Exploring On-Path Processing for Efficient Data-Intensive Workloads on GPU Clusters

Alireza Shateri
alireza@cs.toronto.edu
University of Toronto
Toronto, Ontario, Canada

Christian Tabbah
tabbahch@cs.toronto.edu
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

The growing adoption of GPU-accelerated OLAP systems has sparked a surge of interest in using GPUs for large-scale data analytics. These systems leverage the highly parallel nature of GPUs to accelerate scan-heavy, aggregation-focused queries, which are common in OLAP workloads. In many such systems, data is partitioned and distributed across multiple GPUs, either due to upstream GPU-based pipelines or GPU-aware data ingestion techniques. While GPUs are well-suited for local computation, certain operations, such as with a focus on DISTINCT, a representative operation that requires global aggregation. In this context, we explore the potential of on-path processing, a technique where intermediate CPU hosts perform computation as data travels from source GPUs through intermediate CPU stages to a final destination GPU. While offloading computation from a GPU to a CPU is typically unjustified due to the GPU's superior performance, on-path processing becomes attractive when data from multiple GPUs must be merged before meaningful computation can occur. Performing partial aggregation or duplicate elimination on the CPU, where the data converges, can reduce network traffic and improve end-to-end query performance. This paper investigates how on-path processing can be strategically integrated into GPU-based OLAP pipelines, identifies scenarios where it provides significant benefit, and presents a system architecture that integrates on-path processing into GPU-based pipelines, reducing network traffic during globally scoped operations.

1 INTRODUCTION

Modern data-driven applications—from large-scale analytics platforms to complex machine learning pipelines—must handle ever-growing volumes of data. To meet the computational demands of such workloads, GPU clusters have become a de facto standard due to their massive parallelism and high throughput[2, 4, 10, 16, 20]. In particular, GPU-accelerated OLAP (Online Analytical Processing) systems have gained traction as they align well with the columnar, scan-heavy nature of analytical queries. With growing support in systems such as RAPIDS, BlazingSQL, Spark+RAPIDS, and GOLAP [3, 5, 6, 15] GPU-based analytics is rapidly evolving into a mainstream architectural choice.

In these systems, data is often distributed across multiple GPUs, either as a result of upstream GPU-accelerated ETL pipelines or through GPU-aware ingestion tools that load data directly into GPU memory. This distributed layout naturally arises in modern clusters and enables parallelism, but it also introduces significant challenges when performing global operations—such as GROUP BY, DISTINCT, or JOIN—that require a unified view of the data. In such cases, raw data from multiple GPUs must be transmitted across the network and combined at a final destination GPU before the query can be resolved. This communication step can become a major performance bottleneck, especially when large intermediate results must be shuffled across nodes.[7, 12, 18, 21–23]

To address this inefficiency, we explore on-path processing: a design pattern where intermediate CPU hosts—those already in the data transmission path—perform lightweight computation on data as it travels from source GPUs to the destination. We believe this introduces a novel oportunity to optimize OLAP workflows. [1, 9, 11, 14, 17, 22] These hosts can execute operations such as filtering, partial aggregation, or duplicate elimination to reduce data volume in-flight. Crucially, we do not aim to offload computation from GPUs to CPUs, which would generally be counterproductive given the GPU's superior compute capabilities. Instead, we leverage the converging nature of data in many distributed OLAP workloads—where intermediate hosts naturally receive input from multiple sources—to perform early combination steps that can significantly reduce network overhead.

For example, in operations like DISTINCT, that require a global view of the dataset, sending all raw data to the destination GPU may be wasteful. By merging or partially aggregating data at intermediate CPUs—before the full shuffle completes—on-path processing can reduce overall data movement and improve query performance.

This paper makes the following contributions:

- We introduce the on-path processing model for distributed GPU-accelerated OLAP workloads, motivated by the need to reduce network-bound inefficiencies in global data aggregation.
- We identify query operations, such as DISTINCT, that benefit significantly from intermediate aggregation or transformation.
- We design and implement a prototype system with multiple worker nodes that leverage on-path processing to reduce communication overhead in global operations
- We evaluate our system across synthetic OLAP-style queries and demonstrate that on-path processing can meaningfully reduce network traffic and lower end-to-end query latency.

