MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures

Johns Paul
National University of Singapore
Singapore

Shengliang Lu
National University of Singapore
Singapore

Bingsheng He
National University of Singapore
Singapore

Chiew Tong Lau
Nanyang Technological University
Singapore

ABSTRACT

The recent scale-up of GPU hardware through the integration of multiple GPUs into a single machine and the introduction of higher bandwidth interconnects like NVLink 2.0 have enabled new opportunities of relational query processing on multiple GPUs. However, due to the unique characteristics of GPUs and the interconnects, existing hash join implementations spend up to 66% of their execution time moving the data between the GPUs and achieve lower than 50% utilization of the newer high bandwidth interconnects. This leads to extremely poor scalability of hash join performance on multiple GPUs, which can be slower than the performance on a single GPU. In this paper, we propose MG-Join, a scalable partitioned hash join implementation on multiple GPUs of a single machine. In order to effectively improve the bandwidth utilization, we develop a novel multi-hop routing for cross-GPU communication that adaptively chooses the efficient route for each data flow to minimize congestion. Our experiments on the DGX-1 machine show that MG-Join helps significantly reduce the communication overhead and achieves up to 97% utilization of the bisection bandwidth of the interconnects, resulting in significantly better scalability. Overall, MG-Join outperforms the state-of-the-art hash join implementations by up to 2.5x. MG-Join further helps improve the overall performance of TPC-H queries by up to 4.5x over multi-GPU version of an open-source commercial GPU database Omnisci.

ACM Reference Format:
Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457254

1 INTRODUCTION

The use of higher bandwidth global memory and the availability of larger number of parallel cores have made GPUs the ideal accelerator for in-memory relational query processing [31, 32, 38, 43]. Typically, a single GPU offers 10x higher global memory bandwidth and 100x more cores than a CPU. However, the limited global memory size of GPUs combined with the low bandwidth of CPU-GPU and GPU-GPU interconnects has held back their adoptions for large data sets that do not fit within the global memory of a single GPU [16]. In recent years, we have witnessed the rise of emerging multi-GPU architectures in a single machine. In fact, NVIDIA alone has released five different multi-GPU server configurations. First, the integration of multiple GPUs (up to 20) into a single machine means that it is possible to have hundreds of gigabytes of GPU memory. For example, a machine with 16 GPUs each containing 32 GB of global memory offers a total of 512 GB GPU memory, which is sufficient for many online analytical workloads. Second, the introduction of new high-bandwidth and low-latency interconnects such as NVLink 2.0 significantly resolves the communication overhead among GPUs.

The scale-up multi-GPU architecture has enabled new opportunities of relational query processing on multiple GPUs. Overall, the multi-GPU architecture has paved the way for GPUs becoming primary computing devices for relational query processing on large data sets. Hence, in this study we focus on scenarios where the GPU is being used as a primary computing device and the input data is distributed over the GPUs. In spite of many previous studies on query processing on a single GPU [9, 13, 22–24, 32, 33, 37, 47], there has been little work on studying databases, especially hash joins on these multi-GPU architectures.

To demonstrate this, we study the performance of hash join, whose input relations are equally distributed to multiple GPUs. Specifically, we study two hash join designs for multi-GPU architectures, including UMJ [31] and DPRJ [21]. UMJ makes use of the NVIDIA unified memory feature to move the data between the GPUs, and DPRJ is designed based on the current state-of-the-art join implementation available for distributed GPU architectures. We present the GPU cycles per tuple along with the breakdown for computation and data transfer when executing UMJ and DPRJ on the DGX-1 server from NVIDIA [5]. We use two input relations (with 8-byte tuples) and a join selectivity of 100% for this test. When we increase the number of GPUs, we increase the size of both input relations by keeping 512M[1] tuples of each relation on each GPU. We study the cross-GPU communication and computation time components for DPRJ from NVIDIA GPU profiler (denoted as "DataTransfer" and "Computation", respectively), and we were not able to get such information from the profiler for UMJ. More experimental details can be found in Section 5.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457254

[1] M = 1,048,576

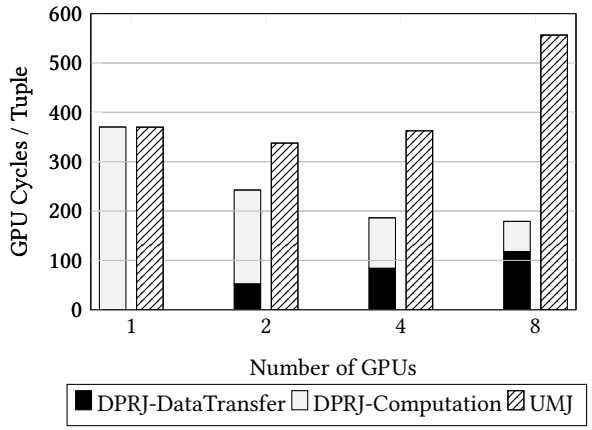


Figure 1: Join performance and execution time break down of partitioned hash join on DGX-1 server from NVIDIA.

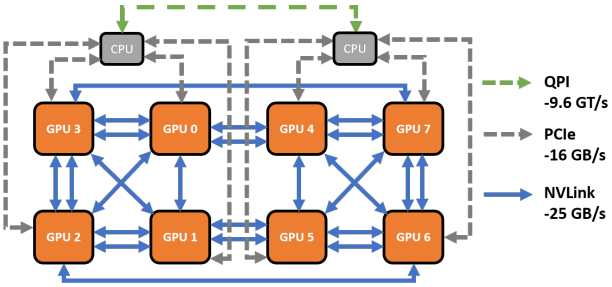


Figure 2: Architecture design of DGX-1.

The results of this study are shown in Figure 1. Both UMJ and DRPJ encounter poor scalability when the number of GPUs is increased from one to eight. In fact, the DPRJ implementation only achieves a 2.13x improvement when the number of GPUs increases from one to eight. This is due to the severe increase in the communication overhead with the increasing number of GPUs, which accounts for up to 66% of the total execution time. Our detailed study further shows that it achieves lower than 50% utilization of the bisection bandwidth of the interconnects (details in Section 5). UMJ on eight GPUs is even slower than on a single GPU. *Overall, the results clearly demonstrate that existing partitioned hash join implementations encounter high communication overhead and achieve poor scalability when executed on modern multi-GPU architectures.*

To understand the reason behind this inefficiency, we study the architecture design of the DGX-1 machine in Figure 2, and make the following observations. First, GPUs in the DGX-1 machine are connected over a combination of PCIe 3.0, NVLink 2.0 and QPI interconnects, resulting in a highly heterogeneous interconnect network design. This highly heterogeneous interconnect design leads to the data movement between certain pairs of GPUs becoming a bottleneck when performing operations like hash join on GPUs. Second, slower PCIe links are shared across multiple GPUs causing severe congestion when multiple GPUs try to gain simultaneous accesses to the same PCIe interconnect hardware. For example, when joining the input relations over 8 GPUs the PCIe 3.0 links between the GPUs with a bandwidth of 16 GB/s becomes the bottleneck.

It is a non-trivial task to reduce the communication cost of hash joins on such multi-GPU architectures. The communication overhead encountered by the hash join implementations on modern multi-GPU architectures depends on many factors like 1) hardware configuration, 2) input data distribution and 3) the data transfer characteristics (e.g., size of data flows). Determining the optimal data transfer schedule and route for each data flow given a set of constraints can be reduced to the classic multi-commodity flow problem, which cannot be solved in a reasonable amount of time, even for eight GPUs [14, 17, 42].

In this study, we propose MG-Join, a scalable partitioned hash join implementation that is optimized for modern multi-GPU architectures. To ensure efficient distribution of communication across the interconnect links, MG-Join adopts a multi-hop routing that is capable of routing data flows over intermediate GPU nodes. Further, through the use of an adaptive policy that takes into consideration the hardware configuration, the input data distribution and the characteristics of different data flows, MG-Join is able to dynamically re-route data flows to minimize the congestion.

To summarize, the major contributions of this work are:

- We argue that multi-GPU architectures enable new opportunities for scale GPU-based query processing on hundreds of gigabytes of GPU memory. However, we study the performance of partitioned hash joins on multi-GPU architectures, and demonstrate that the cross-GPU communication overhead can severely hinder the scalability.
- We develop MG-Join, a scalable partitioned hash join implementation for modern multi-GPU architectures that makes use of the proposed adaptive multi-hop routing policy. The adaptive multi-hop data routing policy helps minimize congestion across shared interconnect links and ensures more efficient use of GPU hardware and interconnect links.
- We further conduct in-depth experiments to demonstrate the efficiency of MG-Join in joining large data sets. Our experiments find that 1) MG-Join achieves up to 97% utilization of the bisection bandwidth of the GPU interconnect links, and 2) MG-Join achieves up to 2.5x performance improvement over existing multi-GPU hash join implementations [21, 31]. Moreover, MG-Join helps improve the overall performance of TPC-H queries by up to 4.5x over multi-GPU version of an open-source commercial GPU database, Omnisci [29].

