architecture, but all
NVIDIA GPUs use 32 thread warps. The 32-threads in a warp run in lock-step, internally
synchronized by the GPU automatically. There is a maximum of 64 resident blocks on a
single multiprocessor. When a warp of threads has to wait for a memory access the GPU
scheduler simply executes a different warp of threads on the GPU. For efficient operation,
significantly more than the minimum number of 192 threads (6 warps) is needed on a single
multiprocessor or else the processor will be idle during periods of memory access. The GPU
switches context between warps very efficiently.

The programmer, however, can control a larger unit of threads called the *block*. The
programmer specifics how threads are to be grouped into blocks and the GPU handles the
assignment of specific blocks to multiprocessors. The multiprocessors then divide the blocks
into warps and schedule the execution of the threads on the cores of the multiprocessor. A
A block should contain enough threads to keep the GPU cores fully occupied during periods of memory latency. Additionally, since threads are assigned by warps, and warps are 32 threads in size, it is best if blocks are a multiple of 32.

Multiple blocks are formed into a grid. A grid can contain up to $2^{31}$-1 blocks. When CUDA functions are called, the number of blocks and grids of those blocks are specified by the programmer. This sets the total number of threads that will be executed by the GPU.

Interaction between the GPU and CPU to transfer data from one memory to the other is handled by special CUDA functions. For example, the `cudaMalloc()` function is used to allocate memory on the GPU. The pointer to this memory, although it is unusable by the CPU, is retained on the CPU side, so it can be passed to CUDA functions that will run on the GPU. All processes on the GPU must originate on the CPU, so this is a critical feature.

Another example is the `cudaMemcpy()` function, which allows the program to copy an array of bytes from the CPUs memory to the GPUs memory. This data can be characters, integers, or any other data type; the function is agnostic and treats all data as simply a stream of bytes. These two functions form the bulk of the data sharing techniques used to transfer data between the CPU side and the GPU side.

It is worth noting that there are alternatives to CUDA available. OpenCL [32] is a platform agnostic alternative to CUDA. Another CUDA alternative, DirectCompute [12], is developed by Microsoft. Like OpenCL it is GPU brand-agnostic, but unlike OpenCL or CUDA it is tied into the Windows platform. In this thesis, CUDA was chosen due to its greater maturity and the specific capabilities that CUDA makes available to the developer.

### 2.3 JSTOR

The data used in this thesis comes from the JSTOR electronic library. JSTOR is a not-for-profit digital library that contains hundreds of thousands of journals, magazines, pamphlets, and other documents. A substantial portion of the JSTOR library is available for download.
This subsection contains documents that cover all aspects of human endeavor and range in date from 1665 to 1922. All the documents in this subset are in the public domain. There are over 450,000 documents in this subset. The digital library provides the full text of these documents, plus the authors, date of publication, title, page range, and some additional meta-data about the document. This metadata was parsed out into the database. To further increase the utility of the database, some documents were text mined for key words to aid in searching the database.
CHAPTER 3

RELATED WORK

There is a varied body of related work that has been presented. Both the fields of GPU Databases and Digital Humanities are fairly new, but they are the descendants of the larger fields of general databases and visualizations. While it is important to delve into what research has been done in these newer fields, it is also important to review work in the foundational fields.

3.1 GPU Databases

The field of general-purpose graphics processor unit databases has seen development in two different categories. Some of the research done has been partial implementations, either as bolt-on additions to existing traditional database systems or at proof-of-concept implementations that explore a specific aspect of the database system. The other main branch of research has been in the implementation of database systems that are native to the GPU itself.

Some bolt-on GPU database modifications have been done with the PostgreSQL database system [7]. PostgreSQL is an open-source database that allows users to define their own functions and procedures in the programming language of their choice. This has allowed researchers to write C++ functions to be executed by the database using CUDA. The GPU
functions are written as procedures that called by PostgreSQL. In the work, a variety of images are retrieved from the database and then a specific procedure is called to manipulate the image with GPU-powered stored procedures and the results are returned to the user.

External procedures have been the focus of work on spacial operators by Bandi et al. [4]. They used external procedures to accelerate spatial operations, or operations that relate data such as intersections and unions. This work showed the potential for increased performance in even CPU-oriented databases by off-loading some work to the GPU.

In addition to add-ons, significant research has been done as proof-of-concept work, examining one piece of the GPU database puzzle in isolation. Relational joins are one area where research has been done [19]. This work demonstrated the effectiveness of the GPU in performing high-speed joins of multiple tables. This work used database primitives for scan, scatter, gather, split, and sort operations which were used as constructs for building join algorithms. These joins showed that concept of using the GPU to perform join operations was feasible.

Additional work has been presented to implement more efficient sorting algorithms [28, 34, 35, 5, 17]. Sorting is an important part of database work, since not only are results sorted, but indexes need to be sorted in order to be effective as indexes. Sorting is an inherently parallel task and significant speedups can be achieved by using the GPU.

Databases wholly implemented on the GPU provide a more complete proof of the efficiency of the GPU in database operations. Bakkum et al. [2] investigated an implementation of the SQLite database management system on the GPU using the CUDA library. This work established the core principles of a purely GPU database implementation by implementing a database capable of performing SELECT queries on integer values. Data was assumed to always reside on the GPU, operating under the assumption that some caching scheme would enable the table data to remain resident on the GPU. Data was loaded onto the GPU through a one-time use of the vanilla SQLite’s own SELECT statements. A selected subset of the SQLite internal instruction set of opcodes was implemented. This set of opcodes
established that single table queries could perform considerably faster on the GPU.

SQLite has proven a fertile ground for additional GPU database research. Chang et al. [6] showed the improvement of using column-major data storage for the tables, which greatly improves the performance of data transfer to and from the GPU. This work used existing CUDA functions and libraries and added new data structures and parameters to the existing vanilla SQLite implementation to replace existing serial functionality with GPU-powered parallel functionality. This functionality allowed for the acceleration of SELECT queries on integer data in a table as well as data sorting and data grouping. Of particular interest in this work is the discovery that for small numbers of records the CPU implementation is faster than the GPU implementation. As the number of records increased, this situation reverses and the GPU implementation becomes faster.

In-memory databases, as databases that ignore the data transfer and disk access costs and work with the assumption that the entire database will fit on a single GPU, have become an area of other research [16, 3]. These works discuss a database that exists entirely on the GPU and never needs to access data outside the GPU.

