GHive: A Demonstration of GPU-Accelerated Query Processing in Apache Hive

Haotian Liu[1], Bo Tang[1,5], Jiashu Zhang[1], Yangshen Deng[1], Xinying Zheng[1], Qiaomu Shen[1], Xiao Yan[1], Dan Zeng[1], Zunyao Mao[1], Chaozu Zhang[1], Zhengxin You[1], Zhihao Wang[1], Runzhe Jiang[1], Fang Wang[2], Man Lung Yiu[2], Huan Li[3], Mingji Han[4], Qian Li[5], Zhenghai Luo[5]
[1] Southern University of Science and Technology, [2] The Hong Kong Polytechnic University
[3] Aalborg University, [4] University of Massachusetts Amherst, [5] Huawei Technologies Co., Ltd.
Corresponding author: tangb3@sustech.edu.cn

ABSTRACT

As a distributed, fault-tolerant data warehouse system for large-scale data analytics, Apache Hive has been used for various applications in many organizations (e.g., Facebook, Amazon and Huawei). Meanwhile, it is a common practice to exploit the large degrees of parallelism of GPU to improve the performance of online analytical processing (OLAP) in database systems. This demo presents GHive, which enables Apache Hive to accelerate OLAP queries by jointly utilizing CPU and GPU in intelligent and efficient ways. The takeaways for SIGMOD attendees include: (1) the superior performance of GHive compared with vanilla Hive that only uses CPU; (2) intuitive visualizations of execution statistics for Hive and GHive to understand where the acceleration of GHive comes from; (3) detailed profiling of the time taken by each operator on CPU and GPU to show the advantages of GPU execution.

1 INTRODUCTION

Hive was open-sourced by Facebook in 2008 and designed for big data analytics at massive scale [7]. Hive provides an SQL-like interface on top of Hadoop MapReduce [1], and thus users can run their queries with high-level SQL semantics instead of implementing low-level MapReduce jobs. Due to its rich and important applications, Hive enjoys rapid technical development from the community, and there are already more than 290 contributors that improve Hive on more than 25,600 issues. For example, Yin et al. introduce optimized columnar file format, physical optimizations, and vectorized query execution [5]. Hortonworks Inc. enhances Hive in four different aspects, i.e., SQL and ACID support, optimization techniques, runtime latency, and federation capabilities [3]. Up to now, Hive can run on various execution engines (e.g., Hadoop MapReduce, Apache Tez and Spark) by converting an analytical query to a set of executable jobs and using Hadoop's resource negotiator YARN for scheduling.

In the database community, many systems have been proposed to improve the performance of Online Analytical Processing (OLAP) with GPUs. In this paper, we present the GHive system, which improves performance of Hive via CPU-GPU heterogeneous computing. Different from existing systems that process OLAP queries using a single GPU [2, 4], GHive supports distributed query processing on multiple machines with a tailored execution engine. The execution engine comes with optimized implementations of SQL operators and judiciously decides to use GPU or CPU to execute each job in a query for efficiency. Other aspects of Hive, such as the SQL interface and Yarn-based job scheduling, are unmodified in GHive for backward compatibility. Specifically, GHive consists of three major technical components.

- Data transfer model: gTable. Jobs that are run on GPU require to transfer data from the address space of Java programs on CPUs to the video memory of GPUs. We design a columnar data model called gTable, which ensures that each column takes up consecutive memory and can be built in CPU memory on the fly for efficient data transfer.
- GPU-based operator library: Panda. To support job execution on GPUs, we develop a GPU-based SQL operator library named Panda, which meets the goals of efficiency, generality, and extensibility. We also devise an indexing-based processing model to reduce the amount of data transferred to GPUs.
- Heterogeneity-aware scheduling scheme. Hive spawns a set of jobs for each query, and running all these jobs on either CPU or GPU could yield sub-optimal performance. We propose a model-based approach to estimate job execution costs on both CPU and GPU, with which each job is placed intelligently for efficiency.

