6-29-2016

Fast Computation on Processing Data Warehousing Queries on GPU Devices

Sam Cyrus
University of South Florida, samcyrus7@gmail.com

Fast Computation on Processing Data Warehousing Queries on GPU Devices

by

Sam Cyrus

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yicheng Tu, Ph.D.
Srinivas Katkoori, Ph.D.
Swaroop Ghosh, Ph.D.

Date of Approval
June 29, 2016

Keywords: Database performance, High performance computing, Big data, Parallel queries on GPU, Data warehouse queries

ACKNOWLEDGEMENTS

I would like to thank Dr. Yicheng Tu for his mentorship and advice with this project. I would also like to thank Hao Li, Ran Rui, Chengcheng Mou and Xinwei Niu for their contributions to this paper. Finally I would like to thank my committee members for their support and encouragement. This work was supported by the University of South Florida Computer Science department.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Current database management systems use Graphic Processing Units (GPUs) as dedicated accelerators to process each individual query, which results in underutilization of GPU. When a single query data warehousing workload was run on an open source GPU query engine, the utilization of main GPU resources was found to be less than 25%. The low utilization then leads to low system throughput. To resolve this problem, this paper suggests a way to transfer all of the desired data into the global memory of GPU and keep it until all queries are executed as one batch. The PCIe transfer time from CPU to GPU is minimized, which results in better performance in less time of overall query processing. The execution time was improved by up to 40% when running multiple queries, compared to dedicated processing.

1. INTRODUCTION

The graphic processing unit (GPU) is a highly specialized architecture that has been used traditionally for gaming. GPU is similar to CPU in that it consists of multiple processors, but the hardware architecture is quite different between the two. GPU has many parallel, simple, multithreaded cores. In contrast, CPU has a substantial cache memory on just a few cores. GPU generally executes one application at a time in a single-instruction-multiple-data (SIMD) processing style. GPU has a limited bandwidth to transfer data back and forth between the GPU and CPU. Little hardware support is available to write conflicts. GPU kernels also cannot allocate dynamic memory. GPUs have been mostly used as dedicated co-processors, which results in underutilization of GPU resources. However, the massive computability and parallel processing capability of GPU makes it an attractive option for running multiple-query executions. Therefore, the use of GPU has been explored in high-power computing and data warehousing that require executing multiple queries simultaneously.

Data warehousing requires processing a large volume of data and the amount of data continues to increase. Often multiple users are querying the system simultaneously. As data workloads continue to increase, traditional CPU architecture is being questioned. For example, most big data software still use Linux running on Intel X86 processors. Furthermore, a very large computer is needed to run SQL queries on terabytes of data. Improving the efficiency of data warehousing has become an important area of research.

Even though GPU has traditionally not been used to run multi-query executions, it does have the potential to provide massive parallel processing capabilities with much higher memory and computation performance than CPU. If this parallel processing function could be exploited in data warehousing, GPU may actually be an ideal solution to improve efficiency in query processing [1]. Data warehousing workloads could be run in a column-oriented database by utilizing parallel processing. As a result, only a fraction of the computing hardware would be needed. Utilizing even a small fraction of the large number of cores in one GPU would result in a dramatic improvement in performance. A much smaller computer would still be able to process a large amount of data. GPU has the potential to drastically improve the efficiency of data warehousing [2].

However, implementing a GPU-based query coprocessor presents several challenges. One issue is how to create a common set of relational query operators, especially since parallel programming is difficult to execute in itself and programming the GPU to perform query processing is an unusual task. Another issue is that the cost model for in-memory databases is CPU-based and may not be applied directly to GPU. A final issue is that GPU and CPU are two heterogeneous processors that are joined via a limited amount of bandwidth, so effective coordination is needed between the two.

With these challenges in mind, this paper presents a novel approach in which all queries are run as a batch in GPU in order to improve the efficiency of query processing. The data is transferred to GPU in chunks, stored, and reused as necessary. The overall running time is improved, because data does not need to transfer to GPU every time a new query is run. This approach is particularly helpful for processing big data, such as with data warehousing queries.

This thesis will cover related work involving GPU efficiency in Chapter 2. The background is covered in Chapter 3. The existing design explains several optimization techniques that have been found to improve GPU efficiency and provides the framework and inspiration for this thesis in Chapter 4. The new design and implementation is presented in Chapter 5. The results and analysis of this approach are presented in Chapter 6. Finally, the conclusions and future work are summarized in Chapter 7.

2. RELATED WORK

Several papers have already examined how GPU can be used as a co-processor to improve the efficiency of database queries [3,4,5]. One aspect has been to exploit key operations for data warehousing such as aggregations, conjunctive selections, and semi-linear queries. Algorithms were written for boolean combinations, predicates, and aggregation queries and implemented in GPU. The results showed that GPU had a substantial performance gain compared to CPU [6].

Another algorithm was developed that allowed databases to be sorted using GPU. Memory and compute intensive queries were delegated to the GPU while I/O and resource management was delegated to the CPU. The task was written as two phases. The first phase involved reading the disk, building keys, sorting using GPU, generating runs, and writing the disk. The second phase involved reading, merging, and writing. This allowed for disk transfers to be pipelined and for excellent I/O performance. The result was a significant improvement in the performance of database processing tasks [7].

A relational join algorithm was also designed to improve GPU performance. A set of data-parallel primitives was used to implement sort-merge, nested-loop, and hash joins. This resulted in efficient communication between processors, the ability to write to random memory locations, and a model for programming that could be applied to general-purpose computing. Memory stalls were reduced and a 2-7x increased performance was seen compared to CPU [8].

Foreign-key joins have been examined in an effort to decrease the overall query time. The foreign key index is scanned in sequence while the primary key table is often accessed randomly. Since GPU connects slowly to the large system memory and quickly to the small internal device memory, it has an ideal architecture for handling foreign-key joins. The GPU VRAM was used to execute random table queries while the foreign key index was streamed through the PCIe bus sequentially. The result was an acceleration of foreign key joins [9].

Another model created a transaction processing engine to perform high-throughput transactions in a short period of time. This involved an engine that grouped multiple transactions into one bulk and then executed that bulk as a single task on GPU. This allowed for many small tasks to be handled concurrently. They found that the throughput was improved 4-10x compared to CPU. They also examined the cost efficiency between GPU and CPU. They found that even though a GPU device is more expensive to purchase, a GPU device using their engine had anywhere from 52-214% higher throughput per dollar compared to CPU [10].

He *et al.* designed an in-memory relational system of query co-processing for GPU. A set of data-parallel primitives (split and sort) was created to implement this relational query processing algorithm. The algorithm exploited the high memory bandwidth and parallelism of the GPU. The algorithm also took into account the GPU-CPU data transfer cost and the resources needed for the computation so that each operator in the query could utilize whichever processor was most suitable (GPU, CPU, or both). They found that this GPU-based algorithm was 2-27x faster than traditional CPU algorithms [11].

Improving the sparse matrix-vector multiplication (SpMV) on GPUs has also been examined. SpMV involves solving partial differential equations and linear systems. These matrices can be quite large. One model proposed a partitioning framework which partitioned the

largest sparse matrix and transferred each partition into a designated storage format based on different storage characteristics. They found that different storage formats of the matrix could greatly affect the SpMV performance. An auto-tuning tool automatically adjusted CUDA-specific parameters to optimize the performance. The result was a significant gain in performance when compared to existing SpMV CUDA kernels [12].

