Acceleration of hybrid CPU-GPU query execution engine in Arrow Format

Version of September 18, 2023

Kexin Su

Acceleration of hybrid CPU-GPU query execution engine in Arrow Format

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Kexin Su
born in Chongqing, China

Quantum Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft, the Netherlands
www.ewi.tudelft.nl

Acceleration of hybrid CPU-GPU query execution engine in Arrow Format

Author: Kexin Su
Student id: 5591597

Abstract

General-purpose GPUs, renowned for their exceptional parallel processing capabilities and throughput, hold great promise for enhancing the efficiency of data analytics tasks. At the same time, recent developments in query execution engines have integrated the support of OLAP operations in a way that benefits from the zero serialization overhead provided by the Apache Arrow memory format. In this project, our objective is to perform a study to evaluate the acceleration potential on GPUs of Arrow-based query execution engines, specifically with libcudf, a C++ GPU DataFrame library with Arrow format. With this purpose, we design and implement four micro-benchmarks for different operators to understand the characteristics of workloads that result in high acceleration, and their possible bottlenecks and limitations. When we exclude data transfer durations, inherently parallelizable workloads exhibit high potential for GPU acceleration. However, this advantage diminishes considerably when considering data transfer overheads. Stemming from these micro-benchmark outcomes, we designed an on-the-fly scheduler at the operator level to dynamically accelerate query execution engines in a hybrid CPU/GPU system. The scheduler can decide whether to distribute an operator on the CPU or GPU based on the input data location, data volume, data-related parameters, and the operator type so that we can accelerate query execution engines in a hybrid CPU-GPU system according to a statistics cost model. The conclusion is that, with the scheduler, we achieve a maximum of 4.88x speedup for Filter Operator, 2.52x speedup for Sort Operator, and 1.52x speedup for Copy Operator when handling an array of 1e8 in length.

Thesis Committee:

University supervisor:	Dr. Zaid Al-Ars, CE/QCE/EWI/TUD, TU Delft
Committee Member:	Dr. Asterios Katsifodimos, WIS/ST/EWI/TUD, TU Delft
Committee Member:	Yongding Tian, CE/QCE/EWI/TUD, TU Delft

Preface

Many thanks to Zaid.
Dedicated to my family.

Contents

Chapter 1

Introduction

Nowadays, GPUs (graphics processing units), which were originally designed for graphics processing purposes, have increasingly been used for general computing purposes. A modern CPU can only have dozens of cores, however, a modern GPU can have thousands of cores, which makes them naturally suited to perform parallel operations. Furthermore, GPUs have a much higher memory bandwidth than CPUs, allowing them to process data at a higher throughput. Given these advantages, GPUs are promising to be the next-generation query execution platform for big data Online Analytical Processing (OLAP).

Nevertheless, the memory capacity of a GPU is typically more limited than what a contemporary CPU can directly access. As a result, GPUs can only process data sets that fit within their memory constraints. This becomes particularly evident in big data applications where the vast amount of data surpasses the preloadable limit of the GPU. Similarly, in streaming data applications that demand real-time processing, data can't be preloaded onto the GPU either.

In situations where it's not feasible to directly preload data into GPU memory, unlike the case of CPUs which have a direct link to CPU memory, GPUs need to transfer the input data from CPU to GPU via the PCIe bus. Unfortunately, the PCIe bus offers a bandwidth that's considerably lower—almost two orders of magnitude less—than that of the GPU's memory bandwidth. This limitation suggests that, for applications that are data-intensive, it is not always worth distributing tasks from CPU to GPU. Moreover, even for compute-intensive applications, GPUs and CPUs are specifically suitable for different types of operators. Therefore, to get the highest possible performance in a hybrid CPU-GPU system, it's vital to strategically distribute operators between CPUs and GPUs.

Hybrid CPU-GPU computing is used in a wide range of applications that require intensive processing performance, such as High-Performance Computing [1], Machine Learning and Artificial Intelligence [2], and Scientific Computing [3]. Hybrid CPU-GPU computing can provide significant performance benefits for many applications that require high computational performance, enabling faster and more efficient processing of complex tasks.

1.1 Motivation

The Apache Arrow community is one of the game changers who want to leverage the power of hybrid CPU-GPU computing to improve the performance of data science applications.

In recent years, Apache Arrow has gained a lot of attention and popularity because it addresses some of the key challenges faced by big data and machine learning applications. It provides an efficient in-memory data format that can be shared across multiple programming languages and platforms which enables fast and efficient data transfer with less overhead between different systems and hardware, including GPUs.

Efforts have already been put into using GPUs to accelerate data science applications with Arrow. For example, researchers have already accelerated big data applications on GPUs to increase the throughput for searching, transforming, and merging data generated by AI and simulation workloads in Arrow formats [1]. Also, big data ingestion and reformatting can benefit from acceleration with dedicated hardware systems such as GPUs and FPGAs [4].

In this thesis, we will focus on using GPUs to accelerate Acero, a C++ query execution engine that is a sub-project under Apache Arrow. It uses the Arrow format as its core data representation in memory, allowing it to benefit from easy integration with accelerators. This engine is used to perform OLAP tasks on streaming big data sets that could require significant processing time. Accelerating these types of workloads can both improve the performance of these applications as well as increase the efficiency of system utilization.

1.2 Thesis goal

The main goal of the thesis is formulated as follows:

- Perform a study to evaluate the acceleration potential on GPUs of the Arrow Acero query execution engine

To fulfill the stated goal, a few research questions must be answered:

- Is it possible to accelerate Acero in a hybrid CPU-GPU system?
- What are the characteristics of workloads that result in high acceleration and what are the possible bottlenecks?
- What are the limitations of speedup achievable using CPU-GPU systems?

1.3 Outline

Chapter 2 contains detailed background information about the CPU-GPU hybrid computing, Apache Arrow, Acero, and Rapids libcudf.

Chapter 3 further explains some glossaries in the thesis context to present the core concepts, i.e., what is a query execution engine, what is the operator model in the query execution engine, and what a scheduler is in a query execution engine.

Chapter 4 contains the benchmark result of the CPU-GPU hybrid accelerated query execution engine. This micro-benchmark result serves as the primary basis for the design of the scheduler in the next chapter.

In Chapter 5, the architecture design and implementation detail of the scheduler will be presented. Morever, we will give an evaluation of the scheduler with the error metrics and the speedup ratio between using GPU acceleration and not using GPU acceleration.

In Chapter 6, we will answer the research questions that come up in Chapter 1, list the contributions and discuss future work.

Chapter 2

Background

Chapter 2 contains detailed background information about the CPU-GPU hybrid computing, Apache Arrow, Acero, and Rapids libcudf.

2.1 CPU-GPU hybrid computing

Due to their massive parallel computing power, GPUs are considered as next-generation high-performance computing engines with a large amount of CUDA cores, high bandwidth device memory, and scalability.

One of the first efforts to use GPUs to accelerate data-centric applications is to preload the complete dataset in GPU memory [5]. This approach has the advantage of vastly reducing data transfers between host and device. In addition, since the GPU RAM has a bandwidth that is roughly 16 times higher than the PCIe Bus (3.0), this approach is very likely to significantly increase performance. It also simplifies transaction management, since data does not need to be kept consistent between CPU and GPU.

However, the approach has some obvious shortcomings: First, the GPU RAM (up to $\approx$ 80 GB, NVIDIA ampere A100, the 80GB version) is rather limited compared to CPU RAM (up to $\approx$ 4 TB, CPU AMD EPYC 7702P) as shown in Figure 2.1, meaning that either only small data sets can be processed, or that data must be partitioned across multiple GPUs. Second, a pure GPU database cannot exploit full inter-device parallelism, because the CPU does not perform any data processing.

Moreover, Both CPU and GPU have distinct advantages tailored to specific applications. As said, GPUs were initially used for graphics rendering, which means their multitude of cores are optimized for executing simple operations concurrently. GPUs leverage SIMD (single instruction, multiple data) to explore massive parallelism. On the other hand, the CPUs, despite having fewer cores, are more complex and are adept at tasks that resist parallelization, such as serial tasks or those with intricate control flow instructions [6]. This distinction is highlighted in sub-figures (a) and (c) of Figure 2.1. As a result, relying solely on a CPU or GPU for processing can lead to considerable performance pitfalls in various contexts.

Based on more recent research [7, 8], GPUs are generally viewed as co-processors to overcome the above drawbacks. The emergence of hybrid CPU-GPU query engines, which distribute

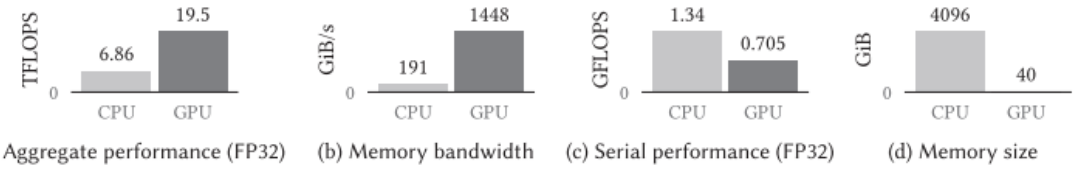


Figure 2.1: Performance comparison between a CPU AMD EPYC 7702P and a GPU NVIDIA ampere A100 [6, p. 3]

certain operations to GPU cores while retaining some on CPU cores, has gained traction lately, driven by the growing need for enhanced computing performance.

However, a challenge with this hybrid strategy is the frequent data transfers from the CPU to the GPU via PCIe for each query. Transferring data over PCIe can be nearly two orders of magnitude less efficient than the GPU memory bandwidth and typically lags behind the CPU memory bandwidth. As a result, the PCIe transfer time becomes the bottleneck and limits performance gains.

2.2 Arrow

Apache Arrow [9] is designed to boost building high performance applications adept at handling and transferring vast datasets. It aims at enhancing the performance of analytical algorithms and the efficiency of moving data between different systems or programming languages.

At the heart of Apache Arrow is its in-memory columnar format – a standardized, language-independent specification for in-memory representing structured, tabular datasets in-memory. This column-centric structure enhances data processing efficiency, as it often circumvents needless data transfers in analytic tasks compared to row-wise format. Morever, it paves the way for vectorized operations that can process multiple values simultaneously. Arrow further boasts the capacity for zero-copy data sharing across diverse programming languages and systems, markedly enhancing data transfer performance.

Several studies have highlighted the benefits of using the Arrow format for data processing and transfer. In the context of data processing, research has been made to use Apache Arrow to expedite in-memory genomics data processing [10]. In terms of data transferring, Arrow Flight—a service that conveys data across networks in the Arrow format—can realize significant enhancements, achieving up to an order-of-magnitude improvement [11] for query frameworks such as Dremio.

Apache Arrow is made for standardization. In the absence of a standard columnar data format, every database and language would need to devise and implement its own unique internal data structure, leading to inefficiencies in data storage. Furthermore, transferring data between different applications with different data formats entails expensive serialization and deserialization operations. Such processes contribute to significant overheads, with serialization and deserialization consuming more than 80% of the time spent in accessing data [11]. Additionally, common algorithms must often be reimplemented for each data format.

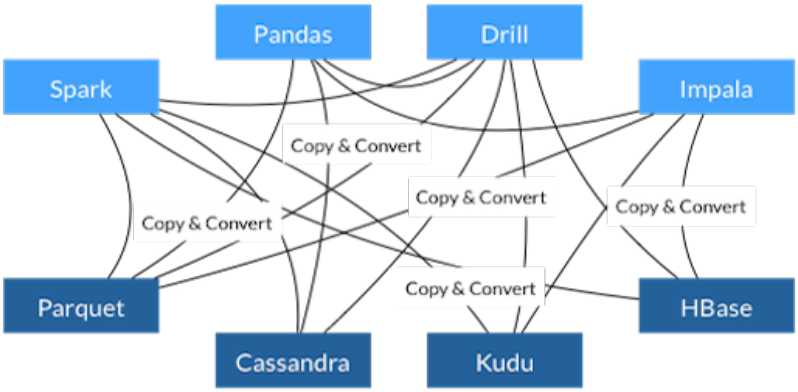


Figure 2.2: High overhead resulting from communicating between big data platforms [9]

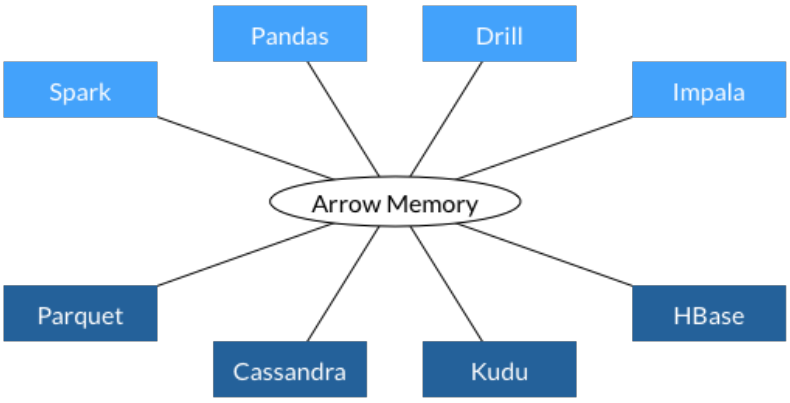


Figure 2.3: In-memory communication using the Apache Arrow standardized data format [9]

Arrow's in-memory columnar data structure provides a ready-made answer to these challenges. Systems incorporating or compatible with Arrow can exchange data with minimal overhead, given that all of them adopt a consistent memory structure. There's no need to craft custom connectors for every distinct system. Beyond these efficiencies, a uniform memory format encourages the consistent use of algorithm libraries, even spanning different programming languages. Figures 2.2 and 2.3 illustrate the efficiency gains when using the Apache Arrow data format to bridge communication among various big data platforms.

2.3 Acero

Acero [12], a subset of the Apache Arrow project, addresses the challenge where repeatedly calling compute functions directly isn't practical due to memory or computational time constraints. Such actions lead to the complete materialization of intermediary data. The Arrow C++ implementation introduces Acero, a streaming query engine, to allow the formulation and execution of computations, even with arbitrarily large inputs, optimizing resource consumption. Acero is adept at OLAP tasks and gains an edge with zero serialization overhead with the Arrow memory format.

Acero is designed to be a query execution engine rather than a fully-featured query engine. This implies it lacks features like a query parser and optimizer, concentrating solely on computations without serving as a complete database server or solution. This targeted emphasis renders Acero versatile, adaptable, and reusable. While it can cater to classical relational algebra, Acero is also equipped to handle advanced functionalities such as window functions and unique domains like time series.

To gauge Acero's performance, we've run a benchmark result of TPC-H Q1 for Acero, a recognized metric for assessing query performance, presented in Table 2.1. The TPC-H Q1 benchmark script can be found within Acero's codebase [1]. For comparison, we've also provided the results for PostgreSQL and SparkSQL [13] at scale factors of 1 and 10. Table 2.1 presents the performance times in milliseconds for the TPC-H Q1 benchmark at scale factors of 1 and 10 for Acero, PostgreSQL, and SparkSQL. PostgreSQL and SparkSQL have significantly longer execution times than Acero at the scale factor of 1 and 10.

Scale Factor	Acero Time (ms)	PostgreSQL Time(ms)	SparkSQL Time(ms)
1	143	24289	3539
10	1395	241404	22487

Table 2.1: Acero TPC-H Q1 benchmark

Acero implements Substrait "consumer" interface, allowing it to accept a standard Substrait plan. As Substrait provides an open standard for execution plans, it simplifies the process of crafting intricate execution plans for Acero. Internally, Acero's execution plan is represented by the ExecPlan, which takes the form of a directed graph composed of operators.

