RMG Sort: Radix-Partitioning-Based Multi-GPU Sorting

Ivan Ilic[1] Ilin Tolovski[2] Tilmann Rabl[3]

Abstract: In recent years, graphics processing units (GPUs) emerged as database accelerators due to their massive parallelism and high-bandwidth memory. Sorting is a core database operation with many applications, such as output ordering, index creation, grouping, and sort-merge joins. Many single-GPU sorting algorithms have been shown to outperform highly parallel CPU algorithms. Today's systems include multiple GPUs with direct high-bandwidth peer-to-peer (P2P) interconnects. However, previous multi-GPU sorting algorithms do not efficiently harness the P2P transfer capability of modern interconnects, such as NVLink and NVSwitch. In this paper, we propose RMG sort, a novel radix partitioning-based multi-GPU sorting algorithm. We present a most-significant-bit partitioning strategy that efficiently utilizes high-speed P2P interconnects while reducing inter-GPU communication. Independent of the number of GPUs, we exchange radix partitions between the GPUs in one all-to-all P2P key swap and achieve nearly-perfect load balancing. We evaluate RMG sort on two modern multi-GPU systems. Our experiments show that RMG sort scales well with the input size and the number of GPUs, outperforming a parallel CPU-based sort by up to 20×. Compared to two state-of-the-art, merge-based, multi-GPU sorting algorithms, we achieve speedups of up to 1.3× and 1.8× across both systems. Excluding the CPU-GPU data transfer times and on eight GPUs, RMG sort outperforms the two merge-based multi-GPU sorting algorithms up to 2.7× and 9.2×.

Keywords: Multi-GPU sorting; radix partitioning; high-speed interconnects; database acceleration

1 Introduction

Today's data volumes oftentimes exceed the size that database systems can analyze efficiently [Gu15, Ja14]. To improve the data processing performance, research and industry exploit modern hardware. GPUs provide high computational power via thousands of cores, and a high-bandwidth memory [NV17, NV20]. For compute-intensive tasks on small, in-GPU-memory data sets, GPUs achieve orders of magnitude higher instruction throughput (e. g. TFLOPS) than CPUs. Thus, they are commonly used as accelerators for deep learning and HPC workloads [SMY20]. However, GPUs experience a slower adoption into the database systems market, because of the transfer bottleneck [CI18, Lu20]. For many GPU-based operator implementations, copying the data to the GPU and back over the PCIe 3.0 interconnect has been the limiting factor [GK18, Lu20, Ra20, RLT20].

In recent years, high-bandwidth, low-latency interconnects, such as NVIDIA's NVLink, AMD's Infinity Fabric, and the Compute Express Link (CXL) have been introduced [AM18,

[1] Hasso Plattner Institute, University of Potsdam, Germany ivan.ilic@student.hpi.de
[2] Hasso Plattner Institute, University of Potsdam, Germany ilin.tolovski@hpi.de
[3] Hasso Plattner Institute, University of Potsdam, Germany tilmann.rabl@hpi.de

doi:10.18420/BTW2023-15

NV18, ST20]. They increase the GPU-interconnect bandwidth close to that of main memory, accelerating CPU-to-GPU and P2P transfers. On hardware platforms with high-speed interconnects, GPUs efficiently accelerate data analytics workloads and core database operations [Lu20, Ma22, Ra20]. Sorting is one such operation, with applications in index creation, duplicate removal, user-specified output ordering, grouping, and sort-merge joins [Gr06]. Over the past years, numerous single-GPU sorting algorithms have been proposed and shown to outperform highly parallel CPU algorithms by orders of magnitude. Parallel radix sort algorithms are best suited for modern GPUs [MG16, SMY20, Ma22].

Modern server-grade systems combine multiple GPUs for an even higher computing power. The research community extended algorithms to utilize multiple GPUs [RLT20, Pa21]. To the best of our knowledge, all published multi-GPU sorting algorithms are sort-merge approaches [GK18, PSHL10, RLT20, Ta13]. The P2P merge sort by Tanasic et al. utilizes inter-GPU communication to merge the previously sorted chunks within GPU memory [Ta13]. The HET merge sort by Gowanlock et al. uses the CPU to merge GPU chunks. Evaluated on modern multi-GPU systems, both algorithms show promising speedups over a single GPU [Ma22]. However, their merging workload increases with the number of GPUs. For HET merge sort, the final multiway merge on the CPU quickly becomes a bottleneck [GK18, Ma22]. For P2P merge sort, scaling up the number of GPUs linearly increases the number of key swaps over the P2P interconnects. During their merge phase, each GPU swaps data with only one other GPU at a time. Thus, multiple merge steps are necessary. This algorithm design made sense in a time when GPUs had no direct P2P interconnects and GPUs communicated with each other via the host-side. On such systems, many concurrent P2P transfers over the PCIe 3.0 tree topology would suffer from shared bandwidth effects and throttle the overall throughput [Ma22]. Today, modern multi-GPU platforms incorporate direct high-bandwidth P2P interconnects. Recent hardware systems support non-blocking all-to-all inter-GPU communication [NV18, NV21b]. In the light of these hardware improvements, we propose RMG sort, a novel radix-partitioning-based multi-GPU sorting algorithm that utilizes the bandwidth of modern P2P interconnects more efficiently. We reduce inter-GPU communication by exchanging the radix partitions between all GPUs once and in parallel, independent of the number of GPUs. Our contributions are:

1. We design a novel multi-GPU sorting algorithm (RMG sort). We employ an MSB radix partitioning strategy to exploit modern P2P interconnects (Section 3).
2. We implement RMG sort in the CUDA framework and publish our source code with automated benchmark scripts to enable reproducible evaluation results[4] (Section 4).
3. We evaluate RMG sort on up to eight GPUs. We compare to parallel CPU-only algorithms and state-of-the-art, merge-based multi-GPU sorting algorithms (Section 5).

[4] `https://github.com/hpides/rmg-sort`

2 Background

In this section, we explain the required background information about the GPU hardware architecture, modern interconnect technologies, and radix sort algorithms.

GPU Architecture. GPUs are designed to support massively parallel computations, hiding memory access latency with concurrently executed computation [NV22b]. They are equipped with thousands of cores that are organized in a specialized hierarchy. The main unit of computation is the streaming multiprocessor (SM) [NV20], equivalent to the compute unit for AMD GPUs [AM20]. One GPU consists of an array of SMs [NV17, NV20]. Each SM can run multiple concurrent groups of threads (thread blocks). A thread block can run up to 1024 threads. The SM schedules these high numbers of threads in groups of 32 consecutive threads, so-called warps, that execute the same instruction. GPUs excel at achieving high instruction throughput rates and provide a high-bandwidth memory. The memory hierarchy is divided into *off-chip* and *on-chip* memory. Off-chip memory mainly consists of global HBM2 memory which all running threads access. It provides peak bandwidth rates of up to 1555 GB/s [NV20]. Compared to main memory, the GPU memory capacity is limited (up to 80 GB). The GPU's L2 cache hides the latency of global memory accesses. In addition, each SM comes with a local, high-bandwidth, low-latency L1 cache to accelerate computation on frequently used data. While the L1 cache automatically hides accesses of all threads of its SM, shared memory needs to be explicitly managed by the programmer.

GPU Interconnects. GPUs are attached to the CPU memory controller via an interconnect. The interconnect topology significantly impacts the performance of multi-GPU applications [Li20]. In the following, we explain modern interconnect technologies. PCIe 3.0 is used as the standard interconnect for many peripheral devices, including GPUs. It supports full-duplex communication at 16 GB/s per direction. PCIe 4.0 doubles this bandwidth rate for a theoretical peak of 32 GB/s. Multi-GPU systems with no direct P2P interconnects only support P2P communication through multi-hop host-side transfers. Over the last few years, hardware vendors introduced high-bandwidth, low-latency GPU interconnects for direct P2P transfers. AMD released the Infinity Fabric interconnect [AM18], while NVIDIA launched NVLink. NVLink 2.0 achieves 25 GB/s per link per direction. One NVLink 2.0-enabled GPU supports six links for a theoretical peak bandwidth of 150 GB/s per direction. NVLink 3.0 doubles the number of links per GPU for a bandwidth of 300 GB/s. NVLink is primarily designed to accelerate inter-GPU communication. NVSwitch is an NVLink-based switch chip by NVIDIA that enables non-blocking, all-to-all, inter-GPU communication at high bandwidth. It connects up to 16 GPUs between each other in a point-to-point mesh [NV18].

