STUDENT PAPER: GPU Acceleration for SQL Queries on Large-Scale Distributed Systems

Linh Nguyen  
Hampden-Sydney College  
1 College Road  
Hampden-Sydney, VA 23943 USA  
nguyen16@hsc.edu

Paul Hemler  
Hampden-Sydney College  
1 College Road  
Hampden-Sydney, VA 23943 USA  
phemler@hsc.edu

ABSTRACT

General purpose GPUs are a powerful hardware with a number of applications in the realm of relational databases. We further extended a database framework designed to allow for GPU execution queries. Our technique is novel in that it implements Dynamic Parallelism, a new feature in recent hardware, to accelerate SQL JOINs. Query execution results in 1.25X speedup on average with respect to a previous method, also accelerated by GPUs, which employs a multi-dimensional CUDA Grid.

More importantly, we divided the queries to run on multiple BW nodes to investigate the scalability of both SELECT and JOIN.

Keywords

GPGPU, BWSIP, CUDA, SQL, Distributed Computing

1. INTRODUCTION

A relational database management system (RDBMS) manages the organization, storage, access, security, and integrity of data. Manipulating an RDBMS requires the use of Structured Query Language (SQL), an industry-standard declarative language capable of performing very complex queries and aggregations over data sets. In an RDBMS, these data sets are organized in structured tables, and SQL serves as an intermediary between a client program and its databases. The explosive growth of various types of unstructured data has called for the next wave of innovation in data storage, management, and analysis, otherwise known as Big Data.

MapReduce presents one such innovation. Initially developed by Google, MapReduce is a programming paradigm widely used for data intensive and large-scale data analysis applications [10]. The architecture is simple abstraction that allows programmers to write a map function that processes key-value pairs associated with the input data. The reduce function then merges all the intermediate values associated with the same intermediate key. The programming model involves three phases: Map; Shuffle and Sort; Reduce.

In some cases, a MapReduce framework has replaced traditional SQL database, though the advantage of one over another remains a debated topic. Both approaches are very general methods through which data can be processed. In fact, there are efforts to combine the two in order to ease programmers’ learning curve. In particular, Apache Hive allows programmers to write queries in SQL-like fashion and then converts the queries to map/reduce [1]. Regardless of the specifics, SQL remains highly applicable in the Big Data era thanks to the power of its declarative syntax. As such, significant efforts have been made in improving the performance of SQL, targeting parallel heterogeneous hardware architectures, which incorporates many processor types in a single machine [5, 6, 7, 8, 11, 12, 13, 14, 16].

The SQL model of data organization fits a parallel execution model extremely well since different processors can work on different portions of a table simultaneously. Among relevant attempts, the most promising approach currently utilizes the newest generation of Graphics Processing Units (GPUs), which are implemented as massively parallel architectures designed for rapidly rendering complex and realistic graphical scenes. More importantly, several general purpose software development frameworks for programming GPUs have become standard. Two such frameworks are Compute Unified Device Architecture (CUDA) [19] and the Open Compute Language (OpenCL) [17]. These frameworks allow applications to simply and effectively harness the power of the GPU. In addition, they have also become a common method of accelerating many compute-intensive applications [9].

This project aims to extend a GPU-based database to allow for the execution of SQL queries on multiple GPUs contained within the Blue Waters (BW) supercomputer system [18]. The specific focus of this project targets two common but extremely important SQL queries: SELECT and JOIN. While most previous papers utilize parallel primitives, our implementation executes with an opcode virtual machine model. In short, the key contributions of this paper include:

- **Dynamic Parallelism Approach** - Earlier GPU hardware did not include the ability to launch new GPU functions directly from threads. The BW compute nodes provide newer GPUs with the Kepler Architecture, which includes this important feature, referred to as Dynamic Parallelism (DP). We employ DP to enhance hardware utilization in accelerating JOIN.

- **Multi-GPU Configuration** - To the best of our knowledge, all previous work has shown speedups on a single-
2. RELATED BACKGROUND

This section provides overviews of SQL queries, and MapReduce. It also discusses current solutions in parallel SQL execution on the GPUs.

