Advanced Concurrency Control Algorithm Design and GPU System Support for High Performance In-Memory Data Management

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Yuan Yuan

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

Xiaodong Zhang, Advisor

Spyros Blanas

Yang Wang
Abstract

The design and implementation of data management systems have been significantly affected by application demands and hardware advancements. On one hand, with the emerging of various new applications, the traditional one-size-fits-all data management system has evolved into domain specific systems optimized for each application (e.g., OLTP, OLAP, streaming, etc.). On the other hand, with increasing memory capacity, and advancements of multi-core CPUs and massive parallel co-processors (e.g., GPUs), the performance bottleneck of data management systems have shifted from I/O to memory accesses, which has led a constructive re-design of data management systems for memory resident data. Although many in-memory systems have been developed to deliver much better performance than that of disk-based systems, they all face the challenge of how to maximize the system’s performance by massive parallelism.

In this Ph.D. dissertation, we explore how to design high performance in-memory data management systems for massive parallel processors. We have identified three critical issues of in-memory data processing. First, Optimistic Concurrency Control (OCC) method has been commonly used for in-memory databases to ensure transaction serializability. Although OCC can achieve high performance at low contention, it causes large number of unnecessary transaction aborts at high contention, which wastes system resources and significantly degrades database throughput. To solve the problem, we propose a new concurrency control method named Balanced Concurrency Control (BCC)
that can more accurately abort transactions while maintaining OCC’s merits at low contention. Second, we study how to use the massive parallel co-processor GPUs to improve the performance of in-memory analytical systems. Existing works have demonstrated GPU’s performance advantage over CPU on simple analytical operations (e.g., join), but it is unclear how to optimize complex queries with various optimizations. To address the issue, we comprehensively examine analytical query behaviors on GPUs and design a new GPU in-memory analytical system to efficiently execute complex analytical workloads. Third, we investigate how to use GPUs to accelerate the performance of various analytical applications on production-level distributed in-memory data processing systems. Most of existing GPU works adopt a GPU-centric design, which completely redesigns a system for GPUs without considering the performance of CPU operations. It is unclear how much a CPU-optimized, distributed in-memory data processing system can benefit from GPUs. To answer the question, we use Apache Spark as a platform and design Spark-GPU that has addressed a set of real-world challenges incurred by the mismatches between Spark and GPU. Our research includes both algorithm design and system design and implementation in the form of open source software.
To my family and friends.
Acknowledgments

The work in this dissertation is not achievable without the support of a group of amazing people.

First and foremost, I would like to express my deepest gratitude to my advisor Professor Xiaodong Zhang. My Ph.D. study has been a wonderful journey under his guidance and continuous support. As my advisor, he has taught me every aspect of research, from identifying important problem to writing the paper and presenting the work. He gives me the freedom to explore the research topics that I am interested in and encourages me to solve important problems to impact the community. During these years, he has helped me grow from an engineer to a researcher. His pursuit of impact not only affects my Ph.D. study but also my future career.

I am grateful to work with my colleagues in the HPCS lab in the Ohio State University: Rubao Li, Yin Huai, Kaibo Wang, Tiao Luo, Kai Zhang, Xiaoning Ding, Feng Chen and Lei Guo. They have given me numerous insightful suggestions on my work and supported me in both my study and daily life. It is great to have them accompany along the rough road. Specially, I would like to thank Rubao Li. We have worked closely on many projects. His deep understanding of the problems and encouragements have helped me overcome various difficulties. He is a great friend.
I would like to thank my committee members Yang Wang and Spyros Blanas for their constructive comments to improve the dissertation.

Last but not least, I would like to thank my families for their love and support. Without them, I cannot reach this point.
Vita

2005.6.........................................................B.S. Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

2008.7........................................................M.S. Computer Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

2008.7 to 2010.8 .................................................Software Engineer, BWStor Inc., Beijing, China

2010.9 to present ...........................................Graduate Research Associate, Computer Science and Engineering, The Ohio State University

Publications


**Fields of Study**

Major Field: Computer Science and Engineering
Table of Contents

Abstract ............................................................................................................................................ ii

Acknowledgments ........................................................................................................................... v

Vita ................................................................................................................................................ vii

List of Tables .................................................................................................................................. xiv

List of Figures ................................................................................................................................. xv

Chapter 1 Introduction ................................................................................................................... 1

1.1 Concurrency Control for In-Memory OLTP databases ............................................................ 2

1.2 In-Memory GPU Analytical Database ..................................................................................... 4

1.3 Effectively Integrating GPUs to CPU-Optimized In-Memory Systems ......................... 5

Chapter 2 Balanced Concurrency Control for In-Memory OLTP Databases .................... 7

2.1 Introduction ............................................................................................................................... 7

2.2 Balanced Concurrency Control ............................................................................................... 11

2.2.1 Background ......................................................................................................................... 12

2.2.2 Essential Dependency Patterns ....................................................................................... 13

2.2.3 Detection of Essential Patterns ....................................................................................... 15
3.1.1 The Framework of Our Study................................................................. 52
3.1.2 Contributions of Performance Insights.............................................. 54
3.2 GPU Query Engine................................................................................. 55
  3.2.1 Engine Structure and Storage Format................................................. 55
  3.2.2 Query Operators.............................................................................. 56
  3.2.3 Implementation Details.................................................................... 57
3.3 Experimental Methodology.................................................................... 59
  3.3.1 Workloads....................................................................................... 59
  3.3.2 Hardware Platform.......................................................................... 59
  3.3.3 Software Platforms.......................................................................... 60
  3.3.4 Measurement Methodology and Tools.............................................. 60
  3.3.5 Measurement of Bandwidth............................................................... 61
3.4 Performance Analysis ............................................................................ 62
  3.4.1 SSBM Query behaviors................................................................. 62
  3.4.2 Effects of Data Compression......................................................... 65
  3.4.3 Effects of Transfer Overlapping..................................................... 69
  3.4.4 Effects of Invisible Join................................................................. 71
3.5 Performance Comparison................................................................. 73
  3.5.1 Comparisons of GPU and CPU...................................................... 73
4.4.2 GPU-Aware Query Optimizer ................................................................. 108
4.4.3 Optimizations ......................................................................................... 109
4.5 GPU Resource Management .................................................................... 110
  4.5.1 Our Solution: User-Level GPU Memory Management ..................... 113
  4.5.2 GPU Abstraction and Task Scheduling .............................................. 115
4.6 Experiments .............................................................................................. 115
  4.6.1 Experimental Environments and Workloads ...................................... 115
  4.6.2 Effectiveness of GPU Sharing .............................................................. 116
  4.6.3 Data Mining ......................................................................................... 118
  4.6.4. Statistical Analysis ........................................................................... 119
  4.6.5 Star Schema Benchmark .................................................................... 120
  4.6.6 TPC-H Benchmark ............................................................................ 122
  4.6.7 A Case for Spark-GPU ........................................................................ 124
4.7 Related Work ............................................................................................ 125
4.8 Summary .................................................................................................. 126
Chapter 5 Conclusions .................................................................................. 128
Bibliography .................................................................................................... 130
List of Tables

Table 1 Breakdown of BCC overhead for TPC-C at low contention with 32 threads..... 33
Table 2 Hardware Specifications................................................................. 60
Table 3 GPU bandwidth measurement ................................................................ 61
Table 4 Compression ratio for fact table columns when sorted on different foreign key columns.................................................................................................................. 67
Table 5 Notations for the Cost Model.................................................................... 84
List of Figures

Figure 1 A transaction schedule with \( T_f \) falsely aborted ........................................ 9
Figure 2 Different types of essential patterns when an unserializable schedule is created .................................................. 15
Figure 3 Comparison between the global clock and a tuple’s TID................................. 21
Figure 4 TPC-C transaction throughputs under low contention .................................. 32
Figure 5 Throughput of TPC-W as the per-thread contention probability is 100% .......... 35
Figure 6 Throughput of TPC-W as per-thread contention possibility varies.................. 38
Figure 7 Throughput of TPC-C NewOrder and Payment with a mix of 50%-50% .......... 39
Figure 8 Breakdown of TPC-C throughput by transaction types ................................. 41
Figure 9 YCSB throughput under high contention ....................................................... 43
Figure 10 BCC memory consumption for saving history hash table per-thread with 50%-50% workload of TPC-C NewOrder and Payment .................................................. 44
Figure 11 Latency of \( \text{NewOrder} \) transactions aborted by both OCC and BCC .......... 46
Figure 12 Latency of \( \text{NewOrder} \) transactions saved by BCC compared with that committed by OCC ................................................................. 47
Figure 13 A 3-Dimension Study of Processing Warehousing Queries on GPUs .......... 53
Figure 14 GPU Query Engine Architecture ............................................................... 55
Figure 15 SSBM Schema ......................................................................................... 59
Figure 16 Baseline of SSBM queries on NVIDIA GTX 680 ........................................... 62
Figure 17 SSBM execution time breakdown ........................................................................... 63
Figure 18 Speedup of data compression .................................................................................. 68
Figure 19 Speedup of transfer overlapping ............................................................................ 70
Figure 20 Speedup of invisible join ......................................................................................... 72
Figure 21 SSBM Performance Speedup .................................................................................. 74
Figure 22 Normalized OpenCL performance over CUDA ....................................................... 77
Figure 23 NVIDIA Versus AMD ............................................................................................. 78
Figure 24 Normalized SSBM performance on AMD GPU ....................................................... 79
Figure 25 SSBM performance comparison on CPU ................................................................. 81
Figure 26 Evaluate join model for different query characteristics ......................................... 87
Figure 27 Error rate of estimated performance on GTX680 .................................................... 88
Figure 28 Normalized kernel execution time on GTX480 ....................................................... 89
Figure 29 Estimated SSBM performance with different GPU hardware configurations ... 90
Figure 30 The overall architecture of Spark-GPU ................................................................. 98
Figure 31 The execution time of the micro benchmark with different number of string
columns in a row ....................................................................................................................... 103
Figure 32 Normalized performance when different number of tasks are running
concurrently on the GPU ........................................................................................................ 112
Figure 33 The architecture of GPU memory management library ..................................... 114
Figure 34 Effectiveness of GPU sharing in Spark-GPU ......................................................... 117
Figure 35 Results of executing one iteration of K-Means ...................................................... 119
Figure 36 Results of executing one iteration of logistic regression........................................ 120
Figure 37 Performance results of Star Schema Benchmark .................................................. 121
Figure 38 Performance Results of 4 TPC-H Queries......................................................... 123
Chapter 1 Introduction

In the digital age, data have become the most important assets. Today’s organizations rely heavily on efficient data management to serve customers’ needs and make strategic decisions. For example, wholesale stores dynamically adjust commodity price based on analysis of sales data and inventory; Internet companies (e.g., Facebook and Google) provide personalized information for users by conducting complex learning algorithms on huge amounts of data; online retailers analyze user’s browse and purchase history to make product recommendations to users. All these applications need to utilize data management systems to efficiently store and process data. Building high performance data management systems has become critical for the success of many organizations.

The design of data management systems has been significantly affected by application demands and hardware advancements. On one hand, with new emerging applications, the traditional “one size fits all” design of data management systems can no longer meet different application’s performance requirements [81]. Various domain-specific data management systems have been developed. For example, nowadays enterprises use online transaction processing (OLTP) systems to store operational data, use analytical systems to store historical data and conduct analysis, use key-value systems to store semi-structured data. Each domain-specific system is specifically designed to process certain applications.
On the other hand, two major hardware advancements that have happened in the past decade have revolutionized the designs of data management systems. One is that main memory capacity has greatly increased with the drop of its cost. It becomes common for an enterprise server to have more than 1 TB of memory. The growth rate of memory capacity is higher than that of hot data. In this case, hot data can be accommodated in a single server’s memory or a cluster’s memory for processing. With memory resident data, the performance bottleneck of data management systems has shifted from I/O to memory accesses and computation [42, 58]. As a result, the conventional designs that focus on optimizing I/O can no longer provide optimal performance [42, 66], which has triggered the design of in-memory data management systems. Many in-memory data management systems have been developed (e.g., [94, 27, 85, 58]). The other is that processors have become increasingly parallel and heterogeneous. Multi-socket multi-core CPUs and massive parallel co-processors GPUs have become common in enterprise servers. These massive parallel computing devices introduce new challenges for each domain-specific in-memory data management system: how to efficiently utilize the parallel resources and handle the increased concurrency level. In this dissertation, we have addressed the following three aspects of the challenges.

1.1 Concurrency Control for In-Memory OLTP databases

With the increase of CPU core counts, concurrency level in the OLTP database increases, so does the possibility of data contention. In this case, the concurrency control method, which has a great impact on the database’s performance and scalability, must perform well for both low contention and high contention workloads. Recent in-memory
OLTP databases (e.g., [27, 85]) have favored Optimistic Concurrency Control (OCC) [55] for good performance and scalability.

OCC is optimistic during transaction execution, which incurs rare inter-core communication and low synchronization cost under low contention. However, OCC is pessimistic when committing transactions. It aggressively aborts transactions to guarantee data consistency. OCC uses a simple criterion: whether a transaction’s read set has been changed to determine whether the transaction should be aborted, which can cause unnecessary transaction aborts. As data contention increases, OCC performs poorly due to large number of unnecessary transaction aborts.

Precisely aborting transactions requires OLTP databases to operate on shared data structures such as wait-for graphs to detect cycles in serialization graphs, which can significantly impact the database’s scalability, especially for low contention workloads [23].

To address the problem, we have designed a new concurrency control method named Balanced Concurrency Control (BCC) that seeks a sweet spot between false abort rate and run time overhead. BCC balances the accuracy and the overhead of transaction validation well. Specifically, in addition to detecting the anti-dependency as OCC does, BCC detects one additional data dependency in a confined search space, which, together with the anti-dependency, forms an essential dependency pattern. This pattern more reliably indicates the existence of a cycle in the transaction dependency graph (i.e. unserializable transaction schedule) than OCC’s criterion but incurs little overhead.
Extensive experiments confirm BCC’s performance advantage over OCC at high contention, which improves the database throughput by up to 3.68x. In the meantime, BCC has comparable performance with OCC at low contention.

1.2 In-Memory GPU Analytical Database

GPUs are massive parallel co-processors, which have higher device memory bandwidth and computing powers compared to multi-core CPUs. As GPUs become general-purpose computing devices, researchers have started to investigate how to improve the performance of in-memory analytical databases with GPUs. Several analytical operations, such as join [45, 52], aggregation [37, 79] and sort [36, 77], have been implemented on GPUs with new algorithms, which greatly improve the performance. However, two critical issues have not been paid serious attention in existing work, which limit the adoption of GPUs by in-memory analytical systems.

The first issue is how to optimize complex analytical queries on GPUs. Existing research mainly focus on accelerating single analytical operation. It is unclear whether in-memory analytical databases can benefit from processing more complex workloads on GPUs, considering the diversified workloads and data characteristics, different hardware environments and software optimizations. The second issue is how to characterize query behaviors on GPUs to best utilize the computing resources in a heterogeneous environment.

To solve the problem, we have designed a GPU analytical query engine with a set of advanced optimizations that can execute star-schema queries [67]. Based on the query engine, we have analyzed the behaviors of analytical queries and the effects of various
software optimization techniques. Moreover, we have developed a GPU analytical model to characterize and quantify query execution time on GPUs. Our work has given insights and directions for adopting GPUs into the query engine in the fittest way.