Through this work, we aim to bridge ideas from in-transit processing in HPC and in-network computing with modern GPU-based analytics, providing a new mechanism to improve performance in data-intensive distributed systems.

2 BACKGROUND

2.1 Distributed GPU Computing

GPUs are highly specialized processors designed for massively parallel computation, making them well-suited for workloads that involve high arithmetic intensity or large-scale data processing. In distributed systems, GPU computing extends beyond a single machine to clusters where multiple GPUs collaborate on large tasks. In a distributed GPU cluster, a central scheduler or control node partitions the workload, assigns tasks to different GPUs, and oversees inter-GPU communication. [3, 8, 13, 16, 19] This setup enables horizontal scalability, allowing systems to process large datasets by leveraging the combined memory and compute capabilities of multiple GPUs.

Distributed GPU computing has proven particularly effective for OLAP (Online Analytical Processing) workloads. These workloads are characterized by large-scale scans across columnar data, aggregation-heavy queries and repetitive computations over massive data partitions. GPU architectures are well-matched to this query profile due to their ability to execute the same operations over large vectors of data in parallel.

In recent years, several distributed OLAP systems have emerged that exploit GPU acceleration to significantly improve query throughput and latency:

GOLAP (GPU-in-Data-Path Architecture for High-Speed OLAP):

GOLAP introduces a novel architecture where GPUs are integrated directly into the data pipeline of an SSD-based analytical database. GOLAP is specifically designed for OLAP workloads and tackles a key performance bottleneck of SSD-backed systems: limited I/O bandwidth. Instead of routing data through the CPU, GOLAP streams heavily compressed columnar data blocks directly from SSDs to GPUs using GPU Direct Storage and decompresses them on-the-fly within the GPU. This compressed scan operator is further enhanced by GPU-optimized pruning techniques, which skip irrelevant data before transfer, and co-execution strategies that balance workload between the GPU and CPU when intermediate results exceed device memory limits. By combining direct I/O, decompression, and query execution on the GPU, GOLAP achieves bandwidths up to 100 GiB/s—competitive with in-memory systems—while maintaining the cost efficiency of SSD-based storage.

RAPIDS and Related Frameworks:

A broader trend in the GPU analytics ecosystem is the rise of general-purpose frameworks that support OLAP-style processing on distributed GPUs. Notably, NVIDIA's RAPIDS library enables in-GPU DataFrame operations and integrates with distributed execution engines such as Dask and Spark to scale workloads across GPU clusters. Other systems such as BlazingSQL and OmniSci similarly leverage GPU acceleration for SQL-like analytics and visualization. While these systems offer powerful abstractions for GPU-based computation, they often follow conventional shuffle-based architectures for global operations, where raw data is transferred across nodes without early combination. As we explore in this work, such architectures may miss opportunities to reduce network overhead through intermediate, on-path computation.

These frameworks reflect a growing trend: OLAP queries are increasingly being executed directly on distributed GPU memory, without staging through host memory or post-hoc data transfer. As a result, data may often reside on GPUs at the time of query execution, with each GPU responsible for a partitioned subset of the dataset. This distributed GPU execution model introduces new challenges, especially for operations that require a global view of the data—such as joins, distinct projections, or group-by aggregations—where intermediate results must be combined across GPUs. Addressing the communication and coordination costs of such operations is critical to improving performance in modern GPU-accelerated OLAP systems.

2.2 Typical Data Flow and Bottlenecks

In distributed GPU-accelerated OLAP systems, operations that require a global view of data—such as GROUP BY, SUM, or DISTINCT—must aggregate data across GPUs located on separate nodes. These global operations typically follow a common data movement pattern that spans both compute and communication subsystems.

