Concurrent Analytical Query Processing with GPUs

Kaibo Wang* Kai Zhang‡* Yuan Yuan* Siyuan Ma* Rubao Lee*
Xiaoning Ding† Xiaodong Zhang*

*The Ohio State University †New Jersey Institute of Technology
‡University of Science and Technology of China

ABSTRACT

In current databases, GPUs are used as dedicated accelerators to process each individual query. Sharing GPUs among concurrent queries is not supported, causing serious resource underutilization. Based on the profiling of an open-source GPU query engine running commonly used single-query data warehousing workloads, we observe that the utilization of main GPU resources is only up to 25%. The underutilization leads to low system throughput.

To address the problem, this paper proposes concurrent query execution as an effective solution. To efficiently share GPUs among concurrent queries for high throughput, the major challenge is to provide software support to control and resolve resource contention incurred by the sharing. Our solution relies on GPU query scheduling and device memory swapping policies to address this challenge. We have implemented a prototype system and evaluated it intensively. The experiment results confirm the effectiveness and performance advantage of our approach. By executing multiple GPU queries concurrently, system throughput can be improved by up to 55% compared with dedicated processing.

1. INTRODUCTION

Multitasking has been a proven practice in computer systems to achieve high resource utilization and system throughput. However, despite the wide adoption of GPUs (Graphics Processing Units) for analytical query processing, they are still mainly used as dedicated co-processors, unable to support efficient executions of multiple queries concurrently.

Due to the heterogeneous, data-driven characteristics of GPU operations, a single query can hardly consume all GPU resources. Dedicated query processing thus often leads to resource underutilization, which limits the overall performance of the database system. In market-critical applications such as high-performance data warehousing and multi-client dataflow analysis, a large number of users may demand query results simultaneously. As the volume of data to be processed keeps increasing, it is also essential for user queries to make continuous progress so that new results can be generated constantly to satisfy the goal of interactive analysis. The lack of concurrent querying capability restricts the adoption of GPU databases in these application fields.

While dedicated usage of GPUs is still needed for latency-critical queries to ensure performance isolation, databases must be improved to support concurrent multi-query execution as an option to maximize the throughput of non-latency-sensitive queries on the GPU device. This consolidated usage of GPU resources enhances system efficiency and functionalities, but it makes the design of query execution engine more challenging. For maximal performance, each user query tends to reserve a large amount of GPU resources. Unlike CPUs where the operating system supports fine-grained context switches and virtual memory abstractions for resource sharing, current GPU hardware and system software provide none of these interfaces for database resource management. For example, GPU tasks cannot be preempted once started; on-demand data loading is not supported during task execution; automatic data swapping service is also missing when the device memory undergoes pressure. As a result, without efficient coordination by the database, multiple GPU queries attempting to execute simultaneously can easily cause low resource usage, system thrashing, or even query abortions, which significantly degrade, instead of enhance, overall system performance.

In this paper we present a resource management facility called *MultiQx-GPU* (**Multi-Q**uery e**X**ecution on **GPU**) to address the above challenges and support efficient executions of concurrent queries in GPU databases. It ensures high resource utilization and system performance through two key components: a query scheduler that maintains optimal concurrency level and workload on the GPUs, and a data swapping mechanism to maximize the effective utilization of GPU device memory. This paper also presents a prototype implementation of MultiQx-GPU in an open-source GPU query engine and discusses several technical issues addressed by our system to ensure its usability and efficiency in practice. Through intensive experiments with a wide range of workloads, we demonstrate the effectiveness and performance advantage of our solution. By supporting concurrent query processing, MultiQx-GPU improves system throughput by up to 55% relative to the system without such support.

This paper makes the following main contributions. First, we have made a strong case for building an effective resource sharing facility as a part of a database to manage concurrent query executions with GPUs. Second, we have shown the effectiveness of our design and implementation of the software facility with intensive experiments. Finally, the

Proceedings of the VLDB Endowment, Vol. 7, No. 11

software framework presented in this paper is open-source and can also be enhanced to support GPU resource sharing activities in other data processing applications, raising the productivity and system utilization.

The rest of the paper is organized as follows. Section 2 introduces the background and motivation of the research. Section 3 outlines the overall structure of MultiQx-GPU. Section 4 and 5 describe the device memory swapping and query scheduling components of MultiQx-GPU respectively. After a summary of the implementation issues in Section 6, Section 7 evaluates the prototype system, Section 8 introduces related work, and Section 9 concludes the paper.

2. BACKGROUND AND MOTIVATION

This section provides background on GPU query processing and motivates this research by exposing the problems of lacking multi-query support. Based on extensive benchmarks over some existing GPU query engines, we show the low resource utilization induced by dedicated query processing and identify several system issues that must be addressed in order to execute concurrent GPU queries efficiently.

2.1 Analytical Query Processing with GPUs

With vectorized cores and high-bandwidth device memory, GPUs have been widely utilized in databases for analytical query processing [32, 11, 17]. In this subsection we describe the architecture of one such system, called *YDB* [1], as an example to briefly introduce state of the art.

YDB is a standalone GPU execution engine for warehouse-style queries. Its front end consists of a *query parser* and *optimizer*, whose designs are based on the YSmart query translation framework [16]. It translates an SQL query into an optimized query plan tree, which is then used by the *query generator* to generate a driver program. This driver program controls the query execution flow; it is compiled and linked with the *GPU operator library* to produce an executable query binary. During execution, the query binary reads table data from a column-format backend storage and invokes the according GPU operators to offload data to GPUs for fast processing. Finally, the query results are materialized into row format and returned to the user.

To explain GPU query execution in more details, consider the following query that computes the total revenue from orders with discounts no less than 1% in each month of 1993:

```
SELECT d_month, SUM(lo_revenue)
FROM lineorder, ddate
WHERE lo_orderdate = d_datekey
      AND d_year = 1993 AND lo_discount >= 1
GROUP BY d_month
```

Figure 1 illustrates an execution plan generated by YDB for the query. It first performs a table scan on the fact table *lineorder*. The selection predicate $lo\_discount \geq 1$ is evaluated to generate a selection vector. With this vector, the scan operator filters *lo_orderdate* and *lo_revenue*, and returns an intermediate table consisting of the two filtered columns to the driver program. Similarly, with a selection predicate *d_year = 1993*, the driver program invokes a scan operation on the dimension table *ddate*, generating an intermediate table with the filtered *d_datekey* and *d_month* columns. Following the scans, the two intermediate tables are joined: a hash table is built on *d_datekey'* and probed with *lo_orderdate'* to generate a filtering vector, which is then used to filter the *d_month'* and *lo_revenue'* columns of

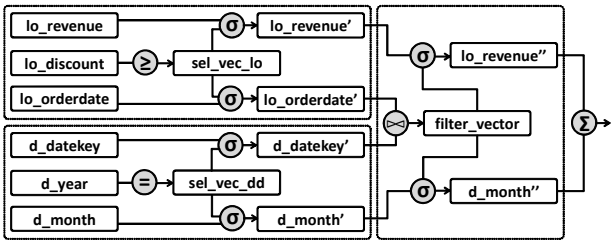


Figure 1: An example query execution plan in YDB.

the intermediate tables. In the end, the join output is aggregated (and materialized) to get the final query result.

The GPU operator library provides the GPU implementations of common database operations such as scans, joins, aggregations, and sorting. These operations are optimized at both kernel[1] and procedure levels in YDB. Shared memory and memory access coalescing are fully exploited to maximize single kernel performance. IOMMU-based direct host memory access (through CUDA [20] unified virtual addressing or OpenCL [15] mapped buffer interfaces) and data compression techniques are supported to mitigate data transfer overhead. To ensure kernel execution efficiency, table tuples are pushed from one operator to another in batches. For data sets that cannot directly fit into device memory, they are partitioned into smaller blocks and processed one by one.

