Scaling SQL to the Supercomputer for Interactive Analysis of Simulation Data

Jens Glaser[1], Felipe Aramburú[2], William Malpica[2], Benjamín Hernández[1], Matthew Baker[1], and Rodrigo Aramburú[2]

[1] Oak Ridge National Laboratory, 1 Bethel Valley Rd, Oak Ridge, TN 37831, USA, glaserj@ornl.gov
[2] Voltron Data, Inc., https://voltrondata.com

Abstract. AI and simulation workloads consume and generate large amounts of data that need to be searched, transformed and merged with other data. With the goal of treating data as a first-class citizen inside a traditionally compute-centric HPC environment, we explore how the use of accelerators and high-speed interconnects can speed up tasks which otherwise constitute bottlenecks in computational discovery workflows. BlazingSQL is SQL engine that runs natively on NVIDIA GPUs and supports internode communication for fast analytics on terabyte-scale tabular data sets. We show how a fast interconnect improves query performance if leveraged through the Unified Communication X (UCX) middleware. We envision that future computing platforms will integrate accelerated database query capabilities for immediate and interactive analysis of large simulation data.

1 Introduction

Data-analytics driven scientific discovery is rapidly transforming the landscape of computational and experimental sciences. With the emergence of pre-exascale and exascale computing platforms, harnessing their capabilities for data analytics tasks is essential to address the need for the analysis of ever larger data sets [15]. These datasets, usually represented as numeric data arrays or tabular data, serve as an input to many operations (e.g. filtering, feature extraction, anomaly detection) that produce new arrays or tables. The database community has

provided automatic query processing along with CPU and IO parallelism and more recently, robust GPU support. These capabilities have been shown to be effective for the analysis of a large-scale molecular docking campaign, to identify potential drug candidates [12]. To explore chemical space, libraries containing billions of SMILES strings [29] and associated text and numerical data of small organic molecules are screened in a high-throughput manner, generating terabytes of data that need to be searched, sorted and merged with other data. Another, *in-situ* application of large-scale data analytics in plasma physics uses OpenPMD [24] to stream simulation data into `dask` distributed data frames [16], which can then further be analyzed with Python-based tools such as RAPIDS [19] and BlazingSQL, which we discuss below. Both examples demonstrate the need for fast manipulation of large-scale structured and semi-structured data close to where it is being produced, without the need for costly conversion, transformation or indexing of data, both in terms of time to solution as well as memory or compute footprint.

However, for distributed applications, data movement across the HPC interconnect is expensive and therefore the communication portion of an algorithm is often the bottleneck, particularly when the compute portion is already highly optimized and runs on GPUs. In distributed, GPU-accelerated data analytics tools, a high-performance communication layer is extremely important to leverage low-latency, high-bandwidth and overall scalability when transporting data across nodes and across GPUs [15]. Our contribution in this respect is the implementation of a high performance communication layer on top of the UCX library [25] to enhance BlazingSQL, a GPU accelerated database query engine.

Seminal work for scaling and optimizing database management systems dates back to the early 1990s [9], when foundations for efficient query algorithms, *e.g.* hash joins, were laid. In the early '2000s, the computer graphics community experimented with general-purpose computing on graphics processing units (GPGPU) as a way to accelerate computational workloads beyond graphics applications, and in particular, to speed up selections, aggregations, and semi-linear queries on GPUs [2, 10, 13, 14] and co-processors [20]. Heterogeneous approaches to database query processing combine GPUs with multi-core CPUs or FPGAs [4, 7, 26] to make optimal use of the available hardware. Specific optimizations include the use of compression techniques [11], off-loading to the level of the storage engine [30], learning techniques for query optimization [5] and design of GPU-CPU heterogeneous query plans [17].

Several popular open-source frameworks for GPU-accelerated database query processing are under active development with contributions from the community: BlazingSQL [3], PG-Strom [23], and OmniSciDB [21]. In particular, we focus on BlazingSQL and the implementation of a new communication layer utilizing the UCX library [25].

2 RAPIDS and the BlazingSQL software architecture

RAPIDS is a suite of software libraries that provide data analytics and machine learning functionality on GPUs. From an end-user viewpoint, they generalize existing software offerings in the Python data ecosystem such as Pandas [27] and scikit-learn [22], maintaining the familiar Python-based API but offering accelerated primitives implemented in `C++` and CUDA. On a lower level, RAPIDS' data processing routines are built on the Apache Arrow library, which is an industry standard that implements an efficient in-memory storage format for columnar data.