The standard data flow for such operations involves four primary steps:

- **GPU to Host Copy:** Each GPU copies the relevant portion of its dataset to its local CPU host's memory. This is typically done using a `cudaMemcpy` or equivalent GPU-to-host transfer.
- **Network Transfer:** The host CPUs then send their data across the cluster interconnect (e.g., Ethernet, Infiniband, or RoCE) to the CPU host of the designated destination GPU node.
- **Host to GPU Copy:** The destination node receives the data in its host memory and transfers it to the local GPU memory, staging it for computation. Advanced setups may bypass host memory using technologies like **GPUDirect RDMA**, allowing GPUs to directly read/write over the network, or use **NVLink** for faster intra-node GPU-GPU transfers.
- **Final Computation:** Once all partitions have arrived, the destination GPU executes the desired global operation—such as computing an aggregate or performing a join.

This architecture reflects the design of many GPU-accelerated analytics frameworks, where GPUs serve as stateless computation engines, and data movement is handled by the host CPUs and network stack. However, as dataset sizes grow and cluster parallelism increases, this workflow can become network-bound. The volume of data transferred to the final GPU node increases with the number of source GPUs, leading to:

- Network congestion when multiple sources send large payloads simultaneously
- Increased end-to-end latency due to data serialization and memory staging
- Underutilized GPU resources, as destination GPUs may wait for data while upstream nodes saturate the interconnect

These inefficiencies are exacerbated when the final computation does not require the full fidelity of the original dataset. For example, if the global operation is a DISTINCT or GROUP BY, many records may be redundant or mergeable—yet the system transmits them in full before reducing them. This observation motivates the use of on-path processing, where intermediate hosts perform lightweight computation—such as filtering or aggregation—during the data transfer phase to reduce overall communication overhead. We explore this model in the next section.

2.3 On-Path Processing

On-path processing introduces an alternative to the traditional all-to-one data transfer model used in distributed GPU analytics. Instead of forwarding raw data directly from source GPUs to the destination GPU, intermediate nodes—typically the CPU hosts as well as specialized NICs participating in the data transfer—perform lightweight computations as data passes through them. This processing happens *in transit*, before data reaches the destination GPU.

Common examples of on-path processing include:

(1) **Filtering:** Removing irrelevant records or columns to reduce payload size.
(2) **Pre-Aggregation:** Computing partial results (e.g., partial sums, counts, or distinct keys) that can later be merged.
(3) **Compression:** Reducing data size through encoding or transformation.
(4) **Domain-Specific Transformations:** Performing contextual processing like feature extraction or tagging.

By the time the destination GPU receives the data, the payload is smaller and more refined. This reduces both the volume of inter-node communication and the overhead of final-stage GPU computation. Moreover, on-path processing takes advantage of otherwise idle CPU resources, enabling a more balanced use of the CPU-GPU pipeline. While CPUs are generally less powerful than GPUs for raw computation, they can be effectively utilized to perform selective, compute-light transformations.

On-path processing is particularly beneficial for OLAP workloads that involve global operations like `GROUP BY`, `DISTINCT`, or filtering with high selectivity. These operations often exhibit significant redundancy or compressibility across partitions, making them ideal candidates for early combination. Integrating on-path computation into distributed GPU pipelines offers a promising path to reducing network bottlenecks, improving throughput, and enabling adaptive data flows in modern analytics systems.

3 DESIGN

Our system implements a distributed execution strategy for the `DISTINCT` operator in OLAP workloads. Initially, we considered implementing a basic aggregation workload, such as computing the sum of values across datasets. However, this approach proved suboptimal for exploring on-path CPU computation. In most realistic scenarios, such aggregation is more efficiently executed entirely on the GPU, where data initially resides, due to the GPU's high parallelism and memory bandwidth. Offloading aggregation to the CPU only makes sense in rare cases—such as when the GPU is fully occupied with other high-priority tasks, when GPU memory is constrained, or when the operation involves irregular control flow that CPUs handle more efficiently. In contrast, operators like `GROUP BY` and `DISTINCT` inherently require a global view of the dataset, making it necessary to consolidate and deduplicate data across GPUs and hosts. This requirement creates a natural point for CPU intervention and data movement, allowing us to explore the benefits of partial computation and deduplication on the CPU prior to RDMA transmission. We choose the `DISTINCT` operator due to its flexibility in shaping workload characteristics: by tuning the randomness of generated input data, we can control the rate of duplicate values, which in turn affects the volume of data transferred across the network. This property makes `DISTINCT` a suitable proxy for evaluating the network and computation trade-offs of on-path processing, reducing the need to separately evaluate more complex operators like `GROUP BY`. In traditional distributed query engines, `DISTINCT` operations can incur substantial network costs, as intermediate tuples from multiple workers must be shuffled and globally deduplicated. To address this, our system leverages both CPU and GPU resources to perform early-stage deduplication and reduce communication overhead by integrating on-path computation into the query execution pipeline.