Finally, Yuan *et al.* examined the current challenges with GPU that have limited this technology from being fully implemented in database processing. They analyzed several software optimization techniques to improve the speed for data transfer and GPU kernel execution. This paper provided the inspiration and framework for the thesis, so it will be discussed in further detail in the next section [13].

3. BACKGROUND

Yuan *et al* examined the SSBM query behavior and tested the effects of various software optimization techniques on various GPU cards. Their results are explained in detail in this section in order to provide a context and framework upon which this thesis is built upon [13].

3.1 Schema

The performance of column-store and row-store were examined for any differences by using the Star Schema Benchmark (SSBM) [14,15]. The SSBM is a data warehousing benchmark that is derived from TPC-H1. However, it has fewer queries than TPC-H, has fewer requirements for what forms of tuning are allowed, is easier to implement than TPC-H, uses a pure star-schema, and the C-Store doesn't have to be modified in order to run.

The benchmark is a fact table that is known also as the LINEORDER table. This is formed from the LINE RECORD and ORDERS tables of TPC-H. It consists of 17 columns with information about individual orders and also a primary composite key using LINENUMBER and ORDERKEY characters. Foreign keys to CUSTOMER, PART, SUPPLIER, and DATE tables (order date and commit date) and attributes of each order (priority, quantity, price, and discount) are contained in the LINEORDER table. The table schema is shown in figure 1. Similar to TPC-H, there is a base scale factor of 1. This can be used to scale the size (populate desired data) of the benchmark. The size of each table is defined according to this populated data factor. A scale factor of S=1 was used, which resulted in a LINEORDER table with 6 x 10^6 tuples and 1

gigabyte of raw data. Also the scale factor of S=20 was used to represent 20 gigabits of data and the LINEORDER table contained 20 x 10^20 tuples.

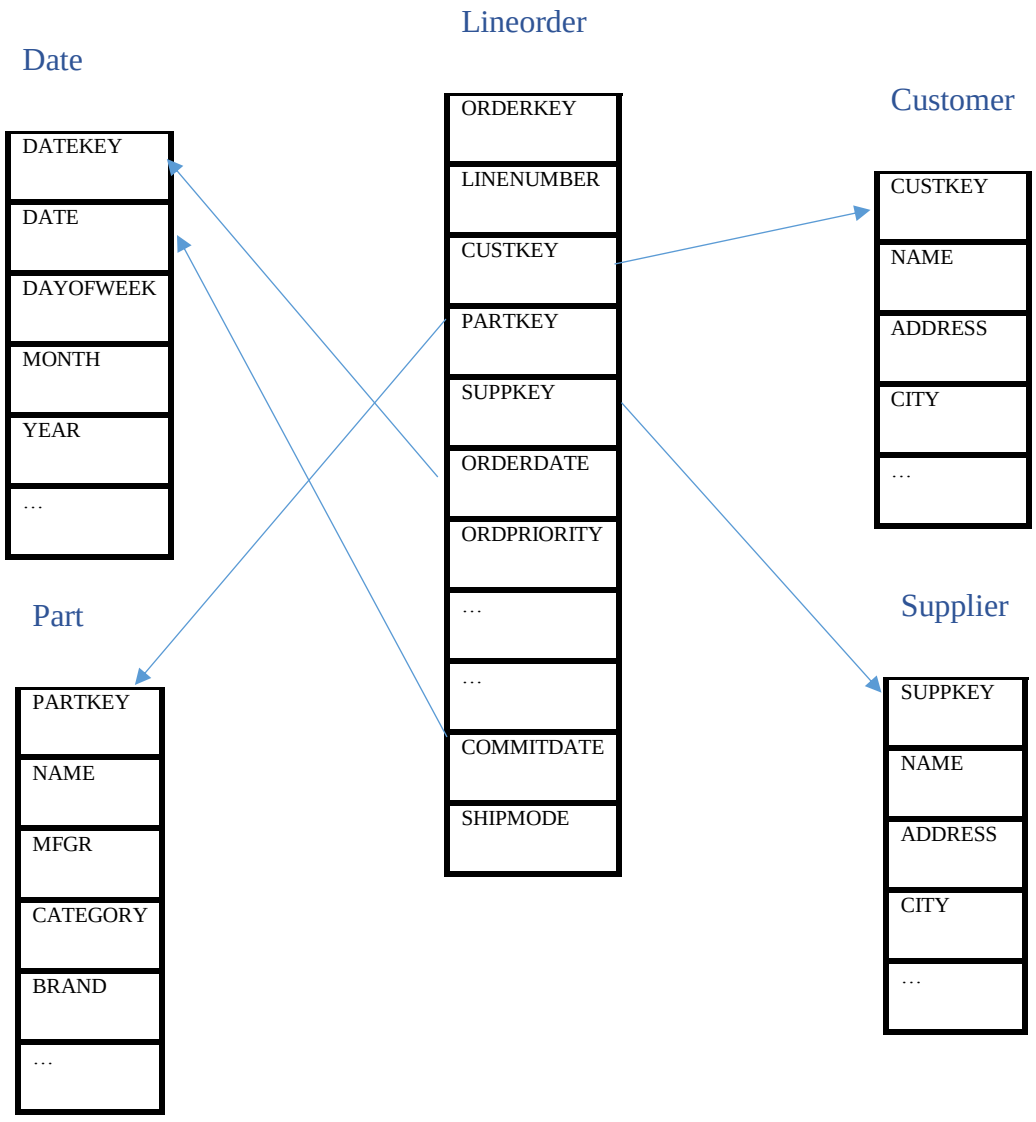

Figure 1. Star Schema benchmark

After the column-store tables are organized, they are connected to the queries and converted to a program that can be compiled by using Ysmart [16]. Ysmart is a standalone GPU execution engine for warehouse style queries. The front end consists of a *query parser* and *optimizer*. It translates an SQL query into an optimized query plan tree. This is then used by the *query generator* to populate a driver program, which controls the query generated flow. It is combined with the GPU operator library to make an executable query binary. The query binary then reads table data from a column format file on the disk storage and causes the GPU operators to offload data to GPUs for more efficient processing. Then the results are placed into rows and returned back to the user.

To further explain GPU query execution, consider this query that calculates the revenue from orders from 1990-1995 with discounts no less than or equal to 5% per year:

```
SELECT dyear, SUM(.lo_revenue) as revenue
FROM ddate, lineorder
WHERE .lo_orderdate = .d_datekey
AND dyear >=1990 AND dyear <= 1995
AND .lo_discount <= 5
GROUP BY dyear
```

An execution plan created by YSmart is shown in Figure 4. The *lineorder* fact table undergoes a table scan. Next the selection predicate *lo discount* <= 1 is measured to create a selection vector. The scan operator then locates the *lo orderdate* and *lo revenue* and generates a table consisting of the two filtered columns to the driver for the CUDA. Also with a selection filter *d year = 1990*, the system driver scans the *ddate* dimension table. Next a table is formed with the filtered *d datekey* and *d year* columns. These two intermediate tables are joined after the

scans. A hash table is created on *d datekey'* and explored with *lo orderdate'* to generate a filtering vector. This is then used to filter the *d month'* and *lo revenue'* columns of the intermediate tables. The join output is materialized to get the final query result.

