Spark-GPU: An Accelerated In-Memory Data Processing Engine on Clusters

Yuan Yuan*, Meisam Fathi Salmi†, Yin Huai‡, Kaibo Wang§, Rubao Lee* and Xiaodong Zhang*

*The Ohio State University †Paypal Inc. ‡Databricks Inc. §Google Inc.
*{yuanyu, liru, zhang}@cse.ohio-state.edu †mfathisalmi@paypal.com ‡yhuai@databricks.com §kaibo@google.com

***Abstract*—**
Apache Spark is an in-memory data processing system that supports both SQL queries and advanced analytics over large data sets. In this paper, we present our design and implementation of Spark-GPU that enables Spark to utilize GPU's massively parallel processing ability to achieve both high performance and high throughput. Spark-GPU transforms a general-purpose data processing system into a GPU-supported system by addressing several real-world technical challenges 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.

I. 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 [1], which is an open source implementation of MapReduce [10]. Several widely used data processing systems have been built on top of Hadoop (e.g., Pig [29] and Hive [22]), 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 [26], GPUs [39], and SSDs [28]. 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 [6]. Utilizing main memory to improve the performance of data analytics applications have become desirable and feasible.

Apache Spark is a representative open-source, distributed in-memory data processing system [2], [40], which has gained popularity for its improved performance over Hadoop. It not only supports executing SQL queries [7], but also provides procedural processing ability for advanced data analytics such as machine learning and graph applications [12]. With in-memory data processing, the performance bottleneck of data analytics applications has shifted from disk I/O and network to computations, as is the case for Spark [31]. 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) [40], and data analytics 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 these applications such as machine learning. These two patterns make GPU an ideal 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 applications. The research community has extensively studied how to utilize GPUs to accelerate various data analytics applications, including SQL queries [18], [39], NoSQL operations [41], [20], machine learning [9] and graph applications [15]. The performance of these applications can be significantly improved with GPUs. However, many of these work adopt a GPU-centric design (e.g., [39], [41]), which redesigns the system based on GPU's characteristics to maximize GPU's performance 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 paper, we explore how to use GPUs to accelerate the performance of various data analytics applications on production-level, CPU-optimized distributed in-memory data processing systems such as Apache Spark. 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 data analytics 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 data analytics applications. Our comprehensive evaluation shows that Spark-GPU can achieve up to 4.83x performance speedup for SQL queries, and achieve 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 paper is organized as follows. Section II describes the overall architecture of Spark-GPU. Section III, Section IV and Section V describe the detailed designs of Spark-GPU. Section VI presents the evaluation results of Spark-GPU. After introducing the related work in Section VII, Section VIII concludes the paper.

II. Spark-GPU Overview

This section introduces the challenges of using GPUs in Spark and presents an overview of Spark-GPU, which has overcome all the challenges and can efficiently execute various data analytics applications on both CPUs and GPUs.

A. Challenges

Data analytics 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 analytics applications running on Spark, the following challenges must be addressed.

- First, Spark uses the iterator model [14] 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 one-element-a-time 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 [3] and OpenCL [4], 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, which is expensive. A software mechanism of minimizing data copying between Java heap memory and native memory must be developed in order to gain high performance from GPUs.
- Third, existing cluster resource manages such as Yarn [36] and Mesos [21] 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.

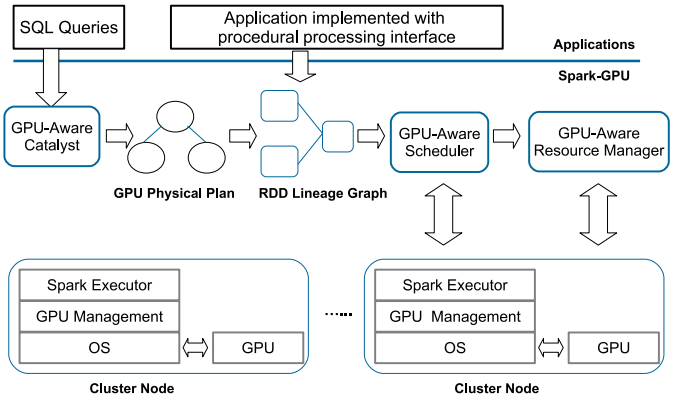

Fig. 1: The overall architecture of Spark-GPU.

