MGI: A Communication Framework for Data Processing in Massive GPU Infrastructures

Di Wu
University of Toronto
Toronto, Canada
diwu@cs.toronto.edu

Hongshi Tan
National University of Singapore
Singapore, Singapore
hongshi@u.nus.edu

Hanzhang Yang
University of Toronto
Toronto, Canada
hanzhang@cs.toronto.edu

Bingsheng He
National University of Singapore
Singapore, Singapore
dcsheb@nus.edu.sg

Qizhen Zhang
University of Toronto
Toronto, Canada
qz@cs.toronto.edu

ABSTRACT

This paper presents MGI, a general communication framework for performing data processing tasks in massive GPU infrastructures. Inter-GPU data transfer performance is crucial to multi-GPU data processing, and existing solutions repeatedly implement the same set of communication optimizations. MGI identifies these techniques and applies them judiciously behind a simple interface. Enabling MGI are (1) a central controller that models relevant hardware resources as an annotated graph and automates infrastructure-level optimizations to construct transfer plans and (2) a scalable data plane where buffers and executors are carefully designed to incorporate device- and link-level optimizations to execute data transfers efficiently. Our experiments on a variety of GPU infrastructures and workloads show that MGI significantly improves multi-GPU data processing performance compared to existing frameworks.

PVLDB Reference Format:
Di Wu, Hongshi Tan, Hanzhang Yang, Bingsheng He, Qizhen Zhang. MGI: A Communication Framework for Data Processing in Massive GPU Infrastructures. PVLDB, 19(9): 2262 - 2275, 2026.
doi:10.14778/3819518.3819549

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/fardatalab/MGI.

1 INTRODUCTION

Massive GPU Infrastructures are a byproduct of the recent Artificial Intelligence (AI) boom: today's largest AI infrastructures consist of hundreds of thousands of powerful GPUs to train and serve large foundation models [6, 8, 28, 54]. These infrastructures, if utilized for data processing, can significantly expand the scale of hardware acceleration: a thousand NVIDIA 40 GB A100 GPUs provide 40 TB HBM2 memory and 20 PFLOPS—data can then be processed *in situ* on GPUs without being spilled into the host memory. Data exchange between the host machine and the GPU via the PCIe bus has been the main obstacle in using GPUs for databases [70], and thus this trend presents new opportunities.

Promise for Data Processing. The availability of large-scale GPU infrastructures has motivated a line of recent work on processing database workloads using multiple GPUs [19, 24, 39, 53, 58, 63, 68, 69]. These multi-GPU systems and algorithms have significantly improved the scale of databases accelerated by GPUs. For instance, with eight V100 GPUs interconnected with a hybrid cube-mesh topology of NVLinks on a DGX server, MG-Join [53] achieved 25× speedup for TPC-H scale factor (SF) 250 compared to its CPU baseline; with eight A100 GPUs fully connected with NVLinks, Lancelot [68] outperforms DuckDB by 20× on TPC-H SF 320; with ten V100 GPUs networked by InfiniBand, GPUDirect join [63] can achieve 5 billion tuples/s join throughput when joining billion-tuple tables; finally, with 1024 A100 GPUs locally connected by NVLinks globally networked by InfiniBand, Gao et al. [19] scaled distributed hash join throughput to 1.7 trillion tuples/s on TPC-H SF 100000.

Communication Bottleneck and Optimizations. Recent multi-GPU efforts have also confirmed the impact of cross-GPU communication. Indeed, even with NVLinks, transferring data across GPUs is severalfold slower than accessing GPU memory (e.g., 300 GB/s single-direction bandwidth with NVLink 3.0 vs. 1550 GB/s HBM2 on 40 GB A100). When scaling to networked GPU servers, the network speed can further become the bottleneck. This is despite advances in data center networking to support large-scale model training and deployment [18, 25], e.g., 400 Gbps RDMA. In response, existing multi-GPU data processing solutions have proposed effective optimizations to improve inter-GPU communication performance, which can be categorized as follows.

Opt 1: Pipelining Overlapping computation and communication has been an essential design in large-scale data-centric systems to offset communication overhead [27, 40, 64], which is also applicable for cross-GPU communication. With pipelined data transfers, rather than wait for all data tuples to arrive, compute kernels on the destination GPU can process available tuples while more data is on the fly. Pipelining achieves most of its performance benefit when communication time and computation time are on par. In addition, it reduces the message buffer size and thus the GPU memory footprint. This optimization has been adopted in many multi-GPU solutions for data processing [19, 39, 53, 58, 63, 69].

Opt 2: Batching To amortize transfer overhead and increase link utilization, small data tuples should be consolidated into larger

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819549

batches. For example, transferring 64 MB chunks is 4.8× faster than transferring 1 MB chunks with `cudaMemcpyPeer` between two A100 GPUs connected via NVLink 3.0 (260 GB/s vs. 55 GB/s). This optimization was incorporated in MG-Join [53] and Vortex [69] with their tuned unit transfer sizes (16 MB and 24 MB, respectively). Batching also effectively improves network communication performance, as adopted in distributed GPU joins [19, 63].

Opt 3: Multi-path Due to advanced interconnectivity between GPUs, there are often multiple data paths between two GPUs. For example, in the hybrid cube-mesh topology on DGX-1 [48], there can be at maximum five equal-bandwidth NVLink 2.0 paths between two V100 GPUs. If fully taken, they can offer 5× throughput increase for cross-GPU communication. Multi-path data transfers have been exploited in MG-Join (multi-path routing) [53], Vortex (multi-path forwarding) [69], and Lancelot (cross-GPU broadcast) [68].

Opt 4: NIC-direct When transferring data to a GPU on a remote server, rather than relaying the messages from the network interface card (NIC) to the server host and then to the destination GPU, GPUDirect [47] allows direct access from the NIC to GPU memory, thereby saving host-GPU PCIe communication. This technique has been employed by GPUDirect join [63] and thousand-GPU join [19].

Problems with Existing GPU Communication Solutions. Although previous proposals for multi-GPU data processing have effectively improved the performance of cross-GPU data transfers, *they lack generality:* they were proposed for specific scale (scaling up on a single GPU server [24, 39, 53, 58, 68, 69] vs. scaling out to networked GPU servers [19, 63]), some for specific interconnect technology (e.g., direct connectivity with NVLinks [69], PCIe [58], or RDMA [19, 63]) and specific workload (e.g., join [19, 53, 58, 63] or sort [39]). Deployment at a different scale with a different topology or interplays with other workloads would invalidate their efficacy. *General GPU communication frameworks such as NCCL [46] and UCX [65] lack efficiency:* they target machine learning workloads and are not optimized for data processing. Computation and communication cannot be overlapped (i.e., no pipelining) with the collective API in NCCL-like or MPI [64], UCX's point-to-point primitives are not broadcast-friendly, and multi-path data transfers need to be manually implemented on a case-by-case basis.

Our Proposal. To support data processing in massive GPU infrastructures, we propose MGI, a general communication framework that provides a simple interface and effective optimization techniques for data processing tasks on GPUs. MGI's interface consists of basic asynchronous send and receive functions. It allows for schema specification and introduces channels to easily create data flows for various communication patterns. Behind the API, MGI provides the communication optimizations (i.e., Opt 1–Opt 4) and applies them automatically. MGI transparently scales up and out and adapts to the topology of the target GPU infrastructure.

Challenges and Contributions. Developing a general and efficient GPU communication framework for data processing presents several significant challenges. First, real-world GPU infrastructures, especially the interconnectivities between GPUs, are heterogeneous. MGI must (and does) apply its optimizations judiciously based on the target GPU infrastructure. Second, specific inter-GPU data movement primitives exhibit sophisticated characteristics. For instance, peer-to-peer (P2P) direct memory access (DMA) achieves low-latency data transfers, but it requires direct or switched PCIe or NVLink connectivity between source and destination GPUs and the peak bandwidth is lower than host-issued `cudaMemcpyPeer`. MGI must (and does) incorporate such sophistication to efficiently harness these primitives. Moreover, unlike CPU programs, GPU applications execute in massively parallel with hundreds of thousands of threads. Uncoordinated communication can easily lead to warp divergence. MGI must (and does) carefully partition application threads and pace its flows with minimal synchronization overhead. MGI achieves all its design goals with (1) *a central controller* that models the target GPU infrastructure as an annotated graph to make global communication decisions, and (2) *a distributed data plane* that optimizes data transfers on individual devices and links. We have implemented various data processing tasks with MGI's API, along with a variety of communication patterns. Our experimental results show that MGI can effectively shorten cross-GPU communication time and improve overall data processing task performance. In summary, this paper makes the following contributions:

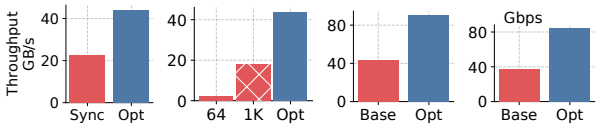
- Motivation for a new communication framework (§2).
- Architecture of MGI (§3).
- Design of MGI's control plane (§4) and data plane (§5).
- Publication of MGI's codebase, and extensive evaluation of data processing performance with MGI using various workloads and hardware setups (§6).

2 WHY A NEW FRAMEWORK

2.1 Massive GPU Infrastructures Are Increasingly Common

GPUs (Graphics Processing Units), originally designed to accelerate graphics rendering targeting gaming and multimedia markets, have evolved to high-throughput computing devices for general applications [31, 36, 62]. A GPU device consists of multiple Streaming Multiprocessors (SMs), each containing numerous simple cores that execute instructions in a Single Instruction, Multiple Threads (SIMT) fashion. GPU memory is organized in hierarchy, including global memory, L2 cache, SM-local shared memory and L1 cache, and registers, each varying in size and access speed. Today's GPUs can execute up to hundreds of thousands of threads in parallel with terabytes/s random memory speed.

Due to the emerging generative capabilities of large foundation models and popularity of AI applications [13, 50], many AI data centers comprising large amounts of GPUs have been built across the globe in recent years [11, 21, 57]. These GPU resources are used to train and deploy large models at scale [6, 8, 28, 54]. For instance, the xAI Colossus Supercomputer consists of 100 thousand NVIDIA H100 GPUs [12], and Meta plans to incorporate more than 1 million GPUs in its data centers [56]. This trend applies not only to tech giants. For individual practitioners, cloud vendors such as AWS [2], Azure [41], GCP [20], and OCI [51] are offering powerful GPUs in their commodity instances. Computing infrastructures in academia, e.g., CloudLab [10], Chameleon Cloud [7], and Digital Research Alliance of Canada [14], are progressively incorporating more GPUs in their expansions. Therefore, massive GPU infrastructures are becoming increasingly available. When they are not used by AI jobs [4, 42], other applications, such as data analytics, can also benefit from the massive-scale acceleration.



