Libcudf project update

Guangyu Meng, John Lalor
University of Notre Dame

March 2024

1 Introduction

In the era of big data, the volume and complexity of data have exploded, necessitating efficient storage and processing techniques. The Apache Parquet file format [2] has emerged as a popular choice for storing and managing large datasets due to its columnar structure, compression capabilities, and support for complex nested data types. Parquet's ability to efficiently store and retrieve data has made it a widely adopted format across various domains, including data warehousing, analytics, and machine learning [3]. However, as data sizes continue to grow, the need for faster and more efficient methods to read Parquet files becomes increasingly crucial. Recently, the libcudf library has been introduced to leverage the parallel processing power of GPUs for reading Parquet files [1]. While this approach shows promise, our initial experiments indicate that libcudf's performance still lags behind the CPU-based libarrow library. In this paper, we explore various techniques to optimize Parquet file reading using libcudf, aiming to fully harness the potential of GPU parallelism and achieve superior performance compared to traditional CPU-based approaches.

2 Dataset description and Experiment setup

This section presents the dataset employed for the benchmark, the hardware platform on which the tests were executed, and the baseline and modified methodologies utilized in the study.

2.1 Dataset description

Property	**Value**
High-level description	Released by DataPelago
Number of rows	240,578
Number of columns	14
Compressed file size	39 MB
Uncompressed file size	50 MB

Note that, the compressed file size is measured by ‘‘`du -sh`’’ on linux, and the uncompressed file size is measured by ‘‘`pyarrow.metadata`’’.

Column Names and Data Types

Column Name	**Data Type**
alert_id	object
account_id	object
team_id	object
schedule_id	object

Column Name	Data Type
escalation_id	object
row_refreshed_at	datetime64[ns]
row_refreshed_at_day	datetime64[s]
shard_id	object
workspace_id	object
envelope_fields_workspace_id_value	object
envelope_fields_resource_ari_value	object
envelope_fields_version	int64
envelope_fields_is_tombstone	bool
envelope_fields_generation_counter	int64

We further expands the original parquet file into millions of rows with or without snappy compression. Here is the de

Table 3: Description of File Sizes with different row number

Row Number	File Size (MB)	
	Snappy Compression	No Compression
24K (original)	39	51
1M	52	180
2M	57	298
4M	65	457

2.2 Hardware platform

We conducted all experiments on the following hardware: CPU: Intel(R) Core i9-10850K CPU @ 3.60GHz with 64GB of DDR4 memory, and GPU: GeForce RTX 3080 with 12GB of memory.

2.3 Baseline and compared methods description

In this section, we introduce the methodologies employed in our study. We also discuss the motivations behind the modifications applied to the baseline method and introduce the process of implementing these modifications at a high-level.

2.3.1 Describe multi-streams and multi-threads methods

In the following implementations, we refer to *multi-streams* and *multi-threads* methods. Here, we provide a general description of these concepts. Typically, reading a Parquet file involves several sequential operations in a pipeline: raw I/O, header decoding, decompression, and data decoding. These steps are often challenging to parallelize. When handling a large Parquet file, the process involves multiple pipeline stages. Multi-streams can be utilized to overlap these stages, thereby enhancing efficiency. In terms of multi-threads, the Parquet file is divided into chunks that are read concurrently across multiple threads. Ideally, by combining multi-threads and multi-streams, we can achieve parallel reading of a Parquet file at both the chunk and kernel levels, optimizing performance.

For comparing performance, we have looked at off-the-shelf libcudf implementations (*base implementations*) as well as more engineered solutions (*custom implementations*).

- Base implementations
  - **lib_arrow:** The benchmark method derived from Apache Arrow.

- **parquet_read:** The baseline method sourced from the libcudf library, version 24.06. This approach loads the entire file into memory, can lead to inefficiencies in data movement, particularly with large files.

- Custom implementations

  Based on our analysis of the textbfparquet_read using the nsight-system profiling tool, we identified that data movement and the `gpuDecodeStringPageData` kernel account for 18.3% and 61.25% of the total time, respectively. Therefore, we have designed custom implementations are designed for both steps.

  - **parquet_read+pool_memory:** pool_memory is provided by the RMM library, which allows to allocate a large "pool" of device memory up-front. Subsequent allocations will draw from this pool of already allocated memory.
  - **parquet_read+pool_memory+ramfs:** ramfs is a type of RAM disk file system. Since ramfs operates directly within RAM, it eliminates the latency typically associated with disk I/O, thereby enhancing file access and manipulation operations."
  - **chunked_parquet_read (2):** We partition the original parquet file into two parts, and then read each chunk sequentially. This is a preparation for the next implementation of multi-threads and multi-streams.
  - **chunked_parquet_read (2) + streams (2) + threads (2):** This approach leverages multiple threads to process respective chunks of the Parquet file, enabling parallel processing of the file's segments. The thread count is set to two, correlating to the partitioning of the Parquet file into two segments. Furthermore, the parquet_read API offers a streaming option. We exploit this feature to design a multi-stream method, utilizing `rmm::cuda_stream_view(streams)`. Notably, multi-streams facilitate asynchronous handling of different kernels within the Parquet read API.