The issue of overhead for data motion is one that sometimes overlooked. Gregg and Hazelwood [18] explored the costs of moving data between the CPU and the GPU and challenge the performance statistics of previous researchers. The paper described the throughput bottleneck of the PCI-Express bus. While the GPU offers extremely fast parallel processing, all data must somehow get to and from the GPU, which is accomplished through the PCI Express bus. Claimed GPU speedups can be misleading if the data transfer times are not considered and the transfer time costs can overwhelm the GPU performance time gains.

Attempts have been made to complete GPU databases for commercial purposes to enjoy the benefits of GPU performance. An open source approach to this is the Alenka project [30]. The Alenka project appears to still be in the conceptual stages. It is intended to implement a database using pure CUDA. The database would work off of a flat-file system instead of the indexed database files such as SQLite contains. However, at this time, there are no papers
published on Alenka and any existing source code is either incomplete or unavailable, and it is mentioned here only for completeness.

In addition to Alenka, there exists the ParStream commercial product [33]. ParStream is an analytic database, as opposed to a transactional database, such as might be used in a data warehouse. ParStream is a commercially available product that is intended for the data warehousing market.

Another area of research is the use of GPUs not to perform the database operations themselves, though they may or may not also be called upon to do the database execution, but to analyze the query plans for the database (e.g., [20, 42]). When an SQL statement is submitted, the database attempts to optimize it so that the SQL statement runs as quickly as possible. This involves evaluating numerous candidate execution plans. Query plan optimizers are already very efficient, so the GPU implementations propose additional options, such as speculative query processing. Speculative query processing is the processing of a query with steps being taken out of order based upon the availability of input data rather than the original order of operations. In instances where the execution of the next operation is certain the processor speculates, of guesses, what to do next. Correct speculation leads to increased performance and incorrect speculation requires repeating the steps with the correct choice. This has the potential to increase database performance in all instances.

Outside of GPU research, there is a long history of parallel research that focuses on using large numbers of CPUs to improve database performance, going back into the previous century [11]. The advantages of the GPU are immediately obvious, of course: using a single device instead of multiple computers. Nonetheless, there are lessons that can be learned from this past research and its importance should not be understated. Query optimization [21] and join processing [29] are two areas where lessons from parallel CPU databases can inform GPU research. These parallel CPU systems, referred to as distributed databases, have characteristics that can be used in GPU research.

The scalability of distributed databases is tremendous, and is used for things such as
Google’s backend [9]. To achieve the same performance, GPU databases would also have to become distributed, which means that the issues of how data is shared will become important. In the distributed world this is well-trod ground [41], though other research [23] is still being done.

3.2 Visualization

Visualizing humanities data is an exercise in applying visualization to text mining. This requires finding methods of spatially organizing text or representations of text. The most common method is the word or tag cloud. Another common method is the graph, either as a variation of the scatter graph or the more traditional bar or line graphs.

Word clouds are increasingly popular and have become the focus of several areas of research. Word clouds have found applications in fields as diverse as Victorian literature [38], the Humanities Word [8], Philosophy [1], medical training [15], and music [25]. The fundamental word cloud encodes word frequency in the size of the word in the cloud and then displays the words packed together. New research explores ways of encoding additional information into the Word Cloud through the use of color, spatial positioning, and additional graphical artifacts.

Color coding has been used in word clouds to indicate additional information. Nguyen et al. [31] explored encoding a variety of data into color characteristics of word clouds. Color of the text or background has been used to display word categories, brightness indicated tag age, and transparency indicated frequency over time. In addition to the color of the text itself, they explored adding textured masks to the background area of text to indicates patterns of data such as frequency over time or calendar information. By placing blocks or lines of color in the background, a sequence of changing colors underlies the word, providing additional information.

Color gradients and glyphs have been further expanded into specific markups that span
multiple word clouds (e.g., [26]). This process uses lines between multiple word clouds to indicate trends between multiple clouds by connecting identical words across multiple clouds. Providing this additional information allows individuals to more easily track data across multiple clouds.

Many word clouds use a random layout (e.g., [14, 40]), but recent research has investigated using placement to encode additional content information. Choosing where each word is positioned allows the encoding of semantic information into the image. Words that are spatially proximate are semantically proximate [36]. Preserving the location of words in a word cloud from one iteration to the next can also be used to encode information [10]. Words are laid out with spatial proximity indicating semantic proximity and the layout ensures that the words will appear in roughly the same area from one iteration to the next. If specific words are of interest, words can be selected as anchors, and those words will be fixed in their location for subsequent word cloud iterations and related words will be populated relative to their fixed positions.

Post processing to coalesce word clouds into a more compact form while still maintaining the meaning in spatial relationships has also been the focus of research (e.g., [43]). The process of seam carving allows the removal of vertical and horizontal seams in the layout of the word cloud without altering the semantic structure.
CHAPTER 4

NEW IMPROVEMENTS AND DESIGN

The database system contains three new improvements to improve database performance on the GPU; a caching method for handling the storage of tables in memory, a technique for handling varying size data elements, and a method for handling joins between tables on the GPU. These three innovations provide significant additional functionality over previous research and increase performance over the standard CPU-based implementation.

4.1 Caching Improvements

The caching system, as shown in Figure 4.1, utilizes three levels of data storage; hard drive, CPU main memory, and GPU main memory. At the lowest and slowest level, data is stored in the main database file on the hard drive for long-term storage. Tables that are requested are then also stored in the CPU main memory as level two cache. Tables are swapped into the GPU main memory, which functions as a level one cache, as they are needed and removed only when needed to make room for more recently required tables. Once in GPU main memory, the data is directly accessible by the virtual database engine. A multi-variable replacement algorithm determines which tables need to be replaced in memory.
The first stage of the caching process is reading information from the hard drive. This process requires accessing and interpreting the database file, extracting the information that composes the table requested, and converting the information into a format that is optimal for use on the GPU. These three steps allow the initial loading of data into the cache.

Accessing and interpreting the file involves understanding file layout structure that SQLite uses. The database file itself is divided into sections of uniform size referred to as pages. The pages form a number of linked lists, each page acting as a node, with one linked list for each table in the database. The data within each linked list is stored in a balanced tree structure known as a B-tree.