BlazingSQL builds on RAPIDS as a `C++` library with a Python interface. The library provides a fast, *ad-hoc* query capability for terabyte-scale tabular data using GPUs, without the need for precomputed data structures such as hash tables. It implements out-of-core computation by optionally using unified memory and/or caching to disk (*e.g.*, NVME), and allows distributing the query processing on 10s-100s of GPUs for data that doesn't fit in single GPU memory. BlazingSQL is built on top of the RAPIDS ecosystem [19], and in particular, the RAPIDS `cudf C++` library. It is thus different from dask-sql [1], which is a purely Python-based implementation of SQL with (currently) experimental support for RAPIDS. The library allows users to execute SQL queries against a variety of file formats, including text files with comma-separated values, Apache parquet and Apache ORC files, and in-memory data representations. Figure 1 depicts the architecture of the BlazingSQL software stack.

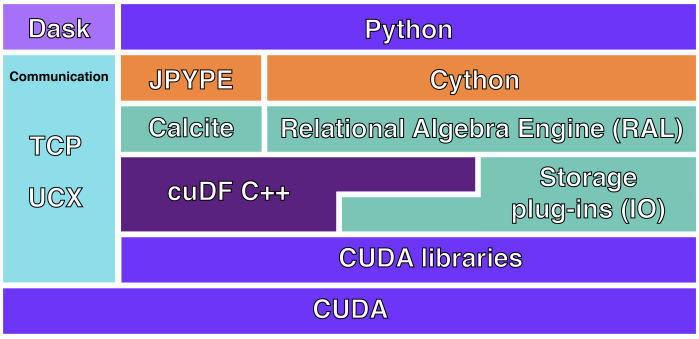

Fig. 1. Overview of the BlazingSQL software stack

The main user interface for BlazingSQL is its Python library `blazingsql`. Through this API, users register tables residing on one of the supported types of filesystems (such as, local or cloud bucket storage) as data sources, execute queries and obtain their results. BlazingSQL returns query results as single-GPU `cudf` DataFrames or distributed, multi-GPU `dask-cudf` DataFrames. When the engine processes an SQL query, it is converted into relational algebra using the

Apache Calcite Java component. The BlazingSQL core relational algebra engine (RAL) then creates a query graph for optimized execution.

The result of the RAL is a directed acyclic graph (DAG), in which the nodes are kernels that take data as input, operate on it, and produce output. The edges are caches that connect producers and consumers of the data. The DAG is uniform across all parallel workers. Every kernel is connected to any other kernel only through a cache. The purpose of the caches is to take the output of a kernel and allow the engine to either leave that information in GPU memory, cache it in CPU, on disk, or on centralized storage. In this way the algorithm allows selective caching of indvidual query nodes and the processing of queries even when the intermediate tables do not simultaneously fit in memory (Fig. 2).

```
SELECT DISTINCT ss_item_sk, ss_ticket_number
    FROM store_sales s, item i
    WHERE s.ss_item_sk = i.i_item_sk
    AND i.i_category_id IN (1, 2, 3)
    AND s.ss_store_sk IN (10, 20, 33, 40, 50)
```

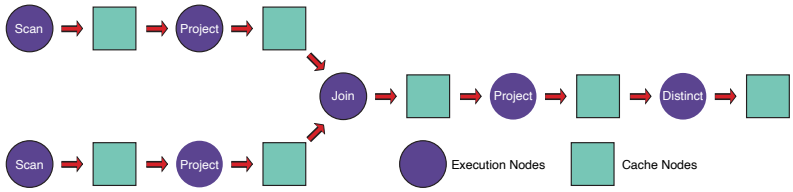

Fig. 2. The first sub-query of the GPU Big Data benchmark (top), and corresponding execution graph (DAG, bottom).

All kernels implement the `Kernel` interface or one of its derived classes. A kernel's purpose is to organize the flow and orchestration of performing complex distributed operations, but it does not perform any of the execution itself. Instead, it creates tasks which are sent to the `TaskExecutor`, which stores a queue of tasks to be completed and is responsible for managing hardware resources and rescheduling tasks that failed due to resource exhaustion. Kernels push information forward into caches and subsequent kernels pull those cached data representations to get inputs. They are decached only right before computation is about to take place.

