The Yin and Yang of Processing Data Warehousing Queries on GPU Devices

Yuan Yuan Rubao Lee Xiaodong Zhang

Department of Computer Science and Engineering
The Ohio State University

{yuanyu, liru, zhang}@cse.ohio-state.edu

ABSTRACT

Database community has made significant research efforts to optimize query processing on GPUs in the past few years. However, we can hardly find that GPUs have been truly adopted in major warehousing production systems. Preparing to merge GPUs to the warehousing systems, we have identified and addressed several critical issues in a three-dimensional study of warehousing queries on GPUs by varying query characteristics, software techniques, and GPU hardware configurations. We also propose an analytical model to understand and predict the query performance on GPUs. Based on our study, we present our performance insights for warehousing query execution on GPUs. The objective of our work is to provide a comprehensive guidance for GPU architects, software system designers, and database practitioners to narrow the speed gap between the GPU kernel execution (the fast mode) and data transfer to prepare GPU execution (the slow mode) for high performance in processing data warehousing queries. The GPU query engine developed in this work is open source to the public.

1. INTRODUCTION

In the past decade, special-purpose graphic computing units (GPUs) have evolved into general-purpose computing devices, with the advent of efficient parallel programming models, such as CUDA [2] and OpenCL [4]. Because of GPU's high computational power, how to accelerate various workloads on GPUs has been a major research topic in both the high performance computing area and the database area [12, 18, 20, 28, 17, 29, 16, 30, 21, 19, 23, 10, 33, 25, 32]. In high performance computing area, GPUs as accelerators have already been widely deployed to process performance-critical tasks. For example, according to the June 2013's Top 500 list, more than 50 supercomputers have been equipped with accelerators/coprocessors (mostly NVIDIA GPUs), compared to less than 5 six years ago. However, in the database area, we can hardly find any major data warehousing system (e.g., Teradata, DB2, Oracle, SQL Server) or MapReduce-based data analytical system (e.g., Hive, Pig, Tenzing) that has truely adopted GPUs for productions, despite the existence of many research papers optimizing various database operations on GPUs which have already shown the significant performance benefits when utilizing GPUs.

To understand the reason behind this fact, this paper addresses the following issues with both technical and experimental bases:

- Where does time go when processing warehousing queries on GPUs? (Section 4.1)
- How do existing software optimization techniques affect query performance on GPUs? (Section 4.2 - 4.4)
- Under what conditions will GPU significantly outperform CPU for warehousing queries? (Section 5.1)
- How do different GPU hardwares and their supporting systems affect the query performance? (Section 5.2)
- How does the advancement of GPU hardware affect query performance? (Section 6)

1.1 The Framework of Our Study

The key to answering these questions is to fundamentally understand how the two basic factors of GPU query processing, which we call the *Yin and Yang* [1] , are affected by query characteristics, software optimization techniques, and hardware environments. The *Yin* represents PCIe data transfer, which transfers data between host memory and GPU device memory. The *Yang* represents kernel execution, which executes the query on the data stored in the GPU device memory. To characterize these two factors, we have conducted a comprehensive three-dimensional study of query processing on GPUs as shown in Figure 1. We target at the star schema queries because they are the typical workloads in practical warehousing systems [31]. Our study is based on the following three sets of research efforts.

Implementation of a GPU Query Engine: We have designed and implemented a GPU query engine using CUDA and OpenCL which can execute on both NVIDIA/AMD

[1]In ancient Chinese philosophy, Yin and Yang represent two opposite forces that are interconnected and interdependent in the natural world.

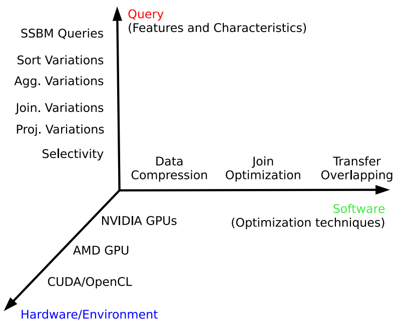


Figure 1: Research Overview: A 3-Dimension Study of Processing Warehousing Queries on GPUs.

GPUs and Intel CPUs. Based on the algorithms proposed in prior research work, we have made the best effort to implement various warehousing operators.

Experimental Evaluation and Performance Comparison: Based on our GPU query engine, 1) we studied warehousing query behaviors and analyzed effects of various software optimization techniques; 2) we compared the performance of warehousing queries on GPU with MonetDB [22], which is a representative high-performance analytical query engine; and 3) we investigated how different GPU hardwares and programming models can affect the performance of warehousing workloads.

Modeling and Predictions: We have proposed an analytical model to characterize and quantify the query execution time on GPUs. The model accuracy is verified by detailed experiments with different hardware parameters. Based on the model, we predict how possible advancement of future GPU hardwares will improve query performance.

1.2 Contributions of Performance Insights

Our comprehensive study quantitatively demonstrates that: 1) GPU only significantly outperforms CPU (4.5x - 6.5x speedups) for certain queries when data are prepared in the pinned host memory; 2) GPU has limited speedups (around 2x) for queries dominated by selection or by intensive random device memory accesses, or when data are not in the pinned host memory; 3) The major obstacle to OpenCL portability is vendors' subtle implementations of the specification which can cause both performance and functional problems for warehousing workloads; and 4) The peak performance increase in the evolving GPU generations has limited performance benefits for processing warehousing queries.

The rest of the paper is organized as follows. In Section 2 we present the implementation of our GPU query engine. Section 3 describes the experimental environment. We study the warehousing query behaviors and the effects of software techniques in Section 4 and conduct detailed performance comparisons in Section 5. In Section 6 we introduce our cost model and explore the impacts of GPU hardware advancement on query processing. We introduce the related work in Section 7 and conclude our work in Section 8.

2. GPU QUERY ENGINE

2.1 Engine Structure and Storage Format

Figure 2 shows the architecture of our query engine. It is comprised of an SQL parser, a query optimizer and an execution engine. The parser and optimizer share the same codes with YSmart [24]. The execution engine consists of a code generator and pre-implemented query operators using CUDA/OpenCL. The code generator can generate either CUDA drive programs or OpenCL drive programs, which will be compiled and linked with pre-implemented operators.

The engine adopts a push-based, block-oriented execution model which executes a given query plan tree in post-order sequence. It tries to keep data in GPU device memory as long as possible until all the operations on the data are finished.

We choose column-store for our engine since we target warehousing workloads. In our implementation, each table is stored as a collection of columns, where each column is stored in a separate file on the disk. Our engine uses the late materialization technique [6] and performs tuple reconstruction through a special GPU kernel when projecting the final results.

In our engine, the codes executed on CPU are responsible for allocating and releasing GPU device memory, transferring data between the host memory and the GPU device memory, and launching different GPU kernels.

2.2 Query Operators

Our engine implements four operators required by star schema queries, each of which is implemented with representative algorithms based on the state of the art of research.

Selection. Selection's first step is to sequentially scan all the columns in the predicates for predicate evaluation, with the result stored in a 0-1 vector. The second step is to use the vector to filter the projected columns.