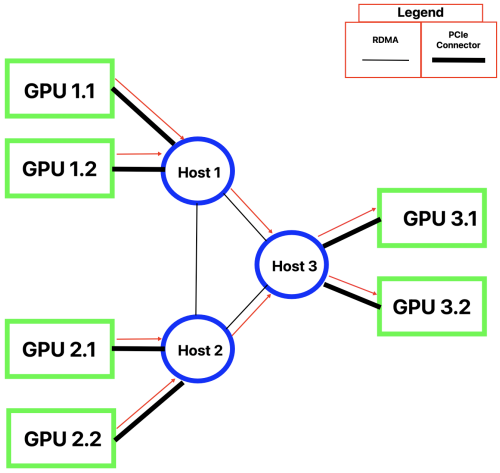


Figure 1: System topology. The setup consists of three RDMA-connected hosts, each equipped with two GPUs. Hosts 1 and 2 evaluate a DISTINCT operator over local partitions and forward partially reduced results to Host 3, which finalizes the DISTINCT computation globally.

Our architecture consists of three RDMA-connected hosts. Hosts 1 and 2 each contain two GPUs and are responsible for executing local fragments of the `DISTINCT` query. Host 3 acts as the final aggregation node, collecting partially deduplicated results, sending it to the final destination GPU to complete the `DISTINCT` computation globally. This topology is illustrated in Figure 1.Furthermore, we assume that data originates in GPU memory on Hosts 1 and 2, having already been partitioned in a previous stage of an OLAP

pipeline. Similarly, we assume that the result of the DISTINCT operator is consumed by a downstream query stage that expects the final output to reside on the GPU of Host 3. This lets us focus our design on optimizing the transfer and processing stages between these endpoints.

3.1 Local DISTINCT Execution and CPU-Side Merge

On Hosts 1 and 2, input data is divided between two local GPUs. In a full implementation, each GPU would scan its partition and evaluate the DISTINCT operator in parallel, eliminating intra-partition duplicates. However, for simplicity and benchmarking purposes, we generate randomly distributed but unique data within each GPU. Since both the baseline and our on-path variant must process this data identically, the simplification does not affect comparative evaluation.

Each GPU's output is copied back to host CPU memory, where a CPU-side merge performs a second DISTINCT pass to eliminate duplicates across the two GPU partitions. This intra-host deduplication ensures that only unique tuples across both GPUs are staged for transmission to Host 3, reducing network traffic and avoiding redundant RDMA writes.

Once a configurable threshold of new distinct values has been collected, the host CPU stages a chunk of the data into a send buffer and transmits it to Host 3 using RDMA. This communication is handled asynchronously, effectively pipelining data transfer with CPU computation to maximize overlap between processing and network I/O.

3.2 Final DISTINCT Aggregation on Host 3

Host 3 runs a UCX-based RDMA server that listens for incoming data chunks from Hosts 1 and 2. Each client writes its results directly into a dedicated region of Host 3's pre-registered RDMA buffer, avoiding unnecessary copies and minimizing network latency.

Although our initial plan was to perform the final DISTINCT computation on the CPU, we ultimately moved this computation to the GPU on Host 3. While the CPU had been responsible for early deduplication stages in the On-Path design, we found that the overhead of performing the final aggregation on the CPU—especially in terms of locking and hash table contention—outweighed the relatively small cost of transferring the data over PCIe to the GPU. Given that GPUs are significantly better suited for parallel operations over large datasets, this final deduplication step is efficiently handled via a sort and unique sequence using CUB primitives.