In this demo, we will show the performance of GHive on OLAP workloads and compare with vanilla Hive. The demo system will prepare the data and queries of the famous SSB benchmark, and the attendees can select queries to run on both Hive and GHive or write their own queries. For each query, the demo system tracks the execution progress and illustrates the overall performance for Hive and GHive with visualizations, such that the attendees can compare Hive with GHive and understand where the acceleration of GHive comes from. Moreover, we will also decompose the overall query execution time among the jobs and profile the involved operators on both CPU and GPU, with which the attendees can intuitively understand the advantages of GPU execution.

2 THE GHIVE SYSTEM

In this part, we introduce the key designs of GHive. Query execution in Hive works as follows. First, a user submits an SQL query via

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

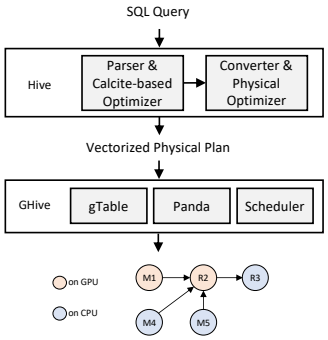

Figure 1: Architecture of GHive, "M(R)" for Map(Reduce) job

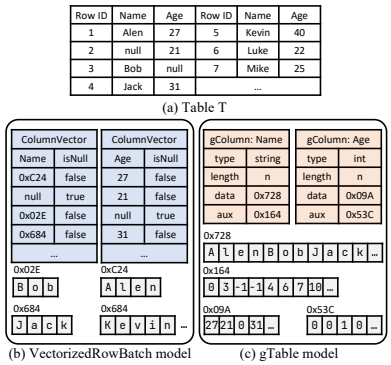

Figure 2: Data models for GPU-based operators

Command Line Interface or HiveServer2. The SQL query is parsed to an *Abstract Syntax Tree*, and then passed to the *Calcite-based Query Optimizer* to generate its optimized *logical plan*. The logical plan is converted into a *physical plan* and then passed to the Physical Optimizer to generate a *vectorized physical plan* [5]. Finally, the vectorized physical plan is translated into a set of executable Map/Reduce jobs and passed to the underlying execution engine such as Apache Tez. The data dependencies among the Map/Reduce jobs can be modeled using a directed acyclic graph (DAG), in which a job takes input from its incident jobs.

As shown in Figure 1, GHive reuses the components of vanilla Hive until generating the vectorized physical plan and executes each job heterogeneously on either CPU and GPU. For example, in Figure 1, job M1 and R2 are executed on GPU while job R3, M4 and M5 are executed on CPU. This architecture ensures backward compatibility as only the execution engine is changed. There are three key components in the GHive execution engine, i.e., data model gTable for the data transfer between CPU and GPU, SQL operator library Panda for job execution on GPUs, and a heterogeneity-aware scheduler that places each job on GPU or CPU.

2.1 Data Transfer between CPU and GPU

In vanilla Hive, the Map/Reduce jobs are all executed on CPUs by running Java Virtual Machine (JVM). To accelerate query processing, GHive executes computation-bound jobs on GPUs by calling CUDA (or OpenCL) APIs using C++. Thus, GHive needs to transfer data from CPU memory to GPU memory for jobs scheduled on GPUs. For efficient data movement, we design tailored data model and data movement strategy. We also parse the operator running plan for GPU jobs such that they can be correctly executed.

Data model. Hive stores tables using the VectorizedRowBatch model, for example, table T in Figure 2(a) is stored as the VectorizedRowBatch in Figure 2(b). Specifically, VectorizedRowBatch uses a columnar layout, where each column in table T (e.g., *Age*) is stored as a ColumnVector, which supports both primitive data types (e.g., int, float) and derived data types (e.g., string, StructColumnVector, DecimalColumnVector). For a string column (e.g., *Name* in T), ColumnVector stores the references to the strings. For example, the reference of Alen (i.e., 0xC24) is stored in the ColumnVector of Name. For primitive data types (e.g., *Age* in T), ColumnVector directly stores the values, see the ColumnVector of *Age* in Figure 2(b). To handle missing values, each ColumnVector has an *isNull* array, which is a bitmap used to indicate whether the value of a row is null or not for this column.