1.3 Effectively Integrating GPUs to CPU-Optimized In-Memory Systems

Existing research work has demonstrated GPU’s performance advantage over CPU for data-parallel, compute-intensive applications because of GPU’s high computational power and memory bandwidth. However, many of these work adopt a GPU-centric design, which redesigns the system based on GPU’s characteristics to maximize GPU’s performance without considering the performance of CPU operations. It is unclear how much a CPU optimized system can benefit from the high performance GPUs considering the different types of optimizations for CPU and GPU.

To answer this question, we use Apache Spark as a platform to explore how to use GPUs to accelerate the performance of various analytic applications on CPU optimized distributed in-memory data processing systems. Specifically, we have designed and implemented Spark-GPU, a CPU-GPU hybrid data analytics system that can not only run SQL queries but also various complex analytic applications on both CPUs and GPUs. We present a set of designs that effectively connect GPUs to Spark to best utilize GPU’s capability. Spark-GPU uses heuristic rules to offload SQL queries to GPUs and provides block processing ability for GPUs to get the best performance of analytic applications. Our comprehensive evaluation shows that Spark-GPU can have up to 4.83x performance speedup for SQL queries, and have up to 16.13x performance speedup for compute-intensive machine learning applications.
The rest of the dissertation is organized as follows. Chapter 2 introduces the Balanced Concurrency Control method for in-memory OLTP databases. Chapter 3 presents our design of GPU analytical database and a comprehensive study of query behaviors on GPUs. In Chapter 4, we describe Spark-GPU, an accelerated in-memory data processing engine on clusters. Finally, we conclude the dissertation in Chapter 5.
Chapter 2 Balanced Concurrency Control for In-Memory OLTP Databases

2.1 Introduction

The rapid increase of memory capacity has made it possible to store the entire OLTP database in the memory of one single server. With memory-resident data, database’s performance bottleneck has shifted from disk I/O to software related overhead such as locking and buffer management [42, 28]. This has triggered the re-design of the database systems for the in-memory data. Of the concurrency control methods that have a great impact on database performance, Optimistic Concurrency Control (OCC) [55] has been favored by recent in-memory databases for high performance and scalability [56, 27, 85, 89, 97, 75].

With the OCC method, a database executes each transaction in three phases: read, validation, and write. In the read phase, the database keeps track of what the transaction reads into a read set and buffers the transaction’s writes into a write set in the transaction’s private storage. In the validation phase, the database validates the transaction’s read set. If the transaction’s read set has been changed, the transaction must be aborted. Otherwise, the transaction proceeds to the write phase, in which the database installs the transaction’s writes to the database storage. The validation and write phases must be executed in the critical section.
OCC is optimistic in the read phase. It assumes that all the transactions can proceed concurrently. Transactions in the read phase cannot block the execution of other transactions. Being optimistic maximizes concurrency level, leading to high scalability and throughput. However, OCC becomes pessimistic in the validation phase. It excessively aborts transactions to ensure serializability. Some aborted transactions may not affect serializability because change in a transaction’s read set is not a sufficient condition that the transaction schedule cannot be serialized. Based on the serializability theory, only transactions forming a cycle in their dependency graph cannot be serialized and should be aborted [39]. This chapter refers to the transactions aborted false-positively as false aborts, to differentiate them from the transactions that actually violate the serializability requirement.

A false abort happens when a transaction is aborted by OCC (i.e. read set changed) but it meets the serializability requirement (i.e., not in a cycle in dependency graph). The differences between these two criteria can be illustrated with the following two transactions: $T_1$: r(A) w(B) and $T_2$: r(A) w(A). Figure 1 shows a schedule of $T_1$ and $T_2$. According to OCC’s validation criterion, $T_2$ can successfully commit since its read set is not changed, while $T_1$ must be aborted since its read set has been changed by $T_2$. However, based on the serializability theory, since there is no cycle in the serialization graph, both $T_1$ and $T_2$ should be committed. If both of them were allowed to commit, the database state would be the same as that after $T_1$ and $T_2$ execute serially. Since $T_1$ is aborted by OCC though it should be allowed to commit based on the serializability requirement, the abort is a false abort.
Figure 1 A transaction schedule with $T_i$ falsely aborted

When data contention is low, aborts, as well as false aborts, are rare, being pessimistic in transaction validation will not cause serious performance issues. For example, in-memory OCC databases can achieve throughput of over 500,000 transactions per second under low-contention workloads [85, 89]. However, when data contention becomes intensive, an increasing number of transactions may be aborted false-positively. Data contention becomes intensive with the increase of CPU core counts. Data sets with skewed characteristics, e.g., those in OLTP workloads [83], also intensify contention. Based on our experiments, false aborts can reduce system throughput by 3.68x under a TPC-W-like workload. It is important for databases to provide good performance in both low contention and high contention scenarios.

To completely remove false aborts, database systems must abort transactions based on cycle detection in serialization graphs. The idea of detecting partial cycle dependency to guarantee serializability was first proposed by Cahill et al. [21] for disk-based snapshot isolation databases. However, the techniques are not directly applicable to in-memory databases where transactions are usually very short due to their prohibitive cost.
Detecting cycles requires the database to operate on shared data structures such as wait-for graphs, which will significantly impact the system’s scalability, especially for low contention workloads [23].

The dilemma lies between improving the validation with an accurate criterion to abort transactions and maintaining a low overhead for transaction execution. In this chapter, we resolve the dilemma by proposing the Balanced Concurrency Control (BCC) method that seeks a sweet spot between being careful and fast. This balances the accuracy and the overhead of transaction validation well. Specifically, in addition to detecting the anti-dependency as OCC does, BCC detects one additional data dependency in a confined search space, which, together with the anti-dependency, forms an essential dependency pattern. This pattern more reliably indicates the existence of a cycle in the transaction dependency graph (i.e. unserializable transaction schedule) than OCC’s criterion. We will show that by examining one additional dependency BCC can effectively reduce false aborts. At the same time, since BCC limits the search space for the additional dependency, the overhead for dependency detection can be effectively controlled through careful system design and implementation.

This chapter makes the following contributions. First, it describes a new concurrency control method for reducing false aborts while retaining OCC’s merits for low contention workloads. Second, it proposes an optimized BCC method leveraging the state-of-the-art in-memory database features. Third, this chapter studies implementation techniques for minimizing run-time overhead. Fourth, to demonstrate BCC’s effectiveness, we implement it in Silo [85], which is a representative OCC-based in-memory database. Our
implementation makes a case of how to adopt BCC in an OCC-based concurrency control kernel. Finally, we comprehensively evaluate BCC’s performance on a 32-core machine. Our results demonstrate that BCC has a decisive performance advantage over OCC when contention becomes intensive. This advantage is due to a reduction in transaction aborts, and an increase in transaction throughput and workload scalability. Meanwhile, BCC has comparable performance with OCC for low contention workloads.

The remainder of this chapter is organized as follows. Chapter 2.2 describes how BCC works in general. Chapter 2.3 presents an optimized BCC method for in-memory databases. Chapter 2.4 and Chapter 2.5 describe the design and implementation of the optimized BCC method. We comprehensively evaluate BCC in Chapter 2.6. We introduce related work in Chapter 2.7 and summarize our work in Chapter 2.8.

2.2 Balanced Concurrency Control

BCC is an optimistic concurrency control method in nature. The key difference between BCC and other optimistic methods lies in the validation phase - how to determine if a transaction schedule is unserializable. In this chapter we first review the concepts of transaction history and data dependency in databases. Then we present BCC’s transaction model and the essential dependency patterns that BCC utilizes in its validation to guarantee serializability. After that we explain how BCC detects the essential patterns and discuss BCC’s overhead.
2.2.1 Background

Transaction history

A transaction history is an execution of database transactions that specifies a partial order of transactional operations on database tuples. Similar to [14], we use $r_i[x_j]$ to represent that transaction $T_i$ reads the version $j$ of tuple $x$, $w_i[x_j]$ to represent that $T_i$ writes the version $j$ of tuple $x$, $c_i$ to represent that $T_i$ is committed and $a_i$ to represent that $T_i$ is aborted. Given a tuple $x$’s two versions, $x_i$ is generated before $x_j$ if $i < j$.

Data dependency

Data dependencies happen between transactions when they operate on the same tuple and at least one of the operations is write. The types of dependencies are determined by the operation type (read or write) and the order in which the transactions commit.

There are three types of data dependencies.

- Write-Read ($wr$) dependency: if transaction $T_j$ reads a tuple that has been committed earlier by another transaction $T_i$, $T_j$ is $wr$ dependent on $T_i$, denoted as $T_i -wr-> T_j$.

- Write-Write ($ww$) dependency: if transaction $T_j$ commits a tuple that has been committed earlier by another transaction $T_i$, $T_j$ is $ww$ dependent on $T_i$, denoted as $T_i -ww-> T_j$.

- Read-Write ($rw$) dependency: if transaction $T_j$ commits a tuple that has been read earlier by another transaction $T_i$, $T_j$ is $rw$ dependent on $T_i$ (or $T_i$ is anti-dependent on $T_j$), denoted as $T_i -rw-> T_j$. 
We use $T_i \rightarrow T_j$ to denote that $T_j$ depends on $T_i$ through any of the above dependency types.

### 2.2.2 Essential Dependency Patterns

BCC assumes the following transaction model.

- Each transaction is executed in read, validation and write phases.
- Each transaction can only read committed tuples.
- The validation and write phases must be executed in the critical section.

Note that BCC and OCC [55] have the same transaction model.

In the validation phase, BCC exploits essential dependency patterns (or essential patterns for brevity) among transactions to determine unserializable transaction schedules. Each essential pattern specifies that certain data dependencies exist between transactions.

We will demonstrate that the existence of an essential pattern is a necessary condition that a transaction schedule is unserializable in databases that satisfy BCC’s transaction model. In this case, BCC ensures serializability by avoiding the essential patterns.

The essential dependency patterns that BCC detects and prevents are described as follows.

**Theorem 1.** In databases that satisfy BCC’s transaction model, when an unserializable transaction schedule is created, the schedule must contain the following transactions $T_1$, $T_2$ and $T_3$ such that

1. $T_3$ is the earliest committed transaction in the schedule;
2. $T_2 - rw \rightarrow T_3$; and
3. $T_1 \rightarrow T_2$ and $T_1$ commits after $T_2$ starts.

The data dependency patterns formed by $T_1$, $T_2$ and $T_3$ are called the essential patterns.
Proof. When an unserializable transaction schedule is created, a cycle must exist in the transaction dependency graph. Let $T_3$ be the first transaction committed in the schedule. To form the cycle, $T_3$ must be dependent on another transaction (i.e. it should be pointed by an arrow in the dependency cycle). Since a transaction can only read committed tuples, this dependency cannot be a $ww$ dependency or a $wr$ dependency (otherwise there would exist another transaction in the schedule that committed earlier than $T_3$ committed, which contradicts with the fact that $T_3$ is the first committed transaction in the schedule). Thus, this dependency must be a $rw$ dependency. Let the transaction $T_3 \text{ } rw$ dependent on be $T_2$ (i.e. $T_2 \rightarrow \text{rw-} \rightarrow T_3$). $T_3$ must commit after $T_2$ starts. To form the cycle, $T_2$ must also be dependent on a transaction in the cycle. Let the transaction be $T_1$ (i.e. $T_1 \rightarrow \rightarrow T_2$). $T_1$ must commit after $T_2$ starts, because $T_3$ commits after $T_2$ starts and $T_1$ commits later than $T_3$ commits.

Theorem 2. Transaction aborted by OCC may not be aborted by BCC, while transactions aborted by BCC will always be aborted by OCC.

Proof. The dependency $T_2 \rightarrow \text{rw-} \rightarrow T_3$ in the essential patterns is the anti-dependency detected by OCC, and the essential patterns examine additional dependencies to decide whether a transaction should be aborted.

BCC utilizes the essential patterns to ensure serializability for three reasons. First, based on Theorem 1, validation based on detecting essential patterns only commits serializable transactions. Second, based on Theorem 2, validation based on detecting essential patterns reduces aborts compared to OCC. Third, the overhead of detecting the
essential patterns can be effectively controlled by limiting the search space: BCC excludes all transaction \( T_1 \) that commit before \( T_2 \) starts.

### 2.2.3 Detection of Essential Patterns

A BCC database aborts a transaction if committing the transaction would create an essential pattern. To detect the essential patterns, the database needs to decide: (1) what data dependencies should be examined in each transaction’s validation phase; and (2) what data dependency information should be kept for validating other transactions.

\[
\begin{align*}
\text{(a)} & \quad w_i[A_i] \rightarrow r_i[B_i] \rightarrow c_1 \\
& \quad r_i[B_i] \rightarrow w_i[A_i] \rightarrow a_2 \\
& \quad w_i[B_i] \rightarrow c_3 \\
\text{(b)} & \quad w_i[A_i] \rightarrow r_i[B_i] \rightarrow c_1 \\
& \quad r_i[B_i] \rightarrow r_i[A_i] \rightarrow a_2 \\
& \quad w_i[B_i] \rightarrow c_3 \\
\text{(c)} & \quad r_i[A_i] \rightarrow r_i[B_i] \rightarrow c_1 \\
& \quad r_i[B_i] \rightarrow w_i[A_i] \rightarrow a_2 \\
& \quad w_i[B_i] \rightarrow c_3 \\
\text{(d)} & \quad r_i[A_i] \rightarrow r_i[B_i] \rightarrow a_1 \\
& \quad r_i[B_i] \rightarrow w_i[A_i] \rightarrow c_2 \\
& \quad w_i[B_i] \rightarrow c_3 \\
& \quad T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3 \xrightarrow{wr}
\end{align*}
\]

Figure 2 Different types of essential patterns when an unserializable schedule is created
Figure 2 shows all possible essential patterns when an unserializable transaction schedule is created. The essential patterns can be divided into two categories based on when they would be created.

The first category contains three essential patterns that would be created at the time a transaction $T_2$ commits, which are shown in Figures 2(a) to 2(c). To detect these patterns, the database needs to validate if any transaction could be $T_2$ in the essential pattern by checking: (1) if the transaction $T_2$ is anti-dependent on a committed transaction $T_3$, or $T_2\rightarrow\text{rw}>T_3$; (2) if the transaction $T_2$ is $\text{ww}$-, $\text{wr}$- or $\text{rw}$-dependent on any concurrency transaction $T_1$.

The second category only manifests with snapshot transactions that always operate on a consistent snapshot of the database. The essential pattern that would be created at the time snapshot transaction $T_1$ commits is shown in Figure 2(d). In this case, when $T_1$’s snapshot time is before $T_2$’s commit time and $T_1$’s read operation happens after $T_2$ commits, the dependency $T_1\rightarrow\text{rw}>T_2$ can only be detected when $T_1$ commits. To detect this pattern, the database needs to validate if any transaction could be $T_1$ in the essential pattern by checking if the transaction $T_1$ is a snapshot transaction and $T_1$ is anti-dependent on a committed transaction $T_2$ ($T_1\rightarrow\text{rw}>T_2$), which is in turn anti-dependent on another committed transaction $T_3$ ($T_2\rightarrow\text{rw}>T_3$). This requires the database to retain all anti-dependency information.

Algorithm 1 summarizes how BCC validates a transaction $T$ to detect and prevent the essential patterns.
BCC examines two data dependencies to detect the essential patterns. Theoretically, examining more dependencies can further reduce false aborts. However, the overheads will increase significantly. If detecting more dependencies, the dependencies can happen not only between concurrent transactions, but also between an active transaction and previously committed transactions that were concurrent with other active transactions. This makes the cost of checking dependency increase exponentially. Even disk-based databases (e.g., PostgreSQL) avoid considering more than two dependencies [75] because of the high overhead [73].

