Distributed GPU Joins on Fast RDMA-capable Networks

LASSE THOSTRUP, Technical University of Darmstadt, Germany
GLORIA DOCI*, Snowflake, Germany
NILS BOESCHEN, Technical University of Darmstadt, Germany
MANISHA LUTHRA, Technical University of Darmstadt & DFKI, Germany
CARSTEN BINNIG, Technical University of Darmstadt & DFKI, Germany

In this paper, we present a *novel pipelined GPU join* that accelerates the performance of distributed DBMSs by leveraging GPU resources on fast networks. A key insight is that we enable pipelined join execution by overlapping the network shuffling with the build and probe phases, thereby significantly reducing the GPU idle time. To demonstrate this, we propose novel algorithms for distributed pipelined GPU joins with RDMA and GPUDirect for both arbitrarily large probe- and build-side tables. In our evaluation, we show our pipelined distributed GPU join can reduce the overall runtime of a full query by up to 6× against a state-of-the-art CPU-only join.

CCS Concepts: • **Information systems** → *Relational parallel and distributed DBMSs*; **Join algorithms**; • **Networks** → **Data center networks**; • **Hardware** → *Networking hardware.*

Additional Key Words and Phrases: Distributed Joins; RDMA; Networks; GPU

ACM Reference Format:
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, Article 29 (May 2023), 26 pages. https://doi.org/10.1145/3588709

1 INTRODUCTION

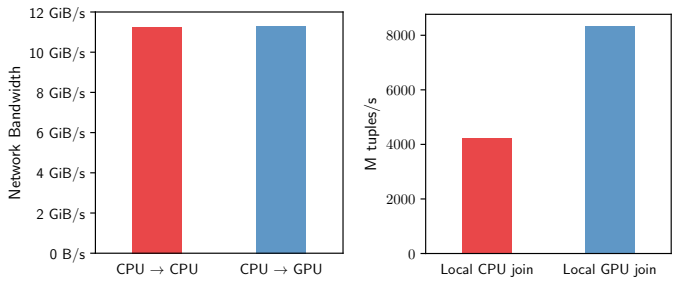
Motivation. GPUs have turned into more general-purpose processing units beyond their use for just conventional graphics processing. In contrast to CPUs, GPUs provide a tremendous amount of processing power and often come with thousands of cores in a single unit of compute that can access GPU-internal memory with high bandwidth. As a result, GPUs have risen in popularity not only in graphics processing and machine learning but for a variety of different workloads. Moreover, with its wider use, GPUs have become a commodity and are today not only available in typical on-premise clusters but also in the cloud, where the GPU-equipped machines are only slightly more expensive compared to CPU-only machines.[1]

In the context of DBMS workload, most previous work on the integration of GPUs as accelerators has focused on the acceleration of OLAP workloads for single-node DBMSs [6, 7, 13, 32, 34, 35].

*Work done while affiliated with Technical University of Darmstadt.
[1]E.g., the Azure cloud with the RDMA-capable instance Standard_NC24r with 4 K80 GPUs (with GPUDirect RDMA) is 10-25% more expensive in comparison to RDMA-capable CPU-only instances (Standard_HB60-15rs & Standard_HC44-16rs).

Authors' addresses: Lasse Thostrup, Technical University of Darmstadt, Germany; Gloria Doci, Snowflake, Germany; Nils Boeschen, Technical University of Darmstadt, Germany; Manisha Luthra, Technical University of Darmstadt & DFKI, Germany; Carsten Binnig, Technical University of Darmstadt & DFKI, Germany.

2836-6573/2023/5-ART29 $15.00
https://doi.org/10.1145/3588709



(a) Network bandwidth **(b)** Join throughput

Fig. 1. GPUs represent an interesting accelerator for distributed joins in scale-out DBMSs because of the following reasons. (a) Network communication to remote CPUs and GPUs share the same characteristics (reported by *ib_write_bw*). (b) With the same overhead for data transfers, GPU joins can significantly outperform CPU joins based on their higher processing power.[2]

This is because OLAP queries map inherently well to the vectorized execution model of GPUs and hence are clearly an interesting workload for being accelerated by a GPU. This way, the DBMS is able to execute query operators such as joins or aggregations in a massively parallel manner on a single-node DBMS. However, how to scale GPU joins and accelerate join queries in the context of distributed DBMSs is rather unexplored.

Hence, in this paper, we aim to show the potential of GPUs as accelerators in distributed DBMSs on clusters with fast RDMA-capable networks. Here, an important aspect is that using high-speed network cards with GPUDirect RDMA [23, 25], data shuffling over the network has the same cost independent of whether the target of the data transfer is remote CPU memory or remote GPU memory. To be more precise, a database node of a scale-out cluster can directly write data to the remote GPU memory (and also read data from the GPU) without the need to first write data to the remote host CPU memory and then copy it to the GPU memory. To bring this into perspective empirically, as shown in Figure 1a, RDMA writes with GPUDirect to remote GPU memory have the same network bandwidth as writing to the remote memory of a CPU. As such, in a distributed DBMS where data anyways need to be shuffled for executing the join operators, the higher speed of GPU joins compared to CPU joins, as shown in Figure 1b, can be leveraged without paying any higher cost for transferring data to the GPU.

However, when leveraging GPUs in a distributed setting, we argue that GPU joins need to be redesigned to work efficiently. To better understand why we first discuss the anatomy of a distributed join operator. A typical scheme for executing join operators in distributed DBMSs is that they first need to shuffle data across the network before the operator itself can be executed. For example, in a partitioned join, the data of the tables to be joined is first shuffled on the join keys over the network before the join is then executed in parallel on the resulting partitions. A key observation of this paper is that using the traditional sequential scheme described above, the GPUs remain *idle* when the CPU cores execute the shuffle operation. For the distributed partitioned join, this opens up an opportunity to better utilize GPUs during the shuffle operation in a *pipelined* manner by overlapping data transfer and actual join computation on the GPUs.

Contributions. In this paper, we thus propose a novel pipelined execution scheme for distributed query execution on GPUs to show the potential of using GPUs as accelerators in distributed DBMSs. We mainly focus on the effects of pipelining for distributed joins that are typically the most

[2]Comparison of GPU join [32] on Tesla V100 without data-transfer overhead vs. CPU join [2] on 4 socket Intel Xeon 8268 (150M ⋈ 300M - 8-byte tuples in random order).

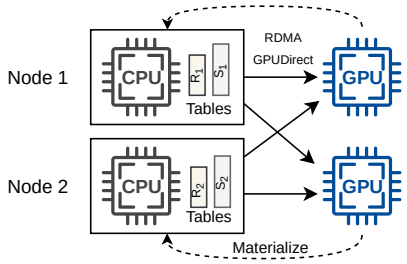


Fig. 2. Distributed Join scheme of our Distributed GPU Joins where tables are stored in CPU memory and are shuffled to GPUs for join processing.

expensive operations when executing OLAP queries [9], particularly for large distributed DBMS. As a concrete contribution, we present two novel GPU-accelerated distributed join algorithms where the data partitions that result from shuffling are processed either in a *GPU-only* or in a *hybrid GPU/CPU* manner.

The first join algorithm (GPU-only) introduces a pipelining execution scheme in a partitioned hash-join by using so-called active GPU kernels, which allow overlapping the network shuffling with the building and probing phases of the join. Moreover, it removes the need for materializing the data of the probing phase and can thus support arbitrarily large probe table inputs. As a second join algorithm, we present the hybrid GPU/CPU join that leverages both GPU and CPU for the join execution. That way, in contrast to the GPU-only join, the hybrid join can support building input tables larger than the aggregated GPU memory by materializing parts of the shuffled tables in CPU memory. In our evaluation, we show that these pipelined GPU joins can reduce the overall runtime by up to 6.8× over the state-of-the-art CPU baseline [33].

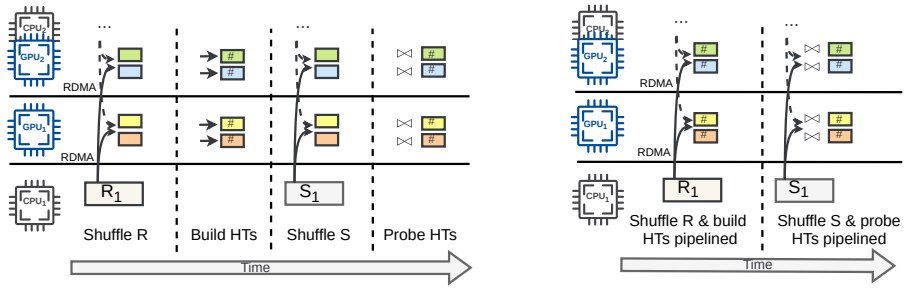
In summary, the main contributions of this paper are:

- A discussion and evaluation of the design space for integrating our *novel pipelined GPU execution* model into distributed DBMSs based on high-speed networks and GPUDirect over RDMA.
- A novel distributed GPU-accelerated join that can efficiently handle probe input tables larger than the available aggregated GPU memory.
- A hybrid GPU/CPU join that supports arbitrarily sized tables for both the build and probe inputs by transparently partitioning tuples across CPU and GPU memory.
- An extensive evaluation investigating acceleration potential over a range of different workloads as well as different settings such as available GPU memory capacities and in the context of complete queries.

Outline. In the remainder of this paper, we first give an overview of our GPU join algorithms. Next, we iterate over the design space for implementing a GPU-accelerated pipelined distributed join in Section 3. In Section 4, we present our detailed design of the GPU-accelerated pipelined join algorithm, followed by the hybrid algorithm in Section 5, which contains our solution for handling build-side tables larger than the collective GPU memory. Finally, we present our evaluation in Section 6, followed by related work in Section 7, and in the end, a conclusion in Section 8.

2 OVERVIEW OF DISTRIBUTED GPU JOIN

In the following, we present an overview of our distributed GPU-accelerated join. We first discuss the overall scheme of the join before subsequently making a case for our pipelined join approach by contrasting it to a blocking variant.



(a) Naïve Blocking Execution Model **(b)** Pipelined Execution Model

Fig. 3. For execution, we contrast two models: (a) a naïve blocking execution where the building and probing phases are distinct from network shuffling of R and S versus (b) a pipelined execution where we overlap the GPU execution with the network shuffling.

2.1 Distributed GPU Join Scheme

In this paper, we target a scheme for the *distributed GPU join* where the input tables for the join are stored partitioned across the CPU memory of the different nodes, as shown in Figure 2.

We argue that this scheme provides significant benefits over a scheme that stores data to be joined in just GPU memory: First, storing data in CPU memory allows us to support joins over input tables (and intermediate results) that are larger than GPU memory. Second, distributed joins typically need to first shuffle the input tables based on the join key. Since GPUDirect RDMA allows data to be shuffled as fast from a CPU as well as from a GPU, the location of input tables does not have any effect on the performance as long as the tables are not pre-partitioned on their join keys. Finally, the scheme allows for several optimizations, such as chaining multiple joins on the GPU by caching small intermediates, as we discuss later, which helps to further improve the runtime.

