Efficiently Joining Large Relations on Multi-GPU Systems

Tobias Maltenberger
tobias.maltenberger@student.hpi.de
Hasso Plattner Institute
Potsdam, Germany

Ilin Tolovski
ilin.tolovski@hpi.de
Hasso Plattner Institute
Potsdam, Germany

Tilmann Rabl
tilmann.rabl@hpi.de
Hasso Plattner Institute
Potsdam, Germany

ABSTRACT

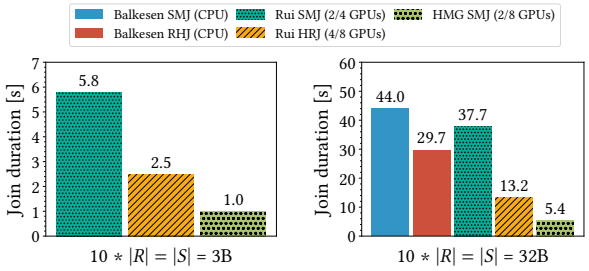
Growing data volumes present a mounting challenge to relational joins. GPUs have gained widespread adoption as database accelerators for operators such as joins due to their high instruction throughput and memory bandwidth. Most published GPU-accelerated joins are single-GPU algorithms that do not leverage modern multi-GPU platforms effectively. The few proposed multi-GPU algorithms either fail to exploit the high-speed P2P interconnects between the GPUs or to handle large out-of-core data natively. In this paper, we present a heterogeneous multi-GPU sort-merge join that overcomes both limitations. It is composed of 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 by up to 15.2× and 5.5×, respectively. Compared to non-P2P-enabled multi-GPU joins, it achieves speedups of 8.7× (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 22% to its end-to-end runtime. If the input relations are pre-sorted, it is up to 14.4× faster than the hybrid-radix join. Our join scales well with the number of GPUs and benefits from data skew with as much as 12% shorter join durations.

PVLDB Reference Format:
Tobias Maltenberger, Ilin Tolovski, and Tilmann Rabl. Efficiently Joining Large Relations on Multi-GPU Systems. PVLDB, 18(11): 4653-4667, 2025.
doi:10.14778/3749646.3749720

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hpides/multi-gpu-sort-merge-join.

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 [36]. Therefore, researchers and engineers continuously adapt join algorithms to harness the latest advances in hardware technology [9, 14, 15, 40, 57]. Modern multi-core architectures led to sophisticated workload partitioning strategies, cache optimization techniques, and single instruction, multiple data (SIMD) operations for relational joins [10, 11, 16, 59, 68, 91]. Similarly, the rise of many-core graphics processing units (GPUs) inspired numerous GPU-accelerated joins [35, 48, 87, 97, 105]. Due to the high instruction throughput and memory bandwidth of GPUs [78, 85], these algorithms often outperform parallel CPU joins by an order of magnitude [97, 105]. Most of the published GPU-accelerated joins are single-GPU approaches that leave the 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 join tuples fit completely into GPU memory. Although the on-chip GPU memory has increased over the past few years up to 256 GB [5], it still sets an upper limit on the size of the input relations that a single-GPU join can process.



(a) IBM AC922 **(b) NVIDIA DGX H100**

Figure 1: Join baseline comparison with 8-byte tuples

Only a few multi-GPU approaches have been proposed. Paul et al. describe a partitioned multi-GPU hash join featuring a multi-hop routing strategy for efficient P2P data transfers between asymmetrically connected GPUs [88]. Their join exploits the P2P interconnects between the GPUs but lacks support for large out-of-core data. It assumes that the input relations and the intermediate join state fit entirely into GPU memory. Moreover, the growing availability of symmetric switch-based P2P interconnects (e.g., NVSwitch) negates the utility of its multi-hop routing strategy [74]. Rui et al. present two multi-GPU join algorithms that can handle large out-of-core data natively: a sort-merge join and a hybrid-radix join [96]. 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. 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 size of the input relations, neither harnesses the high-bandwidth P2P interconnects between the GPUs, which facilitate reducing the data transfers over the typically slower

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749720

CPU-GPU interconnects [60, 61, 67]. Hence, the imperative arises to develop a novel multi-GPU join algorithm that fully utilizes modern multi-GPU systems with high-speed interconnects.

In this paper, we propose a heterogeneous multi-GPU sort-merge join that supports large out-of-core data and 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-assisted merge path partition strategy and executes a binary search-based *m*:*n* 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 4.0, and NVSwitch interconnects such as the IBM AC922 and NVIDIA DGX H100 [49, 81]. We compare its 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 joins [9, 96]. We study the impact of our sort-merge 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. Lastly, we validate our join's real-world applicability using two join-intensive TPC-H queries [110].

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 H100, it achieves up to 8.7× (sort-merge) and 2.5× (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, it yields speedups of 15.2× and 5.5×, respectively. We measure that our join's sort phase contributes as much as 78% to its execution time. We observe that the radix partitioning-based sort strategy is between 12% 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 parallel CPU merge phase saturates the main memory bandwidth. Our join's performance exceeds that of the fastest CPU and GPU baseline by up to 2.7× and 1.2×, respectively, even with a merge phase. We find that the join phase has as little as 22% impact on our join's runtime. If both of the input relations are pre-sorted, it reaches speedups of 14.4× (IBM AC922) and 13.7× (NVIDIA DGX H100) over the hybrid-radix join. We demonstrate that our join scales well with the number of GPUs based on the interconnect topology. Unlike the CPU and GPU baselines, it benefits from data skew with 12% shorter join durations.

With this paper, we make the following contributions:

(1) We propose a novel multi-GPU sort-merge join that exploits the fast P2P interconnects of modern multi-GPU platforms and can handle large out-of-core data natively.
(2) We conduct in-depth experiments for two workloads to study our sort-merge join's efficiency in joining large input relations on multi-GPU systems.
(3) We publish our high-performance join implementation that utilizes state-of-the-art CPU and on-GPU primitives.

2 BACKGROUND

In this section, we outline the hardware characteristics of modern GPU architectures and GPU interconnects.

2.1 GPU Architectures

GPUs offer massively parallel compute capabilities. Unlike CPUs that are designed to execute a few tens of threads as fast as possible and hide 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 [84]. By way of illustration, the two top-of-the-line GPUs, NVIDIA V100 (Volta) and NVIDIA H100 (Hopper), achieve 32/64-bit floating-point throughput rates of up to 15.7/7.8 TFLOPS and 66.9/33.5 TFLOPS, respectively [73, 82]. GPUs are built around a scalable array of streaming multiprocessors (SMs). The NVIDIA V100 and NVIDIA H100 comprise 80/132 SMs, each containing 64/128 INT32 and FP32 as well as 32/64 INT64 and FP64 cores [73, 82]. 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* [62].

In addition to a many-core compute architecture, GPUs feature a high-bandwidth memory hierarchy with *off-chip* and *on-chip* memory [84]. Off-chip memory consists mainly of global memory (HBM). In the case of the NVIDIA V100 and NVIDIA H100, the bandwidth of global memory is 900 GB/s and 3352 GB/s, respectively [73, 82]. 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 80 GB for the NVIDIA H100) [76, 85]. 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 transactions as possible to improve transfer efficiency [44]. The L2 cache hides the global memory latency by caching loads and stores to it. On the NVIDIA V100, its capacity is 6 MB, while on the NVIDIA H100, it is 50 MB [73, 82]. On-chip memory per SM includes low-latency shared memory as well as the L1 cache and the register file. Typically, shared memory serves as user-managed scratchpad memory, while the L1 cache transparently hides the global memory access latency of all parallel threads executed by the SM [45]. On the NVIDIA V100 and NVIDIA H100, shared memory and the L1 cache are physically combined on each SM with a capacity of 128 KB and 256 KB, respectively [6, 21]. The register file size per SM is 256 KB on both 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 [60, 61, 89].

Traditionally, PCIe has been the standard CPU-GPU and P2P interconnect. It is a serial communication bus usually composed of 16 bi-directional *lanes* per *link* [69]. PCIe 5.0 lanes offer a peak data transfer rate of 4 GB/s. One PCIe 5.0 link thus reaches a unidirectional bandwidth of 64 GB/s. If multiple GPUs are connected

to the same PCIe link via a *switch*, the bandwidth is shared between the GPUs [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 [22, 72]. NVLink 2.0 and NVLink 4.0 achieve a data transfer rate of 25 GB/s per link in each direction [73, 82]. NVLink 2.0-enabled GPUs (e.g., NVIDIA V100) support up to six links. Consequently, the uni-directional bandwidth of P2P data transfers between two NVIDIA V100 GPUs is 150 GB/s. NVLink 4.0-powered GPUs (e.g., NVIDIA H100) feature eighteen links and offer P2P data transfers at a rate of up to 450 GB/s between two GPUs. The IBM AC922 harnesses NVLink 2.0 for both its CPU-GPU and P2P interconnects [49]. NVSwitch is an NVLink-based switch for all-to-all P2P communication between up to 16 GPUs [74]. NVLink 2.0- and NVLink 4.0-powered NVSwitch enables bandwidths of 150 GB/s and 450 GB/s between two GPUs [54]. The NVIDIA DGX H100 uses NVLink 4.0-based NVSwitch for its P2P interconnects [81].

Most multi-GPU platforms are dual-socket non-uniform memory access (NUMA) systems with an equal number of GPUs connected to each socket. On such systems, 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*, 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, AMD Infinity Fabric, and Ultra Path Interconnect (UPI) by Intel [3, 7, 19, 52].

In times when PCIe was state-of-the-art, 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 [25, 31, 103, 117]. Since fast interconnects such as NVLink and NVSwitch have emerged, GPU-based joins that outperform their CPU baselines for large input relations have been proposed [64, 65]. Considering the interconnect topology in the design of GPU-accelerated database operations is crucial, though, especially on systems with heterogeneous CPU-CPU, CPU-GPU, and P2P interconnects. On such systems, using the compute power of both multi-core CPUs and many-core GPUs can mitigate the data transfer bottleneck [28, 32, 90, 105].

3 ALGORITHM

In this section, we present a heterogeneous multi-GPU sort-merge join for large out-of-core data exceeding the combined GPU memory capacity. Our algorithm consists of 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 partition with a multi-GPU-accelerated join strategy (see Section 3.3).

3.1 Sort Phase

The multi-GPU-accelerated sort phase sorts the tuples (i.e., key-value pairs) of the two input relations R and S by key in *chunksets* comprising equal-sized *chunks* across the g GPUs. It supports two sort strategies: merge-based (for arbitrarily typed tuples) and radix

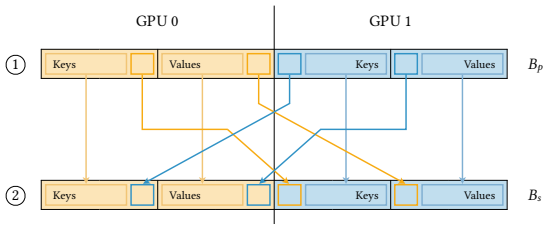


Figure 2: P2P data transfers between $g = 2$ GPUs

partitioning-based (for numerically typed tuples). Although both approaches utilize the high-speed P2P interconnects of modern multi-GPU platforms for inter-GPU data exchanges, they differ in how tuples 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 the two multi-GPU sort strategies 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, both approaches share the logic to interleave data transfers to and from the GPUs with in-core compute operations utilizing two buffers for the keys and values of R and S.

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 [108]. 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. Then, 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 chunks are copied back into main memory.

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, 100, 107]. 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, 73, 75, 101, 107].

We evaluate two on-GPU sort primitives 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` [77, 83]. Our micro-benchmark shows that `thrust::sort_by_key` outperforms `mgpu::mergesort` by up to 4.1× on the IBM AC922 and 4.5× on the NVIDIA DGX H100. 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 GPU memory allocations are very expensive [84], we pass our allocator operating on pre-allocated global memory to the on-GPU sort primitive.

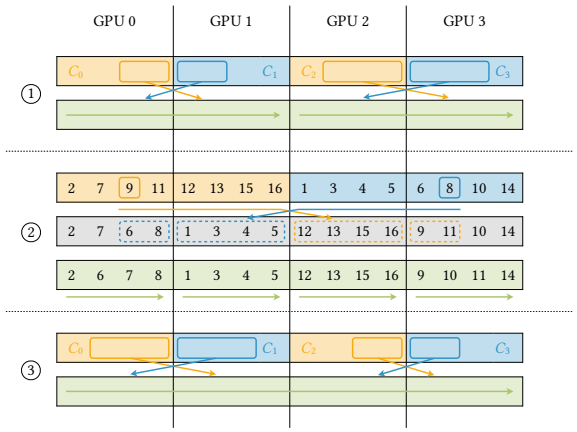


Figure 3: Block shuffling for $g = 4$ GPUs

Multi-GPU P2P Block Shuffling. Bringing the locally sorted chunks into 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. 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 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 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, their by-key order is preserved. Our implementation uses 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 2. 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 on other streams. Once all operations have been completed, C_0 and C_1 contain two sorted key and value blocks that are merged in the 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 3). We follow a recursive divide-and-conquer approach for merging c chunks by bringing the *left* and *right* halves of the chunks into ascending order 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.

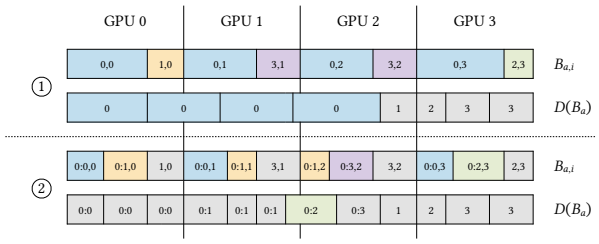


Figure 4: Chunk partitioning for $g = 4$ GPUs

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 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 [30, 77, 83].

We evaluate the performance of the two single-GPU merge primitives 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::merge` on the IBM AC922. On the NVIDIA DGX H100, it outperforms its counterpart from the `mgpu` library by 1.6×. We use `thrust::merge_by_key` as the on-GPU merge primitive in our multi-GPU merge sort implementation. Since the algorithm from the `thrust` library operates on auxiliary key and value buffers and has a memory overhead of up to 64 MB, we pass it our allocator managing pre-allocated global memory.

3.1.2 Multi-GPU Radix Sort. The radix partitioning-based multi-GPU sort approach augments the sort algorithm by Ilic et al. with support for key-value pairs and data exceeding the combined GPU memory capacity [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 redistributed among the GPUs in a single all-to-all P2P bucket exchange so that 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 each chunk resides in 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 zero-initialized buckets with $m = 8$ 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

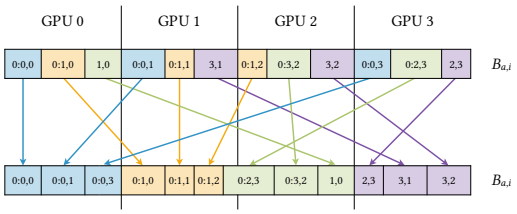


Figure 5: Bucket swapping for $g = 4$ GPUs

cub [70, 79]. 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 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 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 some GPUs to host slightly more key-value pairs than others.

Figure 4 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 among all GPUs reveals that the spanning bucket $B_{0,0}, B_{0,1}, ..., B_{0,3}$ has been eliminated. Since the partitioning strategy yields 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 an all-to-all P2P bucket exchange (see Figure 5). Our implementation uses the secondary key and value buffers for the P2P bucket swapping to avoid blocking 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 on another stream. The CUDA runtime coalesces the asynchronous memory copy operations for the keys and values of adjacent buckets into one memory transaction.

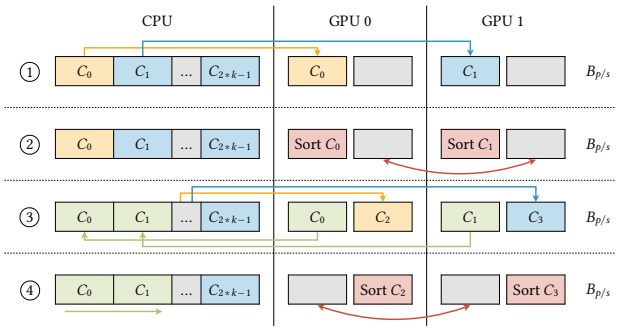


Figure 6: Sort pipeline for large out-of-core data

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 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, 79]. To reduce the number of buckets to sort and, by extension, minimize the kernel launch overhead of the on-GPU sort primitive, our implementation fuses neighboring buckets with the same number of MSB radix partitioning passes whose combined size is less than the 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 another 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), 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 6. It harnesses two non-blocking streams to overlap the data transfers to and from the GPUs and saturate the bi-directional CPU-GPU interconnect bandwidth [47].

Due to the high latency of CUDA memory allocations (e.g., up to 276 ms and 28 ms for 16 GB on the IBM AC922 and NVIDIA DGX H100, respectively) [27, 113, 114], 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

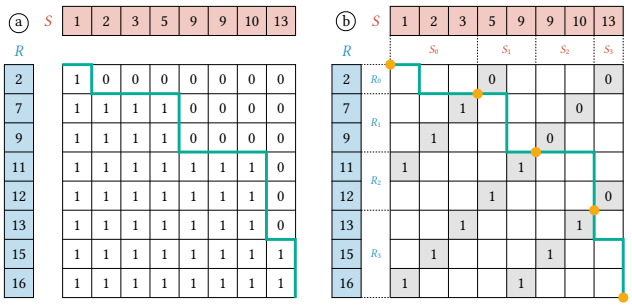


Figure 7: Merge path for the keys in R and S

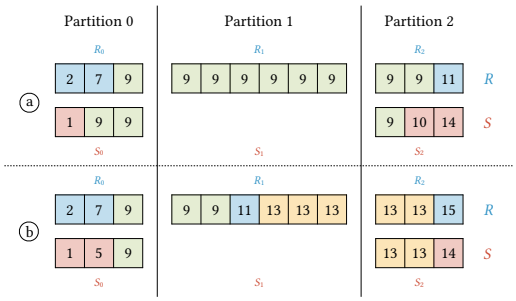


Figure 8: Skewed partition pairs for $p = 3$ partitions

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 utilize a single host allocator and g device allocators for all memory allocations, including those via on-GPU sort, merge, and scan primitives.

3.2 Merge Phase

The CPU-assisted merge phase 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.

3.2.1 CPU Multiway Merge. Merging the k_R and k_S chunksets requires a multiway merge primitive. The best conceivable time complexity of a comparison-based multiway merge algorithm is $O(n * \log(k))$, where n is the number of tuples and k is the number of sorted sublists. Both in-place and out-of-place algorithms with a space complexity of $O(n)$ have been published [18, 51, 98]. `__gnu_parallel::multiway_merge` from the `libstdc++` parallel mode is a runtime-optimal multi-threaded CPU primitive [23, 24]. It uses a register-optimized merge strategy with unrolled loops for $k \in \{2, 3, 4\}$ and a generic loser tree-based strategy for $k \geq 5$ [99]. CPU multiway merge algorithms are memory bandwidth-bound [18, 51]. Maltenberger et al. show that `__gnu_parallel::multiway_merge` saturates the main memory bandwidth of modern HPC systems [67]. Since GPU multiway merge algorithms thus yield no performance gain, we use the primitive in our multi-GPU join implementation.

3.2.2 Zero-Copy Zip Iterator Handling. The multi-threaded CPU primitive `__gnu_parallel::multiway_merge` lacks 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 primitive as is, albeit with time and space overheads of $O(n)$, we adapt it to operate on pointer-based 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 them, 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 key-value pairs is 5.6× (of which 64% is due to eliminating memory allocations). On the NVIDIA DGX H100, using zero-copy zip iterators is 33.3× faster than relying on key-value pairs, where 92% is due to avoiding memory allocations.

3.3 Join Phase

The multi-GPU-accelerated join phase splits the by-key sorted input relations R and S into g *partitions* composed of smaller *subpartitions* via a CPU-assisted merge path partition strategy and joins the disjoint subpartition pairs by key across the g GPUs. Its pipelined execution model allows for running three operations simultaneously on each GPU: (1) copying the keys of a disjoint subpartition pair to global memory, (2) executing the binary search-based m:n merge-join kernel on the keys of a subpartition pair to produce a set of matching key ranges, and (3) copying a set of matching key ranges back into main memory for parallel CPU materialization.

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 partitions, each containing at least three subpartitions whose keys fit into the g GPUs' global memory. We determine the partition and subpartition boundaries through a key-based two-step merge path partitioning [86]. 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 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* [77].

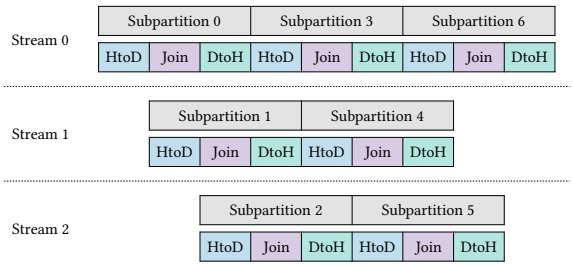


Figure 9: Join pipeline with $s = 3$ streams

Figure 7 illustrates the merge path within a merge matrix for the keys in R and S. It is the traversal path from the upper-left to the lower-right corner while moving rightward if the key in S is smaller than that in R (i.e., the cell value is 1) or downward if the key in S is greater than or equal to that in R (i.e., the cell value is 0). 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 7a). The i-th point on the merge path lies on the i-th cross diagonal in the merge matrix (see Figure 7b). Conceptually, partitioning a merge path into p equal-sized segments, by finding its intersections 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 of Figure 7b, the merge path partitioning of the keys in R and S yields $p = 4$ equal-sized partition pairs (e.g., $R_0 = (2)$ with $S_0 = (1, 2, 3)$). However, since equal keys in R and S might end up in different partition pairs, partitioning a merge path without validating the boundaries yields no valid distribution of the workload for *joining* R and S across p processors. In the example of Figure 7b, 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, thus, 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 8 exemplifies the merge path partitioning of the keys in R and S for $p = 3$ with skewed partition pairs. In both examples, 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 8a), we compute the key's entire range in R and S, exclude it from both input relations, and save the matching key ranges. If no (see Figure 8b), 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 partitions comprising at least three 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 merge-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 for parallel CPU materialization. Each pipeline performs all three operations simultaneously, as depicted in Figure 9. For our sort strategies, 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). For our merge join, however, it enhances performance as the g GPUs join *independently* [13, 56, 106, 112]. 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 CPU threads. If the tuples comprise variable-length values (e.g., strings), it operates on pointers for joining and materializing intermediate join tuples.

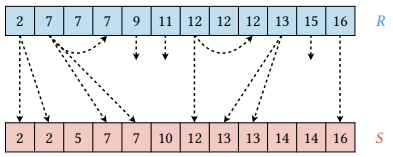
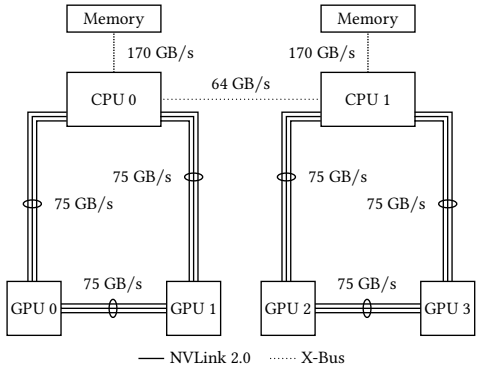
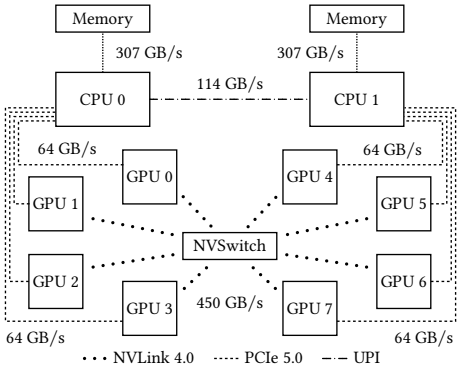


Figure 10: Range search for the keys in R and S

3.3.3 In-Core Join Processing. Once the keys of two 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 *m:n* merge-join kernel. Suppose $|R| \leq |S|$, 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 subpartitions of R and S. In j_R, we store the *last* index in the subpartition of R whose key is equal to that at 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 at index i_R. The matching key range is denoted by $([i_R, j_R], [i_S, j_S])$. Figure 10 shows the range search for the keys of two 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 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 buffer in main memory.

Table 1: Multi-GPU systems

(a) IBM AC922	(b) NVIDIA DGX H100
	
2x IBM POWER9 (16x 2.7 GHz)	2x Intel Xeon 8480CL (56x 2.0 GHz)
4x NVIDIA V100 SXM2 32 GB	8x NVIDIA H100 SXM5 80 GB
2x 256 GB DDR4	2x 1024 GB DDR5

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 to maximize the *occupancy* (i.e., ratio of active warps to possible active warps) of each streaming multiprocessor [20, 46]. It registers a 32-byte L2 cache fetch granularity for the kernel to read eight 32-bit or four 64-bit keys at once from global memory during the binary search-based range searches and, thus, hide the L2 cache latency [80]. Since the number of matching key ranges for two subpartitions of R and S is unknown in advance, our implementation maps the key range buffer residing in pinned main memory into the device address space and transfers each matching key range back concurrently to avoid allocating a fixed-size key range buffer in device memory that remains entirely unused [84].

4 EVALUATION

In this section, we evaluate the performance of our heterogeneous multi-GPU sort-merge join implementation. In Section 4.1, we elaborate on our experimental setup. In Section 4.2, we compare the runtime of our join with that of state-of-the-art CPU-based and GPU-accelerated join algorithms. After that, we analyze our join's execution breakdown (see Section 4.3) and study its scalability for increasing numbers of GPUs (see Section 4.4) and robustness against selectivity and data skew (see Section 4.5). Finally, in Section 4.6, we evaluate its real-world applicability with two TPC-H queries [110].

4.1 Experimental Setup

In this subsection, we provide details of the multi-GPU systems and the methodology used in our performance benchmarks. Besides that, we describe our workloads and baselines.

4.1.1 Hardware Platforms. We evaluate our novel multi-GPU sort-merge join on two dual-socket multi-GPU systems with state-of-the-art interconnects: IBM AC922 and NVIDIA DGX H100 (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 H100 has eight NVIDIA H100 GPUs (with 80 GB of GPU memory) and all-to-all NVLink 4.0-based NVSwitch P2P interconnects offering uni-directional inter-GPU data transfer rates up to 450 GB/s [81]. The platform harnesses PCIe 5.0 for the CPU-GPU interconnects and Ultra Path Interconnect (UPI) with a bandwidth of 114 GB/s per direction between the NUMA nodes.

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 benchmarks to facilitate comparability with related work [2, 8, 14, 59, 96, 97, 105]. 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 attached to the first socket to minimize variability in memory access latency due to NUMA effects. 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 $\mathcal{G}^g$ for benchmarks involving g GPUs based on the platform's interconnect topology (e.g., $\mathcal{G}^2 = \{0, 1\}$ on the IBM AC922 and $\mathcal{G}^2 = \{0, 1\}$ as well as $\mathcal{G}^4 = \{0, 1, 2, 3\}$ on the NVIDIA DGX H100).

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 in line with recent related work [2, 9, 88, 96, 102, 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).

Table 2: Workloads for the scale factor 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

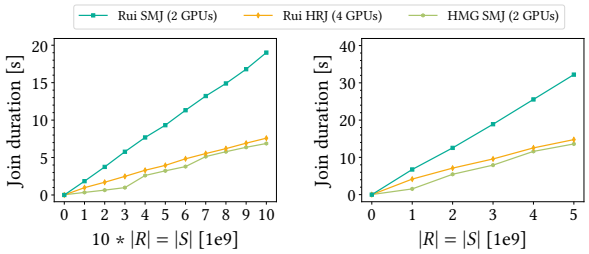
We study two workloads at different scale factors f (see Table 2). In workload **A**, R and S contain 8-byte tuples with 32-bit keys and values, where $10 * |R| = |S|$. In workload **B**, R and S with $|R| = |S|$ comprise 16-byte tuples with 64-bit keys and values.

4.1.4 Join Baselines. We compare the performance of our novel multi-GPU 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. [9]. Both algorithms utilize 256-bit SIMD instructions while employing multi-threaded and cache-conscious workload partitioning and processing strategies. Since the two 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 H100. Both algorithms have been used extensively as baselines for hardware-accelerated joins [17, 39, 58, 65, 96, 97, 109, 111]. Our GPU baselines are the multi-GPU-accelerated sort-merge join and hybrid-radix join by Rui et al. [96]. Both algorithms support large out-of-core data but leave the high-bandwidth P2P interconnects of modern multi-GPU systems unused. Remedying that shortcoming is infeasible, as the two joins assume a shared-nothing architecture.

4.2 Baseline Comparison

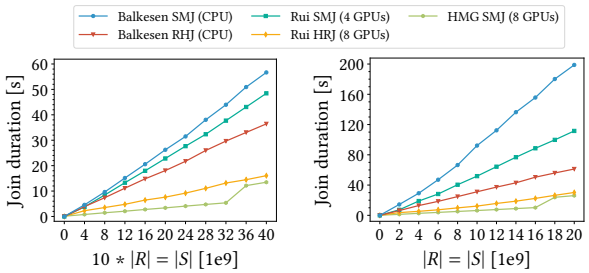
In this subsection, we compare the runtime of our join with that of the CPU and GPU baselines for workloads **A** and **B**.

On the **IBM AC922**, the optimal GPU set for our multi-GPU join with the radix partitioning-based sort strategy is $\mathcal{G}^2 = \{0, 1\}$ (see Section 4.3 and Section 4.4). The multi-GPU-accelerated sort-merge and hybrid-radix join by Rui et al. achieve the shortest join durations with $g = 2$ and $g = 4$ GPUs, respectively [96]. Figure 11a 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 by 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 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 worst-case speedups over the baselines are 2.8× (sort-merge) and 1.1× (hybrid-radix). Figure 11b depicts the join comparison for workload **B** with $f \in [0, 5]$. Our join exhibits a similar performance pattern when R and S comprise 16-byte tuples. It outperforms the join baselines by 4.2× (sort-merge) and 2.5× (hybrid-radix) for up to 1.5B tuples in S. Once $|S|$ exceeds 1.5B 16-byte tuples (i.e., in the out-of-core range until it reaches the main memory capacity), the speedups over the multi-GPU-based sort-merge and hybrid-radix join diminish to 2.4× and 1.1× to 1.2×. Unlike workload **A**, for which only S requires a merge phase, both R and S require a merge phase for workload **B**.



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

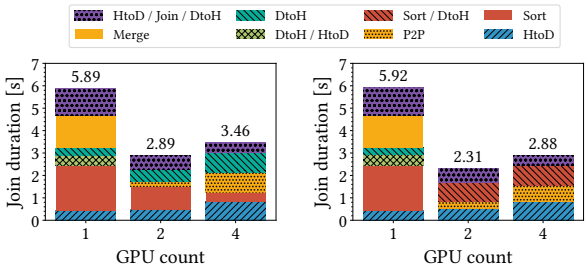
Figure 11: Baseline comparison on the IBM AC922



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

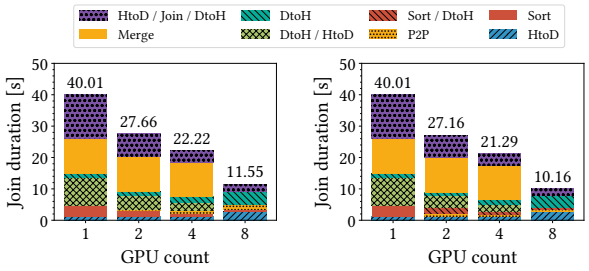
Figure 12: Baseline comparison on the NVIDIA DGX H100

On the **NVIDIA DGX H100**, our 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 112 cores distributed between two NUMA nodes [9]. The fastest GPU sets for the sort-merge and hybrid-radix join by Rui et al. are $\mathcal{G}^4 = \{0, 1, 2, 3\}$ and $\mathcal{G}^8$ with all $g = 8$ GPUs [96]. 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 32B tuples, as illustrated in Figure 12a. It is 8.2× faster than the CPU sort-merge join and 5.5× faster than the CPU radix-hash join. It outperforms the GPU baselines by 7.0× (sort-merge) and 2.4× (hybrid-radix) when the input relation S fits into the GPU memory of all $g = 8$ GPUs. When $|S|$ is in the out-of-core range (from 32B tuples), the worst-case speedups over the fastest CPU and GPU joins are 2.7× 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 by 15.2× and radix-hash join by 4.9× for up to 16B tuples in S, as illustrated in Figure 12b. It is 8.7× (sort-merge) and 2.2× (hybrid-radix) faster than the GPU joins. Since the sort-merge join by Balkesen et al. uses AVX instructions only for 8-byte tuples [9], it performs disproportionately worse for 16-byte tuples.



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

Figure 13: Execution breakdown on the IBM AC922



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

Figure 14: Execution breakdown on the NVIDIA DGX H100

4.3 Execution Breakdown

In this subsection, we analyze the impact of our join's three phases (i.e., *sort*, *merge*, and *join*) on its end-to-end runtime for the merge- and radix partitioning-based 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 $\mathcal{G}^1 = \{0\}$ and $\mathcal{G}^2 = \{0, 1\}$. 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 $\mathcal{G}^2$ (see Section 4.4).

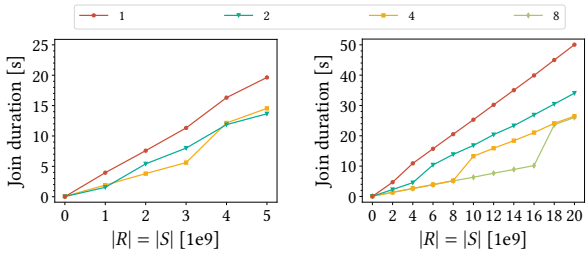
Figure 13a illustrates the critical-path duration breakdown with the *merge-based* multi-GPU sort strategy. Relative to the total execution time of 5.89 s for $g = 1$ GPU, the sort, merge, and join operations amount to 34%, 24%, and 21%, respectively. Since our heterogeneous sort-merge join operates on a chunk size of 750M tuples for 16-byte key-value 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 by 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 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 (see Table 1) [67]. The X-Bus also slows down the concurrent HtoD (24%) and DtoH (26%) data transfers on $g = 4$ GPUs.

Figure 13b shows the critical-path duration 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 P2P interconnects (see Table 1), the P2P bucket swapping is 2.1× slower on four GPUs compared to two GPUs. Simultaneously sorting and copying buckets back into main memory (DtoH) takes 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 with radix sort is always less than or equal to that with merge sort on the IBM AC922.

On the **NVIDIA DGX H100**, we dissect our join's execution for workload **B** with $f = 16$ on $g \in \{1, 2, 4, 8\}$ GPUs. The fastest GPU sets for $g < 8$ are $\mathcal{G}^1 = \{0\}$, $\mathcal{G}^2 = \{0, 1\}$, and $\mathcal{G}^4 = \{0, 1, 2, 3\}$. By choosing $|R| = |S|$ equal to 16B tuples, we maximize GPU utilization for the system's *overall* best GPU set $\mathcal{G}^8$ (see Section 4.4).

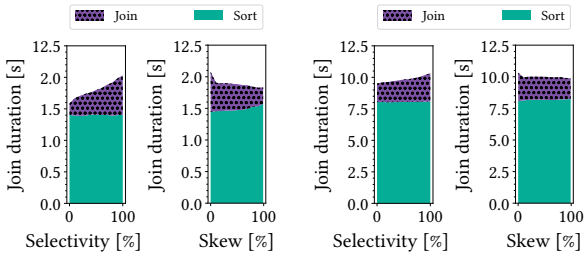
With the *merge-based* multi-GPU sort strategy (see Figure 14a), the performance of our join improves for increasing numbers of GPUs $g \in \{1, 2, 4, 8\}$ from 40.01 s ($g = 1$) to 11.55 s ($g = 8$) up to 3.5×. Up to four GPUs, R and S exceed the GPU memory capacity of $g = 1$ (80 GB), $g = 2$ (160 GB), and $g = 4$ (320 GB) GPUs and require a CPU merge phase. Since our multi-GPU join works with a chunk size of 2B 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. On the NVIDIA DGX H100, the CPU primitive `__gnu_parallel::multiway_merge` runs equally fast for different numbers of sublists (i.e., chunksets) $k \in [2, 5]$. Up to four GPUs, the execution times of the sort, join, and overlapped HtoD and DtoH copy operations halve for $g \rightarrow 2{*}g$ as the bandwidth of the PCIe 5.0 CPU-GPU interconnects is not shared between any GPUs in the optimal GPU sets $\mathcal{G}^1$, $\mathcal{G}^2$, and $\mathcal{G}^4$ (see Table 1). On $g = 8$ GPUs, the runtime of the HtoD and DtoH copy operations is 1.3× higher than on $g = 4$ GPUs due to the shared UPI-based CPU-CPU interconnect bandwidth of 114 GB/s (see Table 1). It amounts to 63% of the total join duration. The impact of the join operation on the critical-path execution time is 20%, while the remaining 18% are split between on-GPU chunk sorting and P2P block shuffling.

With the *radix partitioning-based* sort strategy (see Figure 14b), utilizing $g = 8$ GPUs (10.16 s) yields 3.9× shorter join durations than using $g = 1$ GPU (40.01 s). Our join's performance for $g \in \{2, 4, 8\}$ GPUs is always better with multi-GPU radix sort than merge sort



(a) IBM AC922 (b) NVIDIA DGX H100

Figure 15: Scalability for increasing numbers of GPUs



(a) IBM AC922 (b) NVIDIA DGX H100

Figure 16: Robustness against varying selectivity and skew

(2% to 12%) 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 operations on the critical-path execution time is 5%, 22%, and 6%, respectively.

4.4 Scalability Analysis

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

On the **IBM AC922** with $f \in [0, 5]$ (see Figure 15a), our 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 join's runtime reduces for $g = 1 \rightarrow 2$ GPUs (G_0 and G_1) by 2.6× if R and S fit into the 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 GPUs. The *overall* best GPU set on the IBM AC922 is $\mathcal{G}^2 = \{0, 1\}$.

On the **NVIDIA DGX H100** with $f \in [0, 20]$ (see Figure 15b), 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 by 2.4× up to 4B tuples in S and 1.5× in the out-of-core range. With $g = 4$ GPUs (G_0, G_1, G_2, and G_3), its runtime reduces by 3.9× until $|S|$ equals 8B tuples and 1.9× beyond. Our join is fastest on $g = 8$ GPUs with speedups of up to 3.9× ($g = 1$), 2.6× ($g = 2$), and 2.1× ($g = 4$). $\mathcal{G}^8$ is the *overall* best GPU set.

4.5 Robustness Analysis

In this subsection, we study the impact of selectivity and data skew on our multi-GPU sort-merge join using workload **B** with $|R| = |S|$ equal to 1.5B tuples on the IBM AC922 ($\mathcal{G}^2$) in Figure 16a and 16B on the NVIDIA DGX H100 ($\mathcal{G}^8$) in Figure 16b.

Selectivity Analysis. We relax the foreign key constraint between R and S when decreasing the selectivity factor $\sigma \in [0, 1]$ so that every key in S has *at most one* instead of *exactly one* matching key in R as per Haas et al. [37, 38]. For $\sigma = 0 \rightarrow 1$, the sort phase remains stable, while the join phase slows down by 3.4× (IBM AC922) and 1.5× (NVIDIA DGX H100) as more and more keys in R entail running *three* instead of *two* binary searches (to find their *last* index in R, *first* index in S, and *last* index in S), incrementing the shared join counter, and copying a matching key range into main memory (see Section 3.3.3). Our join's runtime increases by 28% and 9% on the IBM AC922 and NVIDIA DGX H100, respectively.

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. [29]. For $\theta = 0 \rightarrow 1$, our join's execution time, unlike that of our CPU and GPU baselines [9, 96], decreases by 12% (IBM AC922) and 5% (NVIDIA DGX H100). Its *sort* phase becomes up to 9% and 2% slower as the number of MSB radix partitioning passes increases [50]. Its *join* phase, however, becomes up to 2.5× and 1.4× faster as increasingly larger key ranges in S are eliminated for some of the unique keys in R (see Section 3.3.3).

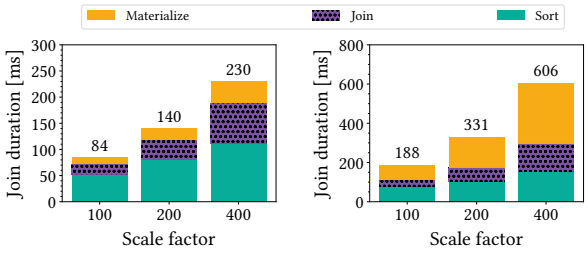
4.6 TPC-H Analysis

In this subsection, we validate our join's real-world applicability by means of the join-intensive TPC-H queries Q3 (see Figure 17a) and Q5 (see Figure 17b). We pre-filter the input relations at different scale factors $s \in \{100, 200, 400\}$ and materialize the resulting join tuples without applying the group-by and order-by clauses during query execution on the NVIDIA DGX H100 with $g = 8$ GPUs [110].

Our heterogeneous multi-GPU sort-merge join scales linearly with s for both **Q3** (involving two join operators) and **Q5** (involving five join operators). Relative to the total execution time, the sort, join, and materialize phases for **Q3** amount to 56%, 28%, and 16%, respectively. Since the intermediate join state for **Q5** is very large, our join spends 46% of its end-to-end runtime materializing tuples in parallel on the CPU during its execution. The remainder of the query execution is split between the multi-GPU-accelerated sort (32%) and join (22%) phases. Under the assumption that tuple materialization is a shared cost for all in-memory join algorithms, our join outperforms the fastest CPU and GPU baselines by 3.2× and 2.0× for **Q3** and 1.8× and 1.5× for **Q5**, respectively.

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 [96]. On the NVIDIA DGX H100, it is up to 8.7× (sort-merge) and 2.5× (hybrid-radix) faster than the GPU baselines and yields 15.2× and 5.5× shorter end-to-end runtimes than the parallel CPU sort-merge and radix-hash joins by Balkesen et al. [9].



(a) Query Q3 **(b) Query Q5**

Figure 17: TPC-H execution on the NVIDIA DGX H100

Out of its three algorithm phases (i.e., *sort*, *merge*, and *join*), the multi-GPU sort phase impacts our join's total execution time by far the most, with as much as 72% on the IBM AC922 and 78% on the NVIDIA DGX H100. The radix partitioning-based sort strategy is 12% 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 merge phase, which is required if an input relation exceeds the combined GPU memory capacity, causes a performance cliff as the CPU merge primitive saturates the main memory bandwidth of 170 GB/s on the IBM AC922 and 307 GB/s on the NVIDIA DGX H100 (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 a single modern HPC system — the IBM AC922 with NVLink 2.0-based CPU-GPU interconnects featuring cache-coherent access to main memory [49]. The hybrid join phase with overlapped copy and compute operations impacts our join's runtime the least, with as little as 28% on the IBM AC922 and 22% on the NVIDIA DGX H100. If both input relations are pre-sorted (e.g., due to prior group-by, order-by, or 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 [104]. In that case, our join is 14.4× (IBM AC922) and 13.7× (NVIDIA DGX H100) faster than the hybrid-radix join.

Scaling the number of GPUs yields *consistently* faster join phases but *occasionally* slower sort phases due to the interconnect topology. On the IBM AC922, the optimal GPU set is $\mathcal{G}_4^2$ although the system has four GPUs. It outperforms the single-GPU setup by 2.6×. On the NVIDIA DGX H100, the best GPU set $\mathcal{G}_8^8$ is 3.9× faster than the single-GPU setup. Contrary to the CPU and GPU baselines, which exhibit stable performance irrespective of data skew [9, 96], our join yields 12% shorter join durations. It outperforms the hybrid-radix join by 2.0× and the radix-hash join by 3.2× for real-world queries.

6 RELATED WORK

Over the past decades, researchers have thoroughly studied joins.

CPU Joins. 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, 91]. Blanas et al. find that hardware-oblivious hash joins with a shared and non-partitioned hash table outperform hardware-conscious hash joins [14]. Balkesen et al. draw the opposite conclusion after evaluating their parallel radix-hash join with bucket chaining [9]. Balkesen et al. further claim that for most workloads, hash joins are faster than sort-merge joins, although the relative performance gap narrows considerably for large input relations [8]. In contrast to these research efforts on parallel CPU-based joins, we focus on multi-GPU-accelerated joins.

Single-GPU Joins. 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 [97]. Sioulas et al. and Wu et al. describe GPU hash joins with bucket chaining and radix partitioning, respectively [105, 115]. Several experimental studies show that GPU-based joins outperform CPU-based joins [57, 95]. Prior publications rarely address the case when 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 [33, 65]. Guo and Chen utilize a CPU-assisted radix partitioning strategy and an in-core GPU sort-merge join. Lutz et al. rely on fast interconnects that provide GPUs with cache-coherent access to main memory. Unlike these single-GPU algorithms, our join utilizes all GPUs of modern multi-GPU platforms.

Multi-GPU Joins. Paul et al. propose a radix-partitioned multi-GPU hash join with a multi-hop routing strategy to minimize P2P data transfer congestion [88]. In contrast to our multi-GPU join, their join cannot handle large out-of-core data natively. It assumes that the input relations and the intermediate join state fit entirely into GPU memory. Besides, the growing availability of symmetric switch-based P2P interconnects (e.g., NVSwitch) over the past few years has rendered their multi-hop routing strategy obsolete [74]. Rui et al. design two out-of-core multi-GPU joins: a sort-merge join and a hybrid-radix join [96]. Unlike our P2P-enabled multi-GPU sort-merge join, both algorithms assume a shared-nothing architecture and cannot harness the P2P interconnects between the GPUs for efficient inter-GPU communication.

Distributed Joins. Distributed joins operate on multiple nodes across high-speed networks that often feature remote direct memory access (RDMA) [92–94]. Barthels et al. propose a distributed CPU radix-hash and sort-merge join utilizing one-sided RDMA [12]. Guo et al. study distributed joins in multi-node multi-GPU clusters based on GPUDirect RDMA [34]. Thostrup et al. propose a pipelined multi-GPU hash join that overlaps its data shuffling with its build and probe phases over GPUDirect RDMA-capable networks [109]. Gao and Sakharnykh present a hash join for multi-GPU clusters featuring a GPU-friendly data compression scheme [26]. Liu et al. describe two distributed CPU-GPU joins with hardware-conscious job scheduling [63]. While these joins are designed for multi-node clusters, our multi-GPU join targets single-node systems.