Join. We implement the unpartitioned hash algorithm that has been proved to perform well for star schema queries on multi-core and many-core platforms [11, 13, 23]. We implement the hash table using both Cuckoo hash [9] and chained hash. For chained hash, hash conflicts can be avoided by making the size of hash table twice the cardinality of the input data with a perfect hash function theoretically [26]. In our study, the chained hash performs well than the Cuckoo hash. This is because star schema queries have low join selectivities, and Cuckoo hash needs more key comparisons than chained hash when there is no match for the key in the hash table.

Aggregation. We implement the hash based aggregation which involves two steps. The first step is to sequentially scan the group-by keys and calculate the hash value for each key. The second step is to sequentially scan the hash value

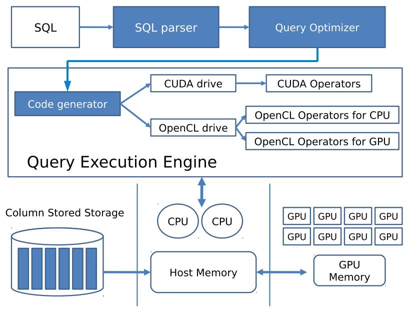


Figure 2: GPU Query Engine Architecture

Table 2: Hardware Specifications

Processors	# of Cores	GFLOPS	Bandwidth(GB/s)
NVIDIA 480	480	1345	177.4
NVIDIA 580	512	1581.1	192.4
NVIDIA 680	1596	3090.4	192.256
AMD 7970	2048	3788.8	264
Intel Core i7	4	112	25.6

and the aggregate columns to generate aggregation results.

Sort. Sort operator will sort the keys first. After the keys are sorted, the results can be projected based on the sorted keys which is a gather operation. Since sort is usually conducted after aggregation, the number of tuples to be sorted is usually small which can be done efficiently through bitonic sort.

2.3 Implementation Details

Use of GPU Memory. Our engine utilizes both device memory and local shared memory. For selection, only device memory is utilized. For join and aggregation, the hash table will be put in the local shared memory when its size is smaller than the local shared memory size. For sort, all the keys are sorted and merged in the local shared memory.

Data Layout. Each column is stored in a continuous memory in GPU device memory, which has the Array-Of-Structure (AOS) format. The Structure-Of-Array (SOA) format, which can provide coalesced access for scanning irregular data, doesn't provide performance benefits for our workloads because the accesses of irregular data (string data from dimension tables) are dominated by random accesses during join operations.

GPU Thread Configurations. The thread block size is configured to be at least 128 and the largest number of thread blocks is configured to be 4096. Each thread in the thread block will process a set of elements from the input data based on its global thread ID. For example, for a configuration with 256 threads per block and 2048 thread blocks, the thread with global ID 0 will process the data with the index of 0, 2048*256, 2*2048*256 until the end of the data.

GPU Thread Output. Our engine avoids the synchronizations among threads when they write to the same memory region at the same time. This is achieved by first letting each thread count the number of results it will generate, and then performing a prefix sum on the count result. In this way each thread knows its starting position in the region and can write to the region without synchronization.

3. EXPERIMENTAL METHODOLOGY

3.1 Workloads

We use the Star Schema Benchmark (SSBM) [27] which has already been widely used in various data warehousing research studies [8, 14]. It has one fact table *lineorder* and four dimension tables *date,supplier,customer,part*, which are organized in a star schema fashion, as is shown in Figure 3. There are a total of 13 queries in the benchmark, divided into 4 query flights. Table 1 summarizes the major characteristics of the SSBM queries. In our experiments, we run the benchmark with a scale factor of 10 which will generate the fact table with 60 million tuples.

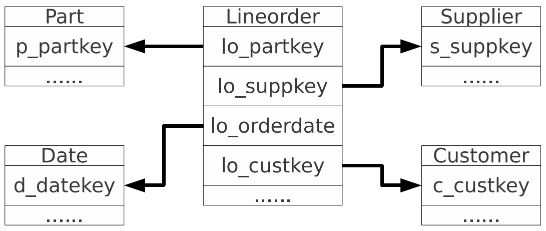


Figure 3: Schema of SSBM

Table 3: GPU Bandwidth Measurement

	480	580	680	7970
Read(GB/s)	114.59	129.95	127.65	202.76
Write(GB/s)	138.34	150.41	153.43	116.44
HtoD pageable(GB/s)	6.30	6.30	6.30	9.80
HtoD pinned(GB/s)	6.65	6.65	12.28	11.13
DtoH pageable(GB/s)	6.20	6.20	6.22	9.19
DtoH pinned(GB/s)	6.64	6.64	12.75	11.81

3.2 Experimental Environments

3.2.1 Hardware Platforms

We conduct our experiments on four GPUs: NVIDIA GTX 480, 580, 680 and AMD HD 7970. NVIDIA GTX 480 and 580 only support PCIe 2.0 while NVIDIA GTX 680 and AMD HD 7970 support PCIe 3.0. Each GPU will be connected to a PCIe 3.0 bus when conducting experiments on it. The host device is equipped with the Intel Core i7 3770k Quad-Core 3.5GHZ processor with 32 GB memory. Table 2 lists the major hardware parameters for these processors.

3.2.2 Software platforms

All the experiments are conducted under Red Hat Enterprise Linux 6.4 (kernel 2.6.32-358.2.1). The NVIDIA GPUs use NVIDIA Linux driver 310.44 with CUDA SDK 5.0.35. The AMD HD 7970 uses AMD Linux driver Catalyst 13.1 with AMD APP SDK 2.8. We use the query performance on MonetDB (version 11.15.3) to represent the state of the art of query performance on CPU. OpenCL query engine on Intel Core i7 is compiled with Intel 2013 XE beta SDK.

3.3 Measurement

3.3.1 Methodology and tools

When measuring the overall query execution time, we assume that data are already in the host memory and exclude the disk loading time.

We use NVIDIA's command line profiling tool *nvprof* in CUDA 5.0 toolkit to profile the query behavior on NVIDIA GPUs. For the OpenCL query engine, we use OpenCL events to collect the kernel execution time and PCIe transfer time. When measuring query performance on MonetDB, we put the data in a ramdisk to exclude the disk loading time.

3.3.2 Measurement of bandwidth

Before conducting detailed experiments on all the GPUs, we first measure two critical parameters: the PCIe transfer bandwidth and the GPU device memory bandwidth. To measure the former, we transfer 256MB data between host memory and GPU device memory. It is worth noting that

Table 1: **SSBM Summary**. The table lists the major operations and the Filter Factors (FF) for each SSBM query. L represents the fact table lineorder and D, S, C and P represent the four dimension tables: date, supplier, customer and part.

