Evaluating Multi-GPU Sorting with Modern Interconnects

Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, Tilmann Rabl
Hasso Plattner Institute, University of Potsdam
{tobias.maltenberger,ivan.ilic}@student.hpi.de,{ilin.tolovski,tilmann.rabl}@hpi.de

ABSTRACT

In recent years, GPUs have become a mainstream accelerator for database operations such as sorting. Most of the published GPU-based sorting algorithms are single-GPU approaches. Consequently, they neither harness the full computational power nor exploit the high-bandwidth P2P interconnects of modern multi-GPU platforms. In particular, the latest NVLink 2.0 and NVLink 3.0-based NVSwitch interconnects promise unparalleled multi-GPU acceleration. Regarding multi-GPU sorting, there are two types of algorithms: GPU-only approaches, utilizing P2P interconnects, and heterogeneous strategies that employ the CPU and the GPUs. So far, both types have been evaluated at a time when PCIe 3.0 was state-of-the-art. In this paper, we conduct an extensive analysis of serial, parallel, and bidirectional data transfer rates to, from, and between multiple GPUs on systems with PCIe 3.0, PCIe 4.0, NVLink 2.0, and NVLink 3.0-based NVSwitch interconnects. We measure up to 35.3× higher parallel P2P copy throughput with NVLink 3.0-powered NVSwitch over PCIe 3.0 interconnects. To study multi-GPU sorting on today's hardware, we implement a P2P-based (P2P sort) and a heterogeneous (HET sort) multi-GPU sorting algorithm and evaluate them on three modern systems. We observe speedups over state-of-the-art parallel CPU-based radix sort of up to 14× for P2P sort and 9× for HET sort. On systems with high-speed P2P interconnects, we demonstrate that P2P sort outperforms HET sort by up to 1.65×. Finally, we show that overlapping GPU copy and compute operations to mitigate the transfer bottleneck does not yield performance improvements on modern multi-GPU platforms.

ACM Reference Format:
Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22), 2022, Philadelphia, PA, USA.*. ACM, New York, NY, USA, 15 pages. https://doi.org/

1 INTRODUCTION

Unprecedented amounts of data make it increasingly challenging to keep the response times of database systems low [12, 23, 30, 32, 73]. Therefore, researchers and engineers continuously adapt the systems to modern hardware technology [2, 8, 9, 15, 25, 28, 75]. Steadily decreasing memory costs have led to the rise of in-memory database systems [22, 34, 68]. Their performance bottlenecks are often not I/O operations anymore but memory bandwidth and compute power [71]. Recent hardware trends in concurrent data processing (e.g., multi-core architectures and massively-parallel accelerators) offer unparalleled performance. Featuring thousands of cores, graphics processing units (GPUs) provide remarkably high instruction throughput and memory bandwidth [4, 45, 50]. Especially in combination with high-bandwidth interconnects that promise to mitigate the data transfer bottleneck, GPUs have become suitable for accelerating essential database operations [38].

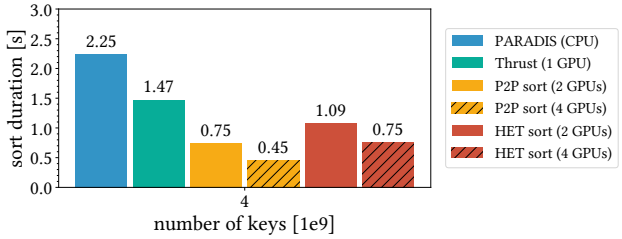


Figure 1: Sorting 16 GB on the DGX A100: CPU vs. GPUs

One such operation is sorting, with applications ranging from index creation and duplicate detection to merge-joins [20]. Numerous GPU-based sorting algorithms have been proposed [10, 33, 40, 43, 57, 59, 64, 65, 71]. However, the vast majority of these algorithms are single-GPU approaches that leave the potential performance gain of sorting across multiple GPUs entirely untapped.

Multi-GPU sorting algorithms employ either a GPU-only approach, utilizing the high-speed peer-to-peer (P2P) interconnects between the GPUs, or a heterogeneous strategy that exploits the computational power of both CPU and GPUs [19, 58, 72, 76]. Tanasic et al. propose, to the best of our knowledge, the only P2P-based multi-GPU sorting algorithm that performs both sorting and merging of the data on the GPUs [72]. Gowanlock et al. describe the latest heterogeneous multi-GPU sorting approach in which the CPU merges chunks of data that have been sorted on the GPUs [19]. However, both authors conducted their experiments when PCIe 3.0 was state-of-the-art in interconnect technology and multi-GPU systems were still emerging. Since then, high-speed interconnects such as NVLink 2.0 and NVLink 3.0-powered NVSwitch, which greatly outperform PCIe 3.0, have become available in multi-GPU systems such as the IBM Power System AC922 or the NVIDIA DGX A100 [26, 38, 52]. Besides, Gowanlock et al. show that their heterogeneous algorithm's final CPU merge phase becomes a bottleneck, which high-bandwidth interconnects will further exacerbate. Thus, it is necessary to re-evaluate P2P-based and heterogeneous sorting on modern multi-GPU platforms.

In this paper, we conduct an in-depth analysis of PCIe 3.0, PCIe 4.0, NVLink 2.0, and NVLink 3.0-based NVSwitch interconnects on three modern multi-GPU platforms. Our interconnect analysis covers serial, parallel, and bidirectional data transfers for multiple GPUs. Based on an evaluation of state-of-the-art single-GPU sorting and merging primitives, we implement a P2P-based (P2P sort) and

SIGMOD '22, June 12-17, 2022, Philadelphia, PA, USA

https://doi.org/

a heterogeneous (HET sort) multi-GPU sorting algorithm. We evaluate various CPU/GPU compute and data transfer optimizations on today's accelerator platforms and incorporate those approaches that boost the end-to-end performance into our implementations. We study P2P and HET sort from an algorithmic point of view and evaluate their performance using our fully automated open-source benchmark suite. We compare the two algorithms for increasing numbers of GPUs and analyze HET sort's scalability for large data exceeding the combined GPU memory.

We show that NVLink 2.0 accelerates CPU-GPU transfers up to 6.0× over PCIe 3.0. NVLink 3.0-based NVSwitch outperforms PCIe 3.0 up to 35.3× and NVLink 2.0 up to 5.5× for concurrent P2P data transfers. Regarding multi-GPU sorting, we measure speedups over PARADIS [13], a state-of-the-art parallel CPU radix sort, of up to 14× for P2P sort and 9× for HET sort. Besides, we observe that P2P sort is 1.65× faster than HET sort on the DGX A100 with its high-speed NVLink 3.0 P2P interconnects (see Figure 1). Contrary to prior research findings [58, 71, 76], we show that overlapping GPU copy and compute operations does not improve the sorting performance for large data on modern multi-GPU systems. We further find that eagerly merging sorted chunks does not reduce HET sort's final CPU merge workload, decreasing its performance. Generally, we identify two limiting factors for scaling to increasing numbers of GPUs on modern platforms: low-bandwidth CPU-CPU interconnects and shared PCIe bandwidth effects for CPU-GPU transfers. On the IBM AC922, where the CPU and the GPUs are exclusively connected via NVLink 2.0, we observe the shortest end-to-end sort durations. Therefore, a platform's interconnect topology heavily impacts the performance of multi-GPU-accelerated sorting.

With this paper, we make the following contributions.

(1) We conduct an extensive analysis of modern CPU-GPU and P2P interconnects, covering serial, parallel, and bidirectional data transfers for multiple GPUs and are the first to evaluate NVLink 3.0-powered NVSwitch (Section 4).
(2) We evaluate state-of-the-art sorting and merging primitives for both CPU and GPU (Section 5, Section 6).
(3) We publish highly optimized P2P-based and heterogeneous multi-GPU sorting implementations together with an automated benchmark suite (Section 5.2, Section 5.3).
(4) We analyze the two algorithms on three modern multi-GPU platforms to evaluate each system's and algorithm's suitability for GPU-accelerated sorting (Section 6.1).
(5) We demonstrate that overlapping GPU copy/compute operations does not mitigate the transfer bottleneck and eagerly merging does not reduce the CPU-GPU load imbalance when sorting heterogeneously on modern systems (Section 6.2).

2 BACKGROUND

In this section, we explain essential background information on GPUs and multi-GPU interconnect technology.

GPU Accelerators. Graphics processing units (GPUs) provide highly parallel compute capabilities. For example, the NVIDIA Tesla V100 reaches 7.8 and 15.7 TFLOPS while the NVIDIA A100 achieves 9.7 and 19.5 TFLOPS for double-precision and single-precision floating-point numbers, respectively. GPUs also provide high-bandwidth memory subsystems. However, with up to 80 GB, their memory capacity is very limited compared to that of main memory [45, 50]. In modern high performance computing (HPC) systems, multiple GPUs are attached to the CPU to increase the compute power and memory. Distinct GPUs are connected to the system's host-side memory controller and to each other via an interconnect bus. Traditionally, PCIe has been the standard interconnect. The theoretical bandwidth of 16 dual simplex PCIe 3.0 lanes is 16 GB/s per direction. For GPU-accelerated database operations, PCIe 3.0 has been the bottleneck since its bandwidth rate is much lower than that of main memory [19, 31, 38, 63, 70, 71].

GPU Interconnects. In recent years, hardware vendors try to mitigate the transfer bottleneck by providing higher bandwidth rates. The latest systems connect the CPU to its GPUs with PCIe 4.0 at a 32 GB/s bandwidth. NVIDIA recently introduced a high-bandwidth interconnect technology, called NVLink. NVLink 2.0 increases bandwidth up to 25 GB/s per link per direction and is primarily designed as a GPU-GPU interconnect, enabling faster P2P communication. One NVLink 2.0-enabled GPU supports six links for a theoretical peak bandwidth of 150 GB/s per direction. The NVLink 3.0-enabled NVIDIA Ampere architecture supports 12 links per GPU for up to 300 GB/s per direction. NVLink interconnects can also serve as CPU-GPU interconnects and even provide cache coherence across the CPU cache hierarchy and the GPU global memory cache [38, 48]. Other vendors advance interconnect technology too. AMD Radeon Instinct GPUs include the high-speed AMD Infinity Fabric interconnect [4]. Intel CXL is an industry-open standard for CPU-to-accelerator interconnects based on PCIe 5.0 [67].

On PCIe-based multi-GPU systems without P2P interconnects, the PCIe interconnects form a balanced tree structure with the CPU as its root. High-bandwidth P2P interconnects (e.g., NVLink) allow for direct P2P data transfers, eliminating the need for interconnect hops. On systems with such interconnects, data transfers less likely compete for shared bandwidth. Modern systems often incorporate heterogeneous interconnects. Thus, the interconnect topology critically influences performance [35]. Most large-scale many-core CPU architectures integrate processor-local main memory. DRAM is attached to a CPU's on-chip memory controller to form a NUMA (non-uniform memory access) node. Multiple NUMA nodes are connected through CPU interconnects so that one CPU can access a remote CPU's memory (see Table 1). CPU interconnects typically provide less throughput and higher latency than CPU memory controllers [26, 39]. Thus, on NUMA platforms, the memory access time depends on the locality of the memory region relative to the CPU issuing the access. NUMA systems typically attach an equal number of GPUs to each node. If no direct connection between the GPUs of different nodes exists, P2P transfers traverse the CPU interconnect. Therefore, some systems connect all GPUs with each other directly. For example, NVIDIA's NVSwitch uses NVLink-based switch chips to achieve non-blocking all-to-all P2P communication [35, 46].