Despite possible differences in implementation details, the core design principles of other analytical GPU query engines are similar to YDB. For example, Ocelot [11] is a hardware-oblivious parallel database engine supporting query executions on either CPUs or GPUs. Its integration with MonetDB [2] requires it to comply with the internal interfaces of MonetDB, but its column-based data stores, operator-at-a-time execution model, and the designs of major GPU operators agree with YDB closely.

2.2 Low Resource Utilization

Current analytical GPU engines such as YDB and Ocelot use GPUs as dedicated query co-processors. The query engine admits one user query at a time, generates and executes a query plan assuming exclusive usage of the GPU device. Although this dedicated query processing scheme simplifies query optimization and algorithm design, it inevitably causes low resource utilization due to the heterogeneous, data-driven features of query processing with GPUs.

A typical query execution comprises both CPU and GPU phases. The CPU phases are in charge of, e.g., initializing GPU contexts, preparing input data, setting up GPU page tables, launching kernels[2], materializing query results, and controlling the steps of query progress. These operations can take a notable portion of query execution time, which may cause GPU resources to be underutilized during these periods. Besides CPU phases, there also exist data dependencies amid various query stages. For example, a kernel cannot be launched until its input data are loaded into device memory or mapped to the GPU page table; aggregations cannot start until the join results are generated. Techniques such as double buffering can be used to mitigate data latency, but their applicability is constrained by the limited opportunities within a single query and the high complexity introduced to GPU operator designs. Assuming dedicated occupation of the device, GPU queries also tend

[1]A kernel is a data-parallel task executed on the GPU.

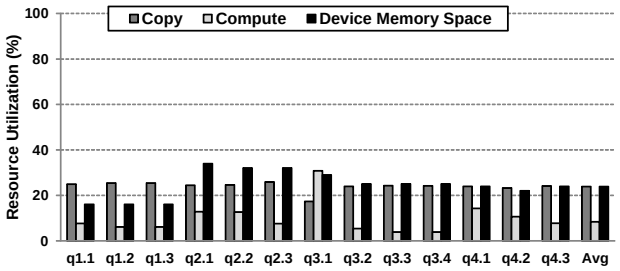


Figure 2: Utilization of GPU resources during dedicated executions of SSB queries with YDB.

to release reserved device memory space lazily to improve data reuses and simplify algorithm implementation. This lowers the effective usage of allocated space.

To show the problem of low resource utilization, we measure the executions of Star Schema Benchmark (SSB [21]) queries on a modern server with an Intel CPU and NVIDIA GPU (platform details in Section 7.1). We use YDB to generate an optimized binary for each of the 13 SSB queries at scale factor 14. For each binary, it is executed dedicatedly on the server for several times. We collect the average utilization of main GPU resources during one query execution. To minimize the influence of disk accesses, all data sets are preloaded into system memory before queries are executed.

Figure 2 depicts the utilization of three major types of GPU resources. The first two bars in each group give the utilization of GPU's copy and compute units. It can be seen that both hardware resources are poorly utilized when a query executes dedicatedly on the GPU. The copy unit, which is in charge of DMA data transferring, is in use for only 24% of query execution time on average. The compute unit, which executes the kernels of GPU operators, is even less utilized, accounting for an average of merely 8% of query makespan (4% at minimum). By further breaking down DMA traffic, we find that the overwhelming majority (over 99%) of data transfers are from the host to the device. Therefore, if a server-class GPU with dual copy units (e.g., an NVIDIA Tesla or Quadro GPU) is used in the production system, the device-to-host copy unit would remain (almost) completely idle through the entire query lifetime, wasting precious PCIe bandwidth resource.

Figure 2 also shows the low utilization of device memory space, as illustrated by the third bar in each group. Queries allocate device memory to hold their working sets for fast access. However, not all the allocated space may be always effectively utilized during query execution. We have instrumented YDB to collect memory traces and computed the space utilization, which is defined as the ratio of device memory space occupied by actively accessed data. It can be seen that, averaged across all queries, only 23% of allocated device memory space is effectively utilized, with the lowest near 16% for queries in the *q1* series. The allocated but underutilized space could not be put into better uses since only a single query was executed at any time in YDB.

The problem of low resource utilization has motivated us to exploit concurrent query processing with GPUs in the beginning. However, will be shown in the next subsection, with some critical components missing, current GPU databases still cannot support concurrent query executions efficiently.

[2]Latest GPU systems support kernel launches from the device, which eases the overhead of CPU phases in some cases.

2.3 Problems with Uncoordinated Query Co-Running

Running multiple queries on the same GPUs can improve resource utilization and system performance. However, as we demonstrate next, these benefits do not come gratuitously. Due to the lack of necessary database facilities to coordinate the sharing of GPU resources, co-running queries naively can cause serious problems such as query abortions or mediocre throughput.

One of the most important functionalities not supported in current database systems is the coordination over GPU device memory usage. To maximize performance, each query tends to allocate a large amount of device memory space and keep its data on the device for efficient reuses. This causes high conflicts when multiple queries try to use device memory simultaneously. Since the underlying GPU driver does not support automatic data swapping, query co-runnings, if not managed by the database, can easily abort or suffer low performance. Even though there are recent proposals to suggest adding such service in the operating system [28], the database engine still needs to provide this functionality on its own in order to take advantage of additional information from query-level semantics for maximizing performance.

To show this demand, we measure how the system performs when co-running SSB queries used in the previous subsection on YDB. For all 69 combinations whose peak device memory consumption exceeds the device capacity (i.e., suffering contention), device memory allocation failures are observed for one or both of the participating queries. Because of high device memory conflicts, some query pairs cannot finish executions successfully every time they are co-ran together. Some others suffer failures sporadically, depending on whether their co-runnings happen to trigger the conflicts. To verify the commonness of the problem, we have also performed similar experiments with Ocelot running TPC-H benchmarks on an AMD GPU. Ocelot supports device memory swapping within a single query, but provides no mechanisms to handle device memory conflicts caused by concurrent queries. We observe similar experiment results — all query co-runnings suffering device memory conflicts cannot finish executions successfully with Ocelot. The underlying GPU drivers used in the YDB and Ocelot experiments are the latest commercial CUDA and OpenCL drivers from NVIDIA and AMD respectively. The problem shown by our experiments is thus general, which exists with both major GPU computing platforms.

Besides device memory swapping, another critical facility missing in current GPU databases is query scheduling. Due to the limited capacity of GPU resources and the diverse demands of user queries, system performance is sensitive to the number of queries co-running on the GPU. Running too many queries can lead to severe resource contention that may cause high overhead. Running too few queries, on the other hand, underutilizes resources and loses the opportunity to maximize system performance. Query scheduling maintains optimal workload on the GPU by controlling the combinations of queries that are allowed to execute concurrently, and thus plays an important role to system throughput.

To demonstrate the necessity of query scheduling, we measure the performance of a system we have developed to support concurrent query executions (see Section 6 for details), running SSB queries at scale factor 14. Without enabling

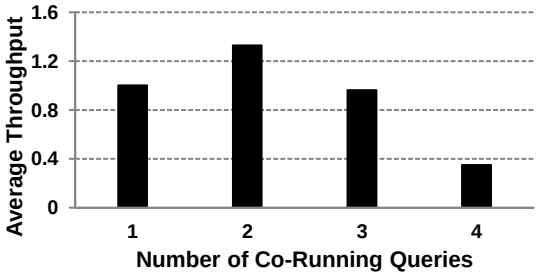


Figure 3: The impact of query scheduling on system throughput.

the query scheduling functionality, we change the number of queries executed concurrently by our system and show the average system throughput achieved under each setting in Figure 3. It can be seen that system throughput improves from running one query at a time to running queries pairwisely, but degrades quickly as the number of co-running queries exceeds two. Noticeably, when four queries are allowed to execute simultaneously, system throughput drops to only 1/4 of the optimal value, which is even 65% lower than running queries one by one. This result shows that a database system without proper query scheduling functionality can easily suffer low system throughput or high system thrashing, which severely undermines the benefits of concurrent query processing.