The merged data is copied into GPU memory and sorted using `cub::DeviceRadixSort`, after which `cub::DeviceSelect::Unique` is used to eliminate duplicates. This ensures global correctness of the DISTINCT operator while leveraging GPU compute capabilities to avoid further bottlenecks.

Since DISTINCT is associative and idempotent, the correctness of the result is preserved across this multi-stage pipeline. This GPU-based final aggregation complements the earlier On-Path CPU deduplication, allowing us to balance network reduction and computational efficiency while preserving end-to-end correctness.

4 IMPLEMENTATION

Our implementation builds on the design described in Section 3, integrating pipelined RDMA communication, GPU data transfers, and multi-threaded CPU coordination to execute the DISTINCT operator efficiently across distributed hosts. The system is over 2000 lines of code in C++ and CUDA.

4.1 RDMA Abstraction via UCX

Our system is built on top of the Unified Communication X (UCX) framework, which provides a flexible and high-performance communication layer for RDMA and other transport protocols. One of the key advantages of UCX is that it abstracts away low-level transport selection. Depending on hardware availability and message size, UCX dynamically chooses the most efficient transport — such as RDMA (e.g., InfiniBand or RoCE) for large messages on RDMA-capable networks, or shared memory and TCP for smaller transfers or fallback paths.

This design allows our system to remain portable across clusters with different hardware capabilities, while still leveraging zero-copy RDMA transfers when available. Additionally, UCX provides asynchronous, non-blocking APIs and in-order delivery guarantees, which are critical for our pipelined architecture. These features simplify the development of efficient data movement mechanisms and ensure correctness without introducing complex synchronization protocols.

4.2 Concurrent Hashmap for In-Host Deduplication

On Hosts 1 and 2, each GPU generates data that is copied to host memory and passed through a CPU-based deduplication filter. To do this efficiently, we adapted a concurrent hashmap implementation from an open-source repository[1]. The original design used a vector of linked lists to represent buckets. However, since our workload involves significantly more lookups than insertions, we modified each bucket to use a std::map, which provides O(logn) lookup time due to its underlying balanced binary search tree.

We use this map as a "seen set" to track which values have already been observed. If a value is new, it is inserted and staged for sending; otherwise, it is discarded. This trade-off significantly improves performance in scenarios with high duplication ratios, which is common in large OLAP workloads.

4.3 Threshold-Based Pipelined Sender Thread

Each host uses a dedicated sender thread to manage outgoing data chunks. Values that pass the deduplication filter are staged into a

[1] https://github.com/diffstorm/ConcurrentHashMap

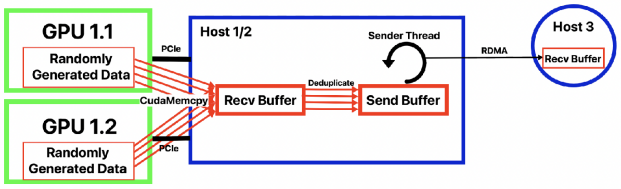


Figure 2: Host-side DISTINCT pipeline. The figure shows the design of Host 1, where GPU-local DISTINCT results are merged on the CPU before being sent via RDMA to Host 3. Host 2 follows the same architecture.

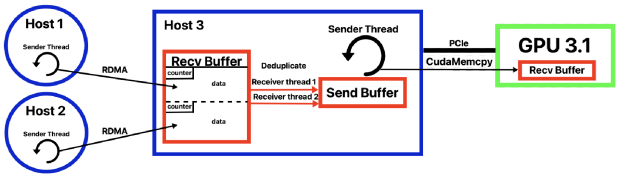


Figure 3: Server-side DISTINCT pipeline. The figure shows the design of Host 3, where the receive buffer is split into two parts, both beging deduplicated through the receiver threads, and then sent off to the GPU using a sender thread and CudaMemCpy.

CPU-side send buffer. The sender thread continuously monitors this buffer and, once a configurable threshold is reached, triggers an RDMA transfer.