3 Results and analysis

Based on the results shown in Table 5, we can summarize the following points:

- Comparing the benchmark `lib_arrow` with the baseline `parquet_read` method, we find that `lib_arrow` is much faster than `parquet_read`, even though the latter leverages the GPU for acceleration.
- `pool_memory+ramfs`: we recommend to use both options to reduce the data movement and improve the performance of `parquet_read`.
- `chunked_parquet_read` dramatically improves performance compared to `parquet_read`.
- Although multi-streams and multi-threads are promising methods, they do not show significant improvements. As the nsight-system tool demonstrates, the kernel and chunks are well overlapped by using multi-streams and multi-threads. The challenge is the `gpuDecodeStringPageData` kernel does not reduce too much (reduce from 226 ms into 205 ms) even though we correctly implement the multi-streams and multi-threads .

3.1 Exploration on the parameter settings for multi-streams and multi-threads

We further explore the parameters setting affect the performance of chunked read function on 2 million rows with and without Snappy compression.

4 Future plan

We will continue to monitor if there is a solution for multi-streams on a single GPU.[1] Simultaneously, we will conduct tests on multiple GPUs to evaluate the performance.

[1] `https://github.com/rapidsai/cudf/issues/15376`

Table 4: Performance for different methods on parquet files.

Row Number	Method name	snappy compression		no compression	
		Time (ms)	Throughput (Gbps)	Time (ms)	Throughput (Gbps)
24K	lib_arrow	38	10.74	39	10.46
1M	lib_arrow	128	11.25	146	9.86
2M	lib_arrow	224	10.64	267	8.93
4M	lib_arrow	411	8.89	440	8.30
24K	parquet_read	123	3.32	161	2.54
1M	parquet_read	355	4.06	312	4.62
2M	parquet_read	390	6.11	390	6.11
4M	parquet_read	404	9.05	415	8.81
24K	parquet_read + pool_memory	127	3.21	155	2.63
1M	parquet_read + pool_memory	336	4.29	298	4.83
2M	parquet_read + pool_memory	387	6.16	379	6.29
4M	parquet_read + pool_memory	390	9.37	398	9.18
24K	parquet_read + pool_memory + ramfs	119	3.43	157	2.60
1M	parquet_read + pool_memory + ramfs	354	4.07	299	4.82
2M	parquet_read + pool_memory + ramfs	386	6.18	363	6.57
4M	parquet_read + pool_memory + ramfs	395	9.25	380	9.62
24K	chunked_parquet_read (2)	430	0.95	246	1.66
1M	chunked_parquet_read (2)	720	2.00	542	2.66
2M	chunked_parquet_read (2)	932	2.56	803	2.97
4M	chunked_parquet_read (2)	1099	3.33	1002	3.65
24K	chunked_parquet_read (2) + streams (2) + threads (2)	206	1.98	178	2.29
1M	chunked_parquet_read (2) + streams (2) + threads (2)	372	3.87	280	5.14
2M	chunked_parquet_read (2) + streams (2) + threads (2)	438	5.45	408	5.85
4M	chunked_parquet_read (2) + streams (2) + threads (2)	647	5.65	597	6.12

Table 5: Performance for different settings for chunks, streams, and threads on 2M rows parquet file.

Method name	snappy compression		no compression	
	Time (ms)	Throughput (Gbps)	Time (ms)	Throughput (Gbps)
chunked_parquet_read (2) + streams (2) + threads (2)	438	5.45	408	5.85
chunked_parquet_read (2) + streams (4) + threads (4)	430	5.54	576	4.14
chunked_parquet_read (4) + streams (4) + threads (4)	770	3.1	649	3.67
chunked_parquet_read (4) + streams (8) + threads (8)	765	3.12	606	3.93
chunked_parquet_read (8) + streams (8) + threads (8)	1284	1.86	921	2.59

References

[1] Ahmedur Rahman Shovon, Landon Richard Dyken, Oded Green, Thomas Gilray, and Sidharth Kumar. Accelerating datalog applications with cudf. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 41–45, 2022.

[2] Deepak Vohra and Deepak Vohra. Apache parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pages 325–335, 2016.

[3] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. An empirical evaluation of columnar storage formats. *arXiv preprint arXiv:2304.05028*, 2023.