(a) Opt 1 (b) Opt 2 (c) Opt 3 (d) Opt 4

Figure 1: Optimizations for inter-GPU communication.

2.2 Communication Optimizations Matter

To verify the significance of optimizing inter-GPU communication, we use isolated microbenchmarks to evaluate the performance improvement each individual optimization brings, from Opt 1 to Opt 4. Our basic setup is a GPU cluster with four A100 GPUs fully interconnected by NVLink 3.0. Each pair of GPUs are connected with two NVLink 3.0 links, providing a theoretical 50 GB/s unidirectional bandwidth. We use `cudaMemcpyPeer` as the GPU-to-GPU communication primitive. The measurement setup and results for each communication optimization are as follows.

To measure the benefit of pipelining, we transfer 1 GB data tuples from the source GPU to the destination GPU. We compare two options: without Opt 1, the destination GPU processes the data tuples after all tuples are received; with Opt 1, the destination GPU processes tuples as they arrive. Figure 1a shows the result: Opt 1 brings a ~2× throughput increase (23 GB/s vs. 44 GB/s).

We next evaluate the effect of batching (Opt 2): we vary how many 256-B tuples are batched on the source GPU before sending them to the destination GPU: 64, 1 thousand, and 64 thousand tuples. Figure 1b shows that the NVLink bandwidth is fully utilized when 16 MB of tuples are batched (the "Opt" bar), while smaller batches utilize only 5% and 41% of the link bandwidth.

In addition to the direct NVLink between a pair of GPUs, there are two indirect paths, each via an intermediate GPU. To evaluate the benefit of multiple paths (Opt 3), we manually enable the two additional paths by separating the data on the source GPU into three flows. Figure 1c shows the effect: although the indirect paths are longer than the direct one, each can contribute 23 GB/s throughput, and thus the total communication throughput increases to 90 GB/s—2.1× faster than the single path.

Finally, we evaluate NIC-direct communication (Opt 4): we transfer data between GPUs on two servers, each equipped with a 100 Gbps ConnectX-5 NIC, with pipelining disabled. With the baseline approach, data is first copied to the host memory, then sent to the remote server with RDMA, and finally copied to the destination GPU. In comparison, Opt 4 uses GPUDirect RDMA to copy the data in the source GPU's memory from the NIC and write it directly to the destination GPU's memory. Figure 1d shows that bypassing the hosts between networked GPUs increases the data transfer throughput from 37 Gbps to 84 Gbps.

2.3 Existing Solutions Are Insufficient

Table 1 provides an overview of communication optimization support in existing solutions. Opt 1–Opt 4 have been partially incorporated in existing data processing systems or algorithms [19, 24, 39, 53, 58, 63, 67–69]. However, there have not been GPU communication frameworks that offer these optimizations, and the effort must be repeated for new infrastructures or developing new solutions.

Table 1: Communication optimization support for data processing in existing solutions. *Including other NCCL-like libraries, such as RCCL, MSCCL, Gloo, and CUDA-aware MPI.

	Data Processing Proposals								Frameworks	
	[19]	[39]	[53]	[58]	[63]	[67]	[68]	[69]	NCCL*	UCX
Opt 1	✓	✓	✓	✓			✓	✓		
Opt 2	✓		✓				✓	✓		
Opt 3			✓		✓	✓		✓		
Opt 4	✓				✓	✓			✓	✓

Table 2: Summary of GPU infrastructures adopted in recent work. *Applicable optimizations.

GPU Infrastructure and Adoption	Connectivity	Opts*
Single-server PCIe [58, 68]	Fig. 2a	1, 2
Single-server PCIe and NVLinks [39, 53, 66, 69]	Fig. 2b	1–3
Single-server PCIe with NUMA [17]	Fig. 2c	1, 2
Single-server PCIe and NVLink with NUMA [39]	Fig. 2d	1–3
Single-server NVLink with NUMA [39]	Fig. 2e	1–3
Single-server hybrid cube mesh [53, 72]	Fig. 2f	1–3
Single-server NVSwitch [30, 39, 68]	Fig. 2g	1–3
Multi-server Ethernet [1]	Fig. 2h	1, 2
Multi-server RDMA [63]	Fig. 2i	1, 2, 4
Multi-server NVSwitch and RDMA [19]	Fig. 2j	1–4

To further show the complexities of generalization, we have surveyed the infrastructures adopted in recent multi-GPU data processing proposals. Table 2 summarizes the setups and applicable communication optimizations. Figure 2 depicts the inter-GPU connectivity and topology in each setup. As can be observed, communication optimizations are coupled with the specific connectivity to a great extent. In addition, the parameters in these optimizations (e.g., batch size) should be fine-tuned based on the hardware (e.g., link speed). Hence, in the era of massive GPU infrastructures, data processing on multiple GPUs can benefit from a general framework that applies the optimizations across heterogeneous setups.

3 THE MGI FRAMEWORK

MGI is a communication framework for data processing in massive GPU infrastructures with the goals below.

- **Fast.** MGI should fully apply communication optimizations to maximize data transfer throughput. MGI should also facilitate developing efficient GPU kernels.
- **General.** Facing up, MGI should be general to applications to support various communication patterns in different data processing tasks. Facing down, MGI should be general to infrastructures to enable and disable optimizations based on the specific setup in its current deployment.

These two goals have determined the interface, architecture, and detailed communication execution strategies of the framework.

3.1 Interface

Host API. Given the variety of operations and data distributions, communication patterns in data processing tasks can be arbitrary

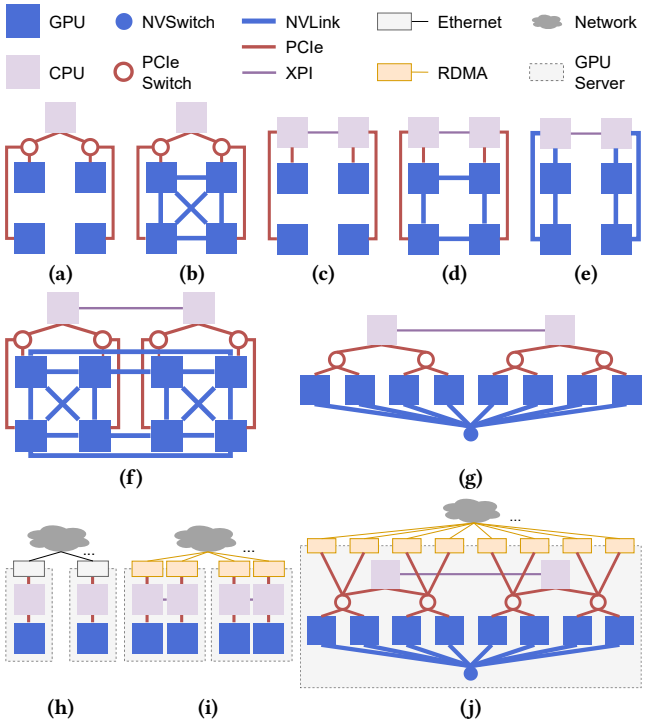


Figure 2: Inter-GPU connectivities adopted in recent multi-GPU solutions. XPI represents cross-socket CPU interconnects (e.g., Intel QuickPath Interconnect [26]).

and irregular, e.g., many-to-many shuffle, one-to-many broadcast, or point-to-point delivery. These cannot be easily expressed with collective-based GPU communication frameworks [46]. Inspired by communication libraries widely adopted by databases [34] and recent proposals for high-performance database networking [64], MGI adopts a host API based on endpoints and channels for composing data flows between GPUs. Specifically, in MGI, communication activities are organized in *Channels*. A channel consists of uniquely-identified *Endpoints*. Each GPU device in the infrastructure can be registered as an endpoint. Table 3 shows the API for applications to create endpoints (`CreateEp`) and form communication channels (`CreatCh`). Specifically, `CreateEp` creates an endpoint for the first available GPU. The user can also specify a different GPU. A global endpoint id is returned upon a successful invocation. `CreateCh` creates a channel with a list of endpoints as sources and a second list of endpoints as destinations and returns a global channel id. Generally, data is produced by source endpoints and consumed by the destination endpoints. To support database-specific use cases, e.g., shuffling a table with a partition attribute for distributed join, a channel can be further specified with the following arguments:

- *Schema* of the communication data, which specifies an ordered list of attributes and their types.
- *PartKey* in the schema, which specifies the attribute in the schema that will serve as the partition key.

Otherwise, data sent by each source endpoint is broadcast to destination endpoints. A GPU endpoint can participate in multiple channels or in one channel twice as both a source and a destination. These two functions are supposed to be invoked by the host end of the application to implement arbitrary data flows, such as shuffling, combining, broadcast, or point-to-point data transfers. Channels and endpoints can be deleted with `DeleteCh` and `DeleteEp`.

Table 3: MGI's API. Endpoints and channels are managed on hosts, while data is directly transferred on GPUs.

API	Function
`CreateEp`([, *Device*])	Create an endpoint and returns its id
`CreateCh`(*SrcEps, DstEps* [, *Schema, PartKey*])	Create a channel and return its id
`DeleteCh`(*ChId*)	Delete a channel
`DeleteEp`(*EpId*)	Delete an endpoint
`Send`(*ChId, SrcBuf* [, *DstEp, Size*])	Send data to a channel and return bytes successfully sent
`Flush`(*ChId*)	Complete all sends previously issued
`Recv`(*ChId, DstBuf* [, *Size*])	Receive data from a channel and returns bytes successfully received

Device API. Unlike existing GPU communication frameworks which initiate communication activities on hosts [46, 65], MGI's data-plane API is directly called on GPU devices. This design avoids breaking pipelining: data transfers occur concurrently with user kernel execution without blocking the latter. Device API also facilitates kernel fusions and megakernels [52]. Using the channel id, user kernels on endpoints can perform inter-GPU communication with MGI's data-plane functions: `Send`, `Flush`, and `Recv`.