2.1 SQL SELECTs and JOINs

In the following discussion, we denote a table as $T_i$ and each column in a table as $c_i$ where $i$ is their corresponding enumeration.

A SELECT statement extracts data from a database in an organized and readable format. A typical SELECT statement is accompanied by clauses such as FROM, WHERE, and ORDER BY. These clauses specify the conditions on which the data is filtered. A predicate includes one or more of these conditions.

$$T_1 = \begin{array}{c|c} c_1 & c_2 \\ \hline 1 & w \\ 2 & z \\ 3 & z \end{array} \quad \Rightarrow \quad T_2 = \begin{array}{c|c} c_1 & c_2 \\ \hline 2 & z \\ 3 & z \end{array}$$

Figure 1: The result rows of a SELECT query: SELECT $\ast$ FROM $T_1$ WHERE $T_1.c_1 \geq 2$

A JOIN query requires at least two tables that share a common attribute, referred to as a foreign key. While there are many types of JOINs, we focus only on the cross-join, denoted $\bowtie$, since it is the simplest JOIN query but still embodies all the computational characteristics of other JOINs. A cross-join computes the Cartesian product of all the keys in the relevant tables. Figure 2 highlights the result rows of a cross-join between $T_1$ and $T_2$ based on the predicate $T_1.c_2 = T_2.c_2$, where $c_2$ is the foreign key. While the result consists of nine rows in total, the predicate reduces the result to just three rows.

A SELECT query involves a loop that examines every row in a table. A JOIN entails a nested loop, where for each row in the first table, the corresponding loop iterates over the entire second table and emits rows that match the predicate. Clearly, this entire process is computationally demanding for very large tables with millions of rows.

$$T_1 = \begin{array}{c|c} c_1 & c_2 \\ \hline 1 & w \\ 2 & z \\ 3 & z \end{array} \quad T_2 = \begin{array}{c|c} c_2 & c_1 \\ \hline x & 5 \\ z & 6 \\ w & 7 \end{array} \quad \Rightarrow \quad T_1 \bowtie T_2 = \begin{array}{c} c_1 \bowtie c_2 \\ \hline 1 \bowtie w & 5 \\ 2 \bowtie z & 6 \\ 3 \bowtie w & 7 \end{array}$$

Figure 2: The cross join $T_1 \bowtie T_2$: SELECT $\ast$ FROM $T_1, T_2$ WHERE $T_1.c_2 = T_2.c_2$

2.2 MapReduce

As previously mentioned, the MapReduce paradigm is an active research area. The three main phases are shown in Figure 3.

Map Phase: The input data is partitioned into splits and each is assigned to Map tasks to be processed in separate nodes in the cluster. These Map tasks perform computations on each input key-value pair from its assigned partition of data. Finally, they generate a set of intermediate results for each key.

Shuffle and Sort Phase: This phase sorts the data generated by the Map tasks from other nodes and divides this data into regions to be further processed by the Reduce task. Therefore, all Map tasks must complete prior to this phase. In addition, this phase distributes the data as necessary to nodes where the Reduce tasks will execute.

Reduce Phase: The Reduce tasks perform additional operations on the intermediate data by merging values associated with a particular key to a smaller set of values to produce the output. The number of Reduce tasks does not need to be equal to the number of Map tasks. For more complex data processing procedures, multiple MapReduce calls may be linked together in sequence.

MapReduce is a simple and scalable approach to achieve high speed and efficient parallel processing over a massive amount of data.

2.3 Parallel SQL Databases
There have been other works in the literature [7, 8, 11, 14] that employ the parallel primitives such as scan, sort, or filter to process queries on the GPUs. Each of these primitives is a kernel launched on a given set of data in an order specified by programmers. Every kernel then needs to retrieve relevant data from global memory. Fundamentally, this approach entails many memory accesses and consequently affects performance.