The data decomposition uses the same coarse-grained partitioning as the distributed `dask-cuda` DataFrame, and the partitions can reside in GPU or CPU memory, on local file systems or on distributed file systems like HDFS. Individual kernels may decompose these partitions further using heuristics.

3 Implementation of communications *via* UCX in BlazingSQL

An important aspect of performing relational algebra operations on distributed datasets is the need to send around messages of varying sizes between peers working on a problem with minimal latency and high bandwidth. BlazingSQL has supported communication *via* TCP sockets since its earliest releases. For high-speed communications, UCX is an open-source, production-grade communication framework for data-centric and high-performance applications. It therefore appears as a natural fit to leverage the native performance of the high-speed Infiniband interconnect. UCX additionally supports accelerated transports such as GPUDirect RDMA and NVLINK. Here we describe an implementation of UCX in a user application that is only based on zero-copy host memory. UCX support is currently available in the `dask.distributed` data analytics framework *via* `ucx-py`. Our implementation, on the other hand, leverages a highly performant, native implementation of UCX on the C/`C++` level.

3.1 False starts for implementing the UCX API in an application code

BlazingSQL uses Python as a user interface and `dask` to initialize the context on every compute node, which acts as an entry point to query execution. We first attempted to take advantage of `ucx-py` and `dask`'s UCX communiacation layer to perform the actual sending and receiving of messages. We defined the context, the workers, and performed all sending and receiving within Python. This first strategy exposed many problems. The GIL (Global Interpreter Lock) in Python prevented messages from being sent using several threads in parallel and the sending and receiving of messages quickly became the bottleneck and performed an order of magnitude worse than our original TCP implementation.

To optimize for latency in sending and receiving messages over UCX, we then kept `ucx-py` for the creation of the context and the workers, and implemented our own Python-based progress routine that needs to be executed to make sure that UCX is moving messages through its queues. Unfortunately this solution remained unstable when used with UCX in multi-threaded mode, and slow in both multi-threaded and single-threaded mode.

Leveraging UCX from `C++` instead of Python therefore suggested itself as the most versatile solution. We moved all code related to UCP workers, endpoints, and message transmission to leverage UCX's C APIs directly. Following the example found in the UCX documentation [28], our first `C++` implementation relied on C-only callbacks that could not receive additional scope variables. The code called `ucp_worker_progress`[3] on every message sent and received, from multiple threads. This approach turned out to be a performance anti-pattern with UCX.

[3] https://openucx.github.io/ucx/api/v1.10/html/group____u__c__p____w__o__r__k__e__r.html

3.2 Final implementation of UCX communications

We created a hierarchy of abstractions that would allow us to alternatively leverage TCP and UCX, and open up the possibility for additional interfaces in the future. We provided for this scenario by separating the sending and receiving of buffers from the actual serialization and deserialization of messages.

All data is placed into page-locked CPU memory before transmission. Even though the very fast GPU to GPU interconnects offer further opportunities for optimization through offloading the communication buffers to GPU memory, it does not make sense to persist data in GPU memory entirely because that memory could rather be used for more performance sensitive applications. We opted to exclusively use GPU memory for the compute portions of the kernels. The greatly reduced GPU memory pressure resulted in satisfactory performance as more memory became available for processing. We implemented the following new `C++` classes and abstractions.

OutputCache This class is a cache that converts all incoming data sources to a structure that is ready for transmission. When data is cached onto CPU, we do so using fixed size chunks that are allocated by an arena allocator, see Fig. 3. If the data is already cached on CPU, adding to the cache does not incur extra costs. The cache chunks are page locked to ensure rapid movement between CPU and GPU. All messages exchanged are therefore of the same size, which is configurable. The default chunk size is 1MB. In preliminary tests, we determined this to be close to the optimum value for Infiniband communication, balancing increased latency for too many small messages with wasted communication bandwidth for too large chunks. Data will stay in the `OutputCache` until there are threads in the sending thread pool.