The B-tree structure is optimal for the standard version of SQLite but not for the GPU database. The B-tree structure allows the standard version to access only a small section of the table in order to traverse to specific rows in the table instead of having to load the entire table from disk. In the GPU version, the entire table needs to be moved to the GPU, so the B-tree structure simply adds to the load time of the first caching stage by forcing extra seeks.
through the database file. Extra seeks are required to jump from one node to the next in the linked list, since there is no guarantee of pages that are logically adjacent in the B-tree being physically adjacent in the database file.

After the B-tree is traversed the data must be converted into a format that is optimal for the GPU. The data in the database file is arranged into rows, where each row consists of a set of values that correspond to the columns in the table. This is referred to as row-major format. This results in hundreds of thousands of very small data groupings. In order to use this efficiently on the GPU the data needs to be re-arranged into column-major format, where the data is arranged into columns, where each column consists of a set of values that correspond to the rows in the table. This converts our data from a large number of small objects to a small number of large objects. This converted data is stored in the CPU’s main memory.

The CPU main memory acts as the base level of the cache. Once tables are loaded into CPU main memory they remain resident for the duration of the database connection’s lifespan. Closing the database connection will empty this cache. The tables are transferred from CPU main memory to GPU main memory if they are needed by the current opcode program.

Not all tables loaded into the CPU main memory will be needed by the opcode program. Some tables are used to make temporary tables, such as indexes, and then the temporary table is used in place of the original table. The original table is not needed by the opcode program. A special pass is made over the opcode program to detect these temporary tables as shown in Figure 4.2. Tables that are only needed to create other tables are not included in the load of tables to the GPU.

Tables loaded into the GPU main memory must have space available for them or the caching process must make space available to them, as shown in Figure 4.3. Tables already on the GPU are not taken into consideration for the amount of free space that must be available on the GPU, but all other tables are evaluated to determine how much free space
Figure 4.2: Table Caching Decision Process
Table Already on GPU?

Add Table Size to Count

More Tables?

Space available on GPU?

Find Aged Candidate

Found Candidate?

Find Unpopular Candidate

Unload Candidate

Space available on GPU?

Load Tables

Figure 4.3: Caching Process
must be on the GPU. If sufficient space is available the tables are loaded normally. If sufficient space is not available on the GPU, then tables must be removed from the GPU cache until sufficient free space is created.

Tables are removed from the GPU cache by finding candidate tables based on age and popularity. The primary criteria is age. Age is determined by how many table references have occurred since a table was last referred to by an opcode program. The tables in the current opcode program are the youngest, and the age of a table is updated every time a table is referenced in the opcode program being evaluated. This means that every table, even in the same query, will have a different age, depending on their order in the opcode program. A cutoff point of ten is used, at which point age is no longer used as a criteria.

This value was chosen after evaluating a number of SELECT statements used in a variety of programs. Past experience as a database administrator shows that most queries have less than ten tables, with the rule of thumb for a large query being about eight tables. More commonly, one to four tables are involved in a query. Naturally, these rules of thumb can vary wildly depending on database design and the type of information used. However, in many cases the cutoff value allows for very popular tables to remain in the top, but tables used rarely quickly sink out of contention.

When no candidate table is found based on age, then the total number of uses is used to choose a candidate for removal. The total number of times a table has been referenced in an opcode table is tracked, and the table with the lowest value becomes the candidate for removal. This value persists even after removal from the GPU, so a table that was popular in the past, but that was removed due to old age, will have a chance to retain that popularity when revitalized by new use. The determination of candidate tables and their removal continues until sufficient space is made available on the GPU.
4.2 Varying Length Data

Processing data whose length is not constant requires special solutions to overcome specific obstacles that the uncertainty of the data size creates. The issues of arrays of separate memory allocations must be resolved by coalescing these arrays into single memory allocations. The issue of assembling disparate meta-data information is resolved by creating special CUDA functions to assemble data on the GPU side. Finally, a solution to the coalescing of result data must be created.

4.2.1 Coalescing Varying Length Data

The issue of arrays of separate memory allocations most often comes in the form of strings. A string is not a discrete object in C/C++, but rather is a pointer to an array of characters. Creating an array of strings results in creating a pointer to an array of other pointers to arrays. This is problematic because memory locations in the CPU memory space have no relation to memory locations in the GPU memory space. Passing an array of pointers simply passes an array of gibberish as far as the GPU memory space is concerned.

The solution is to coalesce all these different arrays into a single array, as shown in Figure 4.4. The individual arrays are retained as separate virtual entities by providing
starting indexes and lengths of each individual array within the coalesced array. While it is possible to create many individual arrays and pass each one individually, the overhead of numerous data copies from the CPU memory space to the GPU memory space rapidly degrades performance. Coalesced arrays reduce this overhead from $O(n)$ to $O(1)$.

4.2.2 Data Loading

The arrays copied over to the GPU must somehow be assembled into a single database object. This assembly must take place on the GPU because only the GPU can interpret or access GPU memory spaces. The pointers to those memory spaces are currently only known to data objects on the CPU, yet only GPU functions can manipulate the GPU memory spaces they refer to. This is not unlike assembling a ship in a bottle (Figure 4.5); tools must be created to manipulate the inside of the memory space from outside the memory space, accessible only through the narrow opening of function calls.

The solution to this issue is to create linking functions that attach one data object to another. For example, the CPU instructs the GPU to make a database object. This object contains pointers to table structures, which in turn contain pointers to column structures to
define a database. The CPU then instructs the GPU to create a table object. The CPU has no mechanism to tell the database object to point to the table object. This issue is resolved by the creation of the linker function. A linker function accepts two pointers as arguments and attaches the second to a specified pointer contained in the object the first pointer refers to.

The process for loading a table is shown in Figure 4.6. The table object is allocated on the GPU and initialized. Then a column types array is created. Next, a linker is called to attach the array to the table object. The process then begins for each column. The columns of the table are processed sequentially. For each column, the data value array is allocated on the CPU, copied to the CPU, and then a linker function is called to attach it to the table object. For string data, which is of varying length, the ”value” contains the starting index of each sub-array, or string of characters, in the coalesced array, which is called a superstring. For this string data, each superstring is allocated, copied, and attached by a linker, and then the array of sub-array lengths is also allocated, copied, and attached by a linker function. Finally, the table object itself is connected to the database object. This means that a single reference can be passed to refer to the database, instead of a large, arbitrary, and potentially unworkable number of references that point to the assorted columns of all the tables in the database.

