Performance Analysis of Big Data ETL Process over CPU-GPU Heterogeneous Architectures

Suyeon Lee
Sogang university, Seoul, Korea
leesy0506@sogang.ac.kr

Sungyong Park
Sogang university, Seoul, Korea
parksy@sogang.ac.kr

***Abstract*—While GPUs have been utilized in the analysis stage of big data processing, the demand for GPU use in the extract-transform-load (ETL) stage has recently been increasing. There have been several research efforts to use GPUs for query processing in database systems. However, most of them are mainly focused on batching input data as much as possible to increase the overall throughput with GPUs. Moreover, they try to execute a query plan by only using a single device (only CPU or only GPU), based on the assumption that the PCIe transfer overhead between CPU and GPU is significant. In contrast, this paper presents several interesting observations and reveals that it is better to use a CPU-GPU heterogeneous query plan in some cases. For example, the heterogeneous query plan proved to be efficient when the input data size is small. This is because the PCIe overhead is not as serious as expected for small-sized data and can be offset with performance gains by partially using the GPU. In addition, even when the data size is large, it is sometimes better to use the CPU-GPU heterogeneous query plan rather than only the GPU plan, since some relational operations prefer the CPU over the GPU. To demonstrate these uses, this paper presents several experimental results and their analyses with Spark SQL in a CPU-GPU heterogeneous computing environment.**

***Index Terms*—Big Data, ETL, Spark SQL, GPU**

I. Introduction

[1] Big data processing is generally divided into an extract-transform-load (ETL) stage and an analysis stage. In the ETL stage, big data platforms receive raw data, perform pre-processing on the received data, and then store the results in a separate data sink. In the analysis stage, machine learning or graph processing jobs are executed on the data generated by the ETL stage. Due to the volume of data handled at the same time and its iterative nature, the use of GPUs has been an active research area, mainly in the analysis stage.

Attempts to introduce GPUs at the ETL stage have only been made recently. Since managing and cleaning up the received data are the main concerns at the ETL stage, most of the tasks rely heavily on the existing query operations developed for database systems. However, two problems arise, since the characteristics of the big data environment are overlooked. The first is that many studies focus only on large batch data to increase the overall throughput with GPUs. This is not suitable for big data environments where the requirements for input data size change frequently. For example, in the case of streaming processing, which is common in the big data environment, processing small amounts of data continuously in real-time is important. However, previous studies are often limited to either a workload with initially large input data [1] [2], or the case where the data is artificially made bigger so that the GPU can handle them properly [3] [4]. The second is that the PCIe transfer overhead is excessively considered in a system design, since it is usually referred to as one of the biggest bottlenecks between the CPU and the GPU. For this reason, many studies choose a method to make the best use of a single device during the entire query process. That is, while one query job is being executed, all operations are executed either only with the CPU or only with the GPU [5] [6]. Although there was a recent study on separating operations between the CPU and the GPU in a streaming process [7], it was proposed over an integrated GPU architecture where both CPU and GPU share system memory and there is no PCIe overhead at all.

In contrast, this paper reveals that the CPU-GPU heterogeneous query plan is suitable for the ETL stage in the big data environment. We justify our approach from two perspectives. First of all, when the data size is small, it is better to run query operations on the CPU than on the GPU in terms of total execution time. Our results also show that CPU-GPU heterogeneous query plans, which offload only partial operations to the GPU, often give the best performance. This indicates that there are different device preferences for each query operation. Besides, CPU-GPU heterogeneous computing is convincing because the PCIe transmission time is not as severe as expected when the data size is small. Therefore, this overhead is negligible compared to the performance gain by the partial use of the GPU. Even with large data, there are still tasks that prefer CPU over GPU depending on the workload. Therefore, the CPU-GPU heterogeneous query plan can show better performance in many respects than blindly executing all query operations on a single device.

To prove this, we conducted various experiments using Apache Spark SQL [8] that supports CUDA-based GPU acceleration from Spark 3.0.0 released in June 2020.

This paper makes the following specific contributions.

- This work is the first attempt to analyze problems when the GPU is applied to the ETL stage.
- This work reveals the necessity of the CPU-GPU heterogeneous query plan at the ETL stage.
- This work is the first analysis of GPU acceleration in Spark SQL, both on a single machine and a cluster. Our observations can answer many open questions regarding performance maximization in Spark SQL.

[1]This work was supported by IITP grant funded by MSIT (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System).