We build directly on the work presented in [3, 4, 5, 6]. This line of research proposes using a virtual machine (VM) instead. From a high-level point of view, a VM is a sequence of jump instructions inside the GPU. Each instruction is identified by an opcode and performs a certain operation. Each thread is mapped to a data point, small enough to be loaded directly into a register.

The source code of the Virginian database, which embodies this approach, is open source [4]. While inspired by SQLite [2], the initial implementation presents a completely custom VM, which we will describe in Section 3. We modified this VM to accommodate the distributed architecture of the BW system.

3. IMPLEMENTATION

The program first builds an abstract syntax tree (AST) from the SQL query. The AST is then processed in several passes to generate a virtual program, namely a set of jump instructions that represents the query. The details of the parsing algorithm and code generations are documented in the Virginian database page [4].

3.1 Virtual Machine Infrastructure

A VM is implemented as a CUDA kernel that executes a sequence of instructions procedurally. Each instruction is identified by an opcode and contains four registers. Each opcode serves a certain functionality.

The aforementioned work partitions a SQL table into chunks of data, called the tablet [4, 5]. Each tablet is a fix-sized group of rows. As a result, accessing an entire table of data may involve multiple tablets, but accessing a single row of data involves only a single tablet. Partitioning is beneficial at two levels of parallelism, especially for SQL SELECT. First, for a heavily distributed table, each of this table’s tablets resides on a different node and thus can be processed independently. Second, for each tablet, threads in a CUDA grid handle their corresponding rows.

Since SELECT operates only on single source table, an extension was developed to this model to support operations over two or more tables, namely JOIN [3]. CUDA’s organization of threads into a grid of up to three dimensions coincides with the structure of a Cartesian product where a two-table join corresponds to a two dimensional grid. We recall the example from Figure 2, where two tables $T_1$ and $T_2$ are cross-joined on the same foreign key. Here, the $x$-dimension of the CUDA grid corresponds to the row index of $T_1$, while the $y$-dimension corresponds to row index of $T_2$.

The shortcoming, however, is that this approach is limited to joining at most three tables since this is the maximum supported dimensions of a CUDA grid.

This method, which we will now refer to as the Grid method, is used as the baseline performance for our novel technique for accelerating JOIN, which will be discussed shortly. In addition, we scale both SELECT and JOIN to execute on multiple node and compare the performance with that of a single node.

3.2 Dynamic Parallelism for SQL JOINs

DP is a new functionality provided by the Kepler Architecture [19]. DP allows kernels to be launched from the device without going back to the host. The kernel, block or thread that initiates the launch is referred to as parent. Also, the kernel, block, or thread that is launched is referred to as the child. DP allows explicit synchronization between the parent and the child through a built-in device function. Launches can be nested from parent to child, then child to grandchild and so on. The deepest nesting level that requires explicit synchronization is the synchronization depth. Parent and child kernels have coherent access to global memory. Shared and local memories are exclusive for parent and child kernels.

The JOIN algorithm involves at least two data arrays, taken from the related data tables. Each thread takes one element from the first array in the JOIN predicate and finds the matching keys from the other array. DP implementation launches a kernel to gather the result elements in parallel for each thread.

3.3 Multi-GPU Configuration

To further improve the performance of large-scale data processing, it is essential to share workloads among distributed compute nodes equipped with GPUs. As discussed in Section 1, this project is the first to examine the possibility of breaking up a data set and running a query concurrently on multiple GPUs. Our custom data structure is useful in the context of a heavily distributed database by enabling efficient management of data between networked machines. In our implementation, tablets play a major role since they vastly simplify the process of moving data to and from different nodes. Tablets allow each node to operate exclusively on known-size records.