In Acero, the building blocks are the operators, known as ExecNodes. Several common operators are already in place to generate, transform, or consume batches of data (ExecBatch) in Arrow format. Additionally, an interface exists to develop and register customized operators. Each ExecNode works with batches received from upstream nodes (its inputs) and, once processed, forwards these batches to downstream nodes (its outputs) through the graph's edges.

2.4 libcudf

A primary obstacle in data-centric applications is managing vast quantities of data through successive processing stages. This leads to numerous intermediary data modifications that require

[1] https://github.com/apache/arrow/blob/main/cpp/src/arrow/acero/tpch_benchmark.cc

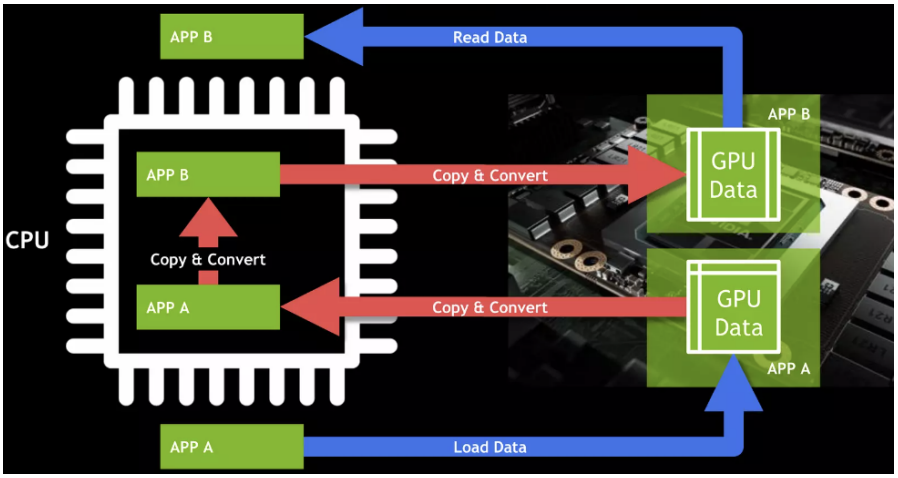


Figure 2.4: Before CuDF [9]

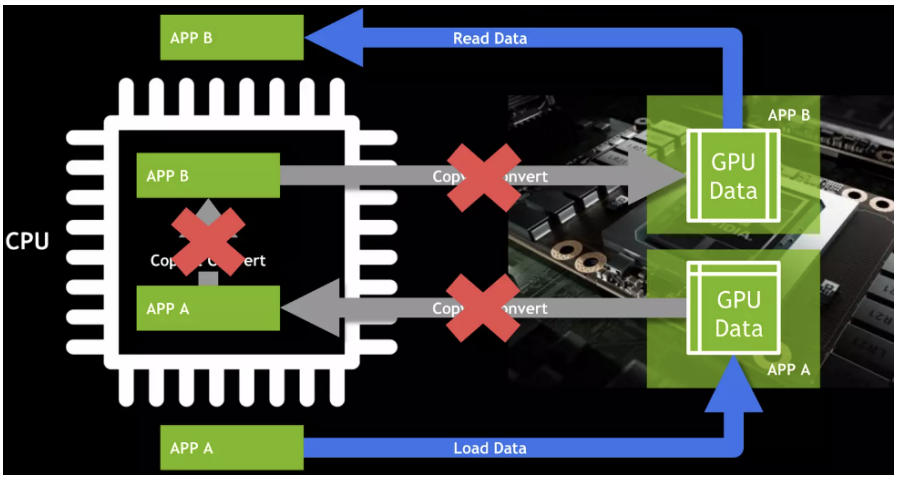


Figure 2.5: After CuDF [9]

further storage and processing. Such intricacies present difficulties when attempting to leverage GPUs in data science applications.

Moreover, Python has established itself as the de-facto programming language for constructing data processing pipelines and implementing algorithms in the data science realm. Yet, there's an absence of a Python API for data manipulation directly on the GPU. This necessitates the use of CUDA C/C++ for GPU acceleration in data science tasks, complicating the deployment of these applications on GPUs.

Rapids is motivated by the above challenges. It is an open-source data science platform that aims to leverage the power of the Arrow format as well the power of GPUs. In order to address the data movement and transformation challenge in data science applications, which limits GPU acceleration potential, data movement overhead can be reduced using the zero-copy standard Arrow in-memory format.

Therefore Rapids comprises multiple libraries for data processing and analysis. One notable inclusion is cuDF, a Python-based data frame library optimized for GPUs and compatible with the Arrow format. As illustrated in Figure 2.4, prior to cuDF's introduction, diverse programming languages and systems, whether based on CPUs or GPUs, necessitated data copying and conversion, leading to burdensome serialization and deserialization overheads. However, with Apache Arrow offering a standardized format, data sharing becomes seamless and is depicted in Figure 2.5. Furthermore, cuDF's Pandas-like API is intuitively designed for data scientists, offering enhanced GPU-driven data processing speeds.

libcudf serves as the foundational C++ library underpinning CuDF. This C++ library facilitates GPU-based data manipulation operations such as loading, joining, aggregating, and filtering. At its core, a GPU DataFrame is columnar data implemented in Arrow memory format. libcudf is built based on CUDA, a general-purpose programming model provided by NVIDIA, employs RMM for memory management. It's strategically designed to leverage the full power of GPU cores, ensuring optimal throughput.

Chapter 3

Core concepts

Chapter 3 further defines and discusses important concepts used in the context of this thesis, i.e. what is a query execution engine, what is the operator model in the query execution engine, and what a scheduler is in a query execution engine.

3.1 Query execution engine

Data management is an advanced field with many interrelated concepts. In this section, we aim to provide a definition of the terms ”database”, ”query engine”, and ”query execution engine”, which will help use these terms consistently throughout the remainder of the text.

Figure 3.1 presents a block diagram of various components included in a database management system. A database management system is a software stack used to store, manage, query, and retrieve data stored in a database, which we will abbreviate as ”database” in the following text. In general, it includes three main components: client communication manager, query engine, and transactional storage manager. In addition, other components are also included such as the process manager and many others. A database efficiently manages and stores an organized collection of data that is stored in a computer system and is used to support various applications and business processes.

A query engine, on the other hand, is a software component that allows users to query in a database and retrieve specific information.

Database users interact with the database through the client communication manager and submit queries. The query engine processes these queries, and the transactional storage manager interacts with the storage to manage transactions.

In simpler terms, a database is like a warehouse where data is stored, while a query engine is like a search engine that allows users to find specific information within that warehouse. The query engine interacts with the data in the database by sending queries to it and receiving responses based on the data stored in the database. In this sense, a query engine can be part of a database, and replaceable by another query engine. Query engines give us the power to exploit massive data and analyze them. The goal of the query engines is to process the input query, executes the query, and give the answer to the query.

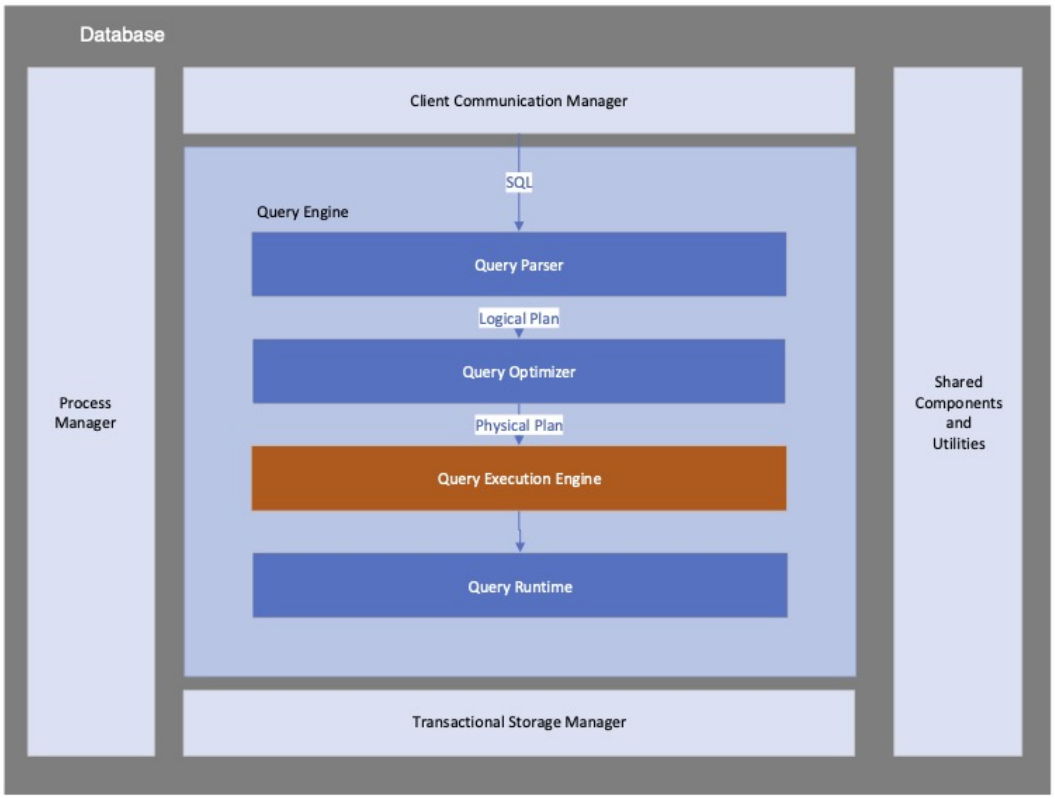


Figure 3.1: Database vs query engine vs query execution engine

Query engines are complex systems themselves that have many components including the parser, the optimizer, the execution engine, and runtime. Depending on the specific implementation of the execution engine, other components can be included as well.

Figure 3.2 shows how the Spark query engine process a query, which consists of two stages: front end and back end. In the front-end stage, the query parser parses the SQL query and only checks for SQL syntax errors. In case of a syntax error, an exception will be raised, otherwise the parser generates an unresolved plan that represents the query. The analyzer then analyzes the plan against the data sources to resolve issues such as column name, table name, etc, and if resolved correctly, it generates a logical plan. Then, the query optimizer applies a number of optimization rules (such as filter combination and push down) to reduce the complexity of the plan. This results in an optimized logical plan.

And in the back-end stage, the planner does the physical planning, selects an optimal physical plan according to a cost model. Finally, the query execution engine executes the physical plan.

The query execution engine is a component of the overall query engine. It processes a physical query plan, represented as a directed acyclic graph comprising execution operators. Each

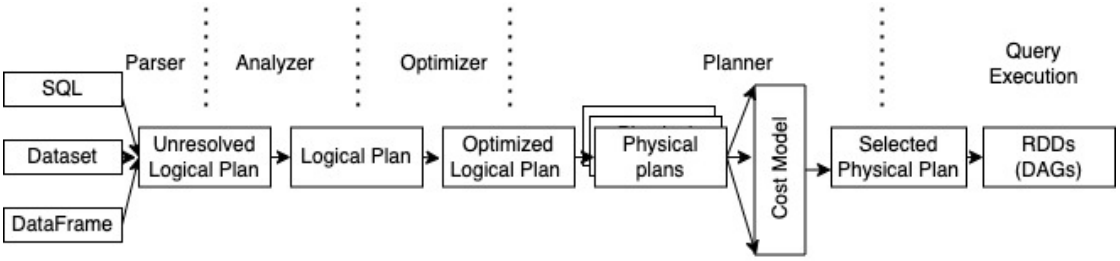


Figure 3.2: Phases of query planning in Spark SQL [14, p. 6]

operator, whether it's a 'group by', 'filter', 'join', or another type, either produces, transforms, or consumes the data it encounters. As data traverses the edges of this graph, it is acted upon by these operators.

Acero [12] is such a query execution engine. Table 3.1 includes a comparison between Acero and other data management and processing systems to show common and unique aspects of Acero.

Acero and Velox are query execution engines, which means they focus only on the compute, receiving a physical plan as input, and executing it. Both do not have a query optimizer but expect to receive an optimized plan as input. These specific two engines are both arrow-native, however, Velox is owned by Facebook and implemented its own Arrow utilities, Acero, a sub-project of Apache Arrow, is integrated within libarrow . Even though any Apache Arrow implementation can work with the C data interface, opting for non-default implementations might introduce performance overheads. Therefore, relying solely on the default libarrow implementation might lead to more compact binaries and potentially superior performance.

SparkSQL is a query engine capable of processing SQL queries and possesses its own query optimizer. Unlike traditional RDBMS, Relational Database Management System, SparkSQL does not provide support for ACID (Atomicity, Consistency, Isolation, Durability) properties, which guarantee data integrity despite errors. DataFusion is another query engine that supports computing on Apache Arrow in-memory data format.

PostgreSQL and DuckDB are both examples of RDBMS that provide support for the ACID properties of database transactions. However, DuckDB also includes support for the Apache Arrow in-memory data format, allowing for efficient data exchange with other systems. On the other hand, PostgreSQL inherently employs a row-wise storage approach, wherein data is stored sequentially by rows, as opposed to columnar storage systems that organize data by columns.

3.2 Operators

Operators in a query execution engine are the building blocks of query processing. They are the basic computational units that perform specific operations on data, such as filtering, sorting, aggregation, joins, and projections.

Table 3.2 lists 9 of the most common types of operators implemented in a query execution engine. Each engine implements these operators internally using specific functions. In Acero,

Table 3.1: Comparison of data management and processing systems

Data system	Definition	Input	Arrow	Columnar	Query Optimizer	Distributed	Streaming	Implementation language	ACID
Acero	Query execution engine	Physical Plan	Yes	Yes	No	No	Yes	C++	No
Velox	Query execution engine	Physical Plan	Yes	Yes	No	No	Yes	C++	No
SparkSQL	Query engine	SQL	No	Yes	Yes	Yes	Yes	Scala	No
DataFusion	Query Engine	SQL	Yes	Yes	Yes	No	Yes	Rust	No
PostgreSQL	Object-oriented RDBMS	SQL	No	No by default	Yes	No itself	No	C	Yes
DuckDB	Table-oriented RDBMS	SQL	Yes	Yes	Yes	No	Yes	C++	Yes

each of these functions is called an execution node (or ExecNode). The table also lists the ExecNode function name corresponding to the listed operator.

Table 3.2: Common operator types and corresponding Acero ExecNodes

Operator type	**Description**	**Acero ExecNode**
Source	A starting point for constructing a streaming execution plan, involving sourcing data from a file, iterating through an in-memory structure, or acquiring data via a network link	source, table_source, scan
Filter	Provides an option to define data filtering criteria, selecting rows for which the specified expression holds true.	filter
Projection	Merges multiple tables or data streams using a shared attribute.	project
Aggregation	Calculates aggregate metrics like count, sum, average, max, or min for a set of rows.	aggregate
Sorting	Orders the rows in a dataset or data stream according to specified attributes.	order_by_sink
Union	Combines two or more tables or data streams into a single table or data stream;	union
Windowing	Splits data into windows based on time or specific criteria and performs operations on each window.	in progress
Join	The join operation merges rows from two or more tables by aligning them on a shared field or group of fields, and it's a standard procedure in relational databases and various data processing platforms.	hash_join
Sink	The execution plan's concluding step yields the query's ultimate result set. The sink operation takes in the output from prior steps in the plan and delivers the final outcome to the user.	sink, consuming_sink, order_by_sink, select_k_sink, table_sink

Operators are organized in a physical plan, which represents the sequence and interdependence of the operators essential for executing a specific query. Query execution engines interpret the physical query plan as a directed acyclic graph, with each node representing an operator. Every query operator may possess zero, one, or several input and output operators. These operators are classified into three primary categories based on their usage, as referenced in [15].