3 RELATED WORK

In recent years, researchers have evaluated modern GPU interconnects. Pearson et al. evaluate single-GPU data transfer primitives for PCIe 3.0, NVLink 1.0, and 2.0 [56]. Li et al. benchmark a wide range of interconnects on multi-GPU systems, and suggest harnessing inter-GPU communication [35, 36]. In contrast, we evaluate the

acceleration of sorting as a database operation when scaling to multiple GPUs and design our data transfer benchmarks accordingly. Besides, we are the first to analyze NVLink 3.0-based NVSwitch.

Within the last years, research that evaluates database-relevant GPU-acceleration with NVLink has emerged. Raza et al. evaluate the performance of GPU interconnects in the database management context and propose a hybrid materialization approach consisting of lazy and eager data transfers to accelerate OLAP workloads [62]. The authors evaluate using two GPUs only, one local GPU per CPU node. Lutz et al. benchmark performance characteristics of NVLink 2.0 and evaluate hash join workloads using various data placement and transfer strategies for large data [38]. They perform their experiments with one GPU only and do not analyze P2P transfers. In contrast, we evaluate data processing algorithms for multiple GPUs, one of which utilizes P2P interconnects.

Rui et al. evaluate three multi-GPU join algorithms for large data on platforms with PCIe and NVLink interconnects [63]. They analyse the data transfers to be the bottleneck, observing a speedup of up to 2.8× with eight GPUs over one. Paul et al. propose a partitioned hash join for multiple GPUs. They evaluate it to outperform prior distributed join algorithms on the DGX-1 using an adaptive multi-hop data distribution strategy [55]. In comparison, this paper evaluates multi-GPU sorting, comparing two algorithms on three modern systems, while analyzing the latest interconnects in-depth.

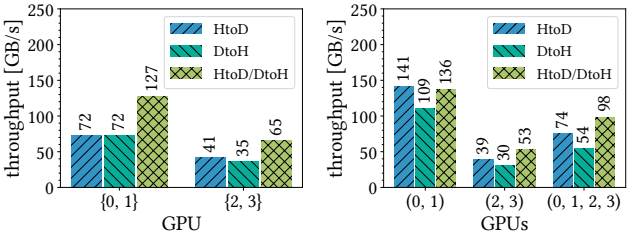
All published multi-GPU sorting algorithms are sort-merge approaches. Tanasic et al. propose the only multi-GPU sort algorithm that performs all computations on the GPUs using P2P memory swaps [72]. Other multi-GPU sort algorithms are heterogeneous approaches for large data: Gowanlock et al. extend the GPU-CPU sort by Stehle et al. to multiple GPUs and observe the CPU merge to be a major bottleneck, next to the CPU-GPU transfers [19, 71]. Ye et al. adapt a deterministic parallel sample sort to multiple GPUs and evaluate it on two GPUs [76]. Peters et al. describe an algorithm that uses k-way merging [58]. All of these algorithms were evaluated on systems with PCIe 3.0 only. We are the first to evaluate multi-GPU sorting on high-bandwidth interconnects and to compare two multi-GPU sorting algorithms to each other.

4 INTERCONNECT ANALYSIS

In this section, we analyze the data transfer performance of three multi-GPU platforms with different interconnect topologies.

4.1 Hardware Systems Overview

Table 1 shows each system's hardware specification and their topology with the bandwidth rates per direction. The IBM Power System AC922 comes with NVLink 2.0 for CPU-GPU and P2P connections. The DELTA System D22x M4 PS is equipped with PCIe 3.0 as the CPU-GPU interconnect and NVLink 2.0 for P2P transfers. Both systems have four NVIDIA Tesla V100 GPUs and only differ in their interconnects and host side. The NVIDIA DGX A100 has eight NVIDIA A100 GPUs that are interconnected with NVLink 3.0-based NVSwitch for fast all-to-all P2P transfers and via PCIe 4.0 to the CPU. All systems have two NUMA nodes but differ in their CPU interconnects. The IBM AC922 has a X-Bus interconnect, the DELTA D22x comes with Intel's Ultra Path Interconnect (UPI), and the DGX A100 uses AMD Infinity Fabric. The IBM AC922 is the only system



(a) Serial **(b) Parallel**

Figure 2: CPU-GPU data transfers on the IBM AC922.

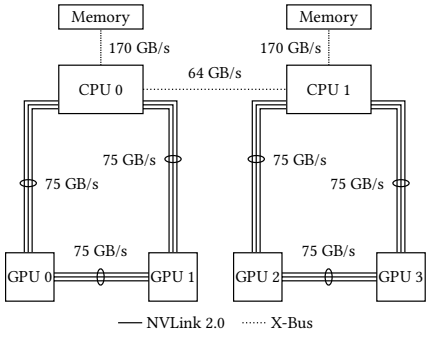
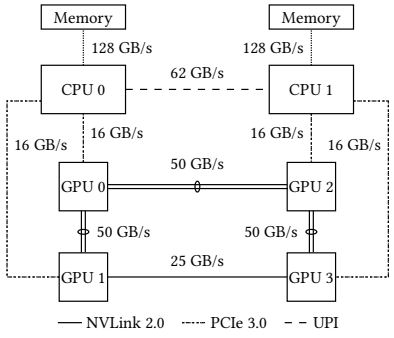
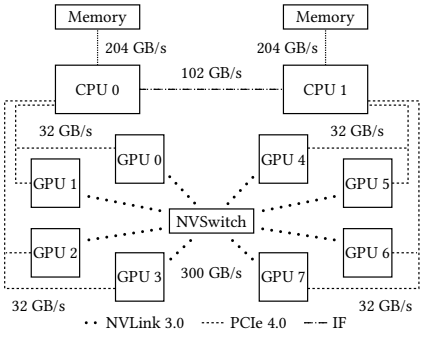
that includes NVLink 2.0 as the CPU-GPU interconnect, enabling efficient GPU-accelerated query processing [38]. The DGX A100 is NVIDIA's most competitive GPU-accelerator platform.

4.2 CPU-GPU Data Transfers

We compare the throughput of CPU-GPU transfers when performing a serial copy and copying to or from multiple GPUs in parallel. Additionally, we evaluate the bidirectional copy throughput in the serial and parallel scenarios. For that, we repeat concurrently executed HtoD and DtoH copies and calculate the average duration of one bidirectional copy. Only after both copy streams finish, bidirectional data transfers are considered to be completed. They are bound by the slower copy stream. For serial transfers, we copy 4 GB from host to device memory (HtoD) and back (DtoH). For the parallel experiment, a host memory buffer of 4 GB is allocated for each GPU, allowing the data transfers to occur concurrently. This results in different interconnect paths being used either shared or exclusively. When copying bidirectionally, the transfers operate on distinct 4 GB buffer per direction. For every experiment, the transfers start from CPU node 0 where the host buffers are allocated in pinned memory. While pageable memory can be moved by the operating system, pinned memory is page-locked. The advantage of pinned memory is that the CUDA driver starts copying without any intermediate transfers [24]. More importantly, pinned memory copies utilize substantially higher transfer rates because the transfer is offloaded to the GPU's copy engine that accesses the host memory with direct memory access (DMA) [38, 56]. Thus, we perform all CPU-GPU copies on pinned memory.

IBM Power System AC922: In Figure 2a, we observe the system's interconnect topology reflected in the results. For GPUs 0 and 1, which are local to CPU node 0, the throughput almost reaches the theoretical peak of 75 GB/s with 72 GB/s. When the copy stream traverses the CPU interconnect to GPUs 2 or 3, we measure a performance drop to 41 GB/s and 35 GB/s for HtoD and DtoH transfers respectively, which is 64% and 55% of the theoretical 64 GB/s X-Bus bandwidth. Our results confirm the measurements by Pearson et al. as we obtain the same numbers [56]. Also, we observe a slight performance overhead for bidirectional copies as the local GPUs achieve a throughput of 127 GB/s, 88% of 2× the unidirectional throughput of 72 GB/s. The bidirectional throughput of remote GPUs is again bound by the X-Bus bandwidth. While a small bidirectional copy overhead is to be expected, the throughput X-Bus achieves is surprisingly low. We analyze the X-Bus throughput for memory copies between NUMA nodes so that the CUDA driver is not involved. We find that the measured bandwidth rate does not

Table 1: Topology and specification of the evaluated hardware platforms

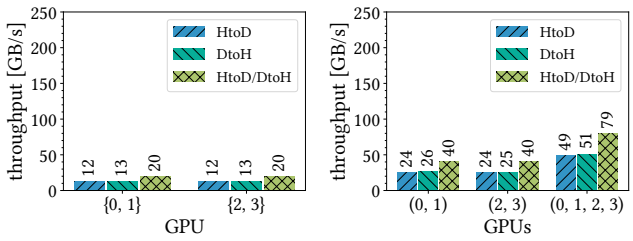
(a) IBM Power System AC922	(b) DELTA System D22x M4 PS	(c) NVIDIA DGX A100
		
2x IBM POWER9 (16 × 2.7 GHz)	2x Intel Xeon Gold 6148 (20 × 2.4 GHz)	2x AMD EPYC 7742 (64 × 2.25 GHz)
4x NVIDIA Tesla V100 SXM2 32 GB	4x NVIDIA Tesla V100 SXM2 32 GB	8x NVIDIA A100 SXM4 40 GB
2x 256 GB DDR4	2x 755 GB DDR4	2x 512 GB DDR4
RHEL 7.6, ppc64le	Ubuntu 18.04, x86_64	Ubuntu 20.04, x86_64
CUDA 11.2, GCC 10.2.1	CUDA 11.2, GCC 10.3.0	CUDA 11.0, GCC 9.3.0

reach the theoretical maximum. The X-Bus interconnect is based on an interface consisting of two X-Links [14]. One X-Link carries the coherency traffic and the payload; the other one is a data-only link [27]. We suspect that the system gets into a state where many retries occur since the CPU tries to drive considerably more bandwidth than the X-Link can sustain. Therefore, retry attempts instead of data transfers consume the X-Bus bandwidth.

When copying in parallel (Figure 2b), we observe near peak performance for HtoD transfers to GPUs 0 and 1. Since we issue two parallel copies, the theoretical aggregated throughput is twice the single local copy throughput (=150 GB/s). However, the bandwidth is not fully utilized for DtoH copies as the throughput reaches only 109 GB/s. Flushing the CPU caches prior to the transfers prevents cache line eviction of dirty data and can accelerate DtoH copies [56]. However, in our case, it does not increase the unexpectedly low throughput. When the two local GPUs $(0, 1)$ each perform bidirectional copies in parallel on the same CPU node (for a total of four copy streams), we measure a throughput of 136 GB/s. This is only 25% more than the throughput the two GPUs achieve for unidirectional DtoH transfers. The throughput is reasonable, though, since that many copies saturate and compete for the main memory bandwidth. When copying to the remote GPUs 2 and 3 in parallel, we observe the throughput of the three transfer types to stay within 82% of their single remote-GPU-copy counterpart as the two copy streams now share the comparatively low X-Bus bandwidth. The throughput for CPU-local GPUs is up to 3.6× higher than on remote GPUs $(2, 3)$. When copying on all four GPUs, we measure the data transfers to reach twice the throughput of the parallel remote copies on GPUs $(2, 3)$ as the total throughput is bound by the slowest copy operation. Overall, the parallel copy on four GPUs only uses 26% (HtoD) and 19% (DtoH) of the maximum throughput achievable if all 4 GPUs were connected with three NVLinks 2.0 locally to one CPU node and main memory bandwidth was high enough.

DELTA System D22x M4 PS: The bandwidth of PCIe 3.0 interconnects is significantly lower than that of UPI. Hence, in Figure 3a, we observe that there is no performance difference between local and remote GPU copies. We measure 12-13 GB/s, which is close to the theoretical peak of 16 GB/s. Bidirectional data transfers reach almost double the unidirectional throughput (within 77-83%). In Figure 3b, we again observe the absence of NUMA effects. Moreover, parallel copy operations on multiple GPUs achieve the desired scaling: When copying on four GPUs, the HtoD, DtoH, and bidirectional transfers reach 4× higher throughput. This is the case because the CPU interconnect bandwidth is high enough to support multiple, shared, PCIe 3.0-bound copy streams and because the system provides an exclusive PCIe switch for each GPU. Still, the parallel copy throughput on all four GPUs is higher on the IBM AC922 despite its strong NUMA-related performance decreases.