### 4.2.3 Coalescing Results

With varying sized input, the output is also potentially of any size, both in number of results and in the size of each result entry itself. This introduces two levels of uncertainty, and it is the second level of complexity that requires a new solution.

The traditional way of handling result sets is not robust enough to handle varying length data. The traditional way to coalesce result sets is to allocate an array of result elements for the results and have each thread claim a slot in the array as needed. This handles one level of uncertainty, the number of result elements, but relies on the fixed length of each result
Figure 4.6: Table Load Process
case OP_ResultRow
{
    int ResultSize = 0;
    // d_ResultSize is the global result size

    // Calculate string length
    for(int t=StartRegister; t <= StartRegister+RegisterCount; t++)
    {
        ResultSize += RegisterSize;
    }

    // Get End point and update global result pointer
    int StartPoint = atomicAdd(d_sizeResults,ResultSize);

    // Collect result set.
    for(int t=StartRegister; t <= StartRegister+RegisterCount; t++)
    {
        ResultSet += RegisterContents + Separator;
    }

    // Place result set into result collection starting at point StartPoint
    ResultCopy(ResultCollection, StartPoint, ResultSize, ResultSet);

    // Update the count of total result entries.
    atomicAdd(d_numResults,1);
    break;
}

Figure 4.7: Atomic Coalescing of Results

set to calculate the start point of any particular slot. Without knowing the length of each element it is impossible to choose a slot in the result set until all previous result sets have been calculated.

The solution to this problem requires atomically claiming space in the result set. Atomic functions are functions that ensure that no other thread can interrupt or execute the instruction until the thread currently executing it has finished. Each thread calls an atomic adding function that adds the length of the result set it contains to a global variable. This function also returns the starting value of the variable before the addition takes place. This allows each thread to claim a unique portion of the result set, using the initial value of the global variable as the starting point and the length of the result set it is processing as the end point. Pseudocode for this process, as contained in the ResultRow opcode, is shown in Figure 4.7.
4.3 Table Joins

Joining multiple tables together in the database requires new improvements to handle the massive amount of computation that can result. Each table added to a query adds a potentially exponential increase in the computation required and an arithmetic increase in the amount of data required. The increased amount of data leads to increased memory traffic, which must be minimized if performance is to be maintained.

Two improvements are explored to manage the vast amount of computation required in large joins. The first technique is to attempt to avoid the problem completely by reducing the total amount of computations required. The second technique is to divide the processing into chunks to avoid overwhelming the processing capabilities of the GPU. Both of these improvements increase the performance of the GPU database.

Reducing the amount of computation required by a join is accomplished by the use of dynamic indexes. Dynamic indexes are indexes created at run time to pre-sort the data where the rows are ordered by the column the current opcode program will require for joining the tables. This allows the virtual database engine to search through the data for a specific value with a binary search instead of a linear search. This reduces the run time from $O(n)$ to $O(\log N)$ in any case where an index can be used.

The use of dynamic indexes has been explored in the standard CPU implementation of SQLite, but they have not been brought explored on the GPU. In GPU computing indexes become especially important because they help manage the risk over excess run time in a query. Any thread that runs more than a few seconds runs the risk of being automatically aborted by the display driver for computer system usability reasons. Indexes reduce computation duration by allowing an entire table to be searched with only a few comparisons.

Instances where indexes are not feasible or where many tables, even with indexes, must be traversed, require a method of throttling the flow of threads into the GPU. An example of such a program is shown in Table 4.1. In this table the presence of three Next opcodes indicates that there are three levels of nested loops, each one of which executes the inner
loop multiple times. The number of iterations of the inner loop is equal to as much as the number of rows in the inner table times the number of rows in the middle table, times the number of rows in the outer table. If each table has only a thousand rows, then the innermost instructions are executed a billion times.

The solution to managing this vast amount of processing is to divide the workload into chunks. Each chunk consists of a grid of threads. How data is assigned to threads in a chunk is critical because of the need to maximize the impact of indexing, minimize memory accesses, and limit the processing time of any individual thread.

The assignment of data is begun by dividing tables into two groups. The code for this division is shown in Figure 4.8. Tables that are not indexes, which are ephemeral, are selected to belong to the static group that will have each row in the table assigned to a separate thread. Tables are selected without regard to the total number of threads allowed in a grid or the number of rows they contain. All tables that are not assigned to the static
// Figure out which tables we are iterating through with NEXT
for(int t=0; t<numTables; t++) // Get all row sizes of tables we are using.
    tableRows[t] = dbProfile.tableInfo[tableMeta[t].indexOnCPU].rows;

int blocksize = SQLCUDA_BLOCKSIZE;
int tableIndex = 0;

while(tableMeta[tableIndex].ephemeral) tableIndex++; // Skip to first nonEphemeral table.

int gridsize=(int)ceil(tableRows[tableIndex] / (double)blocksize);

gridsize *= tableRows[tableIndex]; // Set program counter to line P2.
row[p1]++;
break;
}
group are assigned to the iterate group. Programmatically, the static group is indicated by a positive number of rows and the iterate group is indicated by the row count being negative. Tables in the static group will not be iterated through; a single thread will process a single row and only iterate through tables in the iterate group. If there are no tables in the static group, a table from the iterate group is promoted to the static group.

In the virtual database engine a boolean value is set based on whether the row count is positive or negative and stored in the moveCursor variable for the cursor. This boolean is then used by the Next opcode, as shown in Figure 4.9, to determine whether the virtual database engine should advance the cursor a row and jump the program counter to the indicated line.

The second step is the assignment of the static group to chunks. The size of the grid in each chunk is dependent on the number of nested loops in the opcode program. More nested loops are more likely to result in slower execution, so a correspondingly lower number of threads need to be assigned to the GPU in each chunk. Each row of a non-indexed static group table is assigned a thread in a chunk. This results in a single thread only executing for a single unique combination of rows from each of the static tables. The row to be executed in each thread is calculated from the threads block, grid, and chunk numbers. Chunks are then executed sequentially, which means that threads assigned to the same warp, block, or grid will be accessing the most spatially local rows in a table.