- **Producer operators** like the scan operator to generate a dataset from a source. They do not have input operators, they stand as the primary task initiators in a plan. Their function is to organize data retrieval from sources like in-memory tables, network sockets, or disks. They segment the data into batches, allocate a task for each batch, and forward the data to their subsequent output nodes.

- **Transformer operators** Such as the filter, project, order_by_sink, hash_join operators,

these transform data from one set to another. They process data in batches, receiving each batch from their preceding input operators.

- **Consumer operators** like the sink or sum aggregate operator to apply the aggregate operation to the collection and produce a result.

Apart from being categorized by their usage, the operators can also be classified into two types

- **Computing-intensive operators** involve extensive processing even if the data set is relatively small, such as the hash_join operator.
- **Data-intensive operators** operate on a large data set, like the aggregate operator.

The selection and sequencing of operators play a pivotal role in influencing the performance of the query execution engine. Consequently, optimizing the query execution plan is an important task for query optimization.

Additionally, in a hybrid CPU-GPU environment, the distribution decision for operators is vital for the query execution engine. As highlighted in the previous paper [16], GPUs are better suited for compute-intensive tasks over data-intensive ones. Hence, redirecting operators that demand extensive computation on limited data sets to the GPU is a key strategy.

3.3 Scheduler

GPU can not accelerate all operators at all scenarios, not just because some operators have more logic that make them hard to be paralleled. But also because of additional materialization or data transfer overhead between CPU and GPU, even for the same operators, the data size and the current location of data will also affect the effect of GPU acceleration.

In a CPU-GPU hybrid query execution engine, the scheduler is responsible for managing the execution of queries on both CPU and GPU resources in a coordinated and efficient manner.

The purpose of the scheduler in a CPU-GPU hybrid environment is to distribute the operators to the appropriate resources (i.e., CPU or GPU) to optimize performance and throughput. For example, certain operators may be more efficiently executed on a CPU, while others may benefit from GPU acceleration.

In a CPU-GPU hybrid query execution engine, the scheduler needs to consider various aspects when assigning resources and orchestrating query execution. This includes the current availability of CPU and GPU resources, the nature of the query and its demands, as well as the system's setup.

The scheduler, through smart distribution of operators between the CPU and GPU, ensures optimal query execution with minimal resource contention and bottlenecks. The objective is to maximize query performance and throughput by harmoniously utilizing the advantages of both CPU and GPU.

One way of categorizing the scheduler is by the scheduling time [6]. The scheduling time indicates when a task is allocated to a specific processor by the system. This can either be

static, predetermined prior to program execution, or dynamic, adjusted as the program runs. The processor usage partially determines the scheduling time. In specialized systems, scheduling is invariably static, defined by the system's design. On the other hand, generic systems might adopt either a static or dynamic approach. There's also the possibility of a hybrid scheduling system that initially makes static decisions regarding CPU and GPU resource allocation during query optimization but adjusts them dynamically during the query's execution, influenced by factors like data locality and processor workload. The underlying rationale for these scheduling decisions stems from the system's scheduling strategy, which we delve into subsequently.

Another way of categorizing the scheduler is by the scheduling strategy [6]. The scheduling strategy outlines the criteria a query processing system uses when assigning a task to a particular processor. Again, processor usage plays a role in shaping this strategy. While specialized systems base their scheduling decisions predominantly on the task's characteristics, generic systems consider these traits too. However, generic systems also incorporate additional metrics, such as anticipated task costs derived from a cost model, data locality, and the current load on the processor.

Chapter 4

Methodology

Chapter 4 contains the benchmark result of the CPU-GPU hybrid accelerated query execution engine. This micro-benchmark result serves as the primary basis for the design of the scheduler in the next chapter. Acero is a query execution engine running on CPUs, focusing on computing the data with the Apache Arrow memory format. It takes a physical plan and executes each operator described in Section 3.2.We intend to utilize GPU acceleration to enhance the performance of Acero. As mentioned in the background section, libcudf is a C++ library that underlies CuDF, and it furnishes a C++ GPU DataFrame library capable of loading, joining, aggregating, filtering, and manipulating data, built on top of the Arrow memory format.

The operators Acero uses are the same as the compute kernels of the compute functions of Apache Arrow. We can find computational units in libcudf that correspond to these compute functions. Moreover, libcudf does some performance optimizations for these operators on GPUs. One large improvement is with regard to the CUDA memory allocation. Since many query execution operators require temporary allocations and most libcudf operators are not performed in place, therefore, there are many column allocations and deallocations. However, cudaMalloc and cudaFree are expensive operations as they are synchronous processes that will block the device. libcudf optimizes this by developing RMM (RAPIDS Memory Manager). RMM reduces the frequent temporary memory allocation overhead by using large cudaMalloc allocation as memory pools and using streams to enable asynchronous malloc/free.

Therefore, in order to accelerate Acero with GPU, we can leverage the computational units in libcudf to replace and accelerate the compute functions in Apache Arrow. We first benchmark the performance of these operators on CPU and GPU separately in order to understand the characteristics of these operators and the maximum potential acceleration rate that can be achieved. Furthermore, we developed four micro-benchmarks for each operator to take the data transfer time into account. Later in the next chapter, we will use this result to design and implement an on-the-fly scheduler to decide, given the current location, data input, and operator type, whether it is better to distribute the operator to the CPU or the GPU.

4.1 System setup

This section provides an overview of the system environment used for the study. This information is crucial to better understanding the context in which the research was conducted. The system setup used for the experiments is summarized in Table 4.1, which provides an overview of the hardware and software components. It includes the operating system, kernel version, CPU, GPU, memory, and versions of key software components.

As listed in the table, the operating system used for the study is Ubuntu 20.04.6 LTS, with kernel version 5.11.0-46-generic. The CPU used is an Intel Xeon Gold 6342 with 11 cores, while the GPU is an NVIDIA A10. The NVIDIA driver version is 515.43.04. The system has a total of 86GB of memory available.

The table also lists the version numbers for key software components used in the study. These include CuDF version 22.08.00, Arrow version 8.0.0, CUDA version 11.7, and GCC version 9.4.0.

Table 4.1: System setup information

Component	**Version/Details**
Operating System	Ubuntu 20.04.6 LTS
Kernel Version	5.11.0-46-generic
CPU	Intel Xeon Gold 6342 @ 2.80GHz (11 cores)
GPU	NVIDIA A10
GPU Memory	24GB GDDR6
GPU Memory Bandwidth	600 GB/s
CUDA Cores	72
PCIe Gen4 Memory Bandwidth	64 GB/s
NVIDIA Driver Version	515.43.04
Memory	86GB
CUDF Version	22.08.00
Arrow Version	8.0.0
CUDA Version	11.7
GCC Version	9.4.0

4.2 Benchmarks

4.2.1 Local performance of Acero and libcudf

Firstly, we assess the performance of the corresponding compute kernels separately in Acero and libcudf while only including the execution time (i.e., without data transfer time). We implemented the experiments in C++ by generating a random decimal array with elements ranging from 0 to 9 given an array length and applying the same operator type of Acero and libcudf to the same input array.

The Copy Operator copies the elements from an array in one memory location to another array in a different memory location. The Filter Operator, as the name suggests, provides an option to define data filtering criteria based on a given expression. It selects rows where the given expression evaluates to true. For our experiment, we select the numbers that are smaller than 3, therefore the result data set will be only 30% of the input array. The Sort Operator sorts the array in ascending order.

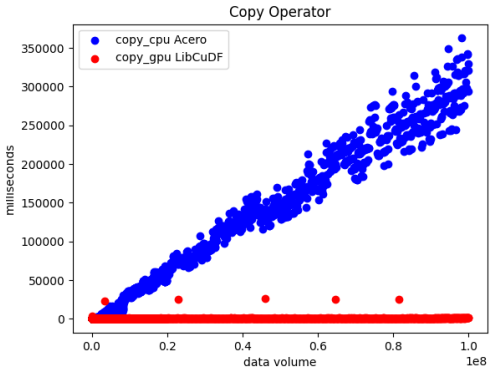
Figure 4.1a and Figure 4.1b are the performance measurements of the Copy Operator. Figure 4.1b is a zoom-in version of Figure 4.1a to emphasize the break-even point. The x-axis in the figure is the array length, which is up to 1e8. We refer to the array length as the data volume in the following chapters. The y-axis is the time cost in milliseconds.

It can be observed that, when the data volume is small (below 1e5 elements), the CPU Copy Operator (i.e., the Acero Copy Operator) costs less time than the GPU Copy Operator (i.e., the libcudf Copy Operator). This is because the GPU has a large overhead as it needs to initiate the kernel. However, as the data volume increases, when the length of the array is around 1e5, the GPU Copy Operator outperforms the CPU Copy Operator.

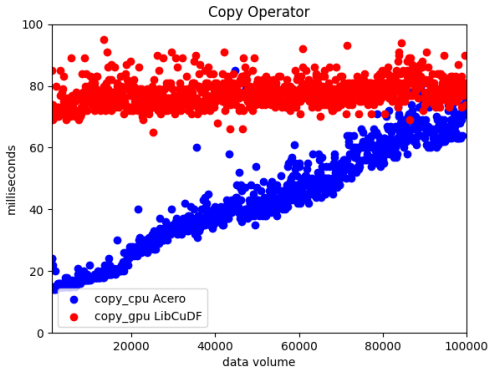
As can be seen in Figures 4.1c and 4.1d, and Figures 4.1e and 4.1f, the same pattern also applies to the Filter Operator and Sort Operator, respectively. However, the break-even point of the Filter and Sort Operators comes much earlier than the Copy Operator at 3e4 and 5e4, respectively, which means the Filter Operator and Sort Operator on the GPU will outperform the CPU at a relatively smaller data volume compared to the Copy Operator.

In all figures, the CPU execution time increases much faster than the GPU, although GPU time does increase slightly in all cases. This results in a rather large acceleration potential for the GPU on these operators.

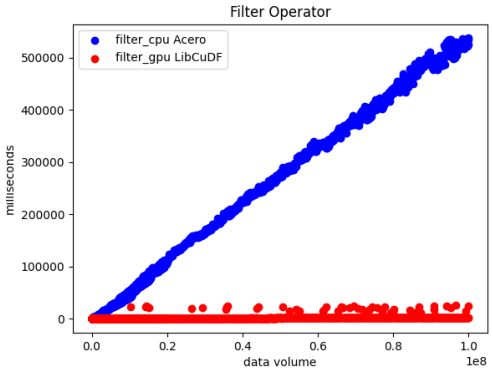
Figure 4.2a and Figure 4.2b provide a visual representation of the acceleration rate that we can achieve by using a GPU instead of the CPU. This acceleration rate can be defined as follows: $\frac{Time\ cost\ of\ the\ CPU\ Operator}{Time\ cost\ of\ the\ GPU\ Operator}$ for a given data volume. The figures show that as the data volume goes up, the acceleration rate of GPU over CPU becomes increasingly prominent since the GPU initialization overhead becomes marginal compared to the total execution time. The figures also show that this acceleration rate converges to a specific value that represents the acceleration potential of the GPU over CPU. When the data volume reaches an amount of around 1e7, the GPU is able to achieve acceleration of more than two orders of magnitude on the Copy Operator and Filter Operator. The acceleration curves continue to increase beyond a data volume of 1e8 which is the memory capacity limit of the GPU. This indicates that for these operators, memory bottlenecks are the limiting factors for the acceleration potential. In contrast, for the Sort Operator, the figure shows that the acceleration converges towards a value of around 12x.



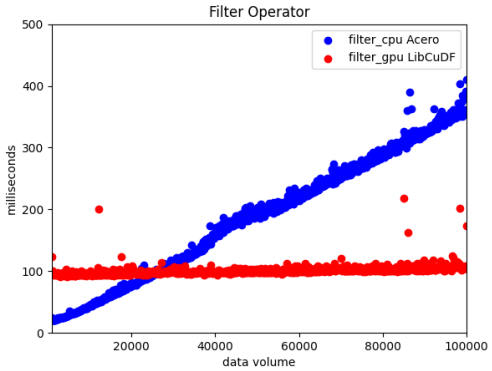
(a) Copy Operator



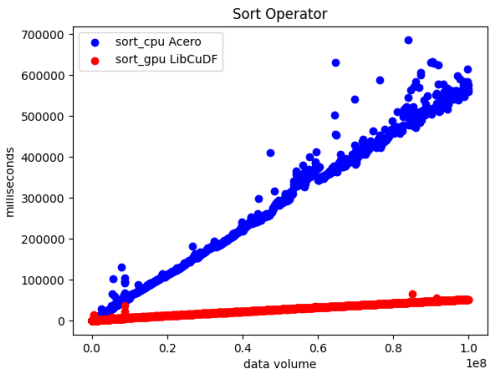
(b) Copy Operator - zoom in



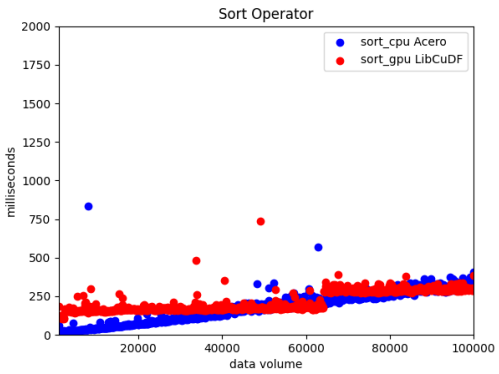
(c) Filter Operator



(d) Filter Operator - zoom in

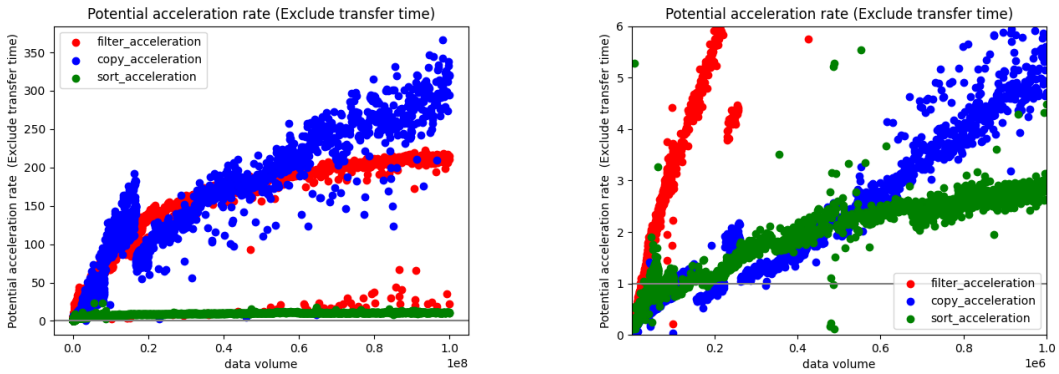


(e) Sort Operator



(f) Sort Operator - zoom in

Figure 4.1: Local operators performance on CPU and GPU



(a) Acceleration rate

(b) Acceleration rate - zoom in

Figure 4.2: Acceleration rate excluding data transfer time

This indicates the relative difficulty of implementing Sort Operators on the GPU due to the difficulty of algorithm parallelization as a result of the use of many control instructions in the algorithm. This means that compute bottlenecks are the limiting factors for the acceleration potential for this operator.