Send Semantics. User kernels on source endpoints send data tuples into a channel by calling the `Send` function with the channel id and the addresses of the tuples. A tuple t is sent with the following rules: (1) if the optional *DstEp* argument is specified, t is sent to *DstEp*; (2) if the schema and the partition key of the channel are both specified, t is sent to the destination endpoint $PartKey\,\%\,|DstEps|$; (3) otherwise, t is broadcast to all destination endpoints. Loopback is discussed in §5.4. To enable pipelining (Opt 1), `Send` is asynchronous and returns immediately when the tuple is accepted in MGI's send buffer, and the number of bytes accepted is returned. Data accepted in MGI's send buffer will be transferred to the destination endpoint(s) using reliable local interconnects (e.g., NVLinks or PCIe) or end-to-end transport protocols (e.g., TCP or reliable RDMA) depending on the remote locations and hardware availability. If MGI's send buffer is full, the function returns zero.

Flush Semantics. To signal the framework that all data tuples have been sent, poll the completions of previously send operations, and synchronize with the receivers, user kernels can invoke `Flush`, which appends an `EOF` tuple into the channel and blocks until all data tuples sent from the current user thread block have been delivered to their destination endpoints.

Receive Semantics. To receive data tuples from a channel, user kernels on destination endpoints invoke `Recv` by providing the channel id *ChId* and the reception buffer *DstBuf*. In case of a schemaless channel, the size of *DstBuf* should be specified. This function is also non-blocking: if data is available in MGI's receive buffer, it is moved to *DstBuf* and its size is returned; otherwise, it returns zero. If no more tuples to be received from the channel, `EOF` is returned.

Example. To demonstrate the usage of MGI's API, we show the pseudocode of a multi-GPU GroupBy implementation in Figure 3.

```
// ON DEVICE
__global__ GroupBy(ChId, Part, PartSize, TupLen) {
    extern __shared__ totSend = 0;
    extern __shared__ ht = MakeHashTable();
    // Pipelined send, receive, and GroupBy
    while (totSend != PartSize) {
        while (Send(ChId, Part[Increment(&totSend)]));
        while (Recv(ChId, &tuple)) AggFunc(ht, tuple);
    }
    Flush(ChId);
    __sync();
    do {
        ret = Recv(ChId, &tuple);
        if (ret == EOF) break;
        if (ret) AggFunc(ht, tuple);
    } while (true);
}

// ON HOST
for(d = 0; d != AllGPUs.size(); d++) {
    parts[d] = LoadPart(d);
    eps[d] = CreateEp(d);
}
schema = {int, long};
partKey = 0;
ch = CreateCh(eps,eps,schema,partKey);
for (d = 0; d != eps.size(); d++)
    GroupBy<<<BLOCKS,THREADS>>>(ch, parts[d], parts[d].
     size(), sizeof(schema));
```

Figure 3: A multi-GPU GroupBy pseudocode with MGI's API.

Host. Each GPU on the server is responsible for processing one partition of the input database table and is registered as an endpoint with `CreateEp`. All GPUs are utilized for the job, and thus all endpoints are both sources and destinations. A communication channel is created with `CreateCh` with specified schema and partition key. Finally, the kernel is launched on each GPU with the channel id.

Device. User threads on each GPU call MGI's API to shuffle the tuples across GPUs based on the GroupBy key. The workflow is pipelined by interleaving `Send`, `Recv`, and the aggregation function. Each thread calls `Flush` to complete sending and continuously calls `Recv` to receive and aggregate tuples until `EOF` is returned.

3.2 Architecture

MGI is a distributed framework comprised of a control plane that generates data transfer plans incorporating global, infrastructure-level optimizations (Opt 3 and Opt 4) and a data plane that executes transfer plans with device- and link-level optimizations (Opt 1 and Opt 2). Figure 4 shows its overall architecture.

Hardware Infrastructure. A GPU infrastructure consists of GPU servers. Each server has CPU, memory, and one or multiple GPU devices. The CPU and GPU(s) on a server are connected by intra-server links or switches, e.g., NVLinks, NVSwitches, and PCIe, and form the server's *local connectivity*. Different GPU servers are connected by a network that consists of NICs and the fabric (i.e., cables and switches) and provides end-to-end connections with a reliable transport protocol (e.g., TCP or RDMA), forming the infrastructure's *network connectivity*. For generality, MGI does not assume a specific local or network connectivity.

Controller. A MGI deployment is managed by its *Controller*, a central component at the heart of the control plane. It is an independent service that can be deployed on a dedicated CPU server in

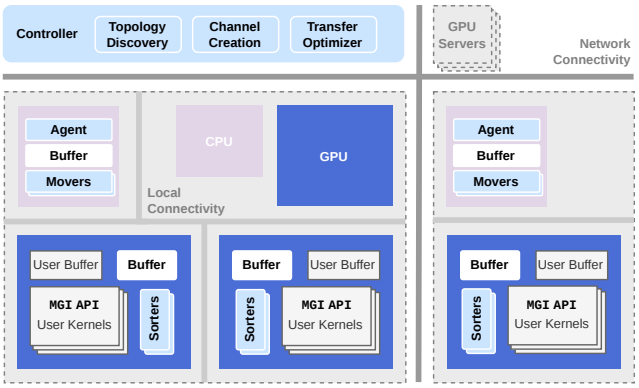


Figure 4: Architecture and system components of MGI.

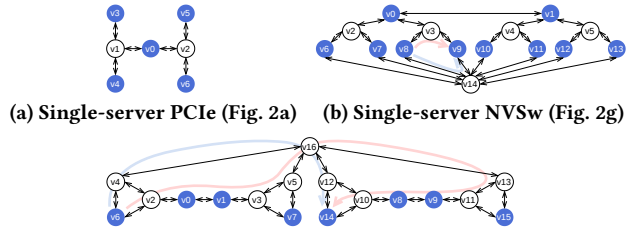
the cluster. Since it consumes limited resources (by default 10 CPU cores and under 1 GB memory), it can also be colocated on one of the GPU servers and further replicated (e.g., with ZooKeeper [3]) across multiple of them for fault tolerance (if the controller becomes unavailable, although existing channels are not affected, new channels cannot be created). Specifically, the controller creates a thread pool to receive and process network requests from all GPU servers and employs three modules to construct global transfer plans: (1) *Topology Discovery*, which generates a GPU network graph with the intra-server and inter-server connectivities augmented with hardware specs, (2) *Channel Creation*, which processes user requests from applications and generates transfer plans, and (3) *Transfer Optimizer*, which takes the topology graph and channel specifications and applies infrastructure-level optimizations, i.e., multi-path (Opt 3) and NIC-direct (Opt 4).

Agent. Each server has an *Agent* that runs on the server's CPU and accepts MGI's host API invocations, i.e., `Create(/Delete)Ep(/Ch)`. On its startup, an agent collects the local connectivities and hardware specifications and reports them to the controller, where the global topology of the infrastructure is discovered. When an application creates a channel, the agent forwards the request to the controller and, when the transfer plan is returned, sets up message buffers and data-plane executors. The channel is then ready for use.

Sorters and Movers. Data-plane operations are collectively executed by workers on each GPU (*Sorters*) and the CPU (*Movers*). Sorters partition data tuples generated by asynchronous `Send` calls (Opt 1) by their destination endpoints. Once sufficient tuples are produced for a specific destination (Opt 2), they will be forwarded. Data forwarding between GPUs is performed by *Movers*.

Message Buffers. To pipeline data transfers, sorters on source endpoints utilize two separate buffers: *Stage Buffers* that accept user data and *Sort Buffers* that store the results of sorting, i.e., tuples partitioned by destinations. When transferring data via the network, the host will also create sort buffers to forward the tuples if NIC-direct is not possible. Destination endpoints create *Receive Buffers* to receive tuples and *Final Buffers* to store tuples to be consumed.

Workflow. An application consists of a host program run on each server's CPU and GPU kernels run on each server's GPUs. To create endpoints and channels, the host program calls MGI's host API to send the requests to the local agent. The agent then communicates with the controller and creates executors and buffers based on



(a) Single-server PCIe (Fig. 2a) (b) Single-server NVSw (Fig. 2g)

(c) Multi-server RDMA (Fig. 2i)

Figure 5: Examples of MGI's GIGs. Edge weights are omitted.

the response. At runtime, the application's GPU kernels call MGI's device API to send and receive data to/from other GPUs. The sorters on source endpoints and the movers on the hosts coordinate to transfer data to the destination endpoints, where the sorters finally dispatch data to the application's thread blocks.

Debugging and Profiling. As an underlying framework for data processing systems, MGI provides essential logging and performance testing support to facilitate system debugging, monitoring, and profiling. Specifically, logging at multiple levels can be enabled/disabled on demand from `info` logs, such as the speed of sorters and movers and the usage of buffers, to `error` logs, such as network connectivity issues. A suite of performance test tools are also provided to measure the throughput of a variety of communication patterns in the deployed infrastructure (§6.3).

Novelty. While the specific optimizations (Opt 1–Opt 4) have been individually explored in prior data processing proposals (Table 1), MGI makes the following novel contributions:

- *Full support of optimizations,* which allows data processing tasks to unlock the combined efficiency of all the optimizations (e.g., for the topology in Figure 2j).
- *Adaptation of optimizations to infrastructure,* which allows data processing tasks to achieve high performance across heterogeneous hardware setups.
- *Framework design,* which allows data processing tasks to benefit without implementing the optimizations.

Achieving these requires addressing challenges in both the control and data planes, which we detail in the next sections.

4 CONTROL PLANE

MGI adopts a central controller to make network-wide communication decisions. This section presents our network abstraction and how MGI generates optimized transfer plans and creates channels.

4.1 Network Abstraction

MGI enables infrastructure-level communication optimizations with its network abstraction, called *GPU Infrastructure Graph* (*GIG* for short). Specifically, a GPU infrastructure is modeled as a directed graph $G = (V, E)$, where the vertex set V includes all types of devices that participate in communication activities (i.e., generating or forwarding communication data), e.g., GPUs, CPUs, NICs, and switches. The edge set E includes all types of links that connect vertices with an associated weight denoting the link capacity in each direction. On top of the basic model, vertices are further classified into *Managed Vertices* (Ⓥ) and *Unmanaged Vertices* (ⓥ) depending on whether MGI has components (buffers and executors) running on the vertex. Currently, MGI manages CPU and GPU vertices and

Table 4: List of hardware components for GIG construction.