The rest of this paper is organized as follows. In Section 2, we present the background on modern multi-GPU architectures. We then present the design overview of MG-Join in Section 3. We present the details of the novel data distribution step in MG-Join in Section 4. We present the experiments in Section 5 and detail the related work in Section 6. Finally, we conclude in Section 7.

2 MODERN MULTI-GPU ARCHITECTURES

In this section, we present background on the design of GPU and interconnect hardware. In this study, we primarily focus on hardware from NVIDIA, which is a major GPU vendor for scale-up hardware.

2.1 GPU Hardware

A single GPU consists of multiple *streaming multiprocessors* (SMs), each of which consists of multiple cores. All cores in an SM share

resources like the registers and shared memory among each other. A GPUs has an L2 cache and a global memory that are shared among all the SMs.

A program executing on the GPU is known as a kernel. A kernel is executed as a grid of thread blocks, which can further be broken down into warps (groups of 32 threads). Each thread block is assigned to a single SM and the cores inside each SM execute the threads in a SIMD fashion, at the granularity of a single warp. Finally, the shared memory is shared within the threads in a thread block but cannot be shared across different thread blocks..

Modern GPUs support features like unified memory and hardware support for handling page faults at runtime. These features abstract away the physical location of the data within multiple GPUs and make the memory buffers available to the kernels executing on all GPUs. Further, when a piece of data located in one GPU is accessed by a kernel executing on another GPU, a page fault is encountered and the data is moved to the GPU that requires the data. This movement of data is handled by the GPU driver, thus presenting a simplified unified memory model to GPU programmers. However, due to the high overhead of page faults [31] on GPUs, unified memory buffers can lead to a significant degradation in the performance of hash joins in multi-GPU architectures, as demonstrated in our experimental studies. That motivates us to develop efficient cross-GPU communications for hash joins.

2.2 GPU Interconnects

As demonstrated in the DGX-1 machine in Figure 2, the GPUs in modern multi-GPU architectures are linked together through a set of heterogeneous interconnects (*links*). In the following, we briefly introduce the interconnects/links that are widely used in modern multi-GPU architectures.

PCIe: PCIe is a bi-directional serial communication bus and can be used for connecting GPUs to either CPUs or other GPUs. A single PCIe bus consists of multiple bi-directional lanes (each Generation 3 PCIe lane capable of achieving a peak theoretical unidirectional bandwidth of 1 GB/s). Modern GPUs often have a single PCIe interface consisting of 16 PCIe lanes, thus achieving a net unidirectional bandwidth of 16 GB/s. Note, multiple GPUs could share the same PCIe bus (using a PCIe bridge), in which case the bandwidth of the bus will be divided among the GPUs during trying to access the bus simultaneously. For example, the DGX-1 server only has 40 PCIe lanes, which are shared among up to 4 GPUs (64 lanes are required for isolated accesses from 4 GPUs).

NVLink: The NVLink interconnect is based on High-Speed Signaling Interconnect (NVHS) [2] and supports CPU-GPU (e.g., IBM Power machine [7]) and GPU-GPU communication. Unlike PCIe, NVLink is an exclusive P2P interconnect and cannot be shared by multiple GPUs. Similar to PCIe, NVLink is bidirectional and each link has 2 sub-links (one in each direction). Version 2 of NVLink interconnect offers bandwidth of up to 25 GB/s per link. Modern NVIDIA hardware supports 4-8 NVLink links per GPU, allowing each GPU to be directly connected to up to 8 GPUs/CPUs. Further, multiple NVLink links can also be used to connect the same CPU-GPU/GPU-GPU pair. Overall, NVLink offers significantly higher bandwidth and communication efficiency than PCIe.

QPI: Intel's Quick Path Interconnect (QPI) is a P2P interconnect and is widely used for communication between processors in multi-socket/NUMA hardware. Similar to NVLink and PCIe, QPI is a bi-directional interconnect with each link supporting up to 25.6 GB/s bandwidth. Depending on the hardware generation, each processor could have multiple QPI links connecting it to other processors in the system. Even though QPI interconnects do not directly connect two GPUs, they are required for moving data between GPUs that are connected to different CPU sockets using PCIe/NVLink in a multi-socket server like DGX-1. Note, when two GPUs connected to different CPU sockets need to communicate, the data cannot be transferred directly. Instead, the data is first moved from the source GPU to the main memory of the associated CPU socket (over PCIe or NVLink). The data is then moved to the CPU socket of the destination GPU over QPI and finally moved to the destination GPU over NVLink/PCIe. This process is referred to as *staging*. Note, staged data transfers are tiled and pipelined by the GPU driver to maximize efficiency. In our experiment, staging fails to achieve high bandwidth utilization and staged data transfers often achieve significantly lower bandwidth than the peak bandwidth.

Design implications. To summarize, the GPUs in modern multi-GPU architectures are interconnected to each other by heterogeneous links. Each of these links have different access capabilities, efficiency, bandwidth, and latency characteristics. Direct transfer (i.e., data transfer in a direct route) between the source and destination GPUs can be highly inefficient. This is because the lower bandwidth PCIe buses are involved in most direct routes between GPU pairs (i.e., in 63% of all direct routes in the DGX-1 machine). Even worse, severe congestion can happen when there are multiple data flows trying to access those interfaces simultaneously. With modern multi-GPU architectures containing up to 16 GPUs capable of communicating simultaneously, hash joins, especially in the data shuffling across GPUs, can lead to severe link congestion, resulting in poor utilization of the interconnect network hardware. Thus, there is a need to dynamically choose the optimal route for each data flow taking into consideration the data flows from other GPUs.

This paper focuses on the server with multiple GPUs with the same configuration. Further, irrespective of the heterogeneity among GPU cards, cross-GPU communication interface is the major bottleneck. Thus, heterogeneity in GPU compute hardware will have a secondary impact on the overall performance. Also, when we work with hardware vendors, they do not recommend a single server with GPUs in different generations, as servers are usually optimized for a certain generation of hardware components and also the GPU compute hardware heterogeneity within a single server could increase the complexity of algorithmic design and load balancing.

3 MG-JOIN: DESIGN OVERVIEW

In this section, we present the design overview of MG-Join, a novel partitioned hash join implementation designed for modern multi-GPU architectures. MG-Join is designed for joining large input relations distributed across multiple GPUs in the same machine.

In this study we specifically focus on equi-joins. However, this does not limit the use of the multi-hop data transmission or adaptive routing for other implementations such as the cartesian products.

This is because, these techniques optimizes the data transfer irrespective of type of operation that is being performed.

3.1 Design Rationales

We have the following rationales when designing MG-Join for modern multi-GPU architectures.

Rationale 1: To ensure efficient data transfer between GPUs, we make use of an adaptive and multi-hop data routing policy that is capable of dynamically re-routing each data flow over the most efficient route between each source and destination GPU pair. The decision is adaptive to each data flow to reduce the congestion that happens at runtime.

Rationale 2: MG-Join breaks down data flows of cross-GPU communication into fine-grained packets of a cost-effective size. On the one hand, a packet can be transmitted as soon as it is generated. This is to maximize the overlap between computation and data transfer. On the other hand, the fine-grained packet size can reduce the pressure of memory usage on the GPU in our proposed multi-hop data transfer.

Rationale 3: We decide to compute a histogram at the beginning of the join to gain the data distribution associated with partitioning information. On the one hand, the histogram computation is rather lightweight, in comparison with cross-GPU communication. Paul et al. [31] has demonstrated that the histogram based partitioning is capable of achieving similar performance to the histogram-free approach [44] when the data is already available on the GPU. On the other hand, histograms are useful in the optimization of multiple phases in our proposed join design.

Rationale 4: Last but not the least, we avoid reinventing the wheel and reuse/adapt state-of-the-art implementations of GPU accelerations in hash joins whenever appropriate. We carefully select the right optimization and design for MG-Join, given that a lot of previous studies have been conducted on GPU-based hash joins [9, 13, 22–24, 33, 37, 47]. For example, we choose the histogram-free local join [44] on each GPU to maximize the overlap between communication and computation, while we choose to generate the histogram at the beginning of MG-Join.

In this study, we specifically focus on equi-joins. However, this does not limit the use of the multi-hop data transmission or adaptive routing for other implementations such as the cartesian products. This is because, these techniques optimize the data transfer irrespective of type of join operations that are being performed.

3.2 Design Overview

In the MG-Join implementation, joining two large input relations, R and S, distributed over multiple GPUs involve the following four phases: *Histogram Generation*, *Global Partitioning Phase*, *Local Partitioning* and *Probe*.

Histogram generation: During the histogram generation phase, each GPU analyzes the input data set that is allocated to it and builds a histogram based on its data. Each entry in the histogram data structure represents the number of tuples in a specific partition and is later used for partitioning the input data among the GPUs.