3. MultiQx-GPU: AN OVERVIEW

To support concurrent query processing, MultiQx-GPU provides the functionalities needed by databases to coordinate GPU resource sharing. In this section we highlight the design principles and overall structure of the system.

The design of MultiQx-GPU abides by two main principles. The first one is ***versatility*** — the techniques presented by the system should be applicable to different GPU databases and computing frameworks for managing GPU resources. GPU database technologies are still evolving very quickly. Different systems have different query engine implementations and may be based on different GPU computing frameworks. The methods employed by MultiQx-GPU therefore should be easily utilized in all these variations. This requires the design of MultiQx-GPU to capture the essential properties of GPU query processing, to build upon the common abstractions of GPU frameworks, and to integrate with existing and future GPU database engines in a non-intrusive manner.

The second principle followed by MultiQx-GPU is ***high efficiency***. Originally designed for gaming and super computing applications, GPU hardware and system software still do not have native support for multitasking. Basic system-level functionalities familiar to the CPU world, such as virtual memory (VM) and fine-grained context switches, are not provided by commercial GPU drivers. This forces MultiQx-GPU to add an extra layer of application-level software to support multi-query capabilities on its own, which, if not taken great care of, could incur high overhead.

Figure 4 shows the position of MultiQx-GPU in the overall GPU database software stack. MultiQx-GPU is built into the database query engine, but remains loosely coupled with existing components in the query engine. It enforces controls over GPU resource usage by transparently intercepting the GPU API calls from user queries. This design

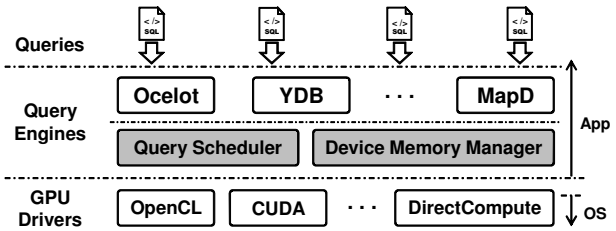


Figure 4: Overview of the MultiQx-GPU solution. Shaded boxes denote the two new components provided by MultiQx-GPU to manage GPU resources.

does not change existing programming interfaces of the underlying GPU drivers, and minimizes the modifications to the other components of the GPU query engine. MultiQx-GPU resides completely in the application space, and does not rely on any OS-level functionalities privileging to the GPU drivers. It can thus be easily ported between different query engine systems (such as Ocelot, YDB, and MapD [17]) and GPU computing frameworks (such as CUDA, OpenCL, and DirectCompute [19]) to enable GPU resource sharing.

MultiQx-GPU comprises two main components providing the support required for concurrent query executions. Working like an admission controller, the ***query scheduler*** component controls the concurrency level and intensity of resource contention on GPU devices. By controlling the queries that can execute concurrently at the first place, query scheduler maintains optimal workload on the GPUs that would maximize system throughput. Once a proper concurrency level is maintained, the ***device memory manager*** component further ensures system performance by resolving the resource conflicts among concurrent queries. Through VM-like automatic data swapping service, it makes sure that multiple queries with moderate resource conflicts can make concurrent progress efficiently without suffering query abortions or causing low resource utilization.

In the next two sections we will elaborate the detailed designs of these two components and explain various decisions made by MultiQx-GPU to minimize overhead. Since the design of query scheduler assumes the capability of the query engine to efficiently resolve resource contention, we first introduce the design of the device memory manager to achieve this basic functionality in the following section.

4. DEVICE MEMORY MANAGER

The primary functionality of the device memory manager is to coordinate the conflicting demands for device memory space from different queries so that they can make concurrent progress efficiently. To achieve this goal, it relies on an optimized data swapping framework and replacement policy to minimize overhead.

4.1 Framework

When free device memory space becomes insufficient, instead of rejecting a query's service request, MultiQx-GPU tries to swap some data out from device memory and reclaim their space for better uses. This improves the utilization of device memory space and makes concurrent executions more efficient. To achieve this purpose, the device memory manager employs a data swapping framework that is motivated by a system called GDM [28]. Different from GDM, our framework resides in the application space, which cannot

rely on any system-level interfaces, but has the advantage of using query-level semantics, for data swapping.

To support data swapping, the framework maintains a swapping buffer in the host memory to contain the query data that need not to reside in the device memory momentarily. When a device memory allocation request is received, it creates a virtual memory area in the swapping buffer and returns the address of the virtual memory area to the query. Device memory space only needs to be allocated when a kernel accessing the data is to be launched. The framework maintains a global list of data regions allocated on the device memory for all running queries. When free space becomes insufficient, the device memory manager selects some swappable regions from the list and evicts them to the swapping buffer. Due to the special features of multi-query workloads, several optimization techniques are employed by the framework to improve performance, as explained below.

Lazy transferring. When a query wants to copy some data to a device memory region (e.g., through *cudaMemcpy* in CUDA), the data are not immediately transferred to device memory until they are to be accessed in a GPU kernel. The swapping buffer serves as the temporary storage for the data to be transferred. This design prevents data from being evicted from device memory immaturely because data only need to be transferred to device memory when they are to be immediately accessed. To further reduce overhead, the memory manager marks the query source buffer copy-on-write. The data can later be transferred directly from the source buffer if it has not been changed.

Page-based coherence management. GPU queries usually reserve device memory space in large regions. The memory manager internally partitions a large region into several small, fixed-size, logical pages. Each page keeps its own state and maintains data coherence between host and device memories independently. Managing data coherence at page units has at least two performance benefits. First, by breaking a large, non-interruptible DMA operation into multiple smaller ones, data evictions can be canceled immediately when they become unnecessary (e.g., when a region is being released). Second, a partial update to a region only changes the states of affected pages, instead of a whole region, which reduces the amount of data that need to be synchronized between host and device memories.

Data reference and access advices. To avoid allocating device memory space for unused regions, the memory manager needs to know which data regions are to be referenced during a kernel execution. It is also beneficial for the memory manager to know how the referenced regions are to be accessed by a kernel. In this way, for example, the content of a region not containing kernel input data needs not to be loaded into device memory before the kernel is issued; the memory manager also needs not to preserve region content into the swapping buffer during its eviction if the data are not to be reused. To achieve this purpose, the memory manager provides interfaces for queries to pass data reference and access advices before each kernel invocation.

4.2 Data Replacement

When free device memory space becomes scarce, the memory manager has to reclaim some space for the kernel to be launched. The replacement policy that selects data regions for evictions plays an important role to system performance.

There are three main differences between data replacement in device memory and conventional CPU buffer pool management. First, the target of device memory replacement is a small number of variable-size regions rather than a large amount of uniform-size pages. GPU queries usually allocate a few device memory regions, whose sizes may differ dramatically depending on the roles of the regions and query properties. Since the physical device memory space allocated for a region cannot be partially deallocated without necessary driver support, a victim region, once selected, must be evicted from device memory completely. Second, unlike CPU databases where data evictions can be interleaved with data computation to hide the latency of replacement, a GPU kernel cannot start execution until sufficient space is vacated on the device memory for all its data sets. This makes GPU query performance especially sensitive to the *latency* of data replacement. Third, device memory not only has to contain input table data and output query results, but also stores various intermediate kernel objects whose content can be modified from both CPU and GPU. This makes the data access patterns of device memory regions much more diverse than buffer pool pages.