NVIDIA DGX A100: In Figure 4, we measure 24-25 GB/s for serial HtoD and DtoH data transfers for local and remote GPUs, which is within the expected range of 75-78% of the theoretical 32GB/s PCIe 4.0 bandwidth. Thus, PCIe 4.0 doubles the measured bandwidth of PCIe 3.0. This is the case for any GPU taken out of the sets $\{0, 1, 2, 3\}$ and $\{4, 5, 6, 7\}$. However, we observe a slight throughput decrease by 18% down to 32 GB/s for bidirectional copies on remote GPUs compared to 39 GB/s on local ones. The



(a) Serial **(b) Parallel**

Figure 3: CPU-GPU data transfers on the DELTA D22x.

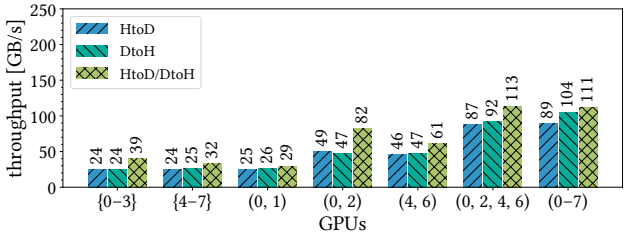
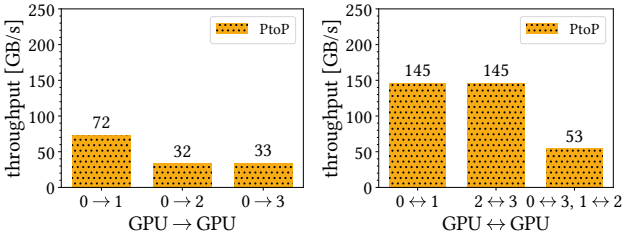


Figure 4: CPU-GPU data transfers on the DGX A100



(a) Serial **(b) Parallel**

Figure 5: P2P data transfers on the IBM AC922

parallel copy throughput results (GPU ids in tuple notation) reveal that some data transfers share bandwidth when executed in parallel. Compared to a single GPU, the throughput does not double when copying with GPUs $(0, 1)$. It does increase 2× for GPU pair $(0, 2)$ because the GPU pairs $(0, 1)$, $(2, 3)$, $(4, 5)$, and $(6, 7)$ each share a PCIe switch. Because one PCIe switch connects one pair of GPUs with CPU memory via only one PCIe 4.0 interconnect instance of 16 lanes, these GPU pairs share the respective bandwidth of 32 GB/s. This also explains why the throughput does not increase from GPUs $(0, 2, 4, 6)$ up to eight. For the remote GPU pair $(4, 6)$, we observe that HtoD and DtoH throughput is almost the same as for the local GPU pair because of the CPU-interconnect's (AMD Infinity Fabric) high enough bandwidth. However, we again observe a bidirectional copy overhead on remote GPUs, as GPU pair $(4, 6)$ achieves 61 GB/s (74%) instead of 82 GB/s – a performance discrepancy that needs to be further investigated. The bidirectional throughput for GPUs $(0, 2, 4, 6)$ and for all eight is bound by the remote performance as we measure roughly twice the throughput of GPU pair $(4, 6)$.

Conclusion. Our results show that even though high-speed NVLink 2.0 significantly accelerates data transfers over PCIe 3.0 and 4.0 on CPU-local GPUs, concurrent multi-GPU usage introduces bottlenecks on two of our platforms. We observe low CPU-interconnect bandwidth on the IBM AC922 as well as an insufficient number of PCIe switches (and the resulting shared PCIe bandwidth) on the DGX A100 to limit the scalability of the systems' multi-GPU copy throughput. We further observe that multiple GPUs do not always concurrently harness full bidirectional bandwidth.

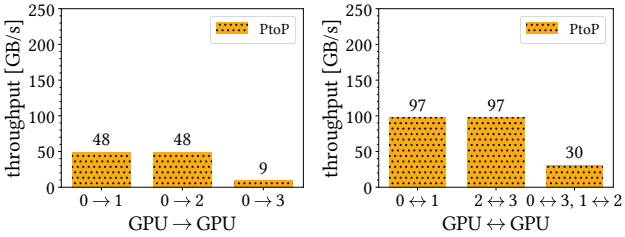
4.3 P2P Data Transfers

We measure the P2P transfer throughput for serial and parallel copy streams. In the serial scenario, 4 GB of data are copied from one GPU to another. In the parallel scenarios, multiple GPUs copy concurrently, operating on distinct blocks of 4 GB each. We evaluate parallel copies for different sets of g GPUs with an increasing $g \in \{2, 4, ..., \hat{g}\}$, $\hat{g}$ being the system's maximum number of GPUs. For a set of g GPUs $(GPU_0, GPU_1, ..., GPU_{g-1})$, we execute the following concurrent copies: $GPU_0 \leftrightarrow GPU_{g-1}$, $GPU_1 \leftrightarrow GPU_{g-2}$, ..., $GPU_{(\frac{g}{2})-1} \leftrightarrow GPU_{\frac{g}{2}}$. As indicated by the arrows, we measure bidirectional P2P throughput for the parallel scenarios.

IBM Power System AC922: The IBM AC922 connects GPUs 0 and 1 via three NVLinks with a throughput of 72 GB/s (Figure 5a). GPUs 2 and 3 are only reachable via X-Bus, resulting in 46% of the direct P2P throughput. Achieving high performance is again strongly dependent on the GPU locality. Figure 5b shows that P2P transfers performed between two directly connected GPUs $(0, 1)$ or $(2, 3)$ reach optimal performance. In the third parallel scenario, all four copy streams are performed between *distant* GPUs, $(0, 3)$ and $(1, 2)$, sharing the X-Bus in both directions. This decreases the throughput to 53 GB/s – 18% of what could be achieved if all four GPUs were connected via three NVLinks in a mesh.

DELTA System D22x M4 PS: The DELTA D22x comes with at most two NVLinks to connect GPUs between each other but does connect more pairs of GPUs than the IBM AC922. In Figure 6a, we observe the throughput of the P2P copies from GPU 0 to GPUs 1 and 2 to reach 48 GB/s of the theoretical 50 GB/s. The performance decrease for distant (host-side traversing) P2P transfers is much steeper. This is the case for the path from GPU 0 to GPU 3, where throughput drops by 81.25% compared to direct P2P copies. While the CPU interconnect bandwidth (UPI) is high enough (62 GB/s), the low PCIe 3.0 bandwidth slows down the data transfers twice: On the way from GPU 0 to CPU node 0 and from the remote CPU node to GPU 3. Compared with the IBM AC922, three NVLinks achieve 8× higher transfer performance (72 GB/s) than PCIe 3.0 (9 GB/s). For parallel transfers on two GPUs, we see optimal throughput in Figure 6b. Since GPU pairs $(0, 3)$ and $(1, 2)$ are not directly interconnected, throughput drops for four GPUs.

NVIDIA DGX A100 The DGX A100 fully connects all GPUs between each other with NVLink 3.0-based NVSwitch. As a result, any data transfer from one GPU i to any other GPU j utilizes its own direct, high-bandwidth P2P connection powered by 12 × NVLinks per direction, for a total theoretical bandwidth of 300GB/s. We measure 279 GB/s for serial P2P transfers, as seen in Figure 7. This constitutes 31× higher throughput compared to the host-side traversing P2P copies via PCIe 3.0 on the DELTA D22x (9 GB/s). The serial P2P copy throughput on the DGX is also 3.9× higher than it is on the IBM AC922 (72 GB/s). The parallel P2P transfers reveal that NVSwitch is explicitly designed to support concurrent all-to-all copies, as the throughput scales well for increasing numbers of



(a) Serial **(b) Parallel**

Figure 6: P2P data transfers on the DELTA D22x

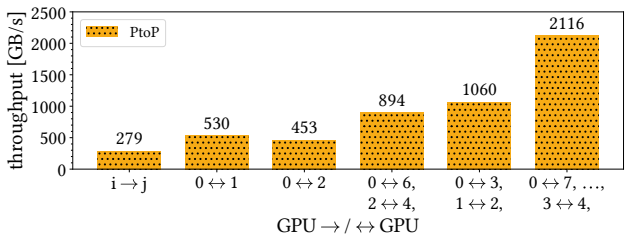


Figure 7: P2P data transfers on the DGX A100

GPUs. We measure a total parallel P2P copy throughput of 894-1060 GB/s for four GPUs, which is 17-20× higher than four GPUs reach on the IBM AC922 and 2116 GB/s between all eight GPUs.

Conclusion. We conclude that P2P copies are by far the fastest on the DGX A100 due to NVLink 3.0-powered NVSwitch. Compared to the other two systems, the DGX A100 achieves up to 5.5× higher throughput on two GPUs and up to 35.3× on four GPUs. On the IBM AC922 and the DELTA D22x, NVLink 2.0 provides high P2P throughput for select GPU pairs. However, both systems do not connect all four system-wide GPUs between each other directly. Thus, concurrent P2P transfers on all GPUs require traversing the poorly interconnected host-side.

5 SORTING ALGORITHMS

In this section, we present our two multi-GPU sorting algorithm implementations: a P2P-based approach building upon Tanasic et al. and a heterogeneous GPU-CPU strategy. First, we study state-of-the-art single-GPU sorting and merging primitives to utilize in our implementations. For both multi-GPU sorting approaches, we re-evaluate if published performance optimizations still hold on today's accelerator platforms with high-bandwidth interconnects. Moreover, we introduce our optimizations and algorithmic extensions. We compare the two algorithms in terms of data transfer volume and computational complexity of their merge phase.

The first phase of the multi-GPU sorting process is the same for the two algorithms. The input data is partitioned into equally sized chunks that fit into GPU memory. Then, g GPUs simultaneously sort their chunk locally. The algorithms differ in the second phase (i.e., how the sorted chunks are merged into the fully sorted output data). Tanasic et al. propose a P2P-based merge strategy on the GPUs while the heterogeneous approach utilizes the CPU for merging.

5.1 Single-GPU Sorting

Since both multi-GPU sorting approaches employ a single-GPU algorithm to sort chunks locally, we study the performance of state-of-the-art single-GPU sorting algorithms. We use the fastest primitive in both of our implementations. Most single-GPU sorting algorithms are either merge-based or radix sort algorithms [43, 49, 65, 71]. Merge sort has the worse time complexity with $O(n*\log(n))$. While radix sort has a computational complexity of $O(n)$, its main bottleneck is the high memory bandwidth demand. At their time of evaluation, Satish et al. find that radix sort performs slightly better than merge sort, claiming that merge sort is better suited for future architectures with increased SIMD widths [65]. However, the hardware developments and algorithmic improvements of the last decade helped to establish radix sort as the fastest single-GPU sorting algorithm. The most recent work in this field is the MSB radix sort by Stehle et al. which lifts the restriction of having to respect the order of preceding sorting passes [71]. This enabled their implementation to consider more bits per sorting pass, effectively reducing the memory bandwidth demand. Moreover, the GPUs' memory bandwidth has increased significantly [45, 50].