- This work uses workloads from the TPC-DS benchmark [9], a state-of-the-art decision support benchmark, and also queries selected according to specific criteria rather than random queries.

II. Motivations and Related Work

Motivations. In the field of big data processing, there have been numerous attempts to introduce GPUs as accelerators. So far, the focus was mainly on applying GPUs to analytics aspects such as machine learning applications. However, as the amount of data increases rapidly, there is a problem that computing power is insufficient not only at the data analysis stage, but also in the data preprocessing phase, which is called the ETL stage. In particular, in the emerging real-time analysis, the ETL process is a step that occurs every time before the analysis process, so its acceleration is essential.

The ETL process of big data can be divided into two main areas. One is batch processing, which processes a large amount of input data in a single process cycle. The other is stream processing, which processes small amounts of data in one process cycle, and there are infinite process cycles with fast data entry rates. In addition to constant active research on the batch process, there has been growing demand for stream processing in recent years. Furthermore, an architecture in which the two are combined and performed simultaneously is also emerging [10]. Therefore, the big data processing framework must consider the size of the data processed in one process cycle, from very small to very large.

Throughput-oriented Approach. In general, one of the reasons to use GPUs in any area is to improve overall throughput by increasing computing power. Earlier research efforts [1] [2] [11] [12] have been focused on this point of view. For this reason, the input data for the ETL process is limited to large batch data for processing in the GPU. On the other hand, input data size in the big data processing area varies and is dynamic, which means that small data must be considered when using the GPU. Existing studies have commonly used the batching method [3] [4] to handle small data so far. In other words, even when the input data is small, it aggregates the input data into large data to force an environment that is advantageous to introduce the GPU. For example, in a streaming environment, small-sized data must be processed continuously, but [3] [4] [5] [6] collected input data for a certain period of time to make a big enough batch to process in the GPU. This method is not appropriate because it excludes real-time processing, which is the biggest characteristic of a streaming environment.

Dedicated Use of CPU or GPU. Another problem is that existing studies consider the PCIe transfer overhead between CPU and GPU to be extremely large and the main performance bottleneck. For this reason, they adopted the *query granularity* method [5] [6] when assigning tasks to the CPU and GPU, which means that a specific query is designed to be executed using only the CPU or only the GPU [13] [14]. The focus was on how to fully utilize a single device. Recently, [7] proposed the *operation granularity* method that distributes different operations in a single query to different devices. However, it is based on the integrated CPU-GPU architectures where both CPU and GPU share the system memory in the CPU and thereby have no PCIe overheads at all. Meanwhile, [15] is one of the early attempts to introduce GPUs to Spark, which focuses on the overall implementation of GPU query processing in Spark. Although this work suggested CPU-GPU heterogeneous query plans, it only considered CPU-intensive operations advantageous to offload to the GPU.

Our Observations. Contrary to previous works, this paper reveals that CPU-GPU heterogeneous computing can cope well with extremely small-sized data as well as large amounts of data. This paper provides a comprehensive evaluation study using various factors and workloads to show the effectiveness of a CPU-GPU heterogeneous query plan rather than a plan using only a single device. This claim is reasonable because there are apparent device preferences for each operation. Also, the PCIe overhead is not as much as expected in this process. The smaller the data size, the stronger the tendency, and even if the data size increases, the same results are still shown.

III. Experimental Setup

A. Framework

When processing big data, one of the most important design issues is to build a system so that data extraction, data transformation, data loading, data analysis, etc. can be processed at once by pipelining. Spark has emerged as a representative runner of the big data framework to enable in-memory-based fast pipeline processing with the support of various libraries. Also, Spark supports both batch processing and streaming processing, making it the most appropriate big data processing framework to address our motivation mentioned in section II.

Among Spark's various core libraries, Spark SQL [8] is used in the ETL process for data extraction, data transformation, and data loading from various sources. When Spark SQL deals with a relational approach to process data, it establishes a query plan in four phases to convert the written SQL code into internal Java code. The first step is to build a logical plan by parsing SQL statements and resolving references. The second step is to create an optimized logical plan by using several query optimization techniques, such as predicate pushdown. The third step is to create a physical plan by specifying the functions that must actually be executed to perform the optimized logical plan. Finally, in the code generation step, the SQL code is converted into Java bytecode that can be executed in the Spark JVM.

