Hasso Plattner Institute
Data Engineering Systems Group

Master Thesis

Efficiently Joining Large Relations on Multi-GPU Systems with Modern Interconnects

Effizientes Joinen großer Relationen auf Multi-GPU-Systemen mit modernen Verbindungen

Tobias Maltenberger

Matriculation Number: 770387

Supervisor
Prof. Dr. Tilmann Rabl

2nd Reviewer
Dr. Michael Perscheid

Advisor
Ilin Tolovski

Submitted: 16.01.2024

Abstract

Growing data volumes present a mounting challenge to relational joins. GPUs have gained widespread adoption as database accelerators for operators such as joins due to their high instruction throughput and memory bandwidth. Most published GPU-accelerated joins are single-GPU algorithms that do not leverage multi-GPU platforms with high-speed P2P interconnects effectively. The few proposed multi-GPU joins either fail to exploit the fast P2P interconnects or lack support for large out-of-core data. In this thesis, we present a heterogeneous multi-GPU sort-merge join composed of three phases: a merge- or radix partitioning-based P2P-enabled multi-GPU *sort* phase, a parallel CPU-based multiway *merge* phase, and a hybrid *join* phase that combines a CPU merge path partition with a binary search-based multi-GPU join strategy. We evaluate our novel multi-GPU join on two platforms with fast NVLink- and NVSwitch-based P2P interconnects. We show that our join outperforms state-of-the-art CPU and GPU baselines regardless of the workload. It outperforms parallel CPU sort-merge and radix-hash joins up to 5.5× and 3.3×, respectively. Compared to non-P2P-enabled multi-GPU joins, it achieves speedups of 5.9× (sort-merge) and 2.5× (hybrid-radix). We measure that our join's hybrid join phase with overlapped copy and compute operations contributes as little as 24% to its end-to-end runtime. If the input relations are pre-sorted, it is 14.4× faster than the hybrid-radix join. Our join scales well with the number of GPUs and benefits from data skew with up to 12% shorter join durations.

Zusammenfassung

Wachsende Datenmengen stellen eine zunehmende Herausforderung für relationale Joins dar. GPUs haben aufgrund ihrer hohen Befehlsdurchsatzrate und Speicherbandbreite weite Verbreitung als Datenbankbeschleuniger für Operatoren wie Joins gefunden. Die meisten veröffentlichten GPU-beschleunigten Joins sind Single-GPU-Algorithmen, welche Multi-GPU-Systeme mit schnellen P2P-Verbindungen zwischen den GPUs nicht effektiv auslasten. Die wenigen vorgeschlagenen Multi-GPU-Joins nutzen entweder die Hochgeschwindigkeits-P2P-Verbindungen nicht oder unterstützen keine die GPU-Speicherkapazität übersteigenden Datenmengen. In dieser Arbeit präsentieren wir einen heterogenen Multi-GPU-Sort-Merge-Join, der aus drei Phasen besteht: einer P2P-fähigen und auf Mischung oder Fachverteilung beruhenden Multi-GPU-Sortierphase, einer hochparallelen CPU-basierten Mischphase und einer hybriden Verbundphase, die eine CPU-Mischpfad-Partitionierung mit einer auf binärer Suche aufbauenden Multi-GPU-Verbundstrategie kombiniert. Wir evaluieren unseren Multi-GPU-Join auf zwei Systemen mit schnellen NVLink- und NVSwitch-basierten P2P-Verbindungen. Wir zeigen, dass unser Sort-Merge-Join die Leistung moderner CPU- und GPU-Referenzalgorithmen unabhängig von der Arbeitsbelastung übertrifft. Er schneidet 5.5× beziehungsweise 3.3× besser ab als nebenläufige CPU-basierte Sort-Merge- und Radix-Hash-Joins. Im Vergleich zu nicht-P2P-fähigen Multi-GPU-Joins erzielt er Beschleunigungen von 5.9× (Sort-Merge) und 2.5× (Hybrid-Radix). Wir messen, dass die hybride Verbundphase mit überlappten Kopier- und Berechnungsvorgängen lediglich 24% zur Gesamtlaufzeit unseres Joins beiträgt. Wenn die Eingaberelationen vorsortiert sind, ist er 14.4× schneller als der Hybrid-Radix-Join. Unser Join skaliert gut mit der Zahl der GPUs und profitiert von Datenschiefe mit um bis zu 12% kürzeren Laufzeiten.

Contents

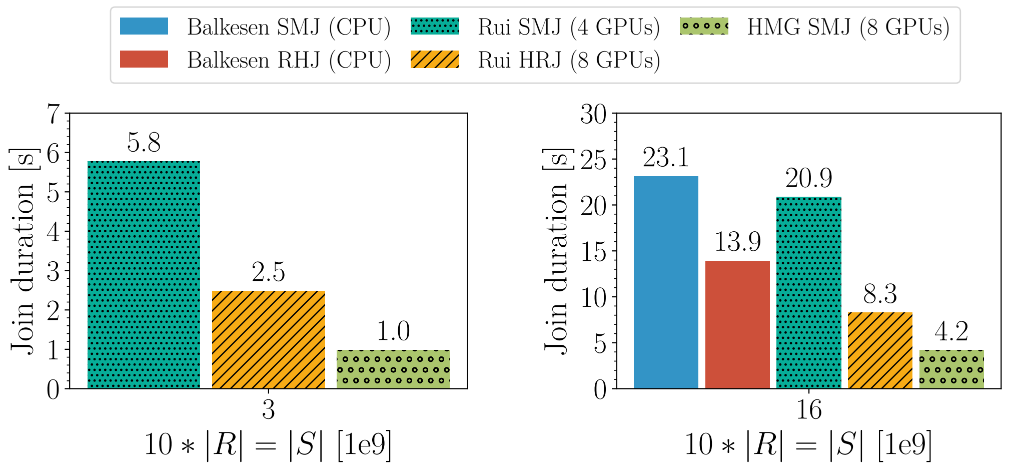
1 Introduction

The join is one of the fundamental operators of any relational database system. Unprecedented amounts of data make it increasingly challenging to process relational joins efficiently [35]. Therefore, researchers and engineers continuously adapt join algorithms to harness the latest advances in hardware technology [7, 12, 13, 39, 57]. Modern multi-core architectures led to sophisticated workload partitioning strategies, cache optimization techniques, and single instruction, multiple data (SIMD) operations for relational joins [8, 9, 14, 59, 68, 92]. Similarly, the rise of many-core graphics processing units (GPUs) inspired numerous GPU-accelerated joins [34, 48, 88, 98, 107]. Due to the high instruction throughput and memory bandwidth of GPUs [77, 79], these algorithms often outperform parallel CPU joins by an order of magnitude [98, 107]. Most of the published GPU-accelerated joins are single-GPU approaches that leave the potential performance gain of joining across multiple GPUs connected via high-speed peer-to-peer (P2P) interconnects entirely untapped. Moreover, they assume that the input relations and all intermediate results fit completely into GPU memory. Although the on-chip GPU memory has increased over the past few years up to 80 GB [77, 79], it still sets an upper limit on the size of the input relations that a single-GPU join can process.

Only very few multi-GPU approaches have been proposed. Paul et al. describe a partitioned multi-GPU hash join based on an adaptive multi-hop routing strategy for efficient P2P data transfers between connected GPUs [89]. By evenly distributing the input relations and processing the relational join across multiple GPUs, the authors fully utilize the GPUs' compute power and P2P interconnect bandwidth but only soften the upper ceiling on the amount of data that can be processed. Rui et al. present two multi-GPU join algorithms for large out-of-core data: a sort-merge join and a hybrid-radix join [97]. The sort-merge join operates in two phases. First, it sorts chunks of the input relations that fit into GPU memory on the GPUs, partitions the sorted chunks through a parallel merge path partitioning in main memory, and merges the partitions concurrently across the GPUs. Second, it partitions the sorted input relations again and joins the partitions on the GPUs. Consequently, the sort-merge join transfers the data over two times via the CPU-GPU interconnects. The hybrid-radix join partitions the input relations into disjoint buckets through radix partitioning and joins the buckets on the GPUs. Although both out-of-core joins break the upper limit on the input relation sizes, neither harnesses the high-bandwidth P2P interconnects between the GPUs, which facilitate reducing the data transfers over the typically slower CPU-GPU interconnects [61, 62, 67]. Hence, the imperative arises to develop novel multi-GPU join algorithms that fully exploit modern multi-GPU systems with fast interconnects.

In this master thesis, we explore the design space of efficient GPU-accelerated join algorithms. We propose a heterogeneous multi-GPU sort-merge join for large out-of-core data that utilizes the high-speed P2P interconnects of modern multi-GPU platforms. It comprises a merge- or radix partitioning-based multi-GPU *sort* phase, a parallel CPU *merge* phase, and a hybrid *join* phase that employs a CPU merge path partition strategy and executes a binary search-based merge-join kernel across multiple GPUs. Our implementation features various data transfer optimizations and utilizes state-of-the-art CPU and on-GPU sort, merge, and partition primitives determined through micro-benchmarks. We evaluate the performance of our multi-GPU sort-merge join on high-performance computing (HPC) systems with fast NVLink 2.0, NVLink 3.0, and NVSwitch interconnects such as the IBM AC922 and NVIDIA DGX A100 [49, 82]. We compare its total runtime for two workloads against that of state-of-the-art CPU and GPU baselines: the multi-threaded CPU sort-merge and radix-hash joins by Balkesen et al. and Rui et al.'s non-P2P-enabled multi-GPU sort-merge and hybrid-radix join, respectively [7, 97]. We also study the impact of our heterogeneous multi-GPU join's three algorithm phases on its execution time and analyze its scalability for increasing numbers of GPUs and robustness against different selectivity and data skew factors.

We show that our novel heterogeneous multi-GPU sort-merge join (HMG SMJ) consistently outperforms the CPU and GPU baselines. On the IBM AC922, it is up to 5.9× and 2.5× faster than the sort-merge join and hybrid-radix join by Rui et al. (see Figure 1a). On the NVIDIA DGX A100, it achieves up to 5.0× (sort-merge) and 2.0× (hybrid-radix) shorter join durations than the multi-GPU baselines (see Figure 1b). Compared to Balkesen et al.'s CPU sort-merge and radix-hash join, respectively, it yields speedups of 5.5× and 3.3×. We measure that our multi-GPU join's sort phase contributes as much as 76% to its execution time. We observe that the radix partitioning-based sort strategy is between 15% and 20% more efficient than the merge-based strategy. Once either of the two input relations exceeds the combined GPU memory capacity, we notice a performance cliff as the CPU merge phase saturates the main memory bandwidth. Our join surpasses the fastest CPU and GPU baseline's performance even with a parallel CPU merge phase up to 2.0× and 1.2×, respectively. We find that the join phase has as little as 24% impact on our join's runtime. If both of the input relations are pre-sorted, it reaches speedups of 14.4× (IBM AC922) and 9.2× (NVIDIA DGX A100) over the hybrid-radix join. We demonstrate that our sort-merge join scales well with the number of GPUs and benefits from skew with up to 12% shorter join durations. Thus, it is an excellent fit as an operator for GPU-accelerated database systems.



(a) IBM AC922 **(b) NVIDIA DGX A100**

Figure 1: Join comparison for input relations with 16-byte tuples

With this thesis, we make the following contributions.

1. We propose a novel heterogeneous multi-GPU sort-merge join that harnesses the fast P2P interconnects of modern multi-GPU accelerator platforms and natively supports handling large out-of-core data.
2. We publish our high-performance implementation that utilizes state-of-the-art CPU and single-GPU sort, merge, and partition primitives.
3. We conduct in-depth experiments for two workloads to study our sort-merge join's efficiency in joining large input relations on modern multi-GPU systems with high-bandwidth interconnects.

The remainder of this master thesis is structured as follows. In Section 2, we outline the fundamental concepts of GPU architectures, GPU interconnects, the CUDA programming model, and the canonical sort-merge join algorithm. Section 3 explains the sort, merge, and join phases of our heterogeneous multi-GPU sort-merge join. In Section 4, we evaluate the end-to-end performance of our multi-GPU-accelerated join on modern multi-GPU systems with fast P2P interconnects. Section 5 discusses the findings of our evaluation, while Section 6 provides an overview of the related work. In Section 7, we conclude our research effort.

2 Background

In this section, we provide information about GPU architectures, GPU interconnects, the CUDA programming model, and the sort-merge join.

2.1 GPU Architectures

GPUs offer massively parallel compute capabilities. Unlike CPUs that are designed to simultaneously execute a few tens of threads as fast as possible and hide the memory access latency through data caches and control flows, GPUs are optimized to run thousands of threads in parallel with lower single-thread performance but considerably higher instruction throughput than CPUs [85]. By way of illustration, the two top-of-the-line GPUs, NVIDIA V100 (Volta) and NVIDIA A100 (Ampere), achieve 32/64-bit floating-point throughput rates of up to 15.7/7.8 TFLOPS and 19.5/9.7 TFLOPS, respectively [77, 79]. GPUs are built around an array of multi-threaded streaming multiprocessors (SMs), as depicted in Figure 2. The NVIDIA V100 and NVIDIA A100 comprise 80 and 108 SMs, each containing 64 INT32 and FP32 as well as 32 INT64 and FP64 cores [74, 76]. SMs employ the single instruction, multiple threads (SIMT) architecture. Instructions are pipelined to leverage instruction-level parallelism (ILP) within a single thread and thread-level parallelism (TLP) through simultaneous multithreading. SMs execute threads in groups of 32 parallel threads called *warps* [63].

In addition to a many-core compute architecture, GPUs offer a high-bandwidth memory hierarchy comprising *off-chip* and *on-chip* memory [85]. Off-chip memory consists primarily of global high-bandwidth memory (HBM). In the case of the NVIDIA V100 and NVIDIA A100, the maximum bandwidth rate of global memory is 900 GB/s and 1555 GB/s, respectively [77, 79]. Usually, the capacity of global memory is orders of magnitude smaller than that of main memory (e.g., 32 GB for the NVIDIA V100 and 40 GB for the NVIDIA A100). Since global memory is only accessible via aligned 32-, 64-, or 128-byte memory transactions, warps coalesce adjacent memory accesses from parallel threads into as few memory transactions as possible to improve transfer efficiency [44]. The L2 cache further hides the access latency of global memory by caching loads and stores to it. On the NVIDIA V100, its capacity is 6 MB, while on the NVIDIA A100, it is 40 MB [74, 76]. On-chip memory per SM includes low-latency shared memory as well as the L1 cache and the register file (see Figure 2). Typically, shared memory serves as user-managed scratchpad memory, while the L1 cache transparently hides the global memory access latency

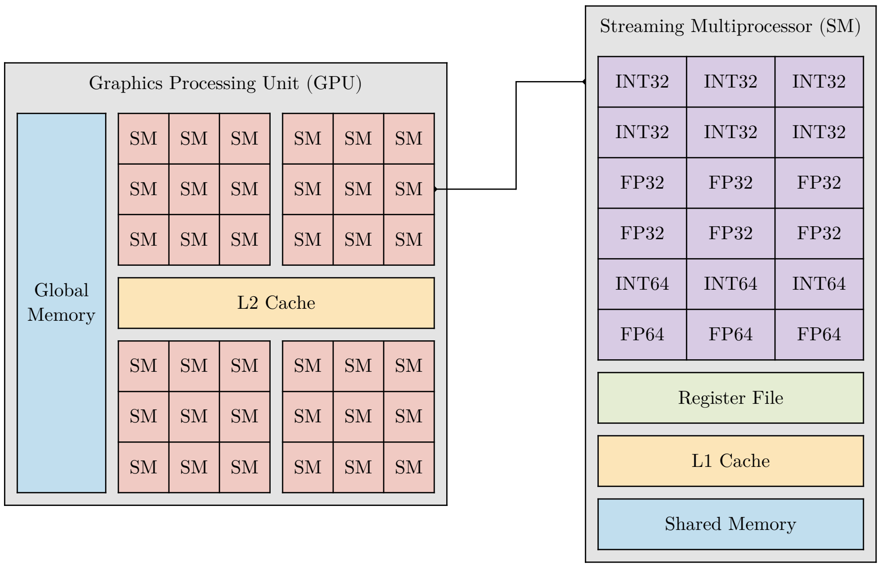


Figure 2: GPU hardware architecture

of all parallel threads executed by the SM [45]. On the NVIDIA V100 and NVIDIA A100, shared memory and the L1 cache are physically combined on each SM with a total capacity of 128 KB and 196 KB, respectively [19, 60]. Moreover, the register file size per SM is 256 KB on both data center GPUs.

2.2 GPU Interconnects

GPUs are attached to the main memory controller via CPU-GPU interconnects. Modern high-performance computing (HPC) systems have multiple GPUs connected through peer-to-peer (P2P) interconnects. The interconnect topology greatly affects the performance of GPU-accelerated applications [61, 62, 90].

Traditionally, PCIe has been the standard CPU-GPU and P2P interconnect. It is a serial communication bus composed of up to 16 bi-directional *lanes* per *link* [69]. PCIe 3.0 lanes offer data transfers at a rate of 1 GB/s. One PCIe 3.0 link reaches, therefore, a theoretical bandwidth of 16 GB/s per direction. PCIe 4.0 lanes provide a peak throughput of 2 GB/s. The uni-directional bandwidth of one PCIe 4.0 link is 32 GB/s. If multiple GPUs are connected to the same PCIe link via a *switch*, the total bandwidth is shared between the GPUs during concurrent data transfers [69].

In recent years, hardware vendors have introduced high-bandwidth interconnects to enable faster CPU-GPU and P2P communication. NVLink is a bi-directional point-to-point interconnect by NVIDIA [73]. NVLink 2.0 achieves a data transfer rate of 25 GB/s per link in each direction [74]. NVLink 2.0-enabled GPUs (e.g., NVIDIA V100) support up to six links. Consequently, the peak bandwidth of P2P data transfers between two NVIDIA V100 is 150 GB/s per direction. NVLink 3.0 has a uni-directional bandwidth of 50 GB/s per link [76]. NVLink 3.0-powered GPUs (e.g., NVIDIA A100) feature twelve NVLink 3.0 links and offer P2P data transfer rates of up to 300 GB/s between two GPUs. Although the technology is commonly utilized for P2P interconnects between GPUs [20], the IBM AC922 harnesses NVLink 2.0 also for its CPU-GPU interconnects [49]. NVSwitch is an NVLink-based switch for non-blocking, all-to-all P2P communication between up to 16 GPUs by NVIDIA [75]. NVLink 2.0- and NVLink 3.0-powered NVSwitch enables a peak bandwidth of 150 GB/s and 300 GB/s, respectively, between any two GPUs of a multi-GPU system simultaneously [54]. The NVIDIA DGX A100 employs NVLink 3.0-based NVSwitch for its P2P interconnects [82].

Most multi-GPU accelerators are dual-socket non-uniform memory access (NUMA) systems with equally many GPUs connected to each socket. On such platforms, data transfers between the local NUMA node and the GPUs of the remote NUMA node involve traversing the CPU-CPU interconnect [66]. In a process called *staging*, the data is moved from local to remote main memory via the CPU-CPU interconnect and, subsequently, from remote main memory to GPU memory via the CPU-GPU interconnect [55]. Copying data between GPUs without P2P interconnects attached to different NUMA nodes entails staging as well. Commercially available CPU-CPU interconnect technologies include IBM X-Bus (XB), AMD Infinity Fabric (IF), and Ultra Path Interconnect (UPI) by Intel [3, 5, 17, 52].

In times when PCIe was state of the art in interconnect technology, researchers suggested that GPU-accelerated database operations cannot efficiently scale to large out-of-core data due to the *data transfer bottleneck* caused by low-bandwidth, high-latency CPU-GPU interconnects [23, 30, 105, 117]. Since fast interconnects such as NVLink and NVSwitch have emerged, GPU-based join algorithms that significantly outperform their CPU baselines for large input relations have been proposed [64, 65]. Nevertheless, considering the interconnect topology in the design of GPU-accelerated database operations is crucial, especially on dual-socket multi-GPU systems with heterogeneous CPU-CPU, CPU-GPU, and P2P interconnects. On such systems, utilizing the compute power of both multi-core CPUs and many-core GPUs can mitigate the data transfer bottleneck [26, 31, 91, 107].

2.3 CUDA Programming Model

CUDA is a general-purpose GPU programming model and interface by NVIDIA [85]. At its core, CUDA provides abstractions as C++ language extensions and a runtime library for writing scalable GPU-accelerated applications.

The CUDA programming model regards threads as the lowest level of abstraction for performing a GPU computation or memory operation and assumes that they are executed on a physically separate *device* (GPU) that operates as a co-processor to the *host* (CPU). CUDA extends the core C++ language with *kernels*. A kernel is a device function that is executed in parallel by multiple threads. Up to 1024 threads are grouped into a one-, two-, or three-dimensional *block* that shares the limited on-chip memory of a single SM core [19, 60]. However, multiple equally shaped thread blocks may run a kernel concurrently and independently on different SM cores. Besides that, thread blocks are organized into a one-, two-, or three-dimensional *grid*, as illustrated in Figure 3. The *execution configuration* syntax allows for specifying the total number of threads that run a kernel (i.e., threads per block times blocks per grid) [46]. During the execution of a kernel, each thread is uniquely identifiable via the built-in 3-component variables: `blockIdx`, `blockDim`, and `threadIdx`. If a kernel operates on a one-dimensional vector, the kernel-wide unique *thread identifiers*, commonly used to partition the data equally across all threads [40], are calculated through `blockIdx.x * blockDim.x + threadIdx.x`. The threads comprising a block may cooperate by exchanging data through shared memory and synchronizing their kernel execution via the intrinsic barrier function `__syncthreads`. All threads have access to linear global memory.

The CUDA toolkit includes the `nvcc` compiler to translate kernels into host- and device-compatible binary code. Moreover, it provides a C++ runtime library that exposes host functions for allocating, deallocating, and copying between host and device memory, managing multiple devices attached to the same host, and executing host and device operations asynchronously. CUDA manages concurrent operations through sequences of commands called *streams*. Although the commands issued on a stream are executed in order, the commands of different streams may get executed out of order. CUDA offers synchronization primitives to ensure the successful completion of all commands issued on a stream [47]. Since many GPUs support the concurrent execution of copy and compute operations, a well-known performance optimization is to overlap streams [42].

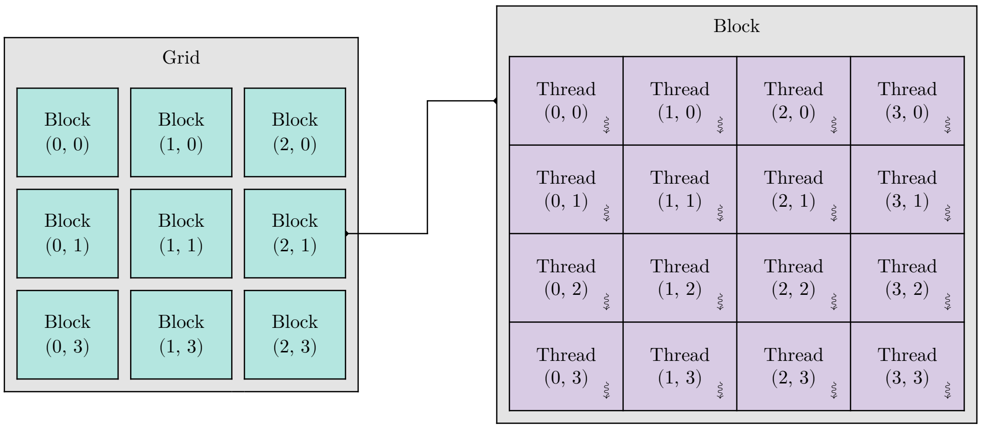


Figure 3: CUDA thread hierarchy

2.4 Sort-Merge Join

Given two input relations R and S, the join of R and S is defined as $R \bowtie_{r(a)\,\theta\,s(b)} S$, where $r(a)\,\theta\,s(b)$ denotes the join predicate [72]. θ is the condition that must hold between the attributes a and b of R and S, respectively. Generally, the θ operator can be $\{=, \neq, <, >, \leq, \geq\}$. If θ is the equality operator, the join is referred to as an *equi-join*, whose output relation Q is defined as $Q = \{t \mid t = rs \wedge r \in R \wedge s \in S \wedge t(a) = t(b)\}$. Put differently, Q contains tuples t consisting of two parts, r and s, where r is a tuple in R and s is a tuple in S. Furthermore, for each tuple t, the values of the join attributes $t(a)$ from r and $t(b)$ from s are equal.

One technique to implement an equi-join is the *sort-merge join* [118]. It consists of two stages: sort and merge (see Algorithm 1). In the sort stage, the algorithm sorts the input relations R and S based on the values of the join attributes a and b to efficiently locate groups of tuples with identical join attribute values (Lines 2 and 3). In the merge stage, the algorithm scans the two input relations R and S sequentially while looking for qualifying tuples r and s with equal join attribute values $r(a)$ and $s(b)$. It starts the two scans at the first tuples of each input relation (Lines 6 and 7) and advances the scan of R as long as the current R tuple's value of the join attribute a is less than its counterpart of the S tuple currently under consideration (Lines 9 and 10). Analogously, the algorithm advances the scan of S while the value of the join attribute b in the current S tuple is less than that in the current R tuple (Lines 11 and 12). It alternates between the R and S scans until it finds a tuple r, belonging to R, and a tuple s, belonging to S, with $r(a) = s(b)$.

Algorithm 1: Canonical sort-merge join

```
 1: function SORT_MERGE_JOIN($R$, $S$, $a$, $b$)
 2:     $sort(R, a)$
 3:     $sort(S, b)$
 4:
 5:     $Q \leftarrow \emptyset$
 6:     $r \leftarrow first(R)$
 7:     $s \leftarrow s' \leftarrow first(S)$
 8:     while $r \neq eof \wedge s \neq eof$ do
 9:         while $r(a) < s'(b)$ do
10:             $r \leftarrow next(R, r)$
11:         while $r(a) > s'(b)$ do
12:             $s' \leftarrow next(S, s')$
13:         $s \leftarrow s'$
14:         while $r(a) = s'(b)$ do
15:             $s \leftarrow s'$
16:             while $r(a) = s(b)$ do
17:                 $insert(Q, rs)$
18:                 $s \leftarrow next(S, s)$
19:             $r \leftarrow next(R, r)$
20:         $s' \leftarrow s$
21:
22:     return $Q$
```

In that case, the algorithm inserts the concatenated tuple rs into the output relation Q (Line 17). However, several tuples of R might have the same join attribute value as the currently considered R tuple r (i.e., belong to the same r group). Similarly, multiple S tuples can belong to the current s group. Thus, the algorithm matches all R tuples of the r group with all S tuples of the s group before resuming the R and S scans at the tuples following the r and s groups (Lines 13 to 20).

Over the past decades, parallel adaptations of the canonical sort-merge join have been proposed to exploit the thread- and data-level parallelism of modern multi-socket, multi-core CPU architectures with vectorized SIMD instructions [2, 6, 59]. Several GPU-accelerated sort-merge joins have been published [32, 48, 98]. Besides, researchers have periodically re-evaluated the relative performance between the sort-merge join and the hash join – often with contradictory findings [6, 27, 59, 103]. In a real-world database system, sort-merge joins are particularly useful for harnessing and preserving *interesting orders* during query execution [106].

3 Algorithm

In this section, we present a novel heterogeneous multi-GPU sort-merge join for large out-of-core data exceeding the combined GPU memory capacity. Our algorithm comprises a multi-GPU-accelerated merge- or radix partitioning-based *sort* phase (see Section 3.1), a parallel CPU-based multiway *merge* phase (see Section 3.2), and a hybrid *join* phase that combines a CPU merge path partitioning approach with a multi-GPU-accelerated join processing approach (see Section 3.3). It harnesses the high-speed P2P interconnects of modern multi-GPU systems for efficient inter-GPU communication and their multi-core CPU and many-core GPU compute capabilities to mitigate the data transfer bottleneck. It works as follows.

Sort Phase. Our multi-GPU join partitions the tuples (i.e., key-value pairs) of the two input relations R and S into *chunksets* that fit into the combined GPU memory of all g GPUs and splits each of the k_R and k_S chunksets into g equal-sized *chunks*. After that, for each chunkset, it copies the individual chunks to the GPUs, brings their tuples in globally ascending order by key across the g GPUs through a merge- or radix partitioning-based sort algorithm, and copies them back into main memory. In the merge-based sort approach, the chunks' tuples are sorted locally by key on each GPU and merged recursively across the g GPUs in multiple stages that entail selecting pivots and swapping blocks of tuples between subsets of all GPUs over the P2P interconnects. In the radix partitioning-based sort approach, each GPU partitions its chunk's tuples based on their key's most significant bits into buckets, exchanges select buckets with other GPUs in an all-to-all P2P bucket exchange, and sorts all buckets locally by key. Our multi-GPU sort-merge join overlaps the data transfers to and from the GPUs to saturate the CPU-GPU interconnects' bi-directional bandwidth regardless of the sort strategy.

Merge Phase. Once R and S reside k_R- and k_S-sorted (i.e., sorted within each of the k_R and k_S chunksets) in main memory, it merges the chunksets of each input relation. More specifically, it constructs a zero-copy zip iterator for the keys and values of each chunkset in R and S, respectively, and brings the elements of the zip iterators in ascending order by key through a highly parallel CPU-based multiway merge algorithm. If $k_R = 1$ or $k_S = 1$ (i.e., R or S consists of only one chunkset), the merge phase for the corresponding input relation is skipped.

Join Phase. Our heterogeneous multi-GPU join then divides the fully sorted input relations R and S into g correlated *partitions* that can be joined independently via a CPU-assisted merge path partition strategy. Since the size of the partition pairs might exceed the GPUs' global memory capacity, they are further divided into smaller correlated *subpartitions*. Subsequently, our multi-GPU-accelerated algorithm joins the keys of the disjoint subpartition pairs for each of the g correlated partitions across all g GPUs. It employs a pipelined copy-compute strategy that entails simultaneously copying the keys of a subpartition pair into global memory, executing the binary search-based in-core merge join on the keys of a subpartition pair to produce matching key ranges, and copying a set of matching key ranges back into main memory. Once all correlated partitions have been joined independently across the g GPUs, the join tuples of R and S are materialized in parallel on the CPU based on the previously identified matching key ranges.

3.1 Sort Phase

In this subsection, we describe our sort-merge join's multi-GPU-accelerated sort phase. It orders the tuples of the two input relations R and S by key in chunksets comprised of equal-sized chunks that are sorted across the g GPUs via a merge- or radix partitioning-based sort algorithm. Although both sort strategies utilize the high-speed P2P interconnects of modern multi-GPU platforms for inter-GPU data exchanges, they differ in how the data are exchanged between the GPUs. While the merge-based approach repeatedly shuffles blocks of tuples between pairwise subsets of all GPUs, the radix partitioning-based approach swaps all tuples simultaneously among all g GPUs. Common to both multi-GPU sort algorithms is the mechanism to partition R and S into k_R and k_S chunksets, respectively, consisting of g equal-sized chunks that fit into the global GPU memory of any of the g GPUs. Moreover, the two sort strategies share the logic to interleave data transfers of chunks to and from the GPUs with in-core compute operations on chunks utilizing two separate buffers for the keys and values of R and S, respectively.

3.1.1 Multi-GPU Merge Sort

The merge-based multi-GPU sort approach extends the algorithm by Tanasic et al. with support for key-value pairs and large out-of-core data [110]. Once the chunks constituting a chunkset have been copied to the g GPUs, they are sorted locally by key on each GPU through a state-of-the-art single-GPU sort primitive. Afterward, they are merged globally by key across all g GPUs in a sequence of pivot selections, block exchanges, and comparison-based single-GPU merge primitive executions. Finally, the sorted chunks are copied back into main memory.

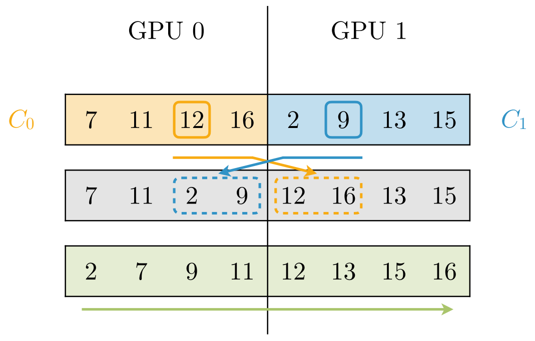


Figure 4: Block shuffling for $g = 2$ GPUs

On-GPU Chunk Sorting. A high-performance, low-overhead single-GPU sort primitive is required for efficiently sorting the chunks' tuples locally by key. Most on-GPU sort algorithms are parallel adaptations of either merge sort with a time complexity of $O(n * \log(n))$ or radix sort with a time complexity of $O(n)$, where n denotes the number of tuples to sort [71, 101, 109]. Over the past decade, radix sort has established itself as the fastest algorithm, as its traditionally high demand for memory bandwidth has been reduced by algorithmic improvements and mitigated by the ever increasing GPU memory bandwidth [1, 70, 74, 76, 102, 109].

We evaluate two state-of-the-art on-GPU sort primitives on two multi-GPU systems for two billion 64-bit tuples with 32-bit keys and 32-bit values: a load-balanced merge sort from the accelerated CUDA C++ primitives library `mgpu` and a least-significant bit (LSB) radix sort from the parallel CUDA C++ algorithms library `thrust` [78, 84]. Our micro-benchmark shows that `thrust::sort_by_key` outperforms `mgpu::mergesort` up to 4.1× on the IBM AC922 and 5.2× on the NVIDIA DGX A100. We, therefore, utilize it as the single-GPU sort primitive in our multi-GPU merge sort implementation. The space complexity of the out-of-place LSB radix sort from the `thrust` library is $O(n)$ as it needs a secondary buffer for the key-value pairs and comes with an overhead of up to 128 MB. Since dynamic GPU memory allocations are very expensive [85], we pass our stack allocator operating on pre-allocated global memory to the on-GPU sort primitive.

Multi-GPU P2P Block Shuffling. Bringing the locally sorted chunks in globally ascending order by key across $g = 2$ GPUs requires a *merge stage* consisting of a pivot selection, a block exchange, and an on-GPU merge step (see Figure 4). In the pivot selection, we calculate a key-based pivot position p in the chunk C_1 and its mirrored position p' in the chunk C_0, where $p' = |C_0| - p$, so that the *first* p' keys in C_0 and the *first* p keys in C_1 are less than or equal to the *last* p keys in C_0 and the last p' keys in C_1. Our implementation uses an adapted binary search kernel

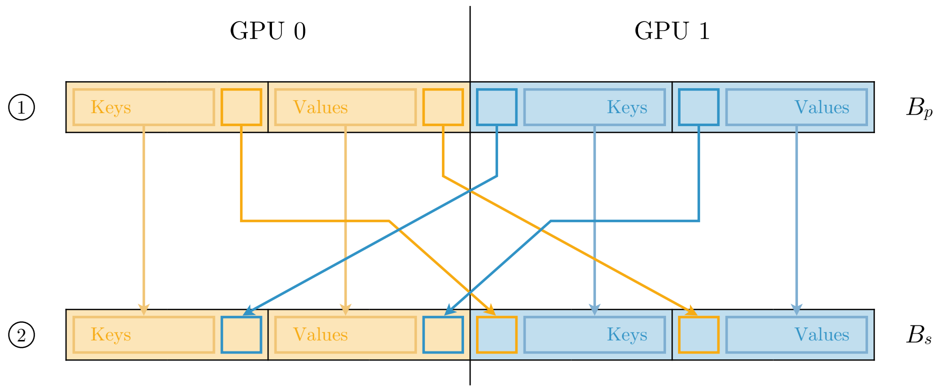


Figure 5: P2P data transfers

that operates on the keys of two sorted chunks via $O(\log(n))$ remote P2P memory reads, where n signifies the chunk size. It chooses the leftmost pivot position p and, by extension, its rightmost counterpart p' to minimize the number of key-value pairs that must be exchanged via the P2P interconnects.

After determining the optimal pivot positions, we swap the *first* p key-value pairs in C_1 with the *last* p key-value pairs in C_0. Since we exchange blocks of consecutive keys and values, respectively, their by-key order is preserved. Our implementation uses asynchronous bi-directional P2P data transfers to swap the equal-sized key and value blocks of C_0 and C_1 between the two GPUs. It copies the blocks from the primary buffers (B_p) to the secondary buffers (B_s) to avoid blocking stream synchronization, as portrayed in Figure 5. Copying the misplaced key and value blocks of C_0 and C_1 between the two GPUs occurs asynchronously on the default streams. Moving the remaining key and value blocks into their secondary buffer on each GPU occurs concurrently in high-bandwidth device memory on other streams. Once all operations have been completed, C_0 and C_1 each contain two by-key sorted key and value blocks that are merged in the final on-GPU merge step.

Merging the sorted chunks across $g \geq 4$ GPUs with $g = 2^h$ and $h > 1$ requires multiple merge stages (see Figure 6). We follow a recursive divide-and-conquer approach for merging c chunks by bringing the *left* and *right* half of the chunks into ascending order by key before and after each recursion tree level. If $c = 2$, we merge each of the $g/2$ chunk pairs at the recursion tree's leaf level across two GPUs (e.g., C_0 with C_1 and C_2 with C_3 in stage ① and stage ③). If $c > 2$, we merge each of the g/c chunk groups via a pivot selection and block exchange between multiple GPUs, followed by an on-GPU merge step (e.g., $C_0 + C_1$ with $C_2 + C_3$ in stage ②). Our implementation merges the g/c chunk groups in each merge stage simultaneously, coordinated by parallel CPU threads.

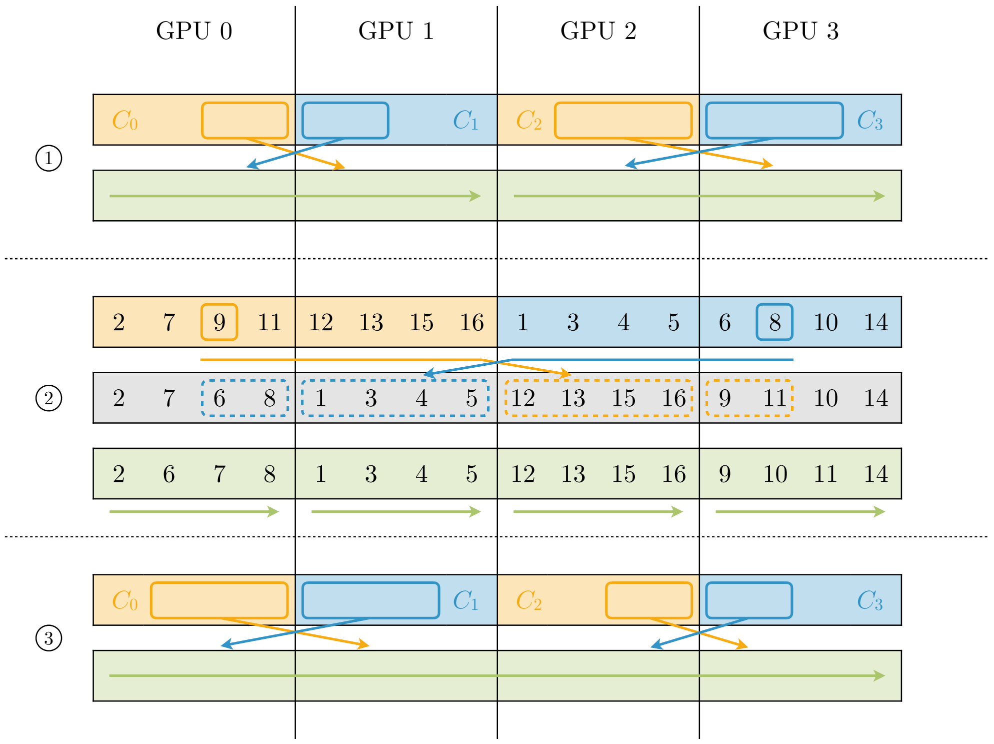


Figure 6: Block shuffling for $g = 4$ GPUs

On-GPU Chunk Merging. A fast on-GPU merge primitive is needed to efficiently merge two correlated and by-key sorted key-value pair blocks constituting a chunk. The two single-GPU merge primitives `mgpu::merge` and `thrust::merge_by_key` are based upon GPU merge path – a load-balanced tile partitioning strategy with a time complexity of $O(n/p + \log(n))$, where n and p are the total number of tuples to merge and processors, respectively [29, 78, 84].

We evaluate the performance of the two single-GPU merge primitives on modern multi-GPU platforms for two billion 64-bit tuples with 32-bit keys and 32-bit values. Our micro-benchmark shows that `thrust::merge_by_key` is up to 1.1× faster than `mgpu::mergesort` on the IBM AC922. On the NVIDIA DGX A100, it outperforms its counterpart from the `mgpu` library 1.2×. We use `thrust::merge_by_key` as the on-GPU merge primitive in our multi-GPU merge sort implementation. Since the out-of-place algorithm from the `thrust` library operates on auxiliary key and value buffers and has a memory overhead of up to 64 MB, we pass our stack allocator managing pre-allocated global memory to the merge primitive.

3.1.2 Multi-GPU Radix Sort

The radix partitioning-based multi-GPU sort approach adds support for key-value pairs and data exceeding the combined GPU memory capacity to the sort algorithm by Ilic et al. [50]. After the chunks have been transferred to the g GPUs, they are locally partitioned by key into buckets through most significant bit (MSB) radix partitioning passes. The buckets are then re-distributed between the GPUs in an all-to-all P2P bucket exchange so that, based on the most significant bits, the keys of GPU G_i are less than or equal to the keys of GPU G_j with $j > i$. Finally, the buckets are sorted locally and copied back into main memory.

On-GPU Chunk Partitioning. Once the keys and values of each chunk reside in one of the g GPUs' global memory, each GPU G_i partitions its chunk into buckets, ensuring that the keys of bucket $B_{a,i}$ precede those of bucket $B_{b,i}$ with $b > a$. First, we compute a device-local histogram with 2^m buckets over the keys' m most significant bits. Instead of reading the keys and atomically incrementing the 2^m with $m = 8$ zero-initialized buckets in global memory, our implementation divides the keys across all thread blocks, computes block-local histograms in considerably faster shared memory, and aggregates the block-local histograms with warp-aligned pre-aggregations into the device-local histogram. Second, we calculate the prefix sum of the device-local histogram to determine the write offsets for the 2^m buckets. Our implementation utilizes `cub::DeviceScan::ExclusiveScan` to calculate the histogram's prefix sum. The single-pass on-GPU prefix scan primitive employs a decoupled look-back strategy to dissociate the latency of local prefix computation from global prefix propagation and is part of the high-performance CUDA C++ library for cooperative warp-, block-, and device-wide primitives called `cub` [70, 80]. Finally, we scatter the keys and the corresponding values into the buckets based on the prefix sum. To avoid blocking synchronization, our implementation scatters the key-value pairs into the secondary buffers. To avoid random writes to global memory, it pre-scatters the keys and values simultaneously within each thread block into the block-local buckets in fast shared memory and copies the block-local key-value buckets sequentially back to global memory.

After each of the g GPUs has partitioned its chunk locally, it sends its device-local histogram to all other GPUs via the P2P interconnects. We compute the logical bucket distribution D to determine whether each device-local bucket fits into the memory of its designated GPU. Multiple device-local buckets $B_{a,0}, B_{a,1}, ..., B_{a,g-1}$ belonging to the same global bucket B_a form a *spanning* bucket if their keys and values do not fit into the global GPU memory of their assigned GPU. We refine spanning buckets through repeated MSB radix partitioning passes on the next m most significant bits until no spanning buckets are left. To avoid treating slightly

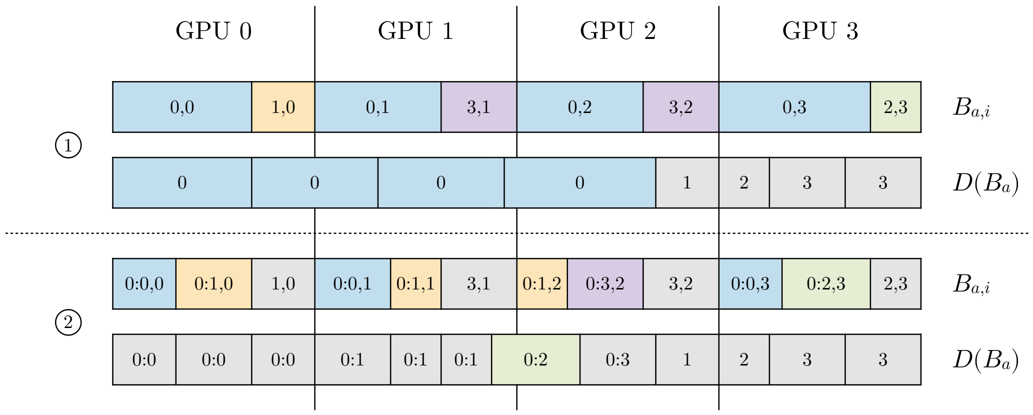


Figure 7: Chunk partitioning

overflowing buckets as spanning buckets and, thus, minimize the number of MSB radix partitioning passes, we define a padding threshold $\epsilon = 0.5\%$ relative to the chunk size, allowing select GPUs to host slightly more key-value pairs.

Figure 7 illustrates the on-GPU chunk partitioning strategy for tuples with 32-bit keys and 32-bit values on $g = 4$ GPUs. It depicts only the keys and the global buckets $B_0, B_1, ..., B_3$ for simplicity. In pass ①, each GPU partitions its chunk locally based on the keys' $m = 8$ most significant bits [32..24) and exchanges its histogram with all other GPUs. The device-local buckets $B_{0,0}, B_{0,1}, ..., B_{0,3}$ form a spanning bucket and require additional MSB radix partitioning passes. In pass ②, each GPU partitions its device-local bucket belonging to the global bucket B_0 on the next $m = 8$ bits [24..16) into smaller buckets (e.g., $B_{0:0,0}$ and $B_{0:1,0}$ on GPU G_0). The device-local histogram exchange between all GPUs reveals that the spanning bucket $B_{0,0}, B_{0,1}, ..., B_{0,3}$ has been eliminated. Since the on-GPU chunk partitioning strategy enforces only nearly perfect load balancing via the ϵ padding threshold, the global bucket $B_{0:2}$ is not a spanning bucket.

Multi-GPU P2P Bucket Swapping. Based on the logical bucket distribution, the GPUs swap misplaced key-value buckets with each other in a non-blocking all-to-all P2P bucket exchange (see Figure 8). Our implementation uses the secondary key and value buffers for the P2P bucket swapping to avoid stream synchronization. If a device-local bucket's source and destination GPUs differ, it issues two asynchronous copy operations (one for the keys and one for the values) over the P2P interconnects on the default stream. If a device-local bucket already resides on the target GPU, it copies the keys and values in high-bandwidth device memory on another stream. The CUDA runtime coalesces the asynchronous memory copy operations for the keys and values of adjacent buckets into one memory transaction.

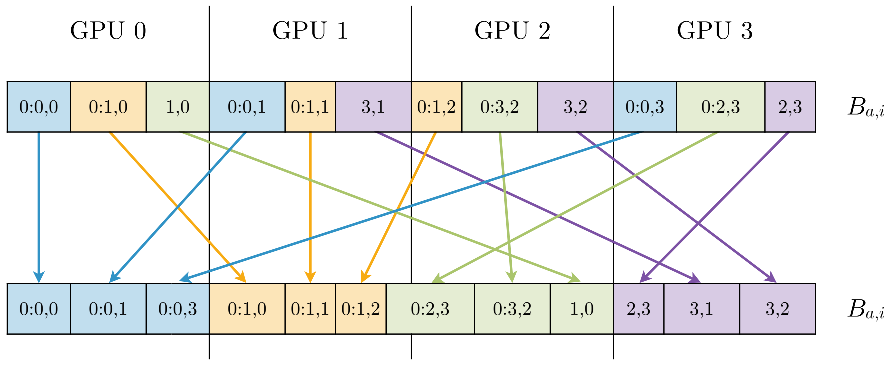


Figure 8: Bucket swapping

On-GPU Chunk Sorting. Once the g GPUs contain only buckets of distinct key ranges with respect to the $(p_{B_a} + 1) * m$ most significant bits, where p_{B_a} is the number of MSB radix partitioning passes of the global bucket B_a, each GPU sorts the buckets of its chunk locally by key and transfers them back into main memory. Our implementation utilizes the out-of-place single-GPU LSB radix sort primitive `cub::DeviceRadixSort::SortPairs` to sort the key-value pairs in each bucket based on the keys' unsorted $w - (p_{B_a} + 1) * m$ least significant bits, where w is the width of the key type [1, 80]. To reduce the total number of buckets to sort and, by extension, minimize the accumulated kernel launch overhead of the on-GPU sort primitive, our implementation fuses small neighboring buckets with the same number of MSB radix partitioning passes whose combined size is less than the experimentally determined fusing threshold $\gamma = 1.0\%$ relative to the chunk size. Sorting the fused buckets occurs on the default stream. Transferring the sorted buckets back into main memory takes place concurrently on a dedicated stream to facilitate overlapped copy and compute operations [42].

3.1.3 Out-Of-Core Data Handling

Since the input relations R and S might exceed the combined global GPU memory capacity of all g GPUs, they are sorted in chunksets. First, we split R and S into k_R and k_S chunksets, each composed of g equal-sized chunks that fit into the g GPUs' global memory. Our implementation queries the CUDA device properties and dimensions the chunksets under a memory utilization limit of 80% to leave space for auxiliary data structures [43]. Second, we sort the k_R and k_S chunksets sequentially across all g GPUs through the merge- or radix partitioning-based multi-GPU sort strategy. Our implementation allocates two chunk-sized key and value buffers in device memory. It transfers the chunks of a chunkset into the primary buffers (B_p),

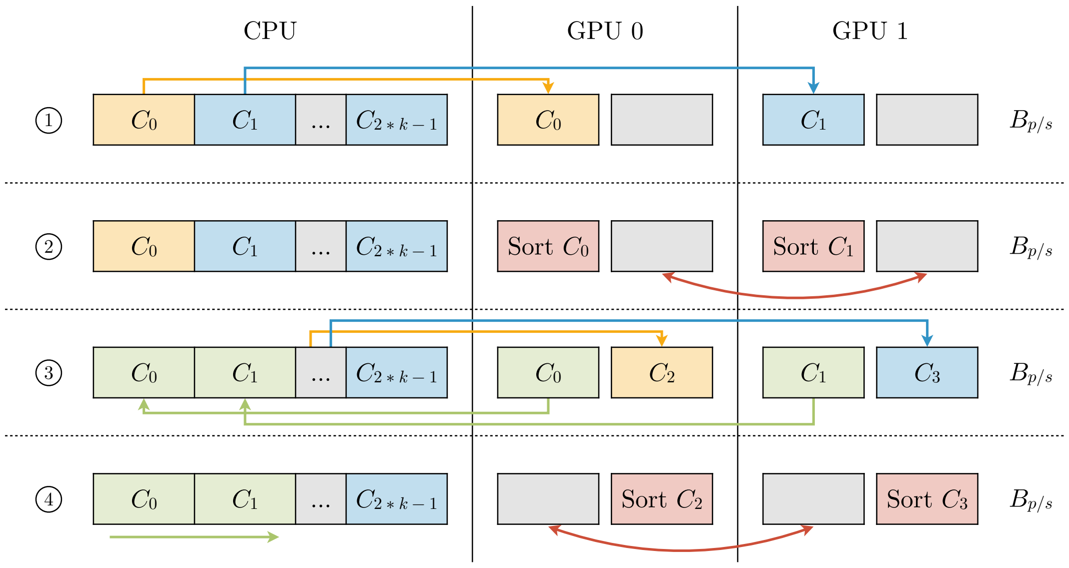


Figure 9: Sort pipeline for large out-of-core data

sorts their tuples by key across all g GPUs utilizing the secondary buffers (B_s), and transfers them back into main memory while copying the chunks of the next chunkset into the flipped primary buffers, as illustrated in Figure 9. It harnesses two non-blocking streams to overlap the data transfers to and from the GPUs and, thus, saturate the bi-directional CPU-GPU interconnect bandwidth.

Due to the high latency and implicit stream synchronization of dynamic CUDA memory allocations [25, 114, 115], the multi-GPU sort implementations operate exclusively on pre-allocated host and device memory. Our C++ stack allocator template with self-defragmentation capabilities makes one *physical* allocation and, subsequently, issues byte-aligned *virtual* allocations. It tracks its allocations (i.e., begin pointer and byte-aligned size pairs) with a doubly linked list. Upon memory allocations, it calculates the begin pointer of the virtual allocation based on the zero-initialized relative offset to the physical allocation, increases the relative offset by the byte-aligned size, and inserts the allocation as the last node into the linked list. During deallocations, it searches for the allocation in reverse, removes it from the doubly linked list, and lowers the relative offset to the accumulated allocation sizes if it was the last node. Our C++ allocator template has two specializations: the *host* allocator allocates 16-byte-aligned pinned (i.e., page-locked) main memory to facilitate high-bandwidth data transfers and the *device* allocator allocates 128-byte-aligned global GPU memory [41]. We use a single host allocator and g device allocators for all memory allocations, including those via on-GPU sort, merge, and scan primitives, in our multi-GPU sort implementations.

3.2 Merge Phase

In this subsection, we explain the CPU-assisted merge phase of our heterogeneous multi-GPU sort-merge join. It merges the tuples of the sorted k_R and k_S chunksets by key through a parallel CPU multiway merge algorithm to bring the two input relations R and S into fully sorted order in main memory. If either of the input relations comprises only one chunkset (i.e., $k_R = 1$ or $k_S = 1$), it is already fully sorted by key and requires no merge phase. The CPU multiway merge primitive operates on zero-copy zip iterators to merge the keys and values of the k_R and k_S chunksets in lockstep. Thus, it avoids copying the keys and values of R and S into temporary key-value pairs during the merge phase. Since the merge primitive is an out-of-place algorithm, it uses a pre-allocated key-value buffer of size $\max(|R|, |S|)$ for merging the chunksets of R and S in sequence.

3.2.1 CPU Multiway Merge

Merging the tuples of the k_R and k_S chunksets belonging to R and S, respectively, requires a CPU multiway merge primitive. The best conceivable time complexity of any comparison-based multiway merge algorithm is $O(n * \log(k))$, where n denotes the total number of tuples and k is the number of sorted sublists (i.e., chunksets). Both in-place algorithms and out-of-place algorithms with a space complexity of $O(n)$ have been published [16, 51, 99]. `__gnu_parallel::multiway_merge` included in the `libstdc++` parallel mode is a runtime-optimal multi-threaded CPU multiway merge primitive [21, 22]. It uses a register-optimized merge strategy with unrolled loops for $k \in \{2, 3, 4\}$ and a generic loser tree-based strategy for $k \geq 5$ [100]. CPU multiway merge algorithms are typically memory bandwidth-bound [16, 51]. Maltenberger et al. measure that `__gnu_parallel::multiway_merge` saturates the main memory bandwidth of modern high-performance computing platforms [67]. We harness the primitive in our multi-GPU join implementation.

3.2.2 Zero-Copy Zip Iterator Handling

The multi-threaded CPU merge primitive `__gnu_parallel::multiway_merge` lacks native support for tuples. Instead of copying the separately stored keys and values of R and S into and out of temporary key-value pairs with an overloaded key-based $<$ operator to employ the CPU multiway merge primitive as is, albeit with time and space overheads of $O(n)$, we adapt its implementation to operate on pointer-based zip iterators and construct key-value zip iterators for the keys and values of the

k_R and k_S chunksets. By internally storing tuples of sequence pointers, as well as dereferencing and applying permutations to the sequence pointers simultaneously, the zero-copy zip iterators allow for merging chunksets of R and S by key without any space overhead. We evaluate the performance of our zip iterator optimization for `__gnu_parallel::multiway_merge` with eight billion 8-byte tuples (i.e., 32-bit keys and 32-bit values) split into three sublists. On the IBM AC922, the speedup over the workaround using temporary key-value pairs is 5.6× (of which 64% is due to eliminating dynamic memory allocations). On the NVIDIA DGX A100, utilizing zero-copy zip iterators is 21.3× faster than relying upon temporary key-value pairs, where 86% is caused by avoiding memory allocations.

3.3 Join Phase

In this subsection, we outline our join's multi-GPU-accelerated join phase. It splits the by-key sorted input relations R and S into g correlated partitions composed of smaller correlated subpartitions via a CPU-assisted merge path partition strategy and joins the disjoint subpartition pairs independently by key across the g GPUs. Its pipelined execution model allows for transferring the keys of a disjoint subpartition pair to global memory, running the binary search-based merge join kernel on the keys of a subpartition pair to produce a set of matching key ranges, and transferring a set of matching key ranges back into main memory concurrently on each GPU. It fully saturates the bi-directional bandwidth of the CPU-GPU interconnects and maximizes the GPUs' streaming multiprocessor utilization.

3.3.1 CPU Merge Path Partition

Once the tuples of R and S reside by-key sorted in main memory, they are divided into g equal-sized correlated partitions, each containing at least three correlated subpartitions whose keys fit into the g GPUs' global memory. We determine the disjoint partition and subpartition boundaries through a key-based two-step merge path partitioning [87]. First, we split R and S into g equal-sized disjoint partition pairs $(R_0, S_0), ..., (R_{g-1}, S_{g-1})$ that can be merged independently across g GPUs. Second, we split each disjoint partition pair (e.g., (R_0, S_0)) into a minimum of three disjoint subpartition pairs (e.g., $(R_{0,0}, S_{0,0}), ..., (R_{0,3}, S_{0,3})$) that can be merged fully independently across $s = 3$ streams on a single GPU. Our implementation utilizes `mgpu::merge_path` from the CUDA C++ primitives library `mgpu` in parallel CPU threads to find the *merge path* in both steps [78].

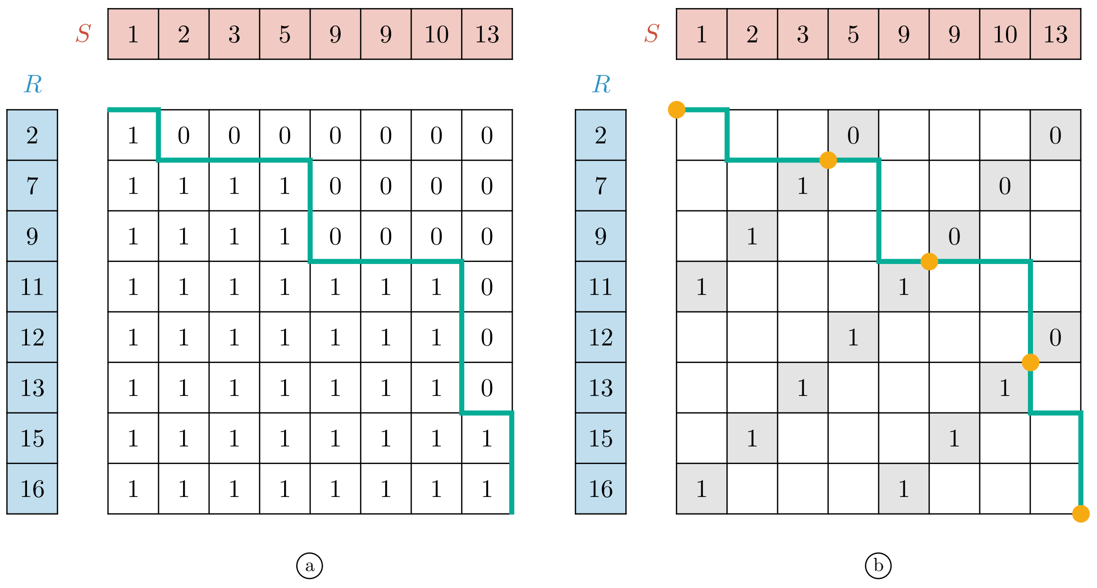


Figure 10: Merge path of *R* and *S*

Figure 10 illustrates the merge path of the keys in R and S. It is the traversal path in a merge matrix from the upper left to the lower right corner while moving only rightward (if the key in S is smaller than that of R) or downward (if the key in S is greater than or equal to that of R). The cells in the merge matrix have a value of 1 to the left bottom and 0 to the right top of the merge path (see Figure 10a). The i-th point on the merge path lies on the i-th cross diagonal in the merge matrix (see Figure 10b). Partitioning a merge path into p equal-sized contiguous segments (i.e., finding its intersection with the $p-1$ equidistant cross diagonals in the merge matrix) distributes the workload for *merging* R and S equally among p processors. In the example, the merge path partitioning of the input relations' keys produces $p = 4$ equal-sized partition pairs (e.g., $R_0 = (2)$ with $S_0 = (1, 2, 3)$).

Since equal keys in R and S might end up in different partition pairs, merge path partitioning yields no valid distribution of the workload for *joining* R and S across p processors. In the example, the key 9 occurs in one partition of R (i.e., $R_1 = (7, 9)$) but in two partitions of S (i.e., $S_1 = (5, 9)$ and $S_2 = (9, 10)$). We, therefore, conduct a boundary validation after each merge path partitioning step. If the last key in partition R_i (or S_i) is equal to the first key in partition S_{i+1} (or R_{i+1}), we compute the key ranges in both partitions via adapted binary searches, exclude the key from both partitions, and store the matching key ranges prematurely.

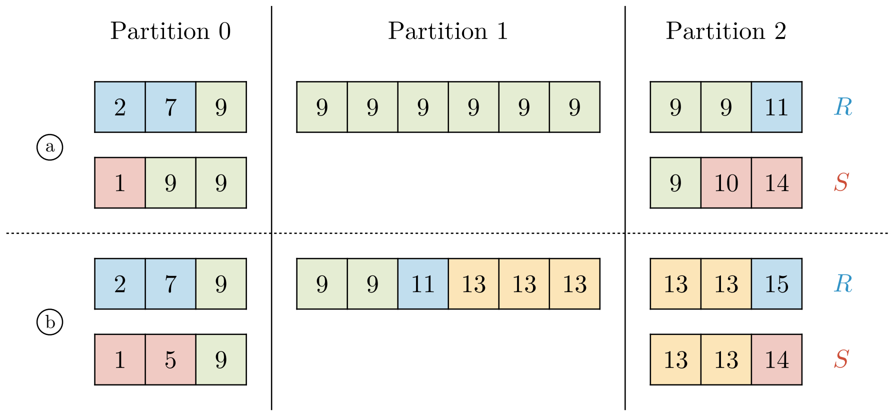


Figure 11: Skewed partition pairs

Figure 11 exemplifies the merge path partitioning of the keys in R and S for $p = 3$ with skewed partition pairs. In both examples, the partition R_1 contains keys that start in the previous (R_0) or end in the next (R_2) partition, while its counterpart S_1 is empty. We eliminate skewed partition pairs in the boundary validation after each merge path partitioning step. If R_i (or S_i) contains keys but S_i (or R_i) is empty, we check if the first key equals the last key in R_i (or S_i). If yes (see Figure 11a), we compute the key's entire range in R and S, exclude it from both input relations, and save the matching key ranges prematurely. If no (see Figure 11b), we apply the same logic with the ranges of the first and last key, respectively.

3.3.2 Multi-GPU Merge Join

After the two input relations R and S have been split into g equal-sized correlated partitions comprising at least three correlated subpartitions, each of the g GPUs joins its disjoint subpartition pairs entirely independently by key in a three-stream join pipeline. First, we distribute the keys of the three or more subpartition pairs evenly among $s = 3$ non-blocking streams in a round-robin fashion for each of the g GPUs. On the host (in main memory), we allocate g resizable buffers for the GPUs' matching key ranges. On the device (in global memory), we allocate s subpartition-sized key buffers on each GPU. Our implementation utilizes our stack allocators operating on pre-allocated memory while enforcing a self-defragmentation strategy during the entire multi-GPU merge join execution to avoid dynamic memory allocations (see Section 3.1.3). Second, we schedule g join pipelines with s concurrent streams. Each stream transfers the keys of a subpartition pair into its key buffer in global memory, executes the join kernel on the keys to produce a set of matching key ranges, and transfers the set back into its key-range buffer in main memory.

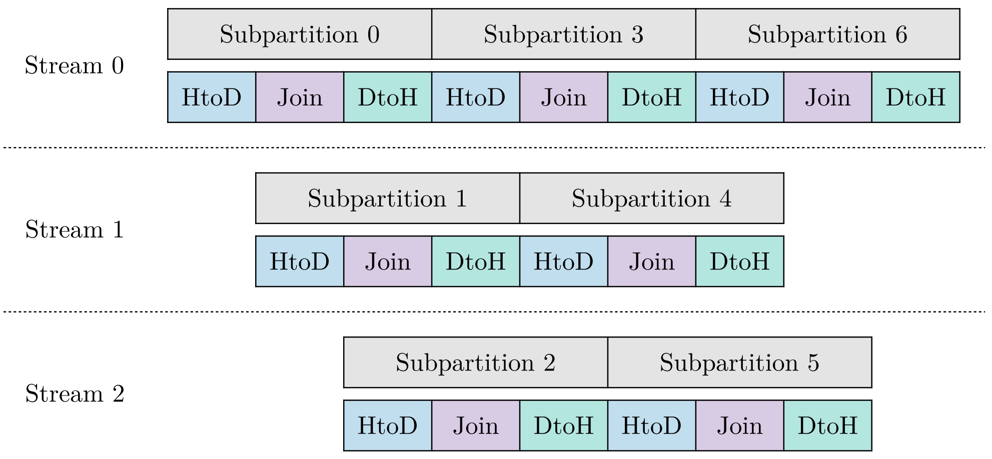


Figure 12: Join pipeline for $s = 3$ streams

Each pipeline performs all three actions simultaneously, as depicted in Figure 12. Unlike our multi-GPU-accelerated sort algorithms, for which overlapping the host-to-device and device-to-host data transfers with the compute operations yields no performance gain as the g GPUs sort *cooperatively* with explicit synchronization points (e.g., block shuffling and bucket swapping) [67], it enhances the performance for the multi-GPU merge join as the g GPUs join *independently* [11, 56, 108, 113]. Finally, we materialize the join tuples of R and S on the CPU based on the matching key ranges $([i_R, j_R], [i_S, j_S])$ with $j_R \geq i_R$ and $j_S \geq i_S$. Our implementation allocates contiguous main memory for the $\sum(j_R - i_R + 1) * (j_S - i_S + 1)$ join tuples of the output relation and materializes the tuples comprising the matched key and the corresponding values of R and S, respectively, in parallel CPU threads.

3.3.3 In-Core Join Processing

Once the keys of two correlated subpartitions of R and S have been copied into a stream's key buffer in global memory, they are joined via a binary search-based merge join kernel. Suppose, without loss of generality, $|R| \leq |S|$, for each unique key at index i_R in the subpartition of R, we conduct three binary searches to find the key's ranges in the correlated subpartitions of R and S. In j_R, we store the *last* index in the subpartition of R, whose key is equal to that of index i_R. In i_S and j_S, we store the *first* and *last* index in the subpartition of S, respectively, whose key is equal to that of index i_R. The matching key range is denoted by $([i_R, j_R], [i_S, j_S])$.

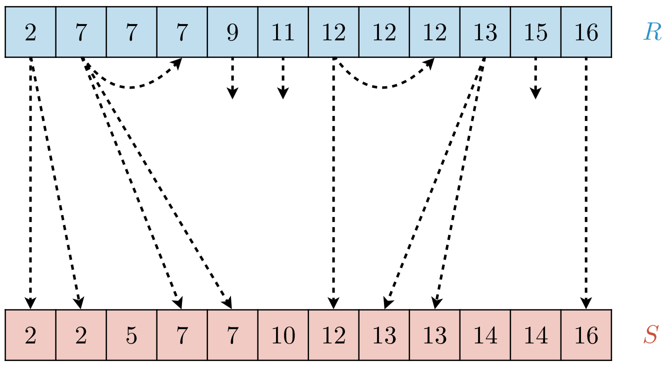


Figure 13: Range search for R and S

Figure 13 shows the range search for the keys of two correlated subpartitions of R and S. The key 2 at index $i_R = 0$ occurs once in the subpartition of R (i.e., $j_R = 0$) and twice in the subpartition of S at index $i_S = 0$ and index $j_S = 1$, resulting in the key range $([0, 0], [0, 1])$. The key 7 at index $i_R = 1$ occurs three times in the subpartition of R (i.e., $j_R = 3$) and two times in the subpartition of S ranging from the first index $i_S = 3$ to the last index $j_S = 4$, resulting in the key range $([1, 3], [3, 4])$. The key 9 at index $i_R = 4$ produces no matching key range. After finding the key's ranges in the correlated subpartitions of R and S, we atomically add $(j_R - i_R + 1) * (j_S - i_S + 1)$ to the zero-initialized join counter shared among all $s = 3$ streams in the same join pipeline and asynchronously transfer the matching key range $([i_R, j_R], [i_S, j_S])$ into the key range buffer in main memory.

Our implementation launches the merge join kernel with up to 128 blocks per grid and 256 threads per block. It specifies the optimal number of resident blocks per streaming multiprocessor in the kernel's launch bounds through recursive C++ templates and, thus, maximizes the *occupancy* (i.e., ratio of active warps to possible active warps) of each streaming multiprocessor [18, 46]. It registers a 32-byte L2 cache fetch granularity for the join kernel to read eight 32-bit or four 64-bit keys at once from global memory during the binary search-based range searches and, as a consequence, hide its latency [81]. Since the number of matching key ranges for two correlated subpartitions of R and S is unknown in advance, our implementation maps the key range buffer residing in pinned (i.e., page-locked) main memory into the device address space and transfers each matching key range back concurrently from the merge join kernel to avoid allocating a fixed-size key range buffer in device memory that might remain entirely unused [85].

4 Evaluation

In this section, we evaluate the performance of our heterogeneous multi-GPU sort-merge join implementation.[1] In Section 4.1, we elaborate on our experimental setup. In Section 4.2, we compare the end-to-end runtime of our multi-GPU sort-merge join with that of state-of-the-art CPU-based and GPU-accelerated join algorithms. After that, we analyze our multi-GPU join's execution breakdown (see Section 4.3) and scalability for increasing numbers of GPUs (see Section 4.4). Finally, we study its robustness against varying selectivity and data skew (see Section 4.5).

4.1 Experimental Setup

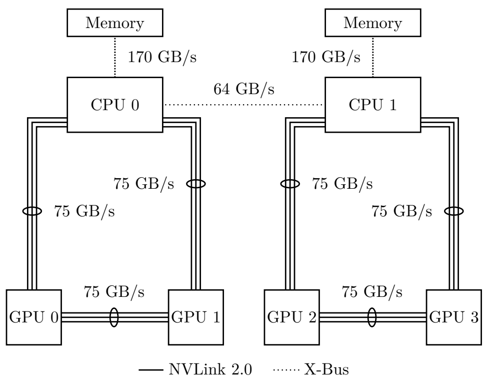
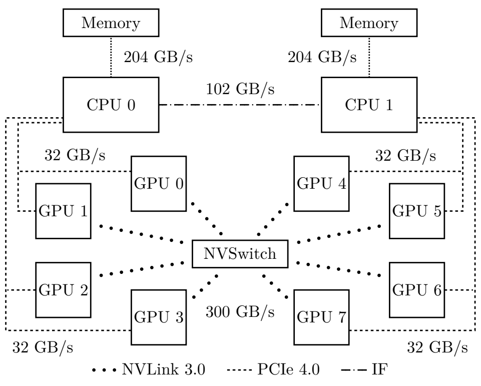
In this subsection, we provide details of the multi-GPU systems and the methodology used in our performance benchmarks. Furthermore, we describe our join workloads and our CPU- and GPU-based join algorithm baselines.

4.1.1 Hardware Platforms

We evaluate our novel heterogeneous multi-GPU sort-merge join on two dual-socket multi-GPU systems with state-of-the-art interconnects: IBM AC922 and NVIDIA DGX A100 (see Table 1). The IBM AC922 features four NVIDIA V100 GPUs (with 32 GB of global high-bandwidth memory) equally distributed across both NUMA nodes [49]. Its CPU-GPU and P2P interconnects are based on three high-speed NVLink 2.0 links with a uni-directional bandwidth of 75 GB/s. Its X-Bus-powered CPU-CPU interconnect has a theoretical bandwidth of 64 GB/s per direction. The NVIDIA DGX A100 has eight NVIDIA A100 GPUs (with 40 GB of GPU memory) and non-blocking all-to-all NVLink 3.0-based NVSwitch P2P interconnects offering uni-directional inter-GPU data transfer rates up to 300 GB/s [82]. The platform harnesses PCIe 4.0 for the CPU-GPU interconnects and Infinity Fabric with a peak bandwidth of 102 GB/s per direction as CPU-CPU interconnect between the two NUMA nodes. Adjacent pairs of NVIDIA A100 GPUs share a PCIe 4.0 link and, thus, its uni-directional bandwidth of 32 GB/s, through a switch.

[1]`http://github.com/hpides/multi-gpu-sort-merge-join`

Table 1: Multi-GPU accelerator platforms

(a) IBM AC922	**(b) NVIDIA DGX A100**
 Memory Memory 170 GB/s 170 GB/s 64 GB/s CPU 0 CPU 1 75 GB/s 75 GB/s 75 GB/s 75 GB/s 75 GB/s 75 GB/s GPU 0 GPU 1 GPU 2 GPU 3 NVLink 2.0 X-Bus	 Memory Memory 204 GB/s 204 GB/s 102 GB/s CPU 0 CPU 1 32 GB/s 32 GB/s GPU 0 GPU 4 GPU 1 GPU 5 NVSwitch GPU 2 GPU 6 GPU 3 300 GB/s GPU 7 32 GB/s 32 GB/s NVLink 3.0 PCIe 4.0 IF
2x IBM POWER9 (16x 2.7 GHz)	2x AMD EPYC 7742 (64x 2.3 GHz)
4x NVIDIA V100 SXM2 32 GB	8x NVIDIA A100 SXM4 40 GB
2x 256 GB DDR4	2x 512 GB DDR4
RHEL 8.2, ppc64-le	Ubuntu 20.04, x86-64
CUDA 11.8, GCC 10.2.1	CUDA 11.4, GCC 9.3.0

4.1.2 Benchmark Methodology

We measure the end-to-end duration of joining the input relations R and S without materializing the tuples in all performance benchmarks to facilitate comparability with related work [2, 6, 12, 59, 97, 98, 107]. We repeat every benchmark three times and report the arithmetic mean of the measured durations across all repetitions, resulting in a standard error of less than 3%. The input relations R and S reside in main memory on the first NUMA node. The GPU-accelerated join baselines and our heterogeneous multi-GPU sort-merge join operate on pre-allocated pinned host memory and global device memory. On each multi-GPU platform, we assume that the GPUs are used exclusively as database accelerators and choose the optimal (i.e., fastest) GPU set $\hat{G}^g$ for performance benchmarks involving g GPUs based on the platform's interconnect topology (e.g., $\hat{G}^2 = \{0, 1\}$ on the IBM AC922 as well as $\hat{G}^2 = \{0, 2\}$ and $\hat{G}^4 = \{0, 2, 4, 6\}$ on the NVIDIA DGX A100).

Table 2: Workloads for scale factors *f*

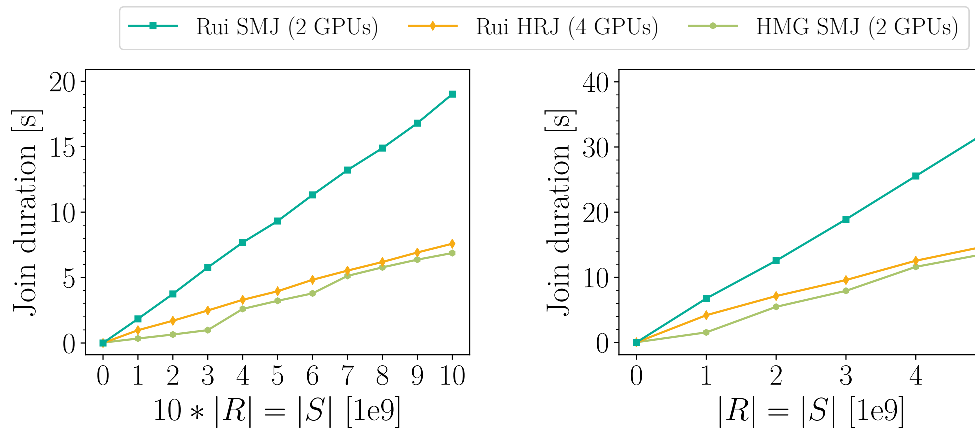
	A	**B**
#key/#value	4/4 bytes	8/8 bytes
$\|R\|$	$f * 1/10 * 10^9$ tuples	$f * 10^9$ tuples
$\|S\|$	$f * 10^9$ tuples	$f * 10^9$ tuples
$\#R$	$f * 800$ MB	$f * 16$ GB
$\#S$	$f * 8$ GB	$f * 16$ GB

4.1.3 Join Workloads

We generate synthetic input relations R and S with narrow tuples (i.e., key-value pairs) in a column-oriented fashion to align with related work [2, 7, 89, 97, 104, 116]. Unless specified otherwise, the keys in R and S are uniformly distributed integers over the entire 32- or 64-bit range that follow a foreign key relationship (i.e., every key in S has *exactly one* matching key in R). We study two workloads for different scale factors f (see Table 2). In workload **A**, R and S contain 8-byte tuples with 32-bit keys and 32-bit values, where $10 * |R| = |S|$. In workload **B**, R and S with $|R| = |S|$ comprise 16-byte tuples with 64-bit keys and 64-bit values.

4.1.4 Join Baselines

We compare the performance of our novel multi-GPU sort-merge join against that of state-of-the-art CPU and GPU joins. Our CPU baselines are the highly parallel NUMA-aware multiway sort-merge join and radix-hash join by Balkesen et al. [7]. Both algorithms utilize 256-bit SIMD instructions while employing multi-threaded and cache-conscious workload partitioning and processing strategies. Since the two CPU joins rely upon the Advanced Vector Extensions (AVX) to the x86 instruction set architecture [4, 53], we evaluate their performance solely on the x86-64-based NVIDIA DGX A100. Both algorithms have been used extensively as baselines for hardware-accelerated joins [15, 38, 58, 65, 97, 98, 111, 112]. Our GPU baselines are the multi-GPU-accelerated sort-merge join and hybrid-radix join by Rui et al. [97]. Both GPU joins support large out-of-core data, but leave the high-bandwidth P2P interconnects of modern multi-GPU systems entirely unused.



(a) Workload A **(b) Workload B**

Figure 14: Join baseline comparison on the IBM AC922

4.2 Baseline Comparison

In this section, we compare the runtime of our novel multi-GPU join with that of our CPU and GPU baselines for the two workloads **A** and **B**.

On the **IBM AC922**, the optimal GPU set for our multi-GPU join with the radix partitioning-based sort strategy is $\hat{G}^2 = \{0, 1\}$ (see Section 4.3 and Section 4.4). The multi-GPU-based sort-merge and hybrid-radix join by Rui et al. achieve the shortest join durations with $g = 2$ and $g = 4$ GPUs, respectively [97]. Figure 14a shows the baseline comparison for workload **A** with $f \in [0, 10]$. Our heterogeneous multi-GPU sort-merge join (HMG SMJ) scales linearly with $|S|$ up to 3B tuples, outperforming the GPU baselines 5.9× (sort-merge) and 2.5× (hybrid-radix). In that cardinality range, our heterogeneous multi-GPU join requires no CPU-based merge phase as S fits into the combined GPU memory of $g = 2$ GPUs (64 GB) with a chunk size of 1.5B tuples. In the following cardinality range, S exceeds the GPU memory capacity of $g = 2$ GPUs and requires a CPU-based merge phase involving $k_S = 2$ (3B to 6B), $k_S = 3$ (6B to 9B), and $k_S = 4$ (9B to 10B) chunksets. On the IBM AC922, the performance of the CPU merge primitive `__gnu_parallel::multiway_merge` *may* deteriorate for increasing numbers of sublists (i.e., chunksets) $k \in [2, 5]$, depending on the total number of tuples. Once S contains more than 9B tuples, the speedups over the baselines reduce to 2.8× (sort-merge) and 1.1× (hybrid-radix). Figure 14b depicts the join comparison for workload **B** with $f \in [0, 5]$. Our multi-GPU join exhibits a similar performance pattern when R and S comprise 16-byte tuples. It outperforms the join baselines 4.2× (sort-merge) and 2.5× (hybrid-radix) for up to

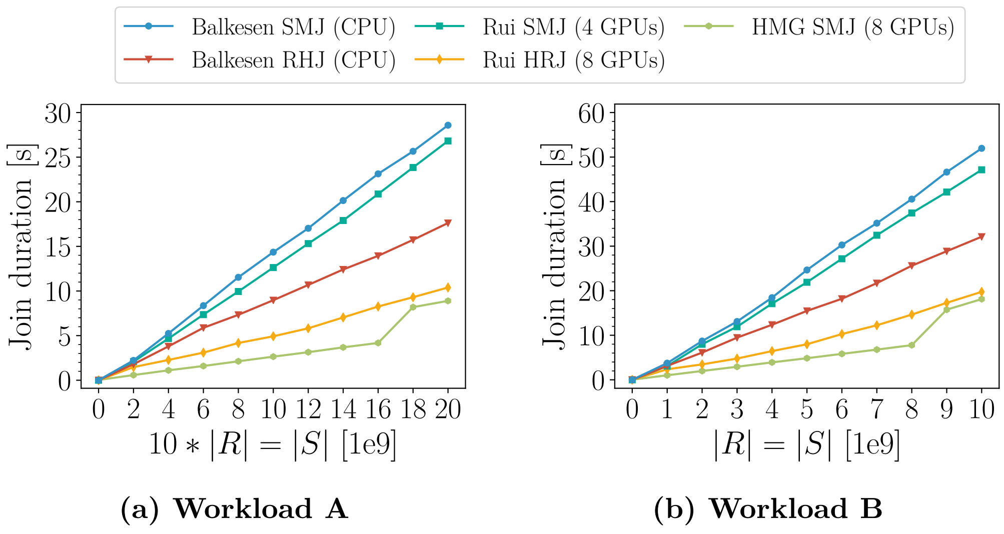
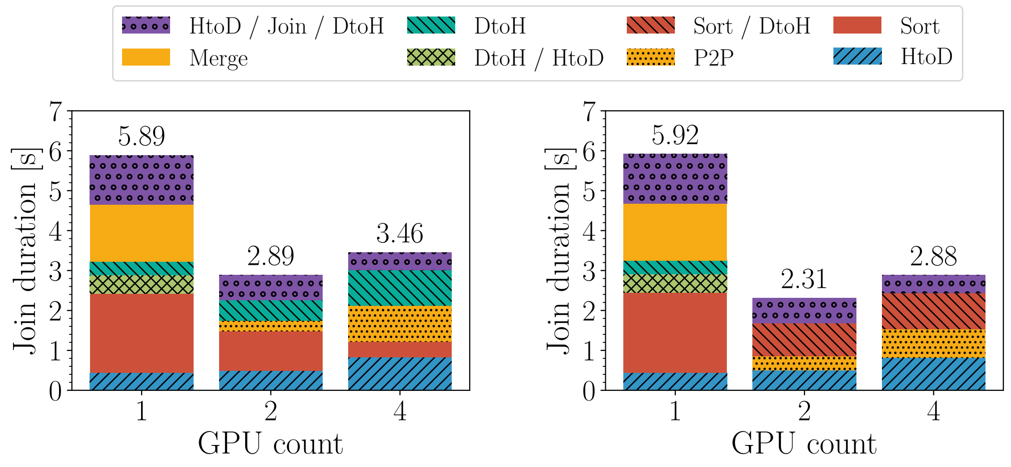


Figure 15: Join baseline comparison on the NVIDIA DGX A100

1.5B tuples in S. Once $|S|$ is greater than 1.5B 16-byte tuples, the speedups over the multi-GPU-based sort-merge and hybrid-radix join are 2.4× and 1.1× to 1.2×, respectively. Unlike workload **A**, for which only S requires a parallel CPU merge phase, R and S need a CPU merge phase for workload **B**.

On the **NVIDIA DGX A100**, our multi-GPU join achieves the fastest runtime with the radix partitioning-based sort strategy and all $g = 8$ GPUs (see Section 4.3 and Section 4.4). The multi-threaded CPU joins by Balkesen et al. efficiently utilize the platform's 128 cores distributed between two NUMA nodes [7]. The fastest GPU sets for the sort-merge and hybrid-radix join by Rui et al. are $\hat{G}^4 = \{0, 2, 4, 6\}$ and $\hat{G}^8$ with all $g = 8$ GPUs [97]. For workload **A** with 8-byte tuples and $10 * |R| = |S|$, our heterogeneous multi-GPU sort-merge join (HMG SMJ) scales linearly with $|S|$ up to 16B tuples, as illustrated in Figure 15a. It is 5.5× faster than the CPU sort-merge join and 3.3× faster than the CPU radix-hash join. It outperforms the GPU baselines 5.0× (sort-merge) and 2.0× (hybrid-radix) when the input relation S fits into the GPU memory of all $g = 8$ GPUs (320 GB). When $|S|$ is greater than 16B tuples, the speedups over the fastest CPU and GPU joins reduce to 2.0× and 1.2× as a CPU merge phase with $k_S = 2$ chunksets is required. For workload **B** with 16-byte tuples and $|R| = |S|$, our join outperforms the CPU sort-merge join 5.2× and radix-hash join 3.3× for up to 8B tuples in S, as illustrated in Figure 15b. It is 4.8× (sort-merge) and 1.9× (hybrid-radix) faster than the GPU joins. Although the sort-merge join by Balkesen et al. uses AVX instructions only for 8-byte tuples [7], it performs proportionally similar for 16-byte tuples.



(a) **Merge sort phase** (b) **Radix sort phase**

Figure 16: Join execution breakdown on the IBM AC922

4.3 Execution Breakdown

In this section, we analyze the impact of our heterogeneous multi-GPU join's three phases (i.e., *sort*, *merge*, and *join*) on its end-to-end runtime for the P2P merge- and radix partitioning-based multi-GPU sort strategies.

On the **IBM AC922**, we study the execution of our sort-merge join for workload **B** with $f = 1.5$ on $g \in \{1, 2, 4\}$ GPUs with $\hat{G}^1 = \{0\}$ and $\hat{G}^2 = \{0, 1\}$. We conduct our analysis for workload **B** with $|R| = |S|$ equal to 1.5B tuples to fill the combined GPU memory of the system's *overall* best GPU set $\hat{G}^2$ (see Section 4.4).

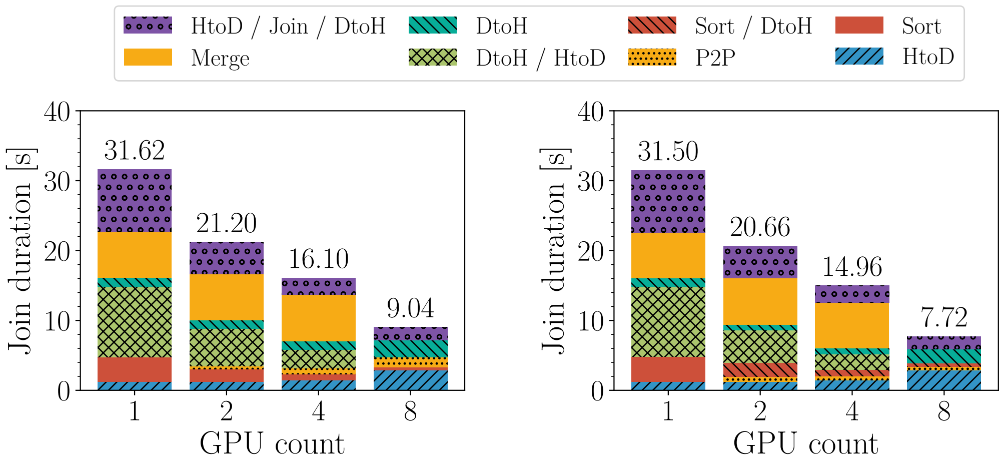
Figure 16a illustrates the end-to-end join duration breakdown with the *merge-based* multi-GPU sort strategy. Relative to the total execution time of 5.89 s for $g = 1$ GPU, the sort, merge, and join operations amount to 34%, 24%, and 21%, respectively. Since our heterogeneous sort-merge join operates on a chunk size of 750M tuples for 16-byte key-values pairs, R and S exceed the global GPU memory capacity (32 GB) and require a CPU merge phase involving $k_R = k_S = 2$ chunksets, each composed of a single chunk. Our join interleaves the host-to-device (HtoD) transfer for the *second* chunkset of R and S with the device-to-host (DtoH) transfer for the *first* chunkset. It executes the HtoD copy operation for the *first* chunkset (7%) and the DtoH copy operation for the *second* chunkset (6%) sequentially. On $g = 2$ GPUs, our multi-GPU join achieves a runtime of 2.89 s, outperforming the single-GPU setup 2.0×. Since R and S each fit fully into the combined global GPU memory of $g = 2$ GPUs (64 GB), no parallel CPU-based merge phase is required.

The sort (0.99 s) and join (0.63 s) times halve in absolute numbers for $g = 1 \rightarrow 2$ GPUs. The P2P block shuffling makes up for only 9% of the total execution time due to the fast NVLink 2.0 P2P interconnects with a uni-directional bandwidth of 75 GB/s. Since the NVLink 2.0-based CPU-GPU interconnects are not shared between the GPUs, our multi-GPU join copies the chunkset of R and S, respectively, into global memory (HtoD) and main memory (DtoH) in half the time for $g = 1 \rightarrow 2$ GPUs. On $g = 4$ GPUs, our join performs 20% worse than on $g = 2$ GPUs (3.46 s vs. 2.89 s). Although the sort (0.40 s) and join (0.44 s) durations roughly halve for $g = 2 \rightarrow 4$ GPUs, the P2P block shuffling between $g = 4$ GPUs is 3.5× slower than between $g = 2$ GPUs due to the limited and rarely attainable X-Bus CPU-CPU interconnect bandwidth of 64 GB/s per direction [67]. The X-Bus also slows down the HtoD (24%) and DtoH (26%) data transfers on $g = 4$ GPUs.

Figure 16b shows the execution breakdown with the *radix partitioning-based* sort strategy. On $g = 1$ GPU, the performance of our sort-merge join is independent of the sort strategy as neither P2P block shuffling (merge) nor P2P bucket swapping (radix) occurs. However, employing the multi-GPU radix sort for $g > 1$ yields 20% ($g = 2$) and 17% ($g = 4$) faster join durations compared to using the multi-GPU merge sort. When $g = 2$ GPUs are utilized, our join spends 15% of its runtime on P2P bucket swapping and 36% on interleaved sorting and copying buckets back into main memory (DtoH). Since GPUs attached to different NUMA nodes lack high-speed NVLink 2.0 P2P interconnects, the P2P bucket swapping is 2.1× slower on four GPUs compared to two GPUs. Simultaneously sorting and transferring buckets back into main memory (DtoH) takes roughly the same time for $g = 2 \rightarrow 4$ GPUs because the compute operations are twice as fast, but the copy operations are twice as slow with $g = 4$ GPUs. Our join's runtime is always on par or better with radix sort than merge sort on the IBM AC922.

On the **NVIDIA DGX A100**, we dissect our novel multi-GPU join's execution for workload **B** with $f = 8$ across $g \in \{1, 2, 4, 8\}$ GPUs. The fastest GPU sets for $g < 8$ are $\hat{G}^1 = \{0\}$, $\hat{G}^2 = \{0, 2\}$, and $\hat{G}^4 = \{0, 2, 4, 6\}$. By choosing a cardinality of 8B tuples for R and S, we maximize the GPU memory utilization for the platform's *overall* optimal GPU set $\hat{G}^8$ (see Section 4.4).

With the *merge-based* multi-GPU sort strategy (see Figure 17a), the performance of our join improves for increasing numbers of GPUs $g \in \{1, 2, 4, 8\}$ from 31.62 s ($g = 1$) to 9.04 s ($g = 8$) up to 3.5×. Until four GPUs, R and S exceed the GPU memory capacity of $g = 1$ (40 GB), $g = 2$ (80 GB), and $g = 4$ (160 GB) GPUs and require a parallel CPU merge phase. Since our multi-GPU join works with a chunk size of 1B tuples for 16-byte key-value pairs, the CPU-based merge phase for each input relation comprises eight ($g = 1$), four ($g = 2$), and two ($g = 4$) chunksets.

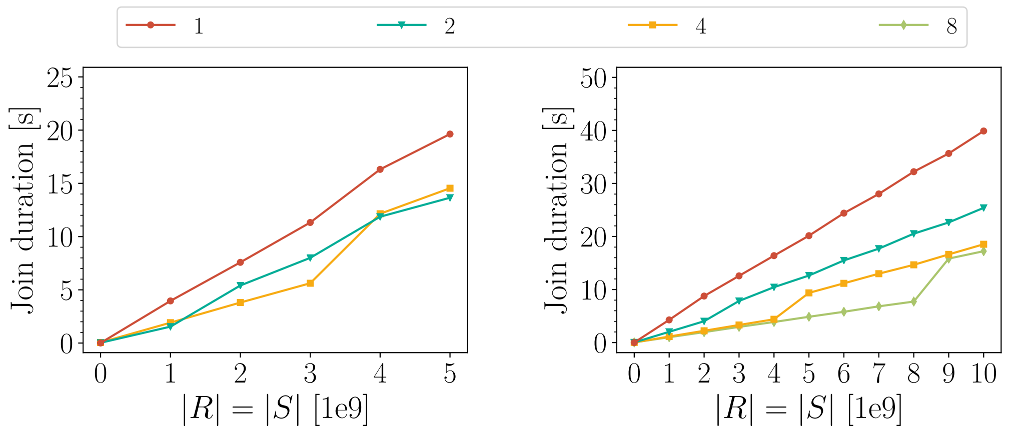


(a) Merge sort phase **(b) Radix sort phase**

Figure 17: Join execution breakdown on the NVIDIA DGX A100

On the NVIDIA DGX A100, the CPU primitive `__gnu_parallel::multiway_merge` runs equally fast for different numbers of sublists (i.e., chunksets) $k \in [2, 5]$. Until four GPUs, the absolute execution times of the sort, join, and overlapped HtoD and DtoH copy operations halve for $g \to 2 * g$ as the PCIe 4.0-based CPU-GPU interconnects' bandwidth is not shared between any GPUs in the optimal GPU sets $\hat{G}^1$, $\hat{G}^2$, and $\hat{G}^4$. On $g = 8$ GPUs, adjacent pairs of GPUs (e.g., G_0 and G_1 or G_4 and G_5) share a single PCIe 4.0 link and, consequently, its uni-directional maximum bandwidth of 32 GB/s, through a switch. The accumulated runtime of the (partially interleaved) HtoD and DtoH copy operations is, therefore, almost identical for $g = 4$ and $g = 8$ GPUs. It amounts to 59% of the total join duration with $g = 8$ GPUs. The impact of the join operation on the execution time is only 20%, while the remaining 21% are split between on-GPU chunk sorting and P2P block shuffling. No CPU merge phase is required on $g = 8$ GPUs.

With the *radix partitioning-based* sort strategy (see Figure 17b), utilizing $g = 8$ GPUs (7.72 s) yields 4.1× shorter join durations than using $g = 1$ GPU (31.50 s). Our sort-merge join's performance for $g \in \{2, 4, 8\}$ GPUs is always better with multi-GPU radix sort than multi-GPU merge sort (3% to 15%) due to the efficient all-to-all P2P bucket swapping instead of the multi-stage P2P block shuffling and overlapped sorting and copying buckets back into main memory (DtoH). On $g = 8$ GPUs, the impact of the sort, join, and P2P data transfer operation on the execution time is 6%, 24%, and 7%, respectively.



(a) IBM AC922 **(b) NVIDIA DGX A100**

Figure 18: Join scalability for different GPU counts

4.4 Scalability Analysis

In this section, we evaluate the performance of our multi-GPU sort-merge join for increasing numbers of GPUs with workload **B**, where $|R| = |S|$.

On the **IBM AC922** with $f \in [0, 5]$ (see Figure 18a), our heterogeneous join scales linearly to 3B tuples in S for $g = 1$ GPU (G_0) despite employing CPU-based merge phases for R and S with $k_R = k_S \in \{2, 3, 4\}$ chunksets across the cardinality range. Once $|S|$ exceeds 3B 16-byte tuples, its relative performance deteriorates slightly when `__gnu_parallel::multiway_merge` employs its loser tree-based strategy for merging $k_R = k_S \geq 5$ chunksets, each comprising a single chunk with 750M tuples. Our multi-GPU join's runtime reduces for $g = 1 \rightarrow 2$ GPUs (G_0 and G_1) 2.6× if R and S fit into the combined GPU memory of $g = 2$ GPUs (up to 1.5B tuples) and roughly 1.4× otherwise. Utilizing $g = 4$ instead of $g = 2$ GPUs yields shorter join durations (30%) only in the range of S from 1.5B to 3B tuples, where a CPU merge phase with $k_R = k_S = 2$ chunksets is necessary for two but not four GPUs. The *overall* fastest GPU set on the IBM AC922 is, thus, $\hat{G}^2 = \{0, 1\}$.

On the **NVIDIA DGX A100** with $f \in [0, 10]$ (see Figure 18b), our join exhibits linear scaling behavior over the entire cardinality range of S for $g = 1$ GPU (G_0). With $g = 2$ GPUs (G_0 and G_1), its performance enhances 2.2× up to 2B tuples in S and 1.6× in the out-of-core range. With $g = 4$ GPUs (G_0, G_2, G_4, and G_6), its runtime reduces 3.7× until $|S|$ equals 4B tuples and 2.2× beyond. Our multi-GPU join is fastest on $g = 8$ GPUs with speedups of up to 4.2× ($g = 1$), 2.7× ($g = 2$), and 1.9× ($g = 4$). $\hat{G}^8$ is the *overall* optimal GPU set.

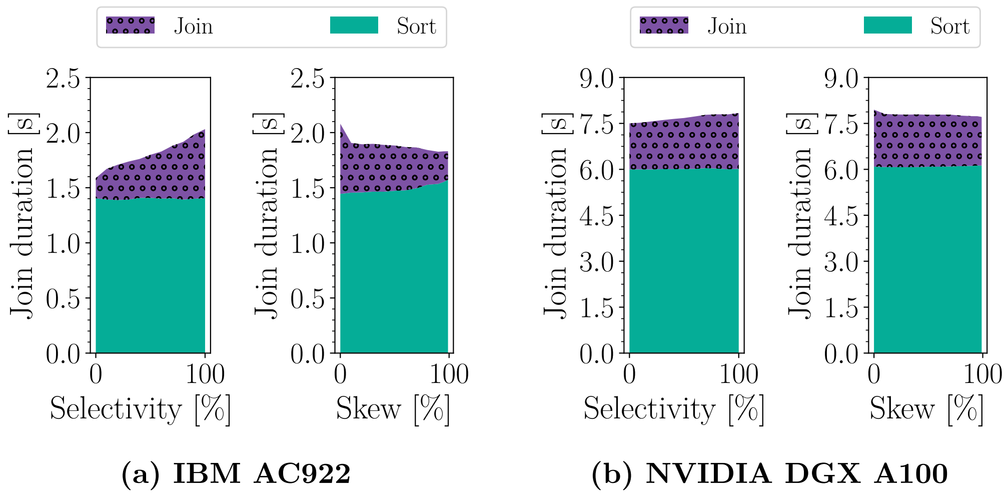


Figure 19: Join robustness against varying selectivity and skew

4.5 Robustness Analysis

In this section, we study the impact of selectivity and data skew on our multi-GPU join using workload **B** with $|R| = |S|$ equal to 1.5B tuples on the IBM AC922 ($\hat{G}^2$) in Figure 19a and 8B on the NVIDIA DGX A100 ($\hat{G}^8$) in Figure 19b.

Selectivity Analysis. We soften the foreign key relationship constraint between R and S when scaling the selectivity factor $\sigma \in [0, 1]$ so that every key in S has *at most one* matching key in R [36, 37]. For $\sigma = 0 \rightarrow 1$, our join's sort phase remains stable while its join phase slows down 3.4× (IBM AC922) and 1.2× (NVIDIA DGX A100) as more and more keys in R entail *three* instead of *two* binary searches (to find their *last* index in R, *first* index in S, and *last* index in S) and atomically incrementing the shared join counter as well as asynchronously copying a matching key range into main memory (see Section 3.3.3). Our join's end-to-end runtime increases by 28% and 4% on the IBM AC922 and NVIDIA DGX A100, respectively. In previous experiments, we assume the worst case $\sigma = 1$.

Skew Analysis. We sample the non-unique keys of S from R according to a Zipf distribution when increasing the skew factor $\theta \in [0, 1[$ as per Gray et al. [28]. For $\theta = 0 \rightarrow 1$, our multi-GPU join's execution time decreases by 12% and 3% on the IBM AC922 and NVIDIA DGX A100, respectively. Its *sort* phase becomes up to 9% and 1% slower as the number of MSB radix partitioning passes increases [50]. Its *join* phase, however, becomes up to 2.5× and 1.2× faster as increasingly larger key ranges in S are eliminated for some of the unique keys in R (see Section 3.3.3). In earlier performance benchmarks, we assume the worst case $\theta = 0$.

5 Discussion

Our heterogeneous multi-GPU sort-merge join outperforms state-of-the-art CPU and GPU joins on modern multi-GPU platforms with high-speed interconnects for large input relations. On the IBM AC922, it achieves speedups of 5.9× and 2.5× over Rui et al.'s multi-GPU sort-merge join and hybrid-radix join, respectively [97]. On the NVIDIA DGX A100, it is up to 5.0× (sort-merge) and 2.0× (hybrid-radix) faster than the GPU baselines and yields 5.5× and 3.3× shorter runtimes than the CPU sort-merge and radix-hash joins by Balkesen et al. [7]. Our join is, therefore, eminently suitable as an operator for GPU-accelerated database systems.

Out of its three algorithm phases (i.e., *sort*, *merge*, and *join*), the multi-GPU sort phase impacts our join's total execution time the most, with as much as 72% on the IBM AC922 and 76% on the NVIDIA DGX A100. The radix partitioning-based sort strategy is 15% to 20% faster than the merge-based strategy with the optimal GPU sets, primarily due to all-to-all P2P bucket swapping instead of multi-stage P2P block shuffling. Thus, the findings by Ilic et al. hold for sorting tuples [50]. The CPU merge phase, which is required if an input relation exceeds the combined GPU memory capacity of all GPUs, causes a performance cliff as the CPU merge primitive saturates the main memory bandwidth of 170 GB/s on the IBM AC922 as well as 204 GB/s on the NVIDIA DGX A100 (see Table 1). Lutz et al. propose a GPU-partitioned join strategy that eliminates the performance cliff stemming from large out-of-core data [65]. However, the authors' strategy is only applicable to the IBM AC922 with fast NVLink 2.0-based CPU-GPU interconnects and not to other bleeding-edge HPC systems such as the NVIDIA DGX A100 and H100 [49, 82, 83]. The hybrid join phase with overlapped copy and compute operations impacts our multi-GPU join's runtime the least, with as little as 28% on the IBM AC922 and 24% on the NVIDIA DGX A100. If both input relations are pre-sorted (e.g., due to prior group-by, order-by, or tree-based index scan operators), our sort-merge join has to execute only the join phase, while radix-based joins fail to exploit the interesting (tuple) orders [106]. In that case, it is 14.4× (IBM AC922) and 9.2× (NVIDIA DGX A100) faster than the hybrid-radix join.

Although increasing the number of GPUs yields *consistently* faster join phases, it *occasionally* results in slower sort phases. On the IBM AC922, the optimal GPU set is $\hat{G}^2$ albeit the system has four GPUs. It outperforms the single-GPU setup 2.6×. On the NVIDIA DGX A100, the optimal GPU set $\hat{G}^8$ involves all eight GPUs and is 4.2× faster than the single-GPU setup. Hence, interconnect topology awareness is crucial. Our join benefits from data skew with up to 12% shorter join durations. It is a robust GPU-accelerated database operator.

6 Related Work

In this section, we classify related research efforts on CPU-based, GPU-accelerated, and distributed join algorithms for relational database systems.

6.1 CPU Joins

Over the past decades, researchers have thoroughly studied parallel CPU-based joins for in-memory query processing. Kim et al. and Polychroniou et al. propose SIMD-optimized sort-merge and hash joins to exploit the data-level parallelism capabilities of modern CPUs [59, 92]. After studying various hash joins, Blanas et al. conclude that simple hardware-oblivious hash joins with a shared and non-partitioned hash table outperform complex hardware-conscious hash joins [12]. Balkesen et al. draw the opposite conclusion after evaluating their parallel radix-hash join that incorporates the bucket chaining method by Manegold et al. [7, 68]. In addition to that, Balkesen et al. revisit the classic performance debate on sort-merge vs. hash joins and claim that for most workloads, the hash join is faster than the sort-merge join, although the relative performance gap narrows considerably for large input relations [6]. In contrast to these research efforts on parallel CPU-based joins, we focus on multi-GPU-accelerated joins.

6.2 Single-GPU Joins

Most published GPU-based joins are in-core single-GPU algorithms that expect both input relations to reside in GPU memory. Rui and Tu propose two GPU-accelerated joins: a radix-partitioned GPU hash join utilizing shared histograms and a merge path-partitioned GPU sort-merge join [98]. More recently, Sioulas et al. outline an efficient GPU hash join harnessing a bucket chaining method to avoid the need to build histograms [107]. He et al. study joins in the context of CPU-GPU co-processing [48]. Several experimental studies compare the performance of CPU- and GPU-based joins and show the superiority of GPU-based joins [57, 96]. However, prior publications on GPU-accelerated joins rarely address the case of large out-of-core data where the size of the input relations exceeds the GPU memory capacity. Only Guo and Chen and Lutz et al. describe mechanisms for joining large input relations [32, 65]. Guo and Chen utilize a CPU-assisted radix partitioning strategy and an in-core GPU sort-merge join for large out-of-core data. Lutz et al. harness fast interconnects that provide GPUs with high-bandwidth, cache-coherent access to main memory to handle large out-of-core data. Unlike these single-GPU algorithms, our multi-GPU sort-merge join utilizes the compute power and memory bandwidth of all GPUs that modern multi-GPU platforms offer.

6.3 Multi-GPU Joins

Regarding multi-GPU-based join processing, Paul et al. propose a radix-partitioned hash join for parallel multi-GPU architectures that follows an adaptive multi-hop routing protocol for the P2P data transfers to minimize data transfer congestion [89]. In contrast to our multi-GPU sort-merge join, the multi-GPU join by Paul et al. cannot natively handle large out-of-core data. Besides, the recent trend towards symmetric switch-based P2P interconnects (e.g., NVLink 3.0-based NVSwitch) between the GPUs of modern multi-GPU platforms makes their adaptive routing protocol obsolete [75]. Rui et al. design two out-of-core multi-GPU joins: a sort-merge join and a hybrid-radix join [97]. Neither of the two algorithms utilizes P2P interconnects for inter-GPU communication, though. Our novel multi-GPU join harnesses the high-bandwidth P2P interconnects between the GPUs.

6.4 Distributed Joins

Unlike CPU- and GPU-based single-node joins, distributed joins operate on multiple nodes across high-speed networks that often feature remote direct memory access (RDMA) [93, 94, 95]. Barthels et al. propose a distributed and massively parallel CPU-based radix-hash and sort-merge join utilizing fast one-sided RDMA memory operations [10]. Based on the initial features of GPUDirect RDMA [86], Guo et al. explore the performance of distributed joins in multi-node multi-GPU clusters with subsequent data shuffling and GPU execution phases [33]. Thostrup et al. propose a pipelined GPU hash join that overlaps its data shuffling with its multi-GPU-based build and probe phases over fast GPUDirect RDMA-capable networks [111]. Gao and Sakharnykh present a distributed hash join for multi-GPU clusters utilizing a GPU-friendly compression scheme to minimize network traffic [24]. While these distributed joins are designed for multi-node clusters, our heterogeneous multi-GPU sort-merge join targets single-node multi-GPU systems.

7 Conclusion

In this master thesis, we present a heterogeneous multi-GPU sort-merge join for large input relations to tackle the challenge of joining unprecedented amounts of data. Our algorithm harnesses the high-bandwidth P2P interconnects of modern multi-GPU systems (e.g., NVLink 2.0, NVLink 3.0, and NVSwitch) to minimize the data transfers via the typically considerably slower CPU-GPU interconnects. It is, thus, to the best of our knowledge, the first P2P-enabled multi-GPU join that supports large out-of-core data exceeding the combined GPU memory capacity. Comprising a merge- or radix partitioning-based multi-GPU *sort* phase, a parallel CPU-based *merge* phase, and a hybrid *join* phase that combines a CPU merge path partition and a binary search-based GPU-accelerated join strategy, our heterogeneous join exploits the compute power of multi-core CPUs and many-core GPUs. We publish our CUDA implementation that overlaps copy and compute operations and utilizes state-of-the-art on-GPU sort, merge, and partition primitives.

We show that our multi-GPU join outperforms up-to-date CPU and GPU baselines regardless of the workload on modern multi-GPU platforms. It is up to 5.5× and 3.3× faster than the multi-threaded CPU sort-merge and radix-hash join baselines. Compared to the multi-GPU sort-merge and hybrid-radix join, it yields speedups of up to 5.9× and 2.5×, respectively. Our multi-GPU join's sort phase impacts its total execution time as much as 76%. The radix partitioning-based multi-GPU sort strategy is 15% to 20% faster than the merge-based strategy, primarily due to utilizing the high-speed P2P interconnects more efficiently. The CPU multiway merge phase saturates the main memory bandwidth on modern high-performance computing systems and negatively impacts our join's performance. Our sort-merge join's hybrid join phase has the least impact on its end-to-end runtime (as little as 24%). If either or both of the two input relations are pre-sorted, it reaches speedups of 14.4× over the multi-GPU hybrid-radix join baseline. We demonstrate that our join scales with increasing numbers of GPUs based on the interconnect topology, although it eventually hits low-bandwidth CPU-CPU and shared-bandwidth CPU-GPU interconnect bottlenecks. It benefits from data skew with up to 12% shorter join durations. We conclude that our multi-GPU join efficiently joins large input relations and is, thus, applicable in GPU-accelerated database systems.

Beyond this thesis, future work should investigate whether a CPU-assisted (radix) partition phase *before* the sort phase instead of the CPU-based merge phase *after* the sort phase enhances our join's performance. Furthermore, it should extend our multi-GPU join with a NUMA-aware workload distribution strategy to mitigate the effects of low-bandwidth CPU-CPU interconnects.

References

[1] A. Adinets and D. Merrill. 2022. *Onesweep: A Faster Least Significant Digit Radix Sort for GPUs*. NVIDIA. Retrieved January 16, 2024 from `https://arxiv.org/pdf/2206.01784.pdf`

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. 2012. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the VLDB Endowment* 5, 10 (2012), 1064–1075. `https://doi.org/10.14778/2336664.2336678`

[3] AMD. 2023. *4th Gen AMD EPYC Processor Architecture*. AMD. Retrieved January 16, 2024 from `https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf`

[4] AMD. 2023. *AMD64 Architecture Programmer's Manual*. AMD. Retrieved January 16, 2024 from `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf`

[5] L. B. Arimilli, B. Blaner, B. C. Drerup, C. F. Marino, D. E. Williams, E. N. Lais, F. A. Campisano, G. L. Guthrie, M. S. Floyd, R. B. Leavens, S. M. Willenborg, R. Kalla, and B. Abali. 2018. IBM POWER9: Processor and System Features for Computing in the Cognitive Era. *IBM Journal of Research and Development* 62, 4/5 (2018), 1–11. `https://doi.org/10.1147/JRD.2018.2859564`

[6] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96. `https://doi.org/10.14778/2732219.2732227`

[7] C. Balkesen, J. Teubner, G. Alonso, and M. T Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *29th International Conference on Data Engineering (ICDE '13)*. IEEE, New York, NY, USA, 362–373. `https://doi.org/10.1109/ICDE.2013.6544839`

[8] M. Bandle, J. Giceva, and T. Neumann. 2021. To Partition, or Not to Partition, That Is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, New York, NY, USA, 168–180. `https://doi.org/10.1145/3448016.3452831`

[9] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. 2014. Memory-Efficient Hash Joins. *Proceedings of the VLDB Endowment* 8, 4 (2014), 353–364. `https://doi.org/10.14778/2735496.2735499`

[10] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proceedings of the VLDB Endowment* 10, 5 (2017), 517–528. `https://doi.org/10.14778/3055540.3055545`

[11] B. Bastem, D. Unat, W. Zhang, A. Almgren, and J. Shalf. 2017. Overlapping Data Transfers with Computation on GPU with Tiles. In *46th International Conference on Parallel Processing (ICPP '17)*. IEEE, New York, NY, USA, 171–

180. https://doi.org/10.1109/ICPP.2017.26

[12] S. Blanas, Y. Li, and J. M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[13] R. Chen and V. K. Prasanna. 2016. Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform. In *24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '16)*. IEEE, New York, NY, USA, 212–219. https://doi.org/10.1109/FCCM.2016.62

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *Transactions on Database Systems* 32, 3 (2007), 1–36. https://doi.org/10.1145/1272743.1272747

[15] X. Chen, Y. Chen, R. Bajaj, J. He, B. He, W.-F. Wong, and D. Chen. 2020. Is FPGA Useful for Hash Joins?. In *10th Conference on Innovative Data Systems Research (CIDR '20)*. CIDR, Chaminade, CA, USA, 1–9. https://www.cidrdb.org/cidr2020/papers/p27-chen-cidr20.pdf

[16] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. 2008. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1313–1324. https://doi.org/10.14778/1454159.1454171

[17] S. Chun, W. D. Becker, J. Casey, S. Ostrander, D. Dreps, J. A. Hejase, R. M. Nett, B. Beaman, and J. R. Eagle. 2018. IBM POWER9: Package Technology and Design. *IBM Journal of Research and Development* 62, 4/5 (2018), 1–10. https://doi.org/10.1147/JRD.2018.2847178

[18] J. Demouth. 2014. *CUDA Pro Tip: Minimize the Tail Effect.* NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/

[19] L. Durant, O. Giroux, M. Harris, and N. Stam. 2017. *Inside Volta: The World's Most Advanced Data Center GPU.* NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/inside-volta/

[20] D. Foley and J. Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17. https://doi.org/10.1109/MM.2017.37

[21] FSF. 2023. *The GNU C++ Library Manual: Parallel Mode.* FSF. Retrieved January 16, 2024 from https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++-manual.pdf.gz

[22] FSF. 2023. *The GNU C++ Library Reference Manual.* FSF. Retrieved January 16, 2024 from https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++-api.pdf.gz

[23] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. 2018. Pipelined Query

Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1603–1618. `https://doi.org/10.1145/3183713.3183734`

[24] H. Gao and N. Sakharnykh. 2021. *Scaling Joins to a Thousand GPUs*. NVIDIA. Retrieved January 16, 2024 from `https://adms-conf.org/2021-camera-ready/gao_adms21.pdf`

[25] I. Gelado and M. Garland. 2019. Throughput-Oriented GPU Memory Allocation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 27–37. `https://doi.org/10.1145/3293883.3295727`

[26] M. Gowanlock, B. Karsin, Z. Fink, and J. Wright. 2019. Accelerating the Unacceleratable: Hybrid CPU/GPU Algorithms for Memory-Bound Database Primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN '19)*. ACM, New York, NY, USA, 1–11. `https://doi.org/10.1145/3329785.3329926`

[27] G. Graefe, A. Linville, and L. D. Shapiro. 1994. Sort vs. Hash Revisited. *IEEE Transactions on Knowledge and Data Engineering* 6, 6 (1994), 934–944. `https://doi.org/10.1109/69.334883`

[28] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 International Conference on Management of Data (SIGMOD '94)*. ACM, New York, NY, USA, 243–252. `https://doi.org/10.1145/191839.191886`