Hardware Components	GIG Elements
CPUs, declared GPUs	Managed vertices
NICs, NVSwitches, PCIe switches	Unmanaged vertices
PCIe lanes, NVLinks, XPI links	Edges

leaves NIC and switch vertices unmanaged. An edge (⟶) can connect any two vertices, regardless of their types, and different types of links only differ in capacity (edge weight).

Figure 5 shows three examples of MGI's GIGs, corresponding to three GPU connectivities in Figure 2. Figure 5a models the GPUs and the CPU on a single-server connected by PCIe (Figure 2a) as a GIG consisting of 5 managed vertices and 2 unmanaged vertex: v_0, v_1–v_2, and v_3–v_6 represent the CPU, PCIe switches, and GPUs, respectively. Figure 5b models a DGX A100 server (Figure 2g), where there are two CPU vertices (v_0, v_1) and eight GPU vertices (v_6–v_{13}). The unmanaged vertices include 4 PCIe switches (v_2–v_5) and NVSwitch (v_{14}). Figure 5c shows a scale-out example with two servers (Figure 2i): the two CPUs on each server are modeled as two managed vertices that are connected (v_0, v_1, and $v8$, v_9, respectively). Each GPU vertex (v_6, v_7, v_{14}, or v_{15}) is connected to the PCIe switch in its NUMA socket, which is an unmanaged vertex (v_2, v_3, v_{10}, or v_{11}). Since the NICs are RDMA NICs that can directly access GPU memory, each NIC vertex (v_4, v_5, v_{12}, or v_{13}) is connected to both its corresponding PCIe switch and GPU. Finally, there is an umanaged vertex representing the network (v_{16}), which connects all the NIC vertices. We next present how GIGs are constructed and utilized for global communication optimizations.

4.2 Graph Construction

A GIG is constructed following a two-level process. The agent on each server first builds a local graph, which serves as a subgraph of the global GIG, based on its local hardware components, as listed in Table 4. More specifically, the agent first includes the GPUs declared by the user and the CPUs on all NUMA nodes as managed vertices. It also detects communication devices: NICs, NVSwitches, and PCIe switches, and the links connecting these devices: PCIe lanes, NVLinks, and XPI links, augmented with their capacities. Without losing details required for communication optimizations, the agent simplifies the subgraph with the rules below:

- All NVSwitches are merged to a single unmanaged vertex.
- Links that have the same ends are merged to a single edge with the aggregated capacity as the weight.

Different NICs are modeled as separate unmanaged vertices. In addition, if a NIC can directly access GPU memory, an edge connects it to the local GPU vertices. These treatments facilitate NIC-direct and multi-NIC optimizations (§4.3). Once the subgraph is constructed, the agent reports it to the controller as shown in Figure 7.

When the controller receives the subgraphs from all agents, the Topology Discovery module (TD) adds an additional unmanaged vertex representing the network and connects it to all NIC vertices (with the link capacities as the edge weights) to form the final GIG.

4.3 Global Optimizations

The Transfer Optimizer module (TO) takes a list of source endpoints *SrcEps* and a list of destination endpoints *DstEps* and explores the

```
Input: G: GIG, S: srcs, D: dsts
Output: Paths from S to D
State: Path set P initialized to empty
1  for s in S, d in D do
2      if P(s,d) is not cached then
3          if s and d are on the same server then
4              Edmonds-Karp-Cache(G, P, s, d)
5          else
6              Edmonds-Karp-Cache(G, P, s, G.v_net)
7              Edmonds-Karp-Cache(G, P, G.v_net, d)
8              P(s, d) = Concat(P(s, G.v_net), P(G.v_net, d))
9              cache P(s, d)

10 Function Edmonds-Karp-Cache(G, P, s, d)
11     if P(s,d) is not cached then
12         P(s,d) = Edmonds-Karp(G.V, G.E, s, d)
13         cache P(s, d)
```

Figure 6: Algorithm in TO.

opportunities for optimizing data transfers from the sources to the destinations with Opt 3 and Opt 4, based on the GIG $G = (V, E)$.

To maximize the bandwidth from *SrcEps* $S = \{s_0, s_1, ...\} \subseteq V$ to *DstEps* $D = \{d_0, d_1, ...\} \subseteq V$, TO solves a *multi-source multi-sink max-flow problem*:

$$\arg\max_{P} \sum_{p \in P, s \in S, d \in D} C_p \tag{1}$$

where C_p is the capacity of a path p from s to d.

Complexity. Let N be the average number of GIG vertices on a server and M be the number of servers in the current deployment. Assume the full connectivity on each server (e.g., Figure 2b), which results in $O(N^2M)$ edges in G. Directly solving the multi-source multi-sink max-flow problem over the entire G is computationally expensive. The worst-case complexity using the Edmonds–Karp algorithm [16] is $O(|V||E|^2) = O(N^5M^3)$, which becomes prohibitive when the cluster size is large (e.g., $N = 10$ and $M = 1000$).

A Hierarchical and Incremental Approach. TO decomposes the problem by (1) exploiting the hierarchical structure of GIGs to lower complexity and (2) reducing the scope to single-source single-sink pairs to incrementally build the final solution and maximize the chance of reusing intermediate results. Figure 6 shows TO's path generation algorithm. Specifically, we reuse the previously generated paths with the `Edmonds-Karp-Cache` function (lines 10–13, `E.K.C.` for short). For each source and destination pair (s, d), we generate its paths $P(s, d)$ only if they have not been generated before (lines 1–2). If s and d are the same server, we directly run `E.K.C.` to generate and cache its paths (lines 3–4). If they are on different servers, we decompose the path generation at the server level (lines 5–9): `E.K.C.` is first applied to generate and cache paths from s to the unmanaged network vertex in G v_{net}, and then from v_{net} to d. These two sets of paths are finally concatenated at v_{net} and cached. This approach reduces the complexity to max-flow at the server level $O(N^5M)$, and we further reuse paths across channels.

Effectiveness. TO enables global optimizations as below.

- Opt 3: Multi-path The algorithm automatically detects multiple communication paths between a source endpoint and a destination endpoint, regardless of the link type. For instance, in Figure 5b, two paths can be taken for v_8 to transfer data to v_9: the NVLink path via v_{14} and PCIe path via v_3.
- Opt 4: NIC-direct Direct GPU accessibility from a NIC is first modeled in the GIG with an edge connecting the NIC and the GPU. The `E.K.` algorithm adopted in TO runs breath-first search (BFS) to find available paths and *thus favors shorter paths*. The direct edges between NICs and GPUs, if present, will always be included in the generated paths from s to v_{net} and from v_{net} to d. Figure 5c shows an example: the shortest path from v_6 to v_{14}, where NIC vertices (v_4 and v_{12}) directly access the GPUs, is selected.

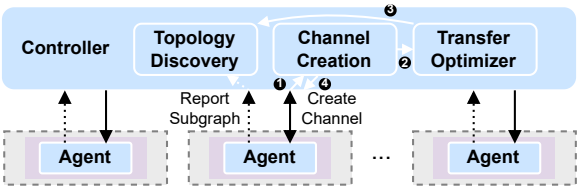


Figure 7: Control messages.

It further expands multi-path communication beyond a single server. Specifically, all NICs on a server are modeled in the GIG as umanaged vertices, which are connected to v_{net}. In decomposition, (1) all paths from s to v_{net} and from v_{net} to d must include NIC vertices, and (2) every NIC, if it can form a path that contributes to the aggregated bandwidth, will be included. As Figure 5c shows, in addition to the shortest path from v_6 to v_14 (via v_4, v_{16}, and v_{12}), the second NIC on each server is used to generate another path: $v_6 \rightarrow v_2 \rightarrow .. \rightarrow v_{10} \rightarrow v_{14}$.

Completeness. TO finds all possible paths between *SrcEps* and *DstEps*, i.e., any path identified by solving the multi-source multi-sink (MSMS) max-flow problem is also identified by TO. Specifically, the MSMS max-flow problem can be reduced to a single-source single-sink max-flow problem by adding a super-source s_{super} connected to each source and a super-destination d_{super} connected with each destination. For any path identified by the Edmonds–Karp algorithm: $s_{\text{super}} \rightarrow s \rightarrow ...(p) \rightarrow d \rightarrow d_{\text{super}}$, the path $s \rightarrow ...(p) \rightarrow d$ can be identified by `E.K.C` against (s, d) and included in $P(s, d)$.

Total Bandwidth as Capacity. C_p in formula (1) is the minimum total bandwidth among all links constituting p. We use total bandwidth (TBW), rather than currently available bandwidth (CABW), as capacity for two reasons. First, CABW changes in real time, which invalidates caching in `E.K.C.` because the generated paths for a fixed source-destination pair can vary based on the instantaneous link utilization. Second, the discovered paths with TBW are candidates used for generating the forwarding table (§4.4). They are not intended for traffic distribution. Which paths to use and how much traffic to distribute are dynamically determined at runtime by MGI's data plane (§5.4).

4.4 Channel Creation

Figure 7 shows the protocol for creating a channel. When MGI's controller receives a channel creation request from an agent (❶), it processes the request with the Channel Creation module (CC). CC first passes the parameters (sources and destinations) to TO (❷). TO references the GIG constructed by TD (❸) and generates

communication paths from the sources to the destinations. When TO returns the paths, CC constructs a channel-wise *forwarding table* for each managed vertex involved in the paths, which is finally sent to the agents of the servers involved in the channel (❹).

Forwarding Table Generation. For each managed vertex, the forwarding table determines the next hop (another managed vertex) based on the destinations of tuples and the paths to the next hop. The table consists of (`destination`, (`next`, `paths`)) entries keyed by destination and is generated as follows: CC visits each path $P(s, d)$ from TO and, for each managed vertex v included in the path starting from s, it includes all unmanaged vertices in P before the next managed vertex v' in a path $P_{v'}$ and populates v's forwarding table with a new entry $(d, (v', P_{v'}))$. If there is an existing entry for d in the table, the new entry is merged by `destination` and then, if needed, by `next`. No entries are generated for d. For example, the two paths from v_6 to v_{14} in Figure 5c lead to an entry $(v_{14}, \{(v_0, \{v_2\}), (v_14, \{v_4 \rightarrow v_{16} \rightarrow v_{12}\})\})$ in v_6's forwarding table.

5 DATA PLANE

This section presents the details of MGI's data plane and how local communication optimizations are incorporated.