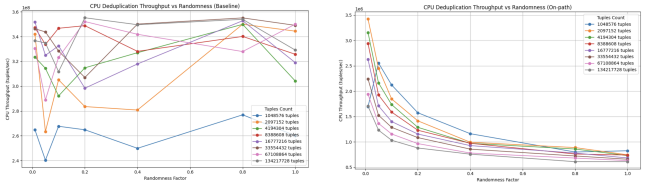
This design introduces a form of pipelining: while the GPU continues generating data and the CPU continues deduplicating, the sender thread overlaps communication by sending previously buffered data. This decoupling of processing and transmission increases throughput and minimizes idle time on both compute and network layers. This can be seen in Figure 2.

4.4 Queue-Based RDMA Transmission

Instead of sending data synchronously, the *send_chunk* function appends RDMA write requests to a queue. A separate internal thread (managed by the UCX client) pulls from this queue and performs the necessary *ucp_worker_progress* operations to complete the RDMA write. This architecture further enhances pipelining. GPU and CPU threads are never blocked on network communication, and RDMA transfers are handled transparently in the background.

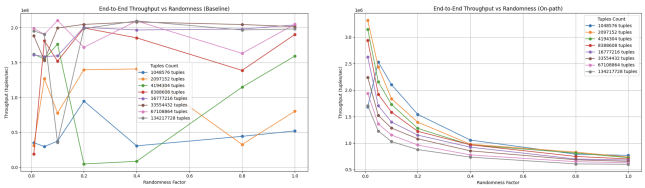
4.5 Buffered CUDA Transfers to Destination GPU

On Host 3, after the final deduplication, we perform a cudaMemcpy to copy unique values to the destination GPU buffer. This step mirrors the CPU-side pipelining described earlier: a send buffer accumulates globally unique values, and a dedicated thread flushes this buffer to GPU memory in bulk. This avoids frequent small transfers, reducing PCIe overhead and increasing overall bandwidth efficiency.



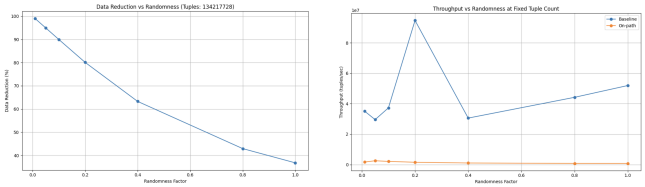
(a) CPU Throughput - Baseline **(b) CPU Throughput - On-path**

Figure 4: CPU throughput comparison of Baseline vs On-path



(a) End-to-End Baseline Throughput **(b) End-to-End On-Path Throughput**

Figure 5: Comparison of End-to-End Throughput between Baseline and On-Path configurations.



(a) Randomness Factor vs Data Reduction **(b) On-Path vs Baseline End-to-End Throughput**

4.6 RDMA Buffer Synchronization and Receiver Logic

To coordinate communication between Hosts 1/2 and Host 3, we partition the RDMA buffer into two sections—one for each client. Each sender writes only to its assigned region, and each receiver thread on Host 3 monitors its region for changes. This can be seen in Figure 3.

Synchronization is achieved using an integer counter embedded at the start of each RDMA buffer region. Clients increment this counter after each chunk write, and set it to −1 when all data has been sent. The receiver thread polls this counter; if it differs from the previously observed value, it triggers a deduplication pass on the new data.

This lightweight mechanism ensures timely processing without complex signaling or locks. Thanks to UCX's in-order delivery guarantees, we can reliably interpret the end-of-stream when the counter becomes −1.

5 EVALUATION

We evaluate the performance of our system by comparing On-Path processing with a Baseline design that skips deduplication entirely. Our experiments vary both the tuple count and the randomness factor, which controls the proportion of unique values in the dataset. The key performance metrics include CPU deduplication throughput and overall end-to-end system throughput.

5.1 CPU Throughput Analysis

Figure 4a and Figure 4b show the CPU throughput for the Baseline and On-Path systems respectively. As expected, the Baseline exhibits significantly higher throughput—on the order of 10^2 higher, compared to the On-Path design. This result aligns with expectations: in the Baseline, the CPU performs no computation and acts merely as a pass-through, allowing data to move quickly through the pipeline.