Although the relations could be partitioned among the GPUs without generating a histogram [44], we choose to generate a histogram according to Rationale 3. Histograms have two major benefits in reducing the communication cost. First, the generation of a histogram before the actual partitioning makes it possible to overlap the compute-intensive partition assignment and the memory-intensive data partitioning during the global partitioning phase. Second, the histogram serves as a simplified bloom filter that avoids unnecessary data transfers during the global partitioning phase.

For generating the histogram, we use the same implementation as Rui et. al. [38] where the histogram is built using the GPU shared memory. This is done to ensure that all the threads have fast access to the histogram in the shared memory. This also means that the maximum number of partitions (P_{max}, also equal to the number of entries in histogram) is limited by the size of the GPU shared memory per SM (M_s), the size of each histogram entry ($\widehat{H_s}$) and the number of thread blocks executing on each SM of the GPU (T_b) as shown by Equation 1.

$$P_{max} = \frac{M_s}{\widehat{H_s} \times T_b} \tag{1}$$

In contrast with the previous studies [13, 23, 24, 37], MG-Join generates the largest number of partitions that can be generated based on Equation 1. The adoption of such a histogram based approach comes with two benefits. First, it helps achieve a more efficient balancing of workload. The partition distribution phase in MG-Join takes care of data skew, including heavy hitters. The use of centralized histogram based approach allows MG-Join to identify and handle heavy hitters such as single-value skew partitions early in execution. This helps minimize any inefficiencies associated with processing data sets with high skew. Second, it helps reduce the amount of work that needs to be done during the local partitioning stage. For example, in a V100 GPU equipped with 64 KB shared memory and requiring at least two thread blocks per SM to ensure efficient use of GPU resources, the histogram kernel can maintain 4,096 4-byte histogram entries in the GPU shared memory.

Global partitioning phase: The global partitioning phase partitions the input relations to ensure that each input tuple is assigned to one or more GPUs and none of the GPUs requires the access of tuples from a remote GPU during the local partitioning and probe phases. This phase involves three steps: 1) the initial partitioning of the input relations on each GPU, 2) assigning the partitions of each GPU to one or more GPUs, and 3) the distribution of the partitioned data based on the partition assignment. In the following, we present more details for 1) and 2), and leave 3) to Section 4.

In Step 1), for partitioning the input data based on the generated local histogram, we use the implementation proposed by Rui et al. [38]. Overall the implementation works as follows. Each thread executing the partitioning kernel reads a unique input tuple, determines the index for the tuple after partitioning based on the histogram and then writes the tuple into the local buffer at the determined index. After partitioning, all the tuples belonging to the same partition will be co-located in the local GPU global memory. We refer readers to the original study [38] for a better understanding of our partitioning implementation.

In Step 2), for the problem of partition assignment, multiple previous studies have explored the optimal solution for the similar problem in distributed CPU nodes [34, 39, 40]. In MG-Join, we adopt the migration and selective broadcast based approach previously proposed by Polychroniou et al. [34]. For each partition, the implementation works by first migrating the tuples of one of the input relations to a subset of the GPUs and then selectively broadcasting the tuples of the other relation to all the nodes containing the tuples of the other relation. We refer the readers to the original implementation [34] as well as the proof of cost optimality for the migration and selective broadcast based system.

Specifically, we have the following three major modifications over [34]. Overall, the modified design takes advantage of the massively parallel architecture of GPUs and quickly computes the network optimal partition assignment.

First, we overlap partition assignment of Step 2) with the partition generation kernel of Step 1) to completely hide its overhead.

Second, in MG-Join, each GPU computes the assignment for all the partitions in parallel based on the histogram. Within each GPU, the partition assignment computation is further parallelized by allowing different warps (groups of 32 threads) to determine the assignment for different partitions in parallel. The assignment for each partition is determined by estimating the benefit of all possible migrations of each relation and then choosing to migrate the relation that achieves the maximum benefit from migration.

Third, as in the original implementation [34], benefit of migrating tuples of a relation from a source GPU to a destination GPU is computed as the difference in the cost of broadcasting all the tuples of the other relation to the source GPU and the cost of migrating the tuples from the source GPU to the destination GPU [34]. In MG-Join, the cost of moving a tuple from one GPU to another is computed as the cost of moving the tuple over the lowest transmission cost path between the GPUs when there is no congestion.

Local partitioning: In this phase, the partitions generated during the global partitioning phase are further partitioned. As detailed in previous studies [38, 44], joining co-partitions on GPUs achieves the peak performance when at least one of the co-partitions fit within the GPU shared memory. Thus, multiple partitioning passes may be required to generate very fine-grained partitions that can fit within the GPU shared memory. Specifically, the data on each GPU received during the data distribution phase is recursively partitioned until the individual partitions are small enough to fit within the GPU shared memory (unless both relations are heavily skewed). To leverage the skew handling techniques, we make use of the partitioning approach proposed by Panagiotis et al. [44].

One remark on choosing the approach by Panagiotis et al. [44] is that this approach does not require the generation of a histogram (unlike that we generate a histogram before the global partitioning phase) and can hence helps achieve more overlap between the data distribution and the local partitioning phase. This demonstrates Rationale 4 that we need to carefully revisit existing GPU-based hash join techniques and choose the suitable ones. Specifically, we start the execution of the data distribution and local partitioning phases together. The local partitioning kernel first begins partitioning the locally available tuples (i.e., tuples that were assigned to the same GPU), followed by the packets arriving from remote GPUs. Since the local partitioning phase does not require a histogram, the packets from the remote GPUs can be processed as soon as they arrive at the destination.

Probe: During the probe phase, the local co-partitions with the same partition id from both R and S relations are joined together. The co-partitions can be joined using nested loop join or hash join (where a hash table is built in the shared memory). Existing literature [44] has demonstrated that both implementations achieve similar performance for most partition sizes. Hence in MG-Join, we simply use the nested loop variant.

4 DATA DISTRIBUTION

We present the design and implementation details of the data distribution mechanism of the global partitioning phase in MG-Join. In the data distribution step of global partitioning, the cross-GPU communications generate data shuffling flows among almost all GPU pairs. The design is rooted at Rationales 1 and 2, presented in Section 3.1. Each data flow is transferred at the granularity of packets, whose route will be determined by the proposed multi-hop adaptive routing policy. We first present the design and implementation of an efficient multi-hop routing, followed by the adaptive routing policy for determining the links in the multi-hop routes.

4.1 Multi-Hop Data Transmission

Existing multi-GPU communications (e.g., [12, 45]) adopt single-hop routing that simply chooses direct route with highest bandwidth (a route that does not require data to pass through any intermediate GPU). Note, in this study, we consider data transfers that are staged in the CPU main memory (when source and destination GPUs are connected to different CPU sockets) as direct transfers as there is no intermediate GPU involved in the data transfer.

As shown in Introduction, the single-hop routing strategy adopted by existing multi-GPU join implementations can often lead to severe congestion in the interconnect links in modern multi-GPU architectures. Thus, in this paper, we propose to use multi-hop transmission for multi-GPU architectures, which offer the flexibility of avoiding congestion and choosing more efficient routes. For each data flow from the source GPU to the destination GPU, data are sent in packets. To multi-hop routing, each packet contains a header with the packet id (4 byte), packet size (4 byte) and a vector of GPU ids (1 byte per entry) representing the route that needs to be taken by the packet. In our experiments, we set the packet size to be 2MB so that it can fully utilize the link bandwidth. The header only adds a negligible overhead per packet.

Challenges: It is challenging to perform efficient multi-hop data transmission on multi-GPU architectures due to the lack of dedicated routing devices and the limited amount of global memory available on individual GPUs. Due to these hardware limitations, each GPU needs to allocate a certain amount of its global memory for routing data packets. The larger the amount of GPU global memory allocated for data routing, the smaller the input data set that can be processed by the GPU. Hence, an efficient multi-hop implementation that makes use of small re-usable routing memory buffers should be designed. Further, to minimize the overall overhead, this re-usable routing buffer based implementation needs to be designed with 1) minimal synchronization across multiple data flows and 2) dynamic and efficient memory allocation on GPUs. In

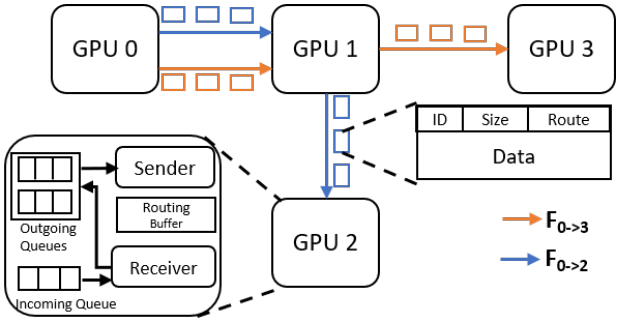


Figure 3: Multi-hop data transmission example in MG-Join.