Radix Sort. Radix sort is a non-comparison-based sorting algorithm with linear computational complexity [Ag96, Gi19, SJ17]. Radix sort algorithms iterate over the keys' bits and partition the keys into distinct buckets based on their radix value. To reduce the number of iterations, radix sort algorithms look at multiple consecutive bits c at a time. Typically, a radix sort algorithm either starts from the most or the least significant bit (MSB or LSB). Given k-bit keys, the number of partitioning passes is $p = \lceil k/c \rceil$. In each partitioning

pass, each of the n input keys is scattered into one of 2^c distinct buckets depending on the currently considered c bits until all k bits have been considered, i. e. the keys are sorted. This leaves radix sort with a computational complexity of $O(n \times p)$. An LSB radix sort algorithm stores the 2^c buckets of the current partitioning pass only, as long as it respects the keys' sort order from preceding rounds. In contrast, an MSB radix sort algorithm refines the partitioning within each bucket in each round, keeping track of increasing numbers of buckets. To scatter the keys into their corresponding buckets in parallel, many GPU-based radix sort algorithms operate out-of-place [Sa10, SJ17, ZB91, ZW12].

3 Algorithm

In this section, we explain our radix-partitioning-based multi-GPU sorting algorithm (RMG sort). It sorts the input keys using only the GPUs. Therefore, it only sorts data sets that fit into the combined device memory of the system's GPUs. We use a most significant bit (MSB) radix partitioning strategy. Our algorithm requires one all-to-all key swap between the GPUs over the P2P interconnects, independent of the number of GPUs. Our algorithm reduces the inter-GPU communication compared to previous sort-merge algorithms. In summary, RMG sort works as follows: First, the unsorted input keys are copied to the GPUs in chunks of equal size. Each GPU partitions its keys locally, starting from the most significant bit, until every radix bucket on each GPU is *small enough* for the following all-to-all P2P key swap between the GPUs. The P2P key swap re-distributes all buckets across all GPUs so that afterwards, 1) each GPU contains keys of a distinct value range and 2) bringing all keys into the global sort order across the g GPUs does not require any further key swaps. In other words, after the P2P key swap, all keys of GPU i have smaller or equal most significant bits compared to the keys of GPU $i + 1$. Then, each GPU sorts its buckets locally to bring the keys across all g GPUs into the final sorted order.

This allows for two optimizations: First, we reduce the final sorting workload. Instead of sorting the entire chunk, each GPU sorts its radix buckets individually. Given that the partitioning phase already examined the most significant r bits of each key, we sort each bucket on the remaining $k - r$ bits. Secondly, we interleave the sorting computation with copying the data back to the CPU. Once a bucket is fully sorted, we transfer it back, while the remaining buckets are still being sorted. Thereby, we hide the time duration of the sorting computation on the GPUs. In the following sections, we explain how the radix partitioning phase ensures that one bucket exchange (P2P key swap) between the GPUs is sufficient, even for skewed data. We outline how we distribute the keys across the GPUs with nearly perfect load balancing and how we accelerate the final sorting computation.

3.1 On-GPU MSB Radix Partitioning

After the n input keys are copied to the g GPUs in equal sized chunks, each GPU partition its keys locally (i. e. in its own device memory). Each GPU first computes the histogram over

its $\lceil n/g \rceil$ keys on the most significant c bits. Calculating the prefix sum on the histogram returns the starting write offsets for each of the 2^c buckets. Using the prefix sum, each GPU partitions its chunk in device memory so that all keys of bucket i precede bucket $i + 1$.

For most data distributions, the probability is high that there is a radix bucket for which every GPU finds associated keys. In fact, for uniformly distributed keys, every GPU likely contains keys that belong to every one of the 2^c possible buckets. The goal of the P2P key swap is to re-distribute the keys across all g GPUs so that all keys that belong to the same bucket are aligned in the device memory of one and the same GPU. We also have to ensure that all keys of GPU i are smaller than or equal to the ones on GPU $i + 1$. We satisfy both constraints by distributing the keys across the GPUs in the order of their radix digit values, i. e. by distributing the buckets in ascending order: The buckets of the smallest radix values to GPU 0, and those with the highest radix values to GPU $g-1$. After each partitioning pass, each GPU sends its histogram to all other GPUs via the P2P interconnects. Thus, each GPU knows about the entire key distribution and computes the logical distribution of buckets, i. e. the placement of buckets across the g GPUs in ascending order. Here, we check whether the current level of partitioning allows for each complete bucket to fit onto its designated GPU. A bucket b on GPU i is *complete* if all of the keys that reside on GPU i and that fall into bucket b are aligned at subsequent addresses in the memory of GPU i. Given g GPUs, each GPU might produce a complete bucket b. A set of at most g complete buckets $[b]_0$, $[b]_1$, ..., $[b]_{g-1}$ forms a spanning bucket under a given logical bucket distribution if all keys that belong to bucket b will not fit into the memory of the designated GPU. A spanning bucket prevents us from performing the P2P bucket exchange because we could not fully sort the spanning bucket without further communication between those GPUs that the bucket spans. We need to refine each spanning bucket in subsequent partitioning passes.

The number of partitioning passes necessary depends on the distribution of input keys. The input data might be highly skewed and contain only leading zeros in the most significant c bits. It is desirable for our partitioning phase to split the keys into reasonably small buckets to avoid load imbalances between the GPUs. We perform multiple partitioning passes on subsequent sets of c bits, starting from the most significant one, until there are no spanning buckets left. Any subsequent partitioning pass refines only the spanning buckets while the buckets that already fit onto one GPU stay untouched. In Figure 1, we show an example of our radix partitioning algorithm on four GPUs. In the example, we sort 32-bit keys while considering $c = 8$ bits at a time. In the first pass, each GPU scatters its keys based on the bits [32..24). We show the result of the local partitioning step in the top half of each pass, i. e. the physical view of the GPU memory. All GPUs find many keys that belong to bucket 0. They exchange their histogram information which allows each GPU to construct the logical distribution of complete buckets, shown in the bottom half of each pass in Figure 1. The complete bucket `[0]` is a spanning bucket after the first partitioning pass. Consequently, we continue with another pass on bits [24..16) on the spanning bucket `[0]`. In the second partitioning pass, each GPU with keys of the spanning bucket `[0]` refines its part (e. g. into two smaller buckets `[0:0]` and `[0:1]` on GPU 0). After the histogram exchange of the second

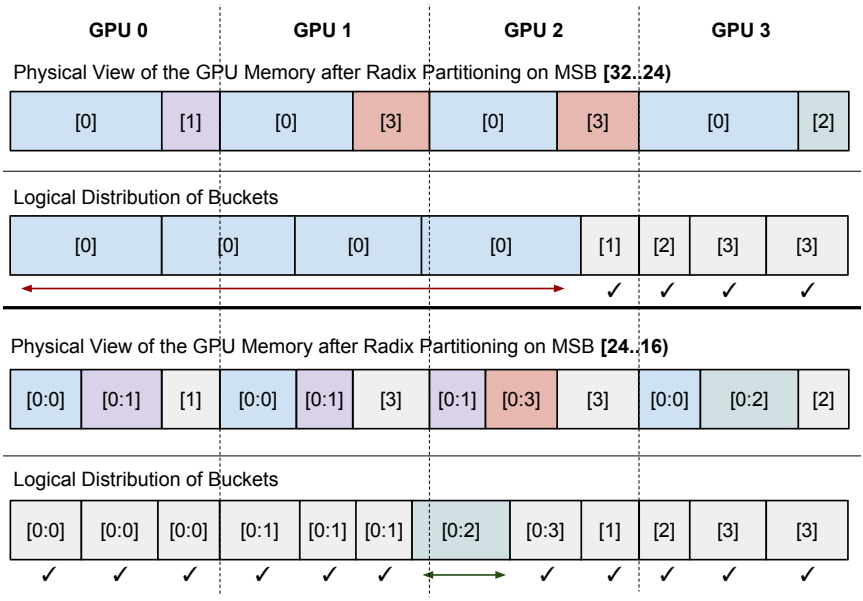


Fig. 1: Radix partitioning phase example

pass, we compute the logical bucket distribution and find that we resolved the spanning bucket. Bucket `[0:2]` (resident on GPU 3) is not a spanning bucket because we allow for nearly-perfect load balancing. Thus, our radix partitioning phase is completed.

3.2 Multi-GPU P2P Bucket Exchange

The bottom of Figure 1 shows an example of a final, logical bucket distribution. It aligns the complete buckets in globally sorted order across the GPUs. After the partitioning phase, the keys of each bucket still reside in the memory of their initial GPU as they have only been partitioned locally. In the multi-GPU P2P key swap, we re-distribute them between all GPUs. Each bucket's destination GPU can either be the same as the source GPU or a remote GPU in which case the memory copy goes over the P2P interconnects. For the P2P bucket exchange, the logical bucket distribution computed from the histogram broadcast of the final partitioning pass is sufficient. We measure that the overhead of calling one asynchronous copy per bucket is negligible even for high numbers of buckets. The CUDA driver appends calls into the CUDA stream queue and performs the copies at peak bandwidth.