B. An Overview of Spark-GPU

Spark-GPU handles all the aforementioned challenges to efficiently execute data analytics applications on GPUs. Figure 1 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. Section III 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. Section IV explains how SQL queries are processed on GPUs.

Third, in order to efficiently execute data analytics 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 analytics applications. Each operation takes an RDD or multiple RDDs 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. Section V presents how Spark-GPU manages GPU resources and schedules tasks.

III. Execution Model and Data Format

GPU is a massively parallel co-processor, which executes GPU kernels [1] 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, to fully utilize GPU's memory bandwidth, data should be accessed in a coalesced manner, where consecutive GPU threads access consecutive GPU memory locations, as shown in Figure 2.

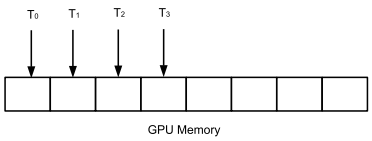

Fig. 2: An example of coalesced GPU memory access.

To meet the first requirement, a system needs to support block processing model [32] that processes a block of data elements at a time. To meet the second requirement, a system needs to organize data into an appropriate 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. 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.

A. Our Solution: GPU-RDD

Spark's design is centered around the concept of 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 the 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 data flows. Moreover, the block interface provides applications the ability to apply an operation on GPU-RDD to all the 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 the number of elements in the partition, data types and others. 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 data analytics 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 a large number of columns, they should be stored in one memory region such that PCIe 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 collection activities, 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.

B. 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 data analytics 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

[1] A GPU kernel is a program executed on the GPU.

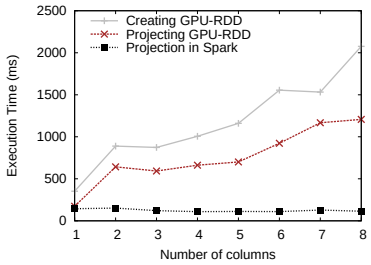

Fig. 3: The execution time of the micro benchmark with different number of string columns in a row.

Java heap and native memory, and between native memory and GPU device memory. In general, the following steps are needed to offload an operation on GPU-RDD to GPUs: (1) copying data from Java heap to native memory to create the GPU-RDD; (2) transferring data to GPU device memory; (3) computing on GPUs; (4) transferring results from GPU device memory to native memory; and (5) copying results from native memory to Java heap. Compared to directly executing the operation on CPUs, 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 3.

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 considered to be offloaded to GPUs.

IV. Query Processing on GPUs

SQL queries are important data analytics applications. A major difference between SQL queries and other data analytics 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 CPUs and GPUs, Spark-GPU implements a set of high performance GPU query operators, and extends Spark's query optimizer to build a query execution plan with both CPU query operators and GPU query operators. In this section 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.

A. GPU Query Operators

Spark-GPU supports 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. Each GPU query operator includes 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 column data on GPUs using standard GPU programming models.

1) GPU Scan: The GPU scan operator implements selection operation on in-memory data, which will return all data that satisfy the 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.

2) GPU Broadcast Join and Hash Join: Join is one of the most important SQL operations. Spark-GPU supports two kinds of equi-join operators on GPUs: GPU broadcast join and GPU hash join. The GPU broadcast join operator first broadcasts the smaller one of the two input tables. Then it joins the broadcasted table with each data partition of the other table. The GPU hash join operator 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 implement the conventional hash join algorithm on GPUs since it performs well especially when the sizes of the two tables differ significantly [8]. 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 twice 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 scan operator, a prefix sum is calculated on the vector such that the results can be generated without synchronizations.

3) 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 *SUM* and *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.

4) 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 sort is relatively small. In this case, GPU shared memory can be used for sorting. Keys (i.e. columns in query's order-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.

B. 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 does the best job, 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) GPU can improve the operator's performance; 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 a 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.

Not 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.

C. Optimizations

Spark-GPU executes queries on both CPUs and CPUs. 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.

```
select s_nation, lo_revenue
from lineorder join supplier
on lo_suppkey = s_suppkey
where s_region = 'ASIA' and lo_discount<5
```

Spark-GPU executes the query with two GPU scan operators followed by a GPU broadcast join operator. 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

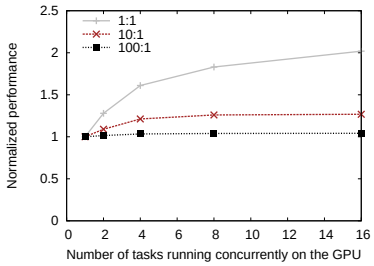

Fig. 4: Normalized performance when different number of tasks are running concurrently on the GPU.