Table 2: NVIDIA A100 GPU sorting 1B integers (4 GB)

Algorithm	Type	Duration
Thrust	Radix Sort	36 ms
CUB	Radix Sort	36 ms
Stehle	Radix Sort	57 ms
MGPU	Merge Sort	200 ms

We re-evaluate state-of-the-art single-GPU sorting primitives on modern hardware and present our results in Table 2. Contrary to previous research [71], we observe that NVIDIA's two CUDA libraries, CUB and Thrust [51, 54], achieve identical performance as they currently use the same underlying LSB radix sort. Additionally, we evaluate the openly available Modern GPU merge sort [49] as well as the original implementation of the radix sort by Stehle et al. [71]. At the time of publication, they evaluated their radix sort to be the fastest sorting algorithm for the GPU, outperforming Thrust, CUB, and MGPU. With release 1.11.0, Thrust's sorting performance has been improved up to 2× for 32 and 64-bit numeric keys. The performance improvements result from reducing the number of memory reads/writes per pass from $3n$ to $2n$ and, thereby, increasing the number of considered digits per sorting pass. This is achieved by calculating the histogram once and further decoupling the parallel computation of the prefix sum [1, 42]. Now, `thrust::sort` not only outperforms MGPU's merge sort (5.5×), but the MSB radix sort by Stehle et al. as well (1.6×). Thus, we show that, contrary to Shanbhag et al.'s findings [66], MSB radix sorts are not inherently better suited for the GPU than their LSB counterparts. Just like CUB, Thrust's sort allows for passing a custom memory buffer which is internally used as an auxiliary buffer, eliminating the need to dynamically allocate memory during the sorting execution. The space complexity of `thrust::sort` is in $O(n)$ as it needs temporary memory as large as the input size n plus a small constant overhead (less than 64 MB). Dynamic memory allocations are expensive and should be avoided in performance-critical applications [47]. On the IBM AC922 for example, we measure allocating 8 GB of GPU memory to take $150ms$. We, therefore, pre-allocate all needed device memory for both multi-GPU sort implementations.

5.2 P2P-Based Multi-GPU Sort

The P2P-based multi-GPU sort algorithm performs all computation on the GPUs. After the data is locally sorted on each GPU, the merge phase produces the globally sorted array across all g GPU chunks, which are then copied back to the host. The comparison-based nature of the merge phase allows for sorting any comparable data. The main benefit of merging on the GPUs is the considerably faster execution because, similar to sorting, highly parallel GPU merge algorithms outperform CPU implementations [21]. Multiple GPUs merge their data by swapping blocks of keys in between each other via P2P transfers, based on a specific pivot.

Algorithm 1 Pivot selection for two sorted arrays A and B

```
1: function SELECT_PIVOT(array A, array B)
2:     low ← 0, high ← size(B)
3:     while low < high do
4:         mid ← high − (high − low)/2
5:         if A[size(A) − mid] ≤ B[mid − 1] then
6:             high ← mid − 1
7:         else
8:             low ← mid
9:     return low
```

Pivot Selection. To merge two sorted arrays A and B of equal size n into one sorted array AB, we need to swap keys between A and B so that every key in A is less than or equal to every key in B. Tanasic et al. present which keys to swap by calculating a pivot position p in B and the mirrored position p' in A where $p' = |A| - p$. The pivot is selected so that the *first* p' keys of A and the *first* p keys in B are $\leq$ to the *last* p keys of A and the *last* p' keys in B [72].

$$\forall i, j \in \{0, \ldots, n-1\}, i < p \wedge j \geq p' : B[i] \leq A[j] \wedge$$
$$\forall i, j \in \{0, \ldots, n-1\}, i \geq p \wedge j < p' : A[j] \leq B[i]$$

The *first* p keys in B are swapped with the *last* p keys in A, fulfilling the condition that $A \leq B$, while guaranteeing the two exchanged blocks to be of equal size p. This allows for perfect load-balancing throughout the whole multi-GPU sort as each GPU's chunk size is kept constant. Since we swap blocks of consecutive keys, it is necessary to bring each array into sorted order individually. As both arrays are sorted initially, the exchanged blocks are themselves sorted. As a result, after the swap, each of the two arrays A and B contains two sorted sublists, which are then merged, bringing the concatenated array into sorted order. This already describes the merge phase for two GPUs. The arrays A and B are distributed to one GPU each. After P2P memory swaps, the two GPU-local merges complete the phase (see Figure 8).

We implement the pivot selection using an adapted binary-search that operates on two sorted arrays, as seen in Algorithm 1. Contrary to Tanasic et al., our pivot selection guarantees to pick the leftmost pivot. This minimizes the number of keys transferred via the P2P interconnects. In the extreme case, when no P2P swap is necessary, our pivot selection returns zero, and we completely skip the P2P swap, further reducing data transfers. The performance gain of this optimization depends on the number of duplicate keys and the data distribution (see Section 6.3). Keys of remote GPUs are accessed via P2P memory reads. The number of remote memory accesses as well as the algorithm's complexity is in $O(\log(n))$, where n is the chunk size. We measure fast execution, even without parallelization.

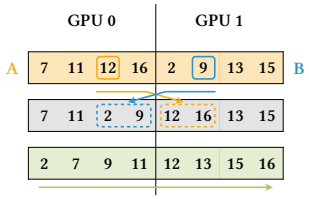


Figure 8: Merge phase for two GPUs

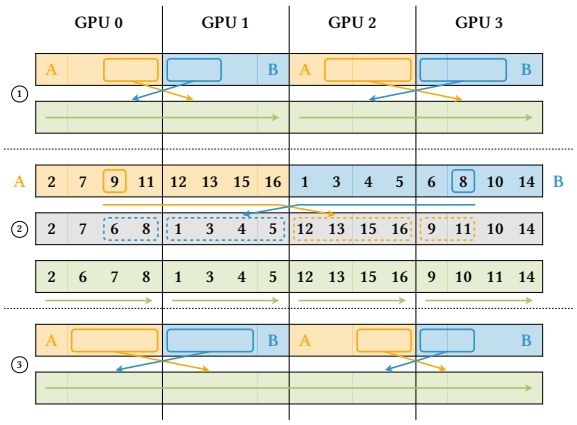


Figure 9: Merge phase for four GPUs

Across our systems, the pivot selection accounts for 0.03% of the total execution time for 2B integers on four GPUs.

After the pivot is selected, we swap blocks of memory via overlapped P2P transfers to utilize the interconnects' bidirectional bandwidth. Tanasic et al. propose not to implement the memory swaps in-place but rather use secondary buffers to avoid synchronizing. As a consequence, for the two chunks that are divided by the pivot, the memory block that is not swapped needs to be locally copied to the same secondary buffer that the P2P copy stream of the corresponding remote chunk writes into. This local copy is performed within device memory. We measure it to be orders of magnitude faster than the interconnect transfer because of the significantly higher GPU memory bandwidth. We measure device-local copies to be faster than P2P transfers – 3× over NVLink 3.0, 5× over three NVLinks 2.0, and 42× over PCIe 3.0. Because the local copies and the P2P transfers happen concurrently, we do not sacrifice performance. Also, we do not introduce any memory overhead because `thrust::sort` already needs an auxiliary memory buffer of size n, which we reuse for the P2P swaps. Since the copy streams write to separate memory addresses, no synchronization is needed and the throughput not throttled. Thus, we confirm that the out-of-place P2P swap optimization still holds on modern systems.

Merge Phase. When the data is distributed to more than two GPUs, the merge phase gets more complex. Figure 9 depicts the merge algorithm on four GPUs. It starts by merging pairs of GPU chunks (C_0 with C_1 and C_2 with C_3). As a result, a sorted array across GPUs 0+1 and 2+3 is obtained in ①. In the global merge stage in ②, a pivot is selected among all chunks. If the pivot falls into C_3, the entire chunk C_1 is swapped with C_2. Additionally, we swap the pivot-determined blocks in C_0 and C_3, which then need to be locally merged. After the global merge stage, all keys in C_0 and C_1 are less than or equal to the ones in C_2 and C_3. If concatenated, however, the arrays across C_0+C_1 and C_2+C_3 are not sorted yet. Subsequently, another pair-wise merge stage produces the final, sorted output in ③. Tanasic et al. design the algorithm for up to four GPUs only. We extend it to any number of GPUs g with $g = 2^k$ (see Algorithm 2). This allows us to fully utilize the DGX A100.

Once the chunks are locally sorted, the merge phase produces the globally sorted array through a series of *merge stages*. In divide-and-conquer fashion, the problem of merging g GPU chunks is solved

Algorithm 2 Merge phase for a set of GPUs G

```
1:  function MERGE_CHUNKS(array G)
2:      g ← size(G)
3:      if g > 2 then
4:          MERGE_CHUNKS(G[0, g/2])
5:          MERGE_CHUNKS(G[g/2, g])
6:      pivot ← SELECT_PIVOT(G)
7:      if pivot > 0 then
8:          chunks ← SWAP_CHUNKS(pivot, G)
9:          MERGE_CHUNKS_LOCALLY(pivot, chunks, G)
10:     if g > 2 then
11:         MERGE_CHUNKS(G[0, g/2])
12:         MERGE_CHUNKS(G[g/2, g])
```

by merging the *left* as well as the *right* half of chunks (Lines 4 and 5). We define a merge stage as the algorithm's subsection when merges occur at a certain recursion tree level. Thus, a merge stage is identified by the number of chunks i that are being merged. When we sort on g GPUs, the merge stage $i = 2$ describes the $g/2$ pairwise merges on the leaf-level while the merge stage $i = g$ defines the global merge at root-level where the pivot is selected across all g chunks. Once the recursion tree is fully constructed down to its leaves, the first merge stage is executed, during which, pairs of chunks are merged into one sorted array by 1) finding the pivot, 2) swapping the corresponding keys, and 3) merging the affected chunks locally (Lines 6–9). As the recursion tree resolves, merge stages are performed on more than two GPUs. Then, executing the three merge steps does not result in a globally sorted array, as already seen in Figure 9. Instead, swapping keys re-distributes them into their correct half of chunks, which are then each merged by subsequent recursive calls (Lines 11, 12). For locally merging on a single GPU, we use Thrust because we evaluate it to outperform MGPU up to 1.7× for two sorted lists of 8 GB each [49, 54].

5.3 Heterogeneous Multi-GPU Sort

The heterogeneous multi-GPU sorting algorithm (HET sort) utilizes the CPU for merging. The locally sorted g chunks are copied back from the GPUs to host memory where a CPU-based multiway merge algorithm produces the globally sorted output data.

Multiway Merge Algorithm. For a fair comparison of P2P sort to HET sort, the CPU multiway merge primitive needs to be sufficiently optimized. The lower bound time complexity of any comparison-based k-way merging algorithm is $O(n*\log(k))$, where k is the number of sublists. Typical implementations are heap-based and require $2 * \log(k)$ comparisons in each step of computing the next smallest key. However, the loser tree data structure performs better as it needs exactly $\log(k)$ comparisons [6]. Efficient algorithms work out-of-place (i.e., they require two times the input size in main memory). In-place approaches have a worse time complexity and perform poorly in practice [7, 11]. Since we work on modern platforms with enough main memory and optimize for end-to-end sort durations, we favor an out-of-place algorithm. Multiway merging on the CPU is memory bandwidth-bound [9, 19]. Balkesen et al.'s two published single-threaded k-way merging primitives [9], used by the authors in a parallel sort-merge join algorithm, do not saturate the memory bandwidth of our systems. The multi-threaded k-way merging algorithm included in the `gnu_parallel` compiler extension utilizes the loser tree data structure and, therefore, has the best conceivable time complexity [17, 18]. We measure its memory bandwidth utilization across our three systems using Likwid [16, 74]. First, we measure the maximum sustainable memory bandwidth on all three systems using the STREAM benchmark adapted to our NUMA architectures [41]. We confirm the observation of Li et al. that the DRAM of modern high-performance systems achieves 75-80% of its theoretical bandwidth rate [37]. Second, we measure `gnu_parallel::multiway_merge`'s performance by merging $n \in \{2, 8, 32\}$ billion integers equally distributed into $g \in \{2, 4, 8\}$ sorted sublists. We find that `gnu_parallel`'s multiway merge saturates main memory with 71-94% across all three platforms. Thus, we utilize it as a competitive CPU merging primitive for HET sort.