[29] O. Green, R. McColl, and D. A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th International Conference on Supercomputing (ICS '12)*. ACM, New York, NY, USA, 331–340. `https://doi.org/10.1145/2304576.2304621`

[30] C. Gregg and K. Hazelwood. 2011. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance without the Answer. In *2011 International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE, New York, NY, USA, 134–144. `https://doi.org/10.1109/ISPASS.2011.5762730`

[31] T. Gubner, D. Tomé, H. Lang, and P. Boncz. 2019. Fluid Co-Processing: GPU Bloom-Filters for CPU Joins. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN '19)*. ACM, New York, NY, USA, 1–10. `https://doi.org/10.1145/3329785.3329934`

[32] C. Guo and H. Chen. 2019. In-Memory Join Algorithms on GPUs for Large-Data. In *21st International Conference on High Performance Computing and Communications (HPCC '19)*. IEEE, New York, NY, USA, 1060–1067. `https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00151`

[33] C. Guo, H. Chen, F. Zhang, and C. Li. 2019. Distributed Join Algorithms on Multi-CPU Clusters with GPUDirect RDMA. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP '19)*. ACM, New York, NY, USA, 1–10.

https://doi.org/10.1145/3337821.3337862

[34] C. Guo, H. Chen, F. Zhang, and C. Li. 2019. Parallel Hybrid Join Algorithm on GPU. In *21st International Conference on High Performance Computing and Communications (HPCC '19)*. IEEE, New York, NY, USA, 1572–1579. https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00216

[35] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[36] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. 1993. Fixed-Precision Estimation of Join Selectivity. In *Proceedings of the 12th Symposium on Principles of Database Systems (PODS '93)*. ACM, New York, NY, USA, 190–201. https://doi.org/10.1145/153850.153875

[37] P. J. Haas, J. F. Naughton, and A. N. Swami. 1994. On the Relative Cost of Sampling for Join Selectivity Estimation. In *Proceedings of the 13th Symposium on Principles of Database Systems (PODS '94)*. ACM, New York, NY, USA, 14–24. https://doi.org/10.1145/182591.182594

[38] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. 2015. FPGA-Based Multithreading for In-Memory Hash Joins. In *7th Conference on Innovative Data Systems Research (CIDR '15)*. CIDR, Chaminade, CA, USA, 1–9. https://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper12.pdf

[39] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. 2013. Accelerating Join Operation for Relational Databases with FPGAs. In *21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM '13)*. IEEE, New York, NY, USA, 17–20. https://doi.org/10.1109/FCCM.2013.17

[40] M. Harris. 2012. *An Easy Introduction to CUDA C and C++*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/

[41] M. Harris. 2012. *How to Optimize Data Transfers in CUDA C/C++*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[42] M. Harris. 2012. *How to Overlap Data Transfers in CUDA C/C++*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/

[43] M. Harris. 2012. *How to Query Device Properties and Handle Errors in CUDA C/C++*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/how-query-device-properties-and-handle-errors-cuda-cc/

[44] M. Harris. 2013. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[45] M. Harris. 2013. *Using Shared Memory in CUDA C/C++*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

[46] M. Harris. 2014. *CUDA Pro Tip: Occupancy API Simplifies Launch Configuration*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/

[47] M. Harris. 2015. *CUDA Pro Tip: Streams Simplify Concurrency*. NVIDIA. Retrieved January 16, 2024 from https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

[48] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[49] IBM. 2018. *IBM Power System AC922: Technical Overview and Introduction*. IBM. Retrieved January 16, 2024 from https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf

[50] I. Ilic, I. Tolovski, and T. Rabl. 2023. *RMG Sort: Radix-Partitioning-Based Multi-GPU Sorting*. HPI. https://hpi.de/fileadmin/user_upload/fachgebiete/rabl/publications/2023/rmg-sort-ilic.pdf

[51] H. Inoue and K. Taura. 2015. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1274–1285. https://doi.org/10.14778/2809974.2809988

[52] Intel. 2019. *2nd Gen Intel Xeon Scalable Processors*. Intel. Retrieved January 16, 2024 from https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-xeon-scalable-datasheet-vol-1.pdf

[53] Intel. 2023. *Intel IA-64 and IA-32 Architectures Software Developer's Manual*. Intel. Retrieved January 16, 2024 from https://cdrdv2.intel.com/v1/dl/getContent/789583?fileName=325462-sdm-vol-1-2abcd-3abcd-4.pdf

[54] A. Ishii and R. Wells. 2022. The NVLink-Network Switch: NVIDIA's Switch Chip for High Communication-Bandwidth Superpods. In *34th Hot Chips Symposium (HCS '22)*. IEEE, New York, NY, USA, 1–23. https://doi.org/10.1109/HCS55958.2022.9895480

[55] F. Ji, A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, R. Thakur, W.-C. Feng, and X. Ma. 2012. DMA-Assisted, Intranode Communication in GPU Accelerated Systems. In *14th International Conference on High Performance Computing and Communication (HPCC '12)*. IEEE, New York, NY, USA, 461–468. https://doi.org/10.1109/HPCC.2012.69

[56] J. Jung, D. Park, Y. Do, J. Park, and J. Lee. 2020. Overlapping Host-to-Device Copy and Computation Using Hidden Unified Memory. In *Proceedings of the 25th Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*. ACM, New York, NY, USA, 321–335. https://doi.org/10.1145/3332466.3374531

[57] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. 2012. GPU Join Processing Revisited. In *Proceedings of the 8th International Workshop on Data Management on New Hardware (DaMoN '12)*. ACM, New York, NY, USA, 55–62. `https://doi.org/10.1145/2236584.2236592`

[58] K. Kara, J. Giceva, and G. Alonso. 2017. FPGA-Based Data Partitioning. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 433–445. `https://doi.org/10.1145/3035918.3035946`

[59] C. Kim, T. Kaldewey, V. W. Lee, R. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1378–1389. `https://doi.org/10.14778/1687553.1687564`

[60] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy. 2020. *NVIDIA Ampere Architecture In-Depth.* NVIDIA. Retrieved January 16, 2024 from `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`

[61] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. `https://doi.org/10.1109/TPDS.2019.2928289`

[62] A. Li, S. L. Song, J. Chen, X. Liu, N. Tallent, and K. Barker. 2018. Tartan: Evaluating Modern GPU Interconnect Via a Multi-GPU Benchmark Suite. In *2018 International Symposium on Workload Characterization (IISWC '18)*. IEEE, New York, NY, USA, 191–202. `https://doi.org/10.1109/IISWC.2018.8573483`

[63] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (2008), 39–55. `https://doi.org/10.1109/MM.2008.31`

[64] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. 2020. Pump up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. ACM, New York, NY, USA, 1633–1649. `https://doi.org/10.1145/3318464.3389705`

[65] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, New York, NY, USA, 1017–1032. `https://doi.org/10.1145/3514221.3517911`

[66] Z. Majo and T. R. Gross. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*. ACM, New York, NY, USA, 1–10. `https://doi.org/10.1145/1987816.1987832`

[67] T. Maltenberger, I. Ilic, I. Tolovski, and T. Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, New York, NY, USA, 1795–1809.

https://doi.org/10.1145/3514221.3517842

[68] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[69] D. Mayhew and V. Krishnan. 2003. PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects. In *Proceedings of the 11th Symposium on High Performance Interconnects (HOTI '03)*. IEEE, New York, NY, USA, 21–29. https://doi.org/10.1109/CONECT.2003.1231473

[70] D. Merrill and M. Garland. 2016. *Single-Pass Parallel Prefix Scan with Decoupled Look-Back*. NVIDIA. Retrieved January 16, 2024 from https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf

[71] D. Merrill and A. Grimshaw. 2011. High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. *Parallel Processing Letters* 21, 2 (2011), 245–272. https://doi.org/10.1142/S0129626411000187

[72] P. Mishra and M. H. Eich. 1992. Join Processing in Relational Databases. *Comput. Surveys* 24, 1 (1992), 63–113. https://doi.org/10.1145/128762.128764

[73] NVIDIA. 2014. *NVIDIA NVLink: High-Speed Interconnect Application Performance*. NVIDIA. Retrieved January 16, 2024 from https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf

[74] NVIDIA. 2017. *NVIDIA V100 Tensor Core GPU Architecture*. NVIDIA. Retrieved January 16, 2024 from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[75] NVIDIA. 2018. *NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch*. NVIDIA. Retrieved January 16, 2024 from https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf

[76] NVIDIA. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA. Retrieved January 16, 2024 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[77] NVIDIA. 2020. *NVIDIA V100 Tensor Core GPU*. NVIDIA. Retrieved January 16, 2024 from https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

[78] NVIDIA. 2021. *mgpu: Patterns and Behaviors for GPU Computing*. NVIDIA. Retrieved January 16, 2024 from https://github.com/moderngpu/moderngpu

[79] NVIDIA. 2021. *NVIDIA A100 Tensor Core GPU*. NVIDIA. Retrieved January 16, 2024 from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

[80] NVIDIA. 2023. *cub: Cooperative Primitives for CUDA C++*. NVIDIA. Retrieved

January 16, 2024 from `https://github.com/NVIDIA/cub`

[81] NVIDIA. 2023. *CUDA C++ Best Practices Guide.* NVIDIA. Retrieved January 16, 2024 from `https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`

[82] NVIDIA. 2023. *NVIDIA DGX A100 System.* NVIDIA. Retrieved January 16, 2024 from `https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf`

[83] NVIDIA. 2023. *NVIDIA DGX H100 System.* NVIDIA. Retrieved January 16, 2024 from `https://docs.nvidia.com/dgx/dgxh100-user-guide/dgxh100-user-guide.pdf`

[84] NVIDIA. 2023. *thrust: C++ Parallel Algorithms Library.* NVIDIA. Retrieved January 16, 2024 from `https://github.com/NVIDIA/thrust`

[85] NVIDIA. 2024. *CUDA C++ Programming Guide.* NVIDIA. Retrieved January 16, 2024 from `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

[86] NVIDIA. 2024. *NVIDIA GPUDirect RDMA.* NVIDIA. Retrieved January 16, 2024 from `https://docs.nvidia.com/cuda/pdf/GPUDirect_RDMA.pdf`

[87] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. 2012. Merge Path: Parallel Merging Made Simple. In *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12).* IEEE, New York, NY, USA, 1611–1618. `https://doi.org/10.1109/IPDPSW.2012.202`

[88] J. Paul, B. He, S. Lu, and C. T. Lau. 2019. Revisiting Hash Join on Graphics Processors: A Decade Later. In *35th International Conference on Data Engineering Workshops (ICDEW '19).* IEEE, New York, NY, USA, 294–299. `https://doi.org/10.1109/ICDEW.2019.00008`

[89] J. Paul, S. Lu, B. He, and C. T. Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21).* ACM, New York, NY, USA, 1413–1425. `https://doi.org/10.1145/3448016.3457254`

[90] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu. 2019. Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects. In *Proceedings of the 2019 International Conference on Performance Engineering (ICPE '19).* ACM, New York, NY, USA, 209–218. `https://doi.org/10.1145/3297663.3310299`

[91] H. Pirk, S. Manegold, and M. Kersten. 2014. Waste Not… Efficient Co-Processing of Relational Data. In *30th International Conference on Data Engineering (ICDE '14).* IEEE, New York, NY, USA, 508–519. `https://doi.org/10.1109/ICDE.2014.6816677`

[92] O. Polychroniou, A. Raghavan, and K. A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD '15).* ACM, New York, NY, USA, 1493–1508.

https://doi.org/10.1145/2723372.2747645

[93] O. Polychroniou, R. Sen, and K. A. Ross. 2014. Track Join: Distributed Joins with Minimal Network Traffic. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1483–1494. https://doi.org/10.1145/2588555.2610521

[94] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. 2016. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-Speed Networks. In *32nd International Conference on Data Engineering (ICDE '16)*. IEEE, New York, NY, USA, 1194–1205. https://doi.org/10.1109/ICDE.2016.7498324

[95] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. 2014. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *30th International Conference on Data Engineering (ICDE '14)*. IEEE, New York, NY, USA, 592–603. https://doi.org/10.1109/ICDE.2014.6816684

[96] R. Rui, H. Li, and Y.-C. Tu. 2015. Join Algorithms on GPUs: A Revisit After Seven Years. In *2015 International Conference on Big Data (BD '15)*. IEEE, New York, NY, USA, 2541–2550. https://doi.org/10.1109/BigData.2015.7364051

[97] R. Rui, H. Li, and Y.-C. Tu. 2021. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proceedings of the VLDB Endowment* 14, 4 (2021), 708–720. https://doi.org/10.14778/3436905.3436927

[98] R. Rui and Y.-C. Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3085504.3085521

[99] A. Salah, K. Li, Q. Liao, M. Hashem, Z. Li, A. T. Chronopoulos, and A. Y. Zomaya. 2020. A Time-Space Efficient Algorithm for Parallel K-Way In-Place Merging Based on Sequence Partitioning and Perfect Shuffle. *ACM Transactions on Parallel Computing* 7, 2 (2020), 1–23. https://doi.org/10.1145/3391443

[100] P. Sanders. 2001. Fast Priority Queues for Cached Memory. *ACM Journal of Experimental Algorithmics* 5, 1 (2001), 1–25. https://doi.org/10.1145/351827.384249

[101] N. Satish, M. Harris, and M. Garland. 2009. Designing Efficient Sorting Algorithms for Manycore GPUs. In *2009 International Symposium on Parallel and Distributed Processing (IPDPS '09)*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/IPDPS.2009.5161005

[102] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. 2010. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 351–362. https://doi.org/10.1145/1807167.1807207

[103] D. A. Schneider and D. J. DeWitt. 1989. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings*

of the 1989 International Conference on Management of Data (SIGMOD '89). ACM, New York, NY, USA, 110–121. https://doi.org/10.1145/67544.66937

[104] S. Schuh, X. Chen, and J. Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917

[105] A. Shanbhag, S. Madden, and X. Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. ACM, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[106] D. Simmen, E. Shekita, and T. Malkemus. 1996. Fundamental Techniques for Order Optimization. In *Proceedings of the 1996 International Conference on Management of Data (SIGMOD '96)*. ACM, New York, NY, USA, 57–67. https://doi.org/10.1145/233269.233320

[107] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *35th International Conference on Data Engineering (ICDE '19)*. IEEE, New York, NY, USA, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[108] M. Sourouri, T. Gillberg, S. B. Baden, and X. Cai. 2014. Effective Multi-GPU Communication Using Multiple CUDA Streams and Threads. In *20th International Conference on Parallel and Distributed Systems (ICPADS '14)*. IEEE, New York, NY, USA, 981–986. https://doi.org/10.1109/PADSW.2014.7097919

[109] E. Stehle and H.-A. Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 417–432. https://doi.org/10.1145/3035918.3064043

[110] I. Tanasic, L. Vilanova, M. Jordà, J. Cabezas, I. Gelado, N. Navarro, and W.-M. Hwu. 2013. Comparison Based Sorting for Systems with Multiple GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU '13)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2458523.2458524

[111] L. Thostrup, G. Doci, N. Boeschen, M. Luthra, and C. Binnig. 2023. Distributed GPU Joins on Fast RDMA-Capable Networks. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26. https://doi.org/10.1145/3588709

[112] Z. Wang, J. Paul, B. He, and W. Zhang. 2017. Multikernel Data Partitioning with Channel on OpenCL-Based FPGAs. *IEEE Transactions on Very Large Scale Integration Systems* 25, 6 (2017), 1906–1918. https://doi.org/10.1109/TVLSI.2017.2653818

[113] B. van Werkhoven, J. Maassen, F. J. Seinstra, and H. E. Bal. 2014. Performance Models for CPU-GPU Data Transfers. In *14th International Symposium on Cluster, Cloud and Grid Computing (CCGRID '14)*. IEEE, New York, NY, USA, 11–20.

https://doi.org/10.1109/CCGrid.2014.16

[114] S. Widmer, D. Wodniok, N. Weber, and M. Goesele. 2013. Fast Dynamic Memory Allocator for Massively Parallel Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU '13)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/2458523.2458535

[115] M. Winter, M. Parger, D. Mlakar, and M. Steinberger. 2021. Are Dynamic Memory Managers on GPUs Slow? A Survey and Benchmarks. In *Proceedings of the 26th Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. ACM, New York, NY, USA, 219–233. https://doi.org/10.1145/3437801.3441612

[116] M. Yabuta, A. Nguyen, S. Kato, M. Edahiro, and H. Kawashima. 2017. Relational Joins on GPUs: A Closer Look. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2663–2673. https://doi.org/10.1109/TPDS.2017.2677451

[117] Y. Yuan, R. Lee, and X. Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210

[118] J. Zhou. 2009. Evaluation of Relational Operators. In *Encyclopedia of Database Systems*. Springer, Boston, MA, USA, 1024–1029. https://doi.org/10.1007/978-0-387-39940-9_154