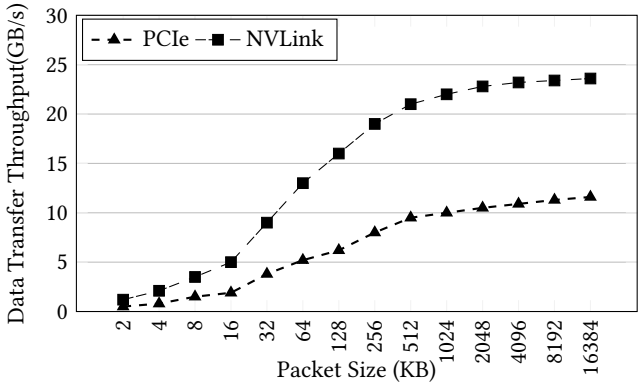


Figure 4: Throughput comparison of NVLink and PCIe interconnects for varying packet sizes.

the remainder of this section, we detail how MG-Join implements multi-hop routing.

Figure 3 shows example of multi-hop routing in MG-Join when routing two data flows ($F_{0\rightarrow2}$ from GPU 0 to GPU 2 and $F_{0\rightarrow3}$ from GPU 0 to GPU 3) on a machine with four GPUs. Now, $F_{0\rightarrow2}$ and $F_{0\rightarrow3}$ takes routes $0 \rightarrow 1 \rightarrow 2$ and $0 \rightarrow 1 \rightarrow 3$, respectively. As shown in the figure, MG-Join adopts a push based routing policy where a source GPU along the route pushes the packet to next destination GPU. Each GPU has two separate modules attached to it as shown in Figure 3: a sender and a receiver. Sender module is responsible for sending out the data packets stored locally; while the receiver module is responsible for either unpacking the received packets destined for that GPU (if it is the final destination of the route) or forwarding packets destined for other GPUs.

Implementation details: At each GPU, we maintain multiple outgoing queues (one for each neighbouring GPU) from which sender module can pick up packets for transmission. Each sender picks up packets from one of the queues in a weighted round-robin fashion. The weight of each queue is set as the number of waiting packets in the queue normalized to the total number of packets waiting to be sent out at the GPU. Further, once a queue is chosen, the sender picks up a batch of packets that follow the same route to the destination and then begins the transmission of these packets. For a batch, all the packets within the same flow can be transmitted out and in quick succession at each sender, ensuring a pipelined transmission of the packets. We experimentally determine the batch size for balancing the pipelined transmission as well as bandwidth utilization. We set the batch size to be eight in our implementation.

We tune packet size and batch size to ensure fine-grained overlapping of data transfer and computation while ensuring efficient use of GPU hardware. Very small data transfer sizes lead to very fine-grained overlapping of data transfer and compute, while it fails to make efficient use of GPU interconnect links. To demonstrate this, we present the bandwidth achieved by different data transfer size on the PCIe and NVLink interconnect in Figure 4, for packet size from 2 KB to 16 MB. As shown in the figure, both NVLink and PCIe are highly inefficient for small packet sizes, with up to 20x performance degradation. However, very large packet sizes can result in in-ability to achieve fine-grained overlapping of data transfer and computation. Further, the performance of the links saturates around 12 MB and does not achieve any improvement beyond that. Finally, the transfer of each batch of packets does come with an associated runtime overhead from the CUDA framework itself.

Taking these factors into account, we use a combination of packet size and batch size to balance between the factors. Using small packet size (e.g., 2MB) allows MG-Join to achieve fine-grained overlapping when large data transfers are not possible, while batching together multiple packets (up to 8) allows MG-Join to maximize interconnect hardware utilization. Note, these settings are determined by profiling interconnects during initialization and then choosing settings that makes efficient use of interconnect hardware. The initiation only needs to run once for the same hardware configuration. In our experiments, we use a packet size of 2MB and the batch size of 8 based on the profiling on the tested machine.

In addition to pushing the packet to the routing buffer at the next GPU, the sender module also appends the pointer to each packet in the incoming queue of the receiver GPU. The receiver module at the GPU that receives the packets will then pick up the pointer from the incoming queue and then either unpacks the data and passes it to the local partitioning phase or appends the pointer to the corresponding outgoing queue if the packet is not destined for the same GPU. The queues only contain the pointers to the packet instead of the actual packet data to avoid redundant copies of the actual data between the queues in the same GPU.

We develop effective buffer management for packet routing. To minimize the synchronization cost, the routing buffer at each GPU is implemented as a collection of buffers, one for each neighboring GPU. Each buffer is implemented as a circular buffer. For the example in Figure 3, the routing buffer at GPU 1 will shared by both $F_{0\rightarrow2}$ and $F_{0\rightarrow3}$. Overall, this design allows MG-Join to 1) reduce the overall size of the routing buffer since the same buffer can be shared among multiple data flows and 2) minimize the synchronization overhead since a data flow only needs to synchronize its buffer access with other data flows arising from the same GPU (i.e no cross GPU synchronizations are required).

To avoid high memory management cost of managing circular buffers accessible by multiple GPUs (sender and receiver), the status of circular buffer is synchronized between GPUs only when sender has no more free slots to push data packets to the receiver. When this happens, sender checks with receiver to determine the number of slots that have been freed by the receiver.

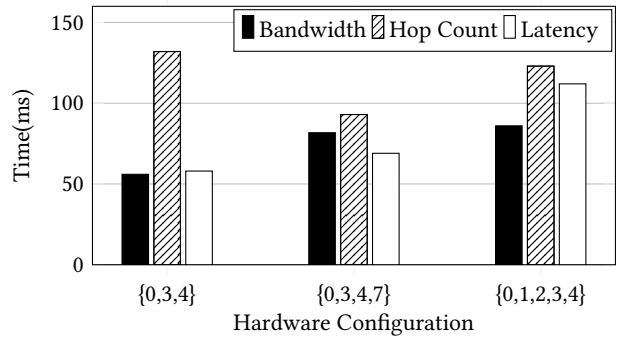
4.2 Adaptive Routing

In the previous section, we explored how multi-hop routing can give more flexible routing choices to the data distribution step of the global partitioning phase. However, there can be many possible routes between each pair of GPUs in modern multi-GPU architectures. For example, there are 64 possible routes without cycles between every pair of GPUs in the DGX-1 machine used in our experiments (Figure 2). Further, the choice of a specific route between a pair of GPUs can impact the choice of routes at other GPUs due to the limited number of DMA engines available per GPU hardware as well as the current data traffic load of each link, making the solution space even larger. In fact, for routing K packets the system need to determine the optimal data transfer schedule from 64^K possible combinations. Hence, designing an efficient routing policy is key to the performance of the data distribution phase. In the following, we first study a number of common static routing policies and identify their inefficiency for the hash join in multi-GPU architectures. This motivates us to design a fine-grained adaptive routing policy.

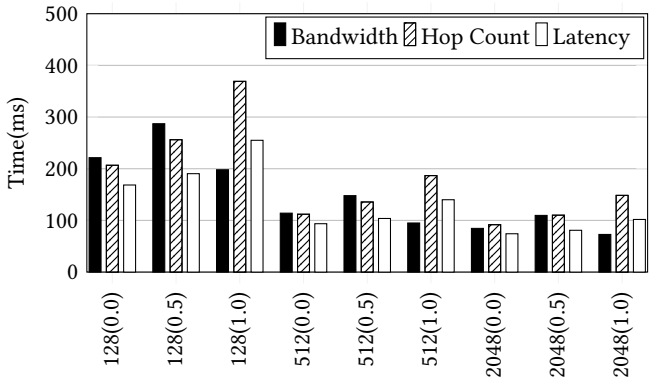
4.2.1 Need for adaptive routing. Determining the global optimal route/schedule for the flows can be reduced to classic multi-commodity flow problem [14, 17, 42]. Hence, it cannot be solved in a reasonable amount of time or scale to more GPUs. This cost can be even more significant for high-performance hardware like multi-GPU architectures. Thus, we need light-weight approaches to determine the route. There have been lightweight static heuristics based on simple metrics like peak bandwidth, latency or hop count. Routing policy using the bandwidth metric will always choose the shortest route with the highest bandwidth to the destination. Further, routing policies using latency and hop count always chooses the route with the lowest latency and the smallest number of hops, respectively.

We study those three commonly used routing metrics under different hardware configurations, data distribution and packet size. In Figure 5a, we study how the different hardware configurations impact the cost of the data distribution step of the global partitioning phase. Note, the results are based on an equi-join of uniformly distributed 1B[2] tuples ($|R|$ and $|S|$ has 512M tuples each) and we use a constant packet size of 2 MB for this experiment. The x-axis lists the identifiers of all the GPUs participating in the MG-Join, i.e., a different combination of numbers/GPU identifiers represent a different underlying network design/hardware configuration. In Figure 5b, we study the impact of different packet sizes and skew (data distribution) on the choice of the routing metric. Note, for this test we use the same input data set and the four GPUs ({0, 3, 4, 7}). $X(Z)$ in the x-axis represents XKB packet size and a Zipf factor of Z for the distribution of the input data.

Overall, those results show that the best static routing policy for routing the data can vary depending on hardware configurations, data distribution and packet size. Generally, bandwidth-based policy performs well for large data flows when executed on a reasonable number of GPUs (4-6). However, the efficiency of long routes begins to fall. Hop count based routing policy avoids long routes. However, the high heterogeneity in the interconnect network design can lead to the hop count based policy picking highly inefficient routes. In contrast, the latency based policy is efficient for small packet sizes



(a) Hardware Configuration.