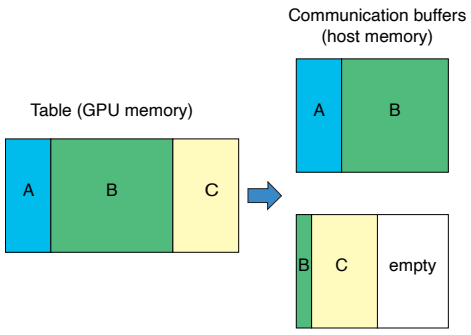

Fig. 3. Serialization of GPU-resident tables in host pinned memory buffers of constant size for communication. The arena allocator provides pinned memory buffers which hold chunks (columns) of the GPU table that is being sent *via* UCX or TCP.

MessageSender The purpose of message sender is to poll the output cache for messages that need to be sent and then to create the appropriate `BufferTransport` class. The message sender is responsible for invoking the functions on the buffer transport.

```
transport->send_begin_transmission();
transport->wait_for_begin_transmission();
for(size_t i = 0; i < raw_buffers.size(); i++) {
    transport->send(raw_buffers[i], buffer_sizes[i]);
}
transport->wait_until_complete();
```

The class waits using a condition variable to allow us to limit the number of messages that are in transmission at any given point in time.

BufferTransport All buffer transports implement the `BufferTransport` interface. The buffer transport is responsible for performing the work of sending buffers and metadata over a specific protocol. In the case of UCX it is the `UCXBufferTransport` that is responsible for calling `ucp_tag_send_nbr`[4] and dispatching buffers that belong to a message. It is not responsible for ensuring that the transmission is complete. We use the `tag` sending API and we split the tag into six bytes for the message id and two bytes which are used to indicate which part of the DataFrame is being transmitted.

MessageListener The message listener is polling using the `ucp_tag_recv_nbr` API and when it receives a new request it is responsible for instantiating the `MessageReceiver` and then polling for any messages that are related to those incoming requests and adding them to the `MessageReceiver`.

MessageReceiver The receiver accepts buffers from the `MessageListener`. The first buffer is the metadata and includes information about the number of buffers that will be received, the routing information of the message, the output location in the DAG, and information that allows us to convert the buffers back into GPU DataFrames when they are needed for further processing.

After all frames have been received, the receiver adds the data to the cache for the next kernel. There are also cases of intra-kernel messaging that usually include only very small payload, where messages might be placed into a more general input cache which is used to send messages that have unique and deterministic identifiers that allow kernels to retrieve them when they expect them. Figure 4 shows a complete example of a kernel sequence with communication.

[4] https://openucx.github.io/ucx/api/v1.10/html/group____u__c__p____c__o__m__m.html

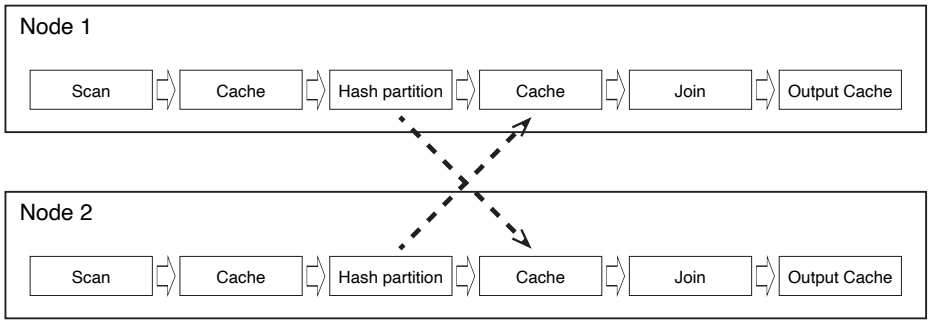

Fig. 4. A kernel is sending a message to the next kernel on another node during a hash join.