5.1 Low-level Constructs

Data Movement Primitives. Inter-server transfers are relatively easy: if NIC-direct is enabled, data is directly transferred by the NIC with the NIC's driver support; otherwise, data is first copied into host memory and then sent via the most efficient transport protocol supported by the NIC, e.g., RDMA or TCP. Intra-server transfers are trickier. When two GPUs are not directly linked or under the same switching domain (e.g., PCIe or NVSwitch), peer-to-peer (P2P) transfers are unavailable—data must be moved to the host memory and then forwarded to the destination GPU traversing PCIe and XPI links. These cases are easily identified in the GIG.

Two options are available for P2P transfers: (1) on-device load/store instructions (adopted by NCCL) or (2) host-issued `Memcpy` (e.g., `cudaMemcpyPeer`), which have different resource and performance implications. Overall, the former facilitates low-latency transfers but consumes more SMs, while the latter incurs high latency for small messages but achieves higher throughput and consumes zero SMs. MGI favors high throughput over low latency for massive data transfers, and thus its P2P transfers are executed by host-issued operations, enabled by device-host synchronization (§5.4).

Thread Coordination. User kernels on GPUs are executed with exceedingly high parallelism, e.g., each of the 108 SMs on A100 can host up to 2048 threads organized in blocks, which total 200 K threads. Coordination at this scale is necessary—uncoordinated access to message buffers can lead to random memory access and many control branches and thus warp divergence [23]—and challenging as synchronization across blocks is an expensive operation. MGI adopts three-level coordination to coalesce accesses to message buffers while minimizing synchronization overhead.

- Within a block, where atomic instructions are efficient, threads are synchronized to always append tuples to the tail of the buffer, facilitating coalescing.
- Between blocks, user blocks are *assigned* to MGI's device executors (i.e., sorters), which poll, sort, and batch tuples by their destinations. All memory accesses issued by sorters in the same block are coalesced with best efforts.
- Between devices and servers, data flows are primarily controlled by MGI's host executors, i.e., movers.

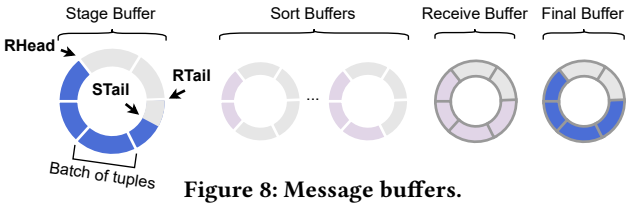


Figure 8: Message buffers.

5.2 Creating Executors

Sorters are GPU threads in MGI's persistent data-plane kernels to sort application-generated tuples into message buffers and apply communication optimizations. Their behavior highly depends on the forwarding table. To reduce complexity, rather than involve excessive control branches in a fully-specified implementation, the kernel is provided in each agent as a *template* parameterized by the forwarding table. An agent creates sorters on each local GPU by first launching the kernels and then placing a hash table that materializes the forwarding table in the GPU global memory.

Movers are host threads that detect the readiness of buffers, execute proper data movement primitives to transfer data across GPUs and servers, and perform flow control. Given that control branches are efficient on CPUs, the implementation of movers is pre-determined and also parameterized by the forwarding table. An agent creates a mover thread on a CPU for each local GPU.

5.3 Preparing Buffers

Message Format. Messages are the basic elements stored in the buffers. Each message consists of a header and a payload. The header encodes minimal information for forwarding: a 2 B channel id and a 2 B destination endpoint id, each supporting a 64 K space. The payload batches data tuples sharing the same next hop.

Buffer Format. To facilitate pipelined data transfers, message buffers in MGI's data plane are organized as rings [15, 35, 64, 71], as shown in Figure 8. Specifically, a ring consists of fixed-length segments, each storing a batch of tuples. Two pointers (`RHead` and `RTail`) coordinate the message producers and consumers. In addition, each segment also maintains `STail` that coordinates tuple batching among producers and signals the completion of a batch.

Preparation. An agent allocates four types of buffers. Specifically, when the transfer plan is received, the agent allocates a stage buffer per user block for each source endpoint, multiple sort buffers for each source endpoint based on the number of neighbors it has in the plan, a receive buffer for each destination endpoint, and a final buffer per user block for each destination endpoint. If CPUs and intermediate GPUs for multi-path are involved, receive buffers are also allocated for them to forward messages.

The segment of each buffer on an endpoint v is sized as $|v.\mathbb{S}| \times |\text{producer threads per block}| \times N$ where $\mathbb{S}$ is the schema defined for the channel and N determines the degree of batching. This segment size allows all the producer threads in the same block to send tuples in parallel and sufficient data is batched before transferring.

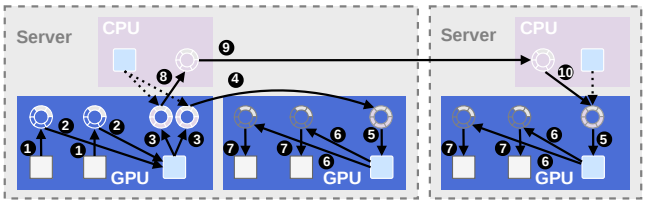


Figure 9: Data transfer pipelines to local and remote GPUs

5.4 Data Transfer Pipelines

Data transfers in MGI are pipelined between the message buffers driven by the data-plane executors, as shown in Figure 9.

Pipeline to Local GPUs. When user threads on source endpoints call `Send`, they prepend the channel id and destination endpoint id to the input tuples and append the tuples at `STails` in the `RTail` segments of the stage buffers and then advance `STails` (❶). The first sorter in a sorter block polls the `RHead` segments of the stage buffers from the assigned user blocks, and once a segment is full (i.e., its `STail` reaches the end), all sorters in the block collectively read the tuples in the segment (❷) and write each tuple to a sort buffer by looking up the forwarding table using the tuple's destination (❸). When finished, the first sorter resets `STail` and advances `RHead`. A mover polls `STails` in the `RHead` segments of the stage buffers on its GPU, and once a segment is full, it executes an `Memcpy` command to transfer the segment to the `RTail` segment of the receive buffer on the destination GPU (❹). The first sorter in a sorter block on destination endpoints polls the `RHead` segment of the receive buffer, and when new segments are received, all sorters collectively distribute the segments by reading the tuples from the receive buffer (❺) and writing to the final buffers (❻). Finally, the tuples are consumed when user threads on the destination endpoints call `Recv` (❼). A special case is loopback, i.e., the destination of a tuple is the source endpoint, for which the pipeline is optimized by direclty placing the tuple in the local receive buffer during sorting (❸). Doing so avoids an extra copy from a sort buffer to the receive buffer.

Pipeline to Remote GPUs. This pipeline shares the same initial (❶–❸) and final (❺–❼) stages with the local one but is deeper. When a mover polls a full `RHead` segment of the sort buffer on the local GPU targeting the CPU, it reads the segment to its receive buffer (❽) and sends it to the mover on the remote server with an efficient transport protocol (RDMA and DPDK are preferred, with TCP/IP as the backup) (❾). The remote mover finally forwards a full `RHead` segment to the destination GPU (❿).

All communication optimizations are incorporated in the pipelines:

- Opt 1: operations generated by MGI's device API, i.e., `Send` (❶) and `Recv` (❼), are on-device and non-blocking.
- Opt 2: every stage that transfers data in the pipeline, i.e., ❷, ❹, ❽, ❾, or ❿, transfers a batch of tuples in a segment.
- Opt 3: when multiple next hops exist in the forwarding table entry (❸ and ❾), all the sort buffers are utilized.
- Opt 4: when the forwarding table shows the next hop of a GPU sort buffer is a remote GPU, a mover directly writes the data to the destination GPU, bypassing ❽ and ❿.

Dynamic Buffer and Path Selection. When a host mover schedules transfers (❹ and ❽), it dynamically selects the sort buffer and the next hop (when multiple paths are present) based on their occupancies, specifically, whether a sort buffer has a full segment and whether a next hop's receive buffer has an empty segment. When the transfer workload is skewed, i.e., a small subset of destinations are significantly more popular than others, MGI's dynamic buffer and path selection can effectively prioritize the sort buffers for the popular destinations. Traffic can also be effectively distributed across multiple paths based on their currently available capacities.

Extensibility. MGI can be extended to incorporate future optimizations at two levels. Device- or link-level optimizations (e.g., data (de)compression) can be integrated in the data plane without affecting the control plane. Global, infrastructure-level optimizations (e.g., on-path processing with SmartNICs) would need to be reflected in the corresponding components of both planes (e.g., TO and new executors). We leave these investigations to future work.

6 EVALUATION

6.1 Methodology

Workloads. We implement `Exchange` with MGI where user kernels on multiple GPUs partition and shuffle data. We further implement `GroupBy` and `Join` atop `Exchange`. Our evaluation also covers various communication patterns (§6.3) and complex queries (§6.5).

Dataset and Performance Metrics. We evaluate the above applications using TPC-H with varying scale factors (SFs). More specifically, we run `Exchange` and `GroupBy` on `lineitem` and join `lineitem` and `orders`, two largest tables in the benchmark. Initially, all input tables are randomly distributed across all GPUs and held in GPU memory. The performance metric for `Exchange` is the overall throughput in gigabytes per second (GB/s) of the operation, and that for `GroupBy` and `Join` is the end-to-end execution time.

Compared Approaches. We compare MGI with two widely-adopted GPU communication frameworks and additional baselines to show the benefits of utilizing multiple GPUs as follows.

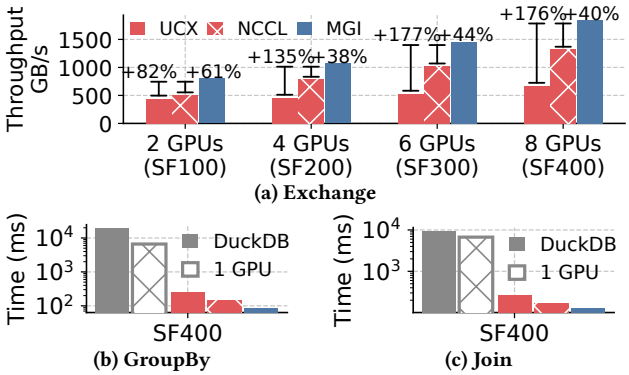
- *NCCL* [46] (v2.27.7), a library from NVIDIA that provides optimized collective operations between GPUs. It can scale out to multiple servers with MPI. We use `ncclSend` and `ncclRecv` to transfer tuples between each pair of GPUs and group calls [49] to let NCCL optimize the communication.
- *UCX* [65] (v1.19), a popular communication framework supporting intra-server and inter-server GPU communication. It provides `send_nbx` and `recv_nbx` for data transfers.
- Additional baselines: (1) *single GPU*, which accelerates each application (radix hash join and aggregation [5]) on a single GPU and spills the state to host memory when GPU memory is insufficient, and (2) *DuckDB* [55] (v1.4.0), an efficient single-machine data analytical system on CPUs.