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.

V. 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., [16]) 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) transferring data to the GPU; (2) accessing data inside the GPU and (3) generating 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 4.

Two observations can be obtained from Figure 4. 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 LEFTOVER resource management [33]. 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.

It is necessary for Spark-GPU to manage GPUs in a fine-grained way to enable GPU sharing. The reasons are twofold. First, Spark-GPU is a general purpose data processing engine. 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.

A. Our Solution: User-Level GPU Management

We design a user-level library to manage GPU memory motivated by existing research (e.g., [23], [38], [37]). 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 (e.g., [23], [38], [37]) require modifications to either GPU programming interface or operating system to improve workload performance in the scenario when there are 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, our design of the library is optimized for the regular scenario where GPU memory contention is rare. The library guarantees that tasks will not crash due to GPU memory contention. When GPU memory contention happens, Spark-GPU will stop scheduling new tasks to the GPU because GPU task's performance can be severely degraded due to frequent data swap activities over PCIe bus.

Figure 5 shows the architecture of the GPU management library. The library works as a layer between the GPU tasks and the standard GPU library. It intercepts all GPU related system calls (e.g., GPU memory allocation, free and kernel

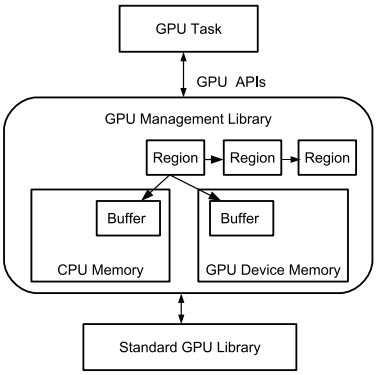

Fig. 5: The architecture of GPU management library.

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 management library when it starts. With the GPU management library, each GPU task has the illusion that it can use the whole GPU resource even through the GPU is shared.

B. 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.

VI. Experiments

In this section we comprehensively study the performance of Spark-GPU. Our goal is to illustrate the strength and limitations of using GPUs in in-memory data processing systems. We first describe the experimental environment and the workloads used in the experiments. Then we present the experimental results.

The experiments have demonstrated that:

- Spark-GPU can improve the SQL query performance by up to 4.83x, data mining and statistical workloads by up to 16.13x, which shows Spark-GPU's advantage in accelerating data parallel analytics workloads with GPUs.
- Sharing GPUs in the cluster improve the performance. The performance of SQL query can be improved by up to 1.61x, while the improvements for data mining and statistical workloads are marginal.

A. Experimental Environments and Workloads

We conducted all the experiments on a cluster with 9 nodes on Amazon EC2. The EC2 instance type was *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 [30] queries and TPC-H benchmark [5] 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 [2] 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 experiments 5 times and report the median results.

B. 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 6 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-H-Q3) 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 the

[2]Scale factor denotes the size of data set in the benchmark

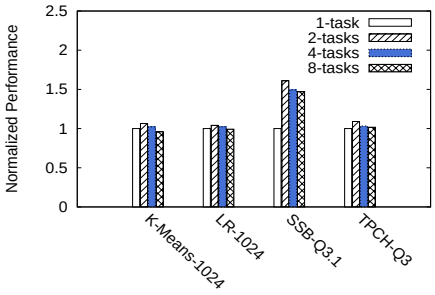

Fig. 6: Effectiveness of GPU sharing in Spark-GPU

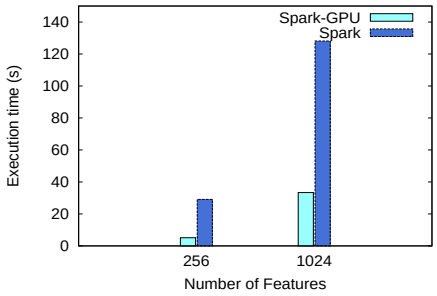

Fig. 7: Results of executing one iteration of K-Means on Spark and Spark-GPU when data have 256 and 1024 features.