![Figure 4: Query plan for SQL SELECT](image)

A table is first divided into tablets. This step is trivial thanks to the partitioning scheme of a SQL table. Each table is sent to a XK compute node through MPI. On each node, each tablet is then processed by the same virtual machine row by row on the node’s GPU. After producing its corresponding set of result rows, each node passes these rows back to the master node, which then reorganizes these rows and emits the final table to the user. Figure 4 illustrates this approach for the SELECT queries.
Similarly, the two tables in a JOIN query are each divided into an array of tablets. Figure 5 demonstrates this plan. Table $T_1$ contains $N$ tablets, and table $T_2$ contains $M$ tablets. Since a JOIN statement is a Cartesian product of two arrays of data, each node is responsible for a different section of the cross product. For instance, Node 0 computes $T_1.Tablet_0 \bowtie T_2.Tablet_0$. The results of these computations are again passed to the master node in the same procedure as in Figure 4.

Splitting data across multiple nodes has a two-fold advantage: data can be processed more quickly and a larger quantity of data can be processed. The database can now support much larger tables while retaining the same performance. For the same data set, more computing powers reduce the processing time.

4. RESULTS

We adopted the test data from [4, 3, 6], which includes 8 million rows randomly generated numerical values. The columns consists of an integer primary key and 2 columns each with distribution in [-100,100], a normal distribution with standard deviation 5, and another with standard deviation 20. Each of these distributions was generated once for a 32-bit integer column and a IEEE 754 32-bit floating point column. The GNU Scientific Library was used to ensure the quality of the random distribution.

A suite of ten different queries were written to thoroughly test the different characteristics for both SELECT and JOIN. Hence, there are twenty queries in total. For SELECT, five of the queries for each suite involve integer values; the rest of the suite test floating point values. For JOIN, only one query operates exclusively on integer values; one query tests floating point exclusively; the rest of the suite test a arithmetic and conditional statements with a mixture of both integer and floating points.

We have little reason to believe results would change significantly with realistic data sets, since all rows are checked. Also, both textual and non-textual data are ultimately represented by numeric values. Besides, we have tested virtually all basic possible combinations of common case queries. This section also highlights the educational values of our project.

4.1 Performance

We analyze in detail the runtime growth of our implementation and then discuss performance improvement relatively to the baseline as well as the scalability of our implementation as a function of the number of nodes.

4.1.1 SELECT

The characteristics of runtime growth on a single node has been discussed [5]. We therefore focus mainly on the scaling factor of the implementation. Figure 6 visually presents the speedups observed as we scale our program. Single-node execution was predictably slower by factors proportional to the number of nodes used in computation.

![Figure 6: Scaling factors for SELECT and JOIN](image)

4.1.2 JOIN

Test were performed using two 3500 row tables containing randomly generated values in the same layout as for SELECT. We chose this number of rows since the Cartesian product of two tables with 3500 rows each will space $3500 \cdot 3500 = 12,250,000$ rows, larger than the test number of rows for SELECT.

Figure 8(a) graphically demonstrates the differences in running times using DP and Grid methods. The mean execution time for all ten queries for Grid is 0.2526 seconds and 0.233 seconds for DP. We noticed that floating points are slightly faster and reported the average between integer and floating points. Figure 8(b) depicts the speedups of DP with respect to Grid, which averaged to be 1.245X.

Scaling SQL JOINs on distributed is less straightforward than with SELECT. Since the Cartesian product is a two-dimensional product, we chose the sizes of the two tables in Figure 5 as \{1 \times 2, 2 \times 2, 3 \times 2, 4 \times 2, 4 \times 3, 4 \times 4, 2 \times 7, 4 \times 4\}. These numbers correspond to the same number of nodes for SELECT and their speedups are also displayed in Figure 6. We chose these dimensions to examine the effects of the ra-
Figure 7: As fewer rows in the Cartesian product are filtered, the GPU becomes less efficient.

Proportions of Cross Product

Proportions of Rows Returned vs. Execution Time

Figure 8: Variations of two different methods for accelerating JOINs.

as workload imbalance. Our implementation automatically solves the first problem since data is evenly divided into tablets. The latter is harder to address since imbalance may occur due to the specific characteristics of an application. For instance, we chose a uniform distribution for our test data since the distribution has equal probability for any random region of data, thus minimizes the chance for load imbalance. It is critical for the programmer to be aware of these challenges in order to address them properly.