The assignment of chunks has the added benefit of speeding memory accesses on the GPU. Each chunk is processing similar data, so each thread is likely to be accessing the same memory addresses. CUDA coalesces memory accesses to improve performance, so when one thread requests data, it is often requesting data that many other threads in the chunk also need. This data will be provided to all the threads in the chunk at the same time, which will result in better performance.
CHAPTER 5

EXPERIMENTAL RESULTS

The results of this thesis are presented in two parts. The first part deals with the results from developing new improvements for GPU databases. Timing data and performance characteristics are discussed. The second part explores the tool set that was created for this thesis. It briefly describes the tool set and demonstrates the use of the tool.

The test machine used for this thesis is a Pentium i7 computer with an NVIDIA GPU. The computer has an Intel i7-3930k CPU running at 3.2 GHz with 64 gigabytes of RAM. The GPU is an NVIDIA GeForce GTX 680. This GPU has 1536 cores, arranged into 8 multiprocessors. The GPU has two gigabytes of global memory and uses a PCI Express 3.0 bus to transfer data between the CPU and the GPU, which has a maximum data transfer speed of 192 GB per second. The test machine uses an Intel 240 GB SSD drive to store the database file. The SSD drive has a sequential transfer read speed of 550 MB per second and a read latency of 80 microseconds.

5.1 GPU Results

The GPU results are divided into three sections; caching performance, string performance, and join performance. Cache performance was analyzed by evaluating timings taken for each discrete step of the SQL execution process and comparing the results before and after
caching to determine the amount of speedup achieve by caching. String performance was compared against the SQLite CPU implementation and against numeric performance of the GPU database itself. The string performance should surpass the CPU implementation and be within an order of magnitude of the numeric performance. Join performance was evaluated by comparing the GPU database performance with the CPU database performance.

5.1.1 Caching Results

Caching occurs in two separate steps; the load to the CPU and the load to the GPU. Caching allowed one or both of these steps to be skipped entirely, removing the time cost of that step entirely from the calculation. The percentage of the total query time that a step consumes indicated the speedup caching provides at that step.

Hard drive load times, as shown in Table 5.1, showed a direct correlation between load time and the raw size of the table. The largest tables had the largest load times and the smallest tables had the smallest load times. While there was an absolute necessity to pay the cost to load these files, it was possible to reduce this load time by both reorganizing the database file into a more linear layout to reduce seek times, and by writing more efficient disk reading algorithms.

Caching minimizes the impact of the long drive load times. Caching the table in the CPU memory space allowed the system to avoid loading from the drive. The CPU memory space was very large and, in the case of this system, can cache the entire database. Caching

<table>
<thead>
<tr>
<th>Table</th>
<th>Rows</th>
<th>Columns</th>
<th>Load Time (ms)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>451661</td>
<td>11</td>
<td>950.867</td>
<td>38,168,951</td>
</tr>
<tr>
<td>DocumentAuthors</td>
<td>491602</td>
<td>2</td>
<td>143.423</td>
<td>4,263,516</td>
</tr>
<tr>
<td>Authors</td>
<td>77318</td>
<td>5</td>
<td>129.940</td>
<td>2,189,673</td>
</tr>
<tr>
<td>Edition</td>
<td>27250</td>
<td>6</td>
<td>10.837</td>
<td>269,031</td>
</tr>
<tr>
<td>PubEdition</td>
<td>27250</td>
<td>3</td>
<td>6.036</td>
<td>202,526</td>
</tr>
<tr>
<td>Publication</td>
<td>341</td>
<td>4</td>
<td>1.595</td>
<td>17,678</td>
</tr>
<tr>
<td>Repository</td>
<td>1</td>
<td>3</td>
<td>1.125</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 5.1: Table Sizes and Load Times
Table 5.2: CPU Cache Performance and Speedup

<table>
<thead>
<tr>
<th>Table</th>
<th>No Cache Time (ms)</th>
<th>Cached Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>1071.445</td>
<td>120.578</td>
<td>8.885</td>
</tr>
<tr>
<td>DocumentAuthors</td>
<td>168.484</td>
<td>25.060</td>
<td>6.723</td>
</tr>
<tr>
<td>Authors</td>
<td>155.367</td>
<td>25.427</td>
<td>6.110</td>
</tr>
<tr>
<td>Edition</td>
<td>24.264</td>
<td>13.428</td>
<td>1.807</td>
</tr>
<tr>
<td>PubEdition</td>
<td>13.895</td>
<td>7.859</td>
<td>1.767</td>
</tr>
<tr>
<td>Publication</td>
<td>12.747</td>
<td>11.152</td>
<td>1.143</td>
</tr>
<tr>
<td>Repository</td>
<td>8.788</td>
<td>7.663</td>
<td>1.146</td>
</tr>
</tbody>
</table>

The impact of caching tables in the CPU memory space corresponded to table size. Table 5.2 shows the execution times with and without level two (CPU) caching and the speedup achieved in a SELECT statement that returns the entire table with no selection criteria. The largest tables took a significant portion of the total execution time to load the table, while the smallest tables showed that caching had little impact. Even with the smallest tables, however, CPU caching was still superior to no caching.

Caching to the GPU allowed the cost of setting up tables on the GPU, copying table data over, and linking GPU data structures together to be avoided. The use of the GPU memory space as Level-1 cache allowed the elimination of this performance penalty for some tables, but the penalty must be paid when a cache miss occurs.

Table 5.3 shows the execution times with and without level one (GPU) caching and the speedup achieved through the use of the GPU cache. The GPU cache speedup was largest
Table 5.4: CPU vs. GPU Comparison with Selection Criteria Performance

for small tables. This was because the majority of the time in small tables was spent on overhead such as setting up tables and allocating memory. These operations have a fixed cost in addition to the cost per unit of data processed. The smaller the table the larger the fraction of the entire query process this fixed overhead was.

5.1.2 Varying Length Data

String performance was evaluated in two ways; by comparing string performance of the GPU database against string performance of the CPU database, and by comparing string performance of the GPU database against numeric performance of the GPU database. The first comparison demonstrated that the GPU database exceeded the CPU implementation. The second comparison showed that the string implementation was efficient and did not suffer serious performance degradation due to the use of varying sized data.