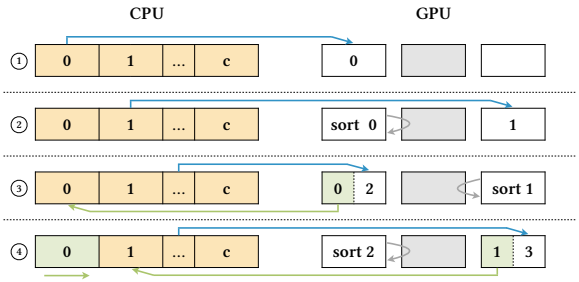


Figure 10: Sorting large data with the 3n-approach (1 GPU)

Sorting Large Data. Contrary to P2P sort, HET sort is not limited by the combined GPU memory. Instead, it is explicitly designed to sort large data sets (i.e., data that exceeds the combined device memory of a multi-GPU system), assuming that the main memory capacity is sufficiently large. It achieves this by sequentially sorting multiple chunk groups. We refer to the set of g chunks, each distributed to one GPU, as a chunk group. As soon as the first chunk group arrives on the GPUs, its chunks are sorted and copied back to host memory. Then, the second chunk group will be sorted. This process continues until all chunks are sorted and reside in main memory, leaving an ever increasing number of sorted sublists for the CPU to merge. It is crucial to utilize the interconnect's bidirectional transfer capability for copying the chunks back and forth. While a sorted chunk is copied back to the host, the next chunk is copied to its GPU concurrently. Modern GPUs are typically equipped with at least two copy engines which provide the necessary hardware-support for efficient transfers. While the bidirectional copy optimization is without alternative, there are two approaches with regards to the number of concurrently performed operations which, in turn, influences the GPU memory usage.

3n-Approach. The first approach reserves three buffers of chunk size n in GPU memory. One buffer stores the data of chunk i while another is used as the auxiliary buffer for sorting chunk i. The third buffer is used for copying chunk $i+1$ from host to device while also copying the sorted chunk $i-1$ back to the host. Performing two copy operations on the same buffer is achieved by an in-place data transfer swap. Thus, this approach allows for interleaving the two copy operations with the sort execution while only requiring three memory buffers on each GPU. Figure 10 illustrates the behaviour. Concurrently executing copy and compute operations hides the

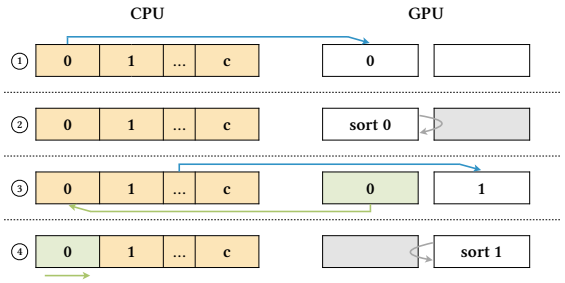


Figure 11: Sorting large data with the 2n-approach (1 GPU)

cost of either the data transfer or the GPU sort (whichever takes less time). Given our data transfer and single-GPU sorting benchmarks, we can be sure that on-GPU sorting is the faster operation. For example, we measure Thrust to sort 16 GB of integers 4.75× faster than the corresponding bidirectional copies take on the DGX A100. On the IBM AC922 and its NVLink 2.0 CPU-GPU interconnects, sorting 12 GB is still 2× faster than the data transfer because of the bidirectional copy overhead. For large enough numbers of chunk groups, the total duration of the GPU sort phase is thus reduced down to the data transfer time. Stehle et al. propose this strategy for their heterogeneous sort for large data [71].

2n-Approach. Our approach reduces the GPU space complexity to $2n$ by introducing a synchronization step: After a chunk is copied from host to device, the GPU sorts the chunk, blocking all copy operations. Once sorted, the chunk is copied back to main memory while simultaneously copying the next chunks to the GPUs. While sorting a chunk, `thrust::sort` uses the second buffer as auxiliary memory. While copying the sorted chunk back and the next one onto the GPU, each copy stream operates on one of the two buffers. At any given point in time, either copy or compute operations are executed (see Figure 11). While the level of parallelism is reduced, the increased GPU buffer size reduces the total number of chunks. For each round of expensive CPU-GPU data transfers, bigger chunks of sorted data are being returned to main memory. Thus, fewer sublists need to be merged in the final multiway merge.

Stehle et al. evaluate their 3n-approach on a single GPU and show that it outperforms state-of-the-art parallel CPU radix sort PARADIS [13, 71]. Other heterogeneous CPU-GPU sorting algorithms propose overlapping copy and compute operations as an optimization to mitigate the transfer bottleneck as well [58, 76]. However, all of these approaches have been evaluated at a time when PCIe was the newest interconnect. Since then, the computational power of GPUs improved while GPU sorting primitives were further optimized. Moreover, we evaluate on platforms with different types of modern interconnects. Thus, it is necessary to re-evaluate if overlapping copy and compute operations at the cost of reducing the effective GPU memory usage is still the best option for sorting large data. We compare both approaches in Section 6.2.

Eager-Merging. Gowanlock et al. propose eager merging as an optimization for large data sets that aims at lowering the load imbalance between the CPU and the GPUs by reducing the CPU's final merging workload [19]. The number of sorted sublists in the final multiway merge can be decreased through eager merging runs which are performed as soon as a certain number of sorted chunks are returned from the GPUs, all while the GPUs are sorting the remaining chunks. Assuming that the time it takes the CPU to merge g chunks is approximately the same as g GPUs need to sort the chunk group, we can perfectly interleave GPU and CPU workloads. Given c chunk groups, each of which consists of g chunks, instead of merging all $c * g$ chunks in one final merge, the eager merging approach lowers the number of final chunks to $c - 1 + g$. We skip eagerly merging the last chunk group, which would postpone the final merge and increase the total sort duration, because at that point, there is no GPU workload left. Gowanlock et al. measure marginal performance improvements when merging eagerly, claiming that for larger data, the speedup factor would increase [19]. However, because they evaluate on outdated hardware, we re-evaluate the performance of eagerly merging on modern systems in Section 6.2.

5.4 Algorithm Discussion

We compare the two algorithms and outline the expected performance differences based on an analysis of the performed data transfers as well as the overall merge phases. For a direct comparison, we only focus on sorting data sets that fit into the combined GPU memory of a multi-GPU system. Then, the CPU-GPU data transfers are exactly the same for the two algorithms as both transfer the entire data set of length n to the g GPUs and back once, distributing a chunk of size n/g to each GPU. Since we use the fastest single-GPU sorting algorithm to sort a chunk for both algorithms, the performance difference can only lie in the merge phase. While P2P sort requires P2P data transfers between the GPUs in its merge phase, the HET sort does not need any more data transfers as the chunks are merged in main memory. Given that we sort g chunks, HET sort's final merge phase has a time complexity of $O(n * \log(g))$. P2P sort, on the other hand, requires $O(g-1)$ merge stages, each of which involves P2P data transfers between the GPUs. In the worst case, one merge stage copies $O(n)$ keys. This results in a total of $O(n * (g-1))$ P2P transfers within the merge phase of P2P sort. On average (for uniform data distributions), however, the pivot falls into the middle of the chunk which results in significantly less data to be swapped: $\theta(\frac{n}{2} * (g-1))$. We have already measured the negligible impact of the pivot selection algorithm on the total execution time. Based on our benchmarks of GPU merging primitives and interconnect analysis, we can conclude that the local two-way merge computation within each GPU's memory does not have a relevant running time either. Thus, we know that the merge phase of P2P sort mainly depends on the P2P memory swap throughput. Hence, we expect P2P sort to outperform HET sort as long as the number of GPUs g is small enough and the system's P2P interconnects relevant for the merge phase provide bandwidth rates approximately as high as the CPU's main memory (i.e., when the P2P swaps utilize NVLink 2.0 or 3.0 interconnects exclusively).

To achieve the best performance with P2P sort on g GPUs, it is not enough to choose the g GPUs whose interconnects combine the highest bandwidth. We need to specify which GPUs perform P2P swaps between each other so that the runtime is minimized. We identify the exact P2P connections for a GPU set (i, j, k, l) using the order of its GPU identifiers i-l, thereby defining which chunks are merged in certain merge stages: GPUs (i, j) and (k, l) are merged in the pair-wise merge stages while, in the global merge stage, the

P2P swaps happen between GPUs (i, l) and (j, k). The copy pattern of our parallel P2P transfer benchmarks in Section 4.3 mimics these merge phase swaps. Depending on the interconnect topology, the order influences the sort duration. On the IBM AC922, for example, the best performing 4-GPU set is $(0, 1, 2, 3)$ because the pair-wise merges happen between NVLink-interconnected GPUs. For the global merge stage, we cannot avoid traversing the host-side. Consequently, the GPU set $(0, 2, 1, 3)$ performs worse for P2P sort. For HET sort, the GPU set order does not influence the performance.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of multi-GPU sorting on systems with fast interconnects. Our analysis in Section 4 shows the importance of choosing the most suitably interconnected GPUs. For example, GPU pair $(0, 2)$ achieves higher CPU-GPU copy throughput than $(0, 1)$ on the DGX A100. Thus, when sorting with g GPUs, we always choose the GPU set with the best transfer performance, which includes optimizing the GPU set order for P2P sort (see Section 5.4). In all experiments, the initial data is stored within host memory of node 0 and the CPU-GPU transfers are included in the sort duration. We disregard the GPU memory allocation times as we pre-allocate the memory, assuming exclusive system usage. We run each experiment 10 times and report the arithmetic mean across all runs. Our open-source benchmark suite automatically runs the experiments and generates plots for the results.

CPU Sort Baseline. For a fair comparison of multi-GPU sorting to the CPU, we evaluate CPU sorting primitives from parallel algorithm libraries, as well as the state-of-the-art CPU radix sort algorithms from the literature. Our benchmarks include the GNU parallel algorithms extension [17, 69], Intel's Thread Building Blocks library [29, 60], the parallel C++17 extension of `std::sort`, the parallel in-place radix sort PARADIS by Cho et al. [13], and the SIMD-enabled LSB radix sort by Polychroniou et al. [61]. We find that PARADIS outperforms the library primitives on all our systems. While the SIMD-enabled LSB radix sort is even faster for data sets of $\leq$ 2B keys on the DGX A100 and $\leq$ 8B on the DELTA D22x, PARADIS provides the best sorting performance for larger data sets while being platform-independent. Polychroniou et al.'s sorting algorithms use Intel SIMD instructions which the IBM AC922 does not support. Thus, we employ PARADIS as our CPU baseline.

6.1 Multi-GPU Sorting Algorithm Comparison

In this subsection, we analyze the performance of the two multi-GPU sorting algorithms (P2P sort and HET sort) for increasing numbers of GPUs. For these experiments, the data to be sorted consists of uniformly distributed integers. We evaluate which algorithm performs better on each of the three platform from Table 1. To compare the two algorithms, we restrict the data size to fit into the combined memory of g GPUs. Thus, the heterogeneous approach sorts one chunk group and merges its g chunks once on the CPU. As a result, the different pipelining strategies described in Section 5.3 do not apply as eagerly merging is not possible and the 3n and 2n-approach perform exactly the same (given the same chunk size).

For a more accurate evaluation, we break down the end-to-end sort duration into four phases (HtoD copies, DtoH copies, on-GPU sorting, and merging), showing which algorithm parts most heavily influence the execution time. The merge phase is performed either on the GPUs via P2P memory swaps or on the CPU, depending on the algorithm. The GPUs execute partly uncoupled (e.g., one GPU can start sorting while another still copies data from the host). We define a phase to end when the last GPU completes executing it.