Query	Operation	FF on L	FF on D	FF on S	FF on C	FF on P	Overall Selectivity
q1.1	$\sigma(L) \bowtie \sigma(D)$	0.47*3/11	1/7	-	-	-	0.019
q1.2	$\sigma(L) \bowtie \sigma(D)$	0.2*3/11	1/84	-	-	-	0.00065
q1.3	$\sigma(L) \bowtie \sigma(D)$	0.1*3/11	1/364	-	-	-	0.000075
q2.1	$L \bowtie \sigma(P) \bowtie \sigma(S) \bowtie D$	-	-	1/5	-	1/25	0.008
q2.2	$L \bowtie \sigma(P) \bowtie \sigma(S) \bowtie D$	-	-	1/5	-	1/125	0.0016
q2.3	$L \bowtie \sigma(P) \bowtie \sigma(S) \bowtie D$	-	-	1/5	-	1/1000	0.0002
q3.1	$L \bowtie \sigma(C) \bowtie \sigma(S) \bowtie \sigma(D)$	-	6/7	1/5	1/5	-	0.034
q3.2	$L \bowtie \sigma(C) \bowtie \sigma(S) \bowtie \sigma(D)$	-	6/7	1/25	1/25	-	0.0014
q3.3	$L \bowtie \sigma(C) \bowtie \sigma(S) \bowtie \sigma(D)$	-	6/7	1/125	1/125	-	0.000055
q3.4	$L \bowtie \sigma(C) \bowtie \sigma(S) \bowtie \sigma(D)$	-	1/84	1/125	1/125	-	0.00000076
q4.1	$L \bowtie \sigma(S) \bowtie \sigma(C) \bowtie \sigma(P) \bowtie D$	-	-	1/5	1/5	2/5	0.016
q4.2	$L \bowtie \sigma(S) \bowtie \sigma(C) \bowtie \sigma(P) \bowtie \sigma(D)$	-	2/7	1/5	1/5	2/5	0.0046
q4.3	$L \bowtie \sigma(S) \bowtie \sigma(C) \bowtie \sigma(P) \bowtie \sigma(D)$	-	2/7	1/25	1/5	1/25	0.000091

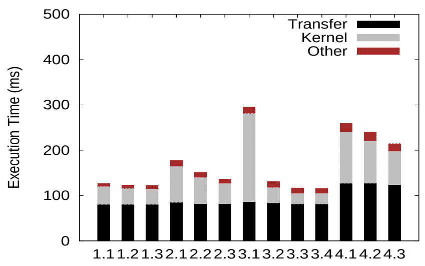


Figure 4: Baseline of SSBM queries on NVIDIA GTX 680.

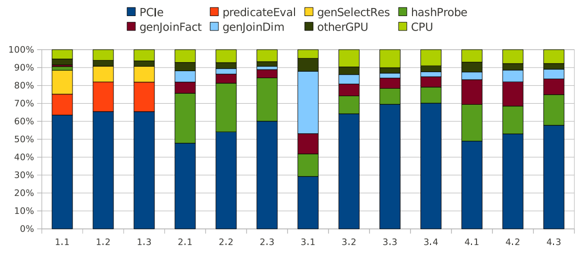


Figure 5: SSBM execution time breakdown

we distinguish the pageable host memory from the pinned host memory. To measure the latter, we launch two GPU kernels which read/write 256MB integers from/to GPU device memory in a coalesced manner. The measured results are reported in Table 3. As is shown in Table 3, the PCIe transfer bandwidth becomes higher when the host memory is pinned (e.g., doubled for GTX 680). The reason is that for pinned memory, data can be directly transferred using GPU DMA engine. However, for pageable memory, data need to be copied to a pinned DMA buffer first before transferred using GPU DMA engine [1].

4. PERFORMANCE ANALYSIS

In this section we present the characterization of query behaviors and the effects of software optimizations when executing SSBM queries on NVIDIA GTX 680.

4.1 SSBM Query Behaviors

Figure 4 shows the baseline SSBM performance conducted on GTX 680 with pinned memory. We breakdown the execution time into PCIe transfer (Transfer), kernel execution(Kernel) and other (Other) which mainly includes time spent on initializing data structures on CPU before launching the kernels and allocating and releasing GPU device memory. As is shown in Figure 4, most execution time for SSBM queries are spent on PCIe transfer and kernel execution. To understand the query behaviors, we further breakdown the execution time and shows the percentage of the major operations for SSBM queries in Figure 5.

Since the size of fact table is much larger than the size of dimension tables, the number of columns of fact table used in the query determines the PCIe transfer time. Queries in the same query flight have almost the same PCIe data transfer time since they process the same amount of data from the fact table.

Query flight 1. The kernel execution time of queries in flight 1 are dominated by selection operations, as is shown in Figure 5. Most of the kernel execution are spent on the predicate evaluation of the selection (predicateEval) and generating selection results (genSelectRes).

Query flight 2. For queries in flight 2, a large portion of their kernel execution time are spent on the hash probing operation in the join operator (hashProbe) and generating join results (genJoinFact and genJoinDim), as is shown in Figure 5. One key difference among their query characteristics is the join selectivity, which decreases from Q2.1 to Q2.3. As higher join selectivity implies higher kernel execution time, the kernel execution time will decrease from Q2.1 to Q2.3, as is shown in Figure 4.

Query flight 3. The query behaviors in flight 3 can be divided into two groups: Q3.1 and Q3.2 to Q3.4. The kernel execution time of Q3.1 are dominated by the access of dimension tables when generating join results (genJoinDim) while the kernel execution time of Q3.2 to Q3.4 are dominated by hash probing probing operation (hashProbe) and accessing the data from the fact table when generating the join results (genJoinFact). We use Q3.1 as an example to illustrate the differences.

```
Q3.1 from SSBM:
select c_nation, s_nation,
      d_year, sum(lo_revenue) as revenue
from  customer, lineorder, supplier, date
where lo_custkey = c_custkey
      and lo_suppkey = s_suppkey
      and lo_orderdate = d_datekey
```

Table 4: Compression ratio for fact table columns when sorted on different foreign key columns.

Column	lo_custkey	lo_partkey	lo_suppkey
lo_custkey	1%	100%	100%
lo_partkey	100%	3%	100%
lo_suppkey	50%	50%	0.1%
lo_orderdate	50%	50%	50%
lo_extendedprice	100%	100%	100%
lo_quantity	25%	25%	25%
lo_discount	25%	25%	25%
lo_revenue	100%	100%	100%
lo_supplycost	50%	50%	50%

```
      and c_region = 'ASIA'  and s_region = 'ASIA'
      and d_year >= 1992 and d_year <= 1997
group by c_nation, s_nation, d_year
order by d_year asc, revenue desc;
```

Q3.1 has a high join selectivity. The join selectivities for *customer* and *supplier* are both 20%. Each of these two joins needs to access the data in the dimension tables to form the results. To be more specific, they access *c_nation* from *customer* and *s_nation* from *supplier*. As the join selectivities are high, there are lots of random accesses to the dimension tables. In this case, a large portion of the kernel execution time are spent on this part. For Q3.2 to Q3.4, they share many characteristics with Q3.1 but with very low join selectivities. Their execution time are dominated by the sequential scan of fact table data in hash probing and generating result operations.

Query flight 4. The execution time of queries in flight 4 are dominated by hash probing and generating join results from fact table. Q4.1 and Q4.2 both have high join selectivities while Q4.3 has a relatively low join selectivity. Q4.1 has similar query characteristics as Q3.1 but doesn't spend much time on accessing the data from dimension tables. The main reason is that the first executed join for Q4.1 doesn't access any column from dimension table while Q3.1 does.

4.2 Effects of data compression

Our GPU query engine supports three light weight data compression schemes that have already been widely used in column-store systems: Run Length Encoding (RLE), Bit Encoding and Dictionary Encoding.