For the join execution, we apply a partitioned hash-join where each GPU executes a build and a probe phase over the partitions resulting from shuffling. The main novelty is that the shuffling and join execution is pipelined, which has many benefits, such as support for much larger input tables and even arbitrarily large probe-side tables, together with overlapping of materialization of the join result back to local CPU memory.

2.2 The Case for Pipelining

As we discussed before, the core idea of our approach is to overlap and pipeline the execution of the shuffling phase (which is driven by the CPUs) with the build and probe phases of the join (executed on the GPUs). Overall, this allows us to avoid that GPU resources are idle during shuffling and, as such, reduces the runtime of the overall join execution. We illustrate this effect in Figure 3b in contrast to the blocking execution, as shown in Figure 3a. In the following, we contrast the details of blocking vs. our pipelining approach.

Naïve Blocking GPU Join. Naïvely mapping the blocking execution scheme of the state-of-the-art distributed hash join approach [4] to a distributed GPU-accelerated join (as illustrated in Figure 3a) does not only come with severe limitations, but it also does not leverage the GPUs in the most optimal manner.

The main reason why a blocking execution scheme of the distributed join is not ideal for GPUs is that the GPU cores would stay idle until the network shuffling phase is finished. The same is true for the building and probing phases where the GPU would be active while the CPU cores would stay idle, resulting in an overall higher runtime as illustrated in Figure 3a (i.e., no work is executed on the CPUs in the build and probe phases of the GPU). Hence, in this paper, we argue for a pipelined execution that overlaps the CPU-driven network shuffling with the GPU join processing.

Moreover, another significant limitation of a blocking scheme for GPUs is that only tables of a certain limited size can be processed. The reason is that when executing the phases of a distributed join non-overlapped on the GPU, the GPUs need to be able to hold all intermediate data, e.g., the output of shuffling the build and probe tables in GPU memory. While recent generations, such as the Nvidia A100, has 40 GB or even 80 GB of internal GPU memory, the GPU memory sizes are still limited and cannot be extended as for CPUs. As such, when using a blocking model for executing a GPU-based join in distributed DBMSs, only joins where the input table sizes and intermediate data size do not exceed the aggregated memory of all available GPUs can be supported. In the following, we cover how we overcome these challenges with the pipelined join approach.

Pipelined GPU Join. The pipelined GPU join approach overlaps the execution of the shuffling of input tables with the building and probing on the GPUs. The conceptual reason why the pipelined scheme is more efficient is that such a scheme, as shown in Figure 3b, helps to efficiently hide the join processing on the GPUs under the data transfer by making use of CPU and GPU cores concurrently. Moreover, such a pipelined scheme clearly reduces the GPU memory consumption since only a chunk of data, instead of a full partition resulting from shuffling, need to be stored in GPU memory. For the probe phase, this means that arbitrarily large probe inputs can be supported and streamed through the GPUs. For the build phase, the pipelined model also has the same benefits since more GPU memory is available for the hash tables, effectively supporting larger build input tables.

However, clearly, the fixed size of GPU memory poses a strong limitation on the size of hash tables that can be kept in GPU memory. To mitigate this limitation and support input tables for the build phase larger than the aggregated GPU memory, in this paper, we propose a hybrid scheme (called hybrid GPU/CPU join) of our pipelined join that uses GPU memory along with CPU memory (and the CPU cores) to execute a distributed join. The details of these two join algorithms: pipelined and hybrid schemes, are presented in Section 4 and Section 5, respectively.

Lastly, the pipelining approach allows to overlap not only the probing of the hash tables but also to hide additional processing given the efficient vectorized execution of GPUs by chaining multiple operations (e.g., multiple joins). Such *chained processing* can be used for typical OLAP queries with several joins between dimensions and the same fact table where we cache small (replicated) dimension tables on the GPU. We show in Section 6.3 how the pipelined approach can speed up a full query from the SSB benchmark with 4 joins up to 6.8× against a non-accelerated CPU query implementation.

Discussion. One might now argue that the pipelined scheme can also be used to overlap network shuffling and join execution to accelerate a distributed CPU join. A key aspect, however, of why pipelining for a distributed CPU join will not yield significant benefits is based on the anatomy of the distributed CPU join, which is very different from the GPU join. While for the CPU join, shuffling data over the network and executing the join by building or probing into a hash table utilizes the same resources, the GPU join can use distinct resources; i.e., when the CPU shuffles the data, the GPU is responsible for consuming the incoming data and building/probing hash tables.

As such, in the distributed GPU join, the GPU would remain idle if not overlapped with the CPU-driven shuffling. In a distributed CPU join, in contrast, this is different since the CPU resources need to be shared between shuffling and join execution if the two phases are overlapped. Moreover, with modern high-speed networks to saturate their high bandwidth, the CPU cores are already highly utilized by shuffling the data, which involves scanning the input tables of the join, partitioning the data for shuffling and sending the data [3]. In fact, for the hardware used in our experiments, all CPU cores are fully utilized during shuffling and thus, reserving dedicated CPU resources for joining would slow down the network shuffle, which anyway dominates the overall CPU join runtime.

Another aspect of GPU join is that GPUs provide additional memory resources for the read/write operations of building and probing hash tables. In contrast, for a CPU join, read/write operations of building and probing hash tables compete with the read/write operations of shuffling on the same memory resources, which hence limits the benefits of overlapping the shuffling with the join execution in the CPU-based join.

3 DESIGN SPACE FOR PIPELINED GPU JOIN

Designing a pipelined GPU join over fast RDMA-enabled networks is non-trivial and involves many design decisions. In the following, we thus first present the relevant background on RDMA and GPUDirect to enable the pipelined join and then discuss the design options for the join.

3.1 RDMA & GPUDirect

While RDMA has been used for distributed data-processing systems [3–5, 20, 29, 36, 38] over high-speed networks, there are some important aspects that need to be considered to directly transfer data from and to the GPUs.

In general, to leverage RDMA, an application can make use of different communication schemes that can be categorized as one-sided (READ / WRITE) or two-sided (SEND / RECEIVE) operations, which refers to the involvement of the sender- & receiver-CPU in the communication. For one-sided operations, only the sender node is actively involved, which typically brings along performance benefits since it avoids any additional overhead on the receiver. For two-sided operations, the receiver must actively be involved by posting RECEIVE requests to steer the placement of data.

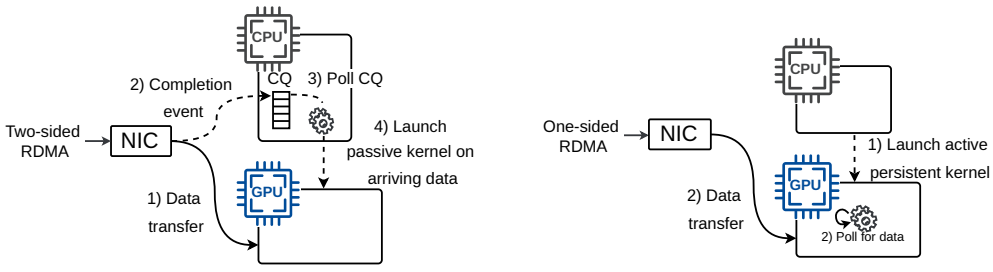
To make use of RDMA on GPUs, GPUDirect RDMA provides a means of transferring data directly over the network from and to the GPU memory using the same RDMA primitives for one- or two-sided communication. With GPUDirect RDMA, data can be copied over the network at the same speed from and to the GPU memory as it can be copied from and to the CPU memory. Furthermore, data can be copied from the sender memory directly to the remote GPU memory without first being copied to the main memory of the remote CPU. This is enabled by exposing the virtual to physical address mapping of the GPU to third-party PCIe devices such as the NIC, thereby allowing the NIC to directly read and write to and from the GPU memory.

However, using GPUDirect RDMA (for distributed DBMSs) is not straightforward and comes with a few important challenges, in particular for implementing a pipelined execution on the GPU. First, for combining the data flow via RDMA and the GPU kernel execution, different design options exist compared to a pure CPU-based solution. Second, another challenge is that GPUs can potentially observe inconsistent data of incoming RDMA writes, such as partially written data or data that is not written sequentially (i.e., from lowest memory address to highest) or even observe RDMA writes arriving out-of-order. As such, we need to carefully design a pipelined execution scheme of a distributed join, as we discuss next.

3.2 Design Alternatives

In the following, we discuss the different design options for our pipelined GPU join. We categorize the design options into two dimensions: *(i)* how to use the RDMA communication primitives for implementing the data flow for shuffling data over the network from CPU to GPU memory and *(ii)* the control flow of how the GPU kernel execution is triggered to consume incoming data for building hash tables and probing into them.

One- vs. Two-sided Data Flow. As previously mentioned, one-sided and two-sided RDMA operations exist to implement the data flow between machines. While these primitives work similarly for GPUs using GPUDirect as for CPUs, there are some important differences. Using one-sided primitives with GPUDirect works the same as for CPUs since GPUDirect allows that CPUs of the



(a) CPU-driven Join / Two-sided RDMA **(b)** GPU-driven Join / One-sided RDMA

Fig. 4. Two design alternatives for implementing a pipelined GPU join over RDMA where either (a) the CPU is central in the control flow with two-sided RDMA or (b) the GPU controls when a new chunk of tuples can be processed without CPU involvement.

sending nodes can write in the remote GPU memory without involving the remote GPUs. When using two-sided operations instead, it is important to note that the RECEIVE requests are driven by the CPU (and not the GPU) in GPUDirect, since the CUDA library does not support calls to RDMA functions. Hence, in a two-sided communication, the CPU is always involved in the data flow even though the GPU is the target.

CPU- vs. GPU-driven Control Flow. Another aspect is whether the CPU or the GPU is driving the control flow, i.e., detecting when new data has arrived (on the GPU) and triggering the execution on the GPU for building/probing into the hash tables. In the following, we discuss two design options for the control flow: a CPU-driven approach and a GPU-driven approach.

(i) In a CPU-driven approach, the CPU actively detects that a new chunk of tuples has arrived (by polling for so-called RDMA completion events). Afterwards, the CPU then instructs the GPU for subsequent processing by launching a GPU kernel. Launching a GPU kernel *after* the write of a new chunk has been detected on the CPU ensures consistency and therefore overcomes the aforementioned challenge of inconsistent data with GPUDirect. However, the CPU-driven approach also introduces additional overhead and latencies for the pipelined join since, for each incoming batch of tuples, the GPU kernel needs to be called.

(ii) In the GPU-driven approach, the kernel launch overhead is removed by using persistent kernels where the GPU instead actively detects new incoming data by polling on a particular memory region for newly arrived data. This technique is also often applied in traditional CPU-based RDMA communication as memory polling has a smaller overhead in comparison to polling after RDMA completion events [8, 37]. However, having such an active GPU kernel running concurrently with RDMA writes poses challenges to the consistency of the incoming data, which we later discuss how to overcome with minimal overhead in Section 4.2.

Join Design Space. By pairing the dimensions of the control flow and the data flow, different designs can be derived to enable a pipelined GPU join, as shown in Figure 4. The first design is a CPU-driven design that makes use of two-sided RDMA as illustrated in Figure 4a and GPU-driven with one-sided RDMA in Figure 4b. In the CPU-driven design (Figure 4a), the CPU is polling for completion events for incoming RDMA data and can subsequently launch a GPU kernel for the next chunk of incoming tuples to process the data written directly to the GPU. On the other hand, as a second design, a completely GPU-driven design (Figure 4b) can be used where a persistent GPU kernel is launched (once) initially, which itself actively detects and polls for incoming data. As can be seen, these two designs differ both in CPU overhead, the overall communication pattern, and how they ensure data consistency of incoming data to the GPU. Therefore, we next take a

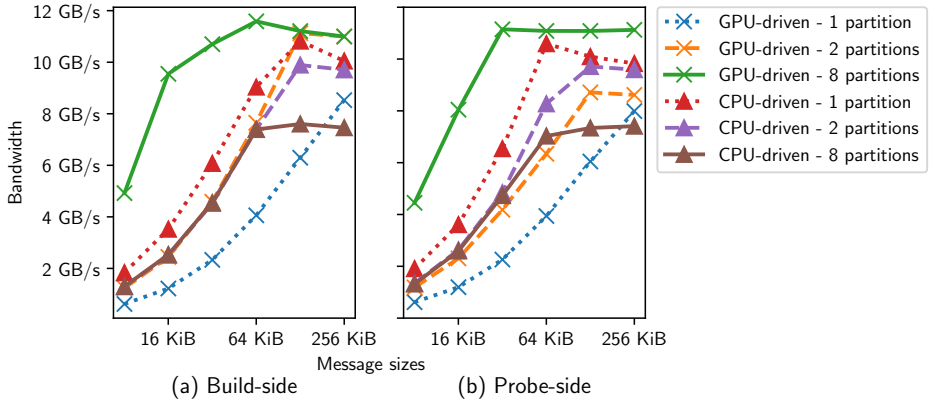


Fig. 5. CPU-driven two-sided RDMA vs. GPU-driven one-sided RDMA for different degrees of partitions and message sizes on a 4 node cluster (1 GPU per node) using 2B ⋈ 8B 16-byte tuples. The GPU-driven approach achieves a higher network utilization with more partitions, whereas CPU-driven does not scale with more partitions.

closer look at the performance of these two variants to find the best candidate for realizing the pipelined GPU join design.

3.3 Design-Space Evaluation

To guide the decision of which of the two alternatives has the most potential for a pipelined GPU join processing, we now evaluate the two designs (i.e., CPU-driven with two-sided RDMA vs. GPU-driven with one-sided RDMA).

A key question that determines the overall performance of a distributed join is which design can better saturate the network bandwidth during shuffling, which is the limiting factor of a distributed join. We thus examine the designs by their ability to saturate the shuffling speed with different degrees of partition parallelism. Moreover, since the join is pipelined, it is paramount that the GPU kernel that executes the join steps (i.e., building or probing) can ingest the incoming data at the speed of the sender CPU cores. Otherwise, the GPU kernel for the join execution would slow down the join resulting in a reduced shuffling speed from senders since they can only send data at the speed at which the GPU can ingest the data.

For the design space evaluation, as shown in Figure 5, we hence execute a shuffling operation with a pipelined GPU kernel for building (a) and probing (b) hash tables that are the core components of our GPU join. We evaluate for varying message sizes, i.e., the size of each chunk of tuples we transfer during the shuffling phase from CPUs to the GPUs, against different number of partitions per GPU or CPU, as seen in the legend. A first observation is that using a higher number of partitions for the CPU-driven approach results in sub-optimal performance due to the CUDA library internally sequentializing the GPU kernel calls, even though a separate CUDA stream is used for each partition. Also, since each node is both sending and receiving (all-to-all shuffling), a higher partition fan-out results in fewer CPU resources available for sender threads in the CPU-driven approach since more receiver threads must be reserved for RDMA control flow and calling GPU kernels. This becomes visible for 8 partitions (brown line) where the performance plateaus at 64 KiB.

On the other hand, the GPU-driven approach does not inhibit the same limitation and thus scales better with the partition parallelism. For example, with 8 partitions per receiver node, the GPU-driven design can efficiently saturate the sender and network throughput. The reason is that the GPU kernels are not launched from the CPU on a per-message basis, as is the case for the

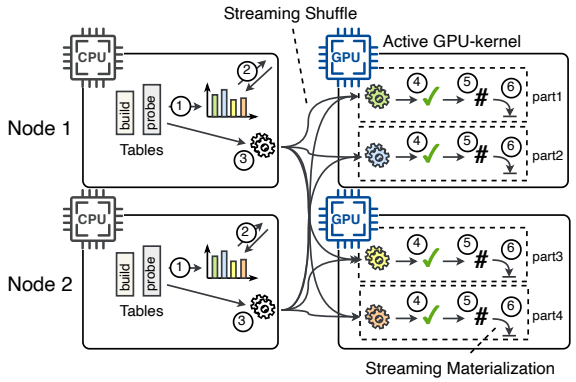


Fig. 6. Pipelined GPU Join where tables are shuffled from CPU to (remote) GPU memory using a streaming shuffle operator that leverages GPUDirect RDMA. Hash tables are built and probed on the GPUs in a pipelined manner using active GPU kernels.

CPU-driven approach. Instead, the GPU kernel is started once (at the beginning of the shuffling phase), and then the kernel actively polls in the GPU-local memory for new incoming data. As a result, the kernel launch contention of the CPU-driven approach where multiple CPU threads are launching GPU kernels in parallel is avoided. This allows the GPU-driven design to achieve a higher GPU utilization and thus higher processing speed through partition parallelism.

Summary. Based on these observations, the GPU-driven approach clearly dominates over the CPU-driven approach since it can better use partition parallelism and thus better utilize the network for a pipelined GPU join. In the remainder, we thus use the one-sided GPU-driven design for realizing our pipelined join.

4 PIPELINED GPU JOIN ALGORITHM

In the following, we first give an overview of the one-sided GPU-driven design before we then explain the details of each step.

4.1 Overview of Execution Steps

In our pipelined GPU join, we aim to execute the shuffling of tuples for the join operation in a concurrent manner with the build and probe phase using so-called *active GPU kernels*. To better understand the overall execution phases on the CPU and GPU, we discuss the general steps for a two-node setup as shown in Figure 6:

Steps on CPUs (Sender-side). The CPUs on the sender-side are responsible for executing the data shuffling. For this, ① the CPUs first build the histograms in parallel on the build- and probe-side input tables that are to be joined. Afterwards ②, the CPUs exchange the local histograms (per database node) to compute a global histogram that allows all nodes to compute the total size per resulting partition before actually executing the shuffling. This global histogram is used by our pipelined GPU join to reserve adequate GPU resources and, in case of insufficient GPU memory, decide which of the partitions to place in GPU memory and which in CPU memory (as later discussed in Section 5). Once the global histograms are computed, the streaming network shuffle phase ③ starts that executes the re-partitioning based on join keys by transferring small chunks of data from the sender CPUs to the different GPUs.

Steps on GPUs (Receiver-side). The GPUs in our join are responsible for detecting incoming data and for executing the build and probe steps of the join, as shown in Figure 6 (right side). A novel aspect of our GPU join is that it executes the GPU steps ④–⑥ in an overlapped mode with the

streaming shuffling using *active GPU kernels*. An active GPU kernel is started once at the beginning of a shuffle phase. Once started, an active kernel then polls for new incoming data chunks in GPU memory — first for the building and then for the probing phase. A challenge for the GPU-side execution with active kernels is to ensure data consistency as well as synchronization between CPU sender and GPU receivers. We accomplish this by proposing a parallelization strategy that aims to optimally map partitions resulting from shuffling to the vectorized execution model of the GPU. Details of this strategy, along with details about all the steps on the GPU, are discussed next.

4.2 Active GPU Kernel

Active GPU kernels not only poll for incoming data, but a core aspect is that they allow us to efficiently parallelize the join, which is composed of the consistency check step ④, build and probe steps ⑤ and successive operations ⑥ such as result materialization (illustrated in Figure 6). In the following, we focus on two aspects regarding the parallelization strategy of the active kernel: *(i)* how to determine the partition fan-out during shuffling and hence to determine the parallelism for an efficient GPU execution, and *(ii)* how to determine the amount of GPU resources for the different (potentially unequal sized) partitions. Moreover, we also briefly explain how we deal with data consistency for incoming RDMA writes as well as some relevant implementation details for the build and probe steps.

Parallelization Strategy. To answer the two aforementioned questions regarding the main aspects of parallelizing the join, we first need to explain some background on the vectorized execution model of GPUs. GPUs follow the SIMT (single instruction, multiple threads) execution model and typically come with thousands of threads; and thus, parallelizing the join processing on a GPU is very different from a CPU-based execution. The threads are grouped into a two-level hierarchy: *thread blocks* forming a group of threads and a *grid* comprising thread blocks. While all threads in a thread block are scheduled on a so-called Streaming Multiprocessor simultaneously, different thread blocks in a grid can be scheduled independently. We next discuss how the two aspects of the parallelization strategy work as discussed above.

(i) Determine Partition Fan-Out. First, we now look into the aspect of how to determine the partition fan-out during shuffling. For this, we use a scheme that over-partitions the data on the receiver GPUs, i.e., we use more partitions than the number of GPUs. As already shown in Section 3.3, this enables more efficient GPU join execution as each partition can be processed independently on the GPU without synchronization as the data arrives. When considering one partition, the maximum degree of parallelization for this partition depends on the size of the incoming message (chunks of tuples). For example, for a message size of 128 KiB with 16 B tuples, the maximum number of threads we can utilize is 128 KiB / 16 B = 8192 threads (since each tuple is processed by max. one thread). However, 8192 threads in this example might not be enough to saturate the message throughput considering high-speed networks, and as such more partitions might be required.

Yet, naïvely increasing the partition fan-out will negatively impact the speed at which the sender CPU threads can partition (and potentially could also exceed the TLB-cache [2]). Therefore, the optimal partition fan-out is achieved when the GPUs are able to saturate the tuple throughput, which in turn depends on CPU-processing speed and network throughput. For this reason, selecting the partition fan-out also highly depends on the specific GPU and CPU architecture and the network speed. So instead of fixing this parameter for our GPU join, we expose the degree of partition parallelism as a parameter one can configure.

In Figure 7, we show the effects of this parameter in a micro-benchmark. From 4 partitions per GPU, the network is becoming saturated for the different message sizes. While choosing a higher partition fan-out does not increase the aggregated network throughput, we empirically found that

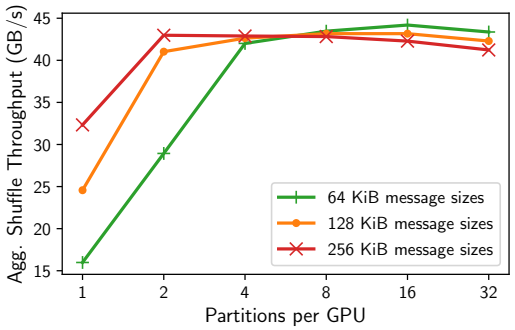


Fig. 7. Aggregated network throughput on 4 nodes for the GPU-driven approach. From 4 partitions per GPU, the network is becoming saturated for the different message sizes.

a fan-out of 16 partitions per GPU yields the most robust performance as more GPU resources can be allocated. Thus, in all our experiments, we are using a partition fan-out to get 16 partitions per GPU, which has shown to provide the best performance across all workloads we used in our evaluation.

(ii) GPU Resource Allocation. The second aspect of parallelizing the GPU join is how to determine the amount of GPU resources for the different (potentially unequal-sized) partitions. For this, we first discuss how we assign multiple thread blocks (grid) to each partition. Using multiple thread blocks per partition instead of limiting it to only a single thread block provides more flexibility in terms of allocating the necessary amount of resources needed to process each partition. However, this scheme raises a challenge of coordination across thread blocks when processing each chunk of tuples. We deal with this using so-called cooperative groups primitives of CUDA that enable us to provide synchronization across all threads assigned to process a specific partition.

Furthermore, for resource allocation, we need to decide how many threads (i.e., thread blocks) we assign to each partition. As the GPU join processing is pipelined, the size of each partition translates into an estimate of how frequently each partition will receive a chunk of tuples, assuming that the tuples of each partition are roughly spread out over the tables. Thus, an important question is how to determine the amount of GPU resources (threads) for the different partitions. One naïve option is to assign a static number of thread blocks to each partition. While this performs well for uniformly sized partitions, this might result in suboptimal performance for unequally sized partitions as a fixed number of thread blocks might not be sufficient for the different partition sizes. Another option is to always allocate the maximum number of threads possible for each partition, but this, in turn, might exceed the number of threads available or result in wasted GPU resources and potentially block other kernels from running. To deal with this problem, we dynamically decide the size of the grids (i.e., the number of thread blocks) depending on the relative size of the partition that can be deduced from the histogram. As an example, for a fan-out of just two partitions, if one partition is twice the size, it will also get allocated twice the threads in comparison to the smaller partition. Deciding on the total amount of threads used on each GPU is dependent on the available hardware resources the join is running on. We show the ability to handle skewed workloads in Section 6.5.

Ensure Consistency. Another challenge we handle in implementing the GPU-driven/one-sided RDMA join is the issue of the GPU kernel observing potentially inconsistent data during concurrent incoming RDMA writes, as discussed in Section 3. These inconsistencies can come in the form of partially written data, data not written sequentially (i.e., from lowest memory address to highest), or out-of-order messages. While the general observations made in existing literature also apply to

GPUs (such as the benefits of using one-sided over two-sided RDMA, inlining or door-bell batching), many techniques that are used to implement efficient RDMA-based communication schemes such as mailboxes or end-of-message polling [8, 10] cannot be directly applied. The reason for this is that all these techniques rely on the ordering guarantees between individual RDMA messages as well as a fixed write-order within one RDMA message, which is not guaranteed on the GPU.

Hence, a solution here is that the GPU kernel performs additional consistency checks on the incoming data chunks by appending a checksum per chunk of data (④ in Figure 6). In case data is only partially written on the GPU, the checksum verification that is executed by the GPU kernel in every iteration during busy polling will fail, and the GPU kernel can retry reading the data chunk and comparing the checksum in its next iteration. As the type of error detection needed does not involve bit-flips, transmission errors, or data written to a wrong memory location (due to the underlying reliable network transport), we use the sum complement checksum that has a very small overhead while still being able to detect when the data has not fully been written.

We hope that NVIDIA will eventually provide memory fence primitives for RDMA, which would render the need for consistency checks obsolete.

Build and Probe Steps. After ensuring the consistency of the chunk of tuples, the tuples are inserted or probed into a partition-specific concurrent hash table (⑤ in Figure 6). For building the hash table, we use a custom design that leverages CUDA atomics to realize a lock-free hash table for our distributed GPU join algorithm. Logically, our custom hash table uses a chained hash table design. However, to avoid costly allocation and lock operations during the build phase, we use a design with two arrays — one array to implement the hash table and another dense array (called the chain array) that stores the chains for all hash buckets to handle collisions. Since we know the total number of tuples that we need to insert into the hash table from the histograms created during the shuffle phase, both arrays are pre-allocated to not incur any overhead for runtime memory allocations.

In the build phase, a batch of tuples for a given partition is then inserted in the hash table using these two arrays as follows: first, a GPU thread stores the new tuple into the chain array by atomically incrementing an offset into the chain array. Afterwards, this offset is written to the hash table array. To handle hash collisions, an atomic exchange operation is used. Specifically, the thread which inserts a new tuple will see the previous offset and use this to connect the tuples in the chain array accordingly. For linking the tuples in a chain, no atomic operation is needed. Finally, during the probing phase, the join key of each tuple is hashed and the chain of potentially colliding join tuples is traversed in the chain array. Contrary to the build phase, the probing does not incur any atomic operations since the hash table is static during probing. In summary, the hash table design is a good fit for the vectorized GPU execution as it does not incur any memory allocations, locking and only a few atomic operations for building.

4.3 Successive Operations

After finishing the probing, our join provides several options for successive operations in a query plan. One option is naturally to simply materialize the join result back to CPU memory (⑥ in Figure 6), as we discuss below. However, as analytical queries typically perform multiple joins and aggregate the join result, we designed the active GPU kernel to be easily extendable to chain multiple GPU-local computations on the distributed join result before materializing results to the CPU. For example, instead of doing only a single probe, we could chain multiple probes that are still executed in a pipelined manner on the GPU. This chained scheme maps especially well to the typical abundance of processing power on the GPU and the pipelined execution model since it allows to hide even more computation under the network cost. We demonstrate this ability to chain multiple operations in our pipelined join to accelerate also complete queries in Section 6.3.

For materializing the result of the join either for subsequent operations on the CPU or delivering the result to the client, we take a streaming materialization approach which asynchronously writes data to the CPU memory, which nicely integrates with our pipelined execution model. This gives us several benefits over naïvely materializing the join result on the GPU and subsequently transferring the result back to the CPU. First, the GPU memory is limited, and thus avoiding result materialization on the GPU frees up resources to store hash tables of the join. Second, streaming the result directly to the CPU means that it comes with only a negligible overhead since the transfer can be overlapped with the pipelined execution of shuffling and GPU execution.

For realizing the streaming data transfer from the GPU to the CPU memory, there exists a range of transfer methods in CUDA: asynchronous memory copy, UM (Unified Memory) and zero-copy through UVA (Unified Virtual Addressing). The CUDA asynchronous memory copy from GPU to CPU memory can only be called from the CPU and, as such, is not applicable to our execution model with active GPU kernels where the CPU is not involved in the execution. Different from this, UM and zero-copy allow the GPU to directly access (read and write) CPU memory, where the CUDA abstraction takes care of utilizing the DMA hardware on the GPU to copy the data to CPU memory. We base our approach on UVA since we found that UM cannot fully pipeline the data migration of memory pages with the GPU processing. For leveraging UVA for pipelined materialization, we allocate the result relation using UVA in CPU memory. On the GPU, we first store the output tuples in intermediate output buffers and only copy a buffer (through CUDA memory copy device-to-device) to the result relation in UVA whenever the output buffer is full. To interleave GPU processing with the transfer to CPU memory, we use two output buffers per partition on the GPU.

4.4 Streaming Shuffle

Finally, we present how we realized efficient pipelined data transfers in the network shuffling phase (③ in Figure 6). To implement the streaming shuffling, we built our solution on top of the *Data Flow Interface* (DFI) [33], which is a high-level abstraction for fast networks that leverages one-sided RDMA communications for data transfers. The core abstraction of DFI is so-called flows that allow senders to push data into a flow and receivers to pull data out of the data flow. In a nutshell, flows support asynchronous communication between senders and receivers, which therefore allows for overlapping computation and communication.

However, DFI in its original design only supports CPU-to-CPU communication and does not come with GPU support. To realize the streaming shuffle operator with support for GPUs, we extended the flow abstraction of the original DFI code. As the main extension, we enabled the end-points of flows (called targets in DFI) to be located on GPUs by extending the buffer design and memory polling operations to allow the GPU to consume tuples out of the DFI flows. Additionally, we enabled GPUDirect RDMA by allocating the GPU-side DFI buffers through CUDA and then registering the memory region to the NIC.

Moreover, as previously mentioned, the GPU has a relaxed memory model that can result in memory inconsistencies when running a kernel concurrently with incoming RDMA data. For this, we extended DFI on the sender-side to compute a checksum over a batch of tuples if the target of a flow is on a GPU. On the target-side, when consuming this batch of tuples out of the DFI flow, we then re-compute the checksum in DFI and compare it with the appended checksum of the batch as discussed before in Section 4.2. Since the data is guaranteed to be eventually written consistently on the GPU [24], on the target side of a flow, we simply re-compute the checksum for the next batch until the checksum is correct and then hand the block to the active GPU kernel for either building or probing the hash tables.

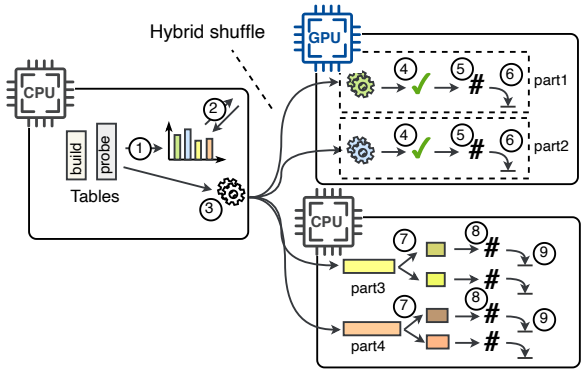


Fig. 8. Overview of Hybrid Join. Tables reside in CPU memory and are shuffled to both GPUs and CPUs. Partitions going to CPUs are executed as a traditional radix hash join.

5 HYBRID CPU/GPU JOIN ALGORITHM

In this section, we present our hybrid CPU/GPU join for supporting joins with arbitrary-sized build-side tables. One way to solve this problem is to pre-partition the build and probe input tables such that they fit in the GPU memory and use multiple rounds of the pipelined GPU approach presented before. However, the performance of this approach would significantly degrade since the pre-partitioning cannot be pipelined with network shuffling or GPU processing. As such, the goal of our hybrid pipelined join is to handle this scenario elegantly without, in the worst case, de-accelerating the join compared to a state-of-the-art CPU baseline.

The intuition behind our approach is to partition the input tables in the partitioning phase across remote GPU and CPU memory. While doing so, we leverage the GPU memory as a primary location for the partitions and only use the CPU memory for the remainder of the partitions that do not fit on the GPU. Figure 8 shows the idea of a hybrid join where a sender CPU is shuffling to both a GPU and CPU. For executing the join on the partitions shuffled to CPU memory, we make use of a state-of-the-art (blocking) CPU join implementation [4, 33]. This design comes with the benefit that its lower performance-bound is the performance of the CPU join algorithm, where any amount of available GPU memory can help to speed up the CPU baseline by using the additional GPU pipelined join. For implementing the hybrid join, we introduce a new hybrid shuffle operator that, in the shuffling step, over-partitions the input tables to a degree where we can maximize the benefit of the GPU pipelining by fully utilizing the available memory.

In order to determine the number of partitions that need to be created by the hybrid shuffling, we aim to place a fixed but configurable number of partitions on the GPU (we use 16 for our setup, as discussed before). However, as the build-side table exceeds the available collective GPU memory, setting the number of partitions to 16 per GPU would result in much fewer partitions actually assigned to the GPUs. We instead increase the fan-out such that the average size of 16 partitions roughly matches the available GPU memory. Consider the following example: assume the build-side table has a size of 40 GB, and we can only store 32 GB when using two GPUs (with 16 GB per GPU). With 32 GB of GPU memory, the average partition size would be 1 GB. To decide the fan-out for a table with 40 GB, ideally, we would therefore create $40/1 = 40$ partitions. However, as the partitioning is done with radix-hashing, only the power of two partitions is applicable (2^n partitions for n bits), yielding 64 partitions. The average partition size for 64 partitions is $40/64 = 0.625$ GB and we can thus place $\lfloor 32/0.625 \rfloor = 51$ partitions on the GPUs (assuming a uniform distribution).

Moreover, often the resulting partitions after shuffling are not equally sized. To decide which partitions to allocate to the GPU, we thus draw on the knowledge of the histograms created in ①

and ② as in the pipelined join (cf. Figure 8, left). Based on the global histograms, we can decide where to place each partition before starting the shuffling phase ③ to ideally allocate partitions to GPUs such that we leverage the full available GPU memory resources to store hash tables of the build phase. A naïve random allocation instead would need to join unnecessarily many tuples on the CPU (which introduces additional runtime overhead) since GPU memory, and thus the computational resources are not fully utilized.

Finally, for the partitions assigned to the GPUs, steps ④–⑥ are executed pipelined on the GPU as earlier described in Section 4. For processing the partitions assigned to a CPU, we execute these steps in a sequential manner based on a CPU-based join (⑦ to ⑨) after the shuffling phase finishes. Therefore, we fully materialize the intermediate partitions for the CPU as the pipelined execution model does not map well to the CPU-based execution as discussed in Section 2.2.

6 EXPERIMENTAL EVALUATION

In our evaluation, we analyze the GPU-acceleration potential using our distributed GPU join algorithms for different workloads and hardware resources. In the following, we first explain the setup before we discuss the results of the different experiments where we compare against various baselines.

6.1 Setup and Workloads

Setup. All experiments were conducted on a 5 node cluster, each node equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main-memory split between both sockets. Each node has two Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x, 100 Gbps) and two Nvidia Tesla V100 GPUs with 16 GB memory, supporting GPUDirect RDMA. The operating system is Ubuntu 18.04.1 LTS with Linux 4.15.0 kernel on all nodes. All joins are implemented with C++17 and compiled with gcc-10.1.0 and nvcc-11.3 (CUDA 11.3).

Workloads. The workloads used in experiments 1 and 3−4 follow the previous work on distributed CPU joins [4], where input tables are partitioned across the distributed nodes. The tables have randomized tuple order, and unless otherwise stated by selected experiments, the joins are evaluated with 16-byte tuples and without materialization of the join result for neither CPU baseline nor GPU-accelerated joins. In the second experiment, we evaluate two full queries comprising several joins and aggregation from the Star-Schema-Benchmark (SSB).

Join Variants. In the evaluation, we compare the performance of the following join implementations:

- CRJ - *CPU Radix Join:* As a CPU baseline, we use state-of-the-art implementation of the distributed radix hash join [33].
- GPJ-B - *GPU Partitioned Blocking Join:* This is a distributed variant of the state-of-the-art single-node GPU partitioned join [32]. The shuffle phase (i.e., histogram creation & data shuffling from [4]) and the GPU execution phase are executed subsequently. Hence, we term this join *blocking*. Data shuffling is either realized with GPUDirect or without GPUDirect.
- GPJ - *GPU Partitioned Pipelined Join*: Our novel GPU-accelerated distributed join that supports pipelining of the network shuffling and the GPU join phases, as earlier explained in Section 4. This join already allows arbitrary-sized probe-side tables.
- GPJ-H - *GPU Partitioned Hybrid Join*: This is our hybrid algorithm where both GPU and CPU are used for join execution. This join allows arbitrary-sized build-side tables that go beyond the aggregated memory capacities of all GPUs, as earlier explained in Section 5.

All joins make use of established optimizations for efficient partitioning, such as software write combine buffers (SWWCBs), non-temporal streaming hints [1, 30], and one-sided RDMA writes [3].

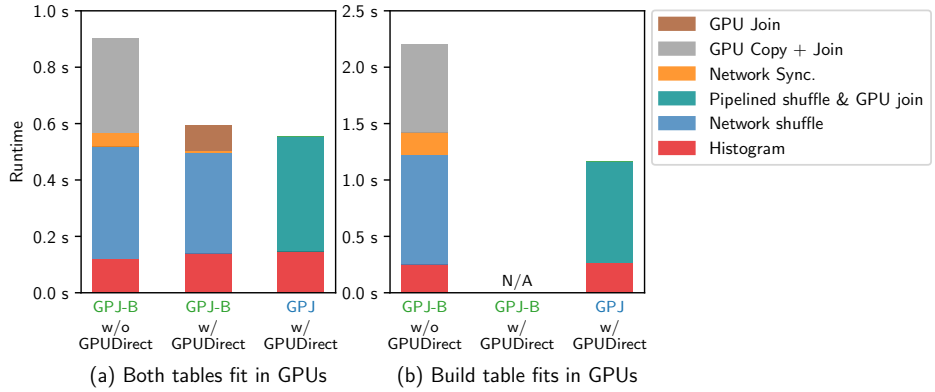


Fig. 9. Blocking (GPJ-B) vs. pipelined (GPJ) GPU join with build-side of 600×10^6 tuples and probe-side of (a) 1.2×10^9 tuples and (b) 4×10^9 tuples. Our approach (GPJ) provides a speedup of approx. 2× while supporting arbitrarily sized tables.

6.2 Exp. 1 - Pipelined GPU Join

6.2.1 Comparison with a Blocking GPU Baseline: In this section, we look at the benefits of the proposed pipelining model in comparison to a distributed blocking GPU join (GPJ-B). The blocking join takes a sequential approach where the two tables are only joined on the GPU once all data has been shuffled. We use the state-of-the-art single-node GPU join [32] to realize the blocking join on each node. The histograms and shuffling (re-partitioning) of the tables are based on [4] with minor modifications for enabling shuffling with GPUDirect, as we discuss next.

We execute GPJ-B both with and without GPUDirect to better show the effect of fast networks. As shown in Figure 9a, using GPUDirect greatly improves the acceleration potential of the GPU since we can leverage it without additional transfer costs. Moreover, we see that our pipelined GPU join, which also uses GPUDirect, can further improve over the blocking GPU join since with our pipelined join, we can overlap the join phases with data transfers.

Another significant advantage of our pipelined GPU join over the blocking GPU join is that for the blocking GPU join (GPJ-B), GPUDirect can only be used if both tables fit in the GPUs, as all input data must be accumulated before executing the join. In Figure 9b, we show this effect by increasing the probe-side table such that only the build-side table fits on the GPUs. As we can see, only GPJ-B without GPUDirect is supported in this case, whereas our pipelined GPU join can support arbitrary probe-side table sizes and, as such, provides a speedup of approximately 2×. Additionally, since GPJ-B performs multi-pass partitioning on the GPU to facilitate a faster join, it incurs a higher memory overhead, which further limits the table sizes that can be stored on the GPUs. In fact, for the workload in Figure 9b, GPJ-B consumes the complete GPU memory while GPJ only takes up half.

6.2.2 Comparison with CPU Baseline: We now evaluate our pipelined GPU join (GPJ) against a non-accelerated CPU baseline (CRJ). Both joins share the execution steps performed on the CPUs and therefore highlighting the benefits of GPU acceleration and pipelining.

Varying Table Sizes (Probe-side). As both joins can support arbitrarily sized probe-side tables, we first compare both joins using a fixed-sized build input table that is joined with a probe-side table of different sizes. More precisely, we used a workload with a build-side table of 32 GB (2×10^9 tuples with 16 bytes), fitting into the 4 GPUs and with probe-side tables up to 128 GB (8×10^9

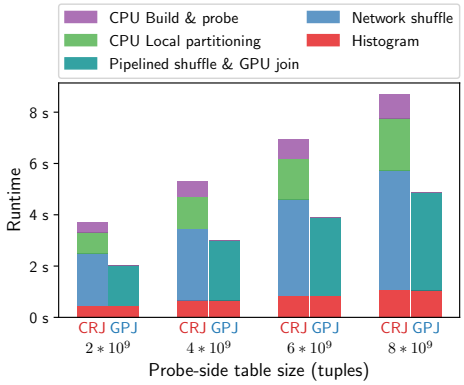


Fig. 10. Varying probe-side table size and a fixed build-side table size of 2×10^9 tuples on 4 nodes. Our approach of GPU-accelerated pipelined join (GPJ) outperforms CPU radix join (CRJ) in all cases.

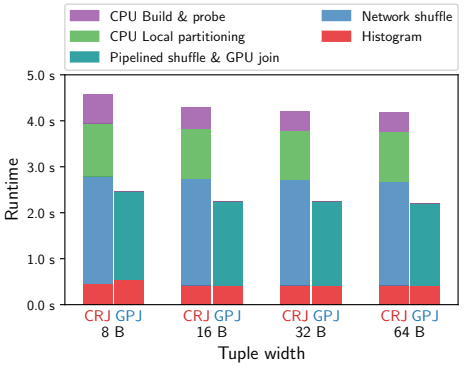


Fig. 11. Our approach (GPJ) also outperforms CRJ for varying tuple widths. Build & probe tables are fixed at 16 GB & 64 GB respectively.

tuples), thereby exceeding the total collective GPU memory that is 64 GB (disregarding the hash tables). Over the different probe-side table sizes, GPJ is up to 1.7× faster than CRJ.

The result can be seen in Figure 10, where we report the time each phase of the joins takes. As we can see, the GPU-accelerated join outperforms the CPU join in all cases. For the GPU-accelerated join, we do not report the time for the network shuffling, and GPU join execution separately due to the GPU pipelining; i.e., both phases are overlapped in this join. Interestingly, we can see that for GPJ, the shuffling and joining phases are shorter than only the shuffling phase of CRJ. The reason is that the RDMA writes performed by the sender CPU (one-sided) are going to the remote GPU and thus relieve pressure on the main memory. This is different for CRJ, where the CPU cores and the NIC are both writing to the main-memory during the shuffling phase.

Varying Tuple Width. Next, we evaluate the GPJ and CRJ join using the same table sizes but with varying tuple widths ranging from 8 to 64 byte tuples. For this experiment, we fixed the table sizes to 16 GB and 64 GB for the build- and probe-side tables, respectively. Our motivation for this experiment is to show the effect of different tuple widths since CPUs and GPUs use different execution and memory models and thus potentially affect the runtime differently.

The results of this experiment are shown in Figure 11. Similar to the above experiment, GPJ clearly outperforms the CPU join (CRJ) in all tuple widths, essentially due to the pipelining. Moreover, both join algorithms inhibit similar behavior with a small decrease in runtime for wider tuples. The reason is that as the tables with wider tuples contain fewer tuples, the overall per-tuple overhead decreases, which leads to a decrease in runtime.

Scalability of Joins. Finally, we show how the proposed pipelined GPU-accelerated join scales against the CPU baseline over an increasing number of database nodes. To show this, we simulate a cluster of up to 10 virtual nodes using 5 physical nodes where we use each of the two NUMA regions. Those regions are each equipped with their own GPU and RDMA NIC and therefore function as independent database nodes as all data shuffled between the logical nodes is transferred over the InfiniBand network (and not the cross-NUMA interconnect). For the workload, we used a build-side table of 300×10^6 tuples and a probe-side table of 1.2×10^9 tuples per node. As an example, for 10 nodes, a total of 48 GB $\bowtie$ 192 GB tables are joined.

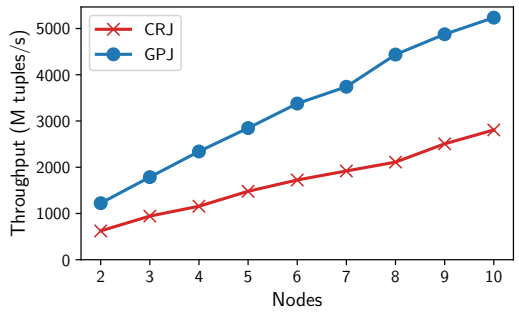


Fig. 12. Scale out experiment (GPJ vs. CRJ) with build-side table of 300×10^6 and probe-side table of 1.2×10^9 tuples per node. Performance increase of $\sim 4.4\times$ is observed when nr. of nodes increases from 2-10.

Figure 12 shows the results with the tuple throughput as the main metric. As we can see, both joins have linear scalability up to the tested 10 nodes. For both cases, we see a performance increase of about 4.4× from 2 to 10 nodes.

Why not pipelining a CPU Join? Finally, we want to make the case why pipelining the network shuffling phase with the subsequent join is not an optimization to apply to a distributed CPU join as well. First, the nature of the shuffling operation is very memory intensive. Thus, all the CPU cores are reading the table and writing out the tuples to partitions, while the NIC is also reading the data to send over the network and writing incoming data from other nodes. As such, first, any additional memory operations (as in our pipelining approach) will further slow down the shuffling and thus the overall join execution since, as reported before, the shuffling is the limiting factor. Second, profiling the shuffling phase of CRJ supports this claim and reports not only over 10% of DRAM memory assesses are stalling but also 100% CPU core utilization which indicates that no idle (free) CPU resources would be available for a pipelined (overlapped) execution.

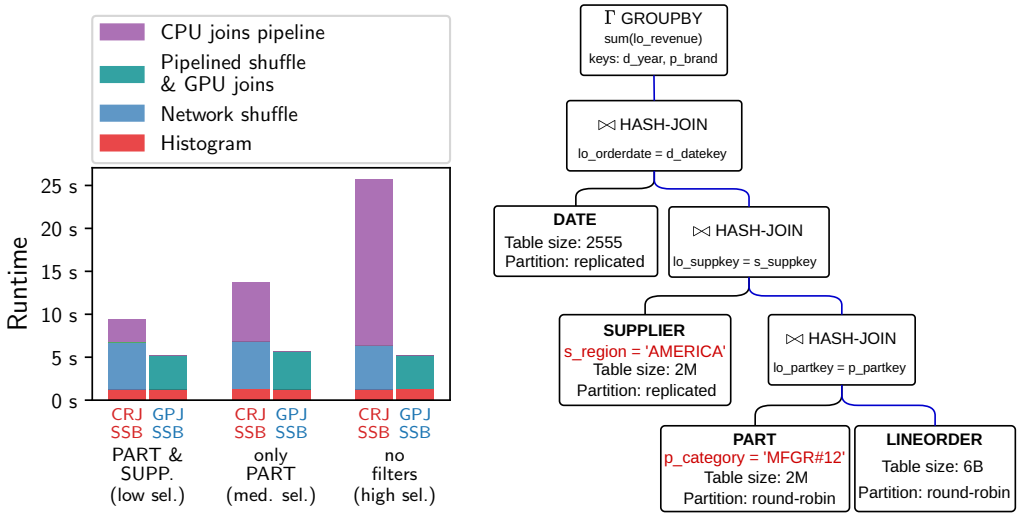
6.3 Exp. 2 - Complete Queries

In this experiment, we show the benefit of our pipelined join for a full query with multiple operations. As we explained before in Section 4.3, when running full queries, we allow multiple joins to be *chained* together (e.g., to chain multiple probe steps for a multi-way join in one pipeline on the GPU). The main intuition why chaining on GPUs is beneficial is that GPUs typically have an abundance of processing power in comparison to their i/o speeds. In fact, when executing a single join, as in the previous experiments, there are still untapped computational resources left for chaining multiple operators together.

To show these effects of chaining, in this experiment, we use query 2.1 and 3.1 of the Star-Schema-Benchmark (SSB) that both involve three hash-joins and an aggregation. For comparison, we run two variants of the queries: one using only the pipelined join GPJ (GPJ-SSB) with the aggregation also on the GPU, and one that runs completely on CPUs using CRJ (CRJ-SSB). For both GPJ-SSB and CRJ-SSB, we use the same execution strategy, where we first build the hash tables on the dimensions tables and then chain together the probing of the LINEORDER tuples into these hash tables, followed by a final aggregation.[3]

Effect of Selectivities. We first execute the SSB query 2.1 with 3 different filter settings, which results in different intermediate result sizes. The query plan is shown in Figure 13b, along with which filters (marked with red) are changed to generate different join selectivities and therefore different

[3]PART and CUSTOMER are partitioned while DATE and SUPPLIER are replicated to enable chaining.



(a) Query Runtime w/ Different Filters

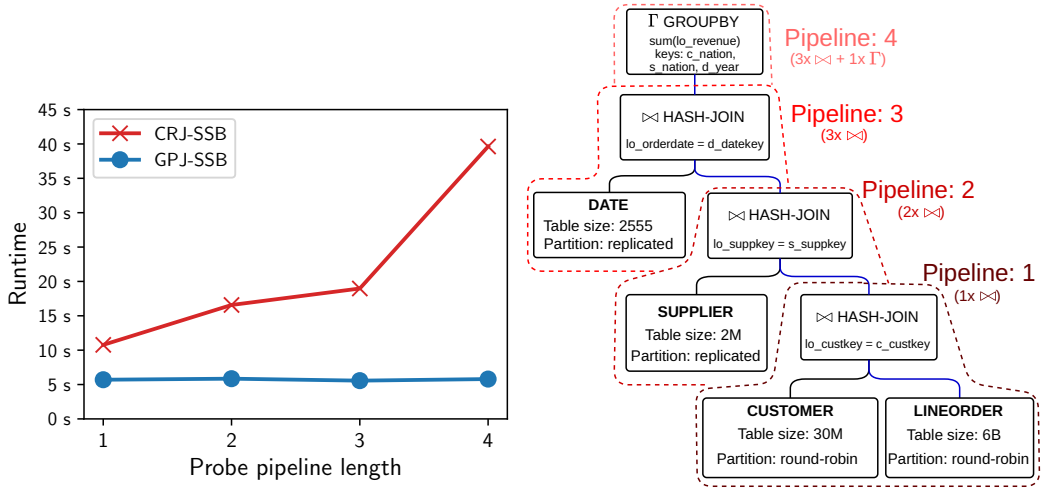
(b) SSB Query 2.1 Plan

Fig. 13. Execution (a) of SSB Query 2.1 (b) with SF 1000 on 4 nodes with different intermediate result sizes stemming from different dimension table filters. Probing of all hash tables and aggregation (blue edges) are chained both in CRJ-SSB and GPJ-SSB. Larger intermediate sizes incur extra runtime overhead for CRJ-SSB, while GPJ-SSB is unaffected due to pipelining on the GPU.

intermediate result sizes. We show the runtimes in Figure 13a. Here an interesting observation is that for our approach GPJ-SSB, a higher number of intermediate tuples does not introduce any runtime overhead due to the parallel execution and the higher processing power of the GPU, which completely hides the additional execution time under the network shuffle phase. Contrarily, we see for CRJ-SSB that larger intermediate sizes introduce higher runtime due to the sequential execution. For the largest intermediate results (i.e., no filters), our approach (GPJ-SSB) is, in fact, 5× faster than the non-accelerated CPU query execution.

Effect of Number of Joins. Next, we evaluate the influence of the length of the (chained) probe pipeline, i.e., the number of joins in the query on the runtime. Similar to the effect of different intermediate sizes, the number of joins also varies the amount of processing needed during the probing stage of the LINEORDER table. We base the experiment on SSB query 3.1 as shown in Figure 14b, but vary the number of joins in the query (pipeline length). For instance, a probe pipeline length of 2 will join together the LINEORDER with CUSTOMER and SUPPLIER and a length of 4 contains the complete query. As can be seen in Figure 14a, the GPU-accelerated query execution (GPJ-SSB) is not affected by the query size since the additional probing is all hidden under the network shuffling. This is in contrast to the CPU-only approach (CRJ-SSB) where more joins result in a higher runtime. For the full query without dimension table filters, a speedup of 6.8× can be observed over CRJ-SSB.

Comparable Resources. Choosing to accelerate a query on GPUs begs the question of whether the observed speedup outweighs the added cost of the GPUs. We thus now evaluate the same SSB queries as before using two hardware setups (with and without GPUs) that have approximately the same hardware cost. For our hardware, as detailed in Section 6.1, a machine with a GPU costs twice



(a) Query Runtime **(b)** SSB Query 3.1 Plan w/o Filters

Fig. 14. Execution (a) with a different number of joins of SSB Query 3.1 (b) with SF 1000, on 4 nodes. Probe-side joining is chained together both in CRJ-SSB and GPJ-SSB. With longer probe pipelines, the runtime of GPJ-SSB is unaffected due to its pipelined design resulting in a reduction in runtime by up to 6.8× against CRJ-SSB.

Table 1. Cost-comparable clusters w/ and w/o GPUs: In both setups (small, large) the hardware costs are comparable. For example, a 4-node cluster with CPUs-only costs as much as 2 nodes with GPUs.

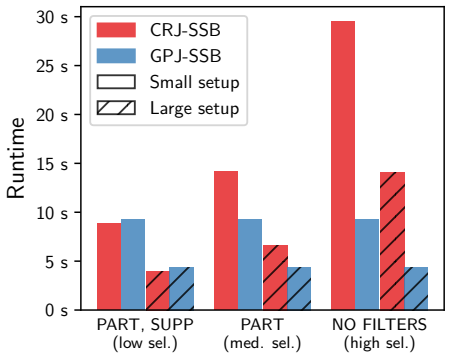
	Small setup	Large setup
GPU-accel (GPJ)	2× nodes w/ 2× GPUs	5× nodes w/ 5× GPUs
CPU-only (CRJ)	4× CPU nodes	10× CPU nodes

as much as the same machine without a GPU. To be more precise, the cost of one Tesla V100 GPU equals that of a server (including the CPU, memory, NIC as well as chassis without the GPU).[4]

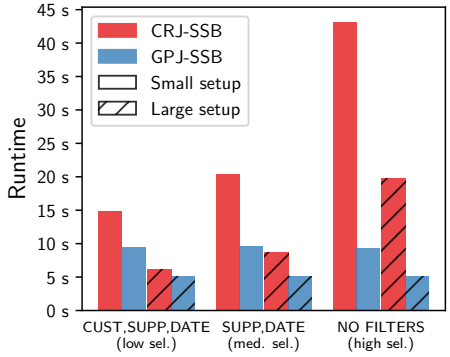
In this experiment we hence use two different *cost-balanced* hardware configurations, as shown in Table 1, that reflects the aforementioned relationship in cost; i.e., for the cluster without GPUs, we use twice as many machines as for a cluster with GPUs. To be more precise, we use a *small* setup that uses 4 CPU machines (w/o GPUs) and compare it to a cluster of 2 machines w/ GPUs. For the *large* setup, we use 10 CPU machines (w/o GPUs) vs. 5 machines w/ GPUs. The results of running the SSB queries 2.1 and 3.1 on these two cluster setups (small and large) with a comparable set of resources can be seen in Figure 15.

Interestingly, even though the CPU-only cluster (for both the small and larger setups) can make use of twice as many nodes as the cost-comparable cluster with GPUs, we observe that the GPU join (GPJ-SSB) can still outperform the CPU join (CRJ-SSB) (or is at least competitive). Clearly, compared to Figure 13a, where the GPJ-SSB can use the same number of nodes as the CRJ-SSB, the benefits are less pronounced. For example, the runtime of using only 2 GPU-nodes in Figure 15a for query 2.1 is double as high compared to the GPJ-SSB in Figure 13a, which uses 4 GPU-nodes.

[4]As hardware prices fluctuate, this experiment serves only as a rough comparison on balanced resources. Moreover, note that we use high-end Tesla V100 GPUs for the cost comparison and thus the ratio of CPU to GPU machines would be even more in favor of the GPU cluster in case we use less expensive GPU hardware.



(a) SSB Query 2.1 Runtime w/ Different Filters



(b) SSB Query 3.1 Runtime w/ Different Filters

Fig. 15. Query runtime with clusters that cause comparable cost as outlined in Table 1 for (a) SSB Query 2.1 and (b) SSB Query 3.1 (SF 1000). The advantages of the GPU join (GPJ) becomes more pronounced with larger intermediate results (i.e., higher selectivity).

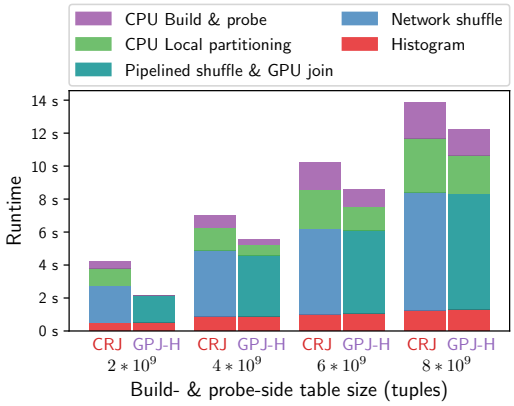


Fig. 16. GPJ-H vs. CRJ when scaling input tables on 4 nodes. With larger build-side tables, more data spills out on CPUs.

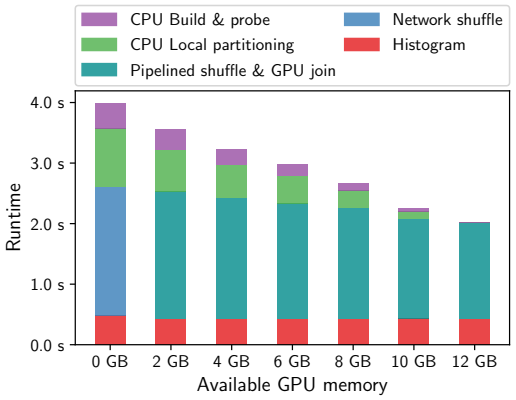


Fig. 17. Hybrid join (GPU-H) adapts to increase in available per-GPU memory allowing reduction in runtime.

However, we can still see in Figure 15a that the runtime of the CPU-only query (CRJ-SSB) linearly increases with larger intermediates while the GPJ-SSB can almost provide constant runtime since the GPU can make efficient use of its high degree of parallelism for larger intermediates. As such, for increasing intermediates the GPJ-SSB provides a significant speedup over the CRJ-SSB of up to 4× for Query 3.1 (no filters) even though GPJ-SSB is executed on a cluster with only half the nodes.

6.4 Exp. 3 - Hybrid Join Execution

In this experiment, we evaluate our hybrid GPU/CPU join algorithm (GPJ-H) on the effect of supporting arbitrarily larger tables than the GPU memory. With GPJ-H, different from GPJ, build-side tables that exceed the collective memory of the GPUs are supported by distributing the partitions across GPU and CPU memory.

Varying Table Sizes (Build & Probe-side). In this experiment, we increase the build- and probe-side tables from 32 GB to 128 GB. While using only 4 nodes with 1 GPU per node, this setup provides a total of 4×16 GB = 64 GB of GPU memory. With the smallest table size, the hash-tables for build-side tables can be fully stored in GPU memory, and thus GPJ-H is effectively the same as the GPJ.

Figure 16 shows the results of this experiment. With increasing table sizes, the runtime of both joins increases. For our hybrid join (GPJ-H), we see that with the increasing table sizes, more partitions will be assigned to CPUs resulting in an increased CPU runtime for GPJ-H as the partitions do not fit the GPU memory anymore. Still, with the hybrid execution, even for the largest tables size of 128 GB, we see a runtime benefit of around 2*s* in comparison to the CPU counterpart (CRJ). This is because, for the largest tables size, roughly a quarter of all tuples are joined on the GPUs and the rest on the CPUs. Thus, with the increasing table size and more partitions being assigned to CPUs, the benefit of pipelining diminishes, resulting in an increase in the runtime.

Varying GPU Memory. The hybrid execution of GPJ-H also allows a GPU-accelerated DBMS to adapt to different GPU memory sizes (and thus adapt to different available hardware). In this experiment, we analyze the effect of varying GPU memory sizes on the join runtime. As workload, we use fixed-size build- and probe-side tables of 2×10^9 tuples with 16 bytes and scale the available GPU memory from 0 B (pure CPU join) to the point where the hash tables (for the given workload) of the whole build-side table fit on the GPUs.

Figure 17 shows the result of this experiment of the GPJ-H over different GPU memory sizes. The main takeaway is that GPJ-H is able to efficiently adapt to different GPU memory sizes and speed up the performance with the increasing available GPU memory. Moreover, even with a small amount of available GPU memory, the hybrid join is able to utilize the GPUs, and hence reduce the join runtime.

6.5 Exp. 4 - Microbenchmarks

In the following, we show the results of three microbenchmarks.

Streaming Materialization. In this experiment, we compare the runtime of GPJ and CRJ with and without materialization enabled. As a setup, we used 4 nodes and 4 GPUs with a build-side table size of 1×10^9 tuples of 16 bytes. In case materialization is used, CRJ writes the result directly to the main memory, while GPJ uses the streamed materialization from GPU to CPU memory as described in Section 4.3. In case no materialization is used, we execute a COUNT aggregation on the join result.

Figure 18 shows the results. What can be seen is that with materialization, the runtime increases slightly both for GPJ and CRJ, which stems from the added write-load on the CPU memory in both cases. Importantly, for GPJ we see that materialization of the result into CPU memory only has minimal overhead, which is in a similar range as for CRJ.

Multiple Concurrent Joins. For OLAP queries, it is common to schedule multiple queries at once on the same cluster. Thus, in this experiment, we evaluate whether there is any inherent disadvantage to accelerating the join with GPUs versus the traditional CPU-only execution when executing multiple joins concurrently. As workload, we execute the same join operator using 200×10^6 and 2×10^9 tuples for build- and probe-side tables concurrently on a cluster of 4 nodes. Moreover, for each join, we allocate dedicated threads (of CPU or GPU cores); i.e., we split the resources depending on the number of concurrent queries that is a common scheme for OLAP queries.

As we can see in Figure 19, when increasing the number of concurrent joins, almost the same relative increase of runtime can be observed for CRJ and GPJ. However, since the absolute runtime of the GPJ is lower compared to the CRJ, as we already showed in the previous experiments, the gap in total gains increases with more concurrent queries (w.r.t. the elapsed runtime for all concurrent

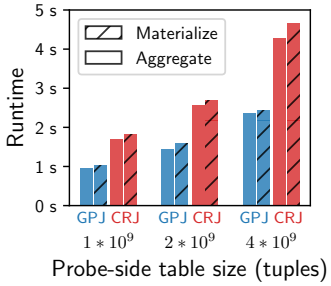


Fig. 18. Cost of materialization on 4 nodes with a build-side table size of 1×10^9 tuples and 100% join selectivity.