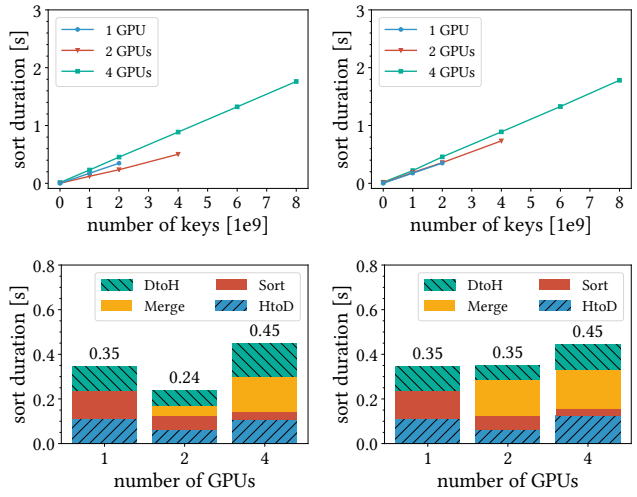
6.1.1 IBM Power System AC922. We start the multi-GPU sorting algorithm comparison with the IBM AC922 (Table 1a). We analyze the single-GPU baseline as well as GPUs $(0, 1)$ and $(0, 1, 2, 3)$.

Figures 12a and 12b (top) show how the two multi-GPU sorting algorithms scale with increasing data sizes. Our first observation is that both algorithms scale linearly with the number of keys for one, two, and four GPUs. We observe that on this system and data sizes that fit onto one GPU, utilizing four GPUs does not improve performance over the single-GPU baseline. Figure 12a shows that P2P sort achieves the best sorting performance, with a total sort duration of $0.24s$ for two GPUs. This constitutes a 1.5× speedup over one and 1.9× speedup over four GPUs. For HET sort, we observe longer sorting times on two GPUs as P2P sort outperforms HET sort by 32% for 4B integers. Using all four GPUs, both algorithms perform the same. Compared to the CPU radix sort PARADIS, we evaluate speedups of up to 14× for P2P sort, and 9× for HET sort.

Sort Duration Breakdown. The sort duration breakdowns in Figures 12a and 12b (bottom) help in analyzing the main bottlenecks. For a clear comparison, each GPU set sorts 2B integers. Thus, each individual GPU sorts less data when using more GPUs. On this system, sorting with a single GPU is not bound by one phase only as all three make up approximately one third of the total duration. For **P2P sort**, scaling to two GPUs results in the sort and the HtoD phase being halved while the DtoH copies are reduced down to only 64% (as already covered in Figure 2). In addition to the merge phase overhead, this explains the 1.5× speedup. Most of the merge phase time is spent on P2P swaps. For two GPUs, the merge phase accounts for about 20% of the total execution time because the GPUs $(0, 1)$ are directly connected via NVLink 2.0 for peak P2P throughput (see Figure 5b). For four GPUs, we measure the merge phase to be 3.6× slower than for two. This is because, the number of merge stages increases and more importantly, the unavoidable CPU-traversing P2P swaps throttle the merge performance due to the low X-Bus bandwidth. **HET sort**'s performance is bound by the CPU merge phase, which makes up 46% of the total sort duration for two GPUs. As discussed in Section 5.3, the CPU merge phase is bound by the main memory bandwidth. Compared to the P2P-based GPU merge, the CPU takes 3.6× longer to merge two chunks. For four chunks, the CPU merge duration increases by only 8%, which explains why the two algorithms perform the same on four GPUs. Given a constant total data size, we measure similar execution times for `gnu_parallel`'s multiway merge with two, three, and four sorted sublists, as the main memory bandwidth is saturated in any case.

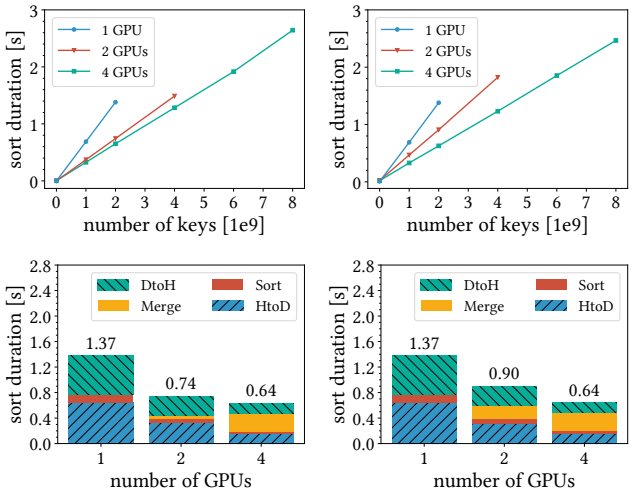
6.1.2 DELTA System D22x M4 PS. Next, we look at the DELTA D22x (Table 1b). We again depict GPU sets (0), $(0, 1)$, and $(0, 1, 2, 3)$.

The measurements in Figures 13a and 13b (top) show linear scaling with increasing data sizes for both algorithms. In comparison to the single-GPU baseline, P2P sort achieves 1.86× the performance with two GPUs while four GPUs sort 2.1× faster than one. This platform scales comparatively well up to four GPUs, although the speedup from two to four is less than from one to two. HET sort



(a) P2P sort (b) HET sort

Figure 12: Multi-GPU sort performance on the IBM AC922



(a) P2P sort (b) HET sort

Figure 13: Multi-GPU sort performance on the DELTA D22x

improves the performance for two GPUs by 52% over one. With four GPUs, it reaches the same 2.1× speedup that P2P sort achieves. Thus, HET sort again shows worse performance than P2P sort on directly NVLink-interconnected GPUs. On four GPUs, both algorithms again sort equally fast due to a lack of P2P interconnects. Both multi-GPU sorts outperform CPU sort PARADIS up to 9×.

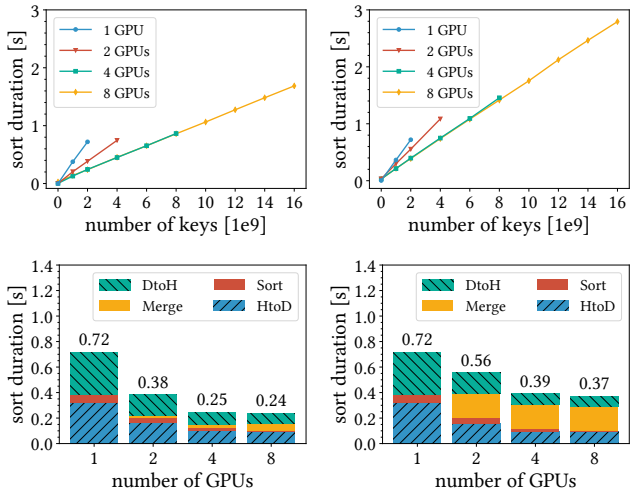
Sort Duration Breakdown. In Figures 13a and 13b (bottom), we observe that the **P2P sort**'s 1.86× speedup for two GPUs over one is higher on this platform than on the IBM AC922. First, the DtoH copy is twice as fast on two GPUs compared to one, which is not the case on the IBM AC922. All phases except the merge scale with 2× speedup on the DELTA D22x. Second, relative to the total execution time, the merge phase has a negligible impact on the DELTA D22x due to the slow CPU-GPU transfers accounting for 84% of the execution time. The low PCIe 3.0 bandwidth also explains why P2P sort is still 3× faster with two GPUs on the IBM AC922 than on this system. In Figure 13a, we see why four GPUs outperform one by a factor of 2.1. The HtoD and DtoH transfers are 4× faster on four GPUs because of exclusive PCIe switches for each GPU and a high enough CPU interconnect bandwidth (Figure 3b). On this system, the main bottleneck shifts from CPU-GPU data transfers for one GPU to P2P transfers in the merge phase for four GPUs. Since the global merge stage is not optimally P2P-interconnected, it is PCIe 3.0-bound (Figure 6) and, thus, the merge phase makes up 45% of the total execution time. **HET sort** is slower than P2P sort on two GPUs because the CPU merges 3.8× slower than GPU pair $(0, 1)$. Similar to the IBM AC922, however, the CPU merges just as fast as the insufficiently interconnected four GPUs. The improvements with four GPUs are only possible on this system because the sort and copy durations scale perfectly.

6.1.3 NVIDIA DGX A100. For the DGX A100 (Table 1c), we evaluate multi-GPU sorting with GPUs (0), $(0, 2)$, $(0, 2, 4, 6)$, and eight GPUs.

In Figure 14a (top), P2P sort shows to achieve 1.9× faster execution time with two GPUs over one. Four and eight GPUs outperform one by factors of 2.9 and 3.0, respectively, reducing the total sort duration down to 0.24*s*. This makes utilizing all eight GPUs the best option for P2P sort, especially since eight GPUs can sort twice as much data as four. HET sort, seen in Figure 14b (top), improves the sort performance of a single GPU by 30% for two GPUs while four GPUs sort 1.84× faster than one. Scaling up to eight GPUs also provides the best HET sort performance, with 1.95× speedup over one GPU. Thus, P2P sort is faster than HET sort for all $g \in \{2, 4, 8\}$ on this system. Again, both achieve significant speedups (up to 7.8× for P2P sort and 5× for HET sort) over CPU-only sorting.

Sort Duration Breakdown. Figures 14a and 14b (bottom) show that sorting 2B integers with this system's NVIDIA A100 GPU takes almost half the time compared to the NVIDIA Tesla V100 of the previous two platforms. We further observe the significant benefit of NVSwitch-powered P2P transfers: The merge phase of **P2P sort** has a very small impact on the total execution time with only 4% for two, 13% for four, and 23% for eight GPUs. While the sort phase also plays a minor role, the data transfers via PCIe 4.0 are P2P sort's main bottleneck on this platform. The insufficient number of PCIe switches limits the system's scalability as transfer times are not reduced from four to eight GPUs (see Figure 4). The CPU merge duration of **HET sort** stays constant for increasing numbers of chunks. Furthermore, `gnu_parallel::multiway_merge` achieves shorter execution times on the DGX A100, merging almost twice as fast as the IBM AC922 for two chunks and 2.75× faster for four. This explains why scaling up to eight GPUs increases HET sort performance on this platform. Nonetheless, the P2P-based GPU merge is still significantly faster than the CPU merge (3.3× for eight GPUs) which is why P2P sort outperforms HET sort.

6.1.4 Conclusion. Across all systems, both multi-GPU sorting approaches show significant speedups over parallel CPU radix sort PARADIS, up to 14× (P2P sort) and 9× (HET sort). Therefore, multiple GPUs significantly accelerate sorting. The speedups are particularly high on the IBM AC922 as it is the system with the lowest number of physical CPU cores. We observe PARADIS to scale very well with increasing numbers of threads, but on the IBM AC922, it does not efficiently utilize 4× hyper-threading. Our experiments also show that, across all systems, the fastest sorting execution is



(a) P2P sort (b) HET sort

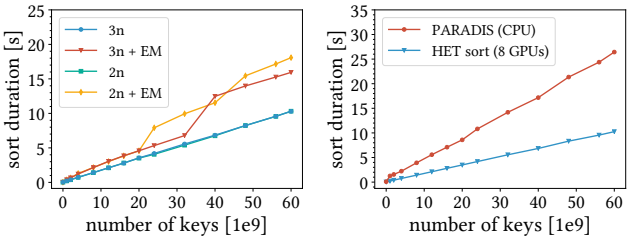
Figure 14: Multi-GPU sort performance on the DGX A100

achieved with P2P sort, outperforming HET sort 1.5× on two GPUs on the IBM AC922 while sorting up to 1.65× faster for any GPU set on the DGX A100. However, the scalability of P2P sort to increasing numbers of GPUs highly depends on the P2P interconnect topology. We show that for GPUs that are NVLink-interconnected, the CPU's merging performance is considerably weaker compared to the multi-GPU merge. When the systems lack direct P2P interconnects, HET sort is only equally as fast as P2P sort. The DGX A100 most strongly favors P2P sort because of its NVSwitch 3.0-powered interconnectivity. We measure the best scalability for this system, as two GPUs are 1.9×, and four GPUs 2.9× faster than one. With regards to the end-to-end sort duration, the CPU-GPU interconnects are a key deciding factor. The IBM AC922 achieves the same sort performance with only two GPUs as the DGX A100 with eight GPUs even though the DGX A100 has faster GPUs. (The NVIDIA A100 GPU sorts almost twice as fast as the Tesla V100.) The IBM AC922 reaches such competitive sorting times as it is the only system with NVLink 2.0 for CPU-GPU transfers. We conclude that the ever-increasing computational power of multiple GPUs, which is now accompanied by high-bandwidth P2P interconnects, require fast CPU-GPU transfers to outscale the CPU more efficiently.