Load Balancing. To reduce the number of partitioning passes, we do not enforce perfect load balancing. Instead, certain GPUs can handle slightly more keys than others. The first partitioning pass very rarely results in a bucket distribution that is perfectly aligned with

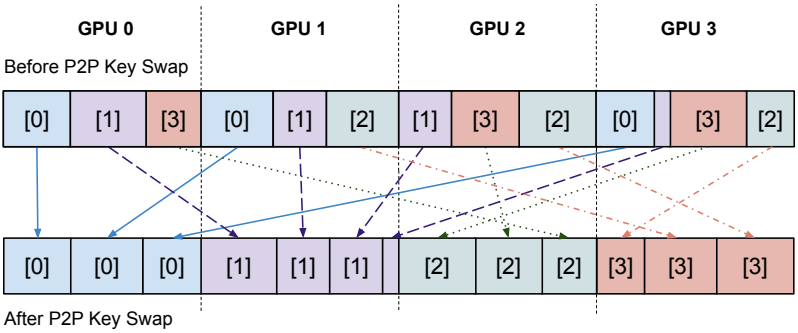


Fig. 2: P2P bucket exchange example

the chunk size. We define a threshold ϵ as the number of keys that each GPU can use as additional padding at the start and the end of its chunk buffer. This avoids treating slightly overflowing buckets as spanning buckets. Whenever a bucket overflows into a GPU by a number of keys $\sigma \leq \epsilon$, these σ keys are assigned to the adjacent GPU that already holds keys of that same bucket. If an overlapping bucket would span over two or three GPUs for a perfectly load-balanced approach, our additional ϵ-padding can, in the best case, avoid an entire partitioning pass. If the spanning bucket spans over more than three GPUs, our nearly-perfect load balancing approach reduces the number of GPUs that the bucket spans by up to two. This is because each GPU employs the ϵ-padding at the start and the end of its chunk. We empirically determine an optimal ϵ-padding of 0.5% of the initial GPU chunk size. With this ϵ, we measure that uniformly distributed keys require one partitioning pass.

For extremely skewed distributions, spanning buckets can occur after the last partitioning pass on the least significant c bits (e. g. if all n keys are of the same value). Having considered all k bits after the radix partitioning phase, there will still be one spanning bucket with n keys over all g GPUs. In fact, any spanning bucket that remains after the last partitioning pass must consist of keys of the same single value. Thus, we can choose arbitrary borders for where to split the spanning bucket. We simply distribute the keys of each last-pass spanning bucket in a perfectly load-balanced manner across the involved GPUs. Since we employ MSB radix partitioning, the number of buckets grows continuously with each partitioning pass. Considering c bits at a time, partitioning one spanning bucket divides it into 2^c sub-buckets. We view the initial input of n keys on the g GPUs as the initial spanning bucket. A GPU can be involved in at most two spanning buckets (one per adjacent GPU). Thus, the maximum possible number of spanning buckets per partitioning pass is $g - 1$. In that case, each spanning bucket spans over two GPUs. In total, we perform a maximum of p partitioning passes, with $p = \lceil k/c \rceil$. In the first pass, we partition the input data as one spanning bucket. This leaves us with an upper bound for the spanning buckets of $s_{max} = (g-1) \cdot (p-1) + 1$. The total number of buckets can not exceed $2^c \cdot s_{max}$. For $c = 8$, 64-bit keys and eight GPUs, this is equal to $256 \cdot s_{max} = 12.545$.

3.3 On-GPU Bucket Sorting

After the P2P key swap, each GPU contains buckets of distinct value ranges. While the different buckets are correctly ordered by their MSB values, the keys within each bucket are still unsorted. To sort each bucket, we use the single-GPU LSB radix sort algorithm `cub::DeviceRadixSort::SortKeys`, provided in NVIDIA's CUB library as it achieves the fastest performance [NV21a, Ma22]. Depending on the number of partitioning passes performed, we have already examined a certain number of most significant bits of each key. We use this information to accelerate the sorting computation. For each bucket, we specify the bit range that the local radix sort sorts on. The final bucket partitioning level is heterogeneous in the following sense: Some buckets are sufficiently partitioned after the first pass while others are refined through multiple passes. As a consequence, for each bucket that we sort, we have taken a different number of most significant bits into account during the partitioning phase. Since we store the partitioning pass p_b that generated each bucket, we determine the bit range to sort on as follows: $[endbit..0]$, with $endbit = k - ((p_b + 1) \cdot c)$. If a bucket went through the maximum number of partitioning passes p, we do not need to sort the bucket at all. Compared to sorting on all bits, specifying a reduced bit range improves the sorting performance of the local radix sort significantly. We measure a speedups between 30% and 200% for one, two and three partitioning passes on the NVIDIA Tesla V100 GPU.

The overhead of a single kernel launch is insignificant. When calling one kernel per bucket to sort, the launch times add up. In order to reduce the total number of buckets and mitigate the associated kernel launch overhead, we fuse neighbouring buckets whose number of keys is below a certain threshold. We can only fuse neighbouring buckets because we have to preserve the buckets' global sort order. We configure the threshold equal to 1% of the initial GPU chunk size. Whenever we fuse two buckets, the bit range that we sort the combined bucket on needs to be extended. To avoid extending the bit range too much, and thereby losing the benefit of the reduced sorting duration, we only fuse buckets of the same partitioning pass. As a result, the combined buckets share their initial bit range $[endbit..0]$ which we extend by the necessary minimum, i. e. by the most significant bit position in which the two bucket values differ. After the buckets have been fused and each final bucket's bit range is determined, each GPU sorts its buckets individually. As soon as a bucket is sorted, we transfer it back to the CPU, asynchronously launching the memory copy that transfers the latest sorted range of keys. This approach effectively overlaps the sorting computation with the device-to-host copy operation.

4 Implementation

In this section, we explain how we implement our MSB radix partitioning phase. Each partitioning pass includes computing histograms and scattering keys accordingly.

4.1 Histogram Computation

After the input data is copied to the GPUs, each GPU computes the histogram on the most significant c bits of its keys. To compute the histogram, we read all keys of the chunk, and atomically increment one of the 2^c bucket counters in our histogram array depending on the key's radix. When parallelizing the computation, we assign each thread block an equal number of keys to process. To achieve peak performance, each thread block first computes a block-local histogram stored in its shared memory. Atomic operations on shared memory are significantly faster than on global memory. After writing each block-local histogram back to global memory, we need to aggregate these partial histograms into the final GPU-global histogram. For this, we implement a second kernel function that reads the block-local histograms from memory and performs global atomic operations. To reduce the number of global atomics, we launch enough threads for each to pre-aggregate a fixed-sized group of block-local histograms. The resulting read pattern to the block-local histograms is perfectly warp-aligned because we orchestrate the memory accesses based on the thread-id.

For very skewed distributions, the shared memory atomics on the block-local histogram are under increased pressure. In the most extreme case, all keys processed by a thread block have the same c bits. Thus, all threads try to increase the same bucket counter of the block-local histogram concurrently. To mitigate this performance degradation, we employ the following lightweight optimization: Each thread stores its first key's bucket value and holds back the atomic increment. For every following key, we check if it falls in the same bucket as the first key, and if so, we increment a local variable. After the thread iterated over its keys, we perform the postponed shared atomic increment. With this optimization, our histogram computation is equally fast on skewed and uniform data.

4.2 Key Scattering

In the key scattering step, each GPU locally partitions its keys based on the computed histogram, i. e. based on the considered c bits. Afterward, all keys of bucket i precede those of bucket $i + 1$. To avoid synchronization between reading from and writing to the same memory buffer with many threads, we perform the key scattering step out-of-place. We allocate two alternating input/output buffers. Depending on the bucket, each key needs to be scattered to different locations in global memory. To avoid random write patterns, we pre-scatter all the keys of a thread block into their respective buckets in shared memory. This allows each thread block to write its buckets back to global memory one after another. As a result, the write pattern is sequential for keys of the same bucket. We launch the same number of threads and thread blocks for our `ScatterKeys` kernel as for the histogram computation. Thus, we can re-use the block-local histograms. For both, the pre-scattering and the global write-back, we know the write position for each key of a bucket. In both cases, we determine the write positions by computing the prefix sum on the corresponding histogram, i. e. either the GPU-global or a block-local one.

Shared Memory Pre-Scattering. Each thread block, scheduled to one SM, is responsible for pre-scattering its share of keys into its shared memory buffer. First, the thread block loads its block-local histogram and computes the prefix sum on it. This gives us the starting write position w_{sh} for each bucket b, i. e. the write position for the first key of b. Then, each thread iterates over its assigned keys, determining their buckets. For a key of bucket b, we performs an `atomicAdd` operation that reads $w_{sh}[b]$ and adds 1. We then scatter the key into the shared memory buffer at the position that we read from $w_{sh}[b]$. Thus, we ensure that any subsequent writes by another or the same thread will be performed on the incremented offset, avoiding write conflicts. We deliberately perform significantly more atomic operations in shared memory than in global memory because they are substantially faster in shared memory. The limited shared memory size (128 KB on the Tesla V100) sets an upper bound for the number of keys that each thread block can pre-scatter. We configure our implementation to process twelve 32-bit keys per thread, and each tread block to run 1024 threads. For 64-bit keys, each thread processes six keys. This results in a shared memory usage of 49.152 KB, leaving enough memory to be used as the L1 cache.

Global Memory Write-Back. We compute the prefix sum on the GPU-global histogram before calling the `ScatterKeys` kernel. It returns the starting write position to the global memory output buffer w_{gl} for each complete bucket. Since all thread blocks concurrently write their share of keys back to global memory, we pre-determine the exact global memory buffer spaces that each thread block writes to. For this, every thread block atomically increments $w_{gl}[b]$ by the number of keys it will write per bucket b. This ensures that the global memory write-back phase needs no further synchronization. The pre-scattering step enables sequential writes. But to achieve peak memory throughput, the write pattern needs to be warp-aligned. We implement each thread warp to be responsible for writing a small, constant number of consecutive buckets, one after the other. Each thread of a warp writes one of 32 consecutive keys of the warp's current bucket, iterating over the bucket's keys with a stride of 32 (= thread warp size [NV22a]). With this approach, the bucket size can negatively influence the memory throughput. If many buckets contained very few keys, the memory throughput would drop considerably as many threads idle. It is desirable that all non-empty buckets contain enough keys to fill at least one thread warp memory transaction. We cannot increase the number of keys per bucket by processing more keys per thread block because the shared memory size is limited. Rather, the number of consecutive bits c considered per partitioning pass influences how many keys can fall into a bucket. For example, if we chose c=16, the number of possible buckets 2^{16} would be higher than the number of keys one thread block processes. This would drastically reduce the memory throughput since most of the buckets would contain very few or no keys, assuming uniform distributions. If c is too small, we increase the number of partitioning passes, ultimately increasing the total sort duration. We configure c=8 as an ideal trade-off between minimizing the number of partitioning passes and maximizing the throughput. Thus, we confirm the findings of Stehle et al. for their single-GPU MSB radix sort [SJ17]. We measure a global write-back throughput of 70-95% of the A100 GPU's peak memory bandwidth.

5 Evaluation

In this section, we compare the performance of RMG sort to highly parallel CPU algorithms (Section 5.1), and two state-of-the-art merge-based multi-GPU sorting algorithms (Section 5.2). We analyze RMG sort for varying distributions and data types in Section 5.3.

Experimental Setup. We evaluate our algorithm on two multi-GPU platforms with state-of-the-art interconnects. We provide system information in Table 1. The IBM Power AC922 is a two-socket system that attaches two NVIDIA Tesla V100 GPUs to each NUMA node [NSH18] (Figure 3, bandwidth per direction). On the IBM AC922, NVLink 2.0 accelerates data transfers between a NUMA node and its two GPUs. Our second system is the DGX A100 [NV21b]. It connects eight NVIDIA A100 GPUs with NVLink 3.0-based NVSwitch for high-speed transfers between all GPUs at 300 GB/s per direction (Figure 4).

Tab. 1: Hardware systems overview

	(a) IBM Power System AC922	(b) NVIDIA DGX A100
CPU	2× IBM POWER9 à 16 cores 2.7 GHz	2x AMD EPYC 7742 à 64 cores 2.3 GHz
GPUs	4× NVIDIA Tesla V100 SXM2 32 GB	8× NVIDIA A100 SXM4 40 GB
RAM	2× 256 GB DDR4	2× 512 GB DDR4
Tools	CUDA 11.2, GCC 10.2.1	CUDA 11.4, GCC 9.3.0

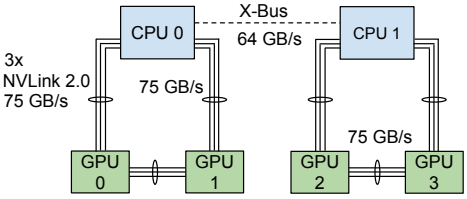


Fig. 3: IBM AC922 topology

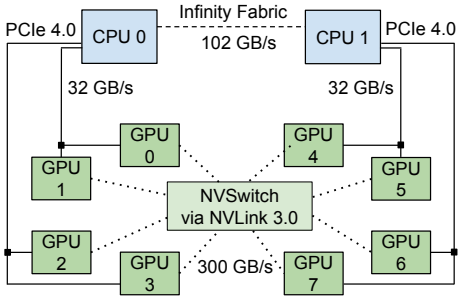


Fig. 4: NVIDIA DGX A100 topology

For all experiments, we measure the end-to-end sort duration which includes the data transfer times between CPU and GPUs. We run every experiment five times and report the arithmetic mean. Our experiments results are stable with a standard error across all runs of less than 4% from the mean. We assume that the input data is not distributed perfectly among the NUMA nodes. Instead, it lies in the main memory of NUMA node 0 only. An optimized NUMA strategy would benefit all three evaluated multi-GPU sorting algorithms equally, and is to consider in future work. We pre-allocate the GPU memory and the pinned host memory because we assume exclusively reserved accelerators. We publish the source code of RMG sort with benchmark scripts to automatically run and plot the experiments.

Optimal GPU Sets. Given a fixed number of GPUs g with $g \in \{1, ..., g_{max}\}$, the interconnect topology determines which exact g GPUs achieve the fastest sorting execution. For instance, when using a P2P-based algorithm on the IBM AC922, the optimal two-GPU-set is the

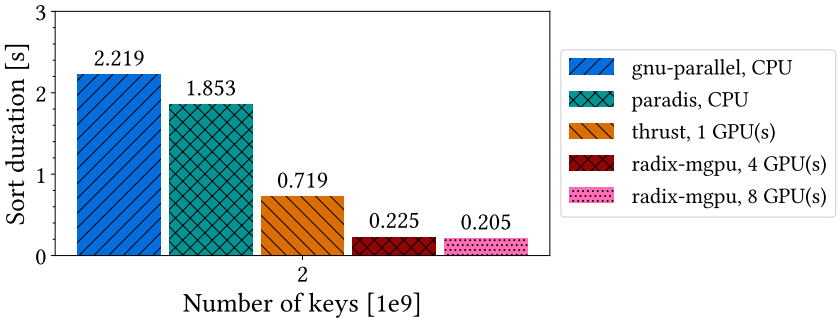


Fig. 5: Sorting performance: CPU vs. GPU(s) on the DGX A100

CPU-local GPU pair $(0, 1)$ (for NUMA node 0). The optimal four-GPU-set on the DGX A100 is $(0, 2, 4, 6)$ as it includes only one GPU of each pair that shares a PCIe switch. Across our evaluation, we depict the performance for optimal GPU sets.

Baselines. To compare the performance of RMG sort to that of the CPU, we use the state-of-the-art parallel CPU radix sort PARADIS as our baseline [Ch15]. We add the parallel multiway merge sort from the GNU parallel algorithms as our second CPU baseline [FS21]. To compare the performance of multiple GPUs against one, we use the same primitive as in RMG sort, namely `cub::DeviceRadixSort` [NV21a]. This LSB radix sort is, to the best of our knowledge, the fastest single-GPU sorting algorithm [Ma22]. As such, it is integrated into NVIDIA's Thrust library as the standard sorting method `thrust::sort` [NV21d].

Memory consumption. Since CUB's device radix sort works out-of-place, RMG sort and the two multi-GPU sorts we compare to have a memory consumption of at least $2n$ for n input keys. For RMG sort, we additionally store histograms and bucket mapping tables. The additional memory overhead of RMG sort depends on the upper bound of spanning buckets, and thus increases with the number of GPUs and partitioning passes. In our implementation, sorting 16 billion 32-bit keys with eight GPUs requires an additional memory overhead of 3.7 GB – 22% of the $2n$ memory overhead (=16 GB). When sorting 8B keys on four GPUs, the additional overhead is 11%. However, as part of future work, we want to improve our implementation and reduce the additional overhead significantly by re-using buffers in each partitioning pass. Then, the additional memory overhead only depends on the number of GPUs. In the above cases, it would constitute 7%, and 3.5%, respectively.

5.1 CPU Comparison

First, we compare the performance of RMG sort to the CPU. We sort two billion uniformly distributed 32-bit integer keys with our proposed RMG sort, and our single-GPU and CPU baselines. We depict the results for the NVIDIA DGX A100 in Figure 5. The system's optimal four-GPU-set is $(0, 2, 4, 6)$. We observe that one GPU achieves a speedup of 3×

over `gnu-parallel` and 2.6× over PARADIS. Given that radix sort algorithms are memory-bandwidth-bound, one main reason why the GPU outperforms the CPU is the GPU's substantially higher memory bandwidth. Compared to `gnu-parallel`, RMG sort is 9.8× faster with four GPUs and 10.8× with eight. Also, RMG sort outperforms PARADIS 8× with four GPUs, and 9× with eight. Overall, eight GPUs achieve the best performance. On the IBM AC922, we measure that RMG sort outperforms the CPU up to 20×. The speedup is higher than on the DGX A100 because the POWER9 CPU has 4× fewer cores than the AMD EPYC 7742 and the IBM AC922 achieves faster CPU-GPU copies with NVLink 2.0.

5.2 Radix-Partitioning vs. Sort-Merge

In this section, we compare the performance of RMG sort to two state-of-the-art merge-based multi-GPU sorting algorithms, the **P2P merge sort** by Tanasic et al. [Ta13], and the heterogeneous merge sort (**HET merge sort**) by Gowanlock et al. [GK18]. P2P merge sort sorts and merges on multiple GPUs using P2P interconnects. By selecting a pivot within the sorted chunks of a GPU pair, blocks of keys are swapped so that the first GPU contains keys smaller than or equal to the keys of the second GPU. Merging the two blocks of keys on each GPU locally brings the data across both GPUs into globally sorted order. The algorithm sorts on more than two GPUs using many subsequent P2P key swaps and GPU-local merge steps. The number of P2P transfers scales linearly with the number of GPUs g. P2P merge sort can only sort on $g = 2^k$ GPUs, $k \in \mathbb{N}$. RMG sort runs on any number of GPUs. HET merge sort uses a parallel multiway merge algorithm on the CPU to merge chunks that the GPUs have sorted, and is not limited by the combined GPU memory capacity. Since RMG sort only sorts data that fits onto the GPUs, we disregard the evaluation of out-of-core data.

To evaluate the performance differences between RMG sort and the merge-based algorithms, we break down the total sort duration of each algorithm into phases. All three algorithms start with the host-to-device (HtoD) copy. For RMG sort, the remaining phases are the radix partitioning, the P2P key swap, and the bucket sorting phase which is interleaved with copying the buckets back to the host. For P2P merge sort, we analyze the HtoD copy, the sort phase, the P2P merge phase on the GPUs, and the device-to-host copy (DtoH). In its sort phase, each GPU chunk is sorted using CUB's single-GPU radix sort. HET merge sort entails the same phases as P2P merge sort except for its CPU-based multiway merge.

Sort Duration Breakdown – IBM AC922. In Figure 6, we depict the sort duration breakdown for RMG sort, P2P merge sort, and HET merge sort for 2B (two billion) uniformly distributed 32-bit integer keys on the IBM AC922. We depict the results for the single-GPU baseline, the GPU pair $(0, 1)$, and all four GPUs. Both RMG sort and P2P merge sort exchange keys via the P2P interconnects. Since the merge phase of P2P merge sort is bound by the P2P key swaps and not the GPU-local merge steps, we display the P2P merge phase using the same plot label and bar pattern as for RMG sort's P2P key swap. We observe that the radix partitioning achieves the shortest duration out of all algorithm phases, scaling linearly to the number of keys, i. e. the GPU chunk size. On two GPUs it makes

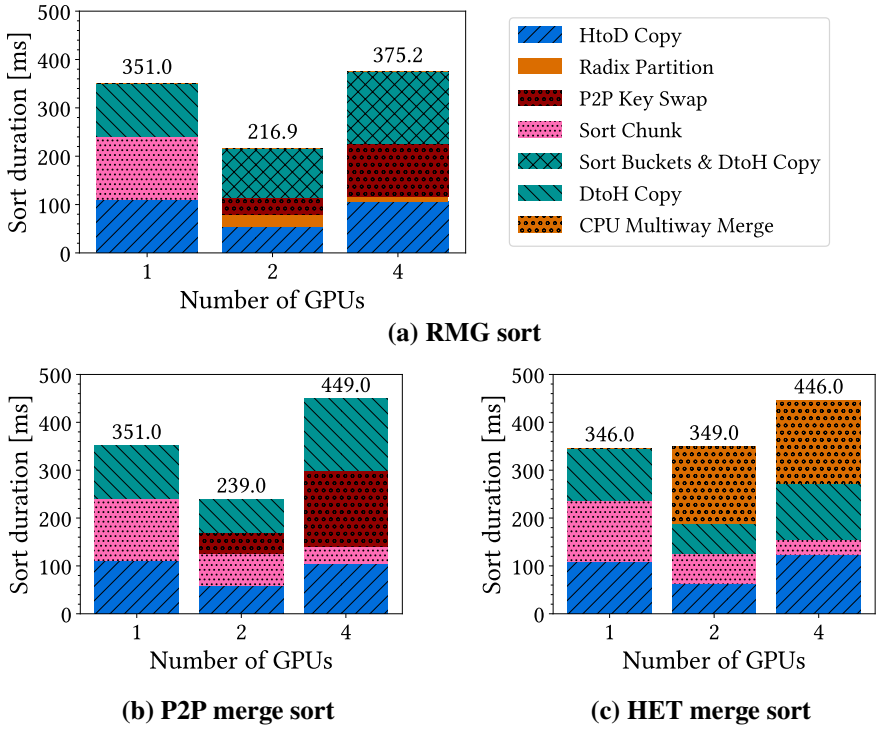


Fig. 6: Sort duration breakdown: Sorting two billion integer keys on the IBM AC922

up 11% of RMG sort's total sort time with 22.8ms, while it takes four GPUs 11.4ms. For the GPU pair (0, 1), the second shortest time duration is the P2P key swap. Powered by a bandwidth of 75 GB/s, the P2P bucket exchange takes 36ms (16% of the total sort duration). While the HtoD copy is halved compared to the single-GPU baseline, the *Sort Buckets & DtoH Copy* phase makes up 47% of the total sort duration. This is because the DtoH copy throughput drops by 30% compared to HtoD copies for parallel transfers from multiple GPUs to the same NUMA node [Ma22]. In addition to this hardware anomaly, we measure that the sort-copy-overlap does not perfectly hide the sorting time. Still, overlapping the sorting computation with the DtoH copy saves 50% of the sorting time (=32ms) on two GPUs. This explains why RMG sort achieves a speedup of 1.6× with two GPUs over one.

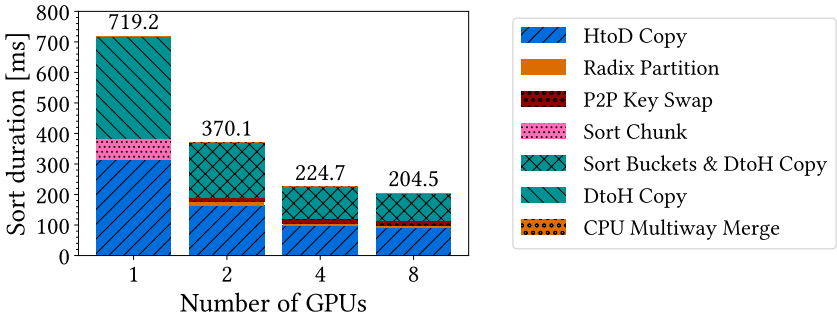
Figure 6a also shows why four GPUs perform worse than two on the IBM AC922. The CPU-interconnect is the system's transfer bottleneck, as the X-Bus reaches only 41 GB/s of the theoretical peak bandwidth of 64 GB/s [Pe19, Ma22]. Thus, the slow CPU-GPU copies on the remote GPUs 2 and 3 slow down the execution. Also, the P2P throughput suffers from the low X-Bus bandwidth, as the P2P key swap takes 3× longer on four GPUs than on two. As a result, RMG sort performs 7% slower on four GPUs compared to one GPU. In Figure 6b, we observe that the number of P2P key swaps of P2P merge sort scales linearly

with the number of GPUs. On two GPUs, P2P merge sort requires only a single key swap, similar to RMG sort, which results in nearly identical transfer times. However, since we overlap the sorting computation with the DtoH copy, RMG sort outperforms P2P merge sort on two GPUs by 11%. When comparing RMG sort with HET merge sort (see Figure 6c), we observe that the CPU multiway merge is significantly slower compared to our on-GPU partitioning. On two GPUs, it takes RMG sort 2.7× less time to partition and swap the keys than it takes the CPU to merge the two sorted chunks. On four GPUs, the P2P throughput of RMG sort is significantly reduced. Still, the radix partitioning and the P2P key swap phase combine for a time duration that is 44% lower than that of the CPU merge. In total, RMG sort outperforms HET merge sort 1.6× on two, and 1.2× on four GPUs for 2B keys.

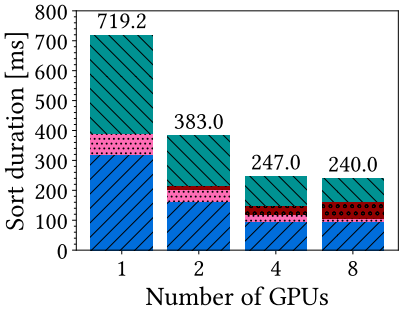
We conclude that on GPUs with high-bandwidth P2P interconnects, GPU-only approaches like RMG sort and P2P merge sort are superior to the CPU-based merge. RMG sort reduces the inter-GPU communication compared to P2P merge sort only for $g > 2$. On this system, where two NUMA-local GPUs are optimal, we outperform P2P merge sort because we overlap the sorting computation with the DtoH copy – an optimization of our MSB radix partitioning. P2P merge sort waits for the last key to be merged before the DtoH copy.
Sorting Large Data – IBM AC922. We observe that the speedup of RMG sort over P2P merge sort increases with larger inputs. RMG outperforms P2P merge sort by 11% for 2B keys, and by 17% for 4B keys. We outperform HET merge sort 1.6× for 2B keys, and 1.7× for 4B keys. Sorting 12B integer keys (48 GB) with four GPUs on the IBM AC922, RMG sort outperforms HET merge sort by 2.1×. P2P merge sort cannot sort more than 2^{32} keys per GPU due to an input size limitation in its on-GPU merge implementation.

Sort Duration Breakdown – DGX A100. Figure 7 depicts the sort duration breakdown sorting 2B (two billion) uniformly distributed keys on the DGX A100. We evaluate the GPU sets $(0, 2)$, $(0, 2, 4, 6)$, and all eight. First, we note that the HtoD and DtoH copies take significantly longer on this system than on the IBM AC922 due to the comparatively low PCIe 4.0 bandwidth. For RMG sort, the CPU-GPU transfers make up 90% of the end-to-end duration. In Figure 7a, we observe that our partitioning phase scales well to increasing numbers of GPUs. It takes 14.4ms on two GPUs, 7.2ms on four GPUs, and 3.6ms on eight GPUs, which constitutes 4%, 3%, and 2% of the total sort duration, respectively. The P2P key swap time stays constant, independent of the number of GPUs. Given the high bandwidth of NVLink 3.0. We measure 14-17ms for two, four and eight GPUs (less than 8% of the total sorting time). The P2P swap time is not reduced when increasing the number of GPUs even though each GPU copies less data. Even though NVSwitch does achieve simultaneous all-to-all transfers at high throughput, P2P transfers between individual pairs of GPUs tend to perform best. Similar to our partitioning phase, the time of the sorting computation gets halved when we double the number of GPUs. However, the sorting time on the A100 GPU is insignificant compared to the HtoD/DtoH copies. Thus, the sort-copy-overlap does not notably improve the end-to-end performance on this system.

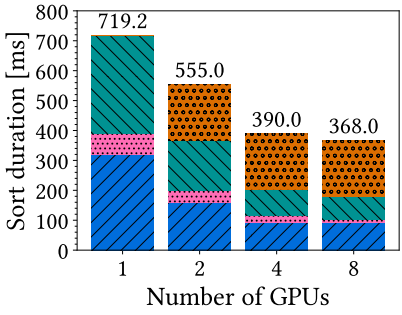
Despite the CPU-GPU copy bottleneck on the DGX A100, we observe RMG sort to scale comparatively well from one to eight GPUs. Compared to a single GPU, we achieve



(a) RMG sort



(b) P2P merge sort



(c) HET merge sort

Fig. 7: Sort duration breakdown: Sorting two billion integer keys on the DGX A100

speedups of 1.9× with two, 3.2× with four, and 3.5× with eight GPUs. We cannot expect much higher speedups on eight GPUs because of the shared bandwidth effects that result from the system's limited number of PCIe switches. As seen in Figure 4, neighbouring pairs of GPUs share a PCIe switch. For parallel CPU-GPU transfers, the throughput for neighbouring GPU pairs cannot exceed the 32 GB/s of one PCIe 4.0 instance. This hardware limitation negatively influences any multi-GPU sorting algorithm, not just RMG sort. In Figure 7b, we again see that the P2P merge phase time of P2P merge sort increases with the number of GPUs. We measure it to take almost 4× longer when eight GPUs merge their chunks compared to two. This explains why RMG sort's speedup factor over P2P merge sort increases: 3% with g=2, 10% with g=4, and 17% with g=8. When RMG sort uses eight GPUs, the radix partitioning phase and the P2P key swap take 20ms, which is 2.7× less than the P2P merge phase takes. In Figure 7c, we again observe the CPU merge as the limiting factor of HET merge sort. Compared to the combined duration of RMG sort's partitioning phase and its P2P swap on two, four, and eight GPUs, the CPU merge takes 6.6-9.2× longer. In total, RMG sort outperforms HET merge sort up to 1.8× on eight GPUs. Thus, if we compare the execution times for the on-GPU (or on-CPU) computation and the P2P transfers only, i. e. excluding the HtoD and DtoH copies, RMG sort outperforms

Tab. 2: Sorting performance by data distribution (2 billion keys, 8 GPUs, DGX A100)

	zero	sorted	nearly-sorted	reverse-sorted	uniform	normal
RMG sort	182ms	190ms	192ms	193ms	205ms	215ms
P2P merge sort	182ms	189ms	200ms	250ms	240ms	243ms

P2P merge sort 2.7×, and HET merge sort 9.2× with eight GPUs. The slow HtoD and DtoH copies reduce the total end-to-end speedup. Still, we demonstrate the potential speedup that RMG sort could achieve if the system included high-speed CPU-GPU interconnects. For the DGX A100, we confirm that P2P-based multi-GPU approaches sort significantly faster than the heterogeneous strategies. We conclude that, compared to P2P merge sort, RMG sort more efficiently utilizes the non-blocking all-to-all P2P transfer capability of NVLink-based NVSwitch. RMG sort scales linearly with the number of GPUs g in the radix partitioning phase and keeps a constant P2P key swap time, independent of g.

Sorting Large Data – DGX A100. We compare the performance of the three multi-GPU sorts for increasing input sizes (up to 16B keys) on eight GPUs on the DGX A100. Compared to P2P merge sort, RMG sort is faster up to 1.3× while outperforming HET merge sort up to 1.8×. For 32B integer keys (128 GB), RMG sort takes about 3 seconds, which is 1.85× faster than HET merge sort. Including the data transfers, RMG sort achieves an end-to-end sorting rate of over 10 billion keys per second and scales linearly to larger input sets.

5.3 Sorting Performance By Data Distributions and Data Types

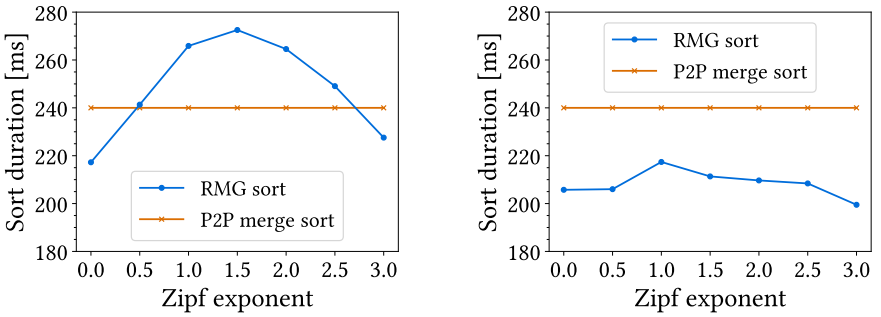
Varying Data Distributions. In Table 2, we compare the sorting time of RMG sort to that of P2P merge sort for varying distributions on the DGX A100 for eight GPUs. HET merge sort is stable across these distributions and purely bound by the main memory bandwidth. We sort 32-bit unsigned integers and find that RMG sort performs better on sorted (13%), nearly-sorted (8%), reverse-sorted (7%), and zero entropy (6%) distributions than for uniform ones. For the zero entropy distribution (i. e. all keys are the same), and already sorted data, RMG sort skips the P2P key swap. Nearly-sorted distributions require almost no key swaps. Additionally, for zero entropy data, we skip sorting the buckets as each one is a last-pass spanning bucket. During the partitioning, we computed the histograms p times, considered all k bits, but never scattered the keys. RMG sort is quick for read-intensive workloads. In the end, P2P merge sort and RMG sort are equally fast for zero and sorted data, while RMG sort is slightly faster on nearly-sorted keys. Reverse-sorted distributions benefit RMG sort as the keys are exchanged only between pairs of mirrored GPUs. We measure that this copy pattern achieves a higher P2P throughput ($1.3 - 4.7\times$). Normal distributions require two partitioning passes. Overall, RMG sort outperforms P2P merge sort for reverse-sorted (30%), uniform (17%), and normal distributions (13%).

Sorting Skewed Data. In this section, we evaluate the performance of RMG sort for Zipfian distributions. For increasing Zipf exponents z, the probability of a key being one of only a

few highly frequent key values increases. Given two billion keys and z=1.0, the probability that a key is one of the top-1000 most occurring keys is 34%. The same probability is at 97.5% for z=1.5. In Figure 8a, we depict the sort duration of RMG sort for varying z. We sort 2B 32-bit keys on two GPUs on the IBM AC922. The sort duration increases for $z > 0$. At the peak at z=1.5, it reaches 273ms, (+26% over the sorting time of the uniform distribution). For $z \geq 1.5$, the sort duration steadily decreases down to the initial duration.

For z=0.5, one pass completes the partitioning phase. We measure average execution times for the key scattering and the histogram computation. However, the number of buckets on the second GPU is greater than our threshold $\text{MAX}_{BRS} = 128$, despite our optimization to fuse small neighbouring buckets. For more than MAX_{BRS} buckets, we sort the entire GPU chunk instead of individual buckets to avoid too many kernel launches. As a result, we cannot overlap the sorting with the DtoH copy, which explains why the total time increases by 30ms. For $0.5 < z \leq 1.5$, fusing neighbouring buckets reduces the total number of buckets significantly, e. g. from 575 to 90 in some cases. For $z \geq 0.75$, RMG sort performs the sort-copy-overlap at peak throughput and the bottleneck shifts to the radix partitioning.

For $0.5 < z \leq 1.5$, the distribution becomes more skewed, and more spanning buckets need to be resolved. For z=1.0, three partitioning passes are necessary, while the z=1.5 requires all four. Thus, we have little to no sorting computation left after the partitioning phase considered (almost) all bits. However, the duration of multiple `ScatterKeys` kernel executions adds up significantly on this system, i. a. because of the many shared memory atomic conflicts. We measure the time of one `ScatterKeys` kernel execution to increase up to 20ms (+25%). This adds 40-70ms to the total sort duration for $z \in [1.0, 1.5]$. For $z \geq 1.5$, the sort duration decreases as almost all keys belong to the same few buckets, approaching the zero entropy distribution. Then, we increasingly skip the key scattering steps during the partitioning passes because all keys of a GPU chunk belong to the same bucket. Computing the histogram (6ms) takes less than the key scattering (16ms). Also, we do not sort the buckets since we partitioned on all $k = 32$ bits. We evaluate the sort duration of P2P merge



(a) 2 GPUs on the IBM AC922 **(b) 8 GPUs on the DGX A100**

Fig. 8: Sorting performance for skewed data (2 billion keys)

sort and HET merge sort to be independent of the Zipf exponent. In the worst case of z=1.5, RMG sort is up to 13% slower than P2P merge sort, but 1.3× faster than HET merge sort.

In Figure 8b, we depict the sort duration of RMG sort for varying Zipf exponents z on the DGX A100 on eight GPUs. We measure significantly fewer differences in the sort duration for skewed data compared to the IBM AC922. The peak sort duration for z=1.0 constitutes a 6% increase over the sorting time of uniform keys. Again, the time duration increases as the radix partitioning phase needs multiple passes. The low CPU-GPU interconnect bandwidth of the DGX A100 is one reason why the performance impact of the Zipf exponent is less significant on this system. Also, distributing the buckets across eight instead of two GPUs results in fewer buckets per GPU. Thus, the number of buckets per GPU is less likely to exceed MAX_{BRS}. For RMG sort, scaling up g reduces the negative performance impact of data skew. Moreover, the accumulated execution times for all histogram computations and key scattering steps are less than on the AC922 because 1) the NVIDIA A100 GPU has a higher global memory bandwidth and faster atomic operations, and 2) each GPU processes fewer keys. For skewed data on the eight GPUs of the DGX A100, RMG sort outperforms P2P merge sort by at least 11% and up to 1.3×, and HET merge sort by 1.7-1.8×.

Sorting Different Data Types. We evaluate RMG sort's performance for unsigned integer and floating-point keys in their 32-bit and 64-bit variant. We sort unsigned key values, i. e. unsigned integer types, and positive floating-point numbers. Extending the algorithm to support negative value ranges is possible without sacrificing performance [Te00]. We sort uniformly distributed integers, and floating-point keys whose values follow a Zipfian distribution with an exponent of 1.0. In that way, the k bits of the floating-point keys are distributed similarly to the uniform integer distribution. When sorting 2 billion keys with two GPUs on the IBM AC922, the sort duration of 32-bit integers is approximately the same as for 32-bit floats. This is expected given that RMG sort's time duration depends on the number of bits per key. However, on the IBM AC922, the sort duration of 64-bit data types is 2.2× higher than for the same number of 32-bit keys. This is the case for all three multi-GPU sorting algorithms, and confirms the findings of Maltenberger et al. [Ma22]. For the same experiment on the DGX A100, sorting 64-bit data types takes exactly 2× longer.

6 Related Work

Various single-GPU sorting algorithms have been proposed [Ba20, Ca17, DZ12, Go06, KW05, LOS10, SHG09, Sa10, SA08]. Ha et al. propose an LSB radix sort algorithm that considers two bits at a time and performs a block-local key shuffle in shared memory to ensure coalesced writes [HKS09]. Merrill et al. design an LSB radix sort algorithm that dynamically adjusts the number of keys a thread processes [MG11]. They also implement an analytical performance model that determines the optimal number of bits per pass, reducing the memory workload for any target architecture. Their approach has been integrated into NVIDIA's high-performance CUB library [NV21a, Ad20]. Stehle et al. publish an MSB radix sort algorithm that increases the number of bits considered at a time to eight [SJ17].

They partition the keys into smaller and smaller buckets until a local sort algorithm sorts the buckets in on-chip memory. These sorting algorithms are single-GPU approaches. We publish a novel multi-GPU sorting algorithm. To the best of our knowledge, all previous multi-GPU sorting algorithms are merge-based. Peters et al. propose a multi-GPU sorting algorithm that sorts large-out-of-core data [PSHL10]. The authors use multiple GPUs to sort chunks. After copying the sorted chunks back to main memory, the CPU finds splitter elements to form disjoint data sets that individual GPUs can merge independently. Gowanlock et al. publish a heterogeneous multi-GPU sorting algorithm for large data where the CPU merges sorted chunks [GK18]. Both these algorithms sort large-out-of-core data, but neither one utilizes inter-GPU communication. Tanasic et al. propose a multi-GPU sorting algorithm for in-memory data that merges chunks using P2P transfers [Ta13].

7 Conclusion

In this paper, we design and evaluate the first radix-partitioning-based multi-GPU sorting algorithm (RMG sort). It outperforms CPU-only algorithms by up to 20×. Our MSB radix partitioning strategy exploits all-to-all P2P interconnects. As a result. RMG sort scales linearly with the input size and reduces the inter-GPU communication as it requires only one all-to-all P2P key swap, independent of g. Our evaluation shows that RMG sort utilizes high-speed P2P interconnects more efficiently than prior work. Compared to two state-of-the-art merge-based multi-GPU sorting algorithms, RMG sort scales best with increasing numbers of GPUs. We outperform P2P merge sort up to 1.3× and HET merge sort up to 1.8×. When we exclude the CPU-GPU copy times to directly compare the on-GPU computation and P2P communication, RMG sort is 2.7× faster than P2P merge sort and 9.2× faster than HET merge sort. Thus, RMG sort benefits from future accelerator platforms given that hardware vendors continue to increase the number of GPUs, and the P2P and CPU-GPU interconnect bandwidth [NV22c, NV21c, NV22d].

RMG sort can improve the performance of an existing out-of-core sort-merge algorithm, reducing the number of chunks that need to be merged. Alternatively, we suggest extending RMG sort by a preliminary partitioning step on the CPU which divides the input into chunks of distinct value ranges that fit into the combined GPU memory. Then, out-of-core data sets are sorted in independent sorting rounds. In future work, we want to analyze RMG sort's performance as part of a real-world database use case. RMG sort can be extended to support key-value pairs. For each key permutation, we use the block-local histograms to rearrange the corresponding values equivalently in the key scattering step.

Acknowledgments
The authors would like to thank Elias Stehle for his input throughout the algorithm design. This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), and the European Union's Horizon 2020 research and innovation program (ref. 957407).

Bibliography

[Ad20] Adinets, A.: , A Faster Radix Sort Implementation. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21572-a-faster-radix-sort-implementation.pdf, October 2020. Last accessed: 2022-04-26.

[Ag96] Agarwal, Ramesh C.: A Super Scalar Sort Algorithm for RISC Processors. In: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, p. 240–246, 1996.

[AM18] AMD: , AMD Radeon Instinct MI60: Unleash Discovery on the World's Fastest Double Precision PCIe Accelerator. https://www.amd.com/system/files/documents/radeon-instinct-mi60-datasheet.pdf, November 2018. Last accessed: 2022-04-26.

[AM20] AMD: , Introducing AMD CDNA Architecture: The All-New AMD GPU Architecture for the Modern Era of HPC and AI (Whitepaper). https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf, December 2020. Last accessed: 2022-04-26.

[Ba20] Baxter, Sean: , Modern GPU: Patterns and Behaviors for GPU Computing. https://github.com/moderngpu/moderngpu, January 2020. Last accessed: 2022-04-26.

[Ca17] Casanova, Henri; Iacono, John; Karsin, Ben; Sitchinava, Nodari; Weichert, Volker: An Efficient Multiway Mergesort for GPU Architectures. Technical report, arXiv:cs.DS, February 2017.

[Ch15] Cho, M.; Brand, D.; Bordawekar, R.; Finkler, U.; Kulandaisamy, V.; Puri, R.: PARADIS: An Efficient Parallel Algorithm for In-Place Radix Sort. Proc. VLDB Endow., 8(12):1518–1529, August 2015.

[CI18] Colgan, Maria; Insider, Oracle Database: , Does GPU Hardware Help Database Workloads? https://blogs.oracle.com/database/post/does-gpu-hardware-help-database-workloads, February 2018. Last accessed: 2022-04-26.

[DZ12] Dehne, Frank; Zaboli, Hamidreza: Deterministic Sample Sort for GPUs. Parallel Processing Letters, 22(3):1–14, September 2012.

[FS21] FSF: , The GNU C++ Library Manual: Parallel Mode. https://gcc.gnu.org/onlinedocs/gcc-11.2.0/libstdc++/manual/manual/parallel_mode.html, July 2021. Last accessed: 2022-04-26.

[Gi19] Gill, Sandeep Kaur; Singh, Virendra Pal; Sharma, Pankaj; Kumar, Durgesh: A Comparative Study of Various Sorting Algorithms. International Journal of Advanced Studies of Scientific Research, 4(1), February 2019.

[GK18] Gowanlock, Michael; Karsin, Ben: 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, pp. 560–569, 2018.

[Go06] Govindaraju, Naga; Gray, Jim; Kumar, Ritesh; Manocha, Dinesh: GPUTeraSort: High Performance Graphics Co-Processor Sorting for Large Database Management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. SIGMOD '06. Association for Computing Machinery, p. 325–336, 2006.

[Gr06] Graefe, Goetz: Implementing Sorting in Database Systems. ACM Comput. Surv., 38(3):1–37, September 2006.

[Gu15] Gupta, Anurag; Agarwal, Deepak; Tan, Derek; Kulesza, Jakub; Pathak, Rahul; Stefani, Stefano; Srinivasan, Vidhya: Amazon Redshift and the Case for Simpler Data Warehouses. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, p. 1917–1923, 2015.

[HKS09] Ha, Linh; Krueger, Jens; Silva, Claudio T.: Fast Four-Way Parallel Radix Sorting on GPUs. Computer Graphics Forum, 2009.

[Ja14] Jagadish, H. V.; Gehrke, Johannes; Labrinidis, Alexandros; Papakonstantinou, Yannis; Patel, Jignesh M.; Ramakrishnan, Raghu; Shahabi, Cyrus: Big Data and Its Technical Challenges. Commun. ACM, 57(7):86–94, July 2014.

[KW05] Kipfer, Peter; Westermann, Rüdiger: , Chapter 46. Improved GPU Sorting. https://developer.nvidia.com/gpugems/gpugems2/part-vi-simulation-and-numerical-algorithms/chapter-46-improved-gpu-sorting, April 2005. Last accessed: 2022-04-26.

[Li20] Li, Ang; Song, Shuaiwen Leon; Chen, Jieyang; Li, Jiajia; Liu, Xu; Tallent, Nathan R.; Barker, Kevin J.: Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE Transactions on Parallel and Distributed Systems (TPDS), 31(1):94–110, January 2020.

[LOS10] Leischner, Nikolaj; Osipov, Vitaly; Sanders, Peter: GPU Sample Sort. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS). pp. 1–10, 2010.

[Lu20] Lutz, Clemens; Breß, Sebastian; Zeuch, Steffen; Rabl, Tilmann; Markl, Volker: 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. Association for Computing Machinery, pp. 1633–1649, 2020.

[Ma22] Maltenberger, Tobias; Ilic, Ivan; Tolovski, Ilin; Rabl, Tilmann: Evaluating Multi-GPU Sorting with Modern Interconnects. In: 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22). Association for Computing Machinery, 2022.

[MG11] Merrill, Duane; Grimshaw, Andrew: High Performance and Scalable Radix Sorting: A Case Study of Implementing Dynamic Parallelism for GPU Computing. Parallel Processing Letters, 21(2):245–272, June 2011.

[MG16] Merrill, Duane; Garland, Michael: Single-Pass Parallel Prefix Scan with Decoupled Look-Back. Technical report, NVIDIA, August 2016. https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf.

[NSH18] Nohria, Ritesh; Santos, Gustavo; Haug, Volker: , IBM Power System AC922: Technical Overview and Introduction. https://www.redbooks.ibm.com/redpapers/pdfs/redp5494.pdf, July 2018. Last accessed: 2022-04-26.

[NV17] NVIDIA: , NVIDIA Tesla V100 GPU Architecture. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017. Last accessed: 2022-04-26.

[NV18] NVIDIA: , Technical Overview NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch. http://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf, April 2018. Last accessed: 2022-04-26.

[NV20] NVIDIA: , NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, September 2020. Last accessed: 2022-04-26.

[NV21a] NVIDIA: , CUB: Cooperative Primitives for CUDA C++. https://github.com/NVIDIA/cub, June 2021. Last accessed: 2022-04-26.

[NV21b] NVIDIA: , DGX A100 System User Guide. https://docs.nvidia.com/dgx/pdf/dgxa100-user-guide.pdf, November 2021. Last accessed: 2022-04-26.

[NV21c] NVIDIA: , NVIDIA Grace CPU. https://www.nvidia.com/en-us/data-center/grace-cpu/, April 2021. Last accessed: 2022-04-26.

[NV21d] NVIDIA: , Thrust: Code at the Speed of Light. https://github.com/NVIDIA/thrust, June 2021. Last accessed: 2022-04-26.

[NV22a] NVIDIA: , CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, January 2022. Last accessed: 2022-04-26.

[NV22b] NVIDIA: , CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, January 2022. Last accessed: 2022-04-26.

[NV22c] NVIDIA: , NVIDIA DGX H100: The Gold Standard for AI Infrastructure. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-h100-datasheet.pdf, March 2022. Last accessed: 2022-04-26.

[NV22d] NVIDIA: , NVIDIA HGX AI Supercomputer: The most powerful end-to-end AI supercomputing platform. https://www.nvidia.com/en-us/data-center/hgx/, January 2022. Last accessed: 2022-04-26.

[Pa21] Paul, Johns; Lu, Shengliang; He, Bingsheng; Lau, Chiew Tong: MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In: Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, pp. 1413–1425, 2021.

[Pe19] Pearson, C.; Dakkak, A.; Hashash, S.; Li, C.; Chung, I.-H.; Xiong, J.; Hwu, W.-M.: 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, pp. 209–218, 2019.

[PSHL10] Peters, Hagen; Schulz-Hildebrandt, Ole; Luttenberger, Norbert: 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, pp. 1–8, 2010.

[Ra20] Raza, Aunn; Chrysogelos, Periklis; Sioulas, Panagiotis; Indjic, Vladimir; Anadiotis, Angelos Christos; Ailamaki, Anastasia: GPU-Accelerated Data Management under the Test of Time. Online proceedings of the 10th Conference on Innovative Data Systems Research (CIDR), pp. 1–11, 2020.

[RLT20] Rui, Ran; Li, Hao; Tu, Yi-Cheng: Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. Proc. VLDB Endow., 14(4):708–720, December 2020.

[SA08] Sintorn, Erik; Assarsson, Ulf: Fast Parallel GPU-Sorting Using a Hybrid Algorithm. Journal of Parallel and Distributed Computing, 68(10):1381–1388, October 2008.

[Sa10] Satish, Nadathur; Kim, Changkyu; Chhugani, Jatin; Nguyen, Anthony D.; Lee, Victor W.; Kim, Daehyun; Dubey, Pradeep: 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. Association for Computing Machinery, pp. 351–362, 2010.

[SHG09] Satish, Nadathur; Harris, Mark; Garland, Michael: Designing Efficient Sorting Algorithms for Manycore GPUs. In: 2009 IEEE International Symposium on Parallel Distributed Processing (IPDPS). Institute of Electrical and Electronics Engineers, pp. 1–10, 2009.

[SJ17] Stehle, Elias; Jacobsen, Hans-Arno: A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In: Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data. Association for Computing Machinery, pp. 417–432, 2017.

[SMY20] Shanbhag, Anil; Madden, Samuel; Yu, Xiangyao: A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics (Extended Version). Technical report, Massachusetts Institute of Technology, March 2020.

[ST20] Sharma, Debendra Das; Tavallaei, Siamak: , Compute Express Link 2.0 White Paper. https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf, November 2020. Last accessed: 2022-04-26.

[Ta13] Tanasic, Ivan; Vilanova, Lluís; Jordà, Marc; Cabezas, Javier; Gelado, Isaac; Navarro, Nacho; Hwu, Wen-mei: Comparison Based Sorting for Systems with Multiple GPUs. In: Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units. Association for Computing Machinery, pp. 1–11, 2013.

[Te00] Terdiman, Pierre: , Radix Sort Revisited. http://www.codercorner.com/RadixSortRevisited.htm, January 2000. Last accessed: 2022-04-26.

[ZB91] Zagha, Marco; Blelloch, Guy E.: Radix Sort for Vector Multiprocessors. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. Supercomputing '91. Association for Computing Machinery, p. 712–721, 1991.

[ZW12] Zhang, Keliang; Wu, Baifeng: A Novel Parallel Approach of Radix Sort with Bucket Partition Preprocess. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication, 2012 IEEE 9th International Conference on Embedded Software and Systems. pp. 989–994, 2012.