Based on these unique characteristics, we propose a policy, called ***CDR*** (Cost-Driven Replacement), that combines the effects of region size, eviction latency, and data locality to achieve good performance. When a replacement decision has to be made, CDR scans the list of swappable regions, and selects the region that would incur the lowest *cost* for eviction. The cost c of a region is defined in a simple formula,

$$c = e + f \times s \times l, \tag{1}$$

where e represents the size of the data that needs to be evicted from device memory, s is region size, l represents the position of the region in the LRU list, and f is a constant, which we call *latency factor*, whose value is between 0 and 1. If two regions happen to have the same cost value, CDR breaks the tie by selecting the less recently used one for replacement.

The first part of Formula 1, e, quantifies the latency of space vacation. Its value depends on the status of the data pages in a region. For example, e is zero if the none of the pages has been modified by kernels on the device memory. If some pages have been updated by the query process from the CPU, the device memory copies of those modified pages would have been invalidated and thus should not be evicted back to the swapping buffer, leading to a value of e less than s. The second part of Formula 1, $f \times s \times l$, depicts the potential overhead if the evicted region would be reused in a future kernel. The value of l is between $1/n$ and 1, depending on the region's position among the n swappable regions in the LRU order. For example, $l = 1/n$ for the least recently used region, $l = 2/n$ for the second least recently used one, and so on. The role of latency factor f is to give a heavier weight to data eviction latency in the overall cost formula.

As will be shown in Section 7.4, CDR delivers higher performance than conventional replacement policies in supporting concurrent query executions, thanks to its capability to identify suitable victim regions that incur low overhead.

5. QUERY SCHEDULER

In an open system where user queries arrive and leave dynamically, the query scheduler maintains optimal workload

on the GPUs by controlling which queries can co-run simultaneously. A query is allowed to start execution if it can make effective use of the underutilized or unused GPU resources without incurring high overhead associated with resource contention. The GPU workload status is monitored continuously, so that delayed queries can be rescheduled as soon as enough resources become available.

A critical issue in query scheduling is to estimate the actual resource demand of a GPU query. As explained in Section 2.2, the amount of resource being effectively utilized by a query can be much lower than its reservation. Scheduling queries based on the maximal reservation can thus cause GPUs to be under-loaded, leading to suboptimal system throughput. In GPU databases, different queries and query phases may have diverse resource consumption, depending on query and data properties such as filter conditions, table sizes, data types, and content distributions. If the query scheduler cannot accurately predict the actual resource demand, a mistakenly scheduled query can easily bring down the overall system performance by large, as has been shown in Section 2.3.

To address the problem, we propose a simple, practical metric to effectively quantify the resource demand of a GPU query. The design of the metric is based on some observations that are generally applicable to analytical GPU databases. First, for GPU query processing, the utilization of device memory space has the principal impact on system throughput and can be frequently saturated under multi-query workloads. Unlike compute cycles and DMA bandwidth that can be freely reused, reusing a device memory region requires data evictions and space re-allocation, which can potentially incur high overhead. This makes system performance strongly correlated with the demand for and utilization of device memory space. Second, to ensure data transfer and kernel execution efficiencies, analytical GPU engines usually employ a *batch-oriented*, *operator-based* query execution scheme. Under this scheme, table data are partitioned into large chunks and pushed from one operator to another for processing. It is thus a good model to consider query execution as a temporal sequence of operators, each of which accepts an input data chunk, processes it with the GPU, and generates an output data chunk that may be passed to the next operator for further processing.

Based on the above observations, we define a metric called *weighted device memory demand*, or briefly *weighted demand*, which is the weighted average of the device memory space consumed by a query's operator sequence. The weight is computed as the percentage of query execution time spent in each operator. The device memory space consumed by an operator equals the maximal total size of device memory regions referenced by any GPU kernel in the operator. Suppose that each operator's execution time and device memory consumption are t_i and m_i respectively, the weighted demand m of the query can be computed by

$$m = \frac{\sum (t_i \times m_i)}{\sum t_i}. \quad (2)$$

The device memory consumption of an operator can be computed from query predicates and table data statistics. The execution time of an operator can be predicted through modeling, as has been shown in our previous work [32].

To accommodate the changes of resource consumption in different query phases, the weighted demand of a query is dynamically updated as the query executes, and is exposed to the query scheduler for making timely scheduling decisions. When a new query arrives, the query scheduler computes its initial weighted demand. If the number exceeds the available device memory capacity, which is measured by the difference between the device memory capacity and the sum of weighted demands of scheduled queries, the query's execution needs to be delayed. The query scheduler considers rescheduling a postponed query every time when a running query's resource demand changes or when a query finishes execution.

6. IMPLEMENTATION

We have implemented a prototype of MultiQx-GPU above the CUDA computing framework and integrated it with YDB to support concurrent query processing. This section summarizes our experiences in building the system.

Our system adds two new components to the original YDB software stack. The memory manager component is implemented in a highly modulated shared library (5200+ lines of C code). It is dynamically linked with the query binary to intercept CUDA API calls (through the *LD_PRELOAD* dynamic linking option on Linux). The query scheduler component is an add-on Python module that wraps around a query binary to control its execution. Most existing YDB components remain unmodified. A small amount of code (120+ lines) is added to the GPU operator library to provide data reference and access advices to the memory manager. The algorithm designs and kernel implementations of all GPU operators are unchanged.

A new programming interface, *cudaAdvice*(*addr*, *flags*), is exported by the memory manager component to receive data advices. In the interface, *addr* denotes the address of the data region to be accessed in the next kernel execution, and *flags* is the advice about how this region is to be accessed, which can be *input*, *output*, or *both*.

Our prototype system employs a process-based query execution model: each user query executes in a separate operating system process. To share information such as region states and resource availability, a shared memory area is created in the host memory. Data replacement requests and responses are communicated between different query processes through POSIX message queues. Copy-on-write is implemented through the *mprotect* system call on Linux. Since *mprotect* does not capture the event when a memory region is freed, we have to additionally override the *free()* function in libc. We set page size to 8MB, which provides the fine granularity required for data coherence management, meanwhile retaining over 99% of PCIe efficiency. The value of latency factor f is set to 0.01, which works empirically well in practice.

The implementation of our prototype system addresses several technical issues which are discussed in the rest of the section. These issues are mainly caused by the undesired behaviors or missing services in the underlying GPU driver.

False synchronizations. GPU kernels execute asynchronously with respect to the host query process. To avoid data races and kernel failures, the GPU driver usually enforces implicit synchronizations when handling some important operations such as data transfers and memory releases [3]. For example, the CUDA driver forcibly inserts a global barrier before a device memory region is released

to ensure that no GPU operations are still accessing the region when its address mapping is removed from the GPU page table. Such implicit synchronizations may be helpful to some other GPU applications. In concurrent databases, however, since the query engine has full knowledge about the data access behaviors of its kernels and the data dependencies among different GPU operations, the extra synchronizations performed by the GPU driver are often unnecessary and can delay the progress of user queries. Our system circumvents this problem with two main mechanisms. First, it creates a dedicated CUDA stream (or two streams, each for one direction, if the GPU card has dual copy engines) for DMA transfers. To prevent data transfer requests from being unnecessarily blocked, our system internally maintains a set of small, pinned buffers, and automatically converts all data transfers into asynchronous DMAs by pipelining data through the pinned buffers in the dedicated stream. Second, our system uses a dedicated garbage collection thread to handle device memory releases, so that the main query thread is never blocked by such operations. This daemon thread also handles the case when a region still being evicted needs to be released.