The acceleration figures indicate the large acceleration potential of GPUs for common database operators. However, these numbers are measured under ideal circumstances without data copy times, which has to be further investigated to identify a more realistic acceleration potential.

4.2.2 Acceleration with data transfer

In Subsection 4.2.1, we measured the operator performance separately with Acero on CPU and with libcudf on GPU. The acceleration rates measured in Figure 4.2a and Figure 4.2b show how promising Acero can be accelerated with the GPU.

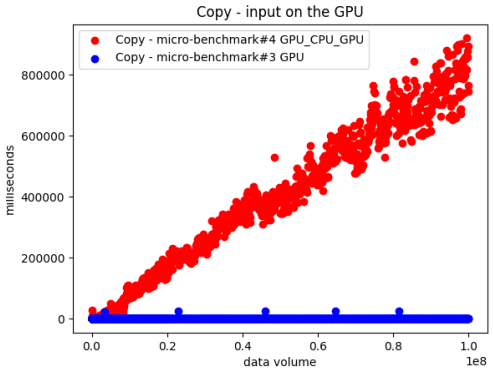
As mentioned before, to achieve this acceleration, we need to transfer the data to another location and then transfer the result back. However, the bandwidth of the PCI-e bus is relatively slow and will limit the potential for acceleration. In order to quantify this, for each operator, we design and implement four micro-benchmarks:

1. micro-benchmark#1 having the data input on CPU and executing the operator on CPU with Acero

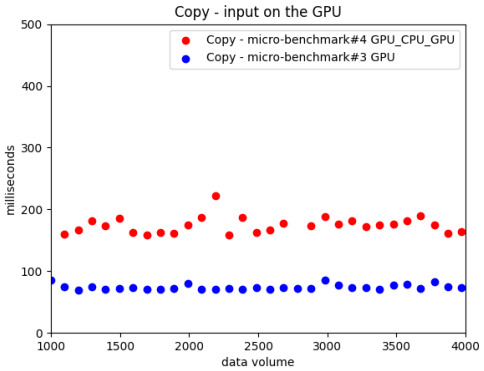
2. micro-benchmark#2 having the data input on CPU, transferring the input to GPU, executing the operator on GPU with libcudf, and transferring the result back to CPU

3. micro-benchmark#3 having the data input on GPU and executing the operator on GPU with libcudf

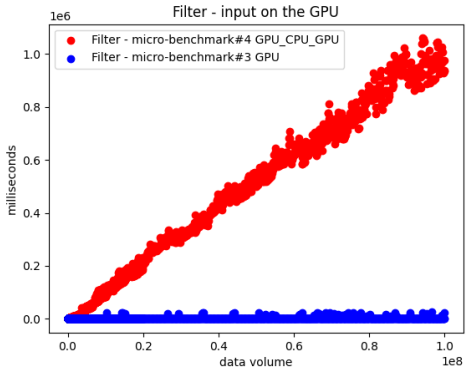
4. micro-benchmark#4 having the data input on GPU, transferring the input to CPU, executing the operator on CPU with Acero, and transferring the result back to GPU



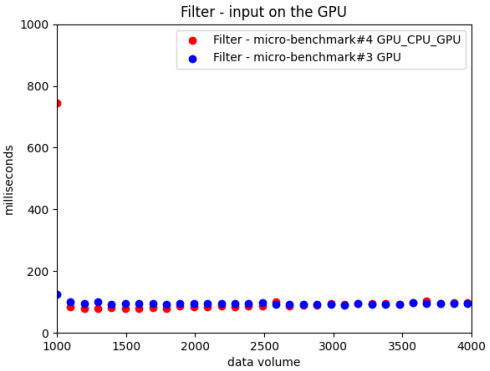
(a) Copy Operator from GPU



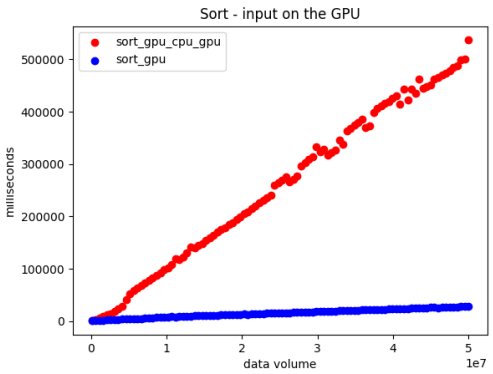
(b) Copy Operator from GPU - zoom in



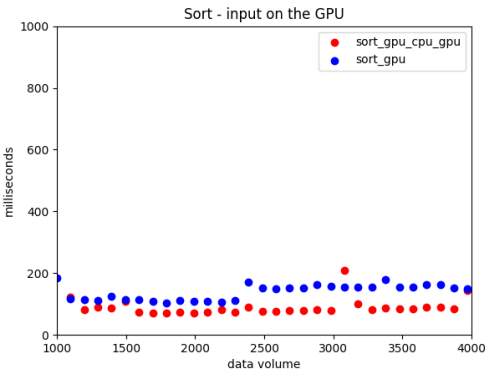
(c) Filter Operator from GPU



(d) Filter Operator from GPU - zoom in

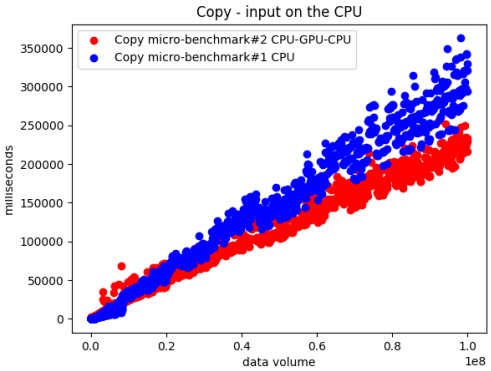


(e) Sort Operator from GPU

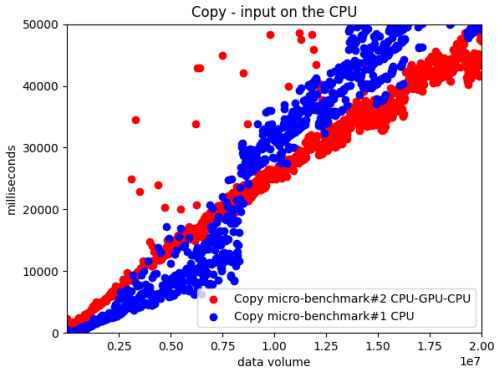


(f) Sort Operator from GPU - zoom in

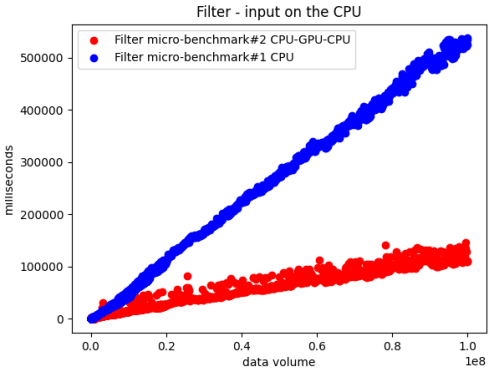
Figure 4.3: Micro-benchmarks with data transfer from the GPU



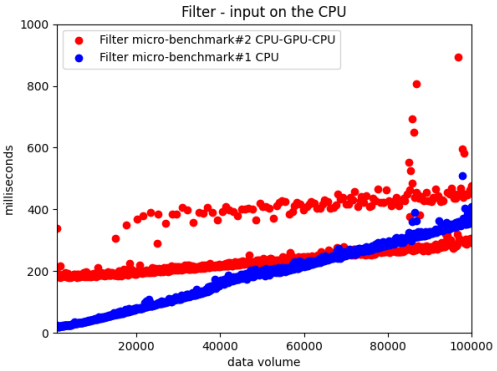
(a) Copy Operator from CPU



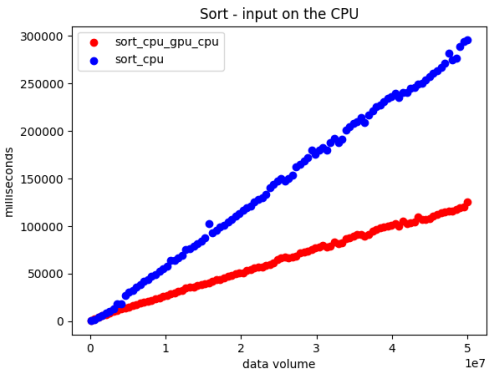
(b) Copy Operator from CPU- zoom in



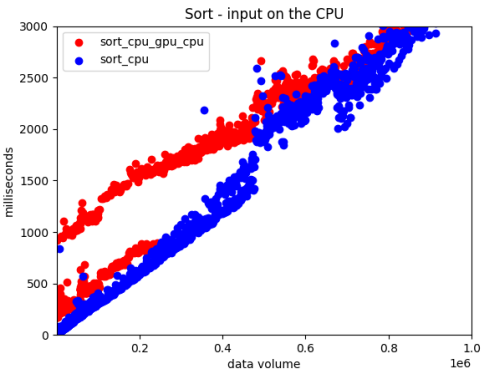
(c) Filter Operator from CPU



(d) Filter Operator from CPU - zoom in



(e) Sort Operator from CPU



(f) Sort Operator from CPU - zoom in

Figure 4.4: Micro-benchmarks with data transfer from the CPU

In Figure 4.3, we measure micro-benchmark#3 (red line) and micro-benchmark#4 (blue line). These benchmarks show the cases where the data is originally located on the GPU. For all three operators we measured, micro-benchmark#3 costs less time than micro-benchmark#4 (starting at a similar time for small data volumes), with the difference becoming only larger with the increase of data volumes. This can be explained by the fact that taking data copy time into consideration in combination with a slower CPU execution leads to lower total execution times. Based on these results, we can conclude that when the input data is located on GPU, it almost always costs less time to execute the operator on GPU for these three operators except for very small data volumes. This suggests the data transfer overhead is large enough that if we already have our input data located on the GPU in one node of the query execution plan, we should have as many consecutive operators executed on the GPU as possible. As this is the query-level optimization that should be improved in the query optimization phase, it is out of the scope of our operator-level optimization in the execution phase.

In Figure 4.4a, we show micro-benchmark#1 (blue line) and micro-benchmark#2 (red line) for the Copy Operator. In Figure 4.4a, we can observe that though micro-benchmark#2 eventually shows an advantage over micro-benchmark#1 for the Copy Operator, it is not as apparent as in Figure 4.1a which does not include the data transfer time over the PCI-e bus.

Figure 4.4b is a zoom-in version of Figure 4.4a which indicates that for the Copy Operator, at around 1e7, micro-benchmark#2 starts to over-perform micro-benchmark#1. That means with the same amount of data volume, if the input data is originally located on the CPU when the length of the data volume is over around 1e7, it is better to distribute the Operator to GPU in order to have less overall time cost, even though that will include the overhead of transferring the input data to the GPU and then transferring the result data back.

The break-even point where the GPU execution starts to over-perform CPU execution at 1e7 comes much later than 1e5 which is the break-even point we measured in Figure 4.1b where we do not consider data transfer back and forth and only consider the local performance.

For the Filter Operator and Sort Operator, we can also observe that the data transfer time reduces the potential acceleration achieved on the GPU due to data transfer delays, though it is not that obvious as it is for the Copy Operator. Even though it includes the transfer time over PCI-e bus, micro-benchmark#2 still has a significant advantage over micro-benchmark#1 as shown in Figure 4.4c and Figure 4.4e. The break-even point at 7e4 in Figure 4.4d and 5e5 in Figure 4.4f is not that much later than 3e4 in Figure 4.1d and 5e4 in Figure 4.1f. Compared to the Copy Operator, this break-even point takes place two orders of magnitude later in time.

The results in Figure 4.3 show that as long as we have the data located on the GPU, then it is always better to execute the operators on the GPU itself, which will result in increasing the acceleration rate. On the other hand, Figure 4.4 shows that if the data is located on the CPU, then we need to have a more detailed acceleration model to evaluate the acceleration potential of the GPU.

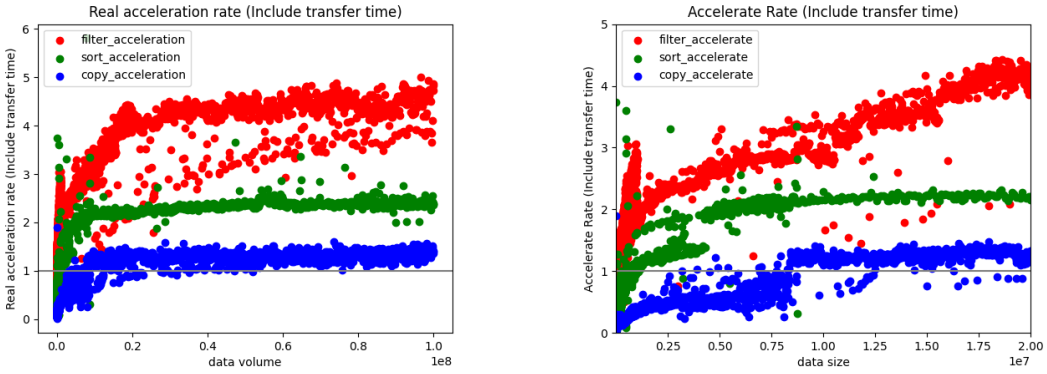
4.3 Acceleration models

In order to include data transfer times (from CPU to GPU and back) into consideration in the calculation of the potential acceleration, we define the real acceleration rate as:

$$\text{Real acceleration rate} \stackrel{\text{def}}{=} \frac{t_{cpu}}{t1_i + t2_{gpu} + t3_o} \tag{4.1}$$

where

- t_{cpu}: the operator execution time on CPU.
- $t1_i$: the transfer time of input data from CPU to GPU over the PCI-e bus.
- $t2_{gpu}$: the operator execution time on GPU.
- $t3_o$: the transfer time of output data from GPU to CPU over the PCI-e bus.



(a) Real acceleration rate

(b) Real acceleration rate - zoom in

Figure 4.5: Real acceleration rate including data transfer time

Figure 4.5 visualizes the real acceleration rate, i.e. the acceleration rate including the data transfer time of the Copy Operator, Filter Operator, and Sort Operator. From the information conveyed in Figure 4.5, we can see that all three operators potential acceleration rates are limited largely by the PCI-e bus transfer time, which are reduced from over two orders of magnitude to an acceleration range between 1x and 5x. All tested operators and their real acceleration rates exhibit a pattern of ramping up from zero and converging to a constant value as the amount of data increases.