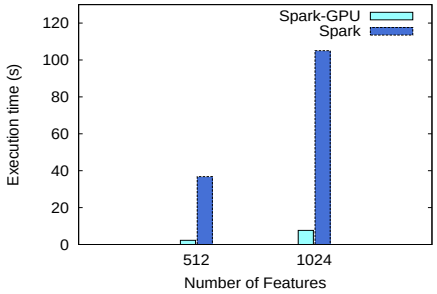

Fig. 8: Performance results of executing one iteration of logistic regression on Spark and Spark-GPU when data have 512 and 1024 features.

current GPU kernel execution with the PCIe data transfer for subsequent GPU tasks. This is consistent with our analysis in Section V.

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.

C. Effectiveness of Spark-GPU

We next examine Spark-GPU's designs by comparing its performance with that of Spark. In these experiments, a GPU was shared between two tasks in Spark-GPU to give the best performance.

Note that there also exist research works on using GPUs in Spark for machine learning workloads (e.g., HeteroSpark [27]). We don't compare Spark-GPU with them because: (1) their codes are not available; and (2) they are not general-purpose systems as Spark-GPU does.

1) 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 step: (1) finding the closest center for each data point and (2) update the cluster centers. When implementing the algorithm on Spark-GPU, we first created a GPU-RDD that stored data in 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 7 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's 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.

2) 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 8.

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.

3) Star Schema Benchmark: Star Schema Benchmark (SSBM) [30] 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 9 shows the performance results of SSBM queries when running on Spark-GPU and Spark. As can 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.

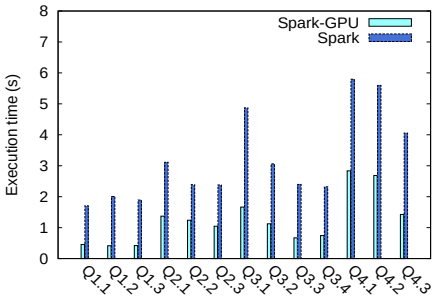

Fig. 9: The performance results of Star Schema Benchmark when executed on Spark and Spark-GPU.