The GPU operator collection makes the GPU implementations of usual database operations available (such as aggregations, scans, sorting and joins). These operations are modified to achieve both kernel and procedure levels in YSmart. Shared memory and memory access coalescing are fully exercised to maximize standalone kernel performance. Direct host memory access based on IOMMU (through CUDA [17] or OpenCL [18]) and data compression techniques are supported to reduce the data transfer overhead. Table rows are pushed from one

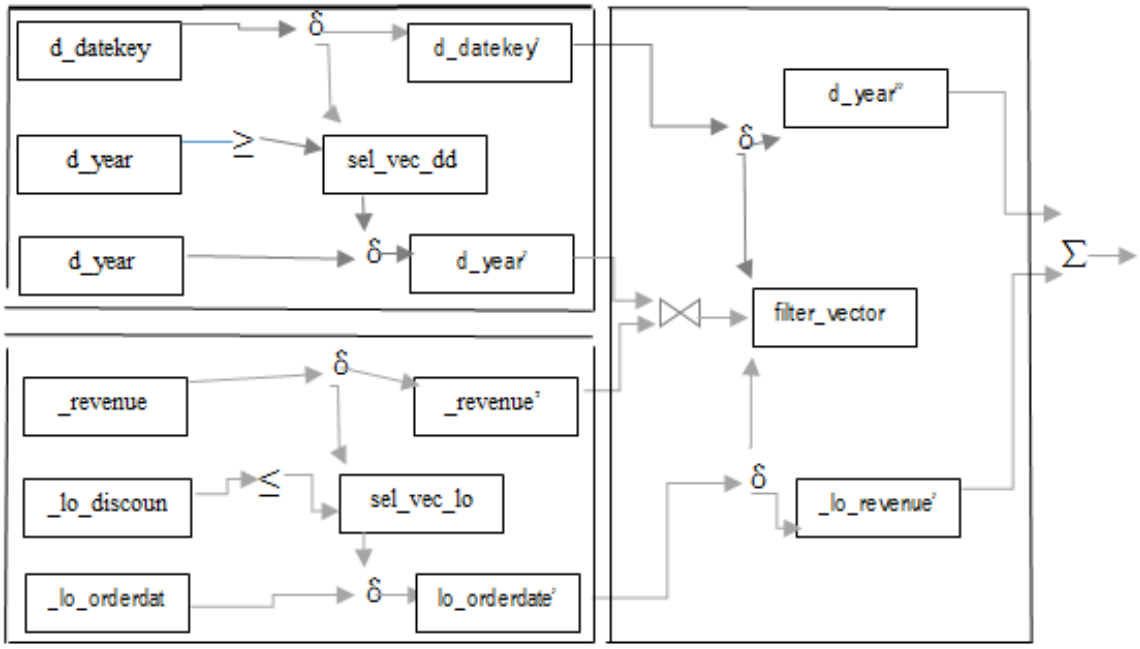

Figure 2. Sample of query running in YSmart

operator to another in one move to ensure kernel execution efficiency. If the data set cannot fit directly into device memory, it is then organized into smaller blocks and processed individually.

The core design behavior of other analytical GPU query engines are similar to YSmart even though there are possible differences in implementation. The implementation with MonetDB [19] needs to comply with the internal interfaces of MonetDB. However, its column-based data stores, operator-a-time execution model, and the major GPU operator designs agree with YSmart for the most part.

3.2 SSBM Query Behavior

In figure 3, all SSBM queries in NVIDIA GTX 680 were run within pinned host memory. Most of the execution time was spent on PCIe transfer. Therefore, PCIe transfer has a big impact on all of the queries. This is because the data needs to transfer to the GPU device in order to compute the data. In addition, the fact table also has a big impact on each query because it is larger than the other tables (part, date, customer, and supplier). The reason why the queries have similar PCIe transfer is because they process almost the same amount of data from the fact table (lineorder table).

In Query 1, the GPU kernel running time spends most of the time in the selection operator to generate the result (figure 3). Query 2 has higher join selectivity. Because more time is spent in join operator, the total kernel execution time is longer for each query. Query 3 uses a hash probing operation and a sequential scan from the lineorder table. The result is joined with four other tables (part, date, customer, and supplier). Query 4 uses more hash probing to create join results that come from the lineorder table. Queries 4.1-4.2 have higher join selectivity and 4.3 has lower join selectivity. Query 4.1 doesn't spend as much time on the dimension table because it has low selectivity on the other tables.

For simplicity, several queries in the same category were combined into one because they had similar selection and join. However, all of the data is still intact by using this approach.

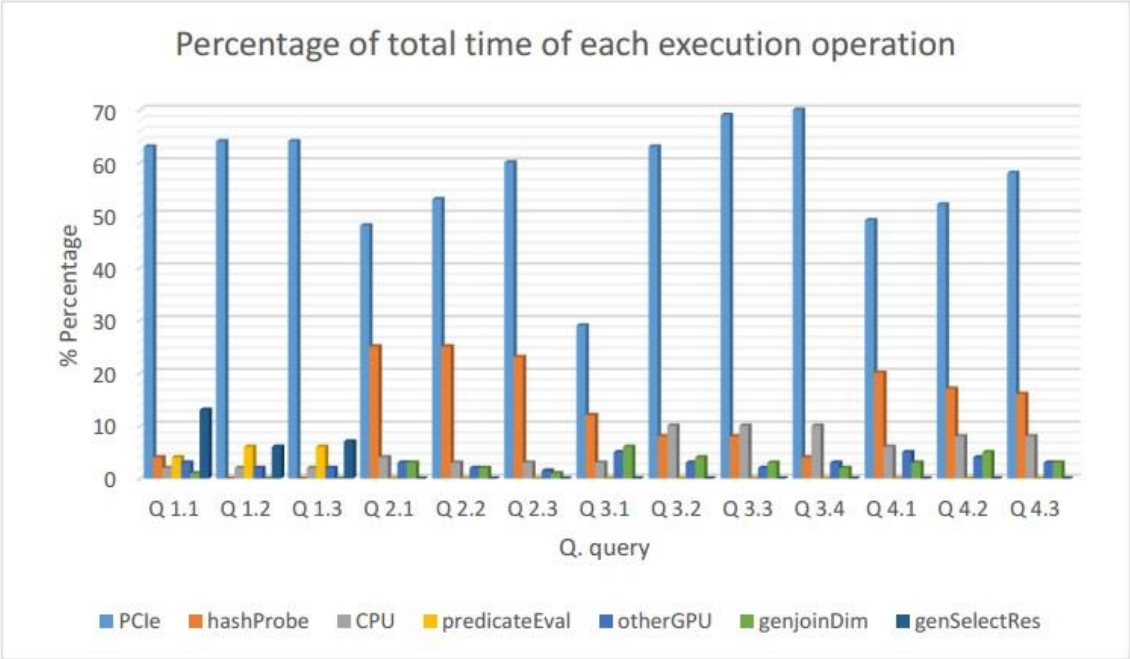

Figure 3. SSBM execution time breakdown (adapted from [13])

3.3 Data Compression

When the queries are run individually, different compression schema (dictionary encoding, bit encoding, and run length encoding) can help to reduce the data that is transferring through PCIe. Since the dimension tables are small, there is no need to compress the column store data. However, the compression technique can be used for the fact table data. Lineorder and the four other tables are stored in different copies on the disk, depending on which column it is. Each copy is stored on a multiple foreign key column.