In contrast, On-Path processing performs deduplication on the CPU before sending data to the destination GPU. This added computation introduces noticeable overhead. Furthermore, as the randomness factor increases (i.e., the number of unique values increases), the CPU throughput in the On-Path system begins to decline. This trend is particularly visible in Figure 4b, where the throughput plateaus or degrades at higher randomness levels. The underlying reason is that more unique values result in more insertions into the staging buffer, which is protected by a lock in our implementation. This lock-based synchronization becomes a bottleneck at scale, further highlighting a limitation of the current design.

5.2 End-to-End Throughput Comparison

Figure 5a and Figure 5b present the total end-to-end throughput for the Baseline and On-Path systems, while Figure 6b directly compares both systems at a fixed tuple count. Although On-Path processing significantly reduces the volume of data transferred over the network, the overall system performance does not improve—in fact, it degrades.

This degradation stems from the fact that CPU-side deduplication becomes the dominant bottleneck in the On-Path design. Despite achieving its goal of reducing communication overhead, the added latency from deduplication outweighs the benefit of reduced data transmission. We believe this outcome is largely due to contention on the shared staging buffer used for transferring deduplicated data to the GPU. Our current implementation uses a single lock-protected buffer, which becomes increasingly inefficient under high load. In future work, we plan to redesign this stage by introducing per-thread send buffers, allowing for lock-free or lock-reduced parallelism to alleviate this bottleneck.

5.3 Randomness Control and Data Reduction

Lastly, Figure 6a illustrates the relationship between the randomness factor and the percentage of unique values in the input. This plot demonstrates that our system can effectively control the uniqueness of input data by tuning the randomness parameter. This ability allows us to simulate different degrees of data redundancy and better evaluate how the system behaves under varying deduplication loads. As the randomness increases, so does the proportion of unique values in the data, which directly impacts both CPU throughput and overall system behavior.

6 CONCLUSION

In this paper, we presented an On-Path processing architecture for distributed GPU-accelerated OLAP systems. Our system aimed to reduce network overhead by performing deduplication on intermediate CPU hosts before transferring data to the final GPU for query resolution. While our experiments showed that On-Path processing successfully reduced data movement, the system's end-to-end performance was ultimately bottlenecked by CPU-side deduplication. Despite not achieving the throughput gains we initially hoped for, the architecture highlights important trade-offs in placing computation along the data path in distributed GPU environments. These findings point to both the challenges and opportunities in designing hybrid CPU-GPU pipelines for large-scale analytics.

7 FUTURE WORK

Due to time constraints—including the difficulty of reserving GPU-equipped machines and the long runtime of distributed tests—we were unable to fully explore the design space of our system. As a result, we did not evaluate the effects of key tunable parameters such as the staging threshold, the number of threads involved in deduplication and data transmission, or the configuration of our concurrent hash table (e.g., number of buckets). These design choices likely played a significant role in the system's observed bottlenecks, and adjusting them may yield very different results.

Although our current implementation did not demonstrate the full potential of On-Path processing, we remain confident in the underlying idea. Prior work, as well as our own microbenchmarks and observations, strongly suggest that reducing data movement can yield meaningful performance benefits when paired with a well-optimized CPU-side pipeline. With more time to iterate, we believe that On-Path processing can provide measurable gains in both throughput and latency.

In future work, we plan to redesign the staging mechanism by introducing per-thread buffers or lock-free data structures to alleviate contention from locking. Once the system is optimized, we aim to extend it into a dynamic architecture that can choose at runtime whether to perform On-Path processing based on workload characteristics, GPU availability, and network load. Ultimately, we also intend to generalize the system to support more complex topologies—spanning multiple GPUs and hosts—bringing it closer to real-world OLAP deployments.

8 LESSONS LEARNED

This project provided valuable experience in building and debugging distributed systems with heterogeneous compute resources. We encountered and overcame challenges related to RDMA setup, synchronization bugs, and GPU memory management, all of which deepened our understanding of low-level performance trade-offs in hybrid CPU-GPU architectures. One key lesson was the importance of carefully balancing computation and communication: while reducing network traffic is often seen as a win, the cost of pre-processing that data can easily overshadow the savings unless carefully optimized. Additionally, we learned the value of automation and observability in performance testing—small improvements in testing workflows can dramatically accelerate iteration speed, especially in systems with multiple moving parts.