We introduce a model to try to reason about the pattern of the acceleration rate and performance bounds for our hybrid CPU/GPU execution engine shown in Figure 4.5a as well as why different operators converge to different acceleration rates. The model is defined as follows.

$$t_{cpu} = L_{sys} * (N * F_{CPU} * G_{CPU}) \tag{4.2}$$

where

- L_{sys}: the system load.
- N: data volume, i.e., the length of the input data array.
- F_{CPU}: the CPU cycles needed for one unit of data volume.
- P_{CPU}: how much time is needed for one CPU cycle, i.e $\frac{1}{\text{CPU frequency} * \text{Number of CPU Cores}}$

$$t1_i = L_{sys} * N * P_{PCI_e} \tag{4.3}$$

where

- L_{sys}: the system load.
- N: data volume, i.e., the length of the input data array.
- P_{PCI_e}: how much time is needed to transfer one unit of data volume, i.e $\frac{1}{\text{PCI-e bandwidth}}$

$$t2_{gpu} = L_{sys} * (T_{initGPU} + N * F_{GPU} * P_{GPU}) \tag{4.4}$$

where

- L_{sys}: the system load.
- $T_{initGPU}$: the time cost to initialize the GPU kernel.
- N: data volume, i.e., the length of the input data array.
- F_{GPU}: The GPU cycles needed for one unit of data volume.
- P_{GPU}: how much time is needed for one GPU cycle, i.e $\frac{1}{\text{GPU frequency} * \text{Number of GPU Cores}}$

$$t3_o = L_{sys} * \alpha * N * P_{PCI_e} \tag{4.5}$$

where

- L_{sys}: the system load.
- α: the dynamic data-related parameters, such as cardinality and selectivity, which affects the result data set. previous studies [17, 18] have provided many ways to estimate this runtime operator-specific dynamic multidimensional parameters. According to that, we assume that these parameters can be obtained as input.
- N: data volume, i.e., the length of the input data array.
- P_{PCI_e}: how much time is needed to transfer one unit of data volume, i.e $\frac{1}{\text{PCI-e bandwidth}}$

$$\begin{aligned}\text{Real acceleration rate} &\overset{\text{def}}{=} \frac{t_{cpu}}{t1_i + t2_{gpu} + t3_o} \\ &= \frac{L_{sys} * (N * F_{CPU} * G_{CPU})}{L_{sys} * (N * P_{PCI_e} + T_{initGPU} + N * F_{GPU} * G_{GPU} + \alpha * N * P_{PCI_e})} \\ &\overset{\text{simplified}}{=} \frac{F_{CPU} * P_{CPU}}{\frac{T_{initGPU}}{N} + F_{GPU} * P_{GPU} + (1+\alpha) * P_{PCI_e}}\end{aligned} \tag{4.6}$$

We assume the system load L_{sys}, though dynamic, equally affects all stages of the process, therefore is simplified in Equation 4.6.

When $N \to 0$, $\frac{T_{initGPU}}{N} \to +\infty$, the denominator of Equation 4.6$\to +\infty$, therefore the real acceleration rate $\to 0$. As N increases, $\frac{T_{initGPU}}{N}$ decreases, the real acceleration rate ramps up; When $N \to +\infty$, $\frac{T_{initGPU}}{N} \to 0$,

$$\text{Real acceleration rate} \xrightarrow{\text{converges to}} C_{real} = \frac{F_{CPU} * P_{CPU}}{F_{GPU} * P_{GPU} + (1+\alpha) * P_{PCI_e}} \tag{4.7}$$

In this equation, we can identify two types of parameters, operator-specific, and hardware-specific.

- operator-specific constants: F_{CPU}, F_{GPU}, α
- hardware-specific constants: P_{CPU}, P_{GPU}, P_{PCI_e}

Therefore, the common pattern shown in Figure 4.5a, that the real acceleration rate ramps up from zero and converges to a constant as the data volume increase can be explained by this model.

In order to model the ideal potential acceleration rate for the different operators excluding data transfer time, as shown in Figure 4.2, we define a similar model as follows

$$\begin{aligned}\text{Potential acceleration rate} &\overset{\text{def}}{=} \frac{t_{cpu}}{t_{gpu}} \\ &= \frac{L_{sys} * (N * F_{CPU} * G_{CPU})}{L_{sys} * (T_{initGPU} + N * F_{GPU} * G_{GPU})} \\ &\overset{\text{simplified}}{=} \frac{F_{CPU} * P_{CPU}}{\frac{T_{initGPU}}{N} + F_{GPU} * P_{GPU}}\end{aligned} \tag{4.8}$$

The potential acceleration rate will converge to

$$\text{Potential acceleration rate} \xrightarrow{\text{converges to}} C_{potential} = \frac{F_{CPU} * P_{CPU}}{F_{GPU} * P_{GPU}} \tag{4.9}$$

Though in Figure 4.2, with our maximum experiment data volume, the potential acceleration has not quite converged yet due to the memory limit of the GPU, can show a trend that $C^{Sort}_{potential} <$

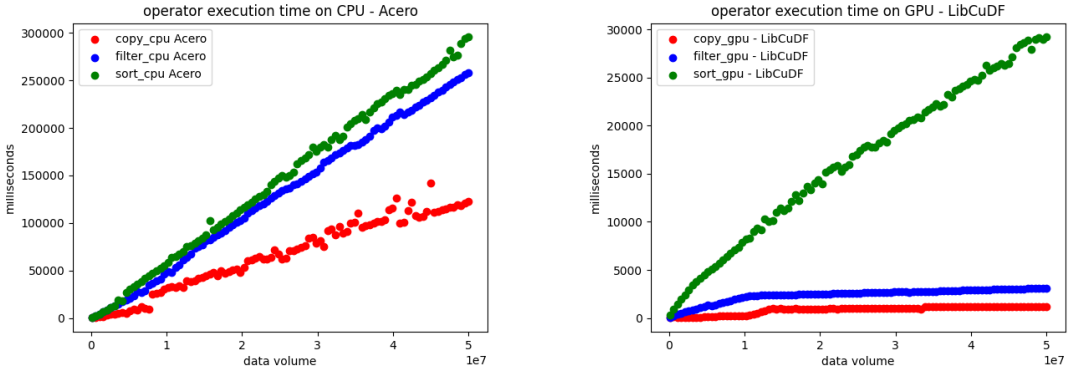
$C^{Filter}_{potential} < C^{Copy}_{potential}$. Since Equation **??** contains operator-specific constants, it also explains why different operators converge to different constants.

Intuitively, we know the Sort Operator has the highest number of instructions and branches. Acero implements the Sort Operator as an O(n) counting sort when the data range is small and as an O(n logn) std::stable_sort comparison-based sort for a bigger data range. The second is the Filter Operator, and the last is the Copy Operator which is an all-memory operation. This can be verified in Figure 4.6a and Figure 4.6b that both on CPUs and GPUs, Sort Operator takes the most execution time, the second is Filter Operator, and the last is Copy Operator.

The GPU is a specialized processor and is built with a different philosophy from the CPU. CPUs are more latency-focused while GPUs are throughput-focus. To achieve high throughput, GPUs have many more cores that they are able to implement by reducing the size of each core, making them much simpler. At the same time, in order to reduce the power consumption of all these cores, they run slower than each core on the CPU so that it consumes less power. In contrast, each core of the CPU is more complicated with a longer pipeline, more registers, larger caches as well as branch prediction capabilities, so that it can run complex code efficiently. Therefore, GPUs can accelerate parallelizable tasks better, while CPUs are better for those tasks that are hard to parallelize or involve a lot of control flow instructions and branches in the algorithm. That is the reason why $\frac{F^{Sort}_{CPU}}{F^{Sort}_{GPU}} > \frac{F^{Filter}_{CPU}}{F^{Filter}_{GPU}} > \frac{F^{Copy}_{CPU}}{F^{Copy}_{GPU}}$, and because P_{CPU} and P_{GPU} are hardware-specific constants, it explains $C^{Sort}_{potential} < C^{Filter}_{potential} < C^{Copy}_{potential}$.

When considering the data transfer time, the ideal potential acceleration rates ($C_{potential}$) are scaled down by different degrees with a ratio (D) to reach the real acceleration rates (C_{real}).

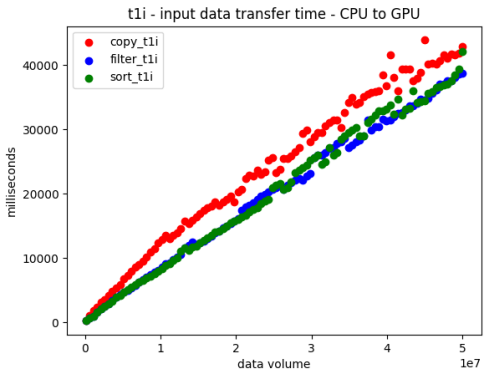
$$\begin{aligned} \text{Downgrade ratio D} &\overset{\text{def}}{=} \frac{C_{potential}}{C_{real}} \\ &\overset{\text{simplified}}{=} 1 + \frac{(1+\alpha) * P_{PCI_e}}{F_{GPU} * P_{GPU}} \end{aligned} \tag{4.10}$$



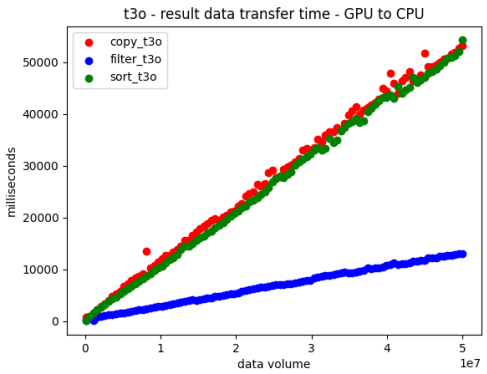
(a) CPU/Acero execution time comparison

(b) GPU/libcudf execution time comparison

Figure 4.6: Operator execution time comparison



(a) t1o - Input data transfer time comparison - CPU to GPU



(b) t3o - Result data transfer time comparison - GPU to CPU

Figure 4.7: Data transfer time comparison

From Equation 4.10, we can note that there are two parts that affect D. One is α, the dynamic data-related parameters that affect the data volume of the result data set. In our example, α of Copy and Sort Operator is 1, but α of Filter Operator is 0.3 because the selectivity of Filter Operator is 0.3, which is shown in Figure 4.7b. So the acceleration rate of the Filter Operator is downgraded less fast than other operators.

The other part is $\frac{P_{PCI_e}}{F_{GPU} * P_{GPU}}$, which indicates the proportion of data transferring time compared with the GPU operator execution time. Because GPU Sort Operator execution time is one order of magnitude larger than GPU Copy and GPU Filter Operator, its real acceleration rate decreases relatively less than the two other operators. These two factors together result in $C_{real}^{Copy} < C_{real}^{Sort} < C_{real}^{Filter}$.

4.4 Other operators

Some operators Acero uses are the same as the compute kernels of the compute functions of Apache Arrow. Therefore in Chapter 4, we found computational units in libcudf that correspond to these compute functions and performed the micro-benchmarks.

However, it is worth noting that some Acero operators can solely be executed within a query plan. Given our objective to evaluate the GPU acceleration potential of the Arrow Acero query execution engine at the operator level, we encounter the challenge of tightly integrating libcudf with Acero in order to accelerate these specific Acero operators. This integration introduces a higher level of implementation complexity. In the meantime, we also intend to gather measurements that indicate the time required for these libcudf operators. As a result, we performed and included these measurements of Left Join Operator and Group Aggregate Sum Operator separately in this section.

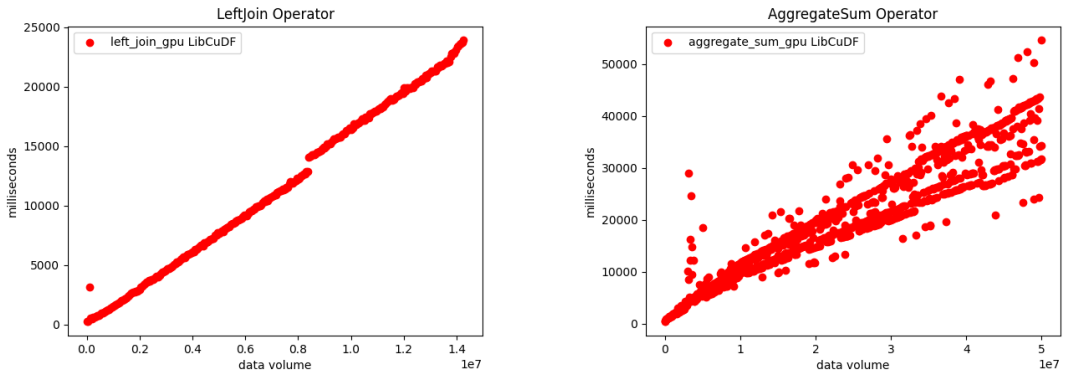
Besides, we add the measurement of Scalar Aggregate Sum Operator in this section. This is because it has a narrower x-range and a smaller data set size (ranging from 0 to 5e7) than the operators (range from 0 to 1e8) discussed in Chapter 4 to save GPU time.

In the end, we compare the libcudf execution time of all the operators we discussed in the thesis.

4.4.1 Left Join Operator

We constructed two GPU memory tables, each containing a single column of randomly generated 32-bit integers modulo N, where the length of each column is also N. We join these two tables on that column with the libcudf operator `cudf::left_join`. By varying N, we generated data points depicted in 4.8a.

It is important to highlight that the Left Join Operator can potentially produce an output size larger than the input data size when the key is not unique. Consequently, we encountered a CUDA out of memory error when N exceeded 14264264, while for other operators, we can have $N > 1e8$. This underscores the limitation of the libcudf join operator in terms of GPU memory utilization. Moreover, in a CPU/GPU hybrid query execution engine, dealing with large result data sizes that need to be transferred back to the CPU can introduce significant overhead compared to other operators.



(a) libcudf Left Join Operator

(b) libcudf Group Aggregate Sum Operator

Figure 4.8: Join Operator and Group Aggregate Sum Operator of libcudf

4.4.2 Group Aggregate Sum Operator

We conducted libcudf Group Aggregate Sum Operator on a value column containing N 32-bit integers ranging from 0 to 9, with a key column of the same length, consisting of 32-bit integers spanning from 0 to 2 i.e. three groups. The results for various values of N from 1 to 5e7 are presented in Figure 4.8b. As we can see from this figure as well from Figure 4.10, Group Aggregate Sum Operator exhibits the most pronounced dispersion and appears to be the most adversely affected by cache-related effects. Interestingly, the Group Aggregate Sum Operator has similar GPU efficiency to the more intricate Sort Operator, yet it falls much behind Scalar Aggregate Sum Operator in performance. Further discussion of Figure 4.10 about this can be found in Section 4.4.4.

4.4.3 Scalar Aggregate Sum Operator

We carried out the same four micro-benchmarks as presented in Chapter 4, yielding Figure 4.9a and Figure 4.9b. Both figures plot data volume on the x-axis against execution time in milliseconds on the y-axis. The former figure, Figure 4.9a, considers inputs located on the GPU. From this, we deduce that for optimal performance, data computation should remain on the GPU as much as possible when the input is on the GPU, which is the same conclusion in Chapter 4.

Nevertheless, Figure 4.9b offers a differing perspective. For all operators in Chapter 4, considering the data transfer times, the GPU eventually surpasses the CPU in performance. However, this isn't true for the Scalar Aggregate Sum Operator. Here, even libcudf outperforms Acero, as illustrated in Figure 4.9c, where the x-axis represents data volume and the y-axis depicts execution times for both Scalar Aggregate Sum Acero and Scalar Aggregate Sum libcudf Operators.