7 CONCLUSION

In this paper, we present a novel multi-GPU sort-merge join using high-bandwidth P2P interconnects for efficiently joining large input relations and show that it outperforms state-of-the-art CPU and GPU baselines. Beyond this paper, future work should extend our join with a NUMA-aware workload distribution strategy to mitigate the effects of low-bandwidth CPU-CPU interconnects.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (ref. 414984028 and ref. 556566056) and SAP.

REFERENCES

[1] A. Adinets and D. Merrill. 2022. *Onesweep: A Faster Least Significant Digit Radix Sort for GPUs.* NVIDIA. Retrieved June 28, 2025 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 June 28, 2025 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 June 28, 2025 from https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf

[5] AMD. 2024. *AMD Instinct MI325X Accelerator.* AMD. Retrieved June 28, 2025 from https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/product-briefs/instinct-mi325x-datasheet.pdf

[6] M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy. 2022. *NVIDIA Hopper Architecture In-Depth.* NVIDIA. Retrieved June 28, 2025 from https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] J. Demouth. 2014. *CUDA Pro Tip: Minimize the Tail Effect.* NVIDIA. Retrieved June 28, 2025 from https://developer.nvidia.com/blog/cuda-pro-tip-minimize-the-tail-effect/

[21] L. Durant, O. Giroux, M. Harris, and N. Stam. 2017. *Inside Volta: The World's Most Advanced Data Center GPU.* NVIDIA. Retrieved June 28, 2025 from https://developer.nvidia.com/blog/inside-volta/

[22] 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

[23] FSF. 2023. *The GNU C++ Library Manual: Parallel Mode.* FSF. Retrieved June 28, 2025 from https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++-manual.pdf.gz

[24] FSF. 2023. *The GNU C++ Library Reference Manual.* FSF. Retrieved June 28, 2025 from https://gcc.gnu.org/onlinedocs/gcc-13.2.0/libstdc++-api.pdf.gz

[25] 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

[26] H. Gao and N. Sakharnykh. 2021. *Scaling Joins to a Thousand GPUs.* NVIDIA. Retrieved June 28, 2025 from https://adms-conf.org/2021-camera-ready/gao_adms21.pdf

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] M. Harris. 2012. *How to Optimize Data Transfers in CUDA C/C++.* NVIDIA. Retrieved June 28, 2025 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 June 28, 2025 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 June 28, 2025 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 June 28, 2025 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 June 28, 2025 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 June 28, 2025 from https://developer.nvidia.com/blog/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/

[47] M. Harris. 2015. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*. NVIDIA. Retrieved June 28, 2025 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 AC922: Technical Overview and Introduction*. IBM. Retrieved June 28, 2025 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 June 28, 2025 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 June 28, 2025 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] 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

[61] 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

[62] 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

[63] H. Liu, B. Tang, J. Zhang, Y. Deng, X. Yan, X. Zheng, Q. Shen, D. Zeng, Z. Mao, C. Zhang, Z. You, Z. Wang, R. Jiang, F. Wang, M.-L. Yiu, H. Li, M. Han, Q. Li, and Z. Luo. 2022. GHive: Accelerating Analytical Query Processing in Apache Hive via CPU-GPU Heterogeneous Computing. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC '22)*. ACM, New York, NY, USA, 158–172. https://doi.org/10.1145/3542929.3563503

[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 June 28, 2025 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] NVIDIA. 2014. *NVIDIA NVLink: High-Speed Interconnect Application Performance*. NVIDIA. Retrieved June 28, 2025 from https://info.nvidianews.com/rs/nvidia/images/NVIDIA%20NVLink%20High-Speed%20Interconnect%20Application%20Performance%20Brief.pdf

[73] NVIDIA. 2017. *NVIDIA V100 Tensor Core GPU Architecture*. NVIDIA. Retrieved June 28, 2025 from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[74] NVIDIA. 2018. *NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch*. NVIDIA. Retrieved June 28, 2025 from https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf

[75] NVIDIA. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA. Retrieved June 28, 2025 from https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[76] NVIDIA. 2020. *NVIDIA V100 Tensor Core GPU*. NVIDIA. Retrieved June 28, 2025 from https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

[77] NVIDIA. 2021. *mgpu: Patterns and Behaviors for GPU Computing*. NVIDIA. Retrieved June 28, 2025 from https://github.com/moderngpu/moderngpu

[78] NVIDIA. 2021. *NVIDIA A100 Tensor Core GPU*. NVIDIA. Retrieved June 28, 2025 from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

[79] NVIDIA. 2023. *cub: Cooperative Primitives for CUDA C++*. NVIDIA. Retrieved June 28, 2025 from https://github.com/NVIDIA/cub

[80] NVIDIA. 2023. *CUDA C++ Best Practices Guide*. NVIDIA. Retrieved June 28, 2025 from https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[81] NVIDIA. 2023. *NVIDIA DGX H100 System*. NVIDIA. Retrieved June 28, 2025 from https://docs.nvidia.com/dgx/dgxh100-user-guide/dgxh100-user-guide.pdf

[82] NVIDIA. 2023. *NVIDIA H100 Tensor Core GPU Architecture*. NVIDIA. Retrieved June 28, 2025 from https://nvdam.widen.net/content/hj0uek1pxq/original/nvidia-h100-tensor-core-hopper-whitepaper.pdf

[83] NVIDIA. 2023. *thrust: C++ Parallel Algorithms Library*. NVIDIA. Retrieved June 28, 2025 from https://github.com/NVIDIA/thrust

[84] NVIDIA. 2024. *CUDA C++ Programming Guide*. NVIDIA. Retrieved June 28, 2025 from https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[85] NVIDIA. 2024. *NVIDIA H100 Tensor Core GPU*. NVIDIA. Retrieved June 28, 2025 from https://nvdam.widen.net/content/vuzumiozpb/original/nvidia-h100-datasheet-2430615.pdf

[86] 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

[87] 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

[88] 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

[89] 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

[90] 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

[91] 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

[92] 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

[93] 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

[94] 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

[95] 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

[96] 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

[97] 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

[98] 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

[99] 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

[100] 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

[101] 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

[102] 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

[103] 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

[104] 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

[105] 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

[106] 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

[107] 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

[108] 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

[109] 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

[110] TPC. 2014. *TPC Benchmark H (Decision Support)*. TPC. Retrieved June 28, 2025 from https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf

[111] 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

[112] 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

[113] 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

[114] 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

[115] B. Wu, D. Koutsoukos, and G. Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27. https://doi.org/10.1145/3709689

[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