VectorizedRowBatch is inefficient for moving data from GPU to CPU as table columns with string type do not take consecutive space in memory. To exemplify, the *Name* of the 3rd and 4th rows of table T, i.e., "Bob" and "Jack", are located at two nonadjacent addresses 0x02E and 0x684, respectively. It is well-known that transferring small and non-consecutive memory blocks cannot fully exploit PCIe bandwidth and thus has high overhead. The VectorizedRowBatch model is also inefficient for job execution on GPUs as reading non-consecutive blocks from GPU memory is slow.

Motivated by the above limitations, we propose a new data model named gTable, which is illustrated in Figure 2(c), to support efficient data movement and GPU-based job execution. Similar to VectorizedRowBatch, gTable adopts a columnar layout and each table column is stored as a gColumn in gTable. Each gColumn has four meta-attributes, i.e., column data type (type), the number of rows (length), the reference to the array of data values (data), and auxiliary information (aux). Different from ColumnVector in VectorizedRowBatch, our gColumn concatenates all strings (except for null) of a string column in the data array and thus guarantees that the strings are located consecutively in memory. We provide such an example using column *Name* in Figure 2(c). The aux array stores the starting position and ending position of each string in the data array. The starting position and ending position of rows with null values are both set as -1. For columns with primitive data types (e.g., *Age*), gColumn stores the values in the data array, and uses a bitmap as the aux array to indicate rows with null values.

Data movement. To conduct data movement, we use a *ByteBuffer* in CPU memory that can be accessed by both Java and C++. In Hive, the input data of Map/Reduce jobs come either row by row or as a VectorizedRowBatch object. In both cases, we append new data to the ByteBuffer on-the-fly. ByteBuffer stores each column of a table separately using the same data array format in gColumn. After all data of a table are collected, we directly access the ByteBuffer in C++ and transform it into the gTable model by address referencing without data copy. Then, gTable is moved from CPU memory to GPU memory via PCIe, and the GPUs execute their assigned jobs. After that, the computation results are moved from GPU memory to CPU memory as a *Result* object in the address space of C++

Table 1: Operators in our GPU-based library Panda

Operators	Supported Features
Hash join	multiple keys, inner join, outer join, semi join
Sort merge join	multiple keys, inner join, outer join, semi join
Group by	multiple keys, avg(), sum(), max(), count()
Select	concat(), substr(), math operations
PTF	multiple keys, avg(), sum(), max(), rank()

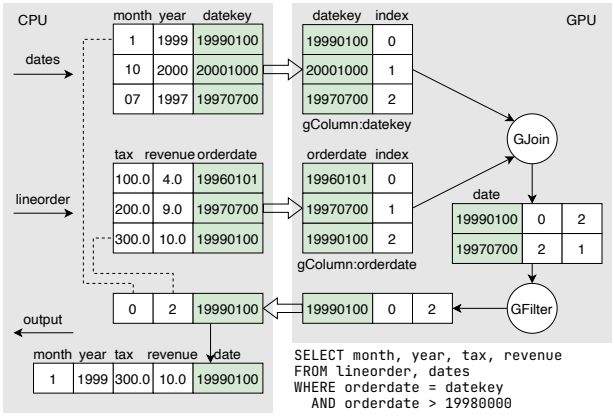

Figure 3: Example of the indexing-based processing model

program, which is then transferred to the address space of Java code and treated as the outputs of GPU jobs.

Operator extraction. In Hive, each Map/Reduce job executes some operators in the query (e.g., select and join), and an operator tree is used to describe the data dependencies among the operators in a job. Thus, for a job that is scheduled on GPU, we need its operator tree such that the GPUs can run the required operators in the right order. For this purpose, we parse the physical plan generated by Hive for a job, which is a structured text document. Specifically, we extract the involved operators by keyword matching and recover the dependencies among the operators by analyzing the structure of the text document.

2.2 GPU Operator Library Panda

To support job execution on GPU, we develop a GPU-based library for SQL operators named Panda following three principles. (i) Generality, Panda supports a complete set of SQL operators and advanced features as shown in Table 1. (ii) Efficiency, the operator implementations in Panda are extensively optimized for performance. (iii) Extensibility, Panda provides general interface, and thus developers can use it in other big data systems (e.g., Spark and Flink) or implement their specialized operators upon it.