Kernel event handling. When a kernel finishes execution, the states of the regions accessed by the kernel have to be timely updated to ensure system correctness and performance. This can be done by letting the GPU driver execute a segment of maintenance code on CPU whenever a kernel finishes execution (e.g., through the *cudaStreamCallback* interface in CUDA). However, this can degrade system performance because the GPU hardware command queue is suspended when the CPU service thread waits to be scheduled by the operating system and executes the maintenance code. To avoid such overhead, our system inserts a short GPU code segment after each kernel. This code has the sole functionality of updating the states of the regions accessed by a kernel when it finishes. By pinning the state fields of data regions in the host memory and mapping them to the device address space, region states can be updated efficiently with a few direct host memory accesses without any interventions from the CPU.

Device memory fragmentation. Data replacement requires frequent allocations and deallocations of device memory space. We find that the device memory space may become slightly fragmented in some cases after the system runs for a long time. When this happens, a device memory allocation operation may fail even if there seems to be sufficient free space on the device memory. This problem is due to the implementation of the physical device memory allocator in the GPU device driver, but we are not able to confirm the exact causes due to lack of public documentations about the commercial CUDA driver. Our system uses a two-round memory allocation procedure to address the problem without relying on the underlying driver details: if the first allocation attempt fails, the device memory manager evicts some data region, no matter whether the free space still seems enough or not, and retries the allocation. Since the release of free space often forces the device driver to re-organize its free memory list, this mechanism effectively addresses the influence of fragmentation in practice. Because fragmentation happens rarely, the extra space eviction caused by this approach also does little harm to system performance.

7. EXPERIMENTS

This section evaluates the performance of MultiQx-GPU thoroughly and verifies the effectiveness of various design decisions in supporting concurrent query executions. Before presenting the results, we first introduce the settings and methodology of our evaluation.

7.1 Settings and Metrics

The experiments are conducted on a workstation equipped with a four-core 3.4GHz Intel Core i7-2600 CPU, 16GB system memory, and an NVIDIA GTX 580 GPU installed in a PCIe 2.0 x16 slot. The maximum device memory capacity is 1.6GB, among which about 1.46GB of space is available for executing database queries. The GPU operator library and the device memory manager component of MultiQx-GPU are compiled and executed with NVIDIA CUDA Toolkit 5.0. The operating system is Red Hat Enterprise Linux Workstation 6.5 with 2.6.32 kernel.

Our experiments mainly use the queries and data sets from the Star Schema Benchmark [21], which is widely used in database research due to its realistic modeling of data warehousing workloads. The table data are generated using the standard benchmark tool and converted into the column format required by the YDB query engine. The scale factor is set to 14 by default, unless otherwise noted, which populates the fact table with about 80 million tuples (6.6GB in total size). The query executables are pre-generated to exclude query parsing, optimization, code generation, and compilation times from performance measurement. To minimize the influence of disk accesses, we load all data sets into host memory before starting each experiment. Figure 5 lists the device memory usage of each query when it executes alone with MultiQx-GPU. The diversity of device memory behaviors makes our workloads more representative.

Several metrics are used in our experiments to characterize system performance from different perspectives. We use *weighted speedup* to measure the throughput of multi-query executions in a closed system, where the co-running queries are fixed. It is defined as the sum of the speedups of participating queries [27, 18]. Suppose n queries execute concurrently, the throughput of their executions is computed as $\sum_{i=1}^{n} \frac{s_i}{c_i}$, in which s_i is the execution time of the ith query when it runs alone and c_i denotes its execution time when it co-runs with other queries. Under this definition, the throughput of running queries sequentially (i.e., without queries executing concurrently) is 1. Since queries have different execution times, we run each query multiple times to ensure its full overlap with other queries. In an open system

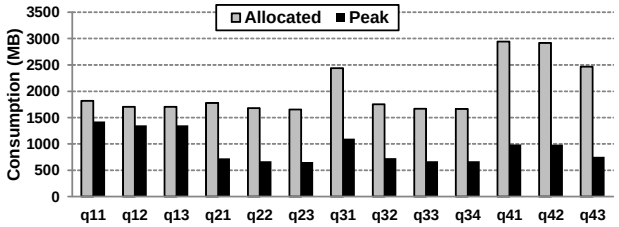


Figure 5: Device memory usage of SSB queries at scale factor 14. q_{xy} denotes the yth query of the xth query flight. The *Allocated* bars show the total device memory space allocated in each query. The *Peak* bars show the maximum device memory space held by each query during its execution.

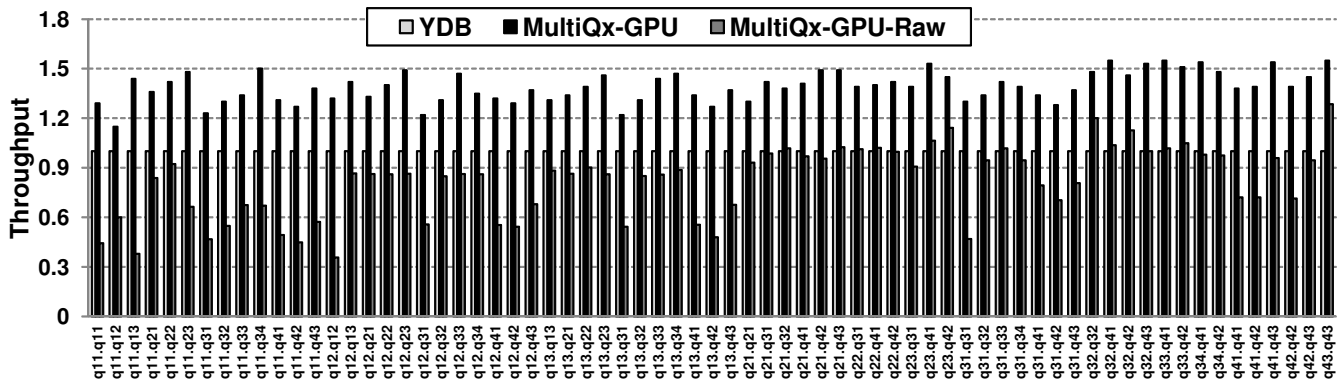


Figure 6: Throughput of pairwise SSB query co-runnings with three different systems. MultiQx-GPU-Raw is a variant of MultiQx-GPU without optimizations. $p.q$ denotes the combination of query p and q.

where user queries arrive and leave dynamically, we measure system performance with the metric of *queries per second*, which is defined as the number of queries processed divided by the total processing time.

To gain insights into the performance numbers observed, we also measure the utilization of GPU resources such as the copy and compute units during workload execution. To validate the effectiveness of various resource management designs to reduce unnecessary data movement, we use another metric, called *DMA efficiency*, that measures the percentage of DMA time used for *effective* data transfers. A data transfer is effective if it is required when the query executes alone. Suppose a instances of query A co-runs with b instances of query B in exactly full overlap, the time spent on DMA data transfers is x for A and y for B when each of them executes alone, and the total DMA time is t during their co-runnings, then the DMA efficiency during the co-running of A and B is $(ax + by)/t$.

In the following text, we first present the overall performance of MultiQx-GPU in executing concurrent queries, and then verify the effectiveness of its various components.

7.2 Performance of Concurrent Executions

Through coordinated sharing of GPU resources, MultiQx-GPU improves system throughput by letting multiple queries make efficient progress concurrently. In this subsection we evaluate the overall performance of MultiQx-GPU in supporting concurrent executions. The evaluation is performed by co-running SSB queries pairwisely. Among 91 possible query combinations, we select the 69 pairs of co-runnings whose peak device memory consumption exceeds device memory capacity (i.e., suffering conflicts). We measure their throughput achieved with MultiQx-GPU, and compare them with the original YDB system. The first two bars of each group in Figure 6 show the results.