From Spark 3.0.0, the execution method of the above query plan can be extended to use GPU in Spark SQL. NVIDIA's Spark-Rapids [16], cuDF [17], and CUDA are used to implement this mechanism. After an optimized logical plan is created, Spark assigns an actionable function to each operation, and then Spark-Rapids extends this stage to replace operation functions with GPU versions. Currently, as long as the function is normally implemented in the Spark-Rapids library, all operations are unconditionally replaced by functions using the GPU. In other words, the ETL stage can

TABLE I: TPC-DS workloads: For network-intensive queries, label is N(query number). Similary, we used I for I/O-intensive queries, C for CPU-intensive queries. Q88 is network-intensive and I/O-intensive, thus labeled as NI88.

Query Number	Characteristics			Query Label
	Numerous Data Scans	Lots of Data Shuffling	Numerous Computations	
Q64		✓		N64
Q94		✓		N94
Q88	✓	✓		NI88
Q78	✓			I78
Q82	✓			I82
Q22			✓	C22
Q67			✓	C67
Q70			✓	C70

TABLE II: Execution type definitions used in the experiments.

Execution Type	Execution Option
Type1	All CPU
Type2	All GPU
Type3	without GPU CSVScan
Type4	without GPU Filter
Type5	without GPU Project
Type6	without GPU ShuffleExchange
Type7	without GPU HashAggregate
Type8	without GPU BroadcastHashJoin
Type9	without GPU Sort
Type10	without GPU Expand
Type11	Type4 + Type5 + Type7
Type12	Type4 + Type5 + Type7 + Type8

be processed using either only CPU functions or only GPU functions.

B. Workload

For the workloads of the experiments, we used the TPC-DS benchmark [9], which is the one most commonly used for evaluating query processes. For accurate experiments, we chose queries with specific environments rather than arbitrary queries. The selected queries were classified as network-intensive, I/O-intensive, and CPU-intensive queries. We labeled each query for convenience and used it in place of a specific query number in the rest of the paper. It is organized in Table I. Also, we used *scale factors* 1, 10, and 100 according to the experiment, representing raw data size in GB. However, the batch data size processed by a single core is much smaller, depending on the workload and configuration.

C. Execution Option

This paper argues that a query plan using a mixture of CPU and GPU is sometimes preferable to a plan using only one of the two resources. To investigate this issue, we measured the execution times for three cases: using only CPU, only GPU, or both CPU and GPU. For each case, we defined execution types depending on execution options, which are summarized in Table II. The first case is Type1, and the second case is Type2. The third case is divided from Type3 to Type12. When executing the queries using the GPU, users can specify some functions not to be offloaded to it. For example, if the user wants a project function for executing the `SELECT` operation to use a CPU other than a GPU, they can submit the entire job with the option `spark.rapids.sql.exec.ProjectExec` turned off. In this way, we measured the execution time when each function of CSV scan, filter, project, shuffle exchange, hash aggregate, broadcast hash join, sort, and expand is not replaced by the GPU. We also observed when multiple operations are not replaced by the GPU, such as Type11 and Type12. We averaged the execution times after five iterations and compared the speedup by normalizing with respect to the time using only the CPU or only the GPU, depending on the purpose of an experiment.

TABLE III: Configurations for the experiments. The configurations are based on Spark 3.0.0, Rapids 0.1.0, cuDF 0.14, and CUDA 10.1.

Single Machine Configuration	
CPU Cores	10
Spark Shuffle Partitions	5
GPU Total Memory	4G
GPU Pinned Memory	1G
Concurrent GPU Tasks	1
Cluster Configuration	
Number of Executors	2
Driver Memory	4G
Memory per Executor	12G
CPU Cores per Executor	12
GPU Memory per Executor	8G
GPU Pinned Memory per Executor	2G
Concurrent GPU Tasks per Executor	2
Spark Shuffle Partitions	5
Spark Max Partition Bytes	512M

D. Testbed Configuration

For the experiments, we configured two types of testbeds using a single machine and a three node cluster (one master and two worker nodes). The single machine is equipped with an AMD Ryzen 5 3600 CPU and NVIDIA GeForce RTX 1650 GPU, where the Spark configurations are summarized in Table III. For the experiments running in a cluster, we decided to use two types of cluster configurations to investigate issues when the performance of the CPU and GPU is not well balanced in the worker nodes (i.e., fast CPU with less performing GPU vs. slow CPU with highly performing GPU, etc.). For instance, each worker node in the first cluster (we call this *Cluster1* in this paper) has an AMD Ryzen 9 3900X CPU and NVIDIA GeForce RTX 2080 Super GPU, while each worker node in the second cluster (we call this *Cluster2* in this paper) has an Intel Xeon Silver 4210 CPU and NVIDIA GeForce RTX 2080 Ti GPU. Both *Cluster1* and *Cluster2* have the same Spark configurations described in Table III.