A significant performance boost is seen when the data is compressed through the above mentioned techniques [20]. For example, the hash probing operation in join selection can be compressed. A hash table can work directly on the compressed data to scan the foreign keys. For queries that have low selectivity and join operation, the PCIe transfer time cannot be reduced even by accessing data in a coalesced manner. The GPU kernel spends most of the time

generating the result, so the compression technique does not result in much benefit. Since all queries are run as one batch, the data compression technique does not reduce the PCIe transfer time. In addition, the queries in the batch are dominated by the selection data.

3.4 Transfer Overlapping Effect

Queries that generate more results and have low selectivities and more columns from the lineorder table are more likely to benefit from transfer overlapping. The transfer overlapping technique can be used between PCIe bandwidth and kernel execution. However, the PCIe transfer bandwidth between pageable to pinned host memory needs to be considered as well.

Queries with more sequential access from the lineorder table spend more time in the kernel to compute the desirable result. These queries will benefit from the transfer overlapping technique. On the other hand, dimension tables have higher selectivity. This results in more kernel time and more random access patterns for the queries. Therefore, the overlapping technique does not benefit this type of query.

Coalesced access was used in the whole engine. However, not much benefit was seen in queries that used the random access pattern from the dimension tables and the sequential access in the lineorder table because it results in low utilization of the memory bandwidth.

3.5 Invisible Join Effect

Using invisible join reduces the random access data reading from dimension tables. It can help to improve these types of queries. Invisible join has previously been shown [21] to help the performance of the star schema queries in CPU. It helps with foreign key join in the fact table to predict the result. It has a materialization benefit because the selection and aggregation operators tend to filter some unneeded rows. Invisible join does not change the amount of transferred data

from the fact table. Therefore it has no effect on PCIe transfer time. The invisible join prediction can reduce the random access data by predicting and rewriting the foreign key join. It can also transform the hash probing into selection operation foreign key columns in the fact table if the foreign key continues within the same value range. Therefore, invisible join can be very inefficient for warehousing data queries. In invisible join, all foreign keys in the fact table and in dimension tables have to be reorganized and all the primary keys need to lie in a continuous range as well. This makes the run time longer and therefore reduces efficiency in star schema.

4. EXISTING DESIGN

4.1 Preprocessing Queries

Preprocessing the data before being sent to the GPU is important for improving performance. Query rules are assessed via several relational algebra operations, such as projections, selections, and joins. When the rule is assessed, the specific operation that is to be performed in the host (facts, variables, constants, and other tables) is sent to the GPU so that all threads can use it. This is in contrast to the operations being performed in the GPU by each individual kernel thread.

4.2 Query Operator

This implementation consists of four operators that are allowed by star schema queries. Each operator represents an algorithm that is well-researched and tested in order to get the intended result.

4.2.1 Selection

This is the primary step to scan any column. The result is a 0-1 vector, which can then be used to filter the desired column.

4.2.2 Join

A non-partition hash algorithm is used which is designed for star schema queries. It can run on a multi-core platform [22, 23]. Cuckoo hash [24] and chained hash were both used for the experiment and platform. If the size of the hash table is big enough, then a hash conflict will not occur [25]. Also star schema queries have low join selectivites. Therefore, chained hash works better than Cuckoo hash in this project.

4.2.3 Sort

The sort operator sorts the keys first. Then the results are projected. The sort operator occurs at the end of the query after the other operators have generated their results. This results in a smaller number of rows that are sorted.

4.2.4 Aggregation

The operator uses two steps to generate the result. The first step involves scanning the group-by keys, which then computes the hash value for each individual key. The second step involves scanning the hash value and combined columns in sequence.

4.3 Workloads

The Star Schema Benchmark [17] was used against 10 queries after combining several queries into one. Each query uses four tables (date, supplier, customer, and part). The tables are set in a star schema design. Figure 1 shows the schema of the 4 tables.

5. NEW DESIGN AND IMPLEMENTATION

5.1 Main Component of GPU Engine

The architecture uses global memory. The lineorder table is loaded in chunks to the global memory until all queries are run. Global and shared memory were used for selection, join, and aggregation. If the data is small, then it is sorted in shared memory instead of global memory.

The data sent to the GPU is arranged into arrays that are stored in the global memory. The results of these rule evaluations are stored in the global memory as well. The GPU engine is organized into several device kernels. When assessing rules of each query, selection and self-join kernels are used first to remove unrelated rows as quick as it can, continued by projection and join kernels. At the end of each rule assessment, any duplicated kernels are eliminated.

5.2 Memory Usage

The memory structure involves using shared and global memory and the use of pointers. Data is stored in shared memory for as long as space allows. When the shared memory becomes full, a pointer directs the data out to the global memory. The shared memory contains thread blocks that in turn contain thread groups. Each group has unique pointers. Only 1 copy of each pointer then exists in the device memory. Data is transferred from these thread groups in the shared memory to the global memory in parallel, which results in increased efficiency. Then the data is transferred from global memory to the result array again in parallel with the use of

pointers. Finally the result is transferred from the result array back to the host memory. Figure 4 illustrates this process.

Coalesced access improves the utilization of memory bandwidth. Coalesced access involves multiple threads in a thread group that are accessed sequentially by the global memory due to the SIMD nature of GPU. This allows multiple threads to be combined into one request. In contrast, non-coalesced access involves each thread requiring its own request. The result is very poor utilization of memory bandwidth. If n defines a memory request, then coalesced access results in (n-1) times less memory requests compared to noncoalesced access. Figure 5 compares coalesced and noncoalesced accesses.