Due to limited PCIe bandwidth, moving data between CPU memory and GPU memory is a major bottleneck in GHive. Thus, the operators in Panda adopt an *indexing-based processing model* for execution, which only moves the required columns (instead of all columns) of the data tables to GPU. This can be done easily as we store each column separately in gColumn. We provide such an example in Figure 3, where the query first joins the lineorder and dates tables, and then conducts filtering. We only move gColumn:datekey of table dates and gColumn:orderdate of table lineorder into GPU memory, as only these two columns will be used by the operators (i.e., join and filter). Before executing operators in GPU, we index each row of gColumn using its row number in the raw tables. The row indexes are kept in the execution process and passed to subsequent operators during execution. For example, in Figure 3, for the first row of the join result table (i.e., date), "0" means that it involves the first row of table lineorder while "2" means that it involves the third row of table dates, and the row indexes are passed to the filter operator. After the execution results are moved to CPU memory, we use the row indexes to fetch the required rows in the data tables to construct the final results, as shown in the lower left corner of Figure 3.

2.3 Heterogeneity-aware Job Placement

In GHive, GPU-based job execution enjoys high parallelism but the overhead of moving data between CPU memory and GPU memory may outweigh the benefits of faster computation. Thus, we design a heterogeneity-aware job scheduling scheme that judiciously places each job on either CPU or GPU.

Denote the end-to-end execution time of a job J on CPU and GPU as $T_C(J)$ and $T_G(J)$, respectively. We execute J on GPU if the following condition holds

$$\frac{T_C(J) - T_G(J)}{T_C(J)} \geq \theta,$$

where θ is a threshold that can be tuned by users (0.2 by default).

Job execution time on CPU. We estimate $T_C(J)$ as

$$T_C(J) = \sum_{\forall op_i \in J} f(op_i, n_i), \tag{1}$$

where op_i is an operator in job J, n_i is the input data size of operator op_i, and $f(op_i, n_i)$ if the execution time of op_i. We obtain the form of $f(op_i, n_i)$ (e.g., linear or quadratic function) by complexity analysis and fit the parameters (e.g., the slope in a linear function) via micro-benchmark experiments.

Job execution time on GPU. We estimate $T_G(J)$ as

$$T_G(J) = T_{pre}(J) + T_{exec}(J) + T_{post}(J), \tag{2}$$

where (i) T_{pre} is the time to move data from CPU memory to GPU memory to prepare for GPU execution; (ii) T_{exec} is the time to run the operators in J on GPU and its expression resembles eq.(1); (iii) T_{post} is the time to post-process the GPU execution results, which includes moving results to CPU memory and constructing exact results for our indexing-based processing model. Note that we do not consider disk and network I/O in the cost models above as these overheads are the same for both CPU and GPU execution.

3 DEMONSTRATION PLAN

In this demo, we plan to show SIGMOD attendees: (i) the superior performance of GHive over Hive in query execution time; (ii) intuitive visualizations for the execution process of Hive and GHive to understand where the performance gain of GHive comes from; (iii) fine-grained profiling on the task and operator level to show the advantages of GPU execution.

Demo setup. We will run Hive and GHive on the data and queries of the famous Star Schema Benchmark (SSB) [6]. We will deploy