2.3 An optimized BCC Method

In-memory databases can execute transactions as snapshot transactions. In some state of the art in-memory databases, such as [85, 89], read-only transactions are executed as snapshot transactions while write transactions are not. There are two reasons for this design. First, read-only transactions may be continuously aborted when running with other write transactions. Running them as snapshot transactions guarantees that they will
never be aborted, although they may read stale data. Second, write transactions dominate the transactions executed by the database. Running them as snapshot transactions could introduce expensive operations like acquiring latches and locks for read operations in some concurrency control kernels, which will degrade the database’s performance and scalability when the database has strived to avoid all centralized hotspots and scalability bottlenecks.

The BCC method requires the database to maintain a history of anti-dependency information to detect the essential pattern shown in Figure 2.2(d). If this is naively implemented, it can add a centralized hotspot to the in-memory database kernel and hurt BCC’s scalability. The overhead is caused by the fact that when a transaction \( T_i \) is a snapshot transaction, the dependency \( T_i \rightarrow rw \rightarrow T_2 \) may not exist when \( T_2 \) commits. In this case, to avoid BCC’s overhead of maintaining historical dependency information, the database must guarantee that read-only snapshot transactions may never appear in any essential pattern.

Existing in-memory OCC databases avoid validating the snapshot transaction by taking an early snapshot time. However, this does not work in BCC. The reason is that the dependency cycle that is shown in Figure 2.2(d) can be created, with transaction \( T_i \) being the snapshot transaction, no matter when the snapshot is taken. Based on BCC’s validation criteria, both transactions \( T_2 \) and \( T_3 \) would be allowed to commit because no essential patterns are detected. To guarantee serializability, the database must validate the snapshot transaction \( T_i \), detect the essential pattern, and abort \( T_i \).
We solve the problem by adding a light-weight synchronization point for snapshot transactions. The idea is that when a new read-only snapshot transaction begins, the database doesn’t immediately start executing the snapshot transaction. Instead, it waits until all the active transactions are finished. During this period, no new transactions will be executed. After all active transactions have finished, the database takes a snapshot for the snapshot transaction and resumes executing transactions as normal.

With the above snapshot mechanism, a read-only snapshot transaction cannot become part of the essential patterns. This can be proved by contradiction. Assume that a read-only snapshot transaction could be part of the essential patterns. Since the snapshot transaction doesn’t write any tuple, it cannot be $T_3$ in the essential patterns. Let the snapshot transaction be $T_2$. Then the essential pattern would be $T_1 \rightarrow_{wr} T_2 \rightarrow_{rw} T_3$. In this case, $T_1$ must commit before $T_2$ takes the snapshot, and $T_3$ must commit after $T_2$ takes the snapshot. This means $T_1$ commits earlier than $T_3$, which contradicts with the fact that $T_3$ is the first transaction committed in the dependency cycle. Let the snapshot transaction be $T_1$. Then the essential pattern would be $T_1 \rightarrow_{rw} T_2 \rightarrow_{rw} T_3$. The snapshot transaction can only be $wr$ dependent on another transaction to form a cycle. Let the transaction that $T_1$ is $wr$ dependent on be $T_0$. $T_0$ must commit before $T_1$ starts. With our snapshot mechanism, $T_2$ and $T_3$ cannot start earlier than $T_1$. Thus $T_0$ must commit earlier than $T_3$, which contradicts with the fact that $T_3$ is the first committed transaction in the dependency cycle.

In this case, the database only needs to validate if a transaction can become the essential pattern’s $T_2$ to guarantee serializability. The overhead of maintaining history
anti-dependency information is avoided. Moreover, there is no need to maintain the read set of read-only snapshot transaction and validate it.

This optimization technique targets long-running read-only transactions where a short execution delay is acceptable. Users can always choose to revert to the original BCC protocol (and accept the additional overhead) if slight latency degradation is unacceptable.

2.4 Detailed BCC Implementation

To support BCC in the database, two components must be implemented. One is a global clock to help detect data dependencies between concurrent transactions, the other is efficient management of the tuples accessed by each transaction. These two components are introduced in Chapter 2.4.1 and Chapter 2.4.2, respectively. Chapter 2.4.3 presents how to detect data dependency and Chapter 2.4.4 discusses phantom problems. In the last part we explain how a BCC database executes transactions.

2.4.1 Global Clock

BCC needs a global clock to help decide if a data dependency should be considered as part of the essential pattern.

Our design of the global clock relies on the following in-memory database’s features. First, to achieve good scalability, in-memory databases generate Transaction IDs (TIDs) in a decentralized way. For example, in Silo [85], which is a representative in-memory database, each TID can be divided into three parts: (1) thread index, which denotes the database thread that generated the TID; (2) the value of the database thread’s local counter; and (3) the value of global epoch, which is a slowly advanced global timestamp in the database. A database thread can generate a TID by reading its local counter and the
global epoch without synchronizations. One important property of the TIDs is that TIDs generated by the same database thread increase monotonically. However, this property doesn’t hold for TIDs generated by different database threads. Second, each tuple in the database has an associated metadata recording the TID of the latest transaction that has written the tuple.

The global clock is designed as a global TID vector. The number of entries in the vector is the same as the number of available threads in the database. Each thread has a corresponding entry in the global clock, which records the thread’s most recently assigned TID. A database thread must update its entry in the global clock every time it assigns a new TID.

The database can determine the order of a global clock value and a TID in two steps. The database first finds the database thread that generated the TID. Then it compares the value in the thread’s global clock entry with the TID. The one with a smaller value happened first. The comparison process is shown in Figure 3.

![Figure 3 Comparison between the global clock and a tuple’s TID](image.png)
The global clock is used in data dependency detections, which will be described in Chapter 2.4.3.

2.4.2 Transaction Data Management

BCC requires the database to keep track of each transaction’s read set and write set to detect data dependencies. In the validation phase, the database checks if the transaction’s read set has been changed and if the transaction’s write set overlaps with other concurrent transactions’ read sets. In this case, the transaction’s write set can be simply kept in the database thread’s local storage and will be released when the transaction finishes. On the other hand, the transaction’s read set must be stored in the shared memory. Next we discuss how to efficiently manage the read sets.

Organization

We use hash table to organize the read set, since it may be searched by multiple database threads. A common approach is to use a shared hash table to store the tuples read by each transaction, but it requires synchronizations when accessing the hash table. For example, if two transactions read the same tuple, they will modify the same entry in the hash table, which must be synchronized. To ease the synchronization overhead, instead of maintaining one shared hash table across the database, each database thread maintains a separate hash table for each transaction. Each entry in the hash table contains a pointer to a tuple and the tuple’s TID. A transaction’s hash table must be kept in the memory until the transaction’s concurrent peers have finished. Hash tables allocated by the same database thread are organized into a history list. Each entry in the list is a triple
<TID, Address, Release>, where TID specifies which transaction the hash table belongs to; Address records the hash table’s starting memory address; and Release determines when the hash table can be released.

**Allocation and release**

A database thread allocates a hash table when it starts a transaction T. It also allocates an entry in the history list to store the hash table’s memory address and T’s TID.

The hash table’s Release is set with the maximum TID in the global clock after T finishes. In this way, any transaction that has a larger TID than Release of T’s hash table must start after T finishes and is not concurrent with T.

To release T’s hash table, the database thread must guarantee that all T’s concurrent transactions have finished. This can be determined by checking if the minimum TID in the global clock is larger than the hash table’s Release. If minimum TID in the global clock is larger, all the active transactions in the database must start after T finishes. The hash table can be safely released.

The release mechanism is conservative. A transaction T’s hash table is not immediately released after T’s concurrent transactions have finished. But it guarantees safe release of each hash table without synchronizations.

**Synchronizations**

Since a hash table can only be modified by one database thread and is not released until all concurrent transactions have finished, the only scenario that needs synchronization is while a thread is inserting into the hash table, another one is searching the hash table for rw dependency (i.e. T_i –rw-> T_j). Without synchronization, an actually
happened \textit{rw} dependency may not be detected by the searching thread, which can cause serializability problem. Protecting the hash table with latch can solve the problem, but it harms scalability thus we avoid it.

We solve the problem by verifying the tuple after inserting it into the hash table. If any change has happened, a \textit{rw} dependency may not be detected. Thus the database thread must discard the old tuple and re-read the tuple. This guarantees that either the \textit{rw} dependency can be detected later, or the thread will read the newest tuple. The synchronization process is shown in Algorithm 2.

```
Algorithm 2 Hash table synchronization
1: read a tuple from database;
2: insert the tuple into the hash table;
3: while true do
4:   read the same tuple from database;
5:   if tuple has changed after inserting into the hash table then
6:     discard the old tuple from hash table;
7:     insert the new tuple into the hash table;
8:   else
9:     break;
10: end if
11: end while
```

\textbf{Garbage collection}

In the BCC database, a transaction \( T \)'s hash table is kept in the database until \( T \)'s concurrent transactions have finished. It is possible that some tuples that stored in \( T \)'s hash table have been garbage collected by the database when a database thread searches \( T \)'s hash table. This does not cause any problem because the hash table contains sufficient information (tuple's address and tuple's TID) to detect the data dependency. The search thread only needs to check if the tuple is in the hash table. There is no need to access the content of the tuple.
2.4.3 Data Dependency Detection

Detect anti-dependency

The anti-dependency is detected with OCC’s criterion by checking whether T’s read set has been changed.

Detect wr and ww dependencies

The database thread first takes a snapshot of the global clock when T starts and stores the value as Start in T’s local memory. To detect wr, every time T reads a tuple, the thread compares the tuple’s TID with Start to decide if TID is generated after Start. If TID is generated later than Start, wr has happened. The ww dependency is detected after T enters the validation phase in a similar way. If any tuple in the transaction’s write set has a TID that is generated later than Start, ww has happened.

Detect rw dependency

The database thread first takes a snapshot of the global clock when T starts and stores the value as Start in T’s local memory. It takes another snapshot of the global clock after T enters the validation phase and stores the value as End in T’s local memory. Start and End define the TID range of the concurrent transactions that T may be rw dependent on.

With Start and End, the database thread simply goes through other thread’s history lists and check if T’s write set overlaps with any hash table whose TID is generated later than Start but earlier than End. If there is overlap, rw has happened.

Since taking the snapshot of the global clock doesn’t need synchronizations, it is possible that while the thread is taking the second snapshot, new transactions have started. These transactions may not be considered as concurrent with T. This will not cause any
problem. The reason is that these new transactions cannot read the tuples in $T$’s write set since $T$ is in the critical section. Thus $T$ cannot be $rw$ dependent on them.

2.4.4 Phantom

Phantom problem can happen when a transaction is executing a range query while a concurrent transaction inserts a new tuple into the range. In the essential patterns, phantom can happen in two cases: (1) $T_2$ is the read transaction and $T_3$ is the insert transaction and (2) $T_1$ is the read transaction and $T_2$ is the insert transaction. The BCC database avoids phantom in the same way as recent in-memory OCC database (e.g., [85, 89]) does, which will abort the read transaction if phantom happens. In the first case, phantom will be detected in the validation of $T_2$ and $T_2$ will be aborted. In the second case, there is no need to detect $T_1$ \(-rw->T_2\) in the validation of $T_2$ since $T_1$ will be aborted when validating $T_1$.

2.4.5 Put Together: A Transaction’s Life

With the above designs, we now illustrate how a BCC database works in different transaction execution phases. The process is shown in Algorithm 3.

When a transaction $T$ starts, the database first assigns $T$ a new TID and updates the corresponding entry in the global clock. Then the database takes a snapshot shot of the global clock, which serves as multiple purposes. First, it is used to help determine the concurrent transactions for the later validation. Second, it is used to set the Release field of previous transaction’s hash table with the maximum TID value in the global clock. Third, it is used to release unused history hash tables by finding the minimum TID in the
global clock and releasing all hash tables whose *Release* are smaller than the minimum

TID. The database also allocates a new hash table for the transaction.

**Algorithm 3 How a BCC database works in different phases**

1. **Transaction Start:**
2. assign a TID and update the global clock;
3. take a snapshot of the global clock;
4. allocate a new hash table and release history hash tables;
5. 
6. **Transaction Validation:**
7. enter the critical section;
8. take a snapshot of the global clock;
9. left_conflict = right_conflict = 0;
10. if there exists anti-dependency then
11. right_conflict = 1;
12. find concurrent transactions;
13. if *T* is wr, ww or rw dependent on its concurrent transactions
   then
14. left_conflict = 1;
15. end if
16. end if
17. if right_conflict == 1 and left_conflict == 1 then
18. set the *Release* field of the transaction’s hash table;
19. abort the transaction;
20. else
21. install the writes and commit the transaction;
22. end if
23. leave the critical section;

When the transaction enters the validation phase, the database first takes a snapshot of
the global clock, which can be used together with the clock taken in line 3 to determine concurrent transactions. Then the database checks if *T* is anti-dependent on any committed transaction. If no anti-dependency exist, *T* will be committed since no essential pattern will be created. Otherwise the database checks if *T* is dependent on any of its concurrent transactions. This requires the database to find all the transactions that are concurrent with *T* and check data dependencies between them. The data dependency is checked in the order of wr, ww and rw. If any data dependency is detected, the
transaction will be aborted. The database will also set the Release of the aborted transaction’s hash table such that the hash table can be immediately released. Otherwise the transaction will be committed.

2.5 Experimental Method

To evaluate BCC’s effectiveness, we have implemented BCC and two phase locking (2PL) in Silo [85], which is a multi-threaded, shared in-memory OCC database. Silo generates TIDs in a decentralized way. It maintains a thread-local read-set and write-set for each transaction. Tuples in the transaction’s write set are locked in a deterministic order before validation starts. After that, Silo assigns a TID to the transaction if the transaction writes to the database and validates the transaction using OCC’s criterion.

2.5.1 Silo with BCC

Multi-Level Circular Buffers

Each thread in BCC database requires a memory space to store history hash tables, which may be allocated, released, and checked frequently. It is necessary to manage this memory space efficiently.

One way to manage the space is to organize it as a single region and use a free list to record the memory blocks available for new allocations. However, this approach would incur serious cache misses when each newly allocated hash table is filled with read sets. This problem can be addressed by utilizing two special characteristics of BCC thread’s memory operations: (1) the hash tables are always released in the same order as they are allocated; (2) the memory demand of each thread for storing history hash tables is usually low, and only occasionally jitters to its maximum requirement.
In our implementation, each thread partitions its memory space into three smaller areas that are managed with three levels of circular buffers. The lowest-level circular buffer is the smallest and can fit into the L1 CPU cache; the next one is slightly larger but is smaller than the L2 cache; the highest-level buffer is the largest one and can be any size that satisfies the maximum memory requirement of a thread. The database thread always tries to allocate memory from a lower-level circular buffer, and only resorts to a higher one when the lower buffer space becomes depleted. In each circular buffer, the memory is always allocated and released in a chase-tail fashion. Since most OLTP transactions are short and the average memory requirement of each database thread is low, this design ensures that most hash table operations can be satisfied in the L1 or L2 (or even L3) CPU cache.

**Global clock**

We implement the global clock as a set of sub-vectors. The number of sub-vectors equals to the number of CPU sockets and each sub-vector is a continuous array aligned on a single cache line on each socket.

**TID generation**

We use Silo’s distributed TID generator to generate TIDs for every transaction. The original Silo only assigns TIDs to transactions that write to the database. We modify the TID generator such that every transaction will be assigned a TID. Every time a database thread generates a TID, it will update its entry in the global clock. For each database thread, we use the last TID generated by the thread to identify the current transaction’s hash table since Silo generates TIDs in the validation phase.
**Snapshot transactions**

We add a synchronization point in the database before a snapshot transaction begins. After the synchronization, the database first advances the global epoch and then starts executing transactions. The snapshot is created based on the current Epoch value. Only tuples written in the previous Epoch can be read by the snapshot transactions.