It can be seen that, by processing multiple queries at the same time, MultiQx-GPU greatly enhances system performance compared with dedicated query executions. The throughput is consistently improved across all 69 co-runnings, by an average of 39% (at least 15%) as compared with YDB. For the co-runnings of *q32* with *q41*, *q33* with *q41*, and *q43* with itself, the improvements are more than 55%. The high performance leap achieved by MultiQx-GPU is mainly attributed to the better utilization of GPU resources. Under the efficient management of MultiQx-GPU, the DMA bandwidth and GPU computing cycles unused by one query can be allocated to serve the resource requirements from other queries. The efficient utilization of resources improves overall system throughput.

To validate the reasons for performance improvement, we have measured the utilization of GPU's compute and copy units during the execution of each query pair on both YDB and MultiQx-GPU. In Figure 7, the left bar graph shows the geometric means of GPU resource utilization averaged across all query combinations on the two platforms. The right bar graph depicts the improvement of resource utilization per query pair achieved on MultiQx-GPU versus on YDB, with both the geometric mean and max-min values shown for each resource type. It can be seen that MultiQx-GPU significantly improves the utilization of both DMA and GPU compute resources over that with YDB. On average, the DMA engine becomes 62% more occupied (from 35% on YDB to 57% on MultiQx-GPU) after concurrent query execution is enabled, while the utilization of GPU cores is improved from 13% to 18% (by 33%) accordingly. The per-query-combination improvements of resource utilization also reflect this trend, with the utilization of DMA and compute engines consistently raised by 61% and 31% on average respectively. Through manual inspections, we have also found close correlations between the degrees of resource utilization increasing and throughput improvement for different query combinations. The diversity of resource utilization improvement shown in the right bar graph of Figure 7 thus explains the difference of throughput enhancement in Figure 6.

Figure 7 also shows the potential to further improve multi-query performance on GPUs. Currently, despite the great improvement of resource utilization with MultiQx-GPU, GPU's DMA and compute units are still not 100% utilized. This is mainly due to the hardware limitation of GPUs and the overhead of data swapping. On one hand, GPU is an exclusive, non-preemptive device: the kernels and DMA com-

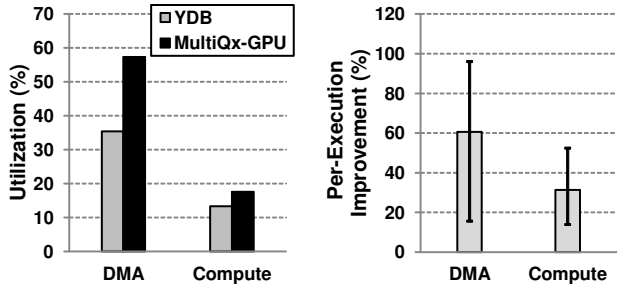


Figure 7: Improvement of GPU resource utilization with MultiQx-GPU relative to YDB.

mands inserted into the hardware command queue are usually executed on the GPU one after another. This serializes query executions at the hardware level and greatly limits the concurrency level of the whole system. The data swapping activities performed by MultiQx-GPU, on the other hand, also limits performance enhancement. Even if the optimization techniques presented in this paper, as will be verified in the next section, are helpful at reducing the overhead, there are still some penalties that cannot be totally avoided.

To measure the performance of MultiQx-GPU processing larger data sets, we have repeated the above experiment at scale factor 28, which corresponds to over 13GB of table data (each query processing about 3GB of data on average, much larger than the device memory capacity). Across all 69 query co-runnings, MultiQx-GPU improves throughput consistently by an average of 33% (up to 54%) above YDB, which is comparable to the performance improvement at scale factor 14 (39% on average, 55% at maximum). This result is well expected because the execution behaviors of GPU operators remain unchanged. When the sizes of intermediate results cannot fit into device memory, the YDB query generator partitions data sets into smaller chunks and generates a query binary that processes them separately. This only increases the number of times each operator is executed, not query behavior.

7.3 Validations of Optimizations

MultiQx-GPU's data swapping framework employs a set of optimization techniques, including lazy transferring, page-based coherence maintenance, and data reference and access advices, to minimize overhead. To validate the effectiveness of these techniques in ensuring system performance, we repeat the experiment performed in the previous subsection on a degraded version of MultiQx-GPU, denoted as *MultiQx-GPU-Raw*, that does not have these optimizations enabled. MultiQx-GPU-Raw transfers data to device memory eagerly, maintains data coherence in units of whole data regions, and does not exploit advices from user queries to assist data swapping. We measure the throughput of query co-runnings achieved with MultiQx-GPU-Raw, and show the results with the third bar of each group in Figure 6.

It can be seen that the optimization techniques have high impact on system performance. Compared with the optimized MultiQx-GPU, MultiQx-GPU-Raw achieves much lower throughput for all query co-runnings. With an average slowdown of 40%, the largest performance loss reaches over 73% due to the high memory conflict between *q11* and *q13*. As a matter of fact, MultiQx-GPU-Raw performs even lower than YDB during most co-runnings; it brings down the throughput of 55 (among 69) query combinations by an average of 16%, up to 64%, compared with YDB. The lack of optimization techniques greatly undermines the usefulness of the MultiQx-GPU system to support concurrent query processing, causing performance degradation, instead of enhancement, relative to dedicated query executions.

The major reason for the low performance of MultiQx-GPU-Raw is the excessive data movement overhead caused by the unoptimized data swapping framework. This is explained in Figure 8. We measure the average DMA efficiencies achieved with MultiQx-GPU-Raw and MultiQx-GPU when executing the workloads, as shown in the left bar graph. It can be seen that, without the optimizations, the efficiency of the DMA engine is only 38%, which means

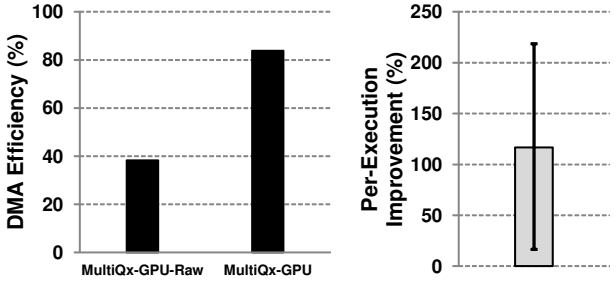


Figure 8: Improvement of DMA efficiency with the help of swapping framework optimizations.

that over 60% of the DMA time is spent on data transferring that would not be necessary during normal query executions. With the optimizations, the DMA engine works much more efficiently, raising DMA efficiency by over 118% (reaching 84%). This significant reduction of unnecessary data transfers directly translates to the high performance gap between MultiQx-GPU-Raw and MultiQx-GPU shown in Figure 6. The right bar graph sheds lights on the same result from the perspective of per-execution DMA efficiency improvement. The efficiency of the DMA engine during the execution of each query pair improves by 117% on average from MultiQx-GPU-Raw to MultiQx-GPU, with the minimum and maximum changes being 17% and 219%.

To further evaluate the effectiveness of each individual optimization technique, we have implemented several modified variants of MultiQx-GPU. Each variant turns off the optimization technique being tested, while keeping all other optimizations enabled. We measure the throughput of the same workload used above under these variants. To save space, we categorize SSB queries into four groups based on their query flight number, and only show the average performance of query co-runnings that belong to different group combinations. For example, *q1.q2* denotes the average throughput for co-running one query from query flight *q1* with another query from query flight *q2*.