We further compare MGI with recent data processing proposals [53, 68] for complex queries (§6.5).

Hardware. Host experiments are run on a cluster where each server has an AMD EPYC 9655 2.6GHz 96-core CPU, 750 GB of DDR5 memory, and 7.8 TB of NVMe SSD. We use recent mass-production GPUs from NVIDIA (e.g., A100 [44] and H100 [45]). Specific GPU infrastructures are explained in each experiment.

Table 5: GPU testbeds in the evaluation.

Testbed	GPU	#GPUs/Server	#Servers	Intra-server GPU Links	Inter-server Links
Testbed1	A100 (108 SMs, 80 GB HBM2e)	8	1	NVLink 3.0 ×12 = 600 GB/s	N/A
Testbed2	H100 (114 SMs, 80 GB HBM3)	4	1	NVLink 4.0 ×6 = 300 GB/s	N/A
Testbed3	H100 (114 SMs, 80 GB HBM3)	4	3	NVLink 4.0 ×6 = 300 GB/s	ConnectX-7 ×4 = 800 Gbps



(a) Exchange (b) GroupBy (c) Join

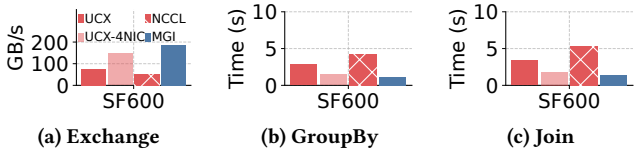
Figure 10: Scale-up performance of different data processing tasks. Higher is better in (a), and lower is better in (b)(c).

6.2 Performance of Multi-GPU Data Processing

Scale-up Performance. Testbed1 in Table 5 has 8 A100 GPUs, each with 108 SMs and 80 GB HBM2e, providing 1935 GB/s bandwidth. GPUs are connected with 12 NVLinks 3.0 (up to 600 GB/s bandwidth) and NVSwitches—same topology as in Figure 2g.

We first compare `Exchange` throughput with MGI to that with UCX and NCCL. Each GPU holds a shard of ~50 GB data in memory and shuffles the tuples by `orderkey`. Figure 10a shows the result. We first observe that the shuffle throughput increases in all communication frameworks with the GPU count due to more NVLinks to utilize and thus higher aggregated bandwidth. Between NCCL and UCX, the former is more efficient and shows better scalability because `Exchange` is essentially an all-to-all operation that NCCL optimizes. Finally, `Exchange` throughput with MGI is significantly higher than existing frameworks: *it is up to 177% and 61% higher than UCX and NCCL, respectively*. In this setup, MGI's benefits primarily originate from pipelining the two steps in `Exchange`: data partitioning and inter-GPU transfers. With UCX and NCCL, the application needs to first partition the tuples on GPUs before calling their communication APIs on hosts to send tuples to destinations.

On top of `Exchange`, we further run `GroupBy` and `Join`. Specifically, in the `GroupBy` experiment, we group the tuples in `lineitem` in SF 400 and apply a `sum` aggregation in each group. Figure 10b compares the running time of `GroupBy` on eight GPUs with MGI, NCCL, and UCX. This experiment also includes the CPU and single-GPU (A100) baselines. First, the CPU baselines are significantly slower than the GPU solutions: DuckDB is 2.8× slower than the single-GPU solution and 77× slower than the multi-GPU solution with UCX. Comparing GPU solutions, single-GPU processing is an order of magnitude slower than scaling to multi-GPUs because the memory of a single GPU is limited and the processing is bottlenecked by data movement over the PCIe bus: `GroupBy` on a single GPU is 27× slower than UCX on eight GPUs. Finally, *between multi-GPU approaches, MGI is most efficient: it is 2.9× faster than UCX and 1.7× faster than NCCL. With MGI, multi-GPU data processing performance is maximized: it is 226× faster than DuckDB and 80× faster than the single-GPU solution.*



(a) Exchange (b) GroupBy (c) Join

Figure 11: Scale-out performance of different data processing tasks. Higher is better in (a), and lower is better in (b)(c).

We make similar findings in `Join` evaluation. Figure 10c reports the performance of `Join` between `lineitem` and `orders` on `orderkey`, which is similar to `GroupBy`. This result validates the communication bottleneck in multi-GPU data processing: once data is transferred to the destination, the GPU can speed up the processing to eliminate compute bottlenecks.

Scale-out Performance. To assess MGI's ability to scale out to multiple GPU servers, we evaluate these data processing operations in Testbed3 in Table 5, which consists of three GPU servers. Each server is equipped with 4 H100 SXM5 GPUs (114 SMs and 80 GB HBM3 memory 3072 GB/s). Each GPU is directly connected to other three GPUs with 6 NVLinks 4.0 providing up to 300 GB/s. For inter-server connectivity, each server has four ConnectX-7 NICs that in total provide 800 Gbps network bandwidth.

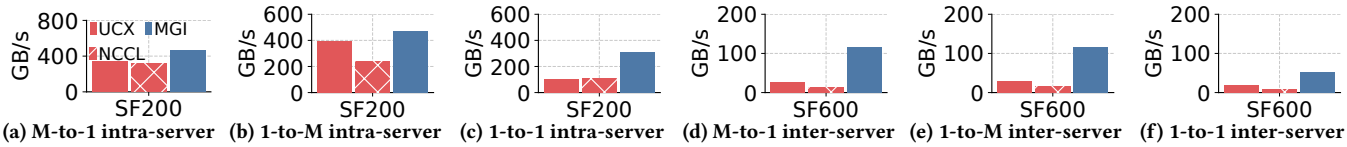
Figure 11 shows the multi-server performance of `Exchange`, `GroupBy`, and `Join` with UCX, NCCL, and MGI. Generally, as in the scale-up evaluation, MGI remains most efficient: it provides 242% and 355% higher `Exchange` throughput than UCX and NCCL, respectively, due to its ability to utilize all available NICs. UCX 1-NIC has better support for RDMA and is more performant than NCCL. Although UCX can be configured to use four NICs (UCX-4NIC), it requires manual setup and still provides lower throughput than MGI. MGI's efficiency further benefits `GroupBy` and `Join` (on average 3.7× faster than NCCL and 2.4× faster than UCX).

6.3 Varying Communication Patterns

The `Exchange` operator represents a *many-to-many* communication pattern. To evaluate MGI's ability to support other communication patterns, we run the following experiments:

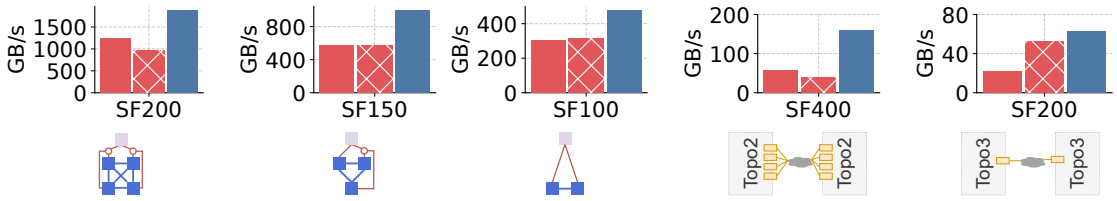
- *Many-to-one*: each endpoint but the first holds a partition of the `lineitem` table and sends $1/N$ of its tuples to the first endpoint, where N is the total endpoints.
- *One-to-many*: the first endpoint partitions its tuples and sends a partition to every other endpoint.
- *One-to-one* (point-to-point communication): the first endpoint sends tuples to an endpoint on a different server.

We evaluate these patterns in both scale-up (Testbed2 in Table 5 that consists of four H100 GPUs inter-connected by NVLinks) and scale-out (Testbed 3) setups. Figures 12a–12c show the throughput



(a) M-to-1 intra-server (b) 1-to-M intra-server (c) 1-to-1 intra-server (d) M-to-1 inter-server (e) 1-to-M inter-server (f) 1-to-1 inter-server

Figure 12: MGI's performance with various communication patterns. Legends are consistent across figures.



(a) Additional Topo1 (b) Additional Topo2 (c) Additional Topo3 (d) Additional Topo4 (e) Additional Topo5

Figure 13: MGI's performance with various inter-GPU topologies. Legends are consistent with Figure 12.

of the three different communication patterns on 4 GPUs of the same server, and Figures 12d–12f show the throughput on 12 GPUs across 3 servers. MGI outperforms the other two frameworks in all cases: 35% (413%) and 47% (991%) higher than UCX and NCCL for scale-up (scale-out) many-to-one communication; 18% (414%) and 99% (789%) higher than UCX and NCCL for scale-up (scale-out) one-to-many communication; and finally, for scale-up (scale-out) point-to-point communication, MGI achieves 200%+ (260%+) higher throughput than both frameworks due to its ability to utilize multiple NVLink paths (RDMA network paths), i.e., Opt 3.

6.4 Varying Inter-GPU Topologies

We next evaluate MGI in different inter-GPU topologies. In addition to the topology in Section 6.2, we generate three more scale-up topologies using resources in Testbed2 in Table 5 and two more scale-out topologies using resources in Testbed3, together representing a spectrum of common topologies in GPU infrastructures.

- *Topo1*: four H100s fully-connected by NVLinks and PCIe.
- *Topo2*: three H100s fully-connected by NVLinks and PCIe.
- *Topo3*: two H100s connected by NVLinks and PCIe.
- *Topo4*: two servers, each internally adopting Topo2 and inter-connected by four NICs.
- *Topo5*: two servers, each internally adopting Topo3 and inter-connected by a single NIC.

Figures 13a–13e (bottom) visualize these topologies. They stress a communication framework in different aspects, e.g., Topo1 tests if it can fully utilize , and Topo3 tests the smallest possible multi-GPU setup. Topo4 and Topo5 test different network conditions.