Performance comparisons were made between the CPU and GPU databases using two
categories of SQL statements. The first category was SQL statements with selection criteria, using both single and compound criteria. The criteria used strings to select the rows to be returned. The third category was SQL statements with joins, using both single and complex joins. These statements used strings both in criteria to select rows and in the return values. The tables used were the Authors and the Document tables. These tables were selected because they contained a large volume of string data and they represented both medium and large sized tables.

The SELECT queries with conditional clauses showed a performance gain by the GPU database over the CPU database. The execution times for these queries are shown in Table 5.4. The single criteria of Query 1 and Query 2 demonstrated the speedup achieved with string condition clauses. These queries also demonstrated that larger tables, which have more processing to perform, achieved a greater speedup. The multiple conditional clauses of Query 3 and Query 4 demonstrated that a greater speedup was achieved when more conditions were evaluated. This indicates that each string condition statement was faster in the GPU database than in the CPU database, and additional condition statements would result in an even greater speedup. Query 5 showed the performance with multiple joins between the CPU database and the GPU database still reflected a significant speedup.

String performance must be comparable to integer performance in the GPU database. This was evaluated by comparing the performance of similar queries on the GPU database. The conditional clauses of the statements were written with one version using strings to select records and the other version using numeric values on the same table. The results are

<table>
<thead>
<tr>
<th>SQL Statement Description</th>
<th>Numeric (ms)</th>
<th>String (ms)</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authors Table, Single Criteria</td>
<td>3.42</td>
<td>6.73</td>
<td>1.97</td>
</tr>
<tr>
<td>Document Table, Single Criteria</td>
<td>6.32</td>
<td>7.44</td>
<td>1.17</td>
</tr>
<tr>
<td>Authors Table, Multiple Criteria</td>
<td>3.74</td>
<td>5.01</td>
<td>1.34</td>
</tr>
<tr>
<td>Document Table, Multiple Criteria</td>
<td>6.73</td>
<td>8.13</td>
<td>1.21</td>
</tr>
<tr>
<td>Complex Join</td>
<td>271.36</td>
<td>298.94</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 5.5: Numeric vs. String Conditional Statement Performance
<table>
<thead>
<tr>
<th>SQL Statement Description</th>
<th>Bytes Processed</th>
<th>VDBE Execution (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT id FROM Authors</td>
<td>505,380</td>
<td>1.31</td>
</tr>
<tr>
<td>SELECT firstname FROM Authors</td>
<td>482,029</td>
<td>1.02</td>
</tr>
<tr>
<td>SELECT id FROM Document</td>
<td>2,791,250</td>
<td>6.06</td>
</tr>
<tr>
<td>SELECT title FROM Document</td>
<td>14,719,204</td>
<td>11.76</td>
</tr>
</tbody>
</table>

Table 5.6: Time of Virtual Database Engine Execution

shown in Table 5.6 and demonstrated that the string performance of the GPU was similar to the numeric performance. The queries demonstrate that the performance between string and numeric SQL SELECT statements was well within an order of magnitude, and was less than a factor of two in difference.

The performance of coalescing the results when varying length data was present was determined by comparing the performance of fixed data results with varying length data results. This comparison was done with two representative tables; the Authors table and the Document table. One field was selected in each statement, returning either a fixed length value (id) or a varying length value (firstname and title). The execution time of the virtual database engine itself was evaluated, without consideration for other issues such as caching, data transfer time, or table setup.

The performance of the coalesced varying length data matched the performance of the fixed length data. The Authors table showed slightly faster execution time for the varying length data. This was appropriate because there was slightly less data to process when coalescing the results. In the much larger Document table, the VDBE took almost twice as long to run. However, this statement returned five times as much data, which must be coalesced into the result set. The larger result set explains the longer run time, indicating the success of this coalescing method.

5.1.3 Join Performance

Join performance was evaluated based on the performance of the GPU SELECT statements that contained joins. The performance of the query on the GPU database was compared
<table>
<thead>
<tr>
<th>ID</th>
<th>SQL Statement</th>
<th>GPU (ms)</th>
<th>CPU (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SELECT * FROM Document d, Authors a WHERE d.title=a.firstname</td>
<td>62.85</td>
<td>2737.81</td>
<td>43.52</td>
</tr>
<tr>
<td>2</td>
<td>SELECT d.id FROM Document d, DocumentAuthors da WHERE d.id=da.documentid AND da.documentid=813</td>
<td>5.83</td>
<td>223.74</td>
<td>38.38</td>
</tr>
<tr>
<td>3</td>
<td>SELECT a.id FROM Document d, DocumentAuthors da, Authors a, Publication pu WHERE (d.id=da.documentid AND a.id=da.authorid AND pu.id=pe.pubid) AND ((da.documentid=81372 AND d.repositoryID = 1) OR pu.id &lt;2 OR pu.publisherid &lt;2 OR a.id &lt;2 OR d.pubeditionID &lt;2)</td>
<td>278.83</td>
<td>8261.43</td>
<td>29.63</td>
</tr>
</tbody>
</table>

Table 5.7: Join Performance

to the performance of the query on the CPU database. Queries that join tables based on indexes and without the benefit of indexes were compared. The results of these queries are displayed in Table 5.7.

The effectiveness of a query without the use of a dynamically created index was demonstrated in Query 1. This query joins two tables on a text field. The results of this query showed a significant increase in performance of the GPU database over the CPU database. The execution of Query 2 allowed the testing of a large data set where indexes were used to accelerate the join. The query showed significant speedup on the GPU database compared to the CPU database, indicating the success of the join methods used in the GPU database. The results of Query 3 also showed a significant speedup, but the speedup was not as great as the speedup of the smaller join statement. This reflected the relatively large time cost of creating indexes compared to the relatively small cost that execution of that index brings; indexes are extremely fast but making them can be slow. In Query 3 more indexes are created, and this has a small negative effect on the efficiency of the query.
5.2 Toolset

The toolset devised for this thesis provides a user-friendly interface for accessing Humanities data stored in a database, mining that data, and displaying visualizations that summarize the results of the mining. A graphical user interface simplifies the process of querying the database. The complexity of data mining is reduced to an automatic choice of the appropriate data mining when a visualization is selected. The toolset is created using DirectX11 for drawing graphical objects and Microsoft foundation Classes to provide the user interface.