(b) Data distribution & Packet size.

Figure 5: Impact of hardware configuration, data distribution and packet size on static routing policy (DGX-1).

and tend to achieve the best performance for many reasonable sized configurations. However, it tends to fall back to highly inefficient PCIe interconnects for configurations with a large number of GPUs resulting in high overhead.

In addition to the commonly used static metrics evaluated in this study, the networking community has also proposed other static metrics which are derived from a combination of these metrics [3, 20, 35]. However, these still suffer from the inefficiencies associated with being unable to adapt to network conditions. To summarize, none of the static policies is a clear winner for different cases. Hence, an adaptive framework needs to be developed to ensure efficient use of interconnect hardware in modern multi-GPU architectures.

4.2.2 Adaptive Routing in MG-Join. The key design goals of the adaptive routing framework in MG-Join is to dynamically route data flows to avoid the congestion in a subset of interconnect links and to improve the bandwidth utilization. We develop a lightweight metric to adaptive routing for transferring each packet in MG-Join. Specifically, we propose an adaptive routing metric (**ARM**) for determining the routes. ARM in MG-Join is computed based on a combination of dynamic (network delay and packet size) and static parameters (latency and bandwidth). Further, MG-Join performs the routing at the granularity of individual packets, making it possible

[2]B = 1024M

to dynamically re-route data flows quickly to minimize the effects of congestion in the network. For routing each packet, the source node picks up the route with the least congestion (the route with the lowest dynamic delay).

We define the ARM values for a packet P over route R to be the total cost of moving the packet over the route. It is computed as a combination of two components: 1) a dynamic delay (D_R) resulting from congestion at the links and 2) the transmission cost of a pipelined transfer of data from source to destination (T_R), as shown in Equation 2.

$$ARM(R, P) = T_R + D_R \tag{2}$$

Transmission cost T_R for a packet P (denoting its size $||P||$) when transmitted over a route R can be computed based on static factors (assuming there is no congestion), as shown in Equation 3. Here, $B_E(||P||)$ is the maximum effective bandwidth achievable by a packet of size $||P||$ over the route R. In the initialization, we profile each type of link for different packet sizes and generate a function of B_E for different types of links. As multi-hop routing in MG-Join is designed to ensure pipelined transmission of packets, the effective bandwidth of the packet will be determined by the slowest link along the route which will be the transmission bottleneck for the entire data flow.

$$T_R = \frac{||P||}{B_E(||P||)} \tag{3}$$

The dynamic delay of route R (D_R) is computed as the sum of the queuing delay (Q_i) and the latency (L_i) of packet transmission over every single link constituting the route R, as shown in Equation 4. To enable quick computation of the dynamic delay of any route, each GPU locally maintains a table containing the Q_i and L_i values of every link in the interconnect network. The latency of data transmission of a link (L_i) is a physical characteristic of the link and does not change dynamically; while the queuing delay of a link (Q_i) changes with the level of congestion in the network. When the system begins execution, the Q_i values of the links are initialized based on the histogram data, by assuming a uniform arrival rate of packets to be forwarded. Later, whenever the Q_i value is updated due to the link transmission load, the GPU linked to it would broadcast the change in the queuing delay to every other GPU in the system. From the estimation, we can see that the dynamic delay value helps indicate congestion. Hence, any significant increase in the queuing delay serves as an indicator of network congestion.

$$D_R = \sum_{i=1}^{|R|-1} Q_i + L_i \tag{4}$$

Implementation details. To minimize synchronization costs across GPUs, the route for each packet of data is determined at the source node where the packet was generated and will not be changed at intermediate nodes. This helps avoid the need for additional synchronization to ensure in-order packet arrival and to avoid circular routes. Based on our experiment, data packets on average require only 2-3 intermediate hops before reaching the destination.

As mentioned in Section 4.1, to reduce the routing overhead, packets from the same data flow are sent in a small batch with the same route. That means, in our implementation, adaptive routing is implemented at the basis of small batches (up to 8 packets per batch). For each batch, we calculate the ARM values for all the possible routes from the source GPU to the destination GPU with the constraint of at most 3 intermediate hops, and choose the route with the smallest ARM value as the route for that batch of packets from the source GPU to the destination GPU. Given the route, the source GPU generates the header of the packet, especially for the vector of GPU ids representing the route.

To summarize, while the static policies are unable to update their routes for each flow depending on the level of congestion in the network, the adaptive routing policy in MG-Join adapts to dynamic network conditions. Further, centralized network routing protocols that support congestion control in traditional packet networks [10, 25, 48] often require frequent synchronizations across GPUs, which MG-Join tries to keep this to a minimum (Sections 4.1 and 4.2). Such frequent synchronization leads to very high overhead for GPU hash join operation.

5 EXPERIMENTS

5.1 Experimental Setup

Hardware. We mainly use the DGX-1 server for our experiments, unless mentioned otherwise. The machine is equipped with 8 V100 GPUs, each containing 80 SMs operating at a boost clock rate of 1.53 GHz. Each GPU has 32 GB of HBM (High Bandwidth Memory), operating at a peak bandwidth of 900 GB/s. Its interconnects are shown in Figure 2. We further use a DGX-Station machine [6] which is equipped with 4 V100 GPUs to demonstrate the generality of the techniques proposed in this study. The DGX-station consists of 4 V100 GPUs interconnected to each other over NVLink and PCIe interconnects. For comparison against CPU, we use a dual socket machine with 512 GB of main memory and two Intel Xeon E5-2698 v4 CPU with 20 physical cores (40 logical cores with hyperthreading) running at 2.20GHz.

Workload. Following previous studies [37, 38, 41, 44, 46], we use a synthetic data set consisting of two relations R and S with a tuple size of 8 bytes (4-byte *key* and 4-byte *id*). The integer *key* values are generated sequentially and then shuffled randomly. Further, for all our experiments, we set $|R| = |S|$ and vary the total input size ($|R| + |S|$) from 512M to 4,096M tuples. We also evaluate the impact of data distribution based on a Zipf distribution.

Implementation details. We apply database compression to reduce the amount of cross-GPU traffic. First, as we use radix-based data partitioning, we use the first n bits of the keys in the global partitioning (n is determined by the hash key and the number of times the data is partitioned recursively during the global partitioning phase). Thus, during the global partitioning, we do not need to transfer the entire key. Instead, the global partitioning phase partitioned the data into groups based on the first n-bits of each key (the remainder bits in the key is not transferred in the global partitioning). Second, the tuple id values are compressed in a block basis (8KB as a block in our implementation). We first apply a delta operation on each tuple (based on the min value within a block) and then perform a null suppression encoding on the column to remove the zero-bits. Both these data compression approaches help MG-Join to achieve reasonable compression ratios (1.3x - 2x in

our experiments) and thus reduce the overall overhead of the data distribution.

Experimental outline. Our experimental evaluation of MG-Join is organized as follows.

- In Section 5.2, we evaluate the individual impact of multi-hop data transmission and the adaptive routing policy used in MG-Join.
- In Section 5.3, we compare the overall performance of MG-Join against UMJ [31] and DPRJ [21]. For communication across GPUs within the same node, DPRJ makes use of CUDA communication APIs that choose the direct routes between GPUs for data transfer.
- In Section 5.4 we evaluate the performance benefit of adopting MG-Join when executing TPC-H queries on large data sets by comparing its performance gains over Omnisci [29] running on CPU and GPU. Although there have been other query processing systems on GPUs such as DogQC [18] and Ocelot [13], they do not support multi-GPU architectures and thus cannot support the scale factor tested in our study. To the best of our knowledge, Omnisci is the state-of-the-art system capable of executing on both CPUs and modern multi-GPU systems. We use all the six queries in TPC-H which do not contain sub-queries (Q3, Q5, Q10, Q12, Q14 and Q19) and have at least one join operation. This is because, currently we do not have any advanced optimization on sub-queries, and view a sub-query as a separate query. The detailed SQL clauses can be found in their benchmark websites [8].

5.2 Evaluating Data Distribution in Global Partitioning Phase

In this section, we focus on the data distribution step of the global partitioning phase of MG-Join. We evaluate the impact of varying the number of GPUs, while keeping each GPU allocated with 512M tuples from each input relation.

Impact of multi-hop data routing: Figure 6 shows the total data transfer throughput achieved during the data distribution step for the multi-hop routing used in MG-Join and direct route (used in DRPJ) when the number of GPUs participating in the join operation is increased from 2 to 8. Note, the total data transfer throughput is computed based on the total amount of data transferred between the GPUs and total execution time of the data distribution stage. Direct routing achieves almost the same bandwidth as MG-Join for a very small number of GPUs. This is because for a small set of GPUs, direct routing is already very efficient. However, as we scale the number of GPUs, the multi-hop routing policy achieves more efficient higher throughput (by up to 2.35x).