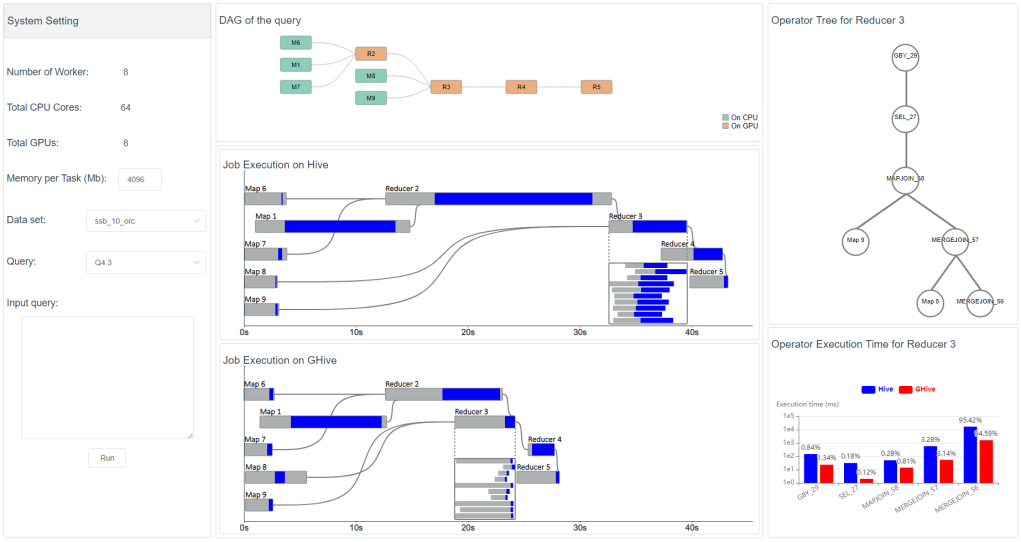

Figure 4: A screenshot of the GHive Demo front-end (best viewed in color and zoomed)

	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Hive	35.92	26.66	26.26	224.29	208.20	197.58	225.65	199.90	196.72	50.47	265.98	238.93	251.89
GHive	35.54	26.40	25.98	142.06	121.27	117.65	173.45	128.05	117.92	37.80	180.93	150.71	167.78

Table 2: End-to-end execution time (in seconds) comparison on the SSB queries, the scale factor is 200

both Hive and GHive on a remote cluster and our demo system will connect to the cluster using a web-based front-end. We plan to use the p3.2xlarge instances on AWS as worker machines, which come with Intel(R) Xeon(R) CPU E5-2686 with 4-core 2.30GHz, 60GB memory and NVIDIA Tesla V100 GPU with 16GB video memory.

A screenshot of our demo system is shown in Figure 4, and we introduce how attendees can interact with the system as follows.

Configuration. On the left panel, attendees can check system paramaters and set the memory size for each task. Attendees can adjust the scaling factor of the SSB data and choose the SSB queries to run with mouse clicks. Moreover, attendees can also define their own queries on the SSB data. When user presses the "run" button, the query will be run on both Hive and GHive for comparison.

Performance comparison. Once attendees run a query, its DAG of Map/Reduce jobs will be shown on the top of the middle panel, and we mark the jobs that are executed on GPUs by GHive. After both Hive and GHive finish query processing, the bottom of the middle panel compares their query execution time. We report the performance of Hive and GHive on SSB with 8 workers in Table 2. The results show that GHive consistently outperforms Hive for all queries and the speedup can be up to 1.8 times. The middle panel also shows the job level profiling results of query execution for Hive and GHive. In particular, each job is represented as a bar, whose starting and ending positions indicate job start and finish time. The highlighted part of each bar indicate the computation time of the job while the gray parts indicate overheads such as disk I/O and network. This allows attendees to reason where the performance gain of GHive comes from. For example, in the example of Figure 4, GHive outperforms Hive because jobs *Reducer2* and *Reducer3* have significantly shorter computation time by using GPU.

Detailed profiling. By clicking a job in the middle panel, the execution statistics of its tasks (a job spawns many tasks on the workers for both Hive and GHive) are shown using a bar chart. Moreover, on the right panel, the operator tree of the job is also illustrated. We also show the percentage of the time taken by each operator in job execution and the execution time of the operators in both Hive and GHive. This helps users to understand the advantage of GPU execution. Note that each operator is executed by many tasks in parallel and we use the total execution time within all tasks. For the example in Figure 4, we can see that the operator *MERGEJOIN_56* takes a large portion of the running time of *Reducer3* for Hive while GHive significantly reduces its running time using GPU.

REFERENCES

[1] 2021. *Apache Hadoop.* https://hadoop.apache.org/
[2] 2021. *OmniSciDB.* https://www.omnisci.com/platform/omniscidb
[3] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. 2019. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *SIGMOD*. 1773–1786.
[4] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in gpu-accelerated analytical engines. In *CIDR*.
[5] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in apache hive. In *SIGMOD*. 1235–1246.
[6] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252.
[7] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *PVLDB* 2, 2 (2009), 1626–1629.