The toolset, named Herodotus, is laid out using a ribbon, which is a set of toolbars placed in multiple tabs that has been popularized by Microsoft products, and three panels as shown in Figure 5.1. The panels are used to navigate the three stages of using the tool; selecting data from the database, choosing a visualization, and viewing the result. The ribbon provides menu options for executing the program and modifying the resulting visualizations.

The Table Select tab in Figure 5.1 is used to assist the user in navigating the database.
The user may be inexpert in the use of SQL. This tab is intended to provide some guidance in writing SQL queries for such users. The tab shows all the tables available in the database and all the fields in the tables. To the right of each field name there are edit boxes that allow the user to enter criteria for that particular field. This simplifies the building of complex criteria by organizing the choices made by the user in a visually clean manner.

The assembly of the criteria into a SELECT statement is automated. The user selects the checkbox next to a field and the criteria associated with that field are automatically added to the SELECT statement. Joins and required tables are automatically determined by the program and added to the SQL statement. Unselecting a field removes the criteria from the SELECT statement and, if joins are no longer needed, removes joins as well. The keywords SELECT, FROM and WHERE are automatically color-coded to assist in reading the SQL statement. The automation of SQL generation simplifies the task of writing SQL statements.

The SELECT statement may be edited manually, overwriting the statement generated by the checkboxes and edit boxes. Manually changing the SQL statement does not alter the checkboxes or their associated edit boxes. In cases where complex conditionals are required, or for users who are adept at SQL and wish to eschew the graphical interface, SQL commands can be entered directly into the text box. SQL will be checked for validity before execution.

The Visualization Tab allows the selection of the type of visualization that the user wishes to view, as shown in Figure 5.2. The user has two options; a word cloud and a three-dimensional spatial graph that is referred to as the “Twine Graph”. Sample images are provided to indicate the general appearance of the visualization. The user selects the visualization that will be applied to the data selected by the query in the Table Select tab.

The selection of data mining options is hidden from the user. Specific visualizations require complex mined data. Rather than requiring the user to navigate this complexity, the appropriate data mining is determined based on the visualization chosen and the data mining is automatically accomplished.
After selection of the desired visualization is made, the user can enable the visualization by pressing the “Execute” button. Execution duration is dependent on the amount of data chosen. A “Halt” button is provided to end the animation of the visualization. Clicking the “Halt” button does not halt the program itself.

The Result Tab contains a window into which the visualizations will be drawn. Other panels may be selected and viewed without disrupting the visualization on the Result Tab. The visualization may be zoomed in and out by using the mouse scroll wheel and moved along the X-axis and Y-axis by using the left and right and the up and down arrow keys on the keyboard. This enables the user to focus attention on specific areas of the visualization.

The word cloud visualization, as shown in Figure 5.3, demonstrates two different values associated with the word. The size of the word block indicates the number of times which the word appears in the text data. The number of times the word appears in each document is summed, and then the values of all words in the cloud are compared to determine the largest and the smallest number of occurrences of words in the word cloud. These two values form the range of possible word occurrences. The percentile rank of a word within this range is then determined using the formula: \( \frac{\text{Word Occurrences} - \text{Minimum}}{\text{Maximum} - \text{Minimum}} \). This percentile is multiplied by the default size of an object in the word cloud.
and a constant minimum size is added to ensure a minimum object size. The formula used is \((\text{Percentile Rank} \times \text{Default Size}) + \text{Minimum size}\). The color is a heat map that indicates what percentage of documents the word appears in. A deep red object appeared in the fewest documents, and a deep green object appeared in the most documents, with the colors shading from red, to orange, and then to green to indicate percentages in between.

The word cloud visualization has several options for viewing that can be chosen from the ribbon menu, as shown in Figure 5.3. The user has the option to toggle between a straight view and an offset viewpoint by selecting the “toggle camera” button to create the alternate view shown in Figure 5.5. This enables the use to achieve a different aesthetic which is more visually appealing to some users. The user also has the option to reduce the number or words or increase the number of words shown by clicking on the appropriate buttons. The
Figure 5.4: Word Cloud Example

The layout may be recalculated to find alternate cloud shapes by clicking the “Layout Change” button. This enables the user to choose different layouts that may be more visually pleasing. The "Reset Display" button restores the cloud to the starting state.

Navigation may also be accomplished through direct mouse manipulation or use of the keyboard. The mouse scroll wheel zooms the image in and out by moving it forward or backwards on the screen. Clicking on the image and dragging across the screen will grab and rotate the image, allowing the user to choose custom angles. The arrow keys shift the entire cloud up and down or left and right. These navigation options allow the user to customize the viewing experience.

The other visualization is the twine graph, as shown in Figure 5.6. The twine graph maps words to objects in three dimensional space. The object shape shows the frequency of
Figure 5.5: Offset Viewpoint in Word Cloud
documents the term appears in; a sphere indicates appearing in the bottom third percent of documents, a cube indicates that the word appears in between a third and two-thirds of the documents, and a tetrahedron indicates that the word appears in more than two-thirds of the documents. The distance from the center point of the cloud of objects indicates the number of times the term appears in the collection of documents. This visualization allows the user to easily categorize the frequency of words in documents by observing the shapes of the words. The positioning of the words in space also shows patterns in the terms in the document set.

This visualization also contains a number of customizations available to the user. The user may toggle any of the three shapes on and off. The number of objects displayed may be increased or decreased. The spacing of the objects can be modified, either by altering the
spacial padding around each object or by altering a multiplier to the radius of each object from the center. This change in spacing can reveal patterns by bringing words with patterns in the order of their frequency together in space. To more easily visualize the center, a center point indicator can be toggled on and off. Two animations are included; one that rotates the objects in space around the center point and one that dynamically increases and then decreases the object spacing in a continuous loop. Rotating the object allows the user to view all view of the arrangement of the objects in space, potentially revealing patterns in the arrangement that are not visible from one specific angle. The ability to animate the spacing of objects also provides a way to rearrange the objects in space while keeping them at the same distance from the origin.
CHAPTER 6

CONCLUSION

In this thesis, new improvements were demonstrated for accelerating database performance through the use of a GPU. A caching scheme was developed to resolve the issue of data transfer and ensure that the correct data was always available to the database despite the space limitations of GPU main memory. A high-performance solution was found for manipulating and processing varying length data such as strings. An efficient method for joining multiple tables was developed. These three improvements were shown to significantly increase the performance of a GPU database.