REFERENCES

[1] J. Bennett, H. Abbasi, M. Wolf, S. Ahern, B. Wang, K. Schwan, S. Klasky, N. Podhorszki, G. Lofstead, M. Parashar, et al. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. *International Journal of High Performance Computing Applications*, 26(3):167–181, 2012.

[2] N. Boeschen and C. Binnig. Gacco - a gpu-accelerated oltp dbms. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1003–1016, New York, NY, USA, 2022. Association for Computing Machinery.

[3] N. Boeschen, T. Ziegler, and C. Binnig. Golap: A gpu-in-data-path architecture for high-speed olap. In *Proceedings of the 2025 ACM SIGMOD International Conference on Management of Data*. ACM, 2025.

[4] H. Chu, S. Kim, J.-Y. Lee, and Y.-K. Suh. Empirical evaluation across multiple gpu-accelerated dbmses. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.

[5] N. Corporation. Rapids: Gpu accelerated data science. https://rapids.ai/, 2024. Accessed: 2025-04-18.

[6] N. Corporation. Rapids accelerator for apache spark. https://nvidia.github.io/spark-rapids/, 2025. Accessed: 2025-04-18.

[7] H. Gao and N. Sakharnykh. Scaling joins to a thousand gpus. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 55–64, 2021.

[8] H. Gao and N. Sakharnykh. Scaling joins to a thousand gpus. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, pages 55–64, 2021.

[9] H. Geng, M. J. Freedman, I. Stoica, and S. Ratnasamy. In-network computation for spark-based analytics. In *2019 IEEE/ACM Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pages 23–29. IEEE, 2019.

[10] B. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operators using a network processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware*, DaMoN '05, page 1–es, New York, NY, USA, 2005. Association for Computing Machinery.

[11] M. Jasny, L. Thostrup, T. Ziegler, and C. Binnig. P4db - the case for in-network oltp. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1375–1389, New York, NY, USA, 2022. Association for Computing Machinery.

[12] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, pages 1017–1032. ACM, 2022.

[13] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1017–1032. ACM, 2022.

[14] C. Mustard, F. Ruffy, A. Gakhokidze, I. Beschastnikh, and A. Fedorova. Jumpgate: In-Network processing as a service for data analytics. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[15] A. Ocsa. Blazingsql: Gpu-accelerated sql engine built on rapids ai. *LatinX in AI (LXAI) Research*, 2021.

[16] J. Paul, S. Lu, B. He, and C. T. Lau. Mg-join: A scalable join for massively parallel multi-gpu architectures. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1413–1425. ACM, 2021.

[17] A. Sapio, T. Li, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Scaling distributed machine learning with in-network aggregation. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 449–463. ACM, 2021.

[18] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig. Distributed gpu joins on fast rdma-capable networks. *Proceedings of the ACM on Management of Data*, 1(1):29:1–29:26, 2023.

[19] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig. Distributed GPU joins on fast rdma-capable networks. *Proc. ACM Manag. Data*, 1(1):29:1–29:26, 2023.

[20] D. Tomé, T. Gubner, M. Raasveldt, E. Rozenberg, and P. Boncz. Optimizing group-by and aggregation using gpu-cpu co-processing. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, pages 1–10, 2018.

[21] Q. Wang, Q. Guan, X. Wang, Z. Zhang, and S. Fu. Ds-sync: Efficient data shuffle and synchronization for distributed dnn training. *arXiv preprint arXiv:2007.03298*, 2020.

[22] J. Zhang, S. Wei, M. J. Freedman, B. Heller, M. Alizadeh, and G. R. Ganger. Switchml: Accelerating distributed machine learning with in-network aggregation. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 285–300. USENIX Association, 2020.

[23] Z. Zhang, C. Li, X. Liu, Y. Shao, and J. Fan. On the scaling of distributed training of deep learning models: A practical study. *arXiv preprint arXiv:2006.10103*, 2020.