For performance benefits, fact table is stored in multiple disk copies. Each copy of the fact table is sorted on a different foreign key column. All the sorted columns are compressed using RLE. The rest columns are compressed using the other two schemes whenever possible. Dimension tables are not compressed since their sizes are much smaller compared to the size of fact table. Table 4 shows the compression ratio for the fact table columns used in SSBM queries.

The query engines can obtain significant performance benefits when directly working on the compressed data [7]. Thus our engine directly operates on the compressed data whenever possible. One representative operation that can directly work on the compressed data is the hash probing operation in join operator. It directly scans the compressed foreign keys and probes the hash table. As foreign key columns are usually compressed with high compression ratios, operating directly on the compressed data will significantly reduce the number of hash probing operations. On the other hand, some operations have to decompress the data during their execution, such as result projection operation. The decompression will generate many irregular device memory accesses which makes it an expensive operation.

Figure 6(a) shows the speedup of PCIe data transfer, kernel execution and the overall performance after data are compressed. Disk loading time is not included in the total execution time. For all queries, data compression can effectively reduce the PCIe transfer time due to the reduced amount of transferred data.

For selection dominated queries, as is the case for queries in flight 1, their kernel execution time cannot benefit much from the data compression technique. Most of their kernel operations access data in a coalesced manner. Although some kernel operations, such as generating the filter vector, can directly work on compressed data, the performance benefit is not much since GPU can well handle coalesced memory accesses.

For queries dominated by join operations, when a large portion of the kernel execution time are spent on generating join results, their kernel execution time cannot benefit much from the data compression technique, as the case for Q3.1, Q4.1 and Q4.2. These queries usually have high join selectivities and several projected columns from both the fact table and the dimension tables. When a large portion of the kernel execution time are spent on hash probing operations, their kernel execution time can benefit greatly from the data compression technique, as is the case for queries in flight 2. When queries have very low selectivities and several projected columns from the fact table, as is the case for Q3.2 to Q3.4, their execution time will be dominated by the coalesced accesses of the data from the fact table. They cannot benefit much from the data compression technique.

4.3 Effects of Transfer Overlapping

Both OpenCL and CUDA support a unified address space for host memory and GPU device memory. The GPU kernels can directly access the data stored in the pinned host memory. No explicit PCIe data transfer is needed. We use transfer overlapping to refer to this technique.

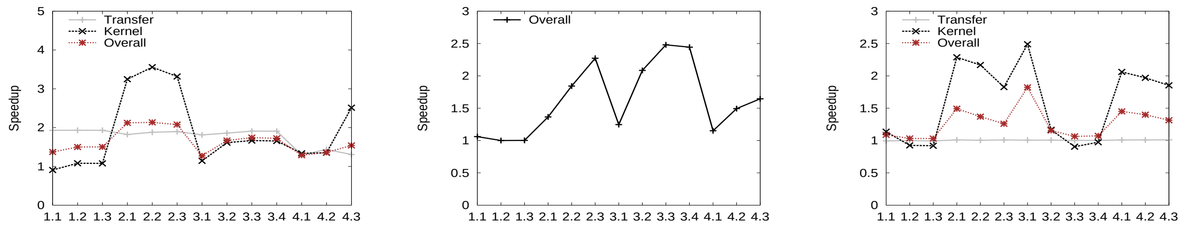
The performance benefits of utilizing transfer overlapping come from two aspects: the increased PCIe transfer bandwidth from pageable memory to pinned memory, and the overlapping between PCIe transfer and kernel execution.

To examine its impact on query performance, we pin the host memory that is used to store the data from fact table. Generally there are two reasons for this. First, the host resident data should be accessed in a coalesced way to fully utilize the PCIe bandwidth. Second, the size of fact table is much larger than the dimension table and most of the PCIe transfer time is spent on transferring data from fact table.

We compare the performance of SSBM queries with transfer overlapping with the baseline. The performance speedup is shown in Figure 6(b). Since the PCIe transfer operations become implicit with transfer overlapping, we only present the speedup of the total execution time.

For queries in flight 1, the performance doesn't improve. This is because some columns from fact tables are accessed more than one time by the kernel. In this case, the relatively low PCIe bandwidth compared to the bandwidth of GPU device memory will counteract the benefits of the overlapping of kernel execution and PCIe data transfer.

When data are accessed only once through PCIe bus, as for queries in flight 2 - 4, query performance will improve.



(a) Speedup of data compression (b) Speedup of transfer overlapping (c) Speedup of invisible join

Figure 6: Effects of different software optimization techniques

As the performance gains mainly come from the sequential access of fact table columns, the more time spent on these operations, the more performance gains the query will get. So queries with low selectivities, and with more columns from fact table are more likely to benefit from transfer overlapping, such as Q2.2 to Q2.3 and Q3.2 to Q3.4. Increased selectivity, and more projected columns from dimension tables will increase the kernel time than spent on hash probing and accessing of data from dimension tables that cannot benefit from this technique because of their random access pattern, as for the rest queries.

4.4 Effects of Invisible Join

Data compression and transfer overlapping can improve the performance of a wide range of queries. However, they are not effective for queries dominated by random accesses to data in the dimension tables, like Q3.1. Invisible join is an optimization technique that can help improve the performance of this kind of queries.

Invisible join was proposed in [8] to improve the performance of star schema joins in CPU environments. It rewrites the foreign key joins into predicates on fact table, which can be evaluated at the same time before generating the final results. One benefit of this technique is that the number of random accesses to dimension tables can be greatly reduced. When rewriting the joins, a between-predicate-rewriting technique can be utilized to transform the operation of probing the hash table into selection operations on foreign key columns in the fact table if the primary keys lie in a continuous value range.

Currently our query engine doesn't support rewriting the joins automatically at run time. In this case, we manually rewrite all the queries before examining their performance. To make the primary keys lie in a continuous range, all the dimension tables are sorted on corresponding columns. After query rewritten, selection on dimension table and hash join operation are completely replaced with selections on the fact table for queries in query flight 1. For the rest queries, selection on dimension tables and hash probing operations are replaced with selections on the fact table.

Figure 6(c) shows the speedup of PCIe data transfer, kernel execution and overall performance when enabling invisible join. Since invisible join doesn't change the amount of transferred data from fact table, it has no impact on PCIe transfer time.

Whether the kernel execution time of a query can benefit from this optimization depends on whether its execution time is dominated by hash probing operation or the operation on data from dimension table. The performance of queries with high selectivities, and with operations on dimension table data are more likely to be improved, as is the case for Q3.1. On the other hand, for queries with low selectivities and have multiple foreign key joins, they cannot benefit much from invisible join technique. In the worst case, the kernel execution time even degrades, for example, for Q3.3 and Q3.4. This is because these queries have a very limited number of accesses to dimension tables. In this case, the benefit brought by invisible join is so small that it cannot counteract the increased kernel time by selection operations on the foreign key columns. For selection dominated queries, as is the case for queries in flight 1, their kernel execution times remain almost the same.

To apply the invisible join technique, both dimension tables and foreign keys in fact table need to be reorganized which may be very costly for general purpose warehousing systems. Therefore, we do not include invisible join technique in our following performance studies.