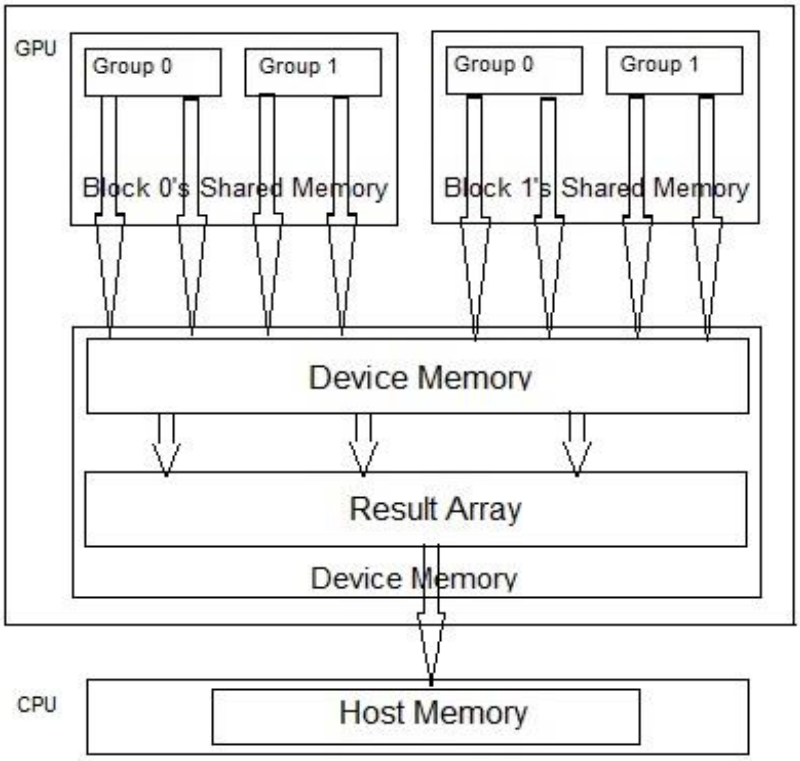

Figure 4. GPU memory

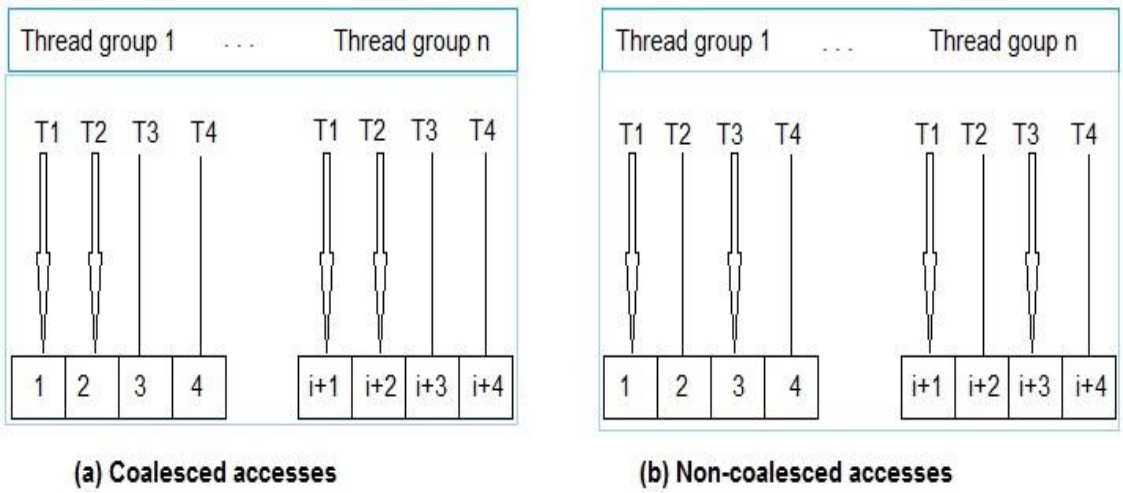

Figure 5. Coalesced and non-coalesced accesses in thread groups

Current GPU engines, such as YSmart, function as dedicated query co-processors. The query engine accepts 1 user query at a time and then executes a query plan, assuming only usage of the GPU device. A dedicated query processing plan simplifies the algorithm design and query optimization. This demonstrates how the resources of GPU can be utilized to run these queries more efficiently.

This memory scheme tries to minimize the number of transfers between CPU and GPU. Its purpose is to send the lineorder table in chunks to GPU memory and to have them stored there for as long as possible, so that they can be reused. In order to do this, the GPU memory available and memory used is tracked. Pointers are maintained with information about each chunk that resides in GPU memory. When a data chunk is requested to be loaded into GPU memory, the pointers are first checked to see if that data already exists. If it is found, then the entry uses the pointer and does not have to transfer the data. If the pointer is not found, then memory is allocated for the data. Next a pointer entry is created. The data's address in memory is returned

either way. If allocating memory for this data requires deallocating other chunks, then any unused chunks/pointers are deallocated first until enough memory is made available.

6. EXPERIMENTAL EVALUATION

6.1 Framework Environments

6.1.1 Software Platform

Ubuntu 14.04.3 LTS (kernel 3.13.0-62) was the operating system used. NVIDIA Linux version 352.41 was the GPU. CUDA SDK 7.5.17 was the driver. Intel Xeon memory size is close to 44MB despite the physical memory.

Table 1. Hardware Specifications

Processors	# of Cores	Memory size	Bandwidth(GB/s)
GeForce GTX 770	1536	2 GB	224.3
GTX TITAN X	3072	12 GB	336.5
Tesla C2075	448	6 GB	144
Tesla K20C	2496	5 GB	208
Intel Xeon	8	~44 MB	51.2

6.1.2 Hardware Platform

The experiment was executed with different GPUs: NVIDIA GeForce GTX 770, GeForce GTX TITAN X, Tesla C2075, and Tesla K20C with support of PCIe 3.0. The host motherboard was Intel Xeon(R) CPU E5-2640 with 64 GB memory. The important factors for these processors are broken down in table 1.

6.2 Measurement

6.2.1 Tools

We eliminate the disk loading time because the assumption is that the data is already in the host memory. NVIDIA's CUDA 7.5 provides the command line profiling tool *nvproof* to profile the query behavior on NVIDIA GPUs. It allows the system to measure the time for each kernel function and what percentage of the kernel is being used.

6.2.2 Measurement of Bandwidth

To help measure the bandwidth, the difference between pageable host memory and pinned host memory needs to be understood. Two GPU kernels are used to read and write 256MB integers to and from the device memory in a coalesced fashion to calculate the bandwidth. The result of this test is in table 2.

PCIe transfer bandwidth becomes higher if the host memory is in pinned memory, because data can be transferred directly by using the GPU DMA engine. Regarding pageable memory, data needs to transfer or copy to a pinned DMA buffer. Then the data can get transferred to the GPU DMA engine [26]. As the above table shows, GTX Titan X was found to be more powerful than the other GPU cards.

Table 2. Bandwidth Measurement

	GTX TITAN X	GeForce GTX 770	Tesla K20C	Tesla C2075
Read(GB/s)	301.50	201.76	128.10	113.59
Write(Gb/s)	165.10	155.44	154.43	128.34
HtoD pagable(GB/s)	10.20	9.10	6.30	6.30
HtoD pinned(GB/s)	14.28	13.03	12.36	6.60
DtoH pagable(GB/s)	8.22	7.09	6.40	6.15
DtoH pinned(GB/s)	14.75	13.80	12.90	6.10

6.3 GPU Kernel Threading

The thread block size range is from 128 to 4096. If the kernel is launched with a larger number of blocks, then it can process more data at once. For example, if 1,000 threads are used and the thread block size happens to be 256, then 1,000/256 = 4 thread blocks are processed.

6.4 Implemented Architecture

The new architecture was used to run the SSBM queries all as one batch using a 1 GB data size on NVIDIA Titan X. The old architecture was also used to run the SSBM queries individually and sequentially again with a 1 GB data size. The execution times are shown in figure 6. The most striking difference is that the old architecture has substantially more transfer time. The new architecture resulted in up to 40% increased efficiency in query processing.