The result is depicted in Figure 9. *NoLazy* represents a variant of MultiQx-GPU without lazy transferring; data are transferred to device memory as soon as the request is received. *NoCow* transfers data to device lazily, but does not have the copy-on-write optimization for the data copied to swapping buffer. *NoRef* denotes the variant without data reference advice; the memory manager assumes that a kernel references all the data regions resident in a GPU context. *NoAccess* enables reference advice, but has no data access advice; every region referenced by a kernel is assumed to contain both data input and data output. Finally, *NoPage* does not maintain data coherence in page units.

It can be seen that missing any optimization technique

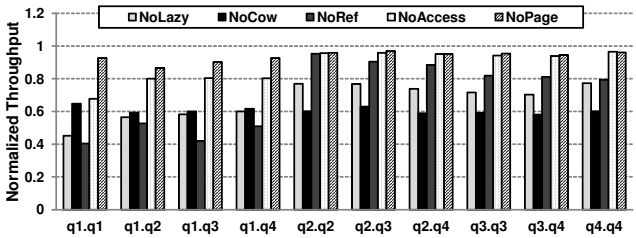


Figure 9: The influence of each individual optimization technique on system performance. Performance is normalized to the fully optimized MultiQx-GPU.

would degrade system performance consistently, by varying degrees under different workloads. For example, removing data reference advice alone lowers MultiQx-GPU performance by an average of 22% (60% at maximum for workloads in the *q1.q1* group). The effect of data reference advice seems much more significant to workloads involving queries in the *q1* series (47% to 60% slowdown) than to others (5% to 20%). This is because *q1* queries have the highest device memory consumption (as can be seen from Figure 5), for which data reference advice would be most effective at reducing unnecessary device memory contentions. The influence of page-level coherence management may seem moderate on average (5.7%), but still can be significant (14%) for the *q1.q2* workloads. This result shows the indispensability of every optimization technique. It is when they work together that MultiQx-GPU achieves its highest performance.

7.4 Experiments with Replacement Policies

By controlling the selection of proper victim regions to evict under resource contentions, data replacement policy plays an important role to system throughput. This subsection presents the results of our experiment with several data replacement policies to support multi-query executions and verifies the effectiveness of CDR in improving MultiQx-GPU performance. We compare the performance of five replacement policies, LRU (Least Recently Used), MRU (Most Recently Used), LFU (Least Frequently Used), RANDOM, and CDR. The first four policies are selected because they are widely used in conventional multitasking and data management systems. We measure the throughput of the same workloads used in the previous two subsections, achieved using MultiQx-GPU (all optimizations are enabled) with different replacement policies. Due to space constraint, we randomly select 6 queries and only present the results for their co-runnings, but similar observations can be made with other queries as well.

As shown in Figure 10, there are no significant differences among the performance of LRU, MRU, LFU, and RANDOM; they perform unevenly, but closely match each other under different workloads. CDR, however, performs much better than other policies across all query co-runnings, consistently improving system throughput by 44% on average (56% at maximum) compared with LRU. The performance advantage of CDR compared with other policies is expected, due to its careful design to select victim regions that minimize space eviction and data swapping costs. On the contrary, the other four policies do not consider the unique features of GPU queries and their concurrent executions. The criteria they use to make replacement decisions are rather random in terms of the benefits to overall system performance, often leading to increased kernel launch latency and

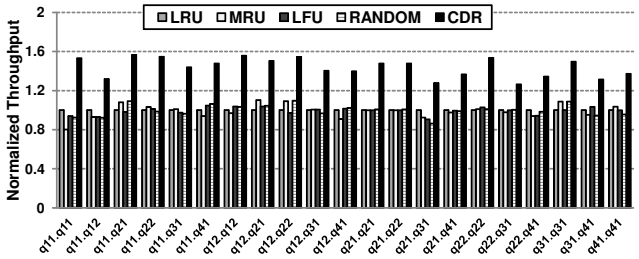
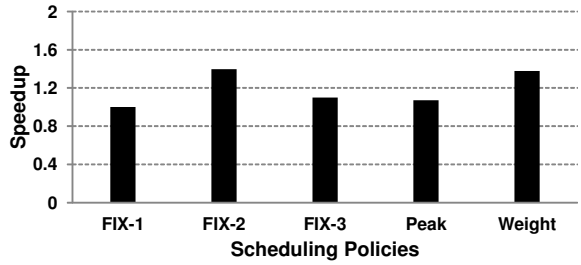
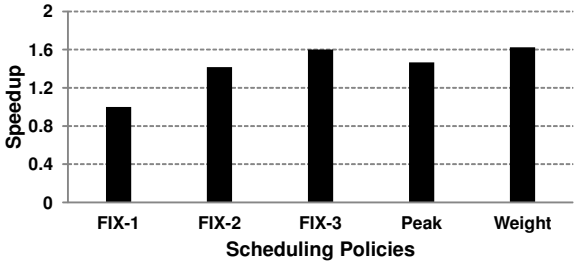


Figure 10: Performance of co-running selected SSB queries under various data replacement policies. Throughput is normalized against LRU.

unnecessary data swapping.



(a) System performance at scale factor 14.



(b) System performance at scale factor 8.

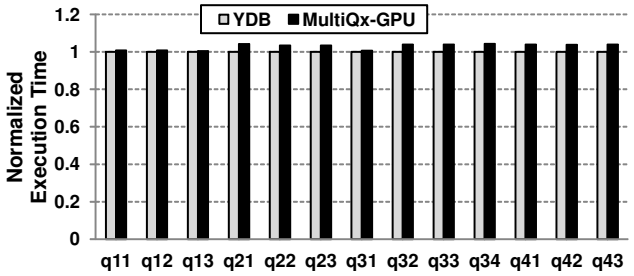
Figure 11: System speedups achieved with different scheduling policies, normalized to FIX-1.

7.5 Effectiveness of Query Scheduling

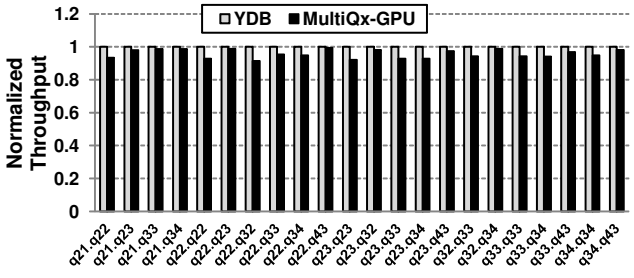
To verify the effectiveness of query scheduling in ensuring system performance, we use a random-load generator to generate a sequence of 100 query requests, which model user queries issued dynamically in an open system. The arrivals of query requests follow Poisson process, with the arrival rate set to 4 queries per second. The query issued at each interval is randomly picked from a pre-generated query set. Applying the same query request trace, we compare the system performance (in terms of queries per second) achieved by MultiQx-GPU under five query scheduling policies: FIX-1, FIX-2, FIX-3, Peak, and Weight. FIX-n denotes the policy that fixes the number of concurrently executing queries to n. For example, FIX-1 corresponds to the policy that executes one query at a time (without concurrent executions). Peak is a scheduling policy that schedules queries based on their peak device memory demands. Weight represents our scheduling policy proposed in Section 5.

Figure 11(a) shows the speedup of each scheduling policy relative to FIX-1 when the scale factor is 14. It can be seen that the speedups achieved by FIX-3 (1.1) and Peak (1.07) are much lower than those delivered by FIX-2 (1.39) and Weight (1.37). At scale factor 14, SSB queries have high device memory demands. Co-running 3 queries together causes too much resource conflict that lowers system performance. Peak, on the other hand, is too conservative at selecting queries to co-run, and thus loses the opportunities to improve system performance by executing more queries concurrently. The performance of FIX-2 is comparable with Weight, because the concurrency level it supports (2) matches this workload. But, as we will show next, it cannot maintain its peak performance under other settings.