5. PERFORMANCE COMPARISON

Having studied query execution behaviors and software optimization effects on GPU, we are in a position to compare our GPU query engine with the CPU counterpart under different conditions. Our purpose is to reveal the advantages and disadvantages of the GPU engine, as well as the suport and limitations of the current GPU programming environments. Specifically, we will answer the following questions:

- Under what conditions will GPU significantly outperform CPU for processing warehousing queries? (Section 5.1)
- Which programming model is more suitable and more supportive for programming warehousing queries on GPU, CUDA or OpenCL? (Section 5.2.1)
- How do different GPU hardwares and their supporting systems affect query performance when their basic harware parameters are similar? (Section 5.2.2)
- With the functional portability of OpenCL, how will the OpenCL query engine that is designed for GPU perform compared to MonetDB? (Section 5.3)

5.1 Comparisons of GPU and CPU

- *The GPU query engine outperforms the CPU query engine for processing all SSBM queries. However, the performance speedup varies significantly depending on query characteristics and system setups.*
- *The key to obtain high query execution performance on GPU is to prepare the data in the pinned memory, where 4.5x-6.5x speedups can be observed for certain queries. When data are in the pageable memory, the speedups are only 1.2x-2.6x for all SSBM queries.*

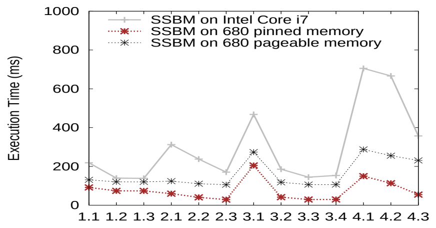


Figure 7: SSBM performance comparison. For the performance on Intel Core i7, the performance of Q4.1 and Q4.2 are the performance on OpenCL engine while the rest are the performance on MonetDB.

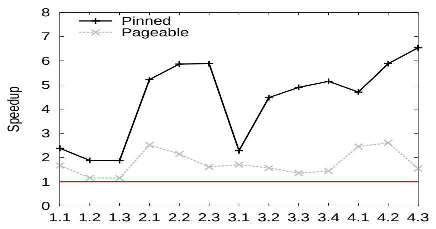


Figure 8: SSBM performance speedup

- *GPU has limited speedups (around 2x) for queries: 1) dominated by selection operations, and 2) dominated by random accesses to dimension tables caused by high join selectivities and projected columns from dimension tables.*

Our comparisons are based on the following two kinds of performance numbers. First, the GPU performance is the performance of the CUDA engine on NVIDIA GTX 680. Second, the CPU performance for each query is the better one between the performance of MonetDB and of our OpenCL query engine on Intel Core i7. We conduct the experiments under two conditions: 1) data are available in the pinned memory; and 2) data are available in the pageable memory. Figure 7 shows the execution time of SSBM queries and Figure 8 shows the performance speedup of GPU over CPU.

5.1.1 Data are available in the pinned memory

When data are available in the pinned memory, both the data compression technique and the transfer overlapping technique can be utilized to accelerate the query execution on GPU. As can be seen in Figure 8, GPU outperforms CPU in all SSBM queries. However, the performance speedup varies significantly. The performance differences come from the differences in query characteristics. Whether we can gain significant speedup when processing query on GPU depends on whether the query can fully benefit from different software optimization techniques and whether it can utilize the GPU hardware effectively. We divide the performance speedup into two categories.

Category of Low speedup. For Q1.1 to Q1.3 and Q3.1, processing on GPU can only gain around 2x speedup, as is shown in Figure 8. Queries in flight 1 are dominated by selection operations. They cannot benefit from the transfer overlapping technique. Although data compression technique can reduce the PCIe transfer overhead, the kernel execution performance cannot be improved. Since selection doesn't involve much computation, processing on GPU will not have significant performance speedup. Q3.1 is dominated by the random accesses of data from dimension tables. It cannot benefit much from both the data compression technique and the transfer overlapping technique. Furthermore, the random accesses cannot effectively utilize the bandwidth of GPU device memory. In this case, we cannot gain significant performance speedup.

Category of High speedup. For Q2.1 to Q2.3, Q3.2 to Q3.4 and Q4.1 to Q4.3, processing on GPU can gain a 4.5x to 6.5x speedup , as is shown in Figure 8. The kernel execution time of Q2.1 to Q2.3 are dominated by the hash probing operation of the join operation. It can benefit from both the data compression technique and the transfer overlapping technique. The kernel execution time of Q3.2 to Q3.4 and Q4.1 to Q4.3 are dominated by both the hash probing operation and the projection of join results from the fact table. The projection of join results can benefit from the transfer overlapping technique. In this case, queries which are dominated by hash probing operation and result projection operation from the fact table can gain a significant speedup when processed on GPU.

5.1.2 Data are available in the pageable memory

When data are available in the pageable memory, only data compression technique can be utilized to accelerate the query execution on GPU. As can be seen in Figure 8, the performance speedup degrades greatly compared to data in the pinned memory. Most SSBM queries only gain a speedup of around 2x. For Q1.2 and Q1.3, the performance speedups are only 1.15x. The main reason is that the PCIe transfer bandwidth cannot be fully utilized when data are in the pageable memory. The benefits of GPU's high memory bandwidth and high computational power are mostly counteracted by the high PCIe transfer overhead.

5.2 Impacts of programming models and GPU hardwares

- *From both the performance and programming perspective, CUDA is more suitable and supportive for processing warehousing queries.*

- *Without using the pinned memory, the NVIDIA OpenCL query engine can have similar performance with the CUDA engine. However, NVIDIA OpenCL haven't well supported pinned host memory.*

- *The performance slowdown when porting NVIDIA CUDA (on GTX 680) to AMD OpenCL (on 7970) is not caused by the differences in hardware efficiencies (PCIe transfers time or kernel executions), but by AMD's OpenCL implementation for GPU memory management.*

- *The major obstacle to OpenCL portability is not performance slowdown of GPU kernel executions but subtle differences of vendor implementations for the OpenCL specification.*

5.2.1 Comparisons of CUDA and NVIDIA OpenCL

To compare these two programming models, we focus on the NVIDIA GTX 680 which can run both CUDA programs and OpenCL programs.

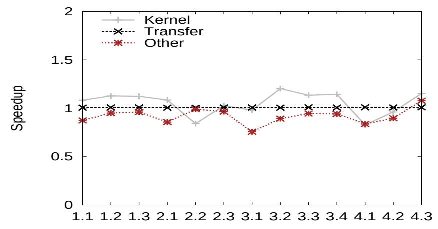


Figure 9: Normalized OpenCL performance over CUDA

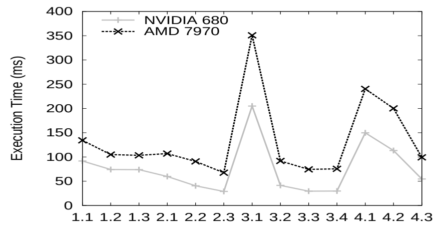


Figure 10: NVIDIA Versus AMD

Programming differences. Since the design of CUDA and OpenCL share many concepts in common, the programming efforts are similar for warehousing queries. However, NVIDIA's OpenCL implementation makes it impossible to apply all software optimization techniques when running OpenCL engine on NVIDIA GPU. The problem is NVIDIA OpenCL doesn't well support pinned host memory. In the experiments we find that on NVIDIA GPU, the sum of regular allocated device memory and the memory allocated with flag CL_MEM_ALLOCATE_HOST_PTR, which should be allocated in pinned host memory [5], cannot exceed the total size of GPU device memory. In this case, we cannot prepare the data in the pinned memory before query execution because of the large size of the data and can only utilize the data compression technique.

Performance differences. Considering the above limitation, we compare the query performance with pageable host memory and data compression technique. The CUDA query engine and the OpenCL query engine use the same thread configurations and the same algorithms. We breakdown the execution time into PCIe transfer (Transfer), kernel execution (Kernel), and other (Other) which mainly includes allocating and releasing GPU device memory and other operations on CPU. We normalize the OpenCL performance on CUDA for each part.

As is shown in Figure 9, warehousing queries implemented in CUDA and in OpenCL have almost the same performance. This differs from the results of the HPC applications where a significant performance difference exists when simply porting the CUDA implementation into OpenCL implementation [15]. The difference is mainly determined by the characteristics of the warehousing workloads, the performance of which are not bounded by computing but by PCIe data transfers and memory accesses. First, both programming models don't affect the PCIe transfer bandwidth. Second, both programming models can well support GPU memory hierarchy. The computation-oriented optimization techniques from CUDA compiler as reported in [15] doesn't apply to warehousing workloads.

5.2.2 *Comparisons of NVIDIA and AMD GPUs*

We compare the performance of SSBM queries on the CUDA query engine on NVIDIA GTX 680 with the performance of SSBM queries on the OpenCL query engine on

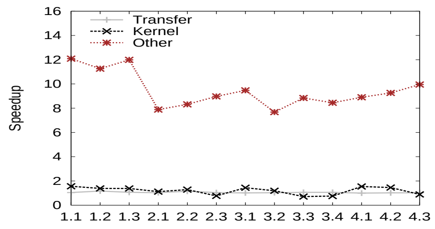


Figure 11: Normalized SSBM performance on AMD GPU

AMD HD 7970. Both engines have been optimized with the data compression technique and the transfer overlapping technique. The results are shown in Figure 10.

As can be seen in Figure 10, SSBM queries on NVIDIA 680 outperforms SSBM queries on AMD 7970. The performance gap is almost constant among all SSBM queries. To understand the performance differences, we remove the transfer overlapping technique from both engines so that we can breakdown the execution time. We breakdown the execution time into PCIe transfer (Transfer), GPU kernel execution (Kernel), and other (Other) which mainly includes allocating and releasing GPU device memory and other operations on CPU. For each part, we normalize the performance on the performance of AMD GPU. The results are shown in Figure 11.

As is shown in Figure 11, these two GPUs have comparable performance for PCIe data transfer and kernel execution. Since these two GPUs have comparable hardwares, we expect that they should have similar performance for SSBM queries. Considering the Other time, AMD GPU has a much longer execution time. To more clearly explain why the CPU time is much longer on AMD GPU, we use a simple data transfer process to illustrate where the time is spent.

We first allocate a buffer from GPU device memory. Then we transfer the data to the buffer. We measure the transfer time in two different ways. In the first way the total time is recoded as the difference between the time when we launch the PCIe transfer and the time when the transfer is finished, both of which are measure using wall clock time. In the second way, we measure the transfer time using OpenCL events. When we examine these two transfer times, the one that is measured in the second way is what we expect based on the measurements of PCIe transfer bandwidth. However, the one that is measured in the first way is much longer than the one measured in the second way. This attributes to the overall performance gap between processing SSBM queries on NVIDIA and AMD. The reason may relate to AMD OpenCL's implementation of memory object management. When allocating a memory from GPU device memory, AMD driver defers the allocation until the memory object is first used. When the memory initialization cost is high, the performance suffers.

5.3 Comparisons of OpenCL query engine on CPU with MonetDB

- *Porting the OpenCL query engine from GPUs to CPU can work well by changing each thread's memory access pattern and thread configurations.*

- *MonetDB outperforms the OpenCL query engine for processing selection dominated queries and join dominated queries with low selectivities.*

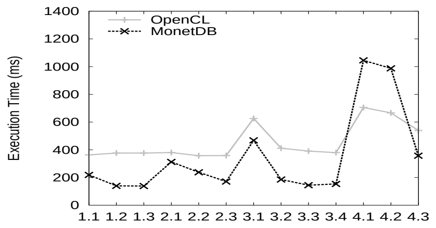


Figure 12: SSBM performance comparison on CPU

- *The OpenCL query engine has comparable or better performance for join dominated queries with high selectivities.*

Our OpenCL query engine can work on both CPUs and GPUs. Studying the performance of the OpenCL query engine on CPUs will help partition the workload among CPUs and GPUs. We compare the performance of the SSBM queries on the OpenCL query engines with the performance on MonetDB. For the OpenCL query engine, we make changes to our GPU based algorithms to adapt to CPU architectures. We change the access pattern of each thread when running on CPU. Each thread will access a continuous range of the data instead of accessing the data in a stride way. All other algorithms remain the same. The execution time is shown in Figure 12.

MonetDB significantly outperforms our OpenCL query engine on CPU for selection dominated queries, such as Q1.1 to Q1.3. The performance gap is caused by the inefficiency of GPU algorithms when executed on CPU. In the implementation of GPU algorithms for selection operator, the where predicates need to be evaluated and the number of results must be calculated before generating selection results. In this case, when there are duplicated columns in the where predicates and the projected columns as is the case for Q1.1 to Q1.3 in SSBM, the GPU selection algorithm will scan the column twice which is not necessary when executed on CPU. This will increase the execution time of our OpenCL query engine when executing on CPU.

The performance gap between our OpenCL query engine on CPU and MonetDB for join dominated queries is determined by join selectivities. We observe that the performance advantage of MonetDB over our OpenCL query engine decreases as the join selectivity increases. For example, for Q3.2 to Q3.4 which have very low selectivities, MonetDB is more than 2x faster compared to our OpenCL engine. However, for Q4.1 and Q4.2, which have high selectivities, our OpenCL engine even outperforms MonetDB. This can be further proved by Q2.1 to Q2.3, as is shown in Figure 12. The performance gap between our OpenCL query engine and MonetDB increases as the query selectivity decreases from 0.008 to 0.00016. We believe this is because MonetDB can effectively utilize CPU cache when join selectivities are low. For our OpenCL query engine on CPU, optimizing for the CPU cache is our future work.

6. MODEL AND PREDICTION

In this section we first introduce our modeling methodology and then show a case study of modeling join. After presenting the model accuracy evaluation, we use the model to predict the impact of GPU hardware advancements on warehousing query performance.

6.1 Model Methodology

Our model focuses on the architecture that GPU is connected with CPU through a PCIe bus. Data must be transferred to the device memory before the query executes and results will be transferred back to the host memory after the query finishes. We assume that data are already available in the host memory and are laid out in a column-store format. When estimating the query execution cost, some statistics like data size and selectivity of the tables are needed. We assume that these statistics are available and can be directly used by our model.

The total cost of executing a query on GPU consists of PCIe data transfer cost and kernel execution cost. While the PCIe data transfer cost can be estimated based on the available table statistics, the key is to estimate the query's kernel execution cost. As we have already discussed, the performance of data warehousing queries on GPUs are mainly bounded GPU device memory accesses. Thus we focus on the estimation of device memory access cost and use device memory access time as the metric to represent the query kernel execution cost.

The memory access time of a given query can be calculated as the amount of actual accessed data in GPU device memory divided by the bandwidth of GPU device memory. In our model we view the GPU device memory as a group of continuous memory segments, each of which is the basic access unit of the device memory. We use the concept thread group to represent the basic threads management unit in GPU, which is similar to NVIDIA's warp and AMD's wavefront. Threads in the same thread group execute in lockstep and when they need to access the device memory, the number of needed memory segments will be calculated and then the corresponding segments are fetched to the thread group.

With the above abstraction of GPU device, we can estimate the number of actual device memory transactions and calculate the amount of accessed data. We don't distinguish between the coalesced access and uncoalesced access to device memory because the memory bus utilization is determined by the number of actual memory transactions, not by whether the access can be coalesced or not. For a given operator, the estimation of the number of memory transactions depends on the implementation of the query operator and the distribution of the data. The estimation of the cost of a complex query is based on the estimation of the cost of each single query operator. The total cost can be calculated as the sum of the costs of all the single query operators. In the next section, we model the join operator of our query engine as a case study for our methodology. Due to limited page space, the models and evaluation for other query operators are presented in `http://www.cse.ohio-state.edu/~yuanyu/report.html` .

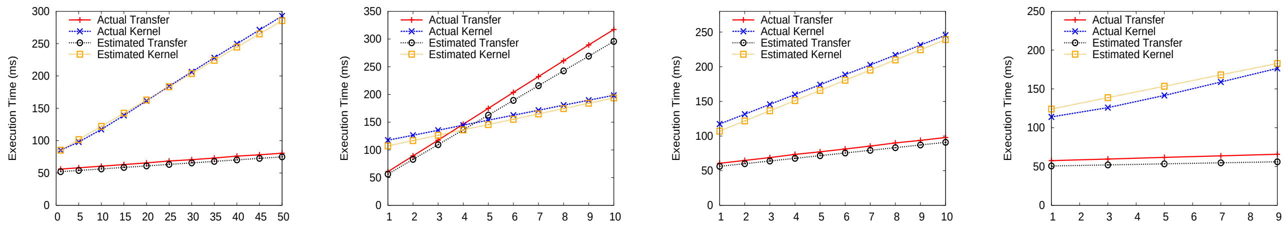
6.2 Cost Model for Join

The notations related to join query are list as follows, while other used notations are listed in Table 5.

r - join selectivity,
$||R||$ - cardinality of the fact table R,
$||S||$ - cardinality of the dimension table S,
n - number of projected columns from fact table,
m - number of projected columns from dimension table,
R_i - the attribute size of the ith projected column from fact table, and

Table 5: Notations for the Cost Model

Notations	Descriptions
B_r	The read bandwidth of the device memory
B_w	The write bandwidth of the device memory
B_i	The transfer bandwidth from host memory to device memory
B_o	The transfer bandwidth from device memory to host memory
C_r	The read segment size of the device memory
C_w	The write segment size of the device memory
S_i	The size of input data
S_o	The size of result
W	The number of threads in a thread group
T_i	The device memory access cost of the ith step to finish the query on GPU
T_t	The data transfer cost



(a) Selectivity (%) (b) # of fact columns (c) # of dim columns (d) Dim column width

Figure 13: Evaluate join model for different query characteristics

S_i - the attribute size of the ith projected column from dimension table.

When calculate the join cost, we assume the join keys are 4-byte integers and are not projected by the join operator, as is the case for all SSBM queries.

When building the hash table, the primary keys of the dimension table are scanned twice. The first scan is to calculate the start output position for each hash key. The second scan is to write the primary keys to the hash table with the tuple ids. While primary keys are sequentially scanned, writes to the hash table can be considered random. Then the approximate cost of memory access is:

$$T_1 = 2 \times \frac{||S||}{W} \times \lceil\frac{4 \times W}{C_r}\rceil \times \frac{C_r}{B_r} + ||S|| \times \lceil\frac{4 \times 2}{C_w}\rceil \times \frac{C_w}{B_w}.$$

When probing the hash table, the foreign keys of the fact table are sequentially scanned. For each foreign key, its hash value is calculated and the number of hash entries for the corresponding bucket is read. If the number is greater than 0, the position and the actual value of the ids of the corresponding dimension tuple is read. The filter is sequentially written either with the ids of the dimension table or 0. The scan of foreign keys, and the read and write of filter are sequential, while other requests can be considered random. Thus the approximate memory access cost is estimated as:

$$T_2 = (\frac{||R||}{W} \times \lceil\frac{4 \times W}{C_r}\rceil + ||R|| + 3 \times ||R|| \times r \times \lceil\frac{4}{C_r}\rceil) \times \frac{C_r}{B_r} + \lceil\frac{4 \times W}{C_w}\rceil \times \frac{||R||}{W} \times \frac{C_w}{B_w}.$$

When projecting the join results, the filter is sequentially scanned. The read of the fact table depends on the data distribution of the foreign keys while the read of the dimension table can be considered as a total random read. We consider the worst case that the foreign keys are uniformly

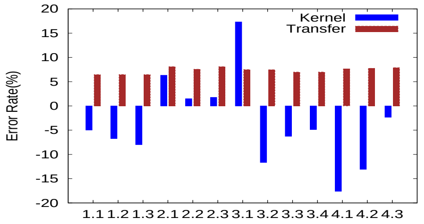


Figure 14: Error rate of estimated performance on 680.

distributed. Then the approximate cost of memory access is calculated as:

$$T_3 = \sum_{i=1}^{n}(\lceil\frac{4 \times W}{C_r}\rceil \times \frac{||R||}{W} + \lceil\frac{R_i}{4}\rceil \times \frac{||R||}{W}) \times \frac{C_r}{B_r} + (\sum_{i=1}^{m}\lceil\frac{4 \times W}{C_r}\rceil \times \frac{||R||}{W} + ||R|| \times r \times \sum_{i=1}^{m}\lceil\frac{S_i}{C_r}\rceil) \times \frac{C_r}{B_r} + ||R|| \times r \times (\sum_{i=1}^{n}\lceil\frac{R_i}{4}\rceil + \sum_{i=1}^{m}\lceil\frac{S_i}{4}\rceil) \times \frac{C_w}{B_w}.$$

The input data size and the result size can be calculated as:

$$S_i = ||R|| \times \sum_{i=1}^{n} R_i + ||S|| \times \sum_{i=1}^{m} S_i + 4 \times (||R|| + ||S||).$$

$$S_o = (||R|| \times \sum_{i=1}^{n} R_i + ||R|| \times \sum_{i=1}^{m} S_i) \times r.$$

The data transfer cost can be calculated as:

$$T_t = \frac{S_i}{B_i} + \frac{S_o}{B_o}.$$

6.3 Model Evaluation

We evaluate our cost model both for the join operator and for the SSBM queries. We use the NVIDIA GTX 680 with pinned host memory as the platform. For join operator, we

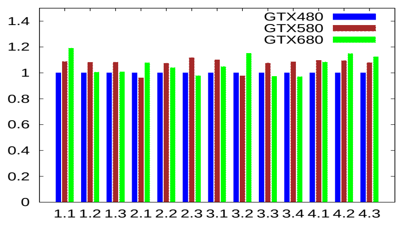


Figure 15: Normalized kernel execution time on GTX 480

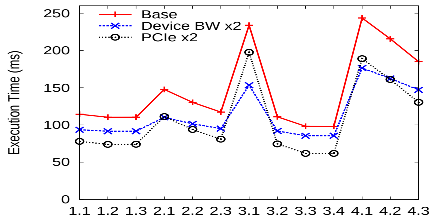


Figure 16: Estimated SSBM performance with different GPU hardware configurations

compare the estimated and actual performance of join operator under various query characteristics. When evaluating our model for SSBM queries, we define the error rate as:

$$error\_rate = \frac{measured\_time - estimated\_time}{measured\_time}$$

Figure 13 and Figure 14 present the evaluation results. As shown in the figures, the estimated performance are very close to the actual performance in most cases, which demonstrates the effectiveness of our cost model. Considering the differences between the estimated and actual performance, generally two factors account for this. First, many work executed on GPU need to be initiated by CPU. In this case information and some data must be transferred between CPU and GPU. Second, GPU is inefficient in handling irregular data accesses. For example, in GTX 680, the GPU memory transaction must be aligned on 32. When threads inside the warp issue unaligned memory access requests, more memory transactions will be generated even if the threads access the data in a coalesced manner. In this way, it is difficult to accurately estimate the memory access, which can be seen from Q3.1 and Q4.1 in Figure 14.

6.4 Impacts of hardware advancement

To study the impact of GPU hardware on query performance, we run SSBM queries on three generations of NVIDIA GPUs: GTX 480, 580 and 680 with pinned memory. We focus on the kernel execution time which is determined by the GPU internal architectures. We normalize the kernel execution time on the kernel execution time of SSBM queries running on GTX 480. The results are shown in Figure 15. As can be seen, the differences of kernel execution time are around 10% from most queries when running on these three GPUs. Compared to the improvement of GPU's peak performance (more than 2 times from GTX 480 to 680), the performance gain is very small. The reason is that the performance of warehousing queries are mainly bounded by GPU device memory accesses. They cannot benefit much from the increased computational power.

To predict the possible impact of the advancement of GPU hardwares on query performance, we use our model, which has been proved effective in estimating query performance on GPUs, to estimate the query performance with different GPU hardware configurations. We double PCIe transfer bandwidth and GPU device memory bandwidth independently based on GTX 680's hardware parameters to see how the overall SSBM performance change. We use the performance of SSBM queries on GTX 680 as the baseline.

The result is shown in Figure 16. Doubling the PCIe transfer bandwidth is more effective than doubling the device bandwidth for most queries. as most queries are still dominated by PCIe data transfer. But as the PCIe transfer bandwidth increases, the query execution time spent on PCIe transfer and kernel execution become comparable. However, in the real world scenario, the bandwidth of GPU device memory grows at a much slower pace compared to the improvement of its peak performance. In this case, the performance of data warehouse queries is not likely to benefit much from the advancement of GPU hardwares.

7. OTHER RELATED WORK

There are already a set of research papers on optimizing various database operations on GPUs [12, 18, 20, 28, 17, 29, 16, 30, 21, 19, 23, 32, 10, 33, 25]. Our unique contribution in this paper is presenting a comprehensive study for complex data warehousing queries with different software optimizations and hardware configurations. Several existing research work are related to our software optimization study. Data compression on GPU has been studied in [16], and transfer overlapping has been studied in [23, 28]. Compared to these work, our work focus on how these techniques can optimize different types of complex queries.

A cost model for GPU query processing was proposed in [19]. The essential difference between it and our model is how to estimate the time spent on GPU device memory access, which is the most important step to accurately estimate the cost of GPU query processing. The previous model assumes a fixed uncoalesced bandwidth that is applied to all different uncoalesced memory accesses. However, this assumption is not consistent with the current NVIDIA GPU where uncoalesced accesses with certain patterns can have a 100% memory bus utilization [3]. Our model takes a hardware feature oriented methodlogy to estimate the acutal memory transations in GPU device memory, which can better estimate the GPU memory bus utilization.

8. CONCLUSIONS

We have comprehensively evaluated GPU query execution performance with detailed analysis and comparisons between GPUs and CPU. We conclude that the reasons why GPUs have not been adopted in data warehouse systems include: 1) GPUs only significantly outperform CPU for processing certain kinds of queries when data are available in the pinned memory; 2) considering both performance and portability, current programming models are not supportive enough for warehousing workloads; and 3) the performance of warehousing queries doesn't increase correspondingly with the rapid advancement of GPU hardwares.

However, our analysis and comparisons give two clear R&D directions for adopting GPUs in the fittest way. First, a CPU/GPU hybrid query engine can maximze the hardware combination efficiency with task scheduling either in the query level or in the operator level. Second, GPUs should run query engine for the purpose of real-time business intelligence analytics for main memory database systems with minimal interference for transactions executed on

CPUs. Furthermore, the role of GPUs could also change considering the potential NVIDIA GPUDirect technique, which allows more efficient communication among GPU devices and storage devices. An important future research topic is to study how to make GPUs directly process data stored in the permanent storage medias.

The query engine is open to the public and can be accessed at `http://code.google.com/p/gpudb/`.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Science Foundation under grants of CNS-0834393 and OCI-1147522.

10. REFERENCES

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

[2] Cuda c programming guide 5.0. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[3] Global memory usage and strategy. `http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf`.

[4] Opencl. `http://www.khronos.org/opencl`.

[5] Opencl programming guide for the cuda architecture. `http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf`.

[6] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, April 2007.

[7] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD Conference*, 2006.

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

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

[10] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 2011.

[11] C. Balkesen, J. Teubner, G. Alonso, and T. Ozsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, 2013.

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

[13] S. Blanas, Y. Li, and J. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[14] G. Candea, N. Polyzotis, and R. Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *PVLDB*, 2(1):277–288, 2009.

[15] P. Du, R. Weber, P. Luszczek, S. Tomov, G. D. Peterson, and J. Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

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

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, 2006.

[18] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD Conference*, 2004.

[19] B. He, M. Liu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems*, 34(4), December 2009.

[20] B. He, K. Yang, R. Fang, M. Liu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[21] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *PVLDB*, 2011.

[22] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

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

[24] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25–36, 2011.

[25] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, 2008.

[26] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[27] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. Star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

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

[29] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort. In *SIGMOD*, 2010.

[30] E. Sitaridi and K. Ross. Ameliorating memory contention of olap operators on gpu processors. In *DaMoN*, pages 39–47, 2012.

[31] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. B. Zdonik. One size fits all? part 2: Benchmarking studies. In *CIDR*, 2007.

[32] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. *PVLDB*, 5(11):1543–1554, 2012.

[33] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *MICRO-45*, 2012.