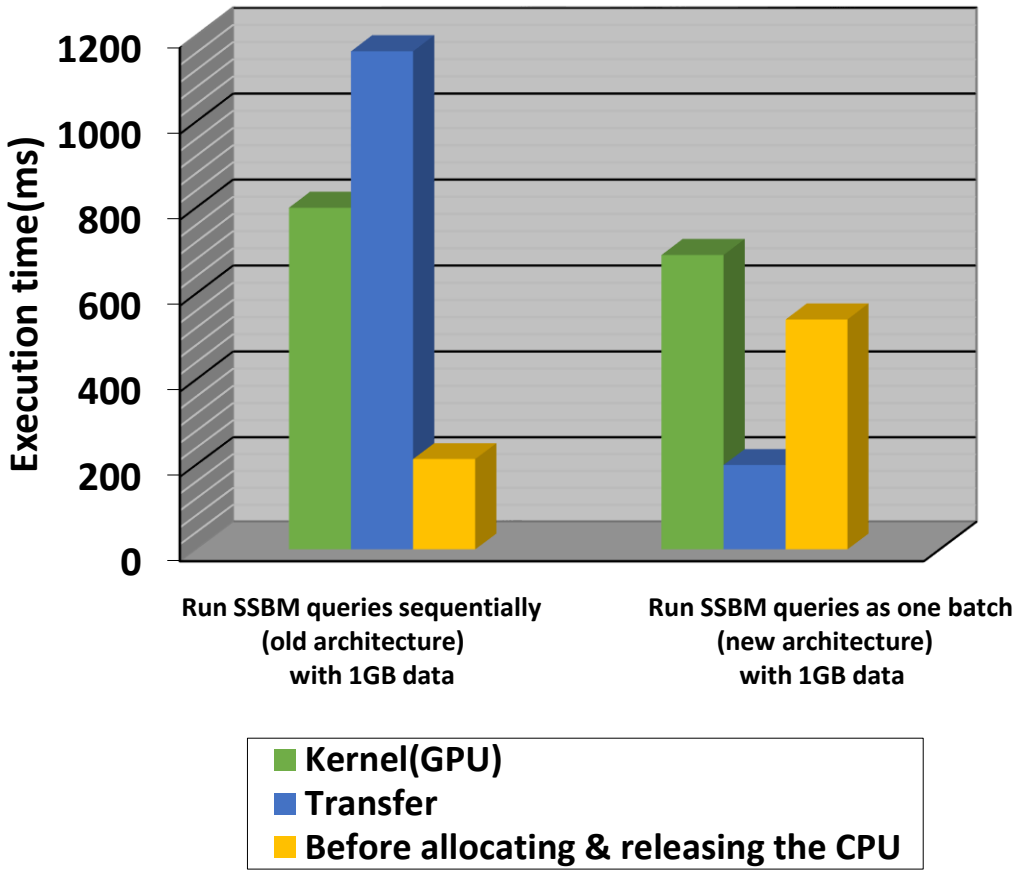

Figure 6. SSBM queries run sequentially or as one batch on Titan X

The old architecture involves transferring the fact table (lineorder table) to GPU every time because each query is run individually and sequentially. Because the lineorder table is the biggest table, this creates a bottleneck. The challenge lies in how to decrease the transfer time.

With this in mind, the new architecture was designed so that the lineorder table using column-store fashion is transferred to GPU one time instead of several times. Therefore, the lineorder table is the largest table compared to the other tables. All queries are run as one batch in order to bring the transfer time down and to take advantage of parallel computing on the GPU kernel. The key is to get better use of the kernel execution instead of spending more time transferring data from the host to device and vice versa. This result is calculated based on the assumption that the data size and running time of the query are known in the engine. In order to get better use from shared and global memory in GPU, a good pattern access is needed. The performance of the data warehousing query is dominated by the device memory access.

The GPU kernel spends most of the time doing the computation in the new architecture. The data in the fact table does not have to be sent to the GPU each time a query is run. Instead the data is transferred once to the GPU, the result is computed, and then it is transferred back to the CPU. The PCIe transfer time is cut almost in half because the data is not being shipped to the GPU multiple times. The size of the lineorder table is much bigger than the dimension table. Therefore, it takes less time to transfer the dimension table. Once the data is opened and read, it can be used for any upcoming queries that use that specific data. The overhead for opening and reading a file from the hard drive is minimized also. This new approach resulted in up to 40% increased efficiency in query processing.

6.5 Comparing Two Different Data Sizes

Two different data sizes (20 GB and 10 GB) were tested with the new architecture. As shown in figure 7, the 20 GB file size had double the total execution time, as well as double the transfer time and CPU time. This is expected since the data size is twice as large.

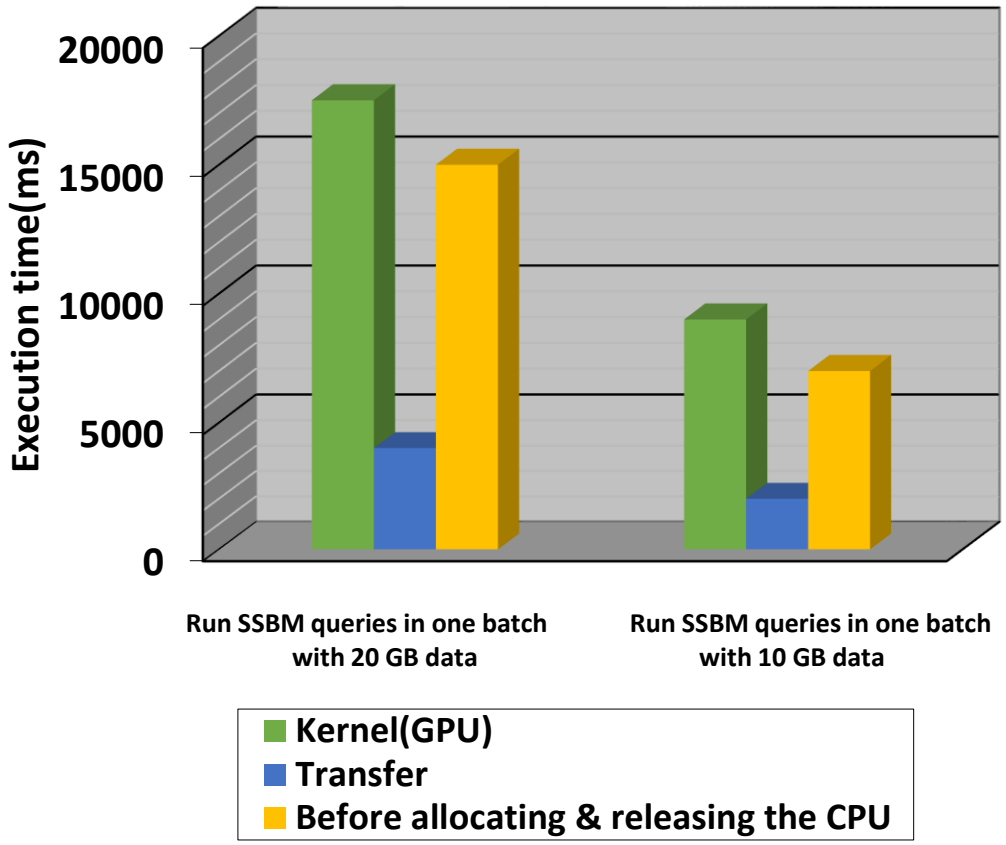

Figure 7. Executing SSBM queries as one batch with different data sizes on Titan X

6.6 Data Access Patterns for Queries That Can Facilitate Upcoming Queries

Primary data access patterns in hash joins have multiple functions. They scan the input tables for hash table creation and probe. They compare and swap during insertion of data into the hash table. They also perform random access read during hash table probing.