4.2 Educational Values

The use of GPU in HPC is becoming extremely popular due to the high computational power and high bandwidth coupled with the availability of (de facto) frameworks. However, effectively programming a hybrid system with MPI and CUDA is especially difficult as the growing complexities of applications require more and more processors. Because the communication patterns and resource scheduling are common to a wide range of domains, we generalize our experiences and hope that they could be helpful to future programmers faced with similar problems.

4.2.1 Multi-GPU Programming

MPI and CUDA combine the two parallel programming frameworks enables solving problems with a data size too large to fit into the memory of a single GPU, or that would require an unreasonably long compute time on a single node. Also, this combination allows programmers to efficiently accelerate existing MPI applications with GPUs.

In multi-GPU programming, an MPI launcher starts multiple instances of an application and distributes these instances across the nodes of the BW. Each instance then launches its own CUDA kernel. A major problem arises when each node does not receive an even amount of data or does not return the same amount of data, referred to as workload imbalance. Our implementation automatically solves the first problem since data is evenly divided into tablets. The latter is harder to address since imbalance may occur due to the specific characteristics of an application. For instance, we chose a uniform distribution for our test data since the distribution has equal probability for any random region of data, thus minimizes the chance for load imbalance. It is critical for the programmer to be aware of these challenges in order to address them properly.

4.2.2 MapReduce Introduction

MapReduce has been shown to work well on the BW [15], which gives a compelling reason to develop a section on how to effectively utilize this paradigm. Database aggregation falls into the MapReduce dwarf since each stage is directly mapped to a phase in MapReduce, as described in Figure 3. In our case, the input data is the original tables and the output data is the result table.

Map: Each of the partitioned tablets in a table is mapped to a node. The node’s number and its corresponding tablet form a key/value pair. Each node then executes the VM generated and then produces its own set of result rows. These are the intermediate results to be passed to next phase.

Shuffle and Sort: Each node sorts its own result table based on programmer’s specification. For instance, these nodes arrange the numeric values in ascending/descending order.

Reduce: The master node is responsible for this phase, collecting other nodes’ intermediate tables based on their keys and then rearrange these tables to into one single final table to display.

5. CONCLUSION

The Virginian database was used as a platform for the
The Blue Waters Student Internship Program (BWISP) impacts me significantly on many aspects of my personal and professional developments.

The Petascale Institute (PI) allowed me the opportunity to learn how to use the Blue Waters supercomputer, using the various parallel programming frameworks it provides. This experience and exchanges with my fellow students and the instructors improved my programming skills greatly. Also, the exposure to other disciplines though these conversations opened my eyes to the vast application of Computer Science (CS). The areas I was introduced to include biomatics, physical simulations, and financial modelling.

After the PI, working on my own project helped me apply these learned skills in practice. I gained invaluable experiences working with large software project that utilizes many software tools and involves various areas of CS, such as Compilers, Computer Architecture, and Software Development. More importantly, the experience provides me with the materials to discuss in my graduate school applications. As a result, I was accepted to all programs I applied to, including well-known institutions in High Performance Computing such as University of Michigan and Georgia Tech.

All in all, the BWISP was an immense success and equipped me with the necessary skills to succeed in my career as a Computer Science researcher.

7. FUTURE WORK

Our implementation has been designed partly to demonstrate a very general framework for GPU data processing. Using this framework, a next step is to implement and test additional database features such as other types of join, including inner, outer, and natural Joins. Modern RDBMSs are extremely complex, and much more work in this area is required to fully replicate this functionality in a GPU-friendly manner. We have shown that our opcode framework will be adaptable to this additional functionality, with modifications to our VM as appropriate to facilitate inter-opcode communication [4, 3].

Regardless of the direction, the general area of GPU application development is ripe for future research.

8. ACKNOWLEDGMENTS

This work is supported by a grant from the Shodor Education Foundation through the Blue Waters Student Internship Program (BWISP) and by the National Center for Supercomputing Applications, who provides the hardware used in this project.

9. REFERENCES