6.2 Sorting Large Data

We evaluate the performance of the heterogeneous multi-GPU sort for data sizes that exceed the combined GPU memory capacity. First, we compare the different approaches and optimizations from Section 5.3. We analyze the performance of the 2n and 3n-approach, with and without eager merging in Figure 15a. Then, we compare the best algorithm variant with the state-of-the-art CPU-only radix sort PARADIS [13] in Figure 15b. We sort up to 60B integers (240 GB) and conduct the experiments on all three systems (see Table 1). The observations are conceptually the same and the takeaways consistent across all systems. Since it achieves the fastest execution, we depict our results for the DGX A100 with eight GPUs.

In Figure 15a, we observe that the eager merging strategy decreases performance for the 2n and the 3n-approach. The main reason is that the time it takes the CPU to merge the eight chunks



(a) HET sort approaches (b) HET sort vs. CPU-only sort

Figure 15: Sorting large data on the DGX A100, 8 GPUs

of one chunk group is significantly higher than the GPUs need for sorting the chunks and returning them back. This is especially the case for the IBM AC922, where one eager merge takes up to 2.2× longer than sorted chunk groups are returned. As a consequence, the queue of merging tasks grows over time, postponing the final merge phase. The supposed benefit of eagerly merging (i.e., the reduced number of sorted sublists for the final merge) does not pay off. We even measure the final merge of the eager merging approach to be 48% (DGX A100) and 70% (IBM AC922) longer than the final merge where all chunks are merged once. Additionally, we observe the effect of main memory bandwidth saturation for the eager merging approaches. The highly parallel multiway merge is already bound by the memory bandwidth for many threads. When concurrently executed with bidirectional transfers to and from the GPUs, we see a decrease in CPU-GPU copy throughput as the transfers and the CPU merge compete for host memory bandwidth. All effects combined, eagerly merging worsens performance 1.5-1.75×.

Figure 15a also shows that the 2n and the 3n-approach sort equally as fast when not eagerly merging. For a fair comparison, both approaches utilize the same total GPU memory of 33 GB. The 2n-approach employs a chunk size of 4.125B integers, while the 3n-approach uses chunks of 2.75B. As discussed in Section 5.3, the copy-compute overlap in the 3n-approach hides the sort duration on the GPUs. We measure the on-GPU sort to be the least influential factor in terms of total execution time. When sorting 60B integers, the full GPU sort phase and the final CPU merge phase each account for half the total duration, (2× 5s). Most of the sort phase duration is made up by data transfers to and from the GPUs. The accumulated time that the GPUs spend on sorting, accounts for only 3% of the total execution time on the DGX A100. Thus, the main advantage of the 3n-approach (i.e., hiding the sorting computation) does not have a significant performance impact. On the IBM AC922, where hiding the on-GPU sorting has the highest relative impact due to NVLink-powered CPU-GPU transfers, we observe a significantly higher merging bottleneck: For 32B integers on two GPUs, the final CPU merge accounts for 77% (10s) of the total execution time (13s), which overshadows the benefit of overlapping GPU copy and compute operations. Since the combined data transfer times are equal, the 2n and the 3n-approach perform almost identically.

In Figure 15b, we compare the multi-GPU HET sort (2n, no eager merges) with CPU-only sorting. We observe that HET sort outperforms the CPU even for increasing data sizes, despite the bidirectional copy overhead. We measure speedups of 2.6× for 60B integers on the DGX A100. On the IBM AC922 with two GPUs and the DELTA D22x with four GPUs, we measure HET sort to outperform the CPU with similar speedups (2.3-2.5×).

Conclusion. We find that, contrary to previous research [19], eagerly merging decreases HET sort's performance, due to host-side bottlenecks (i.e., the weaker CPU merge and the limited main memory bandwidth). Our evaluation shows that, on modern platforms, overlapping copy and compute operations does not notably improve the sorting duration, because, relative to the total execution time, the impact of hiding the GPU sort computation is negligible. Still, multiple GPUs significantly accelerate sorting large data.

6.3 Sorting Varying Data Sets

Lastly, we measure the impact of the data distribution and the data type on the execution time of both multi-GPU sorting algorithms.

Distribution Type. In Figure 16, we show the sort duration of the algorithms for differently distributed data for the IBM AC922 with two GPUs. We observe the performance of P2P sort to be stable for uniform and normal distributions. For reverse-sorted data, the number of P2P swaps is maximal, resulting in the longest execution time. If the data is nearly or already sorted, the sort duration goes down by 9-20% because very little or no P2P swaps are necessary. When sorting with four GPUs on the IBM AC922, we measure even higher speedups (1.4-1.6×) for optimal distributions because the impact of the P2P merge phase on the total execution time is higher. Consistently, we measure less variance for different distributions on the DGX A100 with NVSwitch. HET sort is stable for all distributions as the merge performance is memory bandwidth-bound.

Data Type. We evaluate both multi-GPU sort algorithms for different data types. We sort 4B integers and floats (32-bit) and 2B doubles and longs (64-bit). Thus, in each experiment, we sort 8 GB of data. On the NVIDIA A100, we observe the 32 and 64-bit data type sorting runs to perform very similarly (within 95%). Both GPUs have twice as many 32-bit as 64-bit cores. On the NVIDIA Tesla V100, however, sorting the 32-bit data types takes only 83-88% as long as for 64-bit types. Profiling reveals that `thrust::sort` performs disproportionately better on 32-bit keys on the Tesla V100.

7 DISCUSSION

Having evaluated two multi-GPU sorting algorithms, we observe both to scale linearly with increasing data sizes while outperforming highly parallel CPU-only sorting implementations, even for large out-of-core data. However, we find that P2P sort achieves better performance than HET sort, across three modern accelerator platforms as HET sort tends to become bound by the CPU merge and the main memory bandwidth. Thus, future research should evaluate the suitability of a P2P-based GPU merge for large data.

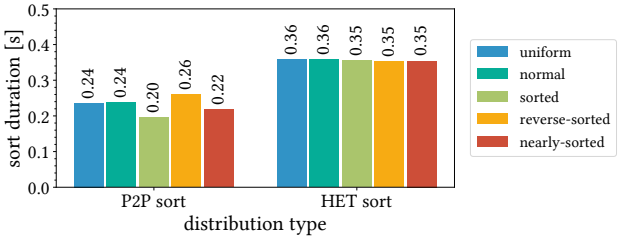


Figure 16: Sorting 2B integers for varying data distributions with 2 GPUs on the IBM AC922

Here, we confirm the conclusion of Gowanlock et al. who claim that fast interconnects make merging on the GPUs necessary [19].

Regarding interconnect bandwidth, we extend prior findings. Many GPU-accelerated algorithms, including P2P sort, are not only interconnect-bound for PCIe 3.0 [62, 63], but also for PCIe 4.0. Tanasic et al. evaluate their P2P-based algorithm on a purely PCIe 3.0 interconnected platform and still measure 3.3× the speedup on four GPUs [72]. At the time, the GPU's computational power was the main bottleneck. However, recent years' advances in GPU performance have outpaced the development of interconnects by far. This is why today, the sorting computation on the GPUs, is, in no case, the bottleneck. Multi-GPU systems that still include PCIe 3.0 interconnects are highly bandwidth-bound. Until now, the state-of-the-art approach of mitigating the data transfer bottleneck has been overlapping compute with copy operations [58, 71, 76]. However, our large data experiments show that this does not hold anymore since the performance of GPUs has become orders of magnitude higher than that of their interconnects. Even for high-bandwidth interconnects, we still observe transfer-related bottlenecks on the host-side to negatively impact the efficiency of modern multi-GPU systems. Low CPU-interconnect bandwidth makes it infeasible to involve remote GPUs on the IBM AC922, if the input data resides in the host memory of a single NUMA node. An insufficient number of PCIe switches limits the benefit of involving neighbouring GPUs due to shared bandwidth effects on the DGX A100. Thus, we conclude that, since P2P throughput has increased, hardware systems now need to provide fast CPU-GPU transfers to improve the scalability of modern multi-GPU systems to all their GPUs.

Nonetheless, we show that modern P2P interconnects accelerate merging considerably. For GPUs with NVLink 2.0 or NSwitch 3.0 as P2P interconnects, we measure speedups up to 14× with P2P sort over state-of-the-art CPU-only sorting. On these systems, inter-GPU communication is not the limiting factor anymore. Modern platforms increasingly include NVSwitch (e.g., top-of-the-line cloud instances) [3, 44]. Thus, assuming fast P2P throughput when designing multi-GPU algorithms becomes more and more viable. For systems with few P2P interconnects, future work should evaluate multi-hop routing for the P2P merge phase, similar to Paul et al.'s work [55]. Data transfers are redirected to their destination over multiple GPUs instead of traversing the host-side via PCIe 3.0. However, this strategy is limited to systems where multi-hop traversals can benefit from high-speed interconnects (e.g., DELTA D22x). More importantly, we suggest to reduce the P2P communication by designing a radix partitioning-based multi-GPU sorting algorithm which would require swapping keys between GPUs only once (all-to-all). This approach would highly benefit systems with many NVSwitch-interconnected GPUs such as the DGX A100.

8 CONCLUSION

In this paper, we conduct an extensive analysis of modern CPU-GPU and P2P interconnects covering serial, parallel, and bidirectional data transfers for multiple GPUs. Furthermore, we evaluate a P2P-based (P2P sort) and a heterogeneous (HET sort) multi-GPU sorting algorithm on three platforms. On all systems, P2P and HET sort significantly outperform the state-of-the-art CPU radix sort. If the GPUs are directly connected via high-speed NVLink 2.0 or NVLink 3.0-based NVSwitch, we show that P2P sort outperforms HET sort up to 1.65× as the CPU's merging performance is a limiting factor.

For P2P sort, we find that CPU-GPU data transfers are the main bottleneck. Since the IBM AC922 is the only system with NVLink 2.0 CPU-GPU interconnects, it achieves the shortest end-to-end sorting durations. Overall, multiple GPUs accelerate sorting considerably and the emerging trend towards increasing CPU-GPU interconnect bandwidth promises more efficient multi-GPU platforms [5, 53].