**2.5.2 Silo with 2PL**

Our 2PL implementation was motivated by [74]. We avoid the centralized lock manager which generates suboptimal performance. Instead, we implement the per-tuple lock, and associate each tuple with a shared read lock and an exclusive write lock. No lock lists are used. To avoid the deadlock detection overhead, we adopt the wait-die 2PL mechanism [76]. A global timestamp allocator, which is implemented as an atomic variable, assigns timestamp to each transaction to differentiate the precedence of transactions.

**2.5.3 Experimental Setup**

All experiments are conducted on a 32-core machine with four 2.13GHz Intel Xeon E7-8830 CPUs and 128GB memory. Hyper-threading is disabled to yield the best base performance of Silo [85]. The operating system is 64-bit Linux with 2.6.32 kernel. The version of GCC compiler is 4.8.2.

To avoid stalls due to user interaction, no network clients are involved in our experiments. Each database thread runs on a dedicated CPU core and has a local workload generator to generate input transactions for itself. Database logging is also disabled. All table data are resident in main memory and no disk activities are involved.
during each measurement. For each measurement, we run the experiment for 10 times, each lasting for 30 seconds, and the median results are reported.

2.6 Experiment Results

In this section we present the performance results of BCC, OCC, and 2PL based on the prototype implementation in Silo. The experiment results confirm our expectations for BCC performance as follows:

- BCC achieves comparable performance and scalability with OCC when the workload contention is low.
- BCC significantly improves transaction throughput for high-contention workloads: BCC improves the throughput by 3.68x over OCC and by 2x over 2PL.
- BCC’s overhead on memory consumption and increased transaction latency is acceptable.

The performance results demonstrate BCC’s usefulness in practice, which can provide good performance in both low contention and high contention scenarios.

2.6.1 Low Contention

We first evaluate how BCC performs when data contention is low. TPC-C [8] and YCSB [25] benchmarks are used in the experiments. Due to limited space, we only present TPC-C’s results here. YCSB’s results are similar.

**TPC-C**

TPC-C is an industry-standard benchmark for evaluating transaction database performance. It models the operations in a wholesale store that consists of a number of warehouses. In the Silo implementation, TPC-C tables are partitioned across the
warehouses. We set the number of warehouses (i.e., the scale factor) to be the same with the number of database threads. In this configuration, each thread will mostly operate on the data items in its own warehouse, which makes the chance of data contention rare.

Figure 4 shows the transaction throughputs achieved by BCC, OCC and 2PL as the number of threads increases. As can be seen from the figure, BCC performs comparably and scales near-linearly as OCC does for the TPC-C workload. When there are 32 threads, BCC delivers an overall throughput of 1.15M transactions per second, which is only 7.29% lower than that achieved by OCC (1.24M). Despite the extra operations introduced by BCC for detecting the essential patterns, inter-core communication induced for checking history hash tables is rare when there are few data contentions. This makes BCC’s overhead low, retaining OCC’s performance benefits for low-contention workloads.

Figure 4 TPC-C transaction throughputs under low contention
To better understand the causes of BCC’s slight overhead compared to OCC, we further break down the slowdown of BCC at 32 threads and show how different kinds of operations in BCC contribute to the throughput degradation. The result is listed in Table 1. It can be seen that the overhead mainly comes from two sources: memory management (Mm), which lowers the throughput by a delta of 4.76%, and accessing global clocks (Clock), which contributes 2.42% to the performance degradation. When the data contention is low, memory management operations mainly include bookkeeping the history list, and allocating and releasing memory for history hash tables. In our implementation each database thread uses a small memory region residing on local NUMA node to store the history list and hash tables. In this case no inter-core communication is needed for memory management operations. Since most OLTP transactions are short, the overhead due to memory management should remain almost constant regardless of the number of cores on the target platform.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Contributions to slowdown (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mm</td>
<td>4.76</td>
</tr>
<tr>
<td>Clock</td>
<td>2.42</td>
</tr>
<tr>
<td>Others</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 1 Breakdown of BCC overhead for TPC-C at low contention with 32 threads

On the other hand, the overhead of accessing the global clock is affected by the number of sockets on the machine. This is because the global clock in BCC is implemented as a distributed vector spread among the sockets. Each database thread needs to read all the distributed vectors at the beginning of a transaction, incurring inter-
socket communication. This overhead is mainly determined by the number of sockets in the machine. However, since the number of sockets in a system is typically small, we believe this overhead (only 2.42% with 4 sockets) is acceptable in practice.

Compared to 2PL, BCC achieves better performance and scalability. With 32 threads, 2PL only delivers a throughput of 0.92M transactions per second, which is 19.9% and 25.8% lower than BCC and OCC respectively. In general, 2PL introduces extra overheads in two aspects. First, 2PL incurs extra locking operations for read operations compared to BCC and OCC. Each read operation needs to acquire and release a latch-protected read lock. Second, 2PL needs a centralized timestamp allocator to accurately determine the order of transactions to avoid deadlock, which becomes a bottleneck with the increase of number of the threads.

2.6.2 High Contention

Next we compare the performance of BCC with OCC and 2PL when data contention is high. A modified TPC-W [10], TPC-C and YCSB [25] are used in the evaluation.

TPC-W

TPC-W is a popular OLTP benchmark simulating the operations of an online bookseller. Compared with TPC-C, TPC-W has more complex read-only transactions. Since read-only transactions are executed as snapshot transactions which Silo never aborts with either OCC or BCC, their performance is similar under both concurrency control methods. We thus exclude them from our experiments with TPC-W, otherwise they would dominate the measured system throughput.
We experiment with the two update-intensive transactions from the TPC-W benchmark: (1) *DoCart* adds a set of random items to the shopping cart and displays the cart; (2) *OrderProcess* processes a set of random orders and updates the database (e.g., updating the stock numbers of the ordered items). To simulate the high contention scenario, we use slightly modified versions of the two transactions: there is one hot item in the orders processed by each OrderProcess transaction, and each *DoCart* transaction has a certain probability to display the hot item. In all our experiments, we let one database thread execute the *OrderProcess* transactions, while all other threads execute the *DoCart* transaction.

In our first experiment, we evaluate the performance of BCC, OCC and 2PL when *DoCart* transaction has the highest contention probability with *OrderProcess* transaction. We set the probability of DoCart adding the hot item to 100%, and measure transaction throughputs as the number of threads varies. The result is presented in Figure 5.

![Figure 5 Throughput of TPC-W as the per-thread contention probability is 100%](image-url)
It can be seen that BCC scales much better than OCC in this experiment. As the number of threads increases, BCC gains increasingly higher performance advantage. With 32 threads, BCC achieves a throughput of 1.03M transactions per second, which is 3.68x over the throughput with OCC (0.28M).

The performance improvement achieved by BCC over OCC is mainly attributed to the reduction of false-aborted DoCart transactions. We can understand this conclusion from the following observations. First, there can be no data dependencies between two DoCart transactions because each DoCart only modifies its own private shopping cart. Second, the hot item displayed (read) by a DoCart transaction has a high probability of having been modified by an OrderProcess transaction when the DoCart transaction tries to commit. In this case, OCC must abort the DoCart transaction due to the appearance of an anti-dependency on a committed transaction. However, it is actually unlikely that a data dependency cycle would form because the rest tuple accesses in both DoCart and OrderProcess are random, making it a false abort to abort the DoCart transaction. BCC effectively reduces such false aborts by checking for one more data dependency besides the anti-dependency.

2PL performs differently. As can be seen, 2PL’s throughput goes through three stages: decrease-increase-decrease. When using 2PL, OrderProcess’s throughput decreases as the number of threads increases because the hot tuple is more likely to be read locked by DoCart, which blocks OrderProcess. The reason for the decrease of 2PL’s throughput when the number of threads increases from 1 to 2 is that the decreased
throughput of \textit{OrderProcess} is larger than the added throughput of \textit{DoCart}. Note that 2PL’s throughput with one database thread is \textit{OrderProcess}’s throughput. As the number of threads increases from 2 to 16, 2PL’s throughput increases. The reason is that \textit{DoCart}’s throughput has increased and it outweighs the decrease of \textit{OrderProcess}’s throughput. As the number of threads further increases, both \textit{DoCart}’s throughput and \textit{OrderProcess}’s throughput decrease because of the high contention. Thus 2PL’s throughput decreases.

Compared to 2PL, BCC doesn’t perform as well as 2PL when the number of threads is 4 or less. However, as the number of threads increases, BCC significantly outperforms 2PL. With 32 threads, BCC’s throughput is 2.03x over that of 2PL. The performance differences are mainly determined by the relationship between 2PL’s synchronization overhead and BCC’s overhead of detecting essential patterns. With 2PL, the workloads are dominated by \textit{DoCart} transactions when the number of threads is 32. 2PL has to synchronize between different threads because each one tries to add a read-lock to the hot tuple. In an optimized in-memory database, the synchronization cost is non-trivial. On the other hand, BCC’s validation doesn’t incur synchronizations for read contention.

The above experiment demonstrates how BCC performs under the highest intensity of per-thread contention. To understand how different contention intensity affects the transaction throughput, we fix the number of threads to 32 and vary the probability that \textit{DoCart} adds the hot item to shopping cart. When the probability is 100%, it is the same with the previous experiment at 32 threads and the contention reaches the highest. When
the probability is 0%, all items in a *DoCart* transaction are randomly chosen and the contention is the lowest. The result is shown in Figure 6.

![Figure 6: Throughput of TPC-W per-thread contention possibility varies](image)

Compared to OCC, BCC performs slightly lower when the contention probability is less than 10%, (by up to 7.95%) due to the overhead of shared memory management and inter-socket communication incurred by accessing the global clock. With the increase of contention probability, the throughput of OCC drops sharply, bottoming at only 285k transactions per second when the probability reaches 100%. On the other hand, BCC’s throughput decreases at a much a slower rate. When the contention probability is 20% or greater, BCC’s benefit of reducing false aborts outweighs its overhead for detecting the essential patterns, which improves the overall throughput.

We can see that 2PL has a similar performance trend with OCC, although 2PL performs better than OCC. When the contention probability is less than 40%, 2PL has
comparable performance with BCC. However, as the contention probability continuously increases, the performance of 2PL decreases much faster as that of BCC. This is because of 2PL’s higher synchronization cost as we have discussed previously.

**TPC-C**

In the TPC-C experiments we use the update-intensive transactions, NewOrder and Payment, which comprises of most transactions in the TPC-C benchmark. We set the scale factor to 2 and the workload mix executed by each thread to \{50\%, 50\%\}. Figure 7 shows the throughput of this TPC-C workload as we increase the number of threads.

![Figure 7 Throughput of TPC-C NewOrder and Payment with a mix of 50%-50%](image)

It can be seen that BCC outperforms OCC when the number of threads exceeds the number of warehouses. With up to 16 threads, the throughput of BCC and OCC both increase with the increase of thread number, but BCC scales better than OCC. BCC improves the throughput by 37\% over OCC with 16 threads. As the number of threads
further increases beyond 16 threads, the performance of both BCC and OCC start degrading with similar trends, but BCC still maintains good performance improvement (up to 35.8%) above OCC. This again confirms BCC’s advantage over OCC through reducing false aborts.

The transactions NewOrder and Payment in TPC-C have much more complex data dependency patterns than the transactions DoCart and OrderProcess in TPC-W do. When operating on the same warehouse, all types of data dependencies can happen between any two concurrent transactions, each of which can be either NewOrder or Payment. Thus it is possible that a cycle would be created in the transaction dependency graph. For example, when two threads are executing the Payment transactions on the same warehouse, they both need to read and update the year-to-date payment, a dependency cycle containing rw and wr dependencies would likely be formed and thus one of the Payment transaction will be aborted by both BCC and OCC. This explains the performance decrease of both BCC and OCC when the contention becomes severe (with more than 16 threads).

BCC also performs better than 2PL. The throughput is improved by up to 1.84x. The poor performance of 2PL is mainly caused by its high synchronization overhead and the lock thrashing behavior. For example, when multiple database threads are executing Payment on the same warehouse, they need to acquire both read lock and write lock on the contended tuple. It is likely that the tuple is read locked by multiple threads thus only one thread can wait for the write lock while the rest are aborted. These aborted transactions cause unnecessary synchronization for others which limits the number of
transactions processed to the database. We observe that with 32 threads, the total number of transaction throughput processed by 2PL (including both committed and aborted transactions) is 0.31M per second, which is significantly lower than that of BCC (0.9M).

To better understand BCC’s performance improvement over OCC and 2PL on TPC-C, we break down the overall throughput by the numbers contributed by different transaction types. The results are shown in Figure 8.

![Figure 8 Breakdown of TPC-C throughput by transaction types](image)

OCC, BCC and 2PL perform differently for NewOrder and Payment. OCC favors Payment transactions over NewOrder transactions while 2PL commits much more NewOrder transactions than Payment transactions. BCC’s performance for NewOrder and Payment lie between.

Compared to OCC, BCC’s performance advantage comes from the improved throughput of the NewOrder transaction. The reason is that many of the rw dependencies
that happen between NewOrder and Payment that do not actually form a dependency circle, thus suffering false aborts with OCC. Examining additional dependency can greatly avoid the aborts and thus improve the overall throughput. However, BCC cannot improve the throughput of Payment transactions. It performs even worse than OCC. This is because each OCC-aborted Payment transaction is likely to reside in a dependency cycle with another transaction of the same type (i.e., true abort). Therefore, the OCC-aborted Payment transactions will also be aborted by BCC. In this case, BCC’s effort of examining additional dependencies only increases transaction execution latency, which in turn degrades the overall throughput.

On the other hand, BCC outperforms 2PL because it performs much better on the Payment transactions. With 32 threads, 2PL can barely commit Payment transactions. 2PL’s poor performance on Payment is caused by the following two reasons. First, there is read write contention between NewOrder and Payment when they operate on the same warehouse. When the contended tuple is read locked by a NewOrder transaction, other NewOrder transactions can continue adding read locks to the tuple while Payment transaction has to wait. In this case, Payment transaction is likely to be aborted to avoid deadlock. Second, the contention two Payment transactions on the same aborts of the Payment transactions because they create dependency cycle.

**YCSB**

YCSB benchmark contains a single table with ten String columns and populated with one million data items. Each transaction randomly accesses 16 tuples with each one having a 20% probability of being an update. Accesses to the tuples follow a Zipfian
distribution. We set the conflict factor $\theta$ to 100 to make the level of data contention high. In this case, all types of data dependencies can happen between two transactions. The results are shown in Figure 9.

As can be seen from Figure 9, BCC performs better than both OCC and 2PL when the number of thread is 8 or greater. With 32 threads, BCC’s throughput is 1.99x over that of OCC and 1.63x over that of 2PL. The reason for the different performance behaviors is similar to the previous high contention benchmarks. BCC’s performance improvement over OCC comes from BCC’s reduction of unnecessary transaction aborts. On the other hand, BCC out-performs 2PL because of 2PL’s high synchronization cost and lock thrashing behaviors.

![Figure 9 YCSB throughput under high contention](image-url)
2.6.3 Memory Consumption and Latency

BCC improves transaction throughput through detecting the essential patterns, with shared memory usage and extra operations. In this part we illustrate BCC’s memory consumption and its impact on transaction latency.

Memory Consumption

With BCC, each thread maintains a memory area to store (1) a list of entries for recent history transactions, and (2) the hash tables of these transactions needed for detecting the essential patterns.