4 Performance results

4.1 Performance of UCX *vs.* IPoIB

We use the NVIDIA GPU Big Data benchmark (`gpu-bdb`[5]), an unofficial derivative of the TPC-BB benchmark[6] ported to RAPIDS and BlazingSQL. The `gpu-bdb` benchmark is a RAPIDS library based benchmark for enterprises that includes 30 queries representing real-world extract-transform-load (ETL) and Machine Learning (ML) workflows in the context of retailers with physical and online store presence. Each query is in fact a model workflow that can include SQL, user-defined functions, careful sub-setting and aggregation, and machine learning. Figure 5 shows a natural language description of one of the queries of the benchmark. The benchmark can be run at various scale factors: scale factor 1 000 is 1 TB of data, scale factor 10 000 is 10TB. In this contribution we run only 27 of the 30 queries, because three of the queries (Q22, 27 and 28) did not complete or had unmet dependencies for external libraries on the POWER9 architecture.

```
For a given product get a top 30 list sorted by number of views
in descending order of the last 5 products that are mostly viewed
before the product was purchased online. For the viewed products,
consider only products in certain item categories and viewed within
10 days before the purchase date.
```

Fig. 5. Example of a query (no. 2) from the GPU Big Data benchmark for retailers with a physical and an online store presence.

[5] https://github.com/rapidsai/gpu-bdb
[6] http://tpc.org/tpcx-bb/default5.asp

The standard nodes of Summit have six NVIDIA V100 GPUs with 16GB of DDR5 memory, sharing 512 GB of DDR4 main memory. The high-memory nodes have six NVIDIA V100 GPUs with 32GB of DDR5 memory, and 2TB of DDR4 main memory. We performed all testing on the high-memory partition.

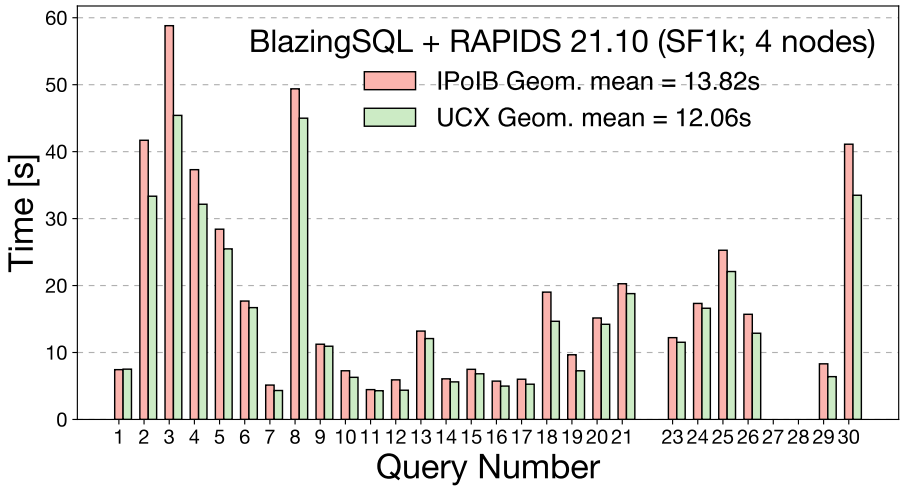

Fig. 6. GPU Big data benchmark at scale factor 1000 (1TB dataset). Shown is the performance for the TCP code path (left/red bars) and the UCX code path (right/green bars). The benchmark was executed on four nodes of Summit (using six 32GB V100 GPUs per node).

We compare the performance of the UCX *vs.* the TCP code path for scale factor 1 000 in Fig. 6, both using the Infiniband interconnect as the low-level transport, using the median runtime of three repeated queries for each `gpu-bdb` query. As can be seen, the geometric average of the time per query is 13.82 s for IPoIB, and 12.06 s for UCX, which amounts to a 15% speed-up resulting from using UCX. However, the improvement is more clearly visible for the longest running queries, which leads us to speculate that the short-running queries involve less communication, and that a larger amount of input data would better expose the benefits of improved communication.

To investigate the query performance for larger data sets and its dependence on the communication protocol, we perform the same comparision for a ten times larger data set (scale factor 10 000). The results are shown in Fig. 7. Of note, the previously slowest query is still an outlier at this scale factor, however now by more than one order of magnitude compared to the geometric mean of all queries. Interestingly, the UCX code path clearly outperforms the TCP code path and delivers an average speed-up of 75%. Communication *via* UCX is therefore consistently superior to IPoIB for all queries. This result demonstrates