Impact of adaptive routing: We evaluate the efficiency of our adaptive routing policy as opposed to other static metric based routing policies. The results of this comparison in Figure 7 clearly show that for a small set of GPUs, MG-Join achieves the same performance as other static metrics. This is because due to the small number of well-connected GPUs, all metrics end up choosing the same route. However, as the number of GPUs participating in the join operation increases, the adaptive routing approach begins to outperform static routing policies based on bandwidth, hop count

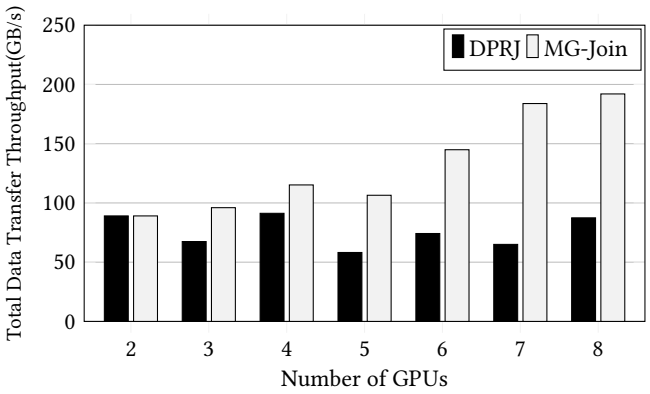


Figure 6: Data Transfer throughput comparison of multi-hop routing in MG-Join against direct routing in DPRJ.

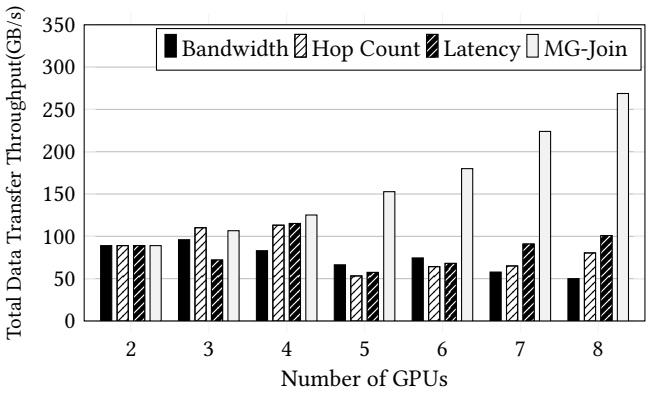


Figure 7: Data Transfer throughput comparison of adaptive routing in MG-Join against static routing policies.

and latency by up to 5.37x, 3.45x and 2.64x respectively. When a larger number of GPUs are involved, the adaptive routing policy is able to dynamically re-route the data flows, making more efficient use of available link bandwidth.

To further demonstrate the efficiency of MG-Join in taking advantage of the GPU and interconnect hardware, we present the average utilization of the bisection bandwidth of the interconnect network hardware achieved by DPRJ and MG-Join in Figure 8 for configurations with 4, 6 and 8 GPUs. Note, the utilization is computed based on a bisection bandwidth of each configuration. The results clearly show that interconnect utilization of the DPRJ implementation drops with the number of GPUs to as low as 30%. As the number of GPUs increases, the network utilization of DPRJ decreases, because the network congestion becomes more severe. In contrast, MG-Join ensures high utilization of GPU interconnect hardware, especially for configurations with a large number of GPUs. Overall, MG-Join achieves 97% utilization for eight GPUs. MG-Join achieves relatively low utilization when the number of GPUs is small. This is because the small number of possible routes limits MG-Join's ability to re-route data flows when congestion is detected in a subset of the links.

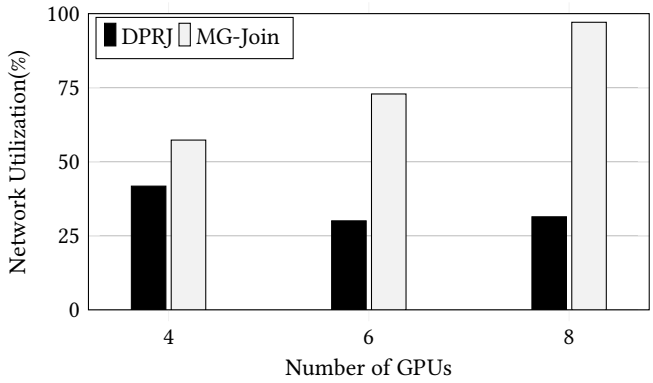


Figure 8: Interconnect hardware utilization of bisection bandwidth in MG-Join and DPRJ.

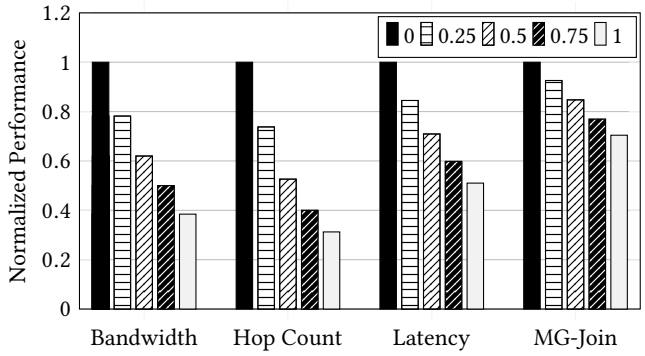


Figure 9: Normalized performance of different routing policies with data skew.

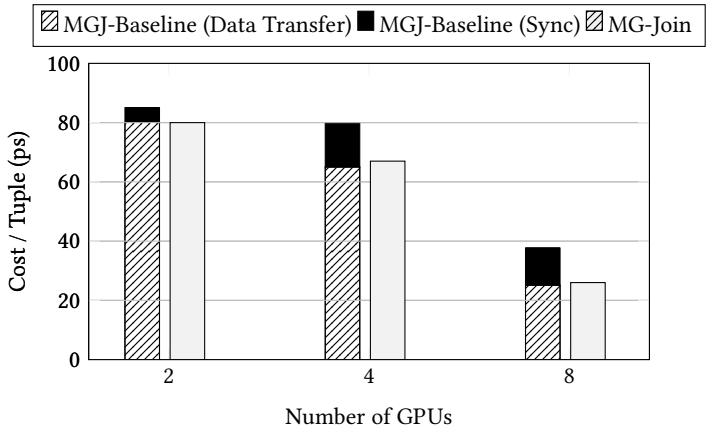


Figure 10: Data transfer cost comparison of MG-Join and MGJ-Baseline.

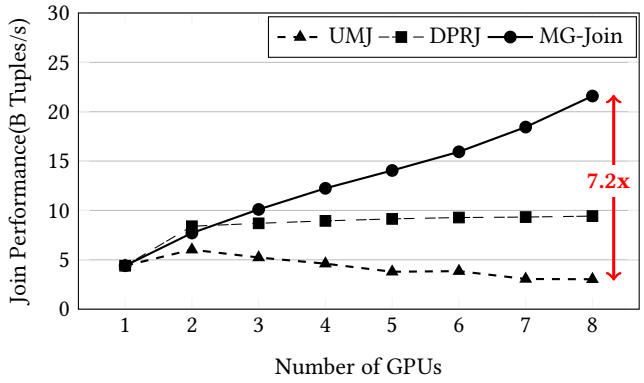


Figure 11: Overall performance comparison of MG-Join against DPRJ and UMJ.

To demonstrate the ability of MG-Join to adapt to different distributions of the input data, we compare the normalized performance of the adaptive routing policy against the three static routing policies (bandwidth, hop count and latency) when input tuples are distributed based on a Zipf distribution among the 8 GPUs in the DGX-1 machine. The result is shown in Figure 9 for Z values varying from 0.0 to 1.0. The adaptive routing policy encounters the least degradation in performance with increase in the input data distribution skew, whereas other static routing policies encounter up to 3x degradation in performance. This demonstrates the ability of adaptive routing to dynamically re-route data flows and allocate the most efficient route regardless of data distributions.

To demonstrate the benefit of our adaptive routing policy based on the ARM metric over centralized congestion control, we compare data transfer cost per tuple of MG-Join against MGJ-Baseline in Figure 10. MGJ-Baseline is simply an implementation where the routing decision is made in a centralized manner, i.e., all GPUs synchronize among every other GPU for the transfer of each batch of packets and the optimal data routing and scheduling decision is made by a centralized process and broadcasted to each participating GPU. To understand the impact of synchronizations, we split the communication cost of MGJ-Baseline into two parts: the actual data transfer cost (MGJ-Baseline (Data Transfer)) and the synchronization cost (MGJ-Baseline (Sync)). MGJ-Baseline (Data Transfer) represents the time spent on moving the data and MGJ-Baseline (Sync) represents the cost of synchronizing across the GPUs. As demonstrated by the results in Figure 10, MGJ-Baseline does help improve the performance of the actual data transfer by a small margin (up to 3%). However, the significant increase in the synchronization cost leads to MGJ-Baseline performing up to 1.5x worse than MG-Join.