Figure 10 BCC memory consumption for saving history hash table per-thread with 50%-50% workload of TPC-C NewOrder and Payment

The size of saved history hash tables determines the memory consumption of BCC. For a given workload with a fixed number of threads, this overhead is usually stably low. Figure 10 shows the average and maximum sizes of memory occupied by history hash tables in each thread, executing the TPC-C NewOrder and Payment workload mix used in
the previous subsection. It can be seen that the average per-thread memory consumption stays below 56KB consistently across all thread counts, with the maximum memory usage not exceeding 1.6MB. The high variance of the memory consumption between the average and the maximum is caused by the conservative hash table release mechanism, which achieves good performance but relies on the process of all database threads to determine when a hash table can be released. When a transaction $T$’s hash table is released, $T$’s concurrent transactions may have already finished for some time. Similar results are observed with other workloads used in our experiments as well.

In our experiments we set the number of entries in the history transaction list to 16K and the total size of memory for storing history hash tables to 4MB, which are more than enough for all the workloads encountered in our experiments. This amounts the total memory consumption of BCC with each thread to about 4.38MB, which is negligible compared with the tens of hundreds of giga-bytes of memory present on a typical enterprise server. This also justifies BCC’s design of using one hash table per transaction for the benefit of performance with low-contention workloads.

**Latency**

To illustrate how BCC affects transaction execution latency, we divide the *NewOrder* transactions processed with BCC in the previous experiment into the following three categories: (1) transactions that are committed with OCC’s validation criterion; (2) transactions that are aborted even after BCC checks; and (3) transactions that are aborted with OCC but committed with BCC. These three types of transactions do not overlap.
For the first type of transactions, BCC’s overhead mainly includes memory management and accessing the global clock, which are similar as the low contention workloads we discussed earlier in Chapter 2.6.1. For the second type of transactions, they are aborted by both OCC and BCC. Besides memory management and global clock operations, BCC performs extra data dependency checking before aborting a transaction. Figure 11 shows the total latency of each aborted NewOrder transaction in this case. BCC increases the latency by up to 26% with 32 threads. However, since the database needs to cleanup an aborted transaction for re-execution, this makes BCC’s overhead negligible.

![Figure 11 Latency of NewOrder transactions aborted by both OCC and BCC](image)

For the third type of transactions, BCC commits a transaction that would otherwise be aborted by OCC (i.e., BCC saves a transaction). This comes at the overhead of increased latency because the database thread needs to validate the transaction’s write set with the history hash tables on other threads. Figure 12 shows the latency of transactions in this
type, compared with transactions that OCC commits. As can be seen, the latency of
transaction saved by BCC is almost twice as that committed by OCC. However, this
overhead is acceptable for two main reasons. First, in high contention scenario, an OCC-
aborted transaction may be aborted several times before it can actually commit.
Considering the high cost of transaction re-executions, the latency of BCC-saved
transactions is often justified. Second, the increased latency is still within tens of
microseconds, which is sufficiently small for most real-world applications. We thus
believe it is reasonable to trade the small increasing of latency for the high improvement
of transaction throughput.

Figure 12 Latency of NewOrder transactions saved by BCC compared with that
committed by OCC.
2.7 Related Work

Partial dependency detection

The idea of detecting partial graph cycles to guarantee serializability was first proposed in the snapshot isolation concurrency control method [21, 33]. Snapshot isolation (SI) has been implemented in major database systems, such as Oracle and PostgreSQL. SI guarantees that read and write transactions won’t block each other to increase system throughput. However, Fekete et al. [34] showed that SI could not guarantee transaction serializability, and Fekete et al. [33] further found a data dependency pattern (dangerous structure) that will always happen when transactions cannot be serialized in snapshot isolation (SI). Cahill et al. [21] demonstrated how to implement the dangerous structure in Berkeley DB. Ports et al. [73] further optimized this method for PostgreSQL. Han et al. [40] further optimized SI for multicore systems. BCC’s essential patterns contain different data dependencies compared to the dangerous structure, which is caused by different record visibilities between SI and the optimistic concurrency control model. In SI a transaction cannot see writes which happen after the transaction starts. However, in BCC, any data dependency may exist between concurrent transactions. Moreover, BCC is designed and optimized for the short latencies that are encountered in main-memory OLTP workloads and not for disk-based implementations.

OCC for in-memory databases

Concurrency control method guarantees transaction serializability and has significant impacts on database performance and scalability. The optimistic concurrency control (OCC) method was originally proposed by Kung and Robinson [55] and has been
implemented in recent in-memory databases [56, 85, 89, 92]. These works mainly study how to efficiently implement the OCC method in in-memory databases. Larson et al. [56] proposed OCC for Microsoft SQL Server’s in-memory OLTP engine “Hekaton” [27] and compared the performance of OCC with two-phase locking. Their implementation of OCC used a centralized timestamp allocator. Silo [85] introduced an implementation of OCC without centralized bottlenecks which can achieve near-linear scalability for low contention workloads. Tran et al. [84] studied transaction behaviors on hardware transactional memory. Wang et al. [89] explored how to build high performance OCC for in-memory databases with hardware transactional memory. Yu et al. [92] studied the scalability of different concurrency control methods on up to 1024 cores with a simulator.

**OCC for distributed systems**

Recently OCC has also been studied in distributed systems. Maat [61] re-designed OCC for distributed systems and removed the need of locking during two-phase commit. ROCOCO [63] broke transactions into atomic pieces and executed them out of order by tracking dependencies, which significantly outperformed OCC. In comparison, BCC focuses on transaction execution for single-node in-memory databases.

**OLTP on modern hardware**

Our design and implementation benefits from existing in-memory OLTP systems. Databases such as Hyper [54] and H-Store [42] adopt the partitioning approach to scale. Harizopoulos et al. [42] analyzed the overheads of the Shore database. Pandis et al. [70] eliminated the overhead of centralized lock manager with partitioning. Porobic et al. [72] systematically compared the performance of shared-nothing and shared-everything OLTP
system designs on multi-socket, multi-core CPUs. Faleiro et al. [31] redesigned the multiversion concurrency control method for in-memory databases by avoiding bookkeeping operations for read and global timestamp allocator, but it requires all the transactions to be submitted to the database before they can be processed.

Doppel [65] introduced an in-memory database designed for transactions that contend on the same data item. It proposed splitting the contended data item across cores such that each core can continue updating the data item in parallel. The per-core value was reconciled before the data item can be read. Doppel’s optimization is orthogonal to BCC: Doppel improves performance when \(ww\) dependencies happen, while BCC avoids false aborts caused by \(rw\) dependencies.

2.8 Summary

In this chapter we have presented the Balanced Concurrency Control (BCC) method for in-memory databases. Unlike OCC that aborts a transaction based on whether the transaction’s read set has changed, BCC aborts transactions based on the detection of essential patterns that will always appear in unserializable transaction schedules. We implemented BCC in Silo, a representative OCC-based in-memory database and comprehensively compared BCC with OCC and 2PL with TPC-W-like, TPC-C-like and YCSB benchmarks. Our performance evaluations demonstrate that BCC outperforms OCC by more than 3x and 2PL by more than 2x when data contention is high; meanwhile, BCC has comparable performance to OCC in low-contention workloads.
Chapter 3 The Yin and Yang of Processing Data Warehousing Queries on GPUs

3.1 Introduction

In the past decade, special-purpose graphic computing units (GPUs) originally designed for computer entertainment applications have evolved into general-purpose computing devices, with the advent of efficient parallel programming models, such as CUDA [4] and OpenCL [7]. 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 [19, 37, 45, 71, 36, 78, 32, 79, 46, 44, 52, 16, 91, 60, 88]. 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 and Pig) that has truly 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?
- How do existing software optimization techniques affect query performance on GPUs?
- Under what conditions will GPU significantly outperform CPU for warehousing queries?
- How do different GPU hardware and their supporting systems affect the query performance?
- How does the advancement of GPU hardware affect query performance?

3.1.1 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*, are affected by query characteristics, software optimization techniques, and hardware environments. In ancient Chinese philosophy, *Yin* and *Yang* represent two opposite forces that are interconnected and interdependent in the natural world. 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 13. We target at the star schema queries because they are the typical workloads in practical
warehousing systems [80]. Our study is based on the following three sets of research efforts.

![Figure 13 A 3-Dimension Study of Processing Warehousing Queries on GPUs](image)

**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 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 [50], which is a
representative high-performance analytical query engine; and 3) we investigated how different GPU hardware 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 hardware will improve query performance.

**3.1.2 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 due to their memory-intensive nature.

The rest of the chapter is organized as follows. In Chapter 3.2 we present the implementation of our GPU query engine. Chapter 3.3 describes the experimental environment. We study the warehousing query behaviors and the effects of software techniques in Chapter 3.4 and conduct detailed performance comparisons in Chapter 3.5.
In Chapter 3.6 we introduce our cost model and explore the impacts of GPU hardware advancement on query processing. We introduce the related work in Chapter 3.7 and summarize our work in Chapter 3.8.

3.2 GPU Query Engine

3.2.1 Engine Structure and Storage Format

Figure 14 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 [57]. 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 [11] and performs tuple re-construction 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.

### 3.2.2 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 GPU 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 [18, 20, 52]. We
implement the hash table using both Cuckoo hash [15] 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 [62]. 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 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.

**3.2.3 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.
3.3 Experimental Methodology

3.3.1 Workloads

We use the Star Schema Benchmark (SSBM) [67] which has already been widely used in various data warehousing research studies [13, 22]. 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 15. There is a total of 13 queries in the benchmark, divided into 4 query flights. In our experiments, we run the benchmark with a scale factor of 10 which will generate the fact table with 60 million tuples.

![SSBM Schema](image)

Figure 15 SSBM Schema

3.3.2 Hardware Platform

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.
<table>
<thead>
<tr>
<th>Processors</th>
<th># of Cores</th>
<th>GFLOPS</th>
<th>Bandwidth(GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA 480</td>
<td>480</td>
<td>1345</td>
<td>177.4</td>
</tr>
<tr>
<td>NVIDIA 580</td>
<td>512</td>
<td>1581.1</td>
<td>192.4</td>
</tr>
<tr>
<td>NVIDIA 680</td>
<td>1596</td>
<td>3090.4</td>
<td>192.256</td>
</tr>
<tr>
<td>AMD 7970</td>
<td>2048</td>
<td>3788.8</td>
<td>264</td>
</tr>
<tr>
<td>Intel Core i7</td>
<td>4</td>
<td>112</td>
<td>25.6</td>
</tr>
</tbody>
</table>

Table 2 Hardware Specifications

3.3.3 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.

3.3.4 Measurement 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.
### 3.3.5 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 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 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].

<table>
<thead>
<tr>
<th></th>
<th>480</th>
<th>580</th>
<th>680</th>
<th>7970</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read(GB/s)</strong></td>
<td>114.59</td>
<td>129.95</td>
<td>127.65</td>
<td>202.76</td>
</tr>
<tr>
<td><strong>Write(GB/s)</strong></td>
<td>138.34</td>
<td>150.41</td>
<td>153.43</td>
<td>116.44</td>
</tr>
<tr>
<td>HtoD pageable(GB/s)</td>
<td>6.30</td>
<td>6.30</td>
<td>6.30</td>
<td>9.80</td>
</tr>
<tr>
<td>HtoD pinned(GB/s)</td>
<td>6.65</td>
<td>6.65</td>
<td>12.28</td>
<td>11.13</td>
</tr>
<tr>
<td>DtoH pageable(GB/s)</td>
<td>6.20</td>
<td>6.20</td>
<td>6.22</td>
<td>9.19</td>
</tr>
<tr>
<td>DtoH pinned(GB/s)</td>
<td>6.64</td>
<td>6.64</td>
<td>12.75</td>
<td>11.81</td>
</tr>
</tbody>
</table>

Table 3 GPU bandwidth measurement
3.4 Performance Analysis

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

3.4.1 SSBM Query behaviors

Figure 16 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 re- leasing GPU device memory. As is shown in Figure 16, most execution time for SSBM queries are spent on PCIe transfer and kernel execution. To understand the query behaviors, we further break- down the execution time and shows the percentage of the major operations for SSBM queries in Figure 17.

![Figure 16 Baseline of SSBM queries on NVIDIA GTX 680](image-url)
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 18. 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 16. 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 16.

**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 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
    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.

**3.4.2 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. All these schemes can achieve an effective compression ratio without incurring high computation costs.

**Run Length Encoding**

We apply RLE to sorted columns. The elements, which are stored in continuous positions in the column with the same value, are replaced with a tuple (value, count) where value represents the repeated value and count represents the number of elements being replaced. A header is added to the compressed column to indicate the compression type and the number of distinct values in the column.
Bit Encoding

The Bit Encoding scheme tries to use the least number of bits to represent a value. When using Bit Encoding, we first identify the largest number of bits needed to represent a value in the column. Then all the values in the column will be stored using this number of bits. Bit Encoding is usually used with Dictionary Encoding compression scheme in our engine.

Dictionary Encoding

We apply Dictionary Encoding to columns that only have a limited number of distinct values. To compress a column using Dictionary Encoding scheme, we first find out all the distinct values in the column, which are stored in an array at the beginning of the column. Then each value in the column is replaced with the value’s index in the array. Since the column usually has a limited number of distinct values, we can further apply the Bit Encoding scheme to achieve a better compression ratio.

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.
<table>
<thead>
<tr>
<th>Column</th>
<th>lo_custkey</th>
<th>lo_partkey</th>
<th>lo_suppkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>lo_custkey</td>
<td>1%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>lo_partkey</td>
<td>100%</td>
<td>3%</td>
<td>100%</td>
</tr>
<tr>
<td>lo_suppkey</td>
<td>50%</td>
<td>50%</td>
<td>0.1%</td>
</tr>
<tr>
<td>lo_orderdate</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>lo_extendedprice</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>lo_quantity</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>lo_discount</td>
<td>25%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>lo_revenue</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>lo_supplycost</td>
<td>50%</td>
<td>50%</td>
<td>50%</td>
</tr>
</tbody>
</table>

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

The query engines can obtain significant performance benefits when directly working on the compressed data [12]. 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 18 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.

### 3.4.3 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 19. 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. 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.

Figure 19 Speedup of transfer overlapping
3.4.4 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 [13] 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 20 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.

3.5 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 support 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?
- Which programming model is more suitable and more supportive for programming warehousing queries on GPU, CUDA or OpenCL?
- How do different GPU hardware and their supporting systems affect query performance when their basic hardware parameters are similar?
- With the functional portability of OpenCL, how will the OpenCL query engine that is designed for GPU perform compared to MonetDB?

3.5.1 Comparisons of GPU and CPU

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 20 shows the execution time of SSBM
queries and Figure 21 shows the performance speedup of GPU over CPU.

3.5.1.1 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 21, 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 21. 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 21. 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.
3.5.1.2 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 21, 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.

3.5.2 Impact of Programming Models and GPU Hardware

3.5.2.1 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.

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 [6], 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.

![Figure 22 Normalized OpenCL performance over CUDA](image)

As is shown in Figure 22, 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 [29]. 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 [29] doesn’t apply to warehousing workloads.

3.5.2.2 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 AMD HD 7970. Both engines have been optimized with the data compression technique and the transfer overlapping technique. The results are shown in Figure 22.

![Figure 23 NVIDIA Versus AMD](image)
As can be seen in Figure 23, 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 24.

![Figure 24 Normalized SSBM performance on AMD GPU](image)

As is shown in Figure 24, these two GPUs have comparable performance for PCIe data transfer and kernel execution. Since these two GPUs have comparable hardware, we expect that they should have similar performance for SSBM queries. Considering the “Other” part, 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 recorded 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.