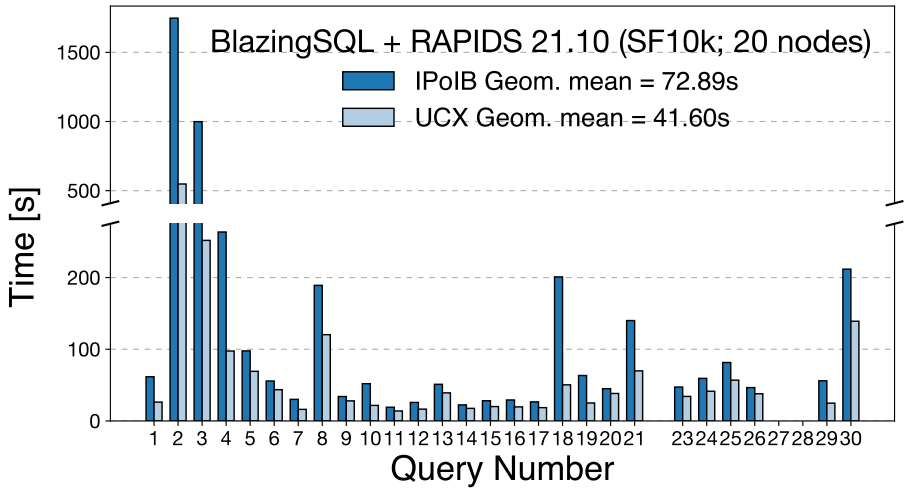

Fig. 7. GPU Big data benchmark at scale factor 10 000 (10TB dataset). Shown is the performance for the TCP code path (left/dark shaded bars) and the UCX code path (right/light shaded bars). The benchmark was executed on 20 nodes of Summit (using six 32GB V100 GPUs per node).

that our optimizations were successful, and that communication indeed becomes the limiting factor for the performance of distributed queries on large data sets.

4.2 Multi-node performance

To investigate the balance between compute- and communication-intensive parts of a query, we perform a strong scaling test at scale factor 10 000. Figure 8 shows the geometric mean for 27 queries as a function of the number of nodes. The performance is already optimal for the smallest node count that allows for successful benchmark completion, *i.e.*, 18 high-memory nodes of Summit. Adding more nodes degrades performance. The absence of strong scaling at this data set size indicates that the queries are strongly memory-bound, and that the available computational work likely fails to saturate the GPU. We believe this to be a general characteristic of GPU-based data analytics. We did not analyze weak scaling efficiency.

5 Implications for future and emerging HPC platforms

With the availability of high-speed interconnects and accelerated transports supporting heterogeneous hardware such as GPUs comes the mandate of leveraging their superior performance in applications. However, the traditional API for fast communication on HPC platforms is the message passing interface (MPI),

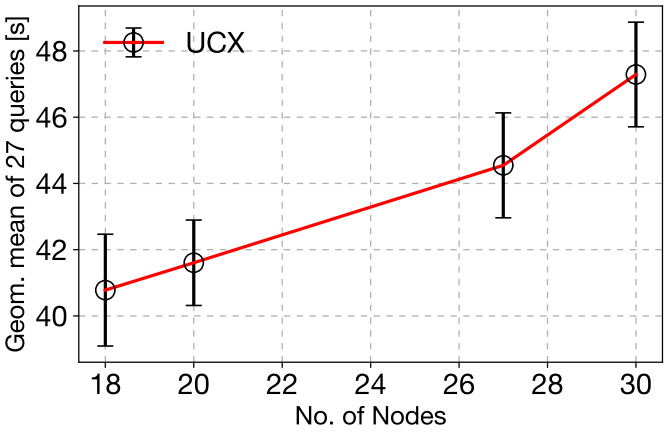

Fig. 8. Geometric mean of the query time of the GPU Big data benchmark at scale factor 10 000 (10TB dataset) on different numbers $N = 18, 20, 27, 30$ of nodes (1 node = 6×32GB V100) on the Summit supercomputer at Oak Ridge National Laboratory, using UCX. The leftmost data point ($N = 18$) corresponds to the smallest number of nodes for which the benchmark completes successfully.

which is particularly suited for synchronous and regular problems involving only numerical data. The MPI stack on the Summit supercomputer uses the Parallel Active Messaging Interface (PAMI) for underlying accelerated transports, but also supports an experimental UCX option. Conversely, many data analytics methods have been developed outside the realm of traditional high-performance computing, and the focus has been on massively parallel compute with little or no communication [8, 18]. Therefore, reconciling the world of HPC and data analytics requires communication middleware that combines aspects of commercial big data processing, such as fault-tolerance, with the reproducible performance of compute- and communication intensive HPC workloads. In our case, we found that UCX combines the best of both worlds: it is considerably more lightweight than a full MPI implementation, yet it includes hardware and system-specific backends for most (if not all) industry-standard interconnects. Nevertheless, the performance benefit and portability is offset by the higher implementation effort needed to program UCX compared to MPI or OpenSHMEM [6], due to the lower-level API.