Figures 13a–13e (top) show how well MGI, UCX, and NCCL can adapt to these topologies. We run `Exchange` on `lineitem` with a scale factor that can fit in the aggregated memory of the GPUs in each topology. We observe that MGI can efficiently adapt to each topology and consistently achieves the highest throughput: in intra-server setups, i.e., Topo1–Topo3, MGI is up to 50% and 94% faster than UCX and NCCL; in inter-server setups, when there are multiple NICs, i.e., Topo4, MGI achieves 274% and 416% higher throughput than UCX and NCCL; although its improvement is reduced in Topo5 as the multi-NIC optimization is inapplicable, it remains the best-performing option.

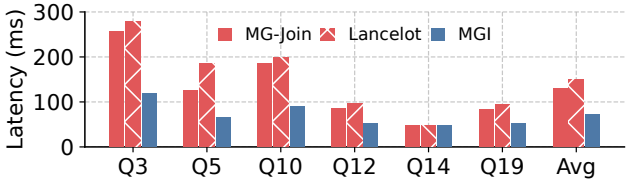


Figure 14: Performance comparison with TPC-H queries.

6.5 Comparing with Data Processing Proposals

To evaluate MGI's benefits for end-to-end query processing and directly compare it with recent proposals for multi-GPU data processing, we implement TPC-H queries on top of MGI and compare the performance with that of MG-Join [53] and Lancelot [68], two single-server multi-GPU data processing proposals. Specifically, we use the six TPC-H queries in the evaluation of MG-Join: Q3, Q5, Q10, Q12, Q14, and Q19 [53]. For each query, we implement the physical plan generated by DuckDB with CUDA kernels and replace `Join` and `GroupBy` that require cross-GPU communication with the ones implemented with MGI in Section 6.1. We measure the performance of each query on scale factor 400 in Testbed1 (eight A100s) and compare it with the performance of MG-Join and Lancelot on the same workload and hardware.

Figure 14 reports the latency of each query achieved by the three approaches. We observe that these queries represent a range of characteristics: some queries shuffle significant amounts data between GPUs (e.g., 97 GB in Q3), while some barely trigger inter-GPU communication (e.g., 2 GB in Q14). MG-Join, benefiting from its inter-GPU communication optimizations, is on average 1.2× faster than Lancelot. With its control plane, MGI automatically applies and tunes all applicable optimizations, efficiently executed by its data plane, for the deployed infrastructure and achieves higher performance than both approaches: on average 1.8× faster than MG-Join and 2.1× faster than Lancelot.

6.6 Focused Experiments

Data Transfer Primitives. MGI transfers data between local GPUs with host-issued `Memcpy`. Alternatively, cross-GPU data transfers can also be implemented with on-device load/store instructions. To show the difference, we transfer tuples between two A100 GPUs in Testbed1 using the two approaches and measure their peak throughput and consumed SMs with different buffer sizes.

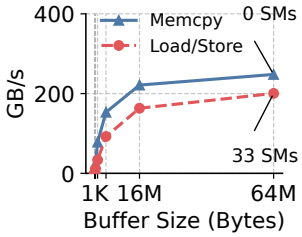


Figure 15: Transfer primitives

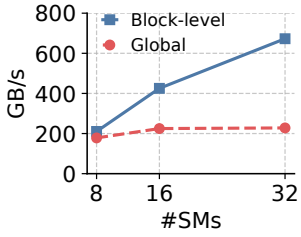


Figure 16: Thread sync.

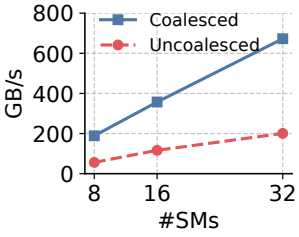


Figure 17: Access patterns

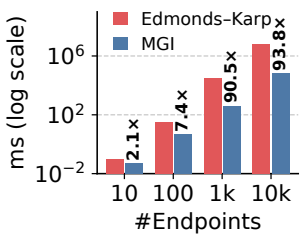


Figure 18: Channel creation

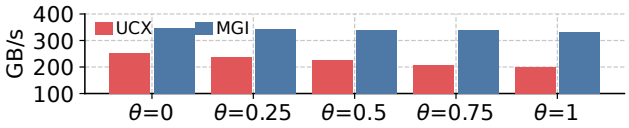


Figure 19: Communication performance under skewness.

Figure 15 shows the result. `Memcpy` can achieve 20%+ higher throughput than the load/store instructions with the same buffer size. In addition, the latter consumes significant GPU SMs to achieve high data transfer throughput (33 SMs for the peak throughput). Therefore, MGI's selected primitive can provide higher data transfer performance without consuming GPU resources.

Thread Synchronization on GPUs. MGI allocates separate buffers on a GPU for different thread blocks and synchronizes threads in each block with `atomicAdd`. Alternatively, it can allocate a single buffer in the global memory and synchronizes threads in all blocks. To evaluate the difference, we measure the throughput of appending 64-byte tuples to the message buffers in global memory on an A100. As Figure 16 shows, with global atomic operations the throughput barely increases as more SMs are utilized for sorters. In comparison, with per-block buffers and intra-block synchronization, the transfer speed scales much better with SMs.

Memory Access Patterns. MGI coalesces memory accesses from threads to buffers on GPUs, i.e., memory accesses target contiguous locations in the global memory to fully utilize memory channels. To show how it matters, we run the same workload as in the previous experiment but with different memory access patterns: uncoalesced accesses, where threads from the same block access disjoint memory locations, and coalesced accesses. The difference is significant (Figure 17): with the same SMs, coalescing brings a 3.4× speedup for global memory accesses compared to uncoalesced accesses.

Channel Creation. To show the performance of MGI's control plane, we measure the time of channel creation at different scales (numbers of endpoints/GPUs). As shown in Figure 18, compared to the baseline using Edmonds-Karp, the hierarchical and incremental TO (§4.3) of MGI drastically reduces the global optimization time and scales better with the size of the GPU infrastructure. For example, with 100 GPUs, while the baseline takes 34 ms to create a channel, MGI is 7.4× faster with 5 ms only.

Handling Skewness. Recall from Section 5.4 that MGI is immune to data skewness. To verify this, we create a broadcast channel with one source and four destinations in Testbed2 (Topo1 in §6.4 where multiple paths present between any pair of GPUs), and the source GPU loads the TPC-H `lineitem` table (SF 50). We use an additional attribute as the partition key and generate values following Zipfian [22] with varying skewness factor θ. The result is reported in Figure 19. When $\theta = 0$, i.e., all destinations receive equal data volumes, both UCX and MGI can fully utilize multiple links. As θ increases, one destination receives significantly more data than the others. In this case, UCX is increasingly bottlenecked by a single path and thus provides lower throughput. MGI, with its dynamic buffer and path selection, maintains the high throughput.

7 RELATED WORK

Accelerating database workloads on GPUs has long been investigated in the community [9, 29, 32, 33, 37, 38, 59–61, 70]. The scope of this work is systems that can scale up or out to multiple GPUs.

Gao et al. [19] scaled distributed hash join to a thousand GPUs with NVIDIA DGX SuperPOD [43]. Thostrup et al. [63] proposed two distributed pipelined hash join algorithms using GPUDirect RDMA. Distributed TQP [67] scales single-GPU data processing to a cluster with NCCL-implemented communication operations.

More efforts have been made to execute database workloads using multiple GPUs on the same server. Rui et al. [58] presented a set of join algorithms for for multi-GPU setups. MG-Join [53] is a distributed join solution that scales up to multiple GPUs on a single server. Maltenberger et al. [39] developed two sorting algorithms targeting multi-GPUs on the same server. Lancelot [68] explores the co-processing between CPU and multi-GPUs on the same server for query execution. Vortex [69] accelerates data analytics on multi-GPUs with decoupled computation and data transfers. HeavyDB [24] is a commercial GPU database system that supports multi-GPU query execution by dividing a query into fragments.

MGI is distinct in its generality as a communication framework to offer communication optimizations that adapt to heterogeneous GPU infrastructures behind an easy-to-use API.

8 CONCLUSION

We presented MGI, a general and fast communication framework natively designed for multi-GPU data processing. It provides an intuitive on-device interface and applies optimizations judiciously. Enabling MGI are a control plane that applies global optimizations with a graph-based network model and a data plane where executors and buffers are carefully designed. Data processing performance with MGI is significantly higher than that with existing general GPU communication frameworks, e.g., NCCL and UCX.

ACKNOWLEDGMENTS

We thank the reviewers for their thoughtful comments, Alireza Shateri for his early contribution to the project, and Maryam Mehri Dehnavi for her generous GPU support. This work was supported in part by NSERC RGPIN-2023-04834 DGECR-2023-00308, Digital Research Alliance of Canada RRG 5846, and Ministry of Education in Singapore AcRF Tier 2 grant No. MOE-T2EP20224-0020.

References

[1] Xin Ai, Hao Yuan, Zeyu Ling, Qiange Wang, Yanfeng Zhang, Zhenbo Fu, Chaoyi Chen, Yu Gu, and Ge Yu. 2024. NeutronTP: Load-Balanced Distributed Full-Graph GNN Training with Tensor Parallelism. *Proc. VLDB Endow.* 18, 2 (2024), 173–186. https://www.vldb.org/pvldb/vol18/p173-ai.pdf

[2] Amazon Web Services. 2025. Recommended GPU Instances. https://docs.aws.amazon.com/dlami/latest/devguide/gpu.html. Accessed: 2026-05-20.

[3] Apache. 2026. ZooKeeper: A Distributed Coordination Service for Distributed Applications. https://zookeeper.apache.org/doc/r3.5.3-beta/zookeeperOver.html. Accessed: 2026-05-20.

[4] Moe Kelly Asya Walters, Andy Walker. 2025. RETHINKING AI DEMAND PART 1: AI DATA CENTERS ARE EXPERIENCING A SURGE OF TRAINING DEMAND - WHAT HAPPENS WHEN THE SURGE IS OVER? https://www.alvarezandmarsal.com/insights/rethinking-ai-demand-part-1-ai-data-centers-are-experiencing-surge-training-demand-what. Accessed: 2026-05-20.

[5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1754–1766. https://doi.org/10.1109/TKDE.2014.2313874

[6] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. 2024. Crux: GPU-Efficient Communication Scheduling for Deep Learning Training. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*. ACM, 1–15. https://doi.org/10.1145/3651890.3672239

[7] Chameleon Cloud. 2025. A configurable experimental environment for large-scale edge to cloud research. https://www.chameleoncloud.org. Accessed: 2026-05-20.