3.5.2.3 Comparisons of OpenCL query engine on GPU with MonetDB

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 25.

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 25. 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.

3.6 Model and Prediction

3.6.1 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. Next, we model the join operator of our query engine as a case study for our methodology.
3.6.2 Cost Model for Join

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.

The notations related to join query are listed in Table 5.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Join selectivity</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>Number of projected columns from fact table</td>
</tr>
<tr>
<td>m</td>
<td>Number of projected columns from dimension table</td>
</tr>
<tr>
<td>R_i</td>
<td>The attribute size of the ith projected column from fact table</td>
</tr>
<tr>
<td>S_i</td>
<td>The attribute size of the ith projected column from dimension table</td>
</tr>
<tr>
<td>B_r</td>
<td>The read bandwidth of the device memory</td>
</tr>
<tr>
<td>B_w</td>
<td>The write bandwidth of the device memory</td>
</tr>
<tr>
<td>B_i</td>
<td>The transfer bandwidth from host memory to device memory</td>
</tr>
<tr>
<td>B_o</td>
<td>The transfer bandwidth from device memory to host memory</td>
</tr>
<tr>
<td>C_r</td>
<td>The read segment size of the device memory</td>
</tr>
<tr>
<td>C_w</td>
<td>The write segment size of the device memory</td>
</tr>
<tr>
<td>S_i</td>
<td>The size of input data</td>
</tr>
<tr>
<td>S_o</td>
<td>The size of result</td>
</tr>
<tr>
<td>W</td>
<td>The number threads in a thread group</td>
</tr>
<tr>
<td>T_i</td>
<td>The device memory access cost of the ith step to finish the query on GPU</td>
</tr>
<tr>
<td>T_r</td>
<td>The data transfer cost</td>
</tr>
</tbody>
</table>

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:
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:

\[
T_2 = \left( \left| R \right| \times \left[ 4 \times \frac{W}{C_r} \right] + \left| R \right| \times 3 \times \left| R \right| \times r \times \left[ \frac{4}{C_r} \right] \right) \times \frac{C_r}{B_r} + \left( \frac{4}{C_w} \right) \times \left| \frac{R}{W} \right| \times \frac{C_w}{B_w}
\]

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

\[
T_3 = \sum_{i=1}^{n} \left( 4 \times \frac{W}{C_r} \times \left| R \right| \times \left[ \frac{4}{C_r} \right] \right) + \sum_{i=1}^{m} \left[ \frac{S_i}{4} \right] \times \frac{C_r}{B_r}
\]

\[
+ \left( \sum_{i=1}^{m} \left[ \frac{R_i}{4} \right] \times \left| \frac{R}{W} \right| \times \left[ \frac{4}{C_r} \right] \right) \times \frac{C_r}{B_r}
\]

\[
+ \left| \frac{R}{W} \right| \times r \times \left( \sum_{i=1}^{n} \left[ \frac{R_i}{4} \right] + \sum_{i=1}^{m} \left[ \frac{S_i}{4} \right] \right) \times \frac{C_w}{B_w}
\]

The input data size and the result size can be calculated as:
\[ S_i = ||R|| \times \sum_{i=1}^{n} R_i + ||S|| \times \sum_{i=1}^{m} S_i + 4 \times (||R|| + ||S||) \]

\[ S_0 = (||R|| \times \sum_{i=1}^{n} R_i + ||R|| \times \sum_{i=1}^{m} S_i) \times r \]

The data transfer cost can be calculated as:

\[ T_t = \frac{S_i}{B_i} + \frac{S_o}{B_o} \]

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

\[ error\_rate = \frac{measured\_time - estimated\_time}{measured\_time} \]

Figure 26 and Figure 27 present the evaluation results. As shown in the figures, the estimated performance is 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 26.

Figure 26 Evaluate join model for different query characteristics
(a) Selectivity (%)  
(b) # of fact columns
(c) # of dim columns  
(d) Dim column width
3.6.4 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 28. 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 is mainly bounded by GPU device memory accesses. They cannot benefit much from the increased computational power.
To predict the possible impact of GPU hardware advancements 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.
Figure 29 Estimated SSBM performance with different GPU hardware configurations

The result is shown in Figure 29. 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 hardware.

3.7 Related Work

There are already a set of research papers on optimizing various database operations on GPUs [19, 37, 45, 71, 36, 78, 32, 79, 46, 44, 52, 88, 16, 91, 60]. 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.
compression on GPU has been studied in [32], and transfer overlapping has been studied in [52, 71]. 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 [44]. 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 [5]. Our model takes a hardware feature oriented methodology to estimate the actual memory transactions in GPU device memory, which can better estimate the GPU memory bus utilization.

3.8 Summary

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 hardware.
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 maximize 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.
Chapter 4 Spark-GPU: An Accelerated In-Memory Data Processing Engine on Clusters

4.1 Introduction

R&D for data processing to handle increasingly big volumes of data has been rapidly advanced in two stages. In the first stage, scalable systems based on scale-out models have been developed. One such system is Hadoop [2], which is an open source implementation of MapReduce [26]. Several widely-used data processing systems have been built on top of Hadoop. We have entered the second stage of R&D striving for high performance in data processing. The efforts mainly come from best utilizing advanced and low cost commodity devices, such as multicores, GPUs, and SSDs. In this stage, one of the most attractive approaches to improve performance is in-memory computing. With the increase of DRAM capacity and drop of its price, more and more application’s data sets can be fit into a cluster’s memory. Utilizing main memory to improve the performance of data analytics applications have become desirable and feasible.

Apache Spark [3] is a representative open source, distributed in-memory data processing system, has gained popularity for its improved performance over the widely-used MapReduce Hadoop. It not only supports executing SQL queries over structured data [17], but also provides procedural processing ability for advanced data analytics such as machine learning and graph applications [35]. With in-memory data processing,
the performance bottleneck of data analytic applications will shift from disk I/O and network to computations, as is the case for Spark [68]. As a result, it becomes critical to well utilize the various computing resources (e.g., multi-core CPUs and GPUs) in modern clusters to further improve the performance of data analytics applications.

We have looked into the specific computing demands for Spark, which are characterized by two unique execution patterns. First, data analytics applications running on Spark have rich data parallelism. Spark abstracts data into Resilient Distributed Datasets (RDDs) [94, 95], and analytic applications on Spark are built with a set of operations on the RDDs. Each RDD operation applies to all the data in the RDD. Second, many data analytics applications are compute-intensive. Complex, iterative computations are conducted in analytic applications such as machine learning. These two patterns make GPU a potential computing device to accelerate Spark’s performance.

GPU is a massive parallel computing device with high computational power and memory bandwidth, which are suitable for data-parallel, compute-intensive applications. The research community has extensively studied how to utilize GPUs to accelerate various data analytic applications, including SQL queries [45, 93], NoSQL operations [96, 48], machine learning [24] and graph applications [41]. The performance of these applications can be significantly improved with GPUs. However, many of these work adopt a GPU-centric design, which redesigns the system based on GPU’s characteristics without considering the performance of CPU operations. Apache Spark is a CPU-optimized data processing system. It is unclear how much it can benefit from the high performance GPUs considering the different types of optimizations for CPU and GPU.
In this Chapter, we use Spark as a platform to explore how to use GPUs to accelerate the performance of various analytic applications on CPU-optimized distributed in-memory data processing systems. Specifically, we have designed and implemented Spark-GPU, a CPU-GPU hybrid data analytics system that can not only run SQL queries but also various complex analytic applications on both CPUs and GPUs. We present a set of designs that effectively connect GPUs to Spark to best utilize GPU’s capability. Spark-GPU uses heuristic rules to offload SQL queries to GPUs and provides block processing ability for GPUs to get the best performance of analytic applications. Our comprehensive evaluation shows that Spark-GPU can have up to 4.83x performance speedup for SQL queries, and have up to 16.13x performance speedup for compute-intensive machine learning applications.

The major contributions we have made are as follows.

- We have identified and analyzed the challenges for effectively using GPUs in distributed in-memory data processing systems.
- We have designed and implemented Spark-GPU, which best utilizes GPU’s capability with reasonable changes in Spark.
- We have comprehensively evaluated the system’s performance with various representative workloads and illustrate the pros and cons of using GPUs in Spark.
- We provide an efficient methodology to integrate GPUs into in-memory data processing systems.

The rest of the Chapter is organized as follows. Chapter 4.2 describes the overall architecture of Spark-GPU. Chapter 4.3, Chapter 4.4 and Chapter 4.5 describe the
detailed designs of Spark-GPU. Chapter 4.6 presents the evaluation results of Spark-GPU. After introducing the related work in Chapter 4.7, Chapter 4.8 summarizes the work.

4.2 Spark-GPU Overview

4.2.1 Challenges

Spark abstracts data into Resilient Distributed Datasets (RDDs) [94], and analytic applications on Spark are built with a set of operations on the RDDs. Each RDD operation applies to all the data in the RDD. In this case, analytic applications running on Spark usually have rich data parallelism, which naturally matches GPU’s parallel architecture. However, due to the unique properties of Spark and GPU, it is a non-trivial task to efficiently use GPUs in Spark. In order to make GPUs well handle data analytic applications running on Spark, the following challenges must be addressed.

- First, Spark uses the iterator model [38] to execute applications. Each RDD in Spark implements an iterator interface, which computes and returns one element of the RDD when it is called. The iterator model has advantages such as simplicity and flexibility, however, it doesn’t match GPU’s architecture and can significantly underutilize GPU resources. To maximize GPU’s performance, the system must support block processing and convert the data into GPU-friendly format before processing on GPUs. These operations can introduce expensive data copying operations, which must be minimized in the system design.

- Second, Spark is implemented in a managed language (i.e. Scala) and runs on top of a Java Virtual Machine (JVM) with automatic memory management and
garbage collection. Data in Spark are represented as Java/Scala objects and are stored on the heap memory of JVM. However, GPU programs are usually implemented with GPU programming models such as CUDA [4] and OpenCL [7], which cannot access data stored in Java heap memory. As a result, to offload computations to GPUs, data must be frequently copied between Java heap memory and native memory. Data copying is expensive. A software mechanism to minimize data copying between Java heap memory and native memory must be developed in order to gain high performance from GPUs.

• Third, existing cluster resource managers such as Yarn [86] and Mesos [49] manage GPUs in a coarse-grained way, which exclusively assigns a GPU to a task. This underutilizes GPU resources. The major challenge to share GPUs is that GPU doesn’t have the same level of operating system support as CPU does. Operating system doesn’t provide virtual memory management for GPU device memory. Instead, each GPU program manages GPU device memory itself. In this case, when multiple GPU programs are running concurrently on GPUs, they may crash due to insufficient GPU device memory.

4.2.2 An Overview of Spark-GPU

Spark-GPU handles all the aforementioned challenges to efficiently execute analytic applications on GPUs. Figure 30 shows Spark-GPU’s architecture. Several components in Spark have been modified.
First, Spark-GPU extends Spark’s iterator model to support block processing on GPUs, which can better exploit GPU’s massive parallelism and high memory bandwidth. Chapter 4.3 introduces how to support and use block processing on top of Spark’s iterator model.

Second, Spark-GPU extends Spark’s SQL module to offload SQL queries to GPUs. Spark-GPU introduces a set of high performance GPU query operators to Spark and extends its query optimizer to generate query plans with both CPU query operators and GPU query operators. Chapter 4.4 explains how SQL queries are processed on GPUs.

Third, in order to efficiently execute data analytic applications on GPUs, Spark-GPU extends Spark’s cluster manager and task scheduler to manage GPUs in the cluster. All scheduling decisions in Spark are based on operations in data analytic applications. Each operation takes an RDD as input and outputs a new RDD. Spark maintains a lineage
graph of RDDs and divides the graph into one or more stages. Each stage is a unit of execution and will be executed by a set of tasks. Spark-GPU manages GPU resources and schedules GPU tasks to GPU nodes. Chapter 4.5 presents how Spark-GPU manages GPU resources and schedules tasks.

4.3 Execution Model and Data Format

GPU is a massively parallel co-processor, which executes GPU kernels in a Single Instruction Multiple Threads (SIMT) way. To maximize GPU’s performance, two requirements must be met. First, each GPU kernel should be launched with a large number of GPU threads, which can utilize GPU computing resources and hide GPU memory access latency to achieve high throughput. Second, data should be accessed in a coalesced manner to fully utilize GPU’s memory bandwidth [90].

To meet the first requirement, a system needs to support block processing model. To meet the second requirement, a system needs to organize data into a suitable format such that they can be accessed in a coalesced way. However, Spark doesn’t meet these two requirements. It adopts the iterator model and computes one element at a time using row format, which may significantly underutilize GPU resources. Completely rewriting Spark’s execution engine to integrate GPUs is unrealistic. In this case, to efficiently harness GPU resources, block processing and other data formats such as column format should co-exist with the iterator model and row format in the system.

4.3.1 Our Solution: GPU-RDD

Spark’s design is centered around RDD. To support block processing, we introduce a new type of RDD: GPU-RDD, which buffers all its data in either row format or column
format in native memory. Each GPU-RDD provides two interfaces to access its data: one is the standard RDD interface, which returns one element each time the interface is called; the other is a block interface, which returns the addresses of its buffered data. The standard interface makes it easy to integrate GPU-RDDs into existing Spark’s data flows. Moreover, the block interface provides applications the ability to apply a GPU-RDD operation to all its buffered data at one time, which can better utilize modern parallel computing devices (e.g., GPUs).

GPU-RDDs can be derived from existing RDDs or other GPU-RDDs. By default, data in GPU-RDDs are stored in column format since it may better utilize GPU’s memory bandwidth. However, applications can always choose row format when creating GPU-RDDs if it guarantees better performance.

Spark-GPU uses BlockRecords to represent data in a GPU-RDD. A BlockRecord corresponds to one partition of data in the RDD. It contains both buffered data and the corresponding metadata such as number of elements in the partition, data types, etc. The buffered data can be stored in either one continuous memory region (for both row format and column format) or different memory regions where only data in the same column are stored in a continuous memory (for column format only). The best way to arrange the data depends on computation patterns. For analytic applications such as SQL queries, it is common that only a small subset of the columns will be used in the computation. In this case, data can be stored in separate memory regions. On the other hand, if all the data are used in the computation or the data have large number of columns, they should be stored in one memory region such that data transfer overhead can be minimized.
Spark-GPU utilizes native memory instead of Java heap memory to buffer data in GPU-RDDs, which has two major advantages. First, it saves one data copying operation inside Java heap memory. Data in native memory can be directly transferred to GPUs to process. Second, it doesn’t increase the overhead of Java memory management. Large usage of Java heap memory leads to more frequent garbage collections, which can significantly degrade the system’s performance.

GPU-RDD’s native memory is released either by JVM at the time when the object is garbage collected or by applications that explicitly execute memory release function calls. Since RDDs are read-only (note that an operation on a RDD will create a new RDD) and GPU-RDDs buffer data, naively allocating memory for GPU-RDDs can pressure system memory usage, which may significantly degrade system’s performance when running out of memory. Spark- GPU optimizes native memory usage when there are several consecutive GPU-RDDs. Instead of buffering data for each GPU-RDD, Spark-GPU only keeps the native memory for the last GPU-RDD of the consecutive GPU-RDDs. All native memory used by other GPU-RDDs will be immediately released.