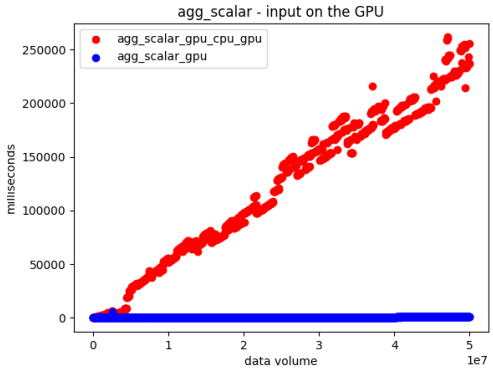
Yet, when compared to other operators as shown in Figure 4.9d and Figure 4.10, the Scalar Aggregate Sum Operator is most efficient compared to other operators. Disregarding data transfer times, the libcudf Scalar Aggregate Sum Operator can achieve a 40x speed up than Acero Operator, as shown in Figure 4.9e. However, when transfer times are included, as seen in Figure 4.9f, the GPU doesn't provide any acceleration for the Scalar Aggregate Operator. This discrepancy can be attributed to the fact that while the potential acceleration rate $C_{potential}$ in Equation 4.9 may be greater than 1, the real acceleration rate C_{real} as per Equation 4.7 can be less than 1 when the downgrade ratio in Equation 4.10 is large and when $C_{potential}$ is relatively small. The downgrade ratio for Scalar Aggregate Operator is large because the GPU computing time is very small compared to the data transferring time.

4.4.4 Comparison of operators

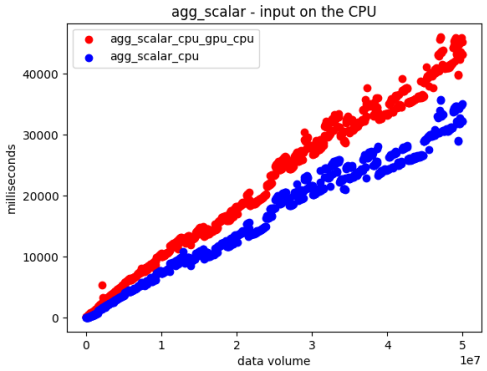
In Figure 4.10, we compare the execution time of libcudf operators. The x-axis represents the data volume, spanning from 0 to 5e7, while the y-axis indicates the execution time measured in milliseconds. The execution time needed for these operators, in descending order, is as follows: Left Join Operator $>$ Group Aggregate Sum Operator $\geq$ Sort Operator $>$ Filter Operator $>$ Copy Operator $>$ Scalar Aggregate Sum Operator.

Left Join Operator requires matching keys from two datasets. This involves searching for each key from one dataset in another, implemented using hash tables. Hash table operations on GPUs can lead to non-coalesced memory accesses (where memory reads/writes are scattered) and might not utilize the full parallelism capabilities of GPUs.

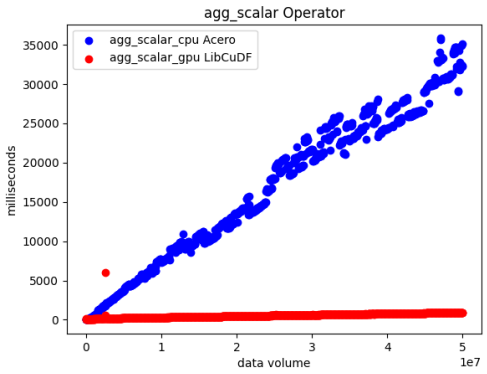
Group Aggregate Sum Operator is based on hashing. In our experiment, with 72 CUDA cores and only 3 groups, there is a potential for underutilization of the available parallel processing capacity during the aggregation phase. The operator needs to hash the input into groups, divide each group into smaller chunks, and assign each chunk to a separate CUDA core. Each CUDA core will then calculate the sum for its assigned chunk in parallel. Once all cores have computed the sum of their respective chunks, it needs to sum up these intermediate sums to get the final sum for the entire group. This process can have random memory accesses, and might require atomic operations.



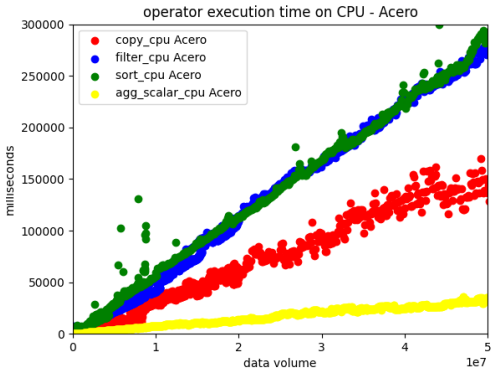
(a) Scalar Aggregate Operator from GPU



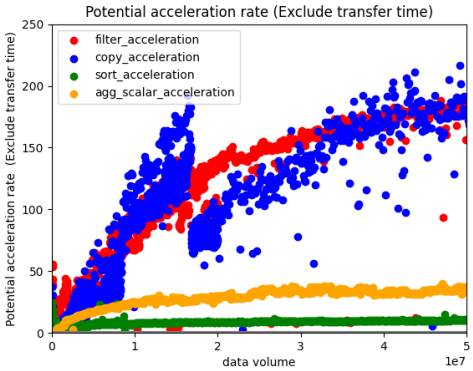
(b) Scalar Aggregate Operator from CPU



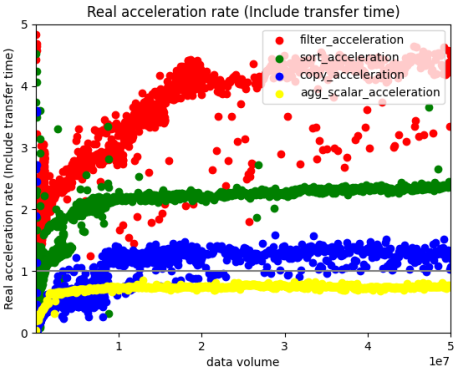
(c) Local performance of Scalar Aggregate Operator on CPU and GPU



(d) Execution time of Acero operators - Scalar Aggregate Operator included



(e) Acceleration rate excluding data transfer time - Scalar Aggregate Operator included



(f) Acceleration rate including data transfer time - Scalar Aggregate Operator included

Figure 4.9: Scalar Aggregator Operator

Sort Operator causes scattered memory reads/writes, especially if the data distribution is not favorable. Some sorting algorithms may also require synchronization steps, especially during merging phases or when operating on shared data.

Filter and Copy are straightforward Operators. Filtering involves evaluating a condition for each data item, and copying is just reading from one location and writing to another. Both of these can be implemented with high degrees of parallelism and coalesced memory accesses.

Scalar Aggregate Sum Operator is the simplest form of aggregation, where values from the entire dataset are combined. GPUs can efficiently compute this using parallel reduction algorithms. The data access pattern here is coalescence, making memory access efficient.

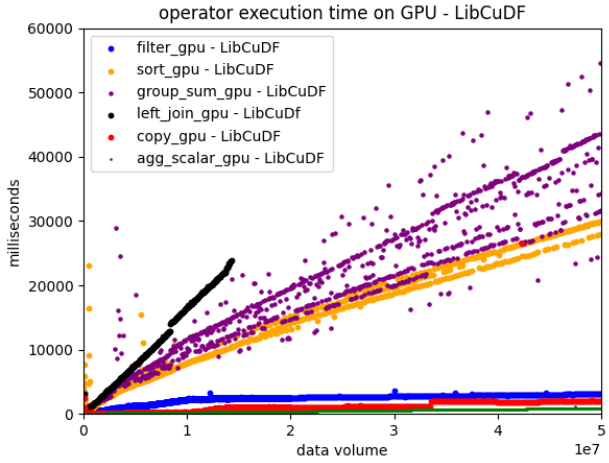


Figure 4.10: Comparing the execution time of libcudf operators

Chapter 5

Scheduler design and evaluation

In this chapter, the architecture design and implementation detail of the scheduler are presented. Chapter 4 presents statistical evidence that demonstrates the feasibility of accelerating Acero using libcudf even when taking into consideration the significant overhead of the PCI-e bus. We also modeled the real acceleration rate to explain why it ramps up from zero and converges to a constant as the data volume increase for each operator, and why different operators converge to different constants.

According to the micro-benchmarks result in Chapter 4, we have the idea that if the original input data is on the CPU when the data volume is small, it is not worth distributing the operator to the GPU to execute. Only when the data volume is bigger than a threshold, the GPU can overcome the data transfer overhead to actually accelerate the operator. However, in real-world execution, the data volume is not known until the runtime of the operator. Therefore in this chapter, we aim to establish that Acero can be dynamically accelerated with an on-the-fly scheduler at the operator level in a hybrid CPU-GPU system.

Given that both Acero and libcudf are complex systems that require significant implementation effort, we designed and implemented a scheduler to simulate the on-the-fly algorithm to dynamically distribute the operators between the CPU and GPU instead of fully integrating the libcudf into Acero.

Similar to other frameworks presented in the literature [19], the scheduler in our study employs a framework that can learn the execution models for various operators with statistical methods treating the underlying hardware system as a black box. This approach further allows for the adaptation of a cost model based on the conditions during runtime. By leveraging this framework, our scheduler provides a means of dynamically distributing the operators between the CPU and the GPU on the fly.

5.1 Design

From Section 4.3, we know the estimation of execution time is a complex task since the total execution time is influenced by many factors. Two of them are dynamic parameters during the system runtime [19], namely, system load and data characteristics. Due to the inherent difficulty in modeling these parameters, the process of effectively guiding a scheduling decision between

CPUs and GPUs becomes challenging, with multiple ramifications. To overcome this issue, the proposed framework considers the underlying system hardware as a black box and adopts a cost model that adapts to the runtime conditions.

We define the problem as follows.

If the original location of the input is at the CPU memory, then

$$\begin{cases} t_{total}^{CPU} = t_c^{CPU} & \text{if computing on CPU} \quad (1) \\ t_{total}^{GPU} = t_i^{CPU->GPU} + t_c^{GPU} + t_o^{GPU->CPU} & \text{if computing on GPU} \quad (2) \end{cases} \tag{5.1}$$

if the original location of the input is at the GPU memory, then

$$\begin{cases} t_{total}^{CPU} = t_i^{GPU->CPU} + t_c^{CPU} + t_o^{CPU->GPU} & \text{if computing on CPU} \quad (1) \\ t_{total}^{GPU} = t_c^{GPU} & \text{if computing on GPU} \quad (2) \end{cases} \tag{5.2}$$

$$\text{Preferred processor} = \begin{cases} \text{GPU} & \text{if } t_{total}^{GPU} < t_{total}^{CPU} \\ \text{CPU} & \text{otherwise} \end{cases} \tag{5.3}$$

where $t_i^{CPU->GPU}$ and $t_i^{GPU->CPU}$ are time cost for the input data to be transferred from the original location to the remote location. This input data transfer time is affected by the input size, the hardware parameters and the system load.

t_c^{CPU} is the operator compute time of Acero on CPU. t_c^{GPU} is the operator compute time of libcudf on GPU. In addition to the input size, the hardware parameters and the system load, the computing time is also affected by the operator specific dynamic multidimensional parameters such as cardinality and selectivity.

$t_o^{CPU->GPU}$ and $t_o^{GPU->CPU}$ are time cost for the result or the output data to be transferred from the remote location to the original location. This output transfer time is affected by the output size, the hardware parameters and the system load. The output size, in turn, depends on the operator-specific dynamic data-related multidimensional parameters.

In summary, the total time cost is influenced by: the input size, the hardware parameters, the system load and the operator specific dynamic data-related multidimensional parameters

We consider the hardware parameters as static and influence all stages. And we assume the system load to be dynamic, hard to model, hard to measure and hard to influence [19]. Therefore, the problem has been simplified into modeling a black box function for each original location and remote location combinations. The black box model uses the variables of input size and operator-specific dynamic data-related multidimensional parameters to estimate the total time cost. Moreover, previous research [17, 18] has provided many ways to estimate these runtime operator-specific dynamic multidimensional parameters. According to that, we assume that these parameters can be obtained prior to scheduling decisions and can be used as input parameters for the operator execution model learned by our framework.

Therefore, for each combination of the original input location, computing location, and operator, there is a black box function. The input variable of the black box function is the data volume and the estimated runtime operator-specific dynamic multidimensional parameters, the output is the estimation of the total time cost. To model these black box functions, various techniques can be used, for example, using static analytical models or learning-based dynamic approaches. To be flexible, with less overhead, and make it easier and faster to train, optimize, and deploy, we opted to use earning-based approaches, specifically, two-degree least squares polynomial models.

The decision model is then designed as shown in Figure 5.1. During runtime, we know the current original input location (L_{oi}, which can either be the CPU or the GPU), the Operator O, the data volume N and the estimated runtime operator-specific dynamic multidimensional parameters (represented by the array α). There is a model pool for each combination of (L_{oi}, O), within it, there are two models $(M_{oi,c1}, M_{oi,c2})$ to estimate the total time cost for different computing locations (L_{c1}, L_{c2}) separately. With each model within the model pool, we can estimate the total time cost for each computing location $(T_{est}(L_{oi}, L_{c1}, N, \alpha))$ given the data volume N and α. The decision component then uses Equation 5.1, Equation 5.2 and Equation 5.3 to decide the computing location L_{cj} to execute the operator on. We keep track of the real execution time of that computing location $(T_{real}(L_{oi}, L_{cj}, N, \alpha))$ and add it into the online dataset for that model. For every $M_{oi,cj}$, there is a corresponding dataset $D_{oi,cj}$. The datasets are initialized during the first system set up and new data points are added to the datasets during runtime to periodically retrain the models.

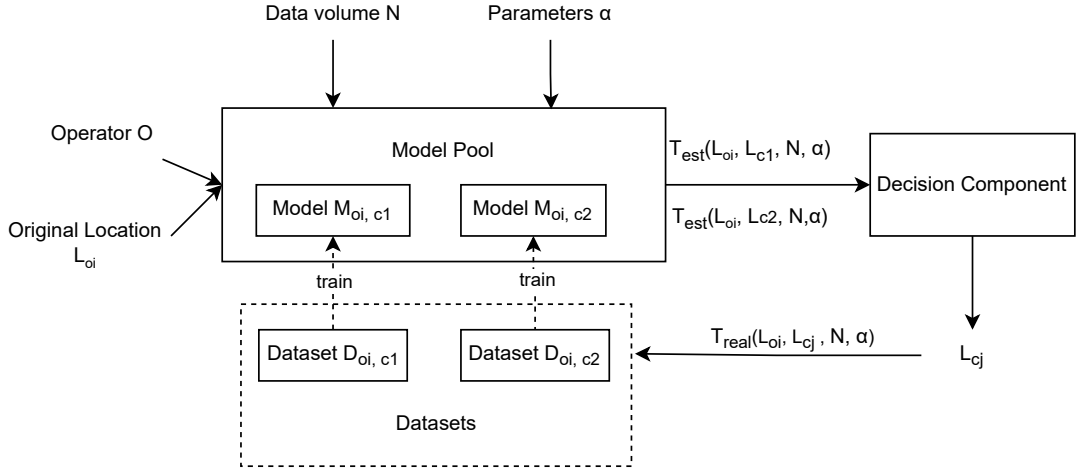


Figure 5.1: Decision model overview

We adopt the modeling workflow described in Figure 5.2 which is similar to the modeling workflow in this paper [19]. It includes two phases: the training phase and the execution phase.

The initial training of the operators' models should be done once the system is set up. At the initial training phase, we ran the micro-benchmarks discussed in Section 4.2.2 varying the data volume to get the time measurement for all data points and build datasets for each combination of (O, L_{oi}, L_{cj}). These datasets are used to fit the two-degree least squares polynomial models. As we only want to prove the feasibility of the scheduler, we consider the runtime operator-specific dynamic multidimensional parameters such as cardinality and selectivity, as fixed.

In the execution phase, we can use the models to guide the scheduling decisions (Figure 5.1). Given the trained model, we first estimate the time it takes to execute the operator on the both the CPU as well as the GPU. Then, the processor with the least estimated time will be selected, and then used to execute the operator. During execution, we keep track of the runtime and add it to our measurement database. Finally, the new measurement time will be used to periodically retrain the model to better adapt to the current workload and input set.

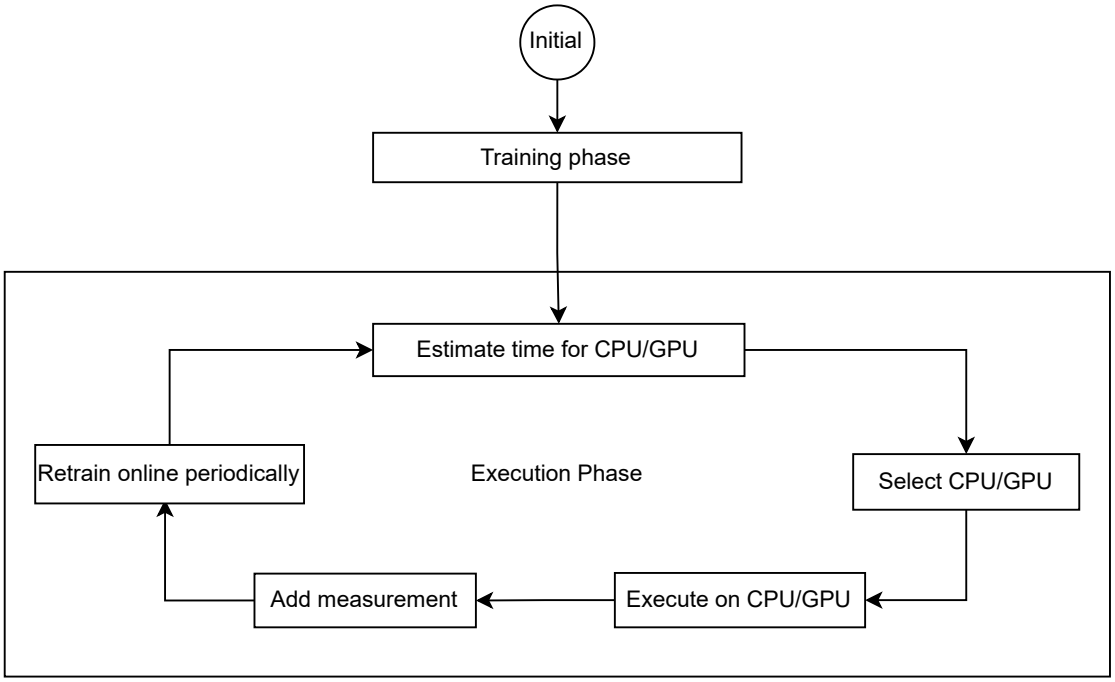


Figure 5.2: Modeling workflow

5.2 Implementation

The actual implementation of the scheduler is presented as a UML diagram in Figure 5.3. The block diagram shows five different classes: scheduler.Scheduler, models.Operator, models.Location, models.ExecutionContext, and models.Algorithm. The main class is the Scheduler Class. A demo code of using this scheduler is as listed below:

```
"""
The scheduler use simple 2 degree Least squares polynomial models
  to estimate duration given a 'ExecutionContext'
  (which consists of four dimension - operator, the current data location, data
      size, data parameters)
"""
scheduler = Scheduler() # will load all dataset and first train all models
ctx = ExecutionContext(
    operator=Operator.Copy, location=Location.CPU, data_size=500000,
        data_parameters=None
)
print(scheduler.schedule(ctx)) # will output the location that takes least
    estimate response time

"""
The scheduler can also online retrain itself periodly, the default retrain period
    is after every 10 times of measurements of that algorithm
```

```
"""
for i in range(10):
    scheduler.add_execution_data(Operator.Copy, Location.GPU, Location.GPU, 100,
        100, None)
# will retrain after 10 times
```

In Figure 5.3, we show the UML diagram of the scheduler to illustrate the structure of the scheduler design. The design comprises five classes, with the scheduler class serving as the main component. The scheduler class is tasked with initial training, periodic retraining, model evaluation, and the scheduling of operators (Operator) based on the current execution context (ExecutionContext). This execution context includes factors such as the current input location (Location, either the CPU or GPU), data volume, and Operator-specific dynamic data-related parameters. For every combination of (operator, origin location, and remote location), there exists a corresponding model paired with a dataset. The Model class is responsible for maintaining datasets and the models. These models undergo an initial training phase and periodic retraining as needed and will be used to predict the execution time.

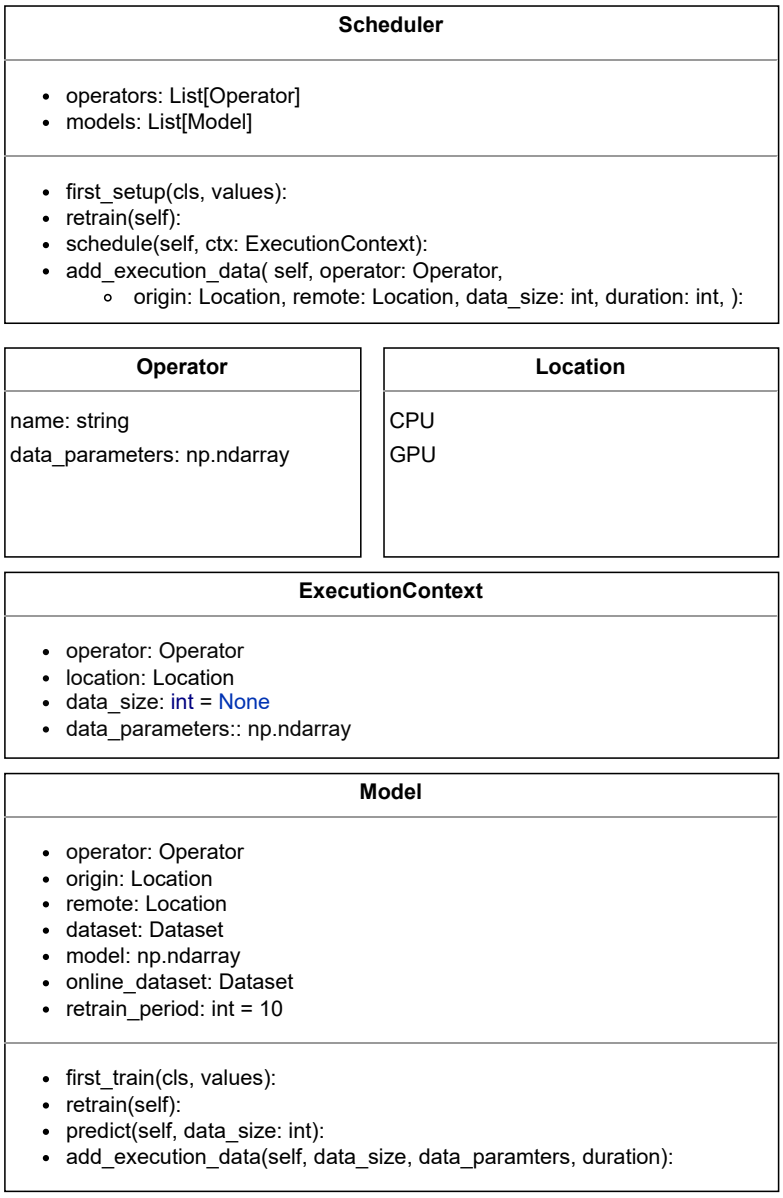


Figure 5.3: Scheduler UML diagram

5.3 Evaluation of the scheduler

In this section, we evaluate the scheduler from three aspects:

1. The error metrics of the time estimation models for each micro-benchmark in Chapter 4 to evaluate how well the time estimation model is able to predict the total time cost it needs for each micro-benchmark.

2. The error metrics of the scheduling decision to evaluate how well the scheduler is able to decide whether the CPU or GPU is better to execute the operators.

3. The speedup ratio after dynamically distributing the operator work between CPUs and GPUs with the scheduler.

5.3.1 Time estimation models evaluation

Chapter 3.3 illustrated that for each combination of (operator, original location, remote location), we execute a micro-benchmark to generate a dataset $D^{r(emote)}_{o(riginal)}$ for each operator that can be used to train the scheduler models. With each dataset $D^{r(emote)}_{o(riginal)}$, we trained a corresponding statistics model $M^{r(emote)}_{o(riginal)}$ to predict the estimated time cost. The statistics model is a polynomial regression model with a degree of 2. We visualize these models in Figure 5.4, Figure 5.5, Figure 5.6, for the Copy, Filter and Sort Operators, respectively. In each figure, subfigure (a) visualizes the execution time in milliseconds (on the y-axis) for test data points across varying data volumes (on the x-axis). It further presents the polynomial model fitted to the combination of (Operator, origin input location=CPU, and compute location=CPU). Subfigures (b), (c), and (d) follow a similar pattern, each representing different combinations of input and compute locations.

The models generally fit well with the test data points, except for Figure 5.4c, which suggests the GPU is significantly affected by the cache effects. In order to evaluate the error of the predictive models, we introduce a number of error metrics. These error metrics are defined as follows:

- Mean Squared Error (MSE): Average of the squared differences between predicted and actual values, penalizing larger errors more heavily.

$$\text{MSE} = \frac{\sum_{i=1}^{D}(x_i - y_i)^2}{|D|} \tag{5.4}$$

- Root Mean Squared Error (RMSE): Square root of MSE (i.e., $\sqrt{\text{MSE}}$), providing a measure of the average magnitude of residuals in the original units of the response variable.

- Mean Absolute Error (MAE): Average of the absolute differences between predicted and actual values, providing a metric that is not influenced by the scale of the data.

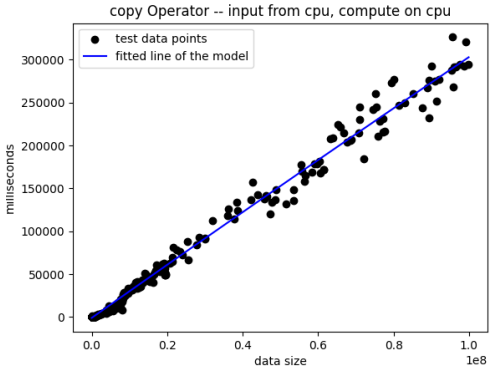
$$MAE = \frac{\sum_{i=1}^{D}|x_i - y_i|}{|D|} \tag{5.5}$$

The error metrics of the polynomial regression models are given in Table 5.1. These metrics have been normalized with the range of the y-axis. $\hat{MSE}$, $\hat{RMSE}$ and $\hat{MAE}$ refer to the normalized versions of MSE, RMSE and MAE, respectively.

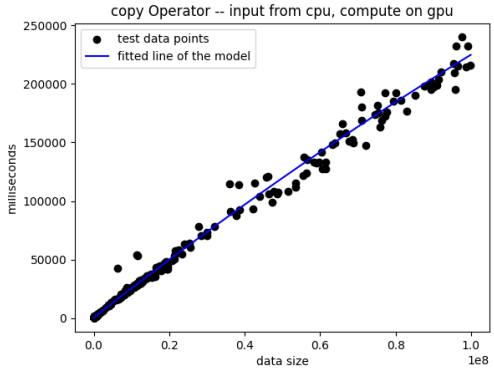
In summary, the table shows that the models generally perform well when the operators are conducted locally on the CPU or GPU. However, when the computing location is different from the location where the original data is located, especially when moving data from the GPU to the CPU, the models tend to have higher error metrics, indicating less accurate time estimates.

5.3.2 Schedule decision models evaluation

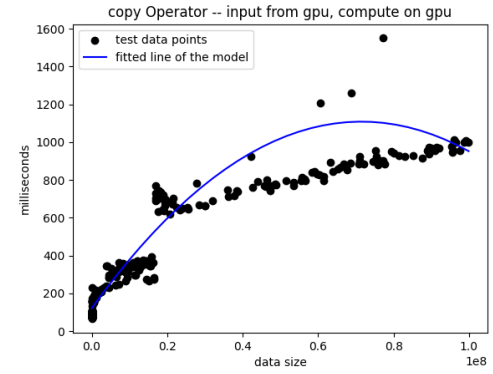
Given an operator and the current location of the input data, the scheduler will estimate the time cost for distributing the operator between CPU and GPU, and make the decision according to



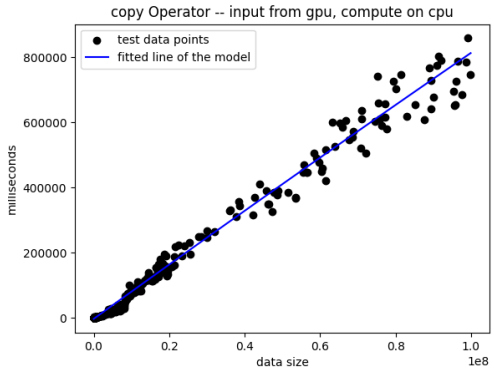
(a) Copy Operator Model – input from the CPU, compute on the CPU



(b) Copy Operator Model – input from the CPU, compute on the GPU

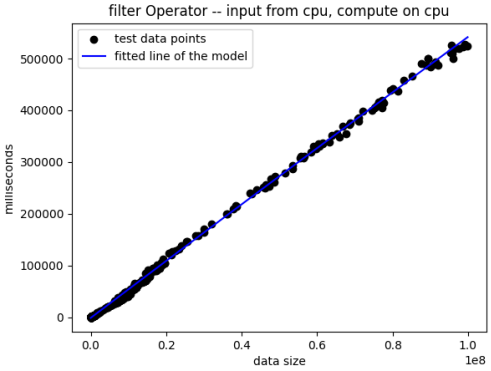


(c) Copy Operator Model – input from the GPU, compute on the GPU

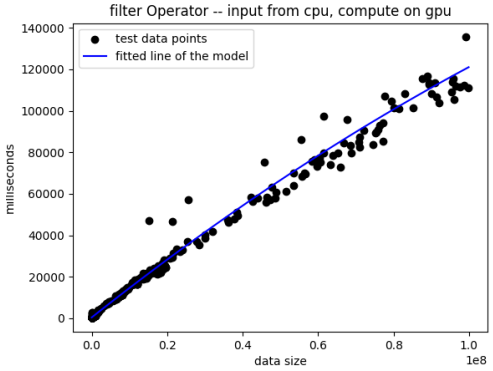


(d) Copy Operator Model – input from the GPU, compute on the CPU

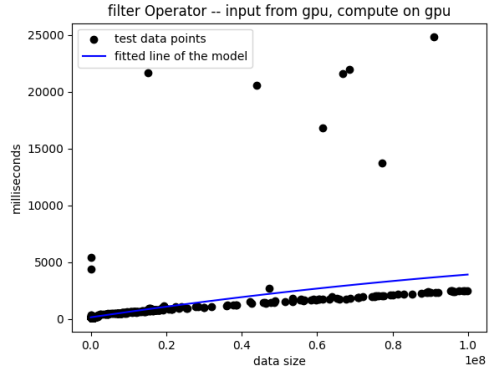
Figure 5.4: Models of Copy Operator



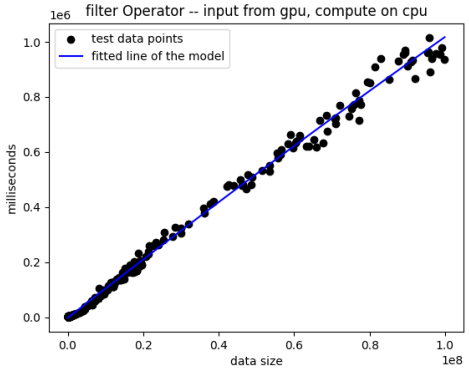
(a) Filter Operator Model – input from the CPU, compute on the CPU



(b) Filter Operator Model – input from the CPU, compute on the GPU



(c) Filter Operator Model – input from the GPU, compute on the GPU



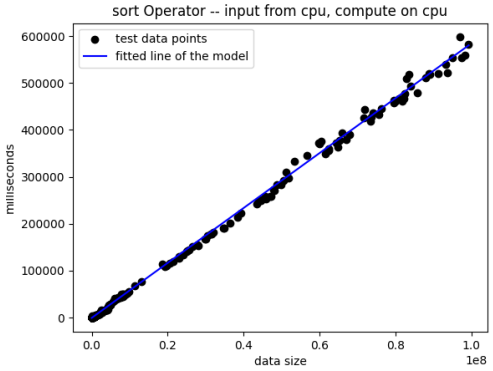
(d) Filter Operator Model – input from the GPU, compute on the CPU

Figure 5.5: Models of Filter Operator

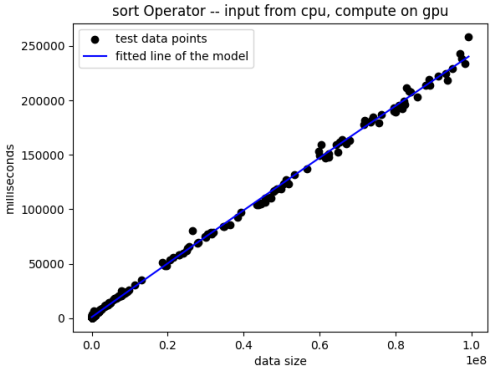
the decision model in Equation 5.3.

This scheduling decision model can be viewed as a binary classification problem with labels CPU and GPU, the predication labels are the decisions made according to the estimation time cost, the true labels are the decisions that should be made according to the real time cost. This allows us to represent the results of this evaluation as a confusion matrix as shown in Table 5.2 along with the accuracy of the classification model.If N means the number of all data points, and $N[predict = A, true = B]$ means the number of data points that the model predict as label A, however the true label is B, the terms in Table 5.2 can be defined as the following equations:

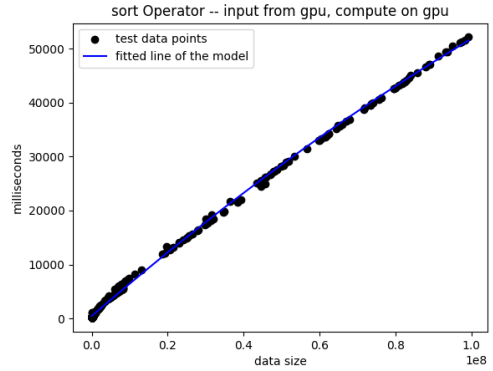
$$TN = \begin{cases} \frac{N[predict=GPU,true=GPU]}{N} & \text{if the original data is on CPU} \quad (1) \\ \frac{N[predict=CPU,true=CPU]}{N} & \text{if the original data is on GPU} \quad (2) \end{cases} \tag{5.6}$$



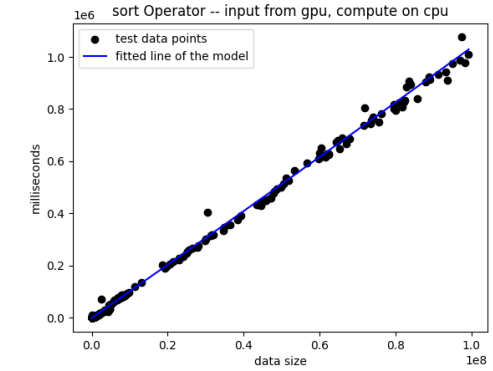
(a) Sort Operator Model – input from the CPU, compute on the CPU



(b) Sort Operator Model – input from the CPU, compute on the GPU



(c) Sort Operator Model – input from the GPU, compute on the GPU



(d) Sort Operator Model – input from the GPU, compute on the CPU

Figure 5.6: Models of Sort Operator

$$FP = \begin{cases} \frac{N[predict=CPU, true=GPU]}{N} & \text{if the original data is on CPU} \quad (1) \\ \frac{N[predict=GPU, true=CPU]}{N} & \text{if the original data is on GPU} \quad (2) \end{cases} \tag{5.7}$$

$$FN = \begin{cases} \frac{N[predict=GPU, true=CPU]}{N} & \text{if the original data is on CPU} \quad (1) \\ \frac{N[predict=CPU, true=GPU]}{N} & \text{if the original data is on GPU} \quad (2) \end{cases} \tag{5.8}$$

$$TP = \begin{cases} \frac{N[predict=CPU, true=CPU]}{N} & \text{if the original data is on CPU} \quad (1) \\ \frac{N[predict=GPU, true=GPU]}{N} & \text{if the original data is on GPU} \quad (2) \end{cases} \tag{5.9}$$

$$\text{Accuracy Score} = \frac{TN+TP}{TN+FP+FN+TP} \tag{5.10}$$

Table 5.1: Evaluate the time estimate models

Operator	Origin	Remote	$\hat{MSE}$	$\hat{RMSE}$	$\hat{MAE}$
copy	cpu	cpu	217.17	0.02	0.01
copy	cpu	gpu	142.90	0.02	0.01
copy	gpu	gpu	1333.14	0.04	0.02
copy	gpu	cpu	7.59	0.08	0.06
filter	cpu	cpu	33.77	0.01	0.00
filter	cpu	gpu	109.86	0.03	0.01
filter	gpu	gpu	1429.03	0.04	0.01
filter	gpu	cpu	170.28	0.09	0.02
sort	cpu	cpu	269.58	0.02	0.01
sort	cpu	gpu	155.34	0.02	0.01
sort	gpu	gpu	6.07	0.01	0.00
sort	gpu	cpu	538.16	0.02	0.01

Table 5.2: Confusion Matrix and Accuracy Scores

	TN	FP	FN	TP	Accuracy Score
copy from cpu	0.6125	0.0275	0.0	0.3600	0.9725
copy from gpu	0.0	0.0	0.0	1.0	1.0
filter from cpu	0.7150	0.0575	0.0	0.2275	0.9425
filter from gpu	0.0	0.0120	0.0	0.9880	0.9880
sort from cpu	0.4164	0.1056	0.0	0.4780	0.8944
sort from gpu	0.0	0.0752	0.0	0.9248	0.9248

The accuracy scores for most of the scheduling decisions are quite high, indicating that the model is performing well in making the correct choices between CPU and GPU based on the estimated time costs. In particular, the model achieves perfect accuracy in decisions when the original data is on GPU and the true decisions are almost all on GPU. Some decisions, such as "Sort from CPU", have slightly lower accuracy but are still well above 80%, suggesting that the model is generally effective.

5.3.3 Speedup of dynamic scheduling evaluation

Here, we use our scheduler to schedule the operator dynamically and evaluate the speedup between the scheduled time cost and the time cost if the operator is executed at the original location. We calculate the speedup ratio for 500 test data points for each combination of the operator and original input data location and give a statistics table of the speedup ratios in Figure 5.7.

For operators such as "copy from GPU," "filter from GPU," and "sort from GPU," dynamic scheduling does not lead to any improvement as the speedup is consistently 1. This suggests that executing these operations at the original location is just as efficient as dynamically scheduling them.

The operators "copy from CPU," "filter from CPU," and "sort from CPU" experience en-

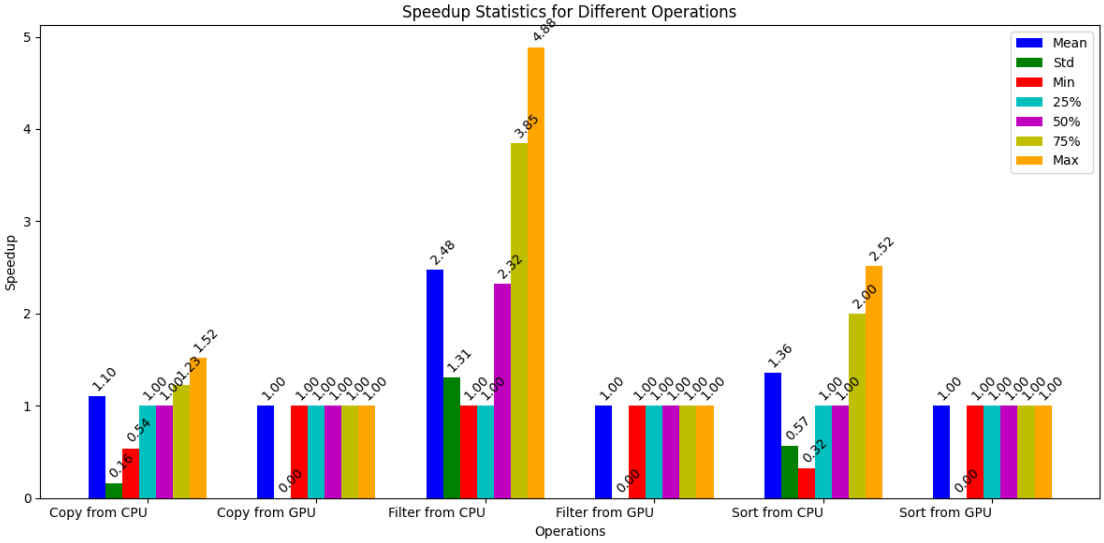


Figure 5.7: Speedup statistics for various combinations of original locations of the data for different operators

hancements through dynamic scheduling, with each showing varying levels of performance boost. On average, these operators (Copy Operator, Filter Operator, and Sort Operator) record speed increases of 1.10x, 2.48x, and 1.36x, and they attain peak speedups of 1.52x, 4.88x, and 2.52x, respectively. The sequence remains consistent when considering the standard deviation. These standard deviation figures provide insight into the variability of the speedup outcomes. A higher standard deviation value indicates a greater fluctuation in the speedup realized across distinct data points.

Chapter 6

Conclusions

This chapter gives an overview of the project's contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

6.1 Answer research questions

In order to achieve the primary objective of this thesis, it is necessary to address a number of research questions.

- 1. Is it possible to accelerate Acero in a hybrid CPU-GPU system?

Yes, Acero is a query execution engine with Arrow data format and can be accelerated with libcudf at the operator level. When disregarding the data transferring time, it demonstrates a remarkable speedup of up to 300x for Filter Operator, 100x for Copy Operator, and 16x for Sort Operator when handling an array of length 1e8. However, when accounting for data transfer times, this promising acceleration rate diminishes significantly. In cases where the data volume is small, the acceleration benefits are unable to offset the associated overhead. Consequently, Acero is not suitable for GPU acceleration with small data volumes.

To address this challenge, we have developed an on-the-fly scheduler at the operator level. This scheduler determines whether to allocate an operator to the CPU or GPU based on the input data location and the type of operator involved. This approach enables us to accelerate Acero within a hybrid CPU-GPU system, guided by a statistical cost model. The results are notable, with a maximum speedup of 4.88x for the Filter Operator, 2.52x for the Sort Operator, and 1.52x for the Copy Operator when processing an array of length 1e8.

- 2. What are the characteristics of workloads that result in high acceleration and what are the possible bottlenecks

Without taking into account of the data transfer time, workloads that can be effectively parallelized, such as Filter Operator, possess greater potential for benefiting from the GPU's substantial parallel processing capabilities and high throughput. Conversely, the queries that

have more complex branching structures can not fully exploit the instruction-level parallelism offered by GPUs, leading to limited acceleration potential.

When factoring in data transfer times, the acceleration rate experiences a significant decline across all operators primarily due to the PCI-e bus serving as the limiting factor in harnessing GPU capabilities. This effect is especially pronounced for operators with short GPU execution times in comparison to the data transfer time.

Furthermore, operators such as Filter Operator, where the size of the result dataset is smaller than the input data size, exhibit a relatively smaller reduction in their acceleration rates.

- 3. What are the limitations of speedup achievable using CPU-GPU systems?

As previously mentioned, the most significant limiting factor is the memory bandwidth of the PCI-e bus. Additionally, the relatively small size of GPU memory serves as another constraint in fully realizing acceleration potential, given that both input and result data must fit within it. Another factor contributing to the limitation is the relative difficulty in implementing some operators on the GPU, stemming from the complexity of algorithm parallelization due to the use of many control instructions within the algorithms.

6.2 Future work

According to the answers to the research questions, we propose future work directions that can further be investigated.

In our research, the acceleration of the GPU is limited by the memory size of a single GPU. This limitation could be addressed by splitting the input into partitions, executing these partitions on a cluster of GPUs in parallel, and merging the output result. This requires Acero to have both shuffle and partitioning capabilities in order to run operators on a cluster of GPUs. However, Acero does not have shuffle and partition nodes yet, though this feature is identified as a high priority for the developers. With this feature, multiple GPUs can be utilized and a shuffle can be done. Therefore, one future research direction could be to distribute the acceleration of Acero to multiple GPUs to remove the GPU memory limit.

Another open research question is related to how libcudf-Accelerated Acero measures up against the standard TPC-H benchmark. In order to answer this question, a complete set of operators of Acero needs to be implemented. By doing this, we can fully integrate and accelerate Acero with libcudf and run the standard TPC-H benchmark to compare with other engines like Spark, Flink, and Velox.

Bibliography

[1] Jens Glaser, Felipe Aramburu, William Malpica, Benjamin Hernandez Arreguin, Matthew Baker, and Rodrigo Aramburu. Scaling sql to the supercomputer for interactive analysis of simulation data.

[2] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *KDD 2022*, 2022.

[3] Nauman Ahmed, Tong Dong Qiu, Koen Bertels, and Zaid Al-Ars. Gpu acceleration of darwin read overlapper for de novo assembly of long dna reads. *BMC Bioinformatics*, 21(13):388, Sep 2020.

[4] Johan Peltenburg, Ákos Hadnagy, Matthijs Brobbel, Robert Morrow, and Zaid Al-Ars. Tens of gigabytes per second json-to-arrow conversion with fpga accelerators. In *2021 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9, 2021.

[5] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. Gpu-accelerated database systems: Survey and open challenges. *Trans. Large Scale Data Knowl. Centered Syst.*, 15:1–35, 2014.

[6] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous cpu/gpu systems. *ACM Comput. Surv.*, 55(1), jan 2022.

[7] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 94–103, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), dec 2009.

[9] Apache Arrow. Apache arrow overview, 2023.

[10] Tanveer Ahmad, Nauman Ahmed, Johan Peltenburg, and Zaid Al-Ars. Arrowsam: In-memory genomics data processing using apache arrow. In *2020 3rd International Conference on Computer Applications Information Security (ICCAIS)*, pages 1–6, 2020.

[11] Tanveer Ahmad. Benchmarking apache arrow flight - a wire-speed protocol for data transfer, querying and microservices. In *Benchmarking in the Data Center: Expanding to the Cloud*, BID'22, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Acero. Acero: A c++ streaming execution engine, 2022.

[13] FLARE. Tpc-h on a single core, 2016.

[14] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[15] A Shaikhha, M Dashti, and C Koch. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming*, 28, 2018.

[16] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. Push vs. pull-based loop fusion in query engines. *Journal of Functional Programming*, 28, 10 2016.

[17] Chungmin Melvin Chen and Nick Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, page 161–172, New York, NY, USA, 1994. Association for Computing Machinery.

[18] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, page 461–472, New York, NY, USA, 2001. Association for Computing Machinery.

[19] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013.