IV. Results

In order to understand the issues that affect the performance of Spark SQL and to conclude that CPU-GPU heterogeneous computing is necessary, we conducted the experiments considering three parameters: *batch data size*, *operation selection*, and *overall workload*.

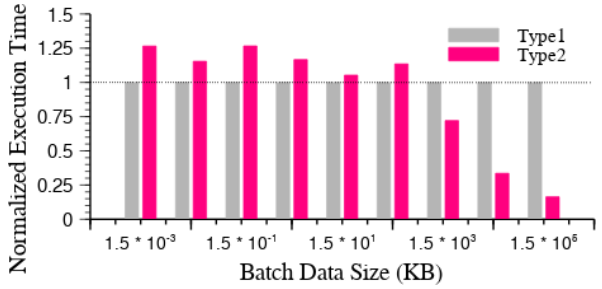


Fig. 1: Normalized execution times by varying batch data size from 1.5 B to 150 MB.

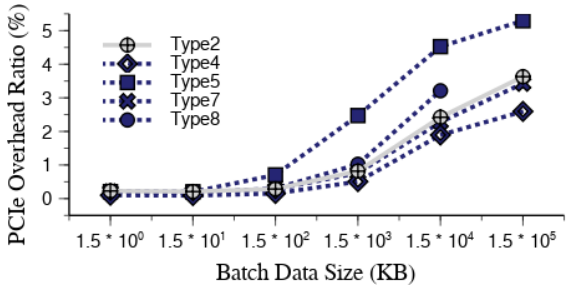


Fig. 2: PCIe overhead ratio by varying batch data size from 1.5 KB to 150 MB.

A. Batch Size per Task

Effect of Batch Data Size. We first varied the batch data size processed by a task from 1.5 B to 150 MB and compared the two cases where the GPU is either used or not used in Figure 1. For the workload, we used simple select-project-join (SPJ) queries. As shown in Figure 1, if the batch data size is less than a specific size, in our case about 1.5 MB, it is more advantageous to use CPU than to use GPU. However, after that, the effect of GPU use is soaring, so it is more beneficial to choose a GPU.

PCIe Overheads. We also compared the cases of using only GPUs and using both CPU/GPU devices in Figure 2 to show that the PCIe overhead can be ignored when the batch data size is small. We obtained the PCIe transfer time by calculating the total amount of time spent on CUDA `MEMCPY` using NVIDIA Nsight Systems, a GPU profiler. For the workload, we used the same SPJ queries as before and the execution types described in Table II. In Figure 2, the y-axis is the ratio of PCIe overhead to the entire process, where the value is calculated as follows.

$$(PCIeTransferTime \ / \ TotalExecutionTime) \ \times \ 100$$

As shown in Figure 2, if the data size is less than 1.5 MB, the PCIe overhead ratio is small, which is about 0.2~0.3%. That is, the entire PCIe transfer time is only 500~700 microseconds, while the total execution time is shown in seconds. It is noteworthy that the data transfer operation through PCIe occurs about 500 times in the workload. As the batch data size increases, the number of operations does not increase, but the average execution time of a single operation rapidly increases. For example, if the data size was 1.5 MB, it took 28 microseconds, while it took 266 microseconds for 15 MB, and 2.56 milliseconds for 150 MB. However, if the data size is less than 1.5 MB, the operating time does not change significantly between 1 and 2 microseconds. Therefore, the PCIe overhead can be ignored until the data size reaches a certain point. In particular, even if queries use both CPU/GPU devices, the overhead is not significantly enhanced compared to Type2. Rather, in the case of Type4 and Type7, there is less PCIe overhead than when using only the GPU.

Summary. Considering that each computing core needs to process data with various sizes, it is essential to use CPUs with GPUs in a query process. In such a case, the PCIe overhead does not have a large amount of impact up to some data size.

B. Operation Selection

In this experiment, selected TPC-DS queries were performed while changing the execution types presented in Table II. Figure 3 and Figure 4 show the performance results in *Cluster1* and *Cluster2*, respectively. Each result is separated into two tables by a scale factor. The cell color indicates the speedup compared to Type1. The first column of each table is the base point and has the brightest color because it only uses the CPU. The larger the speedup, the darker the color. Some cells of Type9 and Type10 are gray because they do not use the corresponding operation of each execution type.

SQL Operations. In Figure 3 and Figure 4, we first classified which operation would be more advantageous to choose the CPU instead of the GPU. For this purpose, the results of Type3 to Type10 were analyzed compared to Type2. For Type4, Type6, and Type7 shown in the left table of Figure 3, there are many cases where those types perform better than Type2. For example, in the case of N64, the performance of Type4 is about 1.28 times better than Type2, Type6 is about 1.38 times better, and Type7 is about 1.78 times better. In addition, Type 7 shows performance that is faster or comparable to Type 2 in every case except for C22. Similar results are also shown in the left table of Figure 4. Therefore, when the data size is small, filter, shuffle exchange, and hash aggregate operations need to select a CPU. If the data size gets larger, it is advantageous to use a GPU in most cases. However, this is not always the case. In some queries, such as N64, N94, and NI88, corresponding operations still have the same CPU preferences even when the scale factor is 10. On the other hand, Type3 and Type9 performed poorly compared to Type2. In the case of the Type3, the performance drops sharply and is even worse than Type1. This was common regardless of the scale factor or cluster type. Therefore, CSV scan and sort operations should be executed on the GPU. In the cases of Type5, Type8 and Type10, more complex factors need to be considered because they are not clearly classified.

As discussed in subsection IV-A, there is currently data transfer overhead when using a GPU. Thus, if continuously executing operations are offloaded together to one device, the cost of data transfer between different devices can be reduced. For example, filter and project operations are usually executed in succession for `SELECT-WHERE` processing. The filter operation has a clear CPU preference, and the project operation is neutral. Therefore, if both operations are offloaded to the CPU, the intermediate overhead will disappear and

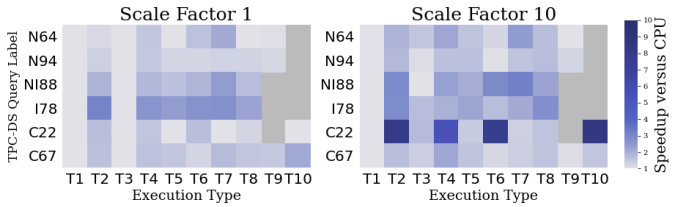


Fig. 3: Speedup values with respect to execution Type1 (all CPU) for scale factors 1 and 10 in *Cluster1*.

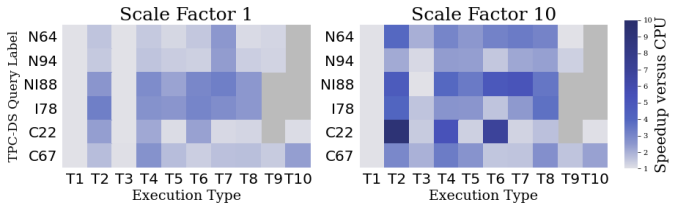


Fig. 4: Speedup values with respect to execution Type1 (all CPU) for scale factors 1 and 10 in *Cluster2*.

achieve further performance improvements. Figure 5 shows the results of these attempts. When multiple operations were set to be performed on the CPU, the execution time was further decreased in C67 and NI88. In Type11, filter and project operations are offloaded to the CPU at once. Hash aggregate operation has an obvious CPU preference, so it is also executed on the CPU. Notice that Type5 was not a better execution type than Type2 in Figure 4. However, in Figure 5, Type11 offsets the data transfer overhead and further improves performance. Similarly, since hash aggregate and broadcast join operations are often run in conjunction, Type12 also performed better than Type2. On the other hand, C22 and N64 are cases where these functions were not executed consecutively. This resulted in too much data transfer overhead, which led to a reduction in performance. It is important to consider not only the simple type of operation, but also the overall composition and order of the query plan.

Query Classification. The queries we used as workloads were classified as network-intensive, I/O-intensive and CPU-intensive. Therefore, analysis was required based on the classification of each query. As shown in Figure 3 and 4, C22 and C67 performed more poorly than Type2 in most cases, regardless of the scale factor. This indicates that the higher the computational workload, the more the effect of GPU with a large number of cores. Therefore, for CPU-intensive workloads, the CPU-GPU heterogeneous query plan can be inefficient in terms of execution time. On the other hand, in the cases of N64, N94, and NI88, many outperformed Type2. This shows that GPU use is not very efficient when there is a large amount of network shuffling, and therefore the CPU-GPU heterogeneous query plan can be used.

Cluster Environments. This experiment was conducted in two clusters with different environments. When looking at the cluster environment as a variable, the point to note is that Figure 4 has many more cells that are darker than Figure 3. It can also be seen in Figure 4 that the performance of most of the queries is much better than the performance

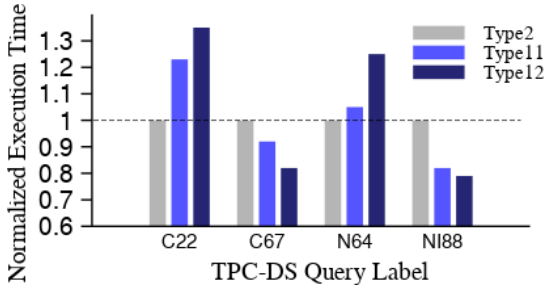


Fig. 5: Normalized execution times of Type11 and Type12 with respect to Type2 (all GPU) for scale factor 1 in *Cluster1*.

of corresponding ones in Figure 3. Note that *Cluster2* is equipped with a much worse CPU than *Cluster1*, and the GPU performance is a little better. Through this, it is expected that the GPU still has a major effect on performance, even in a CPU-GPU heterogeneous computing environment.

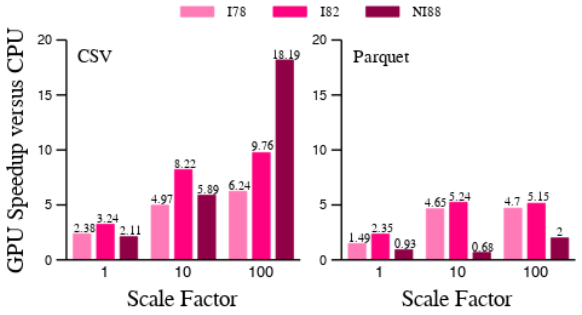
Summary. Some query operations such as filtering, shuffling, and aggregation have apparent CPU preferences. It is essential to build CPU-GPU heterogeneous query plan in light of data size, device preferences for each operation, and other nature of the query.

C. Overall Workload

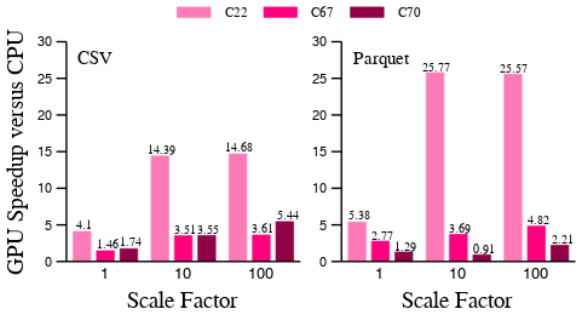
In this subsection, we configured workloads with different characteristics based on cluster availability, input data format, batch data size, and SQL operation type, then compared the performance of each workload. For the input data formats, we used Parquet [18] as a representative for a column store, and CSV as a representative for a row store.

Experiments on a Single Machine. Figure 6(a) shows the experimental results for queries classified as I/O-intensive, and Figure 6(b) depicts the results for queries classified as CPU-intensive. The y-axis in the figures indicates the GPU speedup compared to Type1. Higher values indicate better performance.

In terms of batch data size, the results show that GPU does not significantly improve performance when the scale factor is 1. When the scale factor reaches 10, the performance increases by at least three times when the GPU is enabled. Therefore, as this paper continues to point out, CPU-GPU heterogeneous computing could be one of the solutions for processing small-sized data with a GPU. In terms of workload characteristics, the overall GPU speedup was higher for I/O-intensive queries, but its maximum value was much higher for CPU-intensive queries. This shows that, as is known, the higher the computational workload, the greater the performance gain when using GPUs. In terms of input data format, using Parquet files results in a much shorter absolute execution time than using CSV files. This is because the Parquet file is originally a column store, so the coalesced memory access of GPU can help process the data. However, it can be seen that the GPU speedup is not very high in Parquet, except for C22. In the case of NI88 or C70, using the GPU was even slower than disabling it. This is a problem with Spark SQL. Since Parquet is a column store, Spark-Rapids calls the `HostColumnarToGPU` function, which takes a significant amount of time. For this reason, the



(a) I/O Intensive



(b) CPU Intensive

Fig. 6: GPU Speedup versus CPU by various factors in single machine: (a) is for I/O-intensive workload, and (b) is for CPU-intensive workload.