5.3 Evaluation of Join Performance

In this section, we perform the overall comparison on the entire join performance of MG-Join against DPRJ and UMJ. Figure 11 presents the overall throughput of MG-Join against UMJ and DPRJ. The input tuples are distributed randomly across the GPUs with each GPU allocated with 512M tuples from each input relation. Following previous studies [27, 31, 43], we define throughput as the number of input tuples processed per unit time. We now make the following observations.

First, the UMJ implementation is highly inefficient due to its use of the unified memory feature, which encounters high overhead of handling page faults resulting from page tables getting locked by GPU threads [31]. Further, the effect of the page tables getting locked gets worse with the increasing number of GPUs, as the

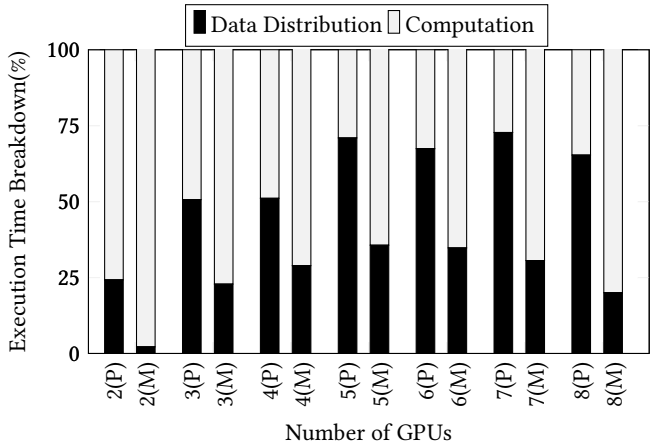


Figure 12: Execution time breakdown for MG-Join and DPRJ with varying number of GPUs.

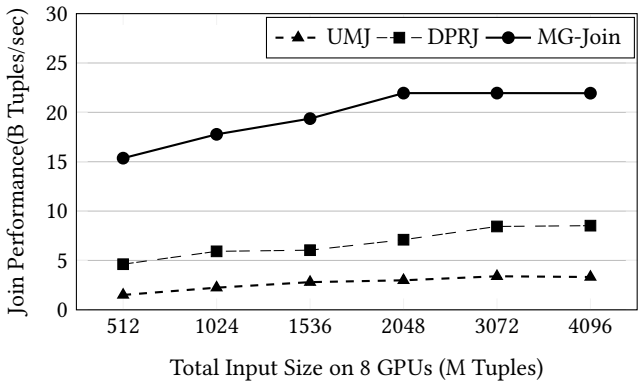


Figure 13: Performance evaluation of MG-Join, DPRJ and UMJ for varying input size on DGX-1.

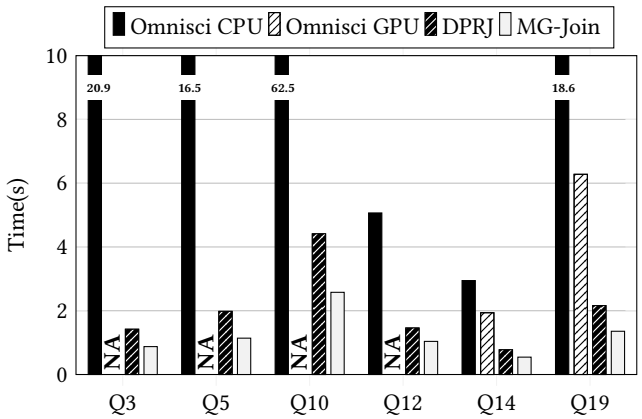


Figure 14: TPC-H query evaluation of MG-Join, Omnisci CPU and Omnisci GPU for a scale factor of 250.

larger number of GPU threads attempting to access the page table simultaneously. As a result, the performance of UMJ on multiple GPUs (from 5 to 8) is even worse than that of a single GPU.

Second, MG-Join outperforms DPRJ by up to 2.5x due to 1) network optimal partition assignment, 2) efficient adaptive routing during data distribution and 3) better overlapping of the compute and data distribution by allowing data routing at the granularity of individual packets. We will present the study on the impact of individual techniques later.

Third, while both DPRJ and UMJ achieve poor scaling with the GPU hardware resources, MG-Join achieves close to linear scaling with GPU hardware resources, demonstrating very good scalability on modern multi-GPU architectures.

To further understand the performance comparison, we compare the execution time breakdown of both DPRJ and MG-Join implementations with the number of GPUs varied in Figure 12. Note, in the figure $X(P)$ and $X(M)$ represents the execution on X GPUs for DPRJ and MG-Join, respectively. Further, the data distribution overhead for both implementations is computed as the overhead of the data transfer that cannot be overlapped with any computation (due to the lack of compute workload as communication is the bottleneck). Now, as shown in the figure, DPRJ spends up to 72% of its execution time moving the data between the GPU, while MG-Join only spends up to 35% of its execution time in cross-GPU data transfers. Further, for larger configurations like the one with 8 GPUs, MG-Join spends less than 20% of its execution time moving the data between the GPUs.

We present the performance of UMJ, DPRJ and MG-Join when the total input size of two relations is varied 512M to 4B tuples on all the 8 GPUs in DGX-1 (Figure 13). Similar to the previous cases, MG-Join outperforms both UMJ and DPRJ for all input sizes. Overall, MG-Join achieves 10.2x and 3.6x better performance than UMJ and DPRJ implementations, respectively.

5.4 TPC-H Query Evaluation

We evaluate impact of MG-Join using six TPC-H queries to demonstrate its ability in improving the overall performance of query execution on modern multi-GPU architectures. For this, we implement GPU versions of 6 TPC-H queries that make use of MG-Join and compare them against CPU and GPU versions of Omnisci. When executing on multiple GPUs, OmniSci adopts a shared-nothing architecture between GPUs, i.e., each GPU processes its own local slice of data. Even though our MG-Join based implementation is capable of processing data sets with much larger sizes, we use a scale factor of 250 for this test due to limitations in Omnisci which prevents it from processing larger data sets on GPUs.

The results of this comparison in Figure 14 show that the MG-Join implementation still outperforms DPRJ, and also outperforms both the CPU and GPU versions of Omnisci by 25x and 4.5x, respectively. Note, we are only able to present the results for Q14 and Q19 for Omnisci GPU since it fails to execute the other queries on the multi-GPU system for a scale factor of 250 (missing values denoted by *NA*). We make further observations from this result.

First, MG-Join in this study helps GPU database systems to make efficient use of the GPU and interconnect hardware in modern multi-GPU systems.

Second, modern multi-GPU systems are able to significantly outperform the CPU version (with two sockets here) for high performance analytical workloads. Due to the explosive growth in deep learning workloads, these multi-GPU servers are becoming increasingly popular. NVIDIA alone has released 5 different multi-GPU

server configurations for such workloads. Further, these devices are also becoming popular as cloud instances, making them even more accessible and cheaper. At the time of this writing, Amazon ec2 instance with 8 GPUs has an hourly cost of as low as $8.39, while a CPU instance (in US West) with 48 physical cores cost $7.776 per hour [1]. We acknowledge that the comparison between CPU and GPU is not extensive in this study and an interesting future work can be comparing the CPU systems with more sockets.

6 RELATED WORK

Joins on single-GPU hardware. Hash join on a single GPU has been studied exhaustively in the literature [9, 13, 22–24, 33, 37, 47]. Kaldewey et al. [27] designed a system that takes advantage of the universal virtual addressing feature that allows GPUs and CPUs to access data directly from each others memory subsystem. Paul et al. [31] later extended this implementation to take advantage of the more recent unified memory feature introduced by NVIDIA. Rui et al. [38] proposed a partitioned hash join implementation that joins the input relations by first partitioning the input data based on histogram and then joining the co-partitions. To ensure efficient execution, this implementation proposed the sharing of histogram data structure in GPU shared memory as well as the recursive partition of input data to generate small partition sizes. More recently, Sioulas et al. [44] proposed an implementation that partitions the data using bucket chains, thus avoiding the need to build histograms.

In comparison, this paper is the first of its kind in extensively studying the communication performance of hash joins on modern multi-GPU hardware. We have two distinct findings. First, as shown in our experiments, static routing metrics (including shortest weighted path) will be highly inefficient due to their inability to take into consideration the congestion impacts, which is a fundamental issue in efficient hash joins on multi-GPU architectures. Second, any adaptive routing scheme needs to be designed to minimize the overhead of synchronization across multiple GPUs, due to the high cost of these synchronizations and lack of dedicated routing hardware on the GPU.

Distributed joins. Distributed joins on RDMA and Infiniband networks have been widely studied in the literature. Track Join [34] and Neo Join [40] are distributed joins that work by first partitioning input data among nodes in the cluster and then joining the co-partitions. These implementations rely on costly determination of the optimal assignment of partitions to the nodes in the cluster. Barthels et al. [11] proposed a distributed join implementation for RDMA based hardware. More recently, Squirrel Join [39] and Flow Join [36] were proposed for efficiently joining skewed input data sets on Infiniband clusters. However, these implementations work on data routing in existing network layer, which is unavailable for modern multi-GPU architectures. Despite the similarity, multi-GPU architectures differ from distributed environments: firstly on a limited number of DMA engines and secondly on heterogenous links. Guo et al. [21] proposed a join implementation for distributed RDMA clusters with GPUs. However, the implementation is designed for RDMA clusters and simply relies on CUDA communication APIs (which makes use of the direct routes between GPUs) for data transfer between the GPUs within the same node. Further, while the implementation explores the use of different possible paths for communication with remote GPUs, it fails to explore multi-hop data transmission or an adaptive congestion aware routing policy that can make efficient use of interconnect bandwidth.