With AI and deep learning entering the forefront of computational sciences, and with high-throughput workflows being executed on HPC platforms, we see an ever-increasing need for multi-node, multi-GPU database query processing. Data-centric workflows serve to organize, manipulate, search and combine massive datasets, which may have been produced by simulation or experiment, e.g., for preparing training data or processing results from large-scale inference. Moreover, workflows that can be expressed in high-productivity languages such as Python are composable, and the BlazingSQL library fills in a gap for high-performance

database query processing. As database query processing becomes more commonplace in HPC, we also see a trend towards interactive use of HPC resources through Jupyter notebooks.

High-performance file systems at HPC facilities are a necessary prerequisite. However, since BlazingSQL supports both bucket storage and traditional POSIX file systems, it is compatible with future developments in HPC data center technology. We believe that gradual convergence between cloud computing and traditional HPC through the deployment of more service-oriented architectures at supercomputing facilities further enables processing of big data alongside traditional compute workloads.

6 Conclusion

We discussed the implementation of a new UCX-based communication layer in the open-source, GPU-accelerated database query engine BlazingSQL. The promise of fast communication over Infiniband lies in accelerating distributed queries that execute in seconds on tens or hundreds of GPUs on a supercomputer to interactively query terabyte-scale datasets. We have shown that a performance-optimal implementation should make use of the low-level, `C++` interface to the UCX library. We benchmarked BlazingSQL performance with UCX against the baseline implementation with IPoIB (TCP) on multiple nodes, and demonstrated superior performance of the UCX code path. In summary, we have shown how to incorporate UCX communication middleware into a library with a high-level user API. The improved communication layer brings about significance performance improvements for distributed database query processing, particularly of large data sets. In addition to performance improvements on the POWER9 platform, the library is portable and runs equally well on x86 architecture. The speed-ups should therefore also manifest themselves on NVIDIA DGX systems, and on workstations with multiple NVIDIA GPUs.

Acknowledgments

We are grateful to Oscar Hernandez (NVIDIA) for initial conceptualization of this research. We thank Arjun Shankar (ORNL) for support. This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. dask-sql. https://github.com/dask-contrib/dask-sql (2021), accessed: 2021-11-5
2. Bakkum, P., Skadron, K.: Accelerating sql database operations on a gpu with cuda. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. p. 94–103. GPGPU-3, Association for Computing Machinery, New York, NY, USA (2010), https://doi.org/10.1145/1735688.1735706