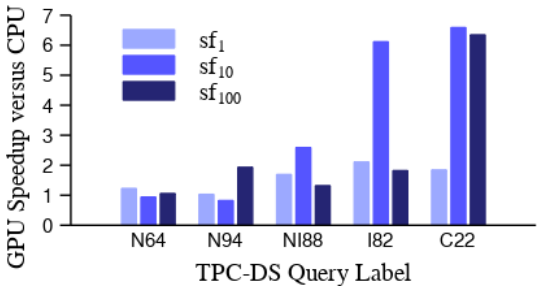


Fig. 7: GPU Speedup versus CPU by various factors in *Cluster1*.

broadcast operation continues to be delayed, resulting in a very long execution time and even timeout.

Experiments on a Cluster. Figure 7 shows the results of the experiments on *Cluster1*. Since there are multiple nodes, network-intensive queries were added. The input data format is fixed to CSV, since we concluded that Parquet might not be adequate to use the GPU as discussed previously.

As shown in Figure 7, the GPU speedup was highest in the order of CPU-intensive queries, I/O-intensive queries, and network-intensive queries. For network-intensive workloads, a large amount of data exchange results in significant data transfer overhead. Therefore, it may be desirable to perform CPU-GPU heterogeneous computing aware of each node's data locality.

Summary. Overall, CPU-GPU heterogeneous query plan can be most efficient when performing network-intensive queries or I/O-intensive queries dealing with small amounts of data.

V. Conclusion

We have identified that CPU-GPU heterogeneous computing should be applied to improve the performance of the ETL process, rather than using GPU blindly. We conducted three types of experiments in Spark SQL by varying batch data size, query types, and overall workload characteristics. The comprehensive evaluation results showed that a certain level of performance gain can be achieved by adequately distributing loads between CPU and GPU. The results from each experiment will necessitate the need for developing an adaptive algorithm to maximize the performance of the ETL process.

References

[1] W. Fang *et al.*, "Mars: Accelerating mapreduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 608–620, 2011.

[2] S. Hong *et al.*, "Gpu in-memory processing using spark for iterative computation," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 31–41.

[3] Z. Chen *et al.*, "Gpu-accelerated high-throughput online stream data processing," *IEEE Transactions on Big Data*, vol. 4, no. 2, pp. 191–202, 2018.

[4] M. Pinnecke *et al.*, "Toward gpu accelerated data stream processing." in *GvD*, 2015, pp. 78–83.

[5] A. Koliousis *et al.*, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 555–569.

[6] T. De Matteis *et al.*, "Gasser: An auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.

[7] F. Zhang *et al.*, "Finestream: Fine-grained window-based stream processing on cpu-gpu integrated architectures," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 633–647.

[8] M. Armbrust *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1383–1394.

[9] O. Nambiar *et al.*, "The making of tpc-ds," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, ser. VLDB '06. VLDB Endowment, 2006, p. 1049–1058.

[10] M. Kiran *et al.*, "Lambda architecture for cost-effective batch and speed big data processing," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 2785–2792.

[11] L. Chen *et al.*, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[12] C. Chen *et al.*, "Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1275–1288, 2018.

[13] P. Bakkum *et al.*, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: Association for Computing Machinery, 2010, p. 94–103.

[14] K. Wang *et al.*, "Concurrent analytical query processing with gpus," *Proc. VLDB Endow.*, vol. 7, no. 11, p. 1011–1022, Jul. 2014.

[15] Y. Yuan *et al.*, "Spark-gpu: An accelerated in-memory data processing engine on clusters," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 273–283.

[16] C. McDonald *et al.* (2020) Accelerating apache spark 3.0 with gpus and rapids. NVIDIA. [Online]. Available: https://developer.nvidia.com/blog/accelerating-apache-spark-3-0-with-gpus-and-rapids

[17] J. Hemstad. (2019) Rapids cuda dataframe internals for c++ developers. NVIDIA. [Online]. Available: https://developer.nvidia.com/gtc/2019/video/S91043

[18] D. Vohra, "Apache parquet," in *Practical Hadoop Ecosystem*. Springer, 2016, pp. 325–335.