The caching data showed that caching can have a dramatic effect on the performance of GPU queries. The ability to cache reads from the database file improved all sizes of tables, with the largest improvements occurring in the largest tables. The larger the table, the greater the speedup possible by caching at the CPU level. The largest tables also take the largest absolute time, so the importance of increased performance for these large tables is magnified.

Caching data to the GPU demonstrated the importance of managing data flow between the CPU and the GPU. The speedup ranged between 1.4 times and 4.1 times, which implies that the penalty for a cache miss on the GPU is a slowdown from the optimal speed by a factor of the same amount. This clearly shows the critical importance of caching. Where
the data is located matters a great deal when evaluating GPU database performance and caching helps to ensure that the data is available at the most efficient location.

String performance showed the value of indexed strings as a mechanism for processing varying length data. The goal of the new methods for handling varying length data was to perform the same operations as fixed length data without a large performance penalty. The string operations were effective at performing the same operations with little or not loss in performance. The coalescing of data into a result set also demonstrated excellent performance. Despite the added complexities of coalescing varying length results, the performance of varying length result coalescing was exactly on par with the results of fixed length result coalescing.

The largest factor influencing performance of SELECT queries was not the differences between fixed and varying length data, but the size of the data set used in the query. As table size increased, the difference in performance between fixed length and varying length data shrank, to the point where performance was almost identical for large joins.

The size of the result set is also an important factor in determining GPU database speed. This is due to the need to coalesce the data on the GPU and transfer it back to the CPU. Both of these steps are potential bottlenecks, and the change from selecting a single field to every field resulted in an execution time a third again as long as the single field cast. Tables with more fields would also result in even larger slowdowns.

Factors that had little influence on execution time were conditional clauses, processing indexed tables, and data type. This is due to the high performance of the improvements used in the GPU database. The complexity of the conditional clauses in the WHERE clause had very little impact on execution time, even in cases where numerous logical operations were performed. Indexed tables, due to the fact that only a very small subset of the rows are processed, also had a marginal effect on execution time. The type of data, whether it was numeric or string, had little impact on performance due to the use of indexed superstrings. String operations were slightly slower, but not dramatically so.
There is are two caveats that must be mentioned when dismissing these three factors as influencing execution times. First, when the conditional clauses forced a table that was using an index to perform evaluations on fields that were not part of the index, the conditional clauses were able to impact performance by removing much of the benefits of the indexes. This performance, impact, was absolute instead of relative; the statement took longer to run on the CPU database as well as GPU database, and the GPU database achieved an even greater percentage speedup on the unindexed join than the indexed join.

The design decision on how to handle strings established the most effective method for developing a GPU database. This method is to avoid using individual strings as much as possible, instead relying on concatenations of strings and indexes of the strings. Additionally, all operations on strings are best served by operating on indexes to a subsection of a superstring and never actually attempting to move or modify the string itself; only the indexes pointing to it. Indexed strings are a key factor to good GPU database performance.

Indexed strings, by which we mean strings that are referred to as meta-data about a superstring that itself is not modified, are not the traditional way to handle string in programming. Normally, strings are handled as individual entities. Each string is created, manipulated, moved, and deleted as a separate object. CUDA contains the capability to do exactly this, conforming to the traditional programming model. However, this technique was found to be extremely inefficient on the GPU.

Joins were also shown to be effectively implemented on the GPU. Assuming that one row is assigned per thread, one of the greatest risks of joins is that the number of threads needed will explode exponentially. This risk is best managed in two ways. The first is the dynamic creation of indexes. This logarithmically reduces the number of threads required. In cases where indexes are not feasible, the required work must be chunked into smaller blocks and fed to the GPU in pieces. These two methods together solved the issue of join performance.
6.1 Future Work

There are several areas of future work in both the GPU database and in the resulting tool created for visualization. The GPU database can be expanded in performance and capability. The visualization tool can be expanded to include new data mining and new visualizations.

The largest increase in performance for the GPU database can be achieved by improving the method by which indexes are created. The cost of creating indexes can be as much as two-thirds of the execution time for the most complex joins sampled in this thesis. This is primarily due to the sorting method used in creating the indexes. The implementation of a highly efficient sorting algorithm for creating indexes could result in a three-fold speedup of large join statements.

Caching could also be improved by the use of a predictive pre-caching scheme. This would expand the opportunities to benefit from the performance improvements of caching. In interactive systems, preemptive caching based on user input as SQL statements are being entered could also increase performance by allowing the caching to be done before the SQL statement is executed.

The database can be expanded to further implement SQL features on the GPU. These additional features, such as INSERT, UPDATE, and DELETE statements, may reveal new challenges when coupled with joins and varying size data types. More complex indexing options could also be supported in the database, which may well provide new avenue of research. The ability to add stored procedures to the GPU database, written to run on the GPU, would open up a whole new arena of research.

The investigation of alternate methods of accessing GPU’s programmatically is another area of future research. CUDA only supports NVIDIA video cards, and ATI cards have a strong reputation for better integer performance than NVIDIA cards. The use of alternate programming models, such as DirectCompute or OpenCL, may also provide altercate solutions to some of the problems that have been addressed specifically with CUDA.

Within the visualization tool, there is room to build modules to do different types of text
mining, such as semantic analysis. Providing more types of text data mining would require
the creation of new visualizations. The inclusion of Semantic analysis or word closeness
evaluation would allow the Humanities researcher to probe the data in the database in
new ways. The visualizations in the system could be improved by adding new dimensions
to the data represented. The layout in the word cloud could be modified to represent
word affinity. These new visualizations would give Humanities researchers more tools to
conduct exploratory research. New visualizations could be added to represent new data
mining summations. Additionally, more configuration options could be added to the existing
visualizations, such as the ability to refine which words are excluded in the word frequency
displays, regardless of their ranking, or the ability to force the inclusion of words, even if
they would not otherwise meet the criteria for inclusion.
BIBLIOGRAPHY


[38] Sara Steger, “Formulaic Emotion: Reading Victorian Deathbed Scenes from a Distance,” The 21st Joint International Conference of the Association for Literary and
Linguistic Computing, and the Association for Computers and Humanities, College Park, MD, 2009, pp. 32-34.