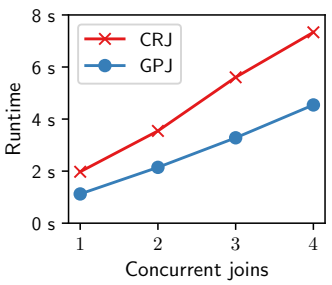


Fig. 19. Runtime of concurrent joins on 4 nodes using 200×10^6 and 2×10^9 tuples for build- and probe-side tables.

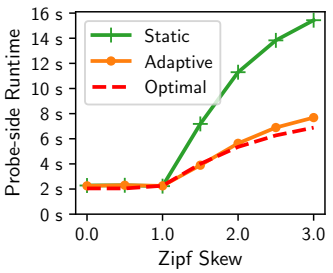


Fig. 20. Runtime for probe-side shuffling with skew on 4 nodes with 6.4×10^9 tuples with adaptive or static grid sizes.

joins). For example, for 4 concurrent joins, the elapsed runtime is approx. 4 seconds for the GPJ, while the CRJ requires almost 8 seconds. As such, the total benefits of the GPJ are also significant under concurrent query execution and there is no strategic disadvantage for the GPU when running queries concurrently.

Effect of Skew. Skewness is often occurring in datasets and especially in the form of foreign-key skew where the probe-side table has many tuples which joins with a small subset of tuples from the build-side table. This skew can sometimes be beneficial since it increases the locality for the heavy-hitters and allows for better caching. However, in the context of a partitioned join, it will result in uneven-sized partitions, and for our pipelined execution algorithm, it means that the bigger partitions on the GPU will receive tuples at a faster rate than others. As discussed in Section 4.2, we handle this by adaptively sizing the allocated resources (grid-sizes) for each partition in relation to the size of the partition, such that the processing speed of the big partitions can be adapted with respect to their sizes.

To show this effect, we plot the observed probe-side runtimes for different skew factors in Figure 20. For a skew of range 0 to 1.0, the runtime is unaffected since we distribute the partitions in a round-robin manner to nodes by the size of the partitions to even out the network skew. Even when using statically allocated GPU resources, the GPUs have enough processing power to handle this level of skew. From zipf 1.0 and up, however, the static approach severely slows down the GPU join execution as the processing of the heavy partitions leads to a straggling behavior. However, with our adaptive optimization, the processing at each GPU can be done approximately at line rate (for the given distribution). The efficiency of our approach can be also seen when comparing the adaptive runtime to the dashed red line, which shows the theoretical optimal runtime given the overall network skew resulting from some nodes receiving more tuples than others. With high enough skew, the size of the largest partition dominates (i.e., it is larger than the uniform share per node) and thus the theoretical optimal runtime also increases since the skew cannot be mitigated completely.

7 RELATED WORK

Single-node Joins: Acceleration of single-node joins with GPUs has been a well-studied topic over the last decade [16–19, 21, 22, 27, 28]. Sioulas et al. [32] investigated how to best utilize the GPU hardware for a single-node join by implementing and evaluating a range of different joins while considering different scenarios for tables larger than GPU memory. However, when tables

do not fit on the GPU, the slow PCIe interconnect limits the performance when compared to a CPU-only baseline.

More recent work evaluated larger-than-memory single-node joins on multi-GPU setups [28]. Lutz et al. [21, 22] evaluated the transfer bottleneck of GPU-accelerated single-node joins against the faster interconnect NVLink 2.0. They found that such interconnects can greatly increase the performance of the GPUs as co-processors both for joins smaller and larger than GPU memory. An interesting future route would be to combine GPUDirect and NVLink. However, currently RDMA NICs are only available with PCIe, and GPUDirect cannot be combined with Nvidia's interconnect NVLink.

Another approach by Shanbhag et al. [31] aims to remove the transfer overhead to the GPU by instead using the GPU as the main processor by storing the working set directly on the GPU. In this setup, analytical processing on the GPU greatly outperforms the CPU but still restricts the total working set sizes for main-memory DBMSs. Targeting a similar setup with GPUs as the main processor, Paul et al. [26] focuses on optimizing the communication paths of transfers between multiple GPUs through their multi-hop algorithm to maximize the cross-sectional bandwidth between GPUs.

Distributed Joins: In the context of high-speed networks, join processing for scale-out distributed DBMSs has been studied by a few works [3, 4, 11, 12]. Barthels et al. [3] implemented a distributed radix hash join over RDMA networks by utilizing efficient one-sided RDMA primitives. While the authors do not explore GPU-acceleration, many findings of their work are still applicable, such as the efficiency of one-sided RDMA.

The approach by Guo et al. [14] is closest to our work, which also explores distributed GPU joins over RDMA. However, they cover only more naïve blocking GPU joins. Moreover, they argue that remote transfer via GPUDirect from and to GPUs is less efficient than using RDMA with CPUs. Our findings are, however, different. Initially, we observed the same unbalanced bandwidth characteristics for RDMA between GPUs and CPUs. The reason for this is that when the NIC and GPU are not placed under the same PCIe switch, a suboptimal performance can be observed as also reported in [15].

8 CONCLUSION & FUTURE WORK

In this paper, we present two novel join algorithms for accelerating distributed joins on high-speed networks with GPUDirect RDMA. We show how our pipelined GPU join can speed up distributed joins up to 2× over a state-of-the-art CPU-based join while supporting arbitrary large probe-side tables. To accelerate joins with build-side tables larger than the collective GPU memory, we present a hybrid join that transparently leverages both GPUs and CPUs for joining the tables. In the context of complete queries, we show that pipelining and the support of successive operations in our distributed GPU join algorithm can additionally speed up the overall execution by up to 6× against a non-accelerated CPU-only version.

In the future, we aim to explore the combination of also involving SSDs for very data-heavy operations such as out-of-memory joins through GPUDirect Storage [23] which allows the GPU to directly access storage without relying on the main-memory.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037), the DFG Collaborative Research Center 1053 (MAKI) as well as hessian.AI, 3AI and gifts from Mellanox. In addition, we gratefully acknowledge support from the Federal Ministry of Education and Research (BMBF) under Grant No. 01IS22091.

REFERENCES

[1] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *IEEE ICDE*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[3] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *ACM SIGMOD* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1463–1475. https://doi.org/10.1145/2723372.2750547

[4] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 517–528. https://doi.org/10.14778/3055540.3055545

[5] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (mar 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[6] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-Accelerated Databases. In *ACM SIGMOD* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1891–1906. https://doi.org/10.1145/2882903.2882936

[7] Sebastian Breß and Gunter Saake. 2013. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proc. VLDB Endow.* 6, 12 (aug 2013), 1398–1403. https://doi.org/10.14778/2536274.2536325

[8] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *USENIX NSDI* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 401–414.

[9] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (April 2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[10] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *IEEE ICDE*. IEEE, 1477–1488. https://doi.org/10.1109/ICDE48307.2020.00131

[11] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. 2009. Spinning Relations: High-Speed Networks for Distributed Join Processing. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware* (Providence, Rhode Island) *(DaMoN '09)*. Association for Computing Machinery, New York, NY, USA, 27–33. https://doi.org/10.1145/1565694.1565701

[12] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. 2010. A Spinning Join That Does Not Get Dizzy. In *IEEE ICDCS (ICDCS '10)*. IEEE Computer Society, USA, 283–292. https://doi.org/10.1109/ICDCS.2010.23

[13] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *ACM SIGMOD* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[14] Chengxin Guo, Hong Chen, Feng Zhang, and Cuiping Li. 2019. Distributed Join Algorithms on Multi-CPU Clusters with GPUDirect RDMA. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 65, 10 pages. https://doi.org/10.1145/3337821.3337862

[15] Khaled Hamidouche, Akshay Venkatesh, Ammar Ahmad Awan, Hari Subramoni, Ching-Hsiang Chu, and Dhabaleswar K. Panda. 2015. Exploiting GPUDirect RDMA in Designing High Performance OpenSHMEM for NVIDIA GPU Clusters. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*. IEEE Computer Society, 78–87. https://doi.org/10.1109/CLUSTER.2015.21

[16] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (Dec. 2009), 39 pages. https://doi.org/10.1145/1620585.1620588

[17] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *ACM SIGMOD* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[18] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 889–900. https://doi.org/10.14778/2536206.2536216

[19] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware* (Scottsdale, Arizona) *(DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *USENIX OSDI* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 185–201.

[21] 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 *ACM SIGMOD* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[22] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling the Operator State on GPUs with Fast Interconnects. In *ACM SIGMOD*.

[23] NVIDIA. 2021. *GPUDirect RDMA*. NVIDIA. https://developer.nvidia.com/gpudirect

[24] NVIDIA. 2021. *GPUDirect RDMA Design Considerations - Synchronization and Memory Ordering*. NVIDIA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html#sync-behavior

[25] NVIDIA. 2021. *Mellanox OFED GPUDirect RDMA*. NVIDIA. https://www.mellanox.com/products/GPUDirect-RDMA

[26] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *ACM SIGMOD* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[27] Ran Rui, Hao Li, and Yi-Cheng Tu. 2015. Join Algorithms on GPUs: A Revisit after Seven Years. In *IEEE BigData (BIG DATA '15)*. IEEE Computer Society, USA, 2541–2550. https://doi.org/10.1109/BigData.2015.7364051

[28] 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 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[29] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *IEEE ICDE*. 1194–1205. https://doi.org/10.1109/ICDE.2016.7498324

[30] 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 *ACM SIGMOD* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 351–362. https://doi.org/10.1145/1807167.1807207

[31] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *ACM SIGMOD* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[32] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *IEEE ICDE*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[33] Lasse Thostrup, Jan Skrzypczak, Matthias Jasny, Tobias Ziegler, and Carsten Binnig. 2021. DFI: The Data Flow Interface for High-Speed Networks. In *ACM SIGMOD* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 1825–1837. https://doi.org/10.1145/3448016.3452816

[34] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. 2014. Red Fox: An Execution Environment for Relational Query Processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing Machinery, New York, NY, USA, 44–54. https://doi.org/10.1145/2544137.2544166

[35] 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

[36] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[37] Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2020. RDMA Communciation Patterns. *Datenbank-Spektrum* 20 (11 2020), 199–210. https://doi.org/10.1007/s13222-020-00355-7

[38] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *ACM SIGMOD* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 741–758. https://doi.org/10.1145/3299869.3300081

Received April 2022; revised July 2022; accepted August 2022