3. BlazingSQL: High Performance SQL Engine on RAPIDS AI. https://blazingsql.com/ (2021), accessed: 2021-10-08
4. Breß, S., Saake, G.: Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. Proc. VLDB Endow. 6(12), 1398–1403 (Aug 2013), https://doi.org/10.14778/2536274.2536325
5. Bre„, S., Beier, F., Rauhe, H., Sattler, K.U., Schallehn, E., Saake, G.: Efficient co-processor utilization in database query processing. Information Systems 38(8), 1084–1096 (2013), https://www.sciencedirect.com/science/article/pii/S0306437913000732
6. Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. pp. 1–3 (2010)
7. Chrysogelos, P., Sioulas, P., Ailamaki, A.: Hardware-conscious query processing in gpu-accelerated analytical engines. In: Proceesings of the 9th Biennial Conference on Innovative Data Systems Research. No. CONF (2019)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
9. DeWitt, D., Gray, J.: Parallel database systems: The future of high performance database systems. Commun. ACM 35(6), 85–98 (Jun 1992), https://doi.org/10.1145/129888.129894
10. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N.K., Luo, Q., Sander, P.V.: Gpuqp: Query co-processing using graphics processors. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. p. 1061–1063. SIGMOD '07, Association for Computing Machinery, New York, NY, USA (2007), https://doi.org/10.1145/1247480.1247606
11. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. Proc. VLDB Endow. 3(1–2), 670–680 (Sep 2010), https://doi.org/10.14778/1920841.1920927
12. Glaser, J., Vermaas, J.V., Rogers, D.M., Larkin, J., LeGrand, S., Boehm, S., Baker, M.B., Scheinberg, A., Tillack, A.F., Thavappiragasam, M., et al.: High-throughput virtual laboratory for drug discovery using massive datasets. The International Journal of High Performance Computing Applications p. 10943420211001565 (2021)
13. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M., Manocha, D.: Fast computation of database operations using graphics processors. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. p. 215–226. SIGMOD '04, Association for Computing Machinery, New York, NY, USA (2004), https://doi.org/10.1145/1007568.1007594
14. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. 34(4) (Dec 2009), https://doi.org/10.1145/1620585.1620588
15. Hernández, B., Somnath, S., Yin, J., Lu, H., Eaton, J., Entschev, P., Kirkham, J., Ronaghi, Z.: Performance evaluation of python based data analytics frameworks in summit: Early experiences. In: Nichols, J., Verastegui, B., Maccabe, A.B., Hernandez, O., Parete-Koon, S., Ahearn, T. (eds.) Driving Scientific and Engineering Discoveries Through the Convergence of HPC, Big Data and AI. pp. 366–380. Springer International Publishing, Cham (2020)
16. Huebl, A.: Openpmd release 1.4.0 with support for data processing through dask. https://github.com/openPMD/openPMD-api/releases/tag/0.14.0 (2021)
17. Lee, S., Park, S.: Performance analysis of big data etl process over cpu-gpu heterogeneous architectures. In: 2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW). pp. 42–47 (2021)

18. Lu, X., Islam, N.S., Wasi-Ur-Rahman, M., Jose, J., Subramoni, H., Wang, H., Panda, D.K.: High-performance design of hadoop rpc with rdma over infiniband. In: 2013 42nd International Conference on Parallel Processing. pp. 641–650 (2013)
19. NVIDIA: Open GPU Data Science-RAPIDS. https://rapids.ai (2021), accessed: 2021-05-26
20. Olsen, S., Romoser, B., Zong, Z.: Sqlphi: A sql-based database engine for intel xeon phi coprocessors. In: Proceedings of the 2014 International Conference on Big Data Science and Computing. BigDataScience '14, Association for Computing Machinery, New York, NY, USA (2014), https://doi.org/10.1145/2640087.2644172
21. OmniSciDB: OmniSciDB: open source SQL-based, relational, columnar database engine. https://github.com/omnisci/omniscidb (2021), accessed: 2021-05-26
22. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, 2825–2830 (2011)
23. PGStrom: PG-Strom: a GPU extension module of PostgreSQL. https://github.com/heterodb/pg-strom (2021), accessed: 2021-05-26
24. Poeschel, F., Godoy, W.F., Podhorszki, N., Klasky, S., Eisenhauer, G., Davis, P.E., Wan, L., Gainaru, A., Gu, J., Koller, F., et al.: Transitioning from file-based hpc workflows to streaming data pipelines with openpmd and adios2. arXiv preprint arXiv:2107.06108 (2021)
25. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., et al.: Ucx: an open source framework for hpc network apis and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. pp. 40–43. IEEE (2015)
26. Shehab, E., Algergawy, A., Sarhan, A.: Accelerating relational database operations using both cpu and gpu co-processor. Computers & Electrical Engineering 57, 69–80 (2017), https://www.sciencedirect.com/science/article/pii/S0045790616310631
27. pandas development team, T.: pandas-dev/pandas: Pandas (Feb 2020), https://doi.org/10.5281/zenodo.3509134
28. UCX: UCX Client-Server. https://openucx.github.io/ucx/api/v1.10/html/ucp_client_server_8c-example.html (2021), accessed: 2021-05-26
29. Weininger, D.: Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. Journal of chemical information and computer sciences 28(1), 31–36 (1988)
30. Woods, L., István, Z., Alonso, G.: Ibex: An intelligent storage engine with support for advanced sql offloading. Proc. VLDB Endow. 7(11), 963–974 (Jul 2014), https://doi.org/10.14778/2732967.2732972