REFERENCES

[1] A. Adinets. 2020. *A Faster Radix Sort Implementation.* NVIDIA. Retrieved October 31, 2021 from https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf

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

[3] Amazon. 2020. *Amazon EC2 P4d Instances.* Amazon. Retrieved October 31, 2021 from https://aws.amazon.com/ec2/instance-types/p4/

[4] AMD. 2018. *AMD Radeon Instinct MI60: Unleash Discovery on the World's Fastest Double Precision PCIe Accelerator.* AMD. Retrieved October 31, 2021 from https://www.amd.com/system/files/documents/radeon-instinct-mi60-datasheet.pdf

[5] AMD. 2019. *AMD Joins Consortia to Advance CXL.* AMD. Retrieved October 31, 2021 from https://community.amd.com/t5/amd-business-blog/amd-joins-consortia-to-advance-cxl-a-new-high-speed-interconnect/ba-p/418202

[6] M. Axtmann, T. Axtmann, P. Sanders, and C. Schulz. 2015. Practical Massively Parallel Sorting. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '15).* Association for Computing Machinery, New York, NY, USA, 13–23. https://doi.org/10.1145/2755573.2755595

[7] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders. 2017. *In-Place (Parallel) Super Scalar Samplesort.* Karlsruhe Institute of Technology. Retrieved October 31, 2021 from http://algo2.iti.kit.edu/axtmann/invtalks/colgate/ipssss.pdf

[8] P. Bakkum and K. Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3).* Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706

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

[10] D. Cederman and P. Tsigas. 2010. GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors. *ACM J. Exp. Algorithmics* 14, 4 (January 2010), 1–24. https://doi.org/10.1145/1498698.1564500

[11] J.-C. Chen. 2006. A Simple Algorithm for In-Place Merging. *Inform. Process. Lett.* 98, 1 (April 2006), 34–40. https://doi.org/10.1016/j.ipl.2005.11.018

[12] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (August 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[13] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri. 2015. PARADIS: An Efficient Parallel Algorithm for In-Place Radix Sort. *Proc. VLDB Endow.* 8, 12 (August 2015), 1518–1529. https://doi.org/10.14778/2824032.2824050

[14] S. Chun, W. D. Becker, J. Casey, S. Ostrander, D. Dreps, J. A. Dreps, R. M. Nett, B. Beaman, and J. R. Eagle. 2018. IBM POWER9 Package Technology and Design. *IBM J. Res. Dev.* 62, 4 (July 2018), 1–10. https://doi.org/10.1147/JRD.2018.2847178

[15] J. Fang, Y. T. B. Mulder, J. Hidders, J. Lee, and H. P. Hofstee. 2020. In-Memory Database Acceleration on FPGAs: A Survey. *The VLDB Journal* 29, 1 (January 2020), 33–59. https://doi.org/10.1007/s00778-019-00581-w

[16] FAU. 2021. *Likwid: Performance Monitoring and Benchmarking Suite.* FAU. Retrieved October 31, 2021 from https://github.com/RRZE-HPC/likwid

[17] FSF. 2021. *The GNU C++ Library Manual: Parallel Mode.* FSF. Retrieved October 31, 2021 from https://gcc.gnu.org/onlinedocs/gcc-11.2.0/libstdc++/manual/manual/parallel_mode.html

[18] FSF. 2021. *The GNU C++ Library Reference Manual: multiway_merge.h.* FSF. Retrieved October 31, 2021 from https://gcc.gnu.org/onlinedocs/gcc-11.2.0/libstdc++/api/a00986.html

[19] M. Gowanlock and B. Karsin. 2018. Sorting Large Datasets with Heterogeneous CPU/GPU Architectures. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* Institute of Electrical and Electronics Engineers, New York, NY, USA, 560–569. https://doi.org/10.1109/IPDPSW.2018.00095

[20] G. Graefe. 2006. Implementing Sorting in Database Systems. *ACM Comput. Surv.* 38, 3 (September 2006), 1–37. https://doi.org/10.1145/1132960.1132964

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

[22] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. 2010. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.* 4, 2 (November 2010), 105–116. https://doi.org/10.14778/1921071.1921077

[23] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD '15).* Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[24] M. Harris. 2012. *How to Optimize Data Transfers in CUDA C/C++.* NVIDIA. Retrieved October 31, 2021 from https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/

[25] M. Heimel, M. Kiefer, and V. Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15).* Association for Computing Machinery, New York, NY, USA, 1477–1492. https://doi.org/10.1145/2723372.2749438

[26] IBM. 2018. *IBM Power System AC922: Technical Overview and Introduction.* IBM. Retrieved October 31, 2021 from https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf

[27] IBM. 2019. *IBM POWER9 Processor User's Manual.* IBM. Retrieved October 31, 2021 from https://ibm.ent.box.com/s/tmklq90ze7aj8f4n32er1mu3sy9u8k3k

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

[29] Intel. 2021. *OneAPI Threading Building Blocks.* Intel. Retrieved October 31, 2021 from https://github.com/oneapi-src/oneTBB

[30] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. 2014. Big Data and Its Technical Challenges. *Commun. ACM* 57, 7 (July 2014), 86–94. https://doi.org/10.1145/2611567

[31] 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).* Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[32] S. Kalid, A. Syed, A. Mohammad, and M. N. Halgamuge. 2017. Big-Data NoSQL Databases: A Comparison and Analysis of "Big-Table", "DynamoDB", and "Cassandra". In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA).* Institute of Electrical and Electronics Engineers, New York, NY, USA, 89–93. https://doi.org/10.1109/ICBDA.2017.8078782

[33] B. Karsin, V. Weichert, H. Casanova, J. Iacono, and N. Sitchinava. 2018. Analysis-Driven Engineering of Comparison-Based Sorting Algorithms on GPUs. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18).* Association for Computing Machinery, New York, NY, USA, 86–95. https://doi.org/10.1145/3205289.3205298

[34] A. Kemper and T. Neumann. 2011. HyPer: A Hybrid OLTP OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *2011 IEEE 27th International Conference on Data Engineering (ICDE).* Institute of Electrical and Electronics Engineers, New York, NY, USA, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[35] 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 (TPDS)* 31, 1 (January 2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[36] 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 IEEE International Symposium on Workload Characterization (IISWC).* Institute of Electrical and Electronics Engineers, New York, NY, USA, 191–202. https://doi.org/10.1109/IISWC.2018.8573483

[37] S. Li, D. Reddy, and B. Jacob. 2018. A Performance and Power Comparison of Modern High-Speed DRAM Architectures. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18).* Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3240302.3240315

[38] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD '20).* Association for Computing Machinery, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[39] 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).* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1987816.1987832

[40] E. Manca, A. Manconi, A. Orro, G. Armano, and L. Milanesi. 2016. CUDA-Quicksort: An Improved GPU-Based Implementation of Quicksort. *Concurr. Comput.: Pract. Exper.* 28, 1 (February 2016), 21–43. https://doi.org/10.1002/cpe.3611

[41] J. McCalpin. 1995. Memory Bandwidth and Machine Balance in High Performance Computers. *IEEE Technical Committee on Computer Architecture Newsletter* 2, 1 (December 1995), 19–25.

[42] D. Merrill and M. Garland. 2016. *Single-Pass Parallel Prefix Scan with Decoupled Look-Back.* Technical Report. NVIDIA. 1–9 pages. Retrieved October 31, 2021 from https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf

[43] 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 (June 2011), 245–272. https://doi.org/10.1142/S0129626411000187

[44] Microsoft. 2021. *Microsoft Azure ND A100 v4-Series*. Microsoft. Retrieved October 31, 2021 from https://docs.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series

[45] NVIDIA. 2017. *NVIDIA Tesla V100 GPU Architecture*. NVIDIA. Retrieved October 31, 2021 from http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[46] NVIDIA. 2018. *NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch*. NVIDIA. Retrieved October 31, 2021 from http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf

[47] NVIDIA. 2020. *CUDA C++ Best Practices Guide*. NVIDIA. Retrieved October 31, 2021 from https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[48] NVIDIA. 2020. *CUDA C++ Programming Guide*. NVIDIA. Retrieved October 31, 2021 from https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[49] NVIDIA. 2020. *Modern GPU: Patterns and Behaviors for GPU Computing*. NVIDIA. Retrieved October 31, 2021 from https://github.com/moderngpu/moderngpu

[50] NVIDIA. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. NVIDIA. Retrieved October 31, 2021 from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[51] NVIDIA. 2021. *CUB: Cooperative Primitives for CUDA C++*. NVIDIA. Retrieved October 31, 2021 from https://github.com/NVIDIA/cub

[52] NVIDIA. 2021. *NVIDIA DGX A100 System*. NVIDIA. Retrieved October 31, 2021 from https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf

[53] NVIDIA. 2021. *NVIDIA Grace CPU*. NVIDIA. Retrieved October 31, 2021 from https://www.nvidia.com/en-us/data-center/grace-cpu/

[54] NVIDIA. 2021. *Thrust: Code at the Speed of Light*. NVIDIA. Retrieved October 31, 2021 from https://github.com/NVIDIA/thrust

[55] J. Paul, S. Lu, B. He, and C. Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[56] 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 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 209–218. https://doi.org/10.1145/3297663.3310299

[57] H. Peters and O. Schulz-Hildebrandt. 2012. Comparison-Based In-Place Sorting with CUDA. In *GPU Computing Gems Jade Edition*, W.-M. Hwu (Ed.). Morgan Kaufmann, Boston, MA, USA, 89–96. https://doi.org/10.1016/B978-0-12-385963-1.00008-3

[58] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. 2010. Parallel External Sorting for CUDA-Enabled GPUs with Load Balancing and Low Transfer Overhead. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1–8. https://doi.org/10.1109/IPDPSW.2010.5470833

[59] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger. 2012. A Novel Sorting Algorithm for Many-Core Architectures Based on Adaptive Bitonic Sort. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 227–237. https://doi.org/10.1109/IPDPS.2012.30

[60] C. Pheatt. 2008. Intel Threading Building Blocks. *J. Comput. Sci. Coll.* 23, 4 (April 2008), 298.

[61] O. Polychroniou and K. A. Ross. 2014. A Comprehensive Study of Main-Memory Partitioning and Its Application to Large-Scale Comparison- and Radix-Sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 755–766. https://doi.org/10.1145/2588555.2610522

[62] S. M. A. Raza, P. Chrysogelos, P. Sioulas, V. Indjic, A. C. Anadiotis, and A. Ailamaki. 2020. GPU-Accelerated Data Management under the Test of Time. In *Online Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*. Conference on Innovative Data Systems Research, Amsterdam, Netherlands, 1–11.

[63] R. Rui, H. Li, and Y.-C. Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (December 2020), 708–720. https://doi.org/10.14778/3436905.3436927

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

[65] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 351–362. https://doi.org/10.1145/1807167.1807207

[66] A. Shanbhag, S. Madden, and X. Yu. 2020. *A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics (Extended Version)*. Technical Report. Massachusetts Institute of Technology. 1–17 pages. Retrieved October 31, 2021 from https://arxiv.org/pdf/2003.01178.pdf

[67] D. D. Sharma and S. Tavallaei. 2020. *Compute Express Link 2.0 White Paper*. CXL. Retrieved October 31, 2021 from https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf

[68] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 731–742. https://doi.org/10.1145/2213836.2213946

[69] J. Singler and B. Konsik. 2008. The GNU libstdc++ Parallel Mode: Software Engineering Considerations. In *Proceedings of the 1st International Workshop on Multicore Software Engineering (IWMSE '08)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/1370082.1370089

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

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

[72] 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-6)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2458523.2458524

[73] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. 2010. Hive – A Petabyte Scale Data Warehouse Using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[74] J. Treibig, G. Hager, and G. Wellein. 2010. Likwid: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *2010 IEEE 39th International Conference on Parallel Processing Workshops (ICPPW)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 207–216. https://doi.org/10.1109/ICPPW.2010.38

[75] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. 2009. SIMD-Scan: Ultra Fast In-Memory Table Scan Using On-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (August 2009), 385–394. https://doi.org/10.14778/1687627.1687671

[76] Y. Ye, Z. Du, D. Bader, Q. Yang, and W. Huo. 2011. GPUMemSort: A High Performance Graphics Co-Processors Sorting Algorithm for Large Scale In-Memory Data. *GSTF International Journal on Computing* 1, 2 (May 2011), 23–28. https://doi.org/10.5176/2010-2283_1.2.34