4.3.2 GPU Processing with GPU-RDDs

Operations on GPU-RDDs can be offloaded to GPUs. Spark-GPU supports several built-in GPU-RDD operations such as filter and map that are executed on GPUs. These built-in operations have data parallelism and are usually used in data preprocessing. To conduct more complex GPU-RDD operations on GPUs, users need to implement their own customized functions, each of which computes one partition of data in the RDD (represented by one BlockRecord).
Since GPU is usually programmed with CUDA or OpenCL while Spark is implemented in Scala, each GPU customized function must consist of a native function that is implemented in CUDA or OpenCL, and a Scala wrapper on top of the native function. The native function utilizes GPUs to implement the core functionality of the customized function. The Scala wrapper provides an interface that can be executed in Spark-GPU and interacts with the native function through Java Native Interface (JNI).

The efforts of developing customized functions for Spark-GPU are similar to that of developing a single node GPU program. Spark-GPU provides primitive GPU operations and system support for efficient usage of GPUs while the analytic applications determine whether they should use GPUs.

Using GPUs with GPU-RDDs is not free, because it introduces overheads of extra data copying operations between Java heap and native memory, and between native memory and GPU device memory. In general, the following operations are needed to offload a GPU-RDD operation to GPUs: (1) copy data from Java heap to native memory to create the GPU-RDD; (2) transfer data to GPU device memory; (3) compute on GPUs; (4) transfer results from GPU device memory to native memory; and (5) copy results from native memory to Java heap. Compared to directly executing the operation on the CPU, offloading it to GPUs introduce data copies illustrated in step (1), (2), (4) and (5). Note that step 5 only happens when the GPU-RDD’s standard data access interface is called.

Data copying is expensive and can degrade GPU’s performance advantage. To illustrate its overhead, we conducted a micro benchmark on Spark-GPU. The micro
benchmark simply created a GPU-RDD from a Spark RDD and then projected all the data by calling GPU-RDD’s standard data access interface. We executed the micro benchmark on a single node with 16 CPU cores and 32 GB memory. We set the total number of rows in the RDD to 2 million and set each column in the row to be a 16-byte string. The reason we used string type is that creating string object in Java heap is expensive. We varied the number of columns in the row and measured the execution time of the micro benchmark. For comparison, we also reported the execution time of directly projecting all the rows in Spark. The results are shown in Figure 31.

![Figure 31 The execution time of the micro benchmark with different number of string columns in a row](image)

The micro benchmark demonstrates the expensive overheads of data copying with GPU-RDDs in Spark-GPU. The overhead increases with the size of data. When there were 8 string columns in a row, the execution time of creating the GPU-RDD was 18.3x longer than projecting the rows in Spark, and the execution time of projecting all the data
in the GPU-RDD to rows was 10.5x longer. The reason for the high overhead is that a Java string object cannot be directly copied to the native memory. The string object must first be converted to a byte array and then copied byte by byte to the native memory, which is expensive. This indicates that if an operation doesn’t conduct much computation on each partition of data, it should not be offloaded to GPUs.

GPU’s performance advantage comes from its high parallelism and high memory bandwidth. To decide if an operation can benefit from GPUs, three factors should be considered: (1) whether the operation is compute-intensive; (2) whether the operation accesses the same data multiple times; and (3) whether there are multiple consecutive GPU operations on the data. If any of the three factors hold, the operation should be offloaded to GPUs. For example, K-Means’ operation of calculating point distance should be offloaded to GPUs for its compute-intensive nature.

4.4 Query Processing on GPUs

SQL queries are important analytic applications. A major difference between SQL queries and other analytic applications is that SQL queries only specify what to compute, not how to conduct the computation. It is the data processing system that determines the query execution logic. In this case, to efficiently execute SQL queries on GPUs, Spark-GPU implements a set of high performance GPU query operators, and extends Spark’s query optimizer to build query execution plan with both CPU query operators and GPU query operators. In this Chapter, we first introduce the design of GPU query operators in Spark-GPU. Then we present the GPU-aware query optimizer and describe optimization techniques to improve query performance on Spark-GPU.
4.4.1 GPU Query Operators

Spark-GPU has implemented five important GPU query operators: GPU scan, GPU broadcast join, GPU hash join, GPU aggregation and GPU sort, which can be used as the building blocks for a wide range of SQL queries. Similar to the customized GPU functions for GPU-RDDs, each GPU query operator consists of a Scala wrapper and a native function. The Scala wrapper implements an iterator interface that returns a data element in row format each time it is called. The native function processes columnar data on GPUs using standard GPU programming models.

**GPU Scan**

The GPU scan operator implements selection operation on in-memory data, which will return all data that satisfy some query predicates. To use the GPU scan operator, data must be either explicitly cached in Java heap memory or stored in native memory.

The major selection operation is computed on GPUs, which contains three steps: (1) evaluating query predicates; (2) calculating output position and (3) projecting the results. In the first step, query predicates are evaluated and a 0/1 vector is maintained to keep track of data that satisfy the predicates. In the second step, a prefix sum is calculated on the 0/1 vector to decide the start writing positions for GPU threads in the result buffer to avoid synchronizations when writing the results. In the last step, based on the 0/1 vector and prefix sum, data that satisfy the query predicates are generated.

**GPU Broadcast Join and Hash Join**

Join is one of the most important SQL operations. Currently Spark-GPU supports two kinds of inner join operators on GPUs: GPU broadcast join and GPU hash join. The GPU
broadcast join operator works for the scenario that the size of at least one of the two join tables is small (e.g., Spark’s default threshold is 10MB). It first broadcasts one of the two tables, which is usually the smaller one. Then it joins the broadcast table with each data partition of the other table. The GPU hash join operator works for other scenarios. It first repartitions the data based on the hash values of join keys, which will shuffle data in both tables. Then it joins each pair of repartitioned data.

Spark-GPU executes the join part on GPUs for both operators. We use conventional hash join algorithm on GPUs since it performs well especially when the sizes of the two tables differ significantly. It has a build phase and a probe phase.

In the build phase, a hash table is built on one table. We store the hash table in a continuous memory inside GPU such that it can be searched efficiently. Each hash table entry is a \((id, value)\) pair where \(id\) denotes the hash key and \(value\) denotes the position of data in the partition. We scan the build table to avoid synchronizations when building the hash table. The first scan simply counts the number of keys that are hashed to each hash value while the second scan directly to writes to the hash table memory without synchronizations based on the prefix sum of the first scan results.

The probe phase is straightforward. The join key column from the other table is scanned to probe the hash table and a 0/1 vector is maintained to indicate which data should be projected. Similar to GPU in-memory scan, a prefix sum is calculated on the vector such that the results can be generated without synchronizations.
GPU Aggregation

Aggregation divides data into groups and calculates various functions inside each group. The GPU aggregation operator is implemented as a partial aggregation followed by a global aggregation. The partial aggregation directly aggregates each partition of the input data, which significantly reduces the amount of data to be shuffled. After that, the aggregation results of each partition is shuffled and the final aggregation results are calculated. Standard aggregation functions such as \textit{SUM} and \textit{AVG} are supported in Spark-GPU.

Spark-GPU only executes partial aggregation on GPUs since the number of data elements to be aggregated in global aggregation is usually small. We use hash aggregations on GPUs. All group by keys are converted into strings to calculate hash values. When calculating the aggregation results, we use standard GPU library’s atomic operations to synchronize GPU threads when they are updating aggregation results for the same group. Since atomic operations on 64-bit words with type long and double are not supported on many GPUs, Spark-GPU converts data with type long or double to float before aggregation on GPUs and converts the result type back when aggregation finishes.

GPU Sort The logic of GPU sort operator is similar to that of GPU aggregation operator. GPU sort operator first sorts each data partition on GPUs. Then it shuffle sorts data in all partitions.

Spark-GPU implements bitonic sort on GPUs. Since sort is usually executed after aggregation in SQL queries, the number of data elements to sorted is relatively small. In this case, GPU shared memory can be used for sorting. Keys (i.e. columns in query’s or-
der by clause) are sorted first on GPUs. If there are multiple order by columns in the query, data will be sorted by each column one by one. After the keys are sorted, the results can be generated using a gather operation.

### 4.4.2 GPU-Aware Query Optimizer

Given an SQL query, the query optimizer finds the best execution plan with existing query operators. Spark’s query optimizer is designed with a set of rules and strategies. Rules are used to generate optimized logical query plan and strategies are used to generate optimized execution plan. Currently Spark’s query optimizer doesn’t have an accurate cost model to estimate which execution plan has the best performance, thus it simply picks the first plan to execute the query.

To generate query execution plans with both CPU operators and GPU operators, Spark-GPU extends Spark’s query optimizer by adding a set of new GPU strategies. Spark-GPU guarantees that if a query plan with GPU query operators is generated, it will be the first physical plan among all plans and thus will be used to run the query.

Given a logical plan, the criteria to determine whether to use a GPU query operator are: (1) the operator can benefit from using GPUs; and (2) there exist a chain of GPU query operators that process the data in native memory before copying the data back to Java heap. If either of the above criteria holds, the query optimizer will choose GPU query operator. Otherwise it will use Spark’s existing CPU query operators. The following GPU strategies are added:

- Join operators are offloaded to GPUs. If the size of one join table is smaller than Spark’s broadcast threshold, GPU broadcast join operator is used. Otherwise GPU
hash join operator is used. The rationale is that the performance of join operator is bounded by memory accesses. It can benefit from GPU’s high memory bandwidth; thus GPU join operators will always be used.

- The children of GPU broadcast join should use GPU operators whenever possible. The rationale is to push as many operations as possible to GPUs when data are in native memory. Note that this doesn’t work for other join operators since they will shuffle data from both tables.

- If an aggregation can be divided into a partial aggregation and a global aggregation, GPU aggregation operator will be used because aggregation is compute intensive operation. The child of the GPU aggregation will also be executed with GPU query operators if possible.

- If the child of sort is a GPU operator, GPU sort operator will be used. The rational is to push as many operations as possible to GPUs when data are in native memory.

Note that not all query operations are suitable for GPUs. For example, if a query only contains a simple scan operator, it will be executed on CPUs.

### 4.4.3 Optimizations

Spark-GPU executes queries on both CPUs and GPUs. However, this doesn’t guarantee optimal performance due to query operator’s iterator interface. We use the following query to illustrate the problem. The query first scans a large table `lineorder` and a small table `supplier` and then joins the tuples that satisfy the scan predicates.
Spark-GPU executes the query with two GPU in-memory scans followed by a GPU broadcast join. Since query operators are connected by the row-format iterator interface, scan results of the large table `lineorder` must be copied from native memory to Java heap and materialized into row format after the GPU scan, and copied back to native memory and batched to column format before joining on GPUs. These data copying operations are unnecessary since the results of GPU in-memory scan of the large table `lineorder` are already in native memory and can be directly used for joins on GPUs.

To solve the problem, Spark-GPU introduces a batch interface to connect GPU query operators. The batch interface executes the operator on GPUs and returns the data in column format in native memory, not in row format in Java heap. The original iterator interface is implemented on top of the batch interface. When a GPU query operator is fetching data from another GPU query operator, it can directly call the batch interface to get the data addresses in native memory, which avoids the unnecessary data copying operations and improves the query performance.

### 4.5 GPU Resource Management

GPUs should be efficiently managed in the cluster. A straightforward approach to manage GPUs is to treat each GPU as a CPU core and manage GPUs in the same way as CPUs. In this case, when a task requires a GPU, a GPU will be exclusively assigned to the task until the task finishes. This coarse-grained management approach has been adopted in MapReduce Hadoop systems (e.g., [43]) to manage GPU resources. Although
this approach makes GPUs available in the cluster, it may underutilize GPU resources, which may in turn degrade the performance of GPU applications.

To understand the problem, we implemented a synthetic GPU workload on Spark-GPU and measured its performance when varying the number of tasks that can be concurrently executed on a GPU. The workload contains three simple operations: (1) transfer data to the GPU; (2) access data inside the GPU and (3) generate a constant number as result. We control the number of times that data are accessed inside the GPU to control the ratio of computation time and PCIe data transfer time to simulate workloads with different computation intensity. Three computation/PCIe ratios were used: 1:1, 10:1 and 100:1. The experiments were conducted on a 16-core node with one GPU. Thus the maximum number of concurrent GPU tasks was 16. The input data were cached in memory and had 16 partitions. The size of each partition was 128MB. For each computation/PCIe ratio, we used the workload performance when only one task can be executed on the GPU as the baseline and normalized all other performance to the baseline. The execution results are shown in Figure 32.
Two observations can be obtained from Figure 32. First, GPU sharing improves the performance of GPU applications on Spark-GPU. When computation/PCIe ratio is 1:1, performance is improved by more than 2x. Second, increasing computation/PCIe ratio will decrease the performance benefits obtained from sharing GPUs. The fundamental factor that determines the benefit of GPU sharing is the amount of work that can be executed in parallel when running tasks on GPUs. To offload an operation to GPUs, Spark-GPU needs to copy data between native memory and Java heap, transfer data between native memory and GPU device memory through the PCIe bus, and execute kernel on GPUs. Among these operations, two combinations of operations are executed serially. First, kernel executions can hardly be executed in parallel due to GPU’s LEFT-OVER resource management [69]. Second, PCIe data transfer in the same direction must be executed serially due to hardware limitations. All other combinations of operations can be overlapped and thus benefit from sharing GPUs.
Sharing GPUs in Spark-GPU is necessary. The reasons are twofold. First, Spark-GPU is a general purpose data processing system. It needs to benefit as many workloads as possible with the high performance GPUs. Second, GPU sharing can improve the system throughput by overlapping operations from different GPU tasks.

The major obstacle for sharing GPUs in the cluster is that operating system and GPU drivers lack support of virtual memory management for GPUs. Currently GPU memory is managed by GPU applications. As a result, when a GPU is shared by multiple GPU tasks, task crash may happen due to insufficient GPU memory. Although some crashes can be avoided by controlling data partition size in the cluster, the system needs to provide the ability to manage GPU’s memory such that it can concurrently execute GPU tasks when necessary.

4.5.1 Our Solution: User-Level GPU Memory Management

We design a user-level library to manage GPU memory motivated by existing research (e.g., [51, 89, 87]). We have two design goals. First, the library should be transparent to both GPU programs and the operating system. Second, the overhead of memory management should be as low as possible. Prior works require modifications to either GPU programming interface or operating system to improve workload performance in the scenario when there is not enough device memory on the GPU. These modifications will greatly limit the GPU workloads that can be processed by the system. In Spark-GPU, we observe that task crashes due to memory contention is rare. Thus the design of our library is optimized for the regular scenario and only guarantees that tasks will not crash due to GPU memory contention.
Figure 33 shows the architecture of the GPU memory management library. The library works as a layer between the GPU tasks and the standard GPU library. It intercepts all GPU memory related system calls (e.g., GPU memory allocation, free and kernel launch) from the tasks running on the node to manage the usage of GPU memory. The library manages GPU memory based on the concept of regions. A region contains a GPU buffer and a CPU buffer. The CPU buffer is used for storing the data in the GPU buffer when the library decides to swap out the region. When a task tries to allocate GPU memory, a new region with a CPU buffer is created. The region’s GPU buffer is not created until the task tries to access the data in the buffer in the kernel. When there are not enough GPU memory, the library scans the existing regions and swaps out an unused region.
In Spark-GPU, each worker in the cluster needs to load the GPU memory management library when it starts. With the GPU memory management library, each GPU task has the illusion that it can use the whole GPU memory even through the GPU is shared.

### 4.5.2 GPU Abstraction and Task Scheduling

Spark-GPU enables GPU sharing in the cluster. It abstracts each GPU into multiple logical GPUs. Each logical GPU can run one GPU task. GPU’s sharing granularity is configurable. Users need to explicitly set the number of available GPUs on each node and how many GPU tasks can be concurrently executed on each GPU. Note that the total number of GPU tasks that can be concurrently executed on one node cannot exceed the total number of CPU cores on the node, since each GPU task needs a CPU core to initiate the task.