[8] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang. 2024. MAST: Global Scheduling of ML Training across Geo-Distributed Datacenters at Hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 563–580. https://www.usenix.org/conference/osdi24/presentation/choudhury

[9] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[10] CloudLab. 2025. Flexible, scientific infrastructure for research on the future of cloud computing. https://cloudlab.us. Accessed: 2026-05-20.

[11] CNBC. 2024. Malaysia is emerging as a data center powerhouse amid booming demand from AI. https://www.cnbc.com/2024/06/17/malaysia-emerges-as-asian-data-center-powerhouse-amid-booming-demand.html. Accessed: 2026-05-20.

[12] Data Center Dynamics. 2024. xAI targets one million GPUs for Colossus supercomputer in Memphis. https://www.datacenterdynamics.com/en/news/xai-elon-musk-memphis-colossus-gpu/. Accessed: 2026-05-20.

[13] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, and S. S. Li. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *CoRR* abs/2501.12948 (2025). https://doi.org/10.48550/ARXIV.2501.12948 arXiv:2501.12948

[14] Digital Research Alliance of Canada. 2025. Available Resources. https://alliancecan.ca/en/services/advanced-research-computing/accessing-resources/resource-allocation-competition/available-resources. Accessed: 2026-05-20.

[15] DPDK Programmer's Guide. 2015. Ring Library – DPDK. https://doc.dpdk.org/guides-2.0/prog_guide/ring_lib.html. Accessed: 2026-05-20.

[16] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (April 1972), 248–264. https://doi.org/10.1145/321694.321699

[17] Zhenbo Fu, Xin Ai, Qiange Wang, Yanfeng Zhang, Shizhan Lu, Chaoyi Chen, Chunyu Cao, Hao Yuan, Zeyu Ling, Zhenbo Fu, Yu Gu, Yingyou Wen, and Ge Yu. 2025. NeutronTask: Scalable and Efficient Multi-GPU GNN Training with Task Parallelism. *Proc. VLDB Endow.* 18, 6 (2025), xx–xx.

[18] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference* (Sydney, NSW, Australia) *(ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 57–70. https://doi.org/10.1145/3651890.3672233

[19] Hao Gao and Nikolai Sakharnykh. 2021. 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*. 55–64. http://www.adms-conf.org/2021-camera-ready/gao_adms21.pdf

[20] Google Cloud Platform. 2025. GPU machine types. https://cloud.google.com/compute/docs/gpus. Accessed: 2026-05-20.

[21] Government of Canada. 2024. Canada to drive billions in investments to build domestic AI compute capacity at home. https://www.canada.ca/en/innovation-science-economic-development/news/2024/12/canada-to-drive-billions-in-investments-to-build-domestic-ai-compute-capacity-at-home.html. Accessed: 2026-05-20.

[22] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. 1994. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. 243–252.

[23] Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of 4th Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2011, Newport Beach, CA, USA, March 5, 2011*. ACM, 3. https://doi.org/10.1145/1964179.1964184

[24] Heavy.AI. 2025. A Revolutionary GPU-Accelerated Database & Analytics Platform. https://www.heavy.ai/. Accessed: 2026-05-20.

[25] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. 2024. Characterization of Large Language Model Development in the Datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association, 709–729. https://www.usenix.org/conference/nsdi24/presentation/hu

[26] Intel. 2015. AN 687: Implementing QPI Using the Transceiver Native PHY IP Core in Stratix V Devices. https://www.intel.com/content/www/us/en/docs/programmable/683888/current/qpi-overview.html. Accessed: 2026-05-20.

[27] Matthias Jarke and Jürgen Koch. 1984. Query Optimization in Database Systems. *ACM Comput. Surv.* 16, 2 (1984), 111–152. https://doi.org/10.1145/356924.356928

[28] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10, 000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association, 745–760. https://www.usenix.org/conference/nsdi24/presentation/jiang-ziheng

[29] Marko Kabic, Bowen Wu, Jonas Dann, and Gustavo Alonso. 2025. Powerful GPUs or Fast Interconnects: Analyzing Relational Workloads on Modern GPUs. *Proc. VLDB Endow.* 18, 11 (2025), 4350–4363. https://doi.org/10.14778/3749646.3749698

[30] Taeyoon Kim, ChanHo Park, Mansur Mukimbekov, Heelim Hong, Minseok Kim, Ze Jin, Changdae Kim, Ji-Yong Shin, and Myeongjae Jeon. 2023. FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation. 17, 4 (Dec. 2023), 863–876. https://doi.org/10.14778/3636218.3636238

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 1106–1114. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[32] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[33] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (2025), 4518–4531. https://www.vldb.org/pvldb/vol18/p4518-li.pdf

[34] Linux man-pages. 2025. socket(2) — Linux manual page. https://man7.org/linux/man-pages/man2/socket.2.html. Accessed: 2026-05-20.

[35] Linux Programmer's Manual. 2020. io_uring(7) — Linux manual page. https://www.man7.org/linux/man-pages/man7/io_uring.7.html. Accessed: 2026-05-20.

[36] David P. Luebke, Mark J. Harris, Naga K. Govindaraju, Aaron E. Lefohn, Mike Houston, John D. Owens, Mark Segal, Matthew Papakipos, and Ian Buck. 2006. S07 - GPGPU: general-purpose computation on graphics hardware. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*. ACM Press, 208. https://doi.org/10.1145/1188455.1188672

[37] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1633–1649. https://doi.org/10.1145/3318464.3389705

[38] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. 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*. ACM, 1017–1032. https://doi.org/10.1145/3514221.3517911

[39] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1795–1809. https://doi.org/10.1145/3514221.3517842

[40] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339. https://doi.org/10.14778/1920841.1920886

[41] Microsoft Azure. 2024. NV sub-family GPU accelerated VM size series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nv-family. Accessed: 2026-05-20.

[42] MIT Technology Review. 2025. China built hundreds of AI data centers to catch the AI boom. Now many stand unused. https://www.technologyreview.com/2025/03/26/1113802/china-ai-data-centers-unused/. Accessed: 2026-05-20.

[43] NVIDIA. 2020. NVIDIA DGX SuperPOD. https://images.nvidia.com/aem-dam/Solutions/Data-Center/gated-resources/nvidia-dgx-superpod-a100.pdf. Accessed: 2026-05-20.

[44] NVIDIA. 2025. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/. Accessed: 2026-05-20.

[45] NVIDIA. 2025. NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/. Accessed: 2026-05-20.

[46] NVIDIA Developer. 2025. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl. Accessed: 2026-05-20.

[47] NVIDIA Developer. 2025. NVIDIA GPU Direct. https://developer.nvidia.com/gpudirect. Accessed: 2026-05-20.

[48] NVIDIA Docs Hub. 2017. NVIDIA DGX-1 With Tesla V100 System Architecture. https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf. Accessed: 2026-05-20.

[49] NVIDIA NCCL. 2025. Group Calls. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/groups.html. Accessed: 2026-05-20.

[50] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[51] Oracle Cloud Infrastructure. 2025. GPU Instances. https://www.oracle.com/ca-en/cloud/compute/gpu/. Accessed: 2026-05-20.

[52] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. 2010. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)* 29, 4 (2010), 1–13.

[53] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. 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*. ACM, 1413–1425. https://doi.org/10.1145/3448016.3457254

[54] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. 2024. Alibaba HPN: A Data Center Network for Large Language Model Training. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*. ACM, 691–706. https://doi.org/10.1145/3651890.3672265

[55] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[56] Reuters. 2025. Meta to spend up to $65 billion this year to power AI goals. https://www.reuters.com/technology/meta-invest-up-65-bln-capital-expenditure-this-year-2025-01-24/. Accessed: 2026-05-20.

[57] Reuters. 2025. Trump announces private-sector $500 billion investment in AI infrastructure. https://www.reuters.com/technology/artificial-intelligence/trump-announce-private-sector-ai-infrastructure-investment-cbs-reports-2025-01-21/. Accessed: 2026-05-20.

[58] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (2020), 708–720. https://doi.org/10.14778/3436905.3436927

[59] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 351–362. https://doi.org/10.1145/1807167.1807207

[60] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[61] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 417–432. https://doi.org/10.1145/3035918.3064043

[62] David Tarditi, Sidd Puri, and Jose Oglesby. 2006. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 325–335. https://doi.org/10.1145/1168857.1168898

[63] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proc. ACM Manag. Data* 1, 1 (2023), 29:1–29:26. https://doi.org/10.1145/3588709

[64] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: the data flow interface for high-speed networks. In *Proceedings of the 2021 International Conference on Management of Data*. 1825–1837.

[65] UCF Consortium. 2025. Unified Communication X. https://openucx.org. Accessed: 2026-05-20.

[66] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. 2023. HongTu: Scalable Full-Graph GNN Training on Multiple GPUs. *Proc. ACM Manag. Data* 1, 4 (2023), 246:1–246:27. https://doi.org/10.1145/3626733

[67] Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-Scale Analytics in the Blink of an Eye. *Proc. VLDB Endow.* 19, 2 (2025), 141–155. https://www.vldb.org/pvldb/vol19/p141-sen.pdf

[68] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2024. Scaling your Hybrid CPU-GPU DBMS to Multiple GPUs. *Proc. VLDB Endow.* 17, 13 (2024), 4709–4722. https://www.vldb.org/pvldb/vol17/p4709-yogatama.pdf

[69] Yichao Yuan, Advait Iyer, Lin Ma, and Nishil Talati. 2025. Vortex: Overcoming Memory Capacity Limitations in GPU-Accelerated Large-Scale Data Analytics. *Proc. VLDB Endow.* 18, 4 (May 2025), 1250–1263. https://doi.org/10.14778/3717755.3717780

[70] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210

[71] Qizhen Zhang, Philip A. Bernstein, Badrish Chandramouli, Jason Hu, and Yiming Zheng. 2024. DDS: DPU-optimized Disaggregated Storage. *Proc. VLDB Endow.* 17, 11 (2024), 3304–3317. https://doi.org/10.14778/3681954.3682002

[72] Xiaoke Zhu, Min Xie, Ting Deng, and Qi Zhang. 2024. HyperBlocker: Accelerating Rule-based Blocking in Entity Resolution using GPUs. *Proc. VLDB Endow.* 18, 2 (2024), 308–321. https://www.vldb.org/pvldb/vol18/p308-xie.pdf