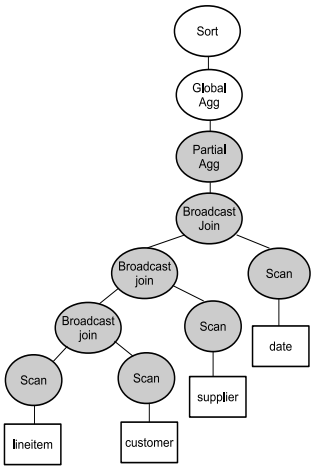

Fig. 10: The physical execution plan of SSB 3.1. The shaded operators were executed with GPU query operators.

```
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. Figure 10 shows query 3.1's physical execution plan. 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) TPC-H Benchmark: TPC-H benchmark [5] 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 11.

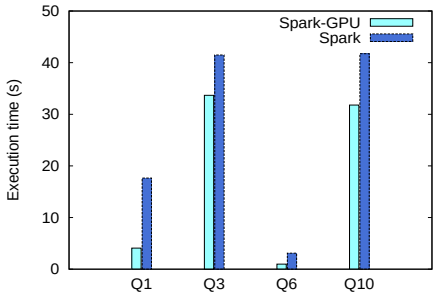

Fig. 11: The performance results of 4 TPC-H queries when running on Spark and Spark-GPU.

As can 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. Figure12 shows query 3's physical execution plan. 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.

D. A Case for Spark-GPU

Spark-GPU provides both high-level SQL interface and low-level procedural programming interface to support different data analytics applications. While SQL applications can transparently benefit from GPUs, other applications need user's

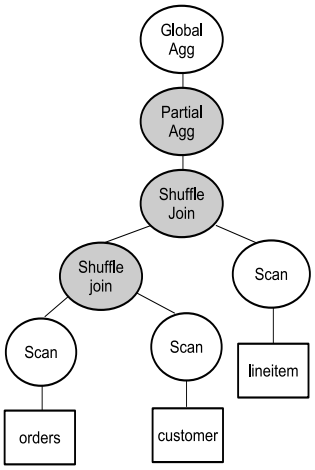

Fig. 12: The physical execution plan of TPC-H query 3. The shaded operators were executed with GPU query operators.

efforts to write GPU kernels if they want to use GPUs. Since Spark already has several built-in libraries to run data analytics applications such as machine learning, one question that has not been answered is whether 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 high 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. 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.

VII. 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 [13], [34], join [18], [24] and aggregation [25] have been improved significantly with optimized algorithms when running the GPU. Complex queries can benefit from using the GPU with various software optimizations [17], [39], [38]. 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 [16] 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. [35] 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. [11] designed a MapReduce framework Glasswing using OpenCL that can exploit various computing devices and overlap operations such as computation and communication. He et al. [19] 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 [27] 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.

VIII. Conclusion

In this work we have explored how to improve the performance of production-level, CPU-optimized distributed 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 analytics 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 Benchmark 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. Spark-GPU represents an effective methodology to build an accelerated in-memory data processing engine on clusters.

IX. Acknowledgments

We thank the anonymous reviewers for their constructive comments. The work was supported in part by the National Science Foundation under grants OCI-1147522, CNS-1162165, and CCF-1513944.

References

[1] Apache Hadoop. https://hadoop.apache.org/.
[2] Apache Spark. http://spark.apache.org/.
[3] CUDA C programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[4] The OpenCL specification v2.0. https://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[5] TPC-H. http://www.tpc.org/tpch/.

[6] R. Appuswamy, C. Gkantsidis, D. Narayanan, O. Hodson, and A. Rowstron. Scale-up vs scale-out for hadoop: Time to rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 20:1–20:13, New York, NY, USA, 2013. ACM.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[8] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48, 2011.

[9] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In S. Dasgupta and D. Mcallester, editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 1337–1345. JMLR Workshop and Conference Proceedings, May 2013.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[11] I. El-Helw, R. Hofman, and H. E. Bal. Scaling mapreduce vertically and horizontally. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 525–535, Piscataway, NJ, USA, 2014. IEEE Press.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[13] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[14] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[15] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

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

[18] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[19] W. He, H. Cui, B. Lu, J. Zhao, S. Li, G. Ruan, J. Xue, X. Feng, W. Yang, and Y. Yan. Hadoop+: Modeling and evaluating the heterogeneity for mapreduce applications in heterogeneous clusters. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 143–153, New York, NY, USA, 2015. ACM.

[20] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. Memcachedgpu: Scaling-up scale-out key-value stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 43–57, New York, NY, USA, 2015. ACM.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[22] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1235–1246, New York, NY, USA, 2014. ACM.

[23] F. Ji, H. Lin, and X. Ma. Rsvm: A region-based software virtual memory for gpu. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 269–278, Piscataway, NJ, USA, 2013. IEEE Press.

[24] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62, New York, NY, USA, 2012. ACM.

[25] T. Karnagel, R. Mueller, and G. Lohman. Optimizing gpu-accelerated group-by and aggregation. 2015.

[26] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. Mcc-db: Minimizing cache conflicts in multi-core processors for databases. *Proc. VLDB Endow.*, 2(1):373–384, Aug. 2009.

[27] P. Li, Y. Luo, N. Zhang, and Y. Cao. Heterospark: A heterogeneous CPU/GPU spark platform for machine learning algorithms. In *10th IEEE International Conference on Networking, Architecture and Storage, NAS 2015, Boston, MA, USA, August 6-7, 2015*, pages 347–348. IEEE, 2015.

[28] T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang. hstorage-db: Heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, 5(10):1076–1087, June 2012.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

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

[31] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[32] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the 17th International Conference on Data Engineering*, pages 567–574, Washington, DC, USA, 2001. IEEE Computer Society.

[33] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. *SIGPLAN Not.*, 48(4):407–418, Mar. 2013.

[34] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 351–362, New York, NY, USA, 2010. ACM.

[35] J. A. Stuart and J. D. Owens. Multi-gpu mapreduce on gpu clusters. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1068–1079, Washington, DC, USA, 2011. IEEE Computer Society.

[36] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[37] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. Gdm: Device memory management for gpgpu computing. *SIGMETRICS Perform. Eval. Rev.*, 42(1):533–545, June 2014.

[38] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with gpus. *Proc. VLDB Endow.*, 7(11):1011–1022, July 2014.

[39] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[41] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.