Spark-GPU schedules tasks based on RDD lineage graph. To schedule tasks to GPUs, the scheduler checks if a stage contains any RDD that is created by a GPU operation (e.g., an SQL GPU operator and a GPU-RDD operation). If a stage contains any GPU operation, it will only be scheduled to nodes that have GPUs and that have at least one available CPU core. GPU tasks on the same node are scheduled to the GPU in a FIFO way.

### 4.6 Experiments

#### 4.6.1 Experimental Environments and Workloads

We conducted all the experiments on a cluster with 9 nodes on Amazon EC2. The EC2 instance type was \textit{g2.x2large}. Each node has a 2.6 GHZ Intel Xeon E5-2670 (Sandy
Bridge Processor and 15 GB memory with a bandwidth of 51.2 GB/s. There is one NVIDIA GK104 GPU on each node. The GPU has 1536 cores, 4GB memory, with a clock frequency of 800MHZ and a memory bandwidth of 192.26 GB/s. The OS on each node was Ubuntu 14.04. The version of NVIDIA driver was 320.48. CUDA 6.5 was used. In the cluster, we configured one node to be Spark’s master node (also HDFS’s namenode) and the rest to be slave nodes. Spark 1.6.0 and Hadoop 2.6.0 were used in the experiments. Spark-GPU was developed on top of Spark 1.6.0.

We examined the performance of Spark-GPU using workloads from four categories: data mining, statistical analysis, Star Schema Benchmark [67] queries and TPC-H benchmark [9] queries. The data mining and statistical workloads we used were K-Means and logistic regression. The data set of K-Means had 2 million data points, each of which had 256 or 1024 features. The number of centers was 2048. The data set of logistic regression had 2 million data points, each of which had 512 or 1024 features. The GPU version of K-Means and logistic regression were implemented based on GPU-RDD. We set the scale factor to 50 for both Star Schema Benchmark and TPC-H benchmark in the experiments.

In the experiments, all the workload’s data were initially cached in the cluster memory. We run each experiment 5 times and report the median results.

4.6.2 Effectiveness of GPU Sharing

We first evaluate the effectiveness of GPU sharing in Spark-GPU. For each workload, we measured its performance on Spark-GPU when 1, 2, 4, and 8 tasks can be executed concurrently on a GPU respectively. We used the performance when one task
can be executed on a GPU as baseline and normalized all other performance to the baseline.

Figure 34 Effectiveness of GPU sharing in Spark-GPU

Figure 34 presents the results of workloads from each category representing the following workload types: compute-intensive workload (K-Means-1024 and LR-1024), shuffle-intensive query (TPC-HQ3) and shuffle-rare query (SSB-Q3.1). The workload details will be discussed later when we study Spark-GPU’s performance. We observe that only shuffle-rare query can significantly benefit from sharing GPUs. The performance is improved by up to 1.61x. The improvements of other workloads are mediocre. The reason is that the benefits of sharing GPUs mainly come from overlapping PCIe transfer with GPU kernel execution. This is consistent with our analysis in Chapter 4.5.

Another observation is that sharing GPU between two tasks is better than other choices. This illustrates that although sharing GPUs can overlap operations to improve
performance, over subscribing can incur resource contention and degrade the performance to some extent.

4.6.3 Data Mining

K-Means is a widely used clustering algorithm. We implemented it on both Spark and Spark-GPU to study the performance. The algorithm mainly contains two steps: (1) finding the closest center for each data point and (2) updating the cluster centers. When implementing the algorithm on Spark-GPU, we first created a GPU-RDD that stored data in a columnar format in one continuous region in the native memory. Note that only one GPU memory copy command was needed for each partition of the data points and coalesced GPU memory accesses can be guaranteed. Then we conducted computation on the GPU-RDD. For each iteration of computation, we offloaded the operation of calculating the closest center to GPUs. A GPU kernel was launched to find the closest center for each data point in one partition and aggregate on the centers locally. After that, a global aggregation was performed to update the centers.

Figure 35 shows the performance result. Spark-GPU significantly outperformed Spark for K-Means workloads. When data points had 256 and 1024 features, Spark-GPU improved the performance by 5.71x and 3.84x respectively. The performance was improved because of K-Means’ compute-intensive nature. K-Means only shuffles centers in each iteration. The dominant operation is finding the closest centers, which can benefit significantly from GPU’s high memory bandwidth and high parallelism.
4.6.4. Statistical Analysis

Logistic regression is a commonly used classification method in statistical analysis. The algorithm finds a value \( v \) that can best separate a data set. In each iteration of computation, a logistic function is applied to every data point in the data set. Then all the results are aggregated to update \( v \). When implementing on Spark-GPU, each data partition was first calculated a local \( v \) on GPUs. After that, Spark-GPU aggregated all local values to update \( v \). Similar to K-Means, we stored all data points in columnar format in one continuous memory region, which reduced the overhead of transferring the data points to GPU device memory and increased the GPU kernel performance. The performance results are shown in Figure 36.
Logistic regression’s performance trend is similar to that of K- Means. Spark-GPU significantly improved the performance. When data had 512 and 1024 features, the performance were improved by 16.13x and 13.73x. The reason for Spark-GPU’s better performance is that logistic regression is bounded by calculations on each data point, which can benefit from GPU’s high computation powers.

4.6.5 Star Schema Benchmark

Star Schema Benchmark (SSBM) [67] evaluates the performance of decision support systems. It has 13 queries, divided into 4 query flights. All queries operate on 4 tables: one fact table lineorder and four dimension tables date, part, supplier and customer. Spark- GPU can execute all queries from Star Schema Benchmark.
To fairly compare the performance of Spark-GPU and Spark, we manually chose the best query execution plan for each SSBM query when running on Spark. Figure 37 shows the performance results of SSBM queries when running on Spark-GPU and Spark. As can be seen, Spark-GPU outperformed Spark for using GPUs. The performance improvements were between 1.92x and 4.83x. We use query 3.1 from the benchmark to as an example to illustrate Spark-GPU’s performance behaviors.

SSB query 3.1:
SELECT c_nation, s_nation, d_year, sum(lo_revenue) as revenue
FROM lineorder, customer, supplier, date
WHERE lo_custkey = c_custkey
    and lo_suppkey = s_suppkey
    and lo_orderdate = d_datekey
    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
Query 3.1 joins three dimensional tables with the fact table, and then groups and sorts the result. Since all dimension tables are much smaller than the fact table, the best way to execute the joins are using broadcast joins. When running on Spark-GPU, all the scans, joins and partial aggregation are executed using GPU operators. The overall join selectivity of query 3.1 is 3.4%. Thus the three broadcast joins dominate the total execution time, which can benefit significantly from using GPUs. Moreover, Spark-GPU can directly work on the columnar data in the native memory from the fact table `lineorder` without converting the data back to rows for consecutive broadcast join operations, which reduces the overhead of block processing and maximize GPU’s performance.

Other SSBM queries have similar patterns as query 3.1. Their execution time are also dominated by operations on fact table `lineorder`, which can be accelerated by using GPUs.

4.6.6 TPC-H Benchmark

TPC-H benchmark [9] is another benchmark to evaluate the performance of decision support systems. It has a snowflake schema and 22 queries, which have more complex behaviors than SSBM queries. Currently not all query features (e.g. nested queries, case clause) from TPC-H can be executed on Spark-GPU. In this case, these queries will be executed on CPUs in the same way for Spark-GPU and Spark. It is our future work to support these features on Spark-GPU. In the experiments we compared the performance of 4 TPC-H queries and the results are shown in Figure 38.
As can be seen, Spark-GPU improved the performance for all 4 TPC-H queries. The performance of query 1 and query 6 were improved by 4.32x and 3.15x, while the performance of query 3 and query 10 were only improved by 1.23x and 1.31x. Query 1 and 6 have query similar query patterns as SSBM queries. On the other hand, query 3 and 10 are different. We use query 3 to explain the performance differences.

```
TPC-H query 3:
SELECT l_orderkey, o_orderdate, o_shippriority, 
      sum(l_extendedprice * (1-l_discount)) as revenue
FROM orders, customer, lineitem
WHERE c_mktsegment = 'MACHINERY' 
    and c_custkey = o_custkey 
    and o_orderdate < '1995-03-26' 
    and l_shipdate > '1995-03-26'
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, orderdate
```

Query 3 joins three tables `orders`, `customer` and `lineitem`, and then groups and sorts join results. Different from SSBM, none of these tables are small enough for broadcast joins, thus shuffle join (e.g., hash join and sort merge join) was used. When executed on
Spark-GPU, joins and the aggregation were executed with GPU query operators. The
scans were executed on CPUs based on Spark-GPU’s query optimizer.

Shuffle joins shuffle data from both tables, which include I/O operations such as
serializing data to local disks before data shuffling and deserializing the data from the
disks before join actually happens. These I/O operations dominate the execution time of
query 3 and thus the overall performance cannot be accelerated by GPUs. This indicates
that shuffle-intensive queries should be executed on CPUs to save GPUs for more
suitable workloads, unless the I/O performance is significantly improved in the future.

4.6.7 A Case for Spark-GPU

The experiments demonstrate Spark-GPU’s performance advantage over Spark. In
general, Spark-GPU can significantly improve the performance of compute-intensive
workloads and shuffle-rare star-schema queries.

Spark-GPU provides both high-level SQL interface and low-level procedural
programming interface to support different analytic applications. While SQL applications
can transparently benefit from GPUs, other applications need user’s efforts to write GPU
kernels if they want to use GPUs. Since Spark already has several built-in libraries to run
analytic applications such as machine learning, one question that has not been answered
is that can these legacy codes easily utilize GPUs without significant changes? A
straightforward way to achieve this is to parallelize all existing Spark RDD operations on
GPUs such that applications built on top of RDDs can automatically utilize GPUs.
However, this approach cannot guarantee good performance. In fact, it performs even
worse than the original Spark based on our preliminary results. The reason is that RDD
doesn’t have semantic information about the applications, which causes unnecessary data copying operations between Java heap memory and native memory, and uncoalesced GPU memory accesses when using GPUs. We believe Spark-GPU ’s approach is the right one to integrate GPUs into existing in-memory data processing systems. In this paper, we make a strong case for the design and implementation of Spark-GPU that is an effective methodology to integrate GPUs into existing in-memory data processing systems.

In the future work, we will support more kinds of workloads (e.g., workloads from SparkBench [64]) on Spark-GPU.

4.7 Related Work

The Graphics Processor Unit (GPU) has become a general purpose computing device because of its high performance. In the past decade, the research community has conducted extensive work on how to use the GPU to accelerate various data parallel operations.

In the relational database, GPU has been used to accelerate both database operators and complex analytic queries. The performance of sort [36, 78], join [45, 52] and aggregation [53] have been improved significantly with optimized algorithms when running the GPU. Complex queries can benefit from using the GPU with various software optimizations [44, 93, 89]. These works demonstrate the performance potential for using GPUs to process SQL queries.

With GPU’s superior performance for data parallel applications, researchers have investigated how to use GPUs in MapReduce systems. Mars [43] designed a MapReduce-
like system Mars on a single node GPU. Mars implemented a set of interfaces such as Map and Reduce on the GPU, which could be used to implement various analytic applications on the GPU. Stuart et al. [82] proposed GPRM, a MapReduce-like GPU framework that accepts user-implemented GPU kernels for Map and Reduce operations and runs them on a GPU cluster. El-Helw et al. [30] designed a MapReduce framework Glasswing using OpenCL that can exploit various computing devices and overlap operations such as computation and communication. He et al. [47] proposed the Hadoop+ system that can execute applications both CPUs and GPUs in a Hadoop cluster. They focused on the resource contention between CPU tasks and GPUs and proposed a model to help allocate GPU resources. These MapReduce-like GPU systems improved the performance of various workloads, which demonstrates GPU’s performance potential for MapReduce systems.

HeteroSpark [59] is a framework that supports executing certain machine learning workloads on Spark with GPUs. It uses Java RMI to transfer data between a CPU’s JVM and a GPU’s JVM, which can incur expensive overhead (e.g. serialization and deserialization of the data) and compromise the system’s fault tolerance ability. Spark-GPU doesn’t introduce any extra communication between cluster nodes and has minimized data movements when using GPUs.

4.8 Summary

In this Chapter we have explored how to improve the performance of production level in-memory data processing systems with GPUs. We have presented the design of Spark-GPU, a CPU-GPU hybrid system built on top of Apache Spark that can exploit GPUs in
the most efficient way. Spark-GPU has addressed a set of real-world challenges incurred by the mismatches between Spark and GPUs. We have comprehensively examined the performance of Spark-GPU with representative data analytic workloads. Our work has the following conclusions: (1) Spark-GPU can accelerate various data analytics workloads, but non-trivial engineering efforts are needed to address critical mismatches between Spark’s Java-based network-centric execution model and GPU’s unique architecture and programming model; (2) Spark-GPU provides speedups at a certain level (up to 4.83x) for traditional data warehousing workloads (represented by TPC-H queries and Star Schema Bench-mark queries). GPU’s performance advantages are significantly impacted by data shuffling in query execution. (3) Spark-GPU can significantly accelerate compute-intensive data mining and statistics analysis applications (up to 16.13x), represented by the K-means clustering and Logistics Regression algorithms.
Chapter 5 Conclusions

In this dissertation, we have addressed three critical challenges to build high performance in-memory data processing systems on massive parallel processors.

The first challenge is to develop a high performance concurrency control method for in-memory OLTP databases. We address the challenge by designing a new concurrency control method called Balanced Concurrency Control (BCC), which can deliver good performance under both low contention and high contention. Compared to Optimistic Concurrency Control (OCC) method that aggressively abort transactions under high contention, BCC nicely balances the accuracy and cost of identifying unserializable transactions. Specifically, BCC checks one more data dependency compared to OCC in a confined search space to guarantee transaction serializability, which makes it possible to keep the overhead low. To demonstrate BCC’s effectiveness, we implemented it in Silo, a high performance OCC in-memory database, with a set of advanced optimizations. Our comprehensive evaluation shows that BCC can perform up to 3.68x better than OCC.

The second challenge is how to accelerate complex analytical queries using the massive parallel co-processor GPUs. To solve the problem, we developed a GPU analytical database that supports star schema queries. We applied various optimizations such as data compression, pipelining and advanced query processing algorithms to the system, and comprehensively studied the performance of complex queries on GPU and
CPU. We have made three major contributions in this study. First, we illustrated the best way to optimize complex analytical queries on GPUs. Second, we presented a cost model that can accurately predict the query performance on GPUs, which can be used for query scheduling in heterogeneous computing environments. Third, we open sourced our GPU database for studying GPU query optimization and future research.

The third challenge is related to efficiently integrating GPUs into existing CPU-optimized in-memory data processing systems. Existing research works adopt a GPU-centric approach, without considering the performance of CPU operations. To illustrate whether existing CPU-optimized data processing systems can benefit from GPUs, we built a GPU in-memory data processing engine - Spark-GPU, which is based on the widely-used Apache Spark system. Spark-GPU has addresses a set of critical issues to improve the system’s performance, including minimizing internal and external data transfers, preparing a suitable data format and a batching mode for efficient GPU execution, and determining the suitability of workloads for GPU with a task scheduling capability between CPU and GPU. We have comprehensively evaluated Spark-GPU with a set of representative analytical workloads to show its effectiveness. Our results show that Spark-GPU improves the performance of machine learning workloads by up to 16.13x and the performance of SQL queries by up to 4.83x. This work is a case study based on Apache Spark, but our experience and insights can be generally applied to other systems that have similar challenges.
Bibliography