Multi-GPU architectures. Recent studies on GPU applications have focused on making efficient use of multiple GPUs by 1) partitioning the data set across GPUs in a way that minimizes communication across GPUs [15] or 2) enabling more efficient communication across GPUs. In this study we focus on improving the communication across the GPUs during the partitioned hash join operation. Numerous communication frameworks have already been proposed in the literature to enable efficient communication across the GPUs in multi-GPU set-ups. Groute [12], Gluon [19], DiGraph [49] and Lux [26] were proposed for handling workload distribution and communication across GPUs within a single machine for graph applications. There are also many deep learning systems on multi-GPU architectures (such as [28, 30]). Further, NCCL [4] is a communication framework from NVIDIA and is designed to allow efficient communication between GPUs. However, all these implementations adopt static routing policies which are highly inefficient on modern multi-GPU hardware, as demonstrated in the comparison with multi-hop and adaptive routing in our experimental studies.

7 CONCLUSION

The rise of multi-GPU architectures with faster interconnects offer new opportunities of relational query processing on hundreds of gigabytes of GPU memory. However, we find that due to the unique characteristics of GPUs and the interconnect hardware, existing hash join designs demonstrate very bad scalability on multi-GPU architectures, which spend up to 66% of its execution time in cross-GPU communications and achieve lower than 50% GPU-GPU interconnect bandwidth utilization. To address this, we propose MG-Join, a scalable partitioned hash join implementation for modern multi-GPU hardware. MG-Join makes use of an adaptive multi-hop data distribution policy that helps ensure efficient use of GPU hardware and interconnect links and minimizes congestion across interconnect links. Our experiments show that MG-Join achieves up to 97% utilization of the interconnect bandwidth on DGX-1 with eight GPUs, and outperforms existing partitioned hash join implementations by up to 2.5x when joining large input data sets distributed over multiple GPUs. MG-Join further helps improve the performance of TPC-H queries by up to 4.5x over OmniSCi.

There are many challenges beyond the scope of this paper but ideal for exciting future work. First, the cost of current multi-GPU architecture is relatively high. We wish the hardware community can further reduce the hardware cost to lower the adoption barrier of multi-GPU architecture. Second, high performance network interconnects such as RDMA can be an opportunity to further improve the scale of multi-GPU architectures for huge data sets.

8 ACKNOWLEDGEMENT

This work is in part supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore.

REFERENCES

[1] Amazon ec2. https://aws.amazon.com/ec2/instance-types/p3/.

[2] DGX White Paper: NVIDIA. https://www.nvidia.com/en-us/data-center/resources/dgx-1-system-architecture-whitepaper/.

[3] Interior gateway routing protocol. https://en.wikipedia.org/wiki/Interior_Gateway_Routing_Protocol.

[4] Nvidia collective communication library. https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/index.html.

[5] Nvidia dgx-1. https://www.nvidia.com/en-us/data-center/dgx-1/.

[6] Nvidia dgx station. https://www.nvidia.com/en-us/data-center/dgx-station/.

[7] POWER8 with NVIDIA NVLink Technology. https://www-355.ibm.com/systems/power/openpower/tgcmDocumentRepository.xhtml?aliasId=POWER8_with_NVIDIA_NVLink.

[8] Tpc-h. http://www.tpc.org/tpch/, 1999.

[9] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Building an efficient hash table on the gpu. In *GPU Computing Gems Jade Edition*, pages 39–53. Elsevier, 2012.

[10] S. S. Baboo and B. Narasimhan. An energy-efficient congestion-aware routing protocol for heterogeneous mobile ad hoc networks. In *2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pages 344–350. IEEE, 2009.

[11] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1463–1475, 2015.

[12] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. *ACM SIGPLAN Notices*, 52(8):235–248, 2017.

[13] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake. Ocelot/hype: Optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13):1609–1612, 2014.

[14] M. Charikar, Y. Naamad, J. Rexford, and X. K. Zou. Multi-commodity flow with in-network processing. In Y. Disser and V. S. Verykios, editors, *Algorithmic Aspects of Cloud Computing*, pages 73–101, Cham, 2019. Springer International Publishing.

[15] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *Proceedings of the VLDB Endowment*, page 13, 2019.

[16] P. Chrysogelos, P. Sioulas, and A. Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. In *Proceesings of the 9th Biennial Conference on Innovative Data Systems Research*, number CONF, 2019.

[17] N. Farrugia, J. A. Briffa, and V. Buttigieg. Solving the multi-commodity flow problem using a multi-objective genetic algorithm. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 2816–2823, 2019.

[18] H. Funke and J. Teubner. Data-parallel query processing on non-uniform data. *Proc. VLDB Endow.*, 13(6):884–897, Feb. 2020.

[19] G. Gill, L. Hoang, R. Dathathri, A. Brooks, K. Pingali, M. Snir, N. Dryden, and H.-V. Dang. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. *ACM SIGPLAN Notices*, 53(4):752–768, 2018.

[20] M. G. Gouda and M. Schneider. Maximizable routing metrics. *IEEE/ACM Transactions on Networking*, 11(4):663–675, 2003.

[21] C. Guo, H. Chen, F. Zhang, and C. Li. Distributed join algorithms on multi-cpu clusters with gpudirect rdma. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[22] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008.

[23] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proceedings of the VLDB Endowment*, 6(10):889–900, 2013.

[24] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proceedings of the VLDB Endowment*, 8(4):329–340, 2014.

[25] S. N. Hertiana, A. Kurniawan, U. S. Pasaribu, et al. Effective router assisted congestion control for sdn. *International Journal of Electrical & Computer Engineering (2088-8708)*, 8(6), 2018.

[26] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

[27] T. Kaldewey. GPU Join Processing Revisited.

[28] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *Proc. VLDB Endow.*, 12(11):1399–1412, July 2019.

[29] OmniSci. https://www.omnisci.com/. 2019.

[30] S. Pal, E. Ebrahimi, A. Zulfiqar, Y. Fu, V. Zhang, S. Migacz, D. Nellans, and P. Gupta. Optimizing multi-gpu parallelization strategies for deep learning training. *IEEE Micro*, 39(5):91–101, Sep. 2019.

[31] J. Paul, B. He, S. Lu, and C. T. Lau. Revisiting hash join on graphics processors: a decade later. *Distributed and Parallel Databases*, pages 1–23.

[32] J. Paul, B. He, S. Lu, and C. T. Lau. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proc. VLDB Endow.*, 14(2):202–214, Oct. 2020.

[33] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS*, 2011.

[34] O. Polychroniou, R. Sen, and K. A. Ross. Track join: distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1483–1494, 2014.

[35] P. Rakheja, P. Kaur, A. Gupta, and A. Sharma. Performance analysis of rip, ospf, igrp and eigrp routing protocols in a network. *International Journal of Computer Applications*, 48:6–11, 2012.

[36] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1194–1205. IEEE, 2016.

[37] R. Rui, H. Li, and Y.-C. Tu. Join algorithms on gpus: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550. IEEE, 2015.

[38] R. Rui and Y.-C. Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery.

[39] L. Rupprecht, W. Culhane, and P. Pietzuch. Squirreljoin: Network-aware distributed join processing with lazy partitioning. *Proc. VLDB Endow.*, 10(11):1250–1261, Aug. 2017.

[40] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *2014 IEEE 30th International Conference on Data Engineering*, pages 592–603, 2014.

[41] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1961–1976. ACM, 2016.

[42] M. Seliuchenko, O. Lavriv, O. Panchenko, and V. Pashkevych. Enhanced multi-commodity flow model for qos-aware routing in sdn. In *2016 International Conference Radio Electronics Info Communications (UkrMiCo)*, pages 1–3, 2016.

[43] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on gpus. page 12, 2019.

[44] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709. IEEE, 2019.

[45] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A High-Performance Graph Processing Library on the GPU. 2015.

[46] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima. Relational joins on gpus: A closer look. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2663–2673, 2017.

[47] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

[48] Y. Yukun, L. Jiangbing, X. Dongliang, R. Zhi, and H. Qing. Centralized congestion control routing protocol based on multi-metrics for low power and lossy networks. *The Journal of China Universities of Posts and Telecommunications*, 24(5):35–43, 2017.

[49] Y. Zhang, X. Liao, H. Jin, B. He, H. Liu, and L. Gu. Digraph: An efficient path-based iterative directed graph processing system on multiple gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 601–614, New York, NY, USA, 2019. Association for Computing Machinery.