To demonstrate how each scheduling policy performs under workloads with less-intensive resource conflicts, we repeat the above experiment with data sets at scale factor 8. The speedup of each policy compared with FIX-1 is shown in Figure 11(b). It can be seen that FIX-1 still performs the



(a) Execution times of SSB queries when they execute alone on YDB and MultiQx-GPU, normalized against YDB.



(b) Throughput of query co-runnings that have no device memory conflicts, normalized against YDB.

Figure 12: Overhead of MultiQx-GPU.

worst among all policies, while Weight continues to deliver the highest speedup (over 62% higher than FIX-1). FIX-2, which performs equally with Weight under scale factor 14, can no longer match up with Weight at scale factor 8 because it does not consider the changing resource demands of user queries. This experiment, combined with the previous one, verifies the effectiveness of the query scheduler to ensure system performance at various system settings.

Figure 11 also shows the benefits of concurrent query executions for speeding up dynamic query workloads. Even though FIX-2, FIX-3, and Peak cannot consistently deliver the highest system performance as Weight does, they all outperform FIX-1, often by large margins.

7.6 Overhead

To efficiently manage GPU resources, MultiQx-GPU creates a swapping buffer in the host memory and performs various maintenance actions such as setting copy-on-write protections, updating the states of data blocks, and scheduling kernels. These management activities may add some overhead to query executions when there are not resource conflicts. In this subsection, we evaluate this overhead and show that it is sufficiently low in practice.

We measure the overhead by comparing the performance of MultiQx-GPU (all resource management functionalities enabled) with YDB at scale factor 14 under two groups of workloads. The first group consists of the solo executions of the 13 SSB queries; the second group comprises the 22 pairs of query co-runnings that do not suffer device memory conflicts. The results are presented in Figure 12. It can be seen that, for single-query executions, the performance of SSB queries achieved with MultiQx-GPU closely matches their performance with YDB, which does not have GPU resource management overhead. MultiQx-GPU increases execution times by at most 4.3% compared with YDB, with the average slowdown being 2.2% across all queries. For query co-runnings, on the other hand, the average throughput achieved with MultiQx-GPU is only 3.4% lower than that with YDB. We believe that the overhead of MultiQx-GPU is negligibly low, especially considering the performance benefits it provides through concurrent query executions.

8. RELATED WORK

The use of GPUs for database applications has been intensively studied in existing research. Some works focus on the designs and implementations of efficient GPU algorithms for common database operations such as join [8, 23, 12], selection [7], sorting [25, 6], and spatial computation [29, 17, 4], achieving orders of magnitude of performance improvement over conventional CPU-based solutions. Other works exploit various software optimization techniques to accelerate query plan generations [10], improve kernel execution efficiency [30, 26], reduce PCIe data transferring [30, 31], and support query co-processing with both GPUs and CPUs [22].

Our work in this paper is mainly related to recent development efforts of GPU query engines, which provide infrastructure software support for the integration of GPUs in real-world database systems. YDB [32], based on which MultiQx-GPU is implemented, is designed for data warehousing query processing. It employs a column-based storage format, and generates query plans that execute in a push-based, batch-oriented fashion. Ocelot [11] is a hybrid OLAP query processor as an extension for MonetDB. By adopting a hardware-independent query engine design, it supports efficient executions of OLAP queries on both CPUs and GPUs. Ocelot provides a memory management interface that abstracts away the details of the underlying memory structure to support portability. The memory manager can also perform simple data swapping within a single query. However, as we mentioned in Section 2, it does not have sufficient mechanisms or policies to support correct, efficient executions of queries in concurrent settings. MapD [17] is a spatial database system using GPUs as the core query processing devices. Through techniques such as optimized spatial algorithm implementations, kernel fusing, and data buffering, MapD outperforms existing CPU spatial data processing systems by large margins. GPUTx [9] is a high-performance transactional GPU database engine. It batches multiple transactional queries into the same kernel for efficient executions on the GPU and ensures isolation and consistency under concurrent updates. The workloads GPUTx targets are short-running, small tasks that would not cause device memory contention. The techniques thus cannot be used for concurrent analytical query processing on GPUs, where tasks usually have long time spans and have high demands for device memory space. HyPE [5] is a hybrid engine for CPU-GPU query co-processing. The idea of its operator-based execution cost model is similar to the weighted demand metric proposed in this paper. Compared with these works, MultiQx-GPU identifies the critical demands and opportunities of supporting concurrent query executions in analytical GPU databases. It addresses a set of issues in GPU resource management to achieve high system performance under multi-query workloads.

In addition, there are several research works on GPU resource management in general-purpose computing systems. PTask [24] adds abstractions of GPUs in the OS kernel to support managing GPUs as first-class computing resources. It provides a dataflow-based programming model and enforces system-level management of GPU computing resources

and data movement. TimeGraph [13] is GPU scheduler to provide performance isolation for real-time graphics applications. Gdev [14] is an open-source CUDA driver and runtime system. It supports inter-process communication through GPU device memory and provides simple data swapping functionality based on the IPC mechanism. GDM [28] is an OS-level device memory manager.

9. SUMMARY AND FUTURE WORK

This paper presents the motivation, design, implementation, and evaluation of MultiQx-GPU, a high-performance software support system for concurrent analytical query processing with GPU devices. MultiQx-GPU provides two critically necessary functionalities, namely query scheduling and device memory swapping, to allow coordinated multi-query executions with GPUs. Our extensive experimental results show that MultiQx-GPU can significantly improve overall throughput when executing data warehousing benchmarks.

We will extend MultiQx-GPU in two directions. First, considering the importance of CPU-GPU co-processing for high-performance query executions, we plan to investigate the probability of combining CPU scheduling in the operating system with the GPU scheduling in our system. Second, under the guideline of minimal changes to query engines, current query scheduler in MultiQx-GPU does not consider data overlapping between different queries. We will study how to coordinate the two layers to support data sharing.

10. ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This work was supported in part by the National Science Foundation under grants CCF-0913050, OCI-1147522, and CNS-1162165.

11. REFERENCES

[1] code.google.com/p/gpudb.

[2] monetdb.org.

[3] docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html.

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

[5] S. Bress. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proc. VLDB Endow.*, 6(12), 2013.

[6] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336, 2006.

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

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

[9] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, 2011.

[10] M. Heimel and V. Markl. A first step towards GPU-assisted query optimization. In *ADMS*, 2012.

[11] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, 2013.

[12] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *DaMoN*, pages 55–62, 2012.

[13] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, pages 2–2, 2011.

[14] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX ATC*, 2012.

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

[16] 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.

[17] T. Mostak. An overview of MapD (massively parallel database). MIT Technical Report, 2013.

[18] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ISCA*, pages 63–74, 2008.

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

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

[21] P. O'Neil, B. O'Neil, and X. Chen. Star schema benchmark. cs.umb.edu/ poneil/StarSchemaB.PDF.

[22] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*, 2014.

[23] H. Pirk, S. Manegold, and M. L. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *VLDB*, pages 27–35, 2011.

[24] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *SOSP*, 2011.

[25] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[26] E. A. Sitaridi and K. A. Ross. Ameliorating memory contention of OLAP operators on GPU processors. In *DaMoN*, 2012.

[27] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, pages 234–244, 2000.

[28] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. GDM: device memory management for GPGPU computing. In *SIGMETRICS*, 2014.

[29] 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. *Proc. VLDB Endow.*, 5(11):1543–1554, 2012.

[30] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Micro*, pages 107–118, 2012.

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

[32] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.*, 6(10):817–828, 2013.