The GPU join consisted of a traditional hash join using unified virtual addressing that accessed the host memory. The efficiency of the join was tested by examining the throughoutput relative to table size. To identify bottlenecks separate from the framework's architecture, throughput was compared to the related hardware abilities. The throughput of the overall, hasthtable creation, and probe steps of the join were plotted against the table size, as shown in figure 8. The overall performance was measured to be up to 4.5 GB/s with minimal impact on table sizes. The hash table creation was only able to reach at most 3.8 GB/s.

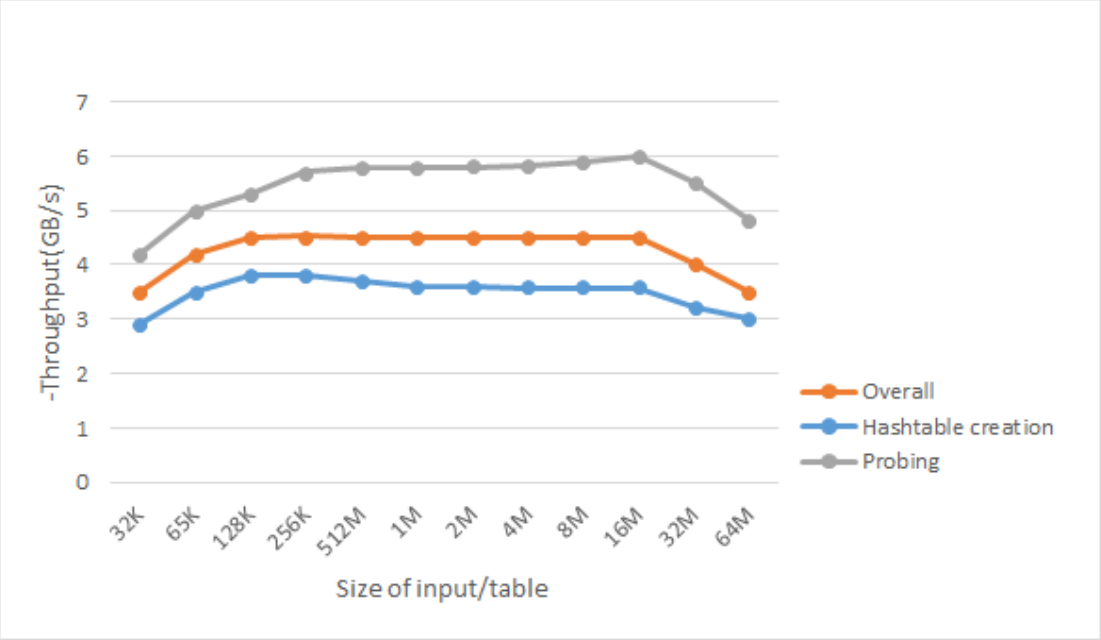

Figure 8. Hash join and throughput in GPU with UVA

The size of the input tables impacts the throughput of the overall, hashtable creation, and probing steps of the join. The overall throughput is the curves mean of the two steps of the hash join (hash table creation and probe). Hash table probing reached 5.9 GB/s, which is the maximum rate that the probe table can be accessed across the PCIe. However, the hashtable creation of 3.8 GB/s in this experiment was substantially slower. The main restriction for concurrent hashtable creation is the locking that is necessary to prevent concurrent threads from overwriting current hashtable data. The current method utilizes compare-and-swap to deal with parallel hashtable access.

The previous hypothesis had a 2.9% match rate, which meant that the time required to relay the data back to the host memory was insignificant compared to the time needed to read the probe. Other research related to CPU memory joins proposed that the time spent materializing join results is insignificant and just results in a constant overhead of a small number of compute cycles per result. However, given that the best join in GPU needs only three cycles per row, even a small number of extra cycles can make a noticeable difference in overhead when more probes generate the results.

6.7 Comparison of GPU Hardware on Query Performance

SSBM queries were run on two different GPUs to test the impact of GPU hardware on query performance. Figure 9 shows that the running time was not dramatically different between the two GPUs.

As mentioned previously in Table 1, Titan X has significantly more cores, memory, and bandwidth compared to Tesla C2075. However, because the improvement in running time was not dramatically different between the two GPUs, the computational power clearly has little impact on warehouse query performance. Instead the GPU device memory access is the driving

force in improved performance. It seems that advancement of GPU hardware will not have a substantial effect on warehouse query performance in the future.

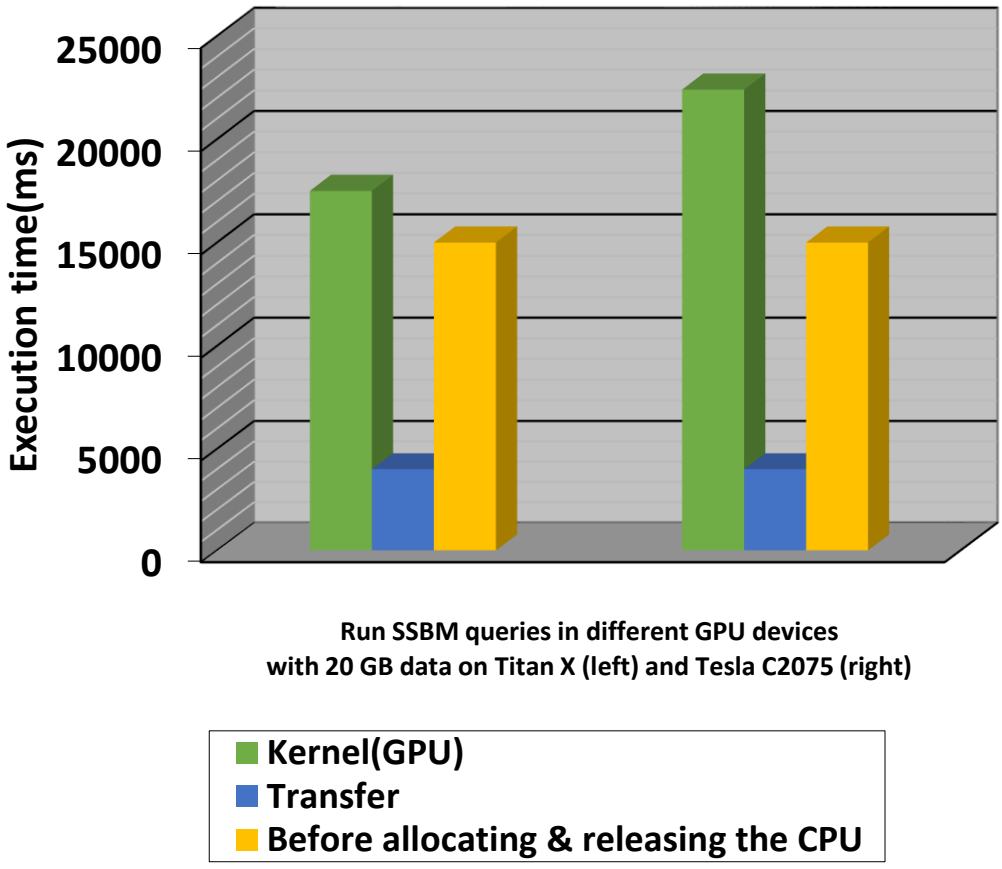

Figure 9. Kernel execution time on different GPU devices

6.8 Query Running Time Breakdown

In order to better explain the above results, a cost model is introduced. This cost model provides a breakdown of the total time required to run the queries as a batch. The total estimated batch processing time is:

$$T_{batch} = \sum_{i=1}^{n} \sum_{j=1}^{s-1} C_{g(1)}$$

Here n is the number of queries in one batch that needs to get processed, S is the number of query relations, and $C_{g(1)}$ is the average cost of producing the query result in a batch. We can normalize the total cost of queries in one batch and separate it evenly by simplifying the formula to:

$$T_{batch} = \text{n (s - 1) } C_1$$

$$= \text{s (s - 1) } C_{\frac{n}{s}}$$

The cost of $C_{n/s}$ ($\frac{n}{s} > 1$) can be different depending on how each query is implemented and what its selectivity looks like. If the cost of $C_{n/s}$ is a linear function on n, we can expect to have:

$$\frac{nC_1}{sC_{n/s}} > 1$$

If the cost of $C_{n/s}$ does not follow a linear function, then an optimized number m (m > 1) can be used:

$$\frac{kC_1}{sC_k} > 1$$

The total cost for one batch n is $\left(\frac{n}{s} \geq m\right)$. Also the value of **m** can be incorporated into the formula. The total cost is then:

$$T_{batch} = s(s-1)\left(\left[\frac{n}{ms}\right] C_m + C_{n-\left[\frac{n}{ms}\right]m}\right)$$

The queries preprocess the data before transferring to the GPU, which results in a better performance. The queries get structured and assessed through a string of relational algebra operations (selection, join, aggregation, and sort). All of the extract data from those operators, specific constants, and other facts are transferred out to the host device and finally to GPU to get in place to launch the threads for the computation. If GPU memory needs to deallocate memory for the new chunks to generate the result, then the result is transferred back to CPU memory before it can release the resources that are currently assigned to any thread, warp, or block.

7. CONCLUSION AND FUTURE WORK

This paper presents a new approach for processing warehouse queries in GPUs that results in a more efficient performance. The results show that GPUs with the use of global memory can dramatically improve whole throughput when running the data warehousing benchmarks for specific queries. The approach involves running queries in a batch using shared and global memory. The lineorder table is opened and read only once instead of shipping the lineorder table every time a query is run. Therefore, the running time is much faster. Up to 40% increased efficiency was seen in query processing. This approach improves the performance of GPU transfer in the CPU-GPU mix query engine and supports the idea that this mixed query engine may be more appropriate for high-performance query executions than traditional CPU architecture.

This approach was also tested on different GPU hardware. The GPU device with more cores, memory, and bandwidth actually did not have much improvement in running time. This is because the device memory access is really the driving force in the improved performance. Therefore, advancing GPU hardware will likely not have a major impact on warehouse query performance.

Future work will involve applying the scheduling techniques used in CPUs to GPUs in order to use the GPU resources more efficiently. The challenge will be to implement this without overwhelming the GPU transfer bandwidth, but still leading to a better performance.

In conclusion, this paper introduces a novel approach for improving GPU efficiency in a CPU-GPU mix query engine when processing large databases. The data is transferred to GPU in chunks, stored, and reused as necessary with an overall improvement in running time. This approach results in increased efficiency in data warehousing queries.

REFERENCES

[1] Ni T. DirectCompute: Bring GPU computing to the mainstream. In GTC, 2009.

[2] Yalamanchili S. Scaling data warehousing applications using GPUs. In FastPath, 2013.

[3] Fang W, He B, Luo Q. Database compression on graphics processors. In VLDB, 2010.

[4] Govindaraju NK, Lloyd B, Wang W, Lin MC, Manocha D. Fast computation of database operations using graphics processors. In SIGMOD Conference, 2004.

[5] Sitaridi E, Ross K. Ameliorating memory contention of olap operators on gpu processors. In DaMoN, pages 39-47, 2012.

[6] Bandi N, Sun C, El Abbadi A, Agrawal D. Hardware acceleration in commercial databases: A case study of spatial operations. In VLDB, 2004.

[7] Govindaraju N, Gray J, Kumar R, Manocha D. GPUTeraSort: High performance graphics co-processor sorting for large database management. In SIGMOD Conference, 2006.

[8] He B, Yang K, Fang R, Lu M, Govindaraju N, Luo Q, Sander P. Relational joins on graphics processors. In SIGMOD Conference, 2008.

[9] Pirk H, Manegold S, Kersten M. Accelerating foreign-key joins using asymmetric memory channels. In ADMS, 2011.

[10] He B, Yu JW. High-throughput transaction executions on graphics processors. In PVLDB, 2011.

[11] He B, Lu M, Yang K, Fang R, Govindaraju N, Luo Q, Sander P. Relational query coprocessing on graphics processors. ACM Transactions on Database Systems, Vol. 34, No. 4, Article 21, Dec 2009.

[12] Guo P, Huang H, Chen Q, Wang L, Lee EJ, Chen P. A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on GPUs. In Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery. ACM, NY, 2011.

[13] Yuan Y, Lee R, Zhang X. The yin and yang of processing data warehousing queries on GPU devices. PVLDB 6(10): 817-823.

[14] O'Neil PE, Chen X, O'Neil EJ. Adjoined dimension column index (ADC index) to improve start schema query performance. In ICDE, 2008.

[15] O'Neil PE, O'Neil EJ, Chen X, Revilak S. Star schema benchmark (SSB). http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[16] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet another SQL-to-MapReduce translator. In *ICDCS*, 2011.

[17] NVIDIA. CUDA C programming guide, 2013.

[18] Khronos OpenCL Working Group. The OpenCL Specification, version 2.0, 2013.

[19] monetdb.org

[20] Abadi DJ, Madden S, Ferreira M. Integrating compression and execution in column-oriented database systems. In SIGMOD Conference, 2006.

[21] Abadi DJ, Madden S, Hachem N. Column-stores vs. row-stores: how different are they really? In SIGMOD Conference, pages 967–980, 2008.

[22] Balkesen C, Teubner J, Alonso G, Ozu T. Main-memory hash joins on multi-core cpus: tuning to the underlying hardware. In ICDE, 2013.

[23] Kaldewey T, Lohman G, Mueller R, Volk P. Gpu join processing revisited. In DaMoN, 2012.

[24] Alcantara DA, Sharf A, Abbasinejad F, Sengupta S, Mitzenmacher M, Owens JD, Amenta N. Real-time parallel hashing on the gpu. ACM Trans. Graph, 28(5), 2009.

[25] Motwanj R, Raghavan P. Randomized Algorithms. Cambridge University Press, 1995.

[26] Amd accelerated parallel processing opencl programming guide (v2.8). http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf