Hasso Plattner Institute
Data Engineering Systems Group

Master Thesis

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

RMG Sort: Ein Multi-GPU Sortierverfahren basierend auf Radix-Partitionierung

Ivan Ilić

Matriculation Number: 786169

Supervisor
Prof. Dr. Tilmann Rabl

2^{nd} **Reviewer**
Prof. Dr. Andreas Polze

Advisor
Ilin Tolovski

Submitted: 01.05.2022

Abstract

Sorting is a fundamental database operation with use cases in index creation, duplicate removal, grouping, and sort-merge joins. In recent years, Graphics Processing Units (GPUs) emerged as database accelerators due to their massive parallelism and high-bandwidth memory. Many single-GPU sorting algorithms have been proposed and shown to outperform highly parallel CPU algorithms. Today's accelerator platforms include multiple GPUs with direct high-bandwidth Peer-to-Peer (P2P) interconnects. However, the few published multi-GPU sorting algorithms do not efficiently harness the all-to-all P2P transfer capability of modern interconnects, such as NVLink and NVSwitch. All previous multi-GPU sorting algorithms are sort-merge approaches. Their merging workload grows for increasing numbers of GPUs.

In this thesis, we propose a novel radix-partitioning-based multi-GPU sorting algorithm (RMG sort). We present a most-significant-bit (MSB) radix partitioning strategy that efficiently utilizes high-speed P2P interconnects while reducing the inter-GPU communication compared to prior merge-based algorithms. Independent of the number of GPUs, we exchange radix partitions between the GPUs in one all-to-all P2P key swap. We analyze the performance of RMG sort on two modern multi-GPU systems with different interconnect topologies. Our evaluation shows that RMG sort scales well with the number of GPUs. We measure it to outperform highly parallel CPU sorting algorithms up to 20×. Compared to two state-of-the-art merge-based multi-GPU sorting algorithms, we achieve speedups of up to 1.26× and 1.8× across both systems.

Zusammenfassung

Sortierverfahren gehören zu den grundlegenden Operationen eines Datenbankmanagementsystems. Sie finden u.a. in der Indexerstellung, der Duplikateneliminierung, der Gruppierung und bei Sort-Merge Joins Anwendung. Innerhalb der letzten Jahre haben sich Grafikprozessoren (englisch: Graphics Processing Unit, kurz GPU) dank ihrer hohen Parallelität und Speicherbandbreite als Datenbank-Beschleuniger etabliert. Viele Grafikprozessor-basierte Sortierverfahren übertreffen parallele Sortierverfahren auf der CPU. Heutige Hochleistungsserver enthalten mehrere Grafikprozessoren und verknüpfen diese untereinander mit Peer-to-Peer (P2P) GPU-zu-GPU-Verbindungen hoher Bandbreite. Allerdings nutzen die publizierten Multi-GPU Sortieralgorithmen die fortschrittliche Kommunikationsfähigkeit moderner P2P-Verbindungen, wie beispielsweise NVLink und NVSwitch, nicht effizient aus. Alle bisherigen Multi-GPU Sortierverfahren sind Mergesort Algorithmen, deren Verschmelzungsaufwand mit zunehmender Anzahl der GPUs steigt.

In dieser Arbeit präsentieren wir einen neuartigen Multi-GPU Sortieralgorithmus (RMG Sort), der auf Radix-Partitionierung basiert. Wir präsentieren eine Radix-Partitionierungs-Strategie, die mit den Bits der höchsten Stellenwerte beginnt (englisch: most significant bit, kurz MSB). Im Vergleich zu vorherigen Mergesort Algorithmen, reduzieren wir die Kommunikation zwischen den GPUs, und nutzen moderne P2P-Verbindungen effizient aus. Unabhängig von der Anzahl der GPUs mit denen wir sortieren, übertragen wir Partitionen zwischen den GPUs in nur einem vollständig allseitigen P2P Austausch. Unsere Auswertung zeigt, dass RMG Sort sehr gut mit einer steigenden Anzahl an GPUs skaliert. Wir messen dass RMG Sort hochparallele Sortierverfahren auf der CPU mit einem Faktor von bis zu 20× übertrifft. Im Vergleich mit zwei modernen Multi-GPU Mergesort Algorithmen, erreicht RMG Sort eine bis zu 1,26- und 1,8-fach schnellere Ausführung.

Contents

1 Introduction

Today's database systems need to process ever-growing amounts of data. Oftentimes, the data volume exceeds the size that database systems can process and analyze efficiently [21]. This creates an increasing demand for improving the performance of database systems and data processing solutions [28]. To tackle the challenge of processing large data sets in a timely fashion, research and industry steadily adapt to and exploit modern hardware developments, such as multi-core CPUs, heterogeneous processor architectures, and massively-parallel accelerators.

Graphics processing units (GPUs) provide unparalleled computational power via thousands of cores, which are supported by a high-bandwidth memory subsystem [38, 40]. Ever since the introduction of general-purpose computing platforms such as CUDA [45], and OpenCL [20], GPUs have evolved from specialized graphics rendering devices into processors suitable to accelerate any computational task that benefits from highly parallel execution. For compute-intensive tasks on small data sets that reside in GPU memory, GPUs achieve orders of magnitude higher throughput rates compared to the CPU. Thus, they are commonly used as accelerators for deep learning and HPC workloads [59]. However, GPUs experience a slower adoption into the enterprise database management systems market, mainly because of the data transfer bottleneck [10, 32]. For many GPU-based operator implementations, copying the data from main memory to the GPU and back via the PCIe 3.0 interconnect bus has been the limiting factor [18, 32, 34, 55, 56].

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 [3, 39, 60]. They increase the GPU-interconnect bandwidth close to that of main memory, accelerating CPU-to-GPU and Peer-to-Peer (P2P) transfers. On hardware platforms with high-speed interconnects, GPUs have been shown to efficiently accelerate data analytics workloads and core database operations [32, 34, 55]. Sorting is one such database operation. Within database systems, sorting has use cases in index creation, user-specified output ordering, duplicate removal, grouping, and sort-merge joins [19]. Over the past years, numerous single-GPU sorting algorithms have been proposed and shown to outperform highly parallel CPU sorting algorithms by orders of magnitude. Parallel radix sort algorithms have been proven to be best suited for modern GPU architectures, achieving the fastest single-GPU sorting performance [35, 36, 41, 44, 59, 63].

Nowadays, modern server-grade hardware platforms combine multiple GPUs for even higher computing power. Thus, the research community started adapting algorithms to utilize multiple GPUs [56, 50]. To the best of our knowledge, all of

the few published multi-GPU sorting algorithms are sort-merge approaches [18, 52, 56, 64]. The P2P-based multi-GPU merge sort by Tanasic et al. (P2P merge sort) utilizes inter-GPU communication to merge the previously sorted chunks within GPU memory [64]. Gowanlock et al. employ a heterogeneous sorting algorithm (HET merge sort) that uses the CPU to merge data chunks that multiple GPUs sorted. Recently evaluated on modern multi-GPU systems, they show promising speedups over a single GPU [34].

However, the merging workload of merge-based multi-GPU sorting algorithms grows with increasing numbers of GPUs. For HET merge sort, the final merge phase on the CPU quickly becomes a bottleneck because the multiway merge is heavily main memory bandwidth-bound. The CPU's main memory (i.e. the on-chip memory controller) provides significantly less bandwidth than the GPU device memory [18, 34, 42, 63]. For P2P merge sort, scaling up the number of GPUs g results in a linear increase in the number of key swaps issued over the P2P interconnects. This is because Tanasic et al. do not employ an all-to-all exchange of keys between the GPUs. Instead, at any point in time during the merge phase, each GPU swaps data with only one other GPU. Therefore, multiple merge steps are necessary. This algorithm design made sense in a time when GPUs had no dedicated, direct P2P interconnects attached. On such systems, many concurrent P2P transfers over the PCIe 3.0 interconnect tree topology suffer from shared bandwidth effects. Today, modern multi-GPU accelerator platforms incorporate direct high-bandwidth P2P interconnects for exclusive bandwidth usage. Most recent hardware systems allow for full non-blocking all-to-all inter-GPU communication (e.g. via NVLink and NVSwitch) [39, 42]. Given these hardware improvements, the design of a novel multi-GPU sorting algorithm becomes necessary to efficiently harness modern P2P interconnect technology. Thus, we raise the following questions:

1. Can we better utilize the non-blocking all-to-all P2P interconnects of modern multi-GPU accelerator platforms, and reduce inter-GPU communication?

2. How can we design a faster and better scaling multi-GPU sorting algorithm?

1.1 Contribution

In this master's thesis, we propose a novel radix-partitioning-based multi-GPU sorting algorithm (RMG sort). In contrast to previous work, our algorithm exploits P2P interconnects, such as NVLink 2.0, NVLink 3.0, and NVSwitch, to their fullest extent. We reduce inter-GPU communication by exchanging the radix partitions between all GPUs in parallel. In contrast to merge-based algorithms, RMG sort requires one P2P key swap independently of the number of GPUs used. Thus, our approach achieves a more efficient P2P interconnect bandwidth utilization, especially when scaling to increasing numbers of GPUs. We evaluate our sorting algorithm on modern multi-GPU systems with state-of-the-art high-bandwidth interconnects. With this master's thesis, we make the following contributions:

1. We design a novel multi-GPU sorting algorithm (RMG sort). We employ an MSB radix partitioning strategy to exploit modern interconnect technology.

2. We implement our multi-GPU sorting algorithm in the CUDA framework and publish our optimized source code together with automated benchmark scripts to enable reproducible evaluation results.

3. We evaluate the performance of our multi-GPU sorting algorithm for up to eight GPUs and compare it to parallel CPU sorting algorithms as well as two state-of-the-art, merge-based, multi-GPU sorting algorithms.

1.2 Scope and Delimitation

The main focus of this thesis is to design, implement and evaluate a novel multi-GPU sorting algorithm. We focus only on developing a high-performance sorting algorithm, rather than embedding it into a real-world database system. The integration of GPU-accelerated operators into a fully-fledged database system exceeds the scope of this thesis. We compare the performance of our multi-GPU sort with that of state-of-the-art parallel CPU sorting algorithms. Thus, our evaluation indicates the potential speedup that a database system can achieve when using multiple GPUs for sorting.

For simplicity, we only support sorting basic numeric keys and unsigned key values, i.e. unsigned integer types, and positive floating-point numbers. Extending the algorithm to support key-value pairs is a valuable addition for future work. Also, radix sort algorithms can be adapted to support negative value ranges without sacrificing performance [66].

We evaluate our algorithm on hardware systems that include NVIDIA GPUs and implement our algorithm in the CUDA framework. Thus, we only support NVIDIA GPUs. The algorithm could, however, be ported to GPUs of other vendors and their runtime APIs.

1.3 Thesis Outline

The thesis is structured as follows. In Section 2, we explain relevant background information with regards to the GPU hardware architecture, modern interconnect technology, and radix sort algorithms. Section 3 explains our radix-partitioning-based multi-GPU sorting algorithm, while we describe how we implement it to achieve peak performance in Section 4. In Section 5, we evaluate the performance of RMG sort on two state-of-the-art multi-GPU accelerator platforms. We discuss our work and our experimental evaluation in Section 6. Finally, we give an overview of the related work in Section 7 and conclude in Section 8.

2 Background

In this section, we explain the background information required to understand our multi-GPU sorting algorithm in terms of its design and its implementation. First, we describe the GPU hardware architecture and how it translates to the CUDA programming model in Section 2.1. In Section 2.2, we give an overview of the different interconnect types that multi-GPU accelerator platforms incorporate, outlining the importance of understanding a system's interconnect topology. Finally, we explain the radix sort algorithm class as well as the key insights into its relevant algorithm variants in Section 2.3.

We evaluate our algorithm on systems with NVIDIA GPUs only. Therefore, we explain the GPU hardware architecture (e.g. Volta, Ampere, etc.) and the GPU programming environment (i.e. CUDA) of NVIDIA GPUs. The explained concepts apply to GPUs of other vendors as well. When we reference the technical terms introduced by NVIDIA throughout this section, we additionally mention the names and terms that other vendors, such as AMD, use to describe the equivalent or similar concept or component of their own.

2.1 GPU Architecture

GPUs are designed to support massively parallel computations. In contrast to the CPU that hides the memory access latency with complex flow control and large data caches, a GPU hides memory access latency with concurrently executed computation [46]. GPUs are equipped with thousands of cores that are organized in a specialized hierarchy. The main unit of computation is the Streaming Multiprocessor (SM) [40], equivalent to the Compute Unit (CU) for AMD CDNA architecture GPUs [4]. One GPU consists of an array of SMs. For example, the latest server-grade GPUs, such as the NVIDIA Tesla V100 (Volta architecture) and the NVIDIA A100 GPU (Ampere architecture), consist of 80, and 108 SMs, respectively [38, 40]. Each SM can run multiple concurrent groups of threads, so-called thread blocks. Unlike the CPU, GPUs do not support branch prediction, but they excel at achieving high instruction throughput rates for arithmetic operations. Within the Tesla V100 and the A100 GPU, each SM contains 64 INT32 cores and 64 FP32 cores that execute simultaneously. For 64-bit floating-point numbers, each SM provides 32 cores. As a consequence, modern data-center GPUs can achieve remarkably high peak instruction throughput rates. For single-precision floating-point numbers, the NVIDIA Tesla V100 reaches up to 15.7 TFLOPS while the NVIDIA A100 achieves 19.5 TFLOPS. The double-precision peak throughput is half as high for both GPUs because each SM includes half as many 64-bit cores. Figure 1 shows a simplified overview of the GPU architecture.

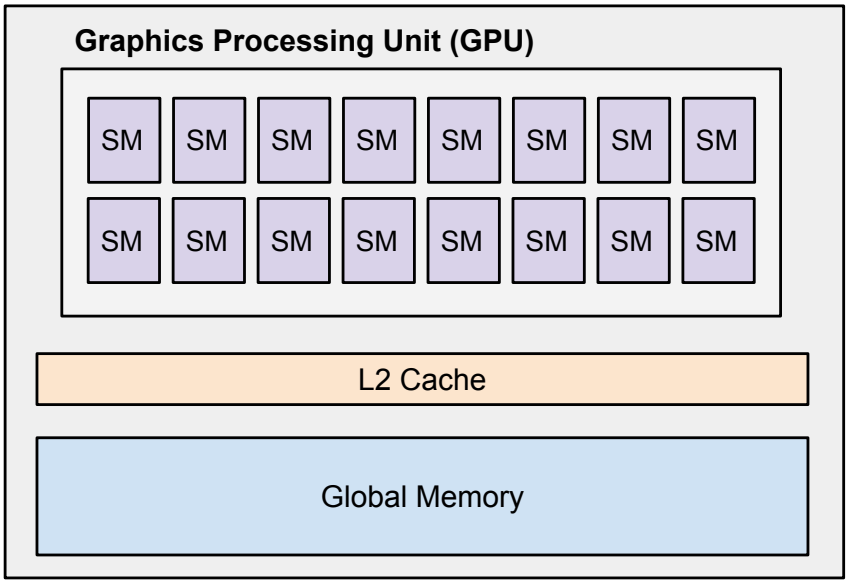


Figure 1: Simplified GPU architecture

GPUs also provide a high-bandwidth memory hierarchy that is tailored towards and supports their massively parallel compute architecture. GPU memory 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 high peak bandwidth rates of up to 900 GB/s (NVIDIA Tesla V100) and 1555 GB/s (NVIDIA A100). Compared to main memory, however, the GPU memory capacity is rather limited. The NVIDIA A100 is available with up to 80 GB of DRAM. The GPU cache hierarchy begins with the L2 cache that hides the latency of global memory accesses (similar to AMD GPUs). The L2 cache size of server-grade GPUs varies between 4 and 40 MB depending on the GPU. Furthermore, GPUs equip their SMs with on-chip memory caches: Each SM comes with a local, high-bandwidth, low-latency L1 cache to accelerate computation on frequently used data. Additional on-chip memory components (i.e. per SM) are the register file and the shared memory subsystem. While the L1 cache automatically hides the memory accesses of all threads running on the same SM, shared memory needs to be explicitly managed by the programmer, and, can thus be seen as a user-managed cache. Since the Volta architecture, NVIDIA GPUs combine the L1 data cache and shared memory for a total of 128 and 196 KB per SM on the Tesla V100 and the A100 GPU, respectively (see Figure 2). Combining the L1 cache with the shared memory can improve the performance of shared memory-agnostic applications. Still, manually handling shared memory enables programmers to achieve maximum performance.

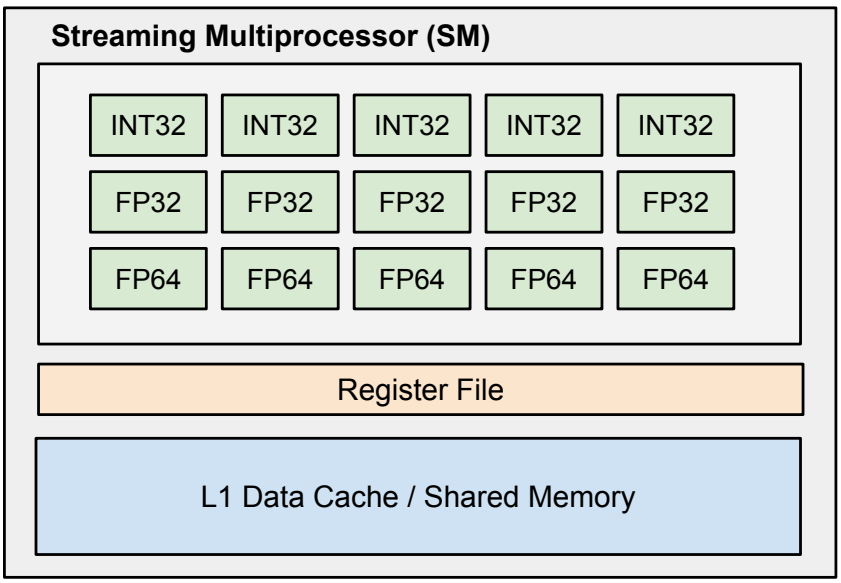


Figure 2: Simplified SM architecture

2.1.1 The CUDA Programming Model

CUDA is a general-purpose parallel computing platform and programming model for NVIDIA GPUs [46]. (AMD offers the ROCm software stack as a general-purpose and HPC programming environment [4].) The CUDA platform includes a small set of C/C++ language extensions together with an API and a compiler that allows developers to program CUDA-capable GPUs using the high-level programming language C or C++. Toolkit-included libraries, like the high-performance parallel algorithms library Thrust [44], further simplify the development of heterogeneous applications that utilize CPUs and GPUs cooperatively. CUDA code typically contains a mixture of *host* code (executed on the CPU), and *device* code (executed on the GPU). Thus, the CUDA programming model assumes the GPU to act as a co-processor and as a physically separate device to the host CPU. The CUDA API offers host-side functions for allocating device memory and for copying host memory to the GPU (i.e. global device memory).

To execute parallel computations on the GPU, programmers write so-called *kernels*. A kernel is a device code function, that is executed in parallel by multiple threads on the CUDA device. The programmer configures the number of threads that execute the kernel when calling the kernel function. For this, the execution configuration syntax is used. It consists of two parameters, the number of thread blocks and the number of threads per block. The number of threads per block

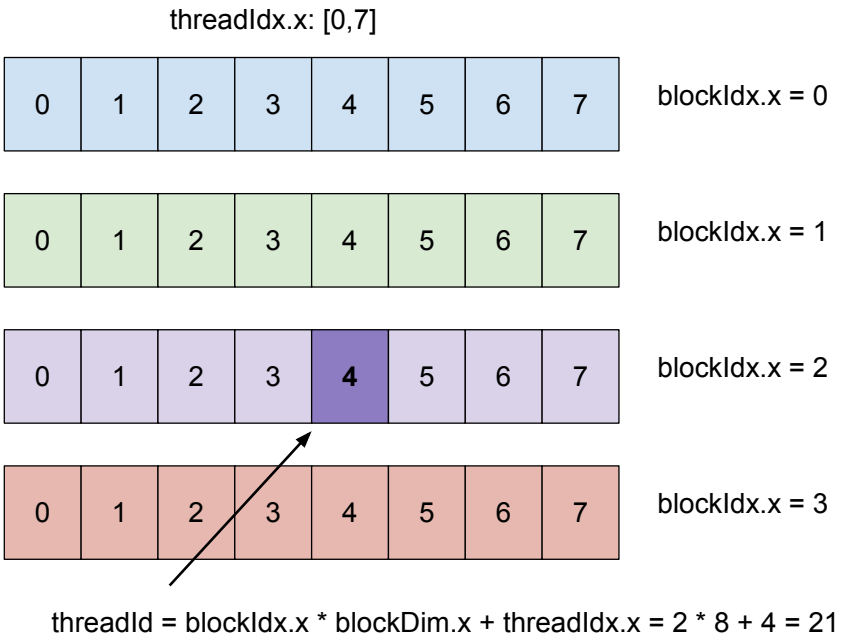


Figure 3: Calculating a unique CUDA thread ID

is limited because all threads of a block run on the same SM, sharing its limited on-chip memory (e.g. the register file and shared memory). The latest NVIDIA GPU architectures (i.e. Pascal, Volta, and Ampere) support a maximum of 1024 threads per block. Thread blocks can be scheduled in any order across any number of cores. This enables CUDA code to scale very well to many cores.

The kernel function is implemented following the SIMT architecture (Single Instruction, Multiple Threads). The kernel code defines the execution behavior and branching path from the perspective of a single thread. Thus, the programmer writes kernel functions for independent, scalar, and scalable threads, allowing for thread-level parallelism in a simple way.

Within the device code of a kernel function, a thread's unique *thread ID* is exposed to the programmer through built-in CUDA variables, e.g. thread-level, and block-level indices and dimensions. Figure 3 shows an example of how each thread can calculate its kernel-wide unique thread ID for a kernel that is launched with four thread blocks and eight threads per block. The variable `blockDim.x` holds the number of threads per block. The variable `threadIdx.x` is the thread index within the respective block (`blockIdx.x`) that the running thread is a part of. A common way of dividing a given computational task across many threads equally utilizes this globally unique thread ID. Typically, each thread calculates which

small subset of the input data to process based on its thread ID. Nonetheless, the programmer can orchestrate memory accesses to arbitrary addresses in global memory to perform compute operations on this data. To achieve peak memory bandwidth and compute resource utilization, however, it is necessary to understand exactly how the GPU schedules such high numbers of threads in parallel, and how it services the memory operations that each thread requests.

2.1.2 Thread Scheduling

A streaming multiprocessor creates and executes the threads of an affiliated thread block. A thread block can run up to 1024 threads in total. The SM schedules these high numbers of threads in groups of 32 consecutive threads, so-called thread warps, and stores the necessary execution context in on-chip memory. As a result, context switches happen at a very low cost. All threads of a running warp start with the same program counter, while each thread holds its instruction address. Therefore, the threads of a scheduled warp can execute independently from one another. However, out of its 32 threads, the warp executes only those threads that share the next instruction to be issued. The threads that are executing the current common instruction are called active threads. The other disabled threads have to idle until they are scheduled. Whenever a warp encounters a data-dependent conditional branch and threads diverge, the execution of the different possible branches is serialized. First, those threads that take path A execute in parallel, while the rest idles. Then, the threads that take the branch path B execute, and so on. Thus, to maximize the number of threads running in parallel within a warp, the programmer needs to avoid branching and/or orchestrate the conditions in such a way that thread divergence happens at warp borders. The threads of different warps can always execute independently. Ideally, the only constraint on the level of parallelism within a thread block is given by hardware limitations, i.e. the size of the register file and shared memory.

Since the Volta architecture, the SM stores the execution context (i.e. the program counter, the call stack, and registers) of each thread in on-chip memory during the whole runtime of a kernel. This enables the threads of the same warp to run completely independently and in parallel. The performance implications of branch divergence within a warp are therefore heavily mitigated for GPUs of that and later architectures.

To achieve peak instruction and memory throughput on the GPU, it is crucial to maximize the number of concurrently executing threads because high numbers of threads increase the chances that the GPU's computational units (cores) are fully utilized at any given time. However, having many thread warps run in parallel

does not yet guarantee optimal hardware resource utilization since a significant portion of threads might suffer from high instruction latency. Optimal performance is achieved when the warp scheduler always has enough ready-to-execute thread warps to schedule. This hides the latency of other waiting thread warps. Thus, the number of clock cycles it takes until a thread is ready to execute its next instruction needs to be minimal. One main reason that prevents a thread warp's instruction to be ready to execute is that the required input operands are not yet available because they need to be loaded from memory first.

The memory latency of global memory is orders of magnitude higher than that of on-chip memory (e.g. the L1 cache, and shared memory). Therefore, the negative performance impact of a sub-optimal memory access pattern to global memory is much more significant than misaligned accesses to on-chip memory. The global memory is part of the device memory and can only be read or written to via 32-, 64-, or 128-byte memory transactions. Moreover, all global memory accesses performed by a thread are handled as part of its thread warp. For the current instruction executed by a thread warp, the GPU groups together the memory accesses to global memory of all active threads within that thread warp. Here, the GPU tries to coalesce the required reads and writes in such a way that as few memory transactions as possible are performed. Since the byte-granularity of device memory transactions is commonly much higher than the number of bytes of the variables the kernel instruction operates on, over-fetching can become a performance bottleneck. If the threads of the same thread warp read 4-byte integers from distant memory addresses in global memory (e.g. because of a random access pattern), then 28 bytes of each 32-byte memory transaction will be loaded unnecessarily. This decreases the memory throughput, highlighting the importance of aligned memory access patterns within thread warps.

2.2 System Interconnects

As physically separate devices, GPUs are attached to the CPU's memory controller via an interconnect bus. Traditionally, the PCIe interconnect has been used to connect the GPU to the CPU. Given that its bandwidth rate is significantly lower than that of main memory, GPU-accelerated data processing operations on such systems are often data transfer-bound. Today's high-performance computing (HPC) systems include multiple GPUs to increase the total computational power and the available GPU memory. Consequently, it became relevant to interconnect GPUs between each other. The interconnect topology defines exactly how the GPUs are connected to the host-side and between each other. It significantly impacts the performance of multi-GPU-accelerated applications [31]. This is especially relevant for modern systems that often incorporate heterogeneous interconnect technology [37, 42]. A specific interconnect topology can render the utilization of only a certain subset of GPUs efficient while employing all system-wide GPUs can decrease the end-to-end performance [34]. In the following, we explain the different interconnect technologies of modern multi-GPU accelerator platforms. Given that we implement and evaluate our multi-GPU sorting algorithm on two hardware systems, we focus on explaining the concrete interconnects that these systems come with. To keep this thesis as vendor-independent as possible, we mention the interconnect advances of other hardware vendors as well.

2.2.1 CPU Interconnects

Since most multi-GPU accelerator platforms are Non-Uniform Memory Access (NUMA) systems, we first explain CPU interconnects. To enable one NUMA node to access the memory of a remote NUMA node, the CPU sockets are connected via CPU interconnects. CPU interconnects commonly provide higher latency and less throughput than the CPU's memory controller [33]. Consequently, memory accesses of one CPU to the main memory of another remote NUMA node are slower than local accesses.

There are many different CPU interconnects available. Intel's QuickPath (QPI) interconnects are point-to-point links for the Intel Xeon Skylake architecture, that connects multiple processors while supporting cache coherency across all NUMA memory regions [11]. The QPI has later been replaced by its successor, the Intel UltraPath (UPI) interconnect. AMD's latest Infinity Fabric is a high-bandwidth, general-purpose interconnect that is used as the CPU interconnect for multi-socket systems including AMD EPYC processors [27]. IBM's Power Systems interconnect their CPU sockets with the X-Bus interconnect [37].

Typically, NUMA systems attach equally as many GPUs to each NUMA node.

Data transfers from one NUMA node to the GPUs that are attached to the remote NUMA node traverse the CPU interconnects as well as the CPU-GPU interconnects. If a system lacks direct P2P interconnects between the GPUs of different NUMA nodes, P2P copies have to traverse the host-side via multiple hops, including the CPU interconnect.

2.2.2 GPU Interconnects

In this section, we give an overview of state-of-the-art GPU interconnects.

PCIe. PCIe 3.0 (Peripheral Component Interconnect Express) is used as the standard interconnect bus for many peripheral devices, including GPUs. It is exclusively used to connect GPUs to the CPU. The PCIe interconnect supports full-duplex communication at 1 GB/s per lane. Thus, data can be simultaneously transferred in both directions at full speed. One PCIe 3.0 link typically connects two devices with 16 lanes for a total, theoretical bandwidth of 16 GB/s per direction.

Recently, the first multi-GPU systems with PCIe 4.0 interconnects became available on the market. PCIe 4.0 doubles the bandwidth rate of PCIe 3.0 for a theoretical peak of 32 GB/s per direction. With up to four PCIe 4.0 links per CPU, a two-socket system can attach a total of eight GPUs.

Multi-GPU systems with no further direct P2P interconnects only support inter-GPU communication through multi-hop host-side transfers. On such systems, the P2P throughput between GPUs suffers from shared bandwidth effects as the host-side (i.e. PCIe lanes, PCIe switches, CPU memory controller) becomes the bottleneck.

NVLink. Over the last few years, hardware vendors introduced high-bandwidth, low-latency GPU interconnects for direct P2P transfers, enabling faster inter-GPU communication. AMD released the Infinity Fabric interconnect [3], while NVLink is NVIDIA's fast interconnect technology. NVLink 1.0 interconnects provide 20 GB/s per link per direction while NVLink 2.0 interconnects achieve 25 GB/s. One NVLink 2.0-enabled GPU supports up to six links. If two GPUs were interconnected with all of their six NVLink 2.0 links directly, P2P copies would benefit from a theoretical peak bandwidth of 150 GB/s per direction.

The latest NVLink 3.0 increases the number of links per GPU up to 12, doubling the theoretical peak P2P bandwidth between two GPUs up to 300 GB/s. As mentioned, NVLink is primarily designed for accelerating inter-GPU communication. However, the IBM Power System AC922 connects its GPUs to its POWER9

CPUs via NVLink as well. NVLink-based CPU-GPU interconnects significantly mitigate the data transfer bottleneck, as their bandwidth comes close to that of main memory. Thus, NVLink interconnects can enable GPU systems to accelerate data processing workloads efficiently [32, 34].

NVSwitch. NVSwitch is an NVLink-based switch chip introduced by NVIDIA in 2018. It enables non-blocking, all-to-all, inter-GPU communication at high bandwidth rates by connecting up to 16 GPUs between each other in a point-to-point network (i.e. hybrid cube mesh) [39]. When the GPUs are interconnected via NVLink 3.0-based NVSwitch, any GPU i benefits from a theoretical peak bandwidth of 300 GB/s per direction to any other GPU j. More importantly, all GPUs can transfer data between each other simultaneously without sharing the bandwidth.

2.2.3 The Data Transfer Bottleneck

For single-GPU systems, the infamous data transfer bottleneck simply resulted from the fact that the interconnect's bandwidth was significantly lower than that of CPU and GPU memory. In multi-GPU systems, there is not *one* data transfer bottleneck anymore as different interconnect topologies introduce different pitfalls. For more details, we refer to the work of Maltenberger et al. [34]. We summarize the main scenarios where the device-side or host-side hardware components throttle the measured data transfer throughput in a multi-GPU system:

1. **Lack of P2P interconnects.** A limited number of direct, high-bandwidth P2P interconnects can result in P2P transfers between different GPU pairs having varying costs. For example, some systems connect GPU i with GPU j via NVLink 2.0, but do not include a direct P2P path to GPU k. Thus, P2P transfers from GPU i to k require traversing the host-side via multiple interconnect hops. If the system attaches its GPUs to the CPU via PCIe interconnects, the P2P copy from GPU i to k is much more expensive than that to GPU j. Some algorithms employ a multi-hop routing strategy between GPUs [50]. The P2P transfer from GPU i to k is re-directed from GPU i to j first, and then traverses from GPU j to k, thus, avoiding the host-side entirely. However, this strategy is only suitable for certain systems with a fitting interconnect topology. In this example, it only works if GPU j is directly interconnected with GPU k.

2. **Shared PCIe switches.** On systems that connect the GPUs to the host-side via PCIe interconnects, the number of available PCIe switches per NUMA node critically influences the performance of concurrent CPU-GPU

transfers to/from multiple GPUs. Especially on hardware platforms with a total of eight GPUs (or more), the CPU oftentimes does not include enough PCIe switches to attach all four GPUs exclusively. Then, each pair of GPUs is attached to the CPU through the same, shared PCIe switch (i.e. via one PCIe instance of 16 lanes). As a consequence, data transfers to/from neighbouring GPUs suffer from shared bandwidth effects if performed simultaneously. Then, the throughput of copying 8 GB of data per GPU from CPU node 0 to GPUs 0 and 1 concurrently is equal to that of a single CPU-GPU copy of 8 GB from CPU node 0 to either GPU 0 or 1.

3. **Low NUMA interconnect bandwidth.** A CPU interconnect of insufficient bandwidth B_c makes it infeasible to include GPUs of remote NUMA nodes. Then, the CPU-GPU transfer throughput to the remote GPUs can not exceed the CPU interconnect bandwidth B_c, even though the CPU-GPU interconnect itself has a high bandwidth B_i, with $B_i > B_c$. Therefore, the data arrives at the local GPUs significantly earlier, which delays the overall performance if the multi-GPU algorithm includes a synchronization step between the GPUs.

4. **Low main memory bandwidth.** Given that the CPU-GPU interconnects bandwidth and the number of GPUs per CPU increases, the main memory is increasingly put under pressure. The main memory bandwidth B_m poses an upper limit to the throughput that multiple, concurrent data transfers to g GPUs can achieve. Thus, if the main memory bandwidth is insufficient to support g concurrent transfers at full interconnect speed, the achieved throughput will be throttled to B_m GB/s, even if each individual CPU-GPU interconnect could provide a bandwidth rate B_i so high that $g * B_i > B_m$.

2.3 Radix Sort

Radix sort is a non-comparison-based sorting algorithm that proves to be very efficient for data set sizes that go beyond a certain threshold, due to its linear computational complexity [2, 16, 54, 58, 63]. As the radix sort algorithm does not perform any comparisons, the otherwise relevant lower boundary for comparison-based sorting algorithms of $O(n \times \log n)$ does not apply. Instead of performing comparisons, radix sort algorithms iterate over the keys' bits and partition the keys into distinct buckets based on their radix value.

The word radix (or base) refers to the number of unique digits that are used to represent a number. In the binary system, there are two possible digit values (bits): 0 and 1. To reduce the number of iterations, radix sort algorithms look at multiple consecutive bits c at a time. We present the notation that we use throughout this work in Table 1.

There are two fundamentally different radix sort algorithm variants with regards to where the iteration over the keys' bits starts. This can either be from the most or the least significant bit (MSB or LSB). Given k-bit keys, the number of partitioning passes (or rounds) 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 according to its radix value in the c bits that are considered in the current pass until all k bits have been considered. This leaves radix sort with a computational complexity of $O(n \times p)$. The final partitioning pass arranges the keys into the fully sorted order.

When iterating over the keys' bits, 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 keys' partitioning within each bucket in each round recursively. Thus, an MSB radix sort needs to keep track of increasing numbers of buckets. However, the MSB approach can ignore the sort order of preceding rounds as this information is

Table 1: Radix sort notation

Symbol	Description
n	Number of input keys
k	Number of bits per key
c	Number of consecutive bits considered at a time
p	Maximum number of partitioning passes

implicit in the way that each bucket is refined in each pass (i.e. further partitioned into smaller buckets).

Even when considering many consecutive bits c per partitioning pass, radix sort has a comparatively high memory bandwidth demand. Since the few necessary compute operations are cheap to perform, radix sort algorithms and radix partitioning approaches quickly become memory bandwidth-bound. Single-GPU radix sort algorithms outperform parallel CPU radix sort algorithms, mainly because GPUs are equipped with memory subsystems that provide significantly higher bandwidth rates than the CPU's main memory [34, 63, 59].

A common data structure used in radix sort algorithms is the histogram. It is used to keep track of how the input keys distribute across the different buckets. More precisely, for each of the 2^c buckets, the histogram stores the number of keys that fall into that bucket. Thus, radix sort algorithms that rely on histograms resemble counting sort algorithms. After reading the input keys once to compute the histogram, the keys need to be partitioned in memory so that all keys of bucket i proceed all keys of bucket $i+1$. Figure 4 illustrates a simplified partitioning phase of a radix sort algorithm for an example of $n = 8$ keys with $k = 3$ bits per key and $c = 3$ considered bits at a time.

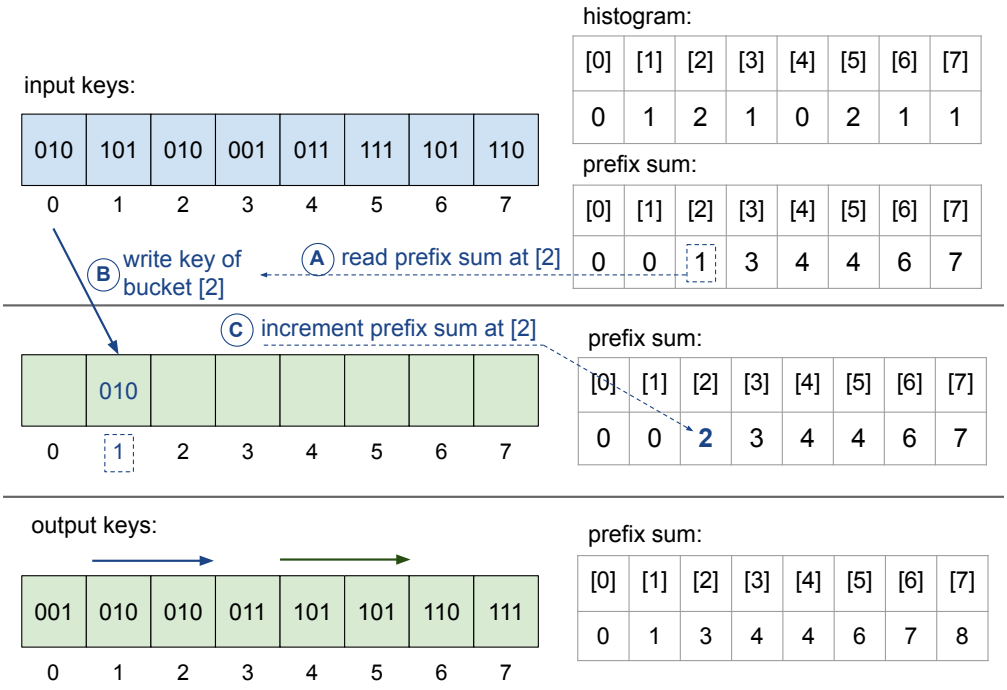


Figure 4: Simplified radix sort example with histogram and prefix sum

Given the information about the number of keys per bucket, we obtain the starting write offsets for each bucket by calculating the prefix sum on the histogram. Then, the input keys are read once again. This time, each key is written to a new location based on the write offset of the bucket it belongs to. To ensure that the following keys of the same bucket are written to subsequent and unique positions, the bucket's write offset needs to be incremented by one (as one key has successfully been scattered). In the example, one partitioning pass on c bits is sufficient to fully sort the input. On 32-bit or 64-bit keys, multiple partitioning passes become necessary.

To scatter the keys into their corresponding buckets efficiently and in parallel, most radix sort algorithms operate with an auxiliary memory buffer of size n [58, 63, 67, 68]. In each partitioning pass, the role of the two memory buffers alternates between holding the input keys and the partitioned output keys. Thus, radix sort algorithms often have a relatively high space complexity, i.e. memory consumption. To avoid costly dynamic memory allocations, the radix sort primitives of state-of-the-art GPU-based parallel programming libraries, such as Thrust and CUB [44, 41], allow for passing a pre-allocated memory buffer to the function call, which will be used as auxiliary memory internally.

Sorting Floating-Point Numbers. While the radix sort algorithm is intuitive for integer data types, given their binary representation, radix sort algorithms can equally as well sort floating-points data types. The IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines the binary representation format of floating-point numbers [26]. It is the most widely used standard for how to represent and operate floating-point numbers. floating-point data type implementations that follow the IEEE-754 standard reserve the most significant bit as the sign bit. The following eight bits are reserved for the exponent, and the remaining 23 bits represent the mantissa. Since the exponent is stored at more significant bits than the mantissa, and the exponent is always positive, floating-point numbers can be sorted via radix sort algorithms in the same way as integer keys. When comparing a floating-point numbers as if it was an integer type, greater floating-point numbers are interpreted as such compared to smaller values. Both, for negative integer and floating-point numbers, radix sort algorithm need to be adjusted [66, 25].

3 Algorithm

In this section, we explain our proposed radix-partitioning-based multi-GPU sorting algorithm (RMG sort). It sorts the input keys using only the GPUs. Since we do not involve the CPU in any computation, we can only sort data sets that fit into the combined device memory of the system's GPUs. Since we design a radix-partitioning-based algorithm, we avoid the need to merge data. The GPUs partition the keys into buckets based on their radix values. We exchange the keys of certain buckets between the GPUs and finally sort the buckets on each GPU. We use an incremental most significant bit (MSB) radix partitioning strategy on the GPUs to determine which keys belong to which GPU. Our algorithm requires one all-to-all key swap between the GPUs via the P2P interconnects, independent of the number of GPUs used. Our algorithm reduces the inter-GPU communication compared to previous sort-merge algorithms, especially for many GPUs.

In summary, our proposed algorithm works as follows: First, the unsorted input keys are copied to the GPUs in chunks of equal size. Each GPU partitions its chunk's 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 the keys 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 exchange. 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 any subsequent GPU j with $j > i$. Thus, the keys of GPU i are all less than or equal to the keys of GPU $i + 1$. After the key swap, each GPU sorts its keys locally in order to bring all keys across the g GPU chunks into the final, sorted output order. We can reduce the final sorting workload because we partitioned the keys into distinct buckets based on their MSB radix value before the P2P key swap and because we respect this partitioning order during the P2P key swap. Instead of sorting the entire chunk, each GPU sorts its radix buckets individually. Given that the radix partitioning phase already examined the most significant r bits of each key, we sort on the remaining $k - r$ bits in the bucket sorting phase. Finally, we copy the sorted GPU chunks back to CPU memory. Because we sort the distinct buckets of each GPU individually, we can interleave the sorting computation with copying the data back to the CPU. Once a bucket is fully sorted, we already transfer it back, while the remaining buckets are still being sorted. Thereby, we effectively hide the time duration of the sorting computation on the GPUs.

In the following subsections, we give a more detailed explanation of how the radix partitioning phase ensures that one exchange of buckets (P2P key swap) between

the GPUs is sufficient, even for highly skewed data distributions (Section 3.1). Furthermore, we explain how our algorithm determines to distribute the keys across the GPUs in such a way that the global sort order is respected, and nearly perfect load-balancing is achieved (Section 3.2). Finally, we explain how we use the bucket information gained during the radix partitioning phase to accelerate the final sorting computation after the P2P key swap (Section 3.3).

3.1 On-GPU MSB Radix Partitioning

After the n input keys are split up and copied to the g GPUs in equal sized chunks, each GPU starts with its radix partitioning phase. During the entire partitioning phase, each involved GPU partitions its keys locally (i.e. in its local device memory only). 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 computed histogram gives us the starting write offsets for each of the 2^c buckets. Using the prefix sum, each GPU partitions all keys of its chunk in its local device memory so that all keys of bucket i precede all keys of bucket $i+1$. Given that we keep the histogram data, we know how many keys belong to each bucket. The keys that belong to the same bucket still remain in unsorted order as we have only taken the most significant c bits into account.

For most data distributions, the probability is high that there is a radix bucket for which every GPU finds dedicated keys, i.e. keys that belong to that bucket. In fact, for uniformly distributed keys, every GPU 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 gathered and aligned in the device memory of one and the same GPU (i.e. *complete*, but not *spanning* buckets). 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. Bucket `[0]` represents keys with c leading `0`s, bucket `[1]` represents the keys that have their most significant c bits' digit value equal to `1`, and the bucket of number 2^c contains the keys that have only `1`s in their c most significant bits. We distribute the buckets across the GPUs in ascending order with respect to the radix value that they represent. Thus, we distribute the buckets of the smallest radix values to GPU 0, while GPU g gets the buckets with the highest radix values. Afterwards, sorting the buckets individually on each GPU brings the keys across all GPUs into the final, globally sorted output order.

Determining how exactly to re-distribute the keys that are spread across the GPUs according to their dedicated bucket, requires exchanging the histogram data be-

tween the GPUs. Every GPU sends its own histogram to all the other involved GPUs via the P2P interconnects. In that way, each GPU knows about the entire key distribution and uses it to determine how to proceed. More precisely, using the combined multi-GPU histogram, each GPU computes the logical distribution of buckets across the g GPUs and checks whether the current level of partitioning allows for each complete bucket to fit onto one GPU only. As soon as there is one bucket that spans over more than one GPU, the level of partitioning is not sufficient.

Spanning Buckets. Depending on the data distribution of the input keys, more than one radix partitioning pass might be necessary. We can not know in advance how many different buckets our partitioning pass on the most significant c bits will generate. The input data might be highly skewed and contain only leading zeros in the most significant c bits. In that case, we simply do not have enough buckets to distribute across the g GPUs and a more refined partitioning of the large buckets is necessary. Even if the key distribution is such that the partitioning pass on the most significant c bits leaves us with more buckets than we have GPUs, some buckets might contain significantly more keys than others. We can not predetermine how many keys the different buckets will contain. It is desirable for our radix partitioning phase to split the keys into buckets that are small enough, because a fine-grained partitioning of keys enables avoiding load imbalances between the GPUs. When each GPU contains approximately the same number of keys after the P2P key swap, the computational workload is equally divided and we maximize the performance. Therefore, we perform multiple partitioning passes on subsequent sets of c bits, starting from the most significant one, until all buckets are *small enough*, i.e. there are no spanning buckets left. Using our radix partitioning strategy, we ensure that the P2P key swap distributes the keys between the GPUs with nearly perfect load balancing while respecting the globally sorted bucket order.

A spanning bucket is not necessarily the bucket that contains the most keys because we specifically order the buckets based on their radix value. A bucket of bigger size might be placed at the beginning of the GPU chunk and fit onto that GPU while a subsequent bucket with a higher radix value and less keys is placed at the end of the same GPU chunk that can not fit all of those keys anymore. In any case, a spanning bucket prevents us from performing the re-distribution of buckets (i.e. the P2P keys swap) because we could not fully sort the spanning bucket without further communication between those GPUs that the bucket spans. Thus, we perform another partitioning pass. Any subsequent radix partitioning round only refines the partitioning of the spanning buckets. The buckets that already fit onto one GPU stay untouched.

In Figure 5, we show an example of our radix partitioning algorithm phase on four GPUs. In the example, we sort 32-bit keys while considering $c = 8$ bits at a time. In the first partitioning pass, each GPU scatters its keys based on their radix value in the most significant eight bits [32..24). We show the result of the local partitioning step in the top half of each partitioning pass, referred to as the physical view of the GPU memory. For the sake of simplicity, the input does not contain keys for every possible bucket. However, 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 (i.e. the bottom half of each partitioning pass shown in Figure 5). As a consequence of many keys having eight leading zeros, the complete bucket `[0]` is a spanning bucket after the first partitioning pass. Consequently, no P2P key swap is yet possible.

We continue with another partitioning pass on bits [24..16) on the spanning bucket `[0]` only. We color the buckets that are already sufficiently partitioned in light gray color in Figure 5, while the buckets that are subject to the current partitioning pass are colored based on the bucket they belong to. In the second partitioning pass, each GPU j, that contains keys belonging to the complete but spanning bucket `[0]`, refines its bucket $\texttt{[0]}_{(j)}$ into multiple sub-buckets – at least one, at most 2^c. For example, bucket `[0]` on GPU 0 is refined into two smaller buckets. It contains keys that have either another eight zeros in the most significant bits [24..16), or `00000001` as their most significant bits [24..18]. After the histogram exchange, the constructed logical distribution of complete buckets shows that the spanning bucket was heavily reduced. However, bucket `[0:2]` (physically resident on GPU 3) now remains a spanning bucket as it would span the GPUs 1 and 2 if we were to perform the P2P key swap. Thus, a third partitioning pass on bits [16..8) is necessary. Since the third partitioning pass results in a bucket distribution with no spanning buckets, the radix partitioning phase is completed.

3.2 Multi-GPU P2P Bucket Exchange

At the bottom of Figure 5, we show an example of the final bucket distribution that the radix partitioning phase determines. This distribution of complete buckets aligns them in a globally sorted order across the g GPUs. After the radix partitioning phase, the keys of each complete, non-spanning bucket are still scattered across different GPUs as they reside in the device memory of their initial GPU (seen at the top half of each partitioning pass in Figure 5).

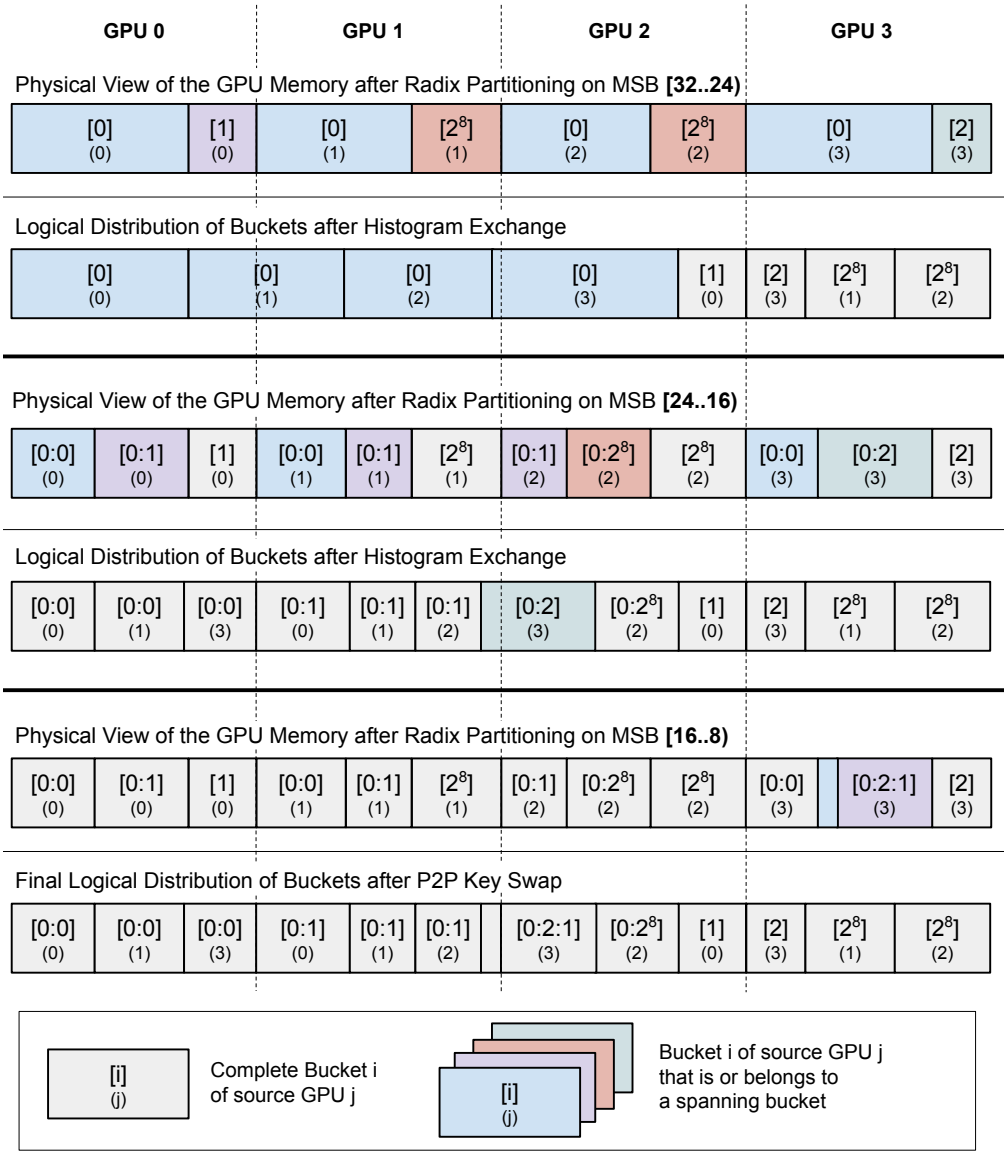


Figure 5: Radix Partitioning Example

Up until this point, the keys have only been partitioned into buckets locally on each GPU. In the multi-GPU P2P key swap, we re-distribute the keys between all GPUs according to the determined bucket distribution. Thus, we do not swap individual keys, but rather entire buckets of a GPU. The P2P key swap is therefore really an exchange of GPU-local buckets. Each bucket's destination GPU can either be the same as the source GPU or a remote GPU in which case the memory copy is performed over the P2P interconnects. As seen in the example in Figure 5, bucket $\texttt{[0:0]}_{(1)}$ of GPU 1 needs to be transferred to GPU 0 next to the bucket

`[0:0]`$_{(0)}$ that is already on GPU 0. For each bucket, we need to know not only the destination GPU but also the exact write offset to the destination GPU's device memory.

As explained in the previous section, we compute the logical distribution of buckets after each partitioning pass to check if there are any spanning buckets left. Consequently, the final radix partitioning pass already leaves us with all the information necessary to perform the multi-GPU P2P key swap. Determining if there are spanning buckets requires to compute the destination GPU and write offset for each bucket to be transferred to in a potential bucket exchange.

3.2.1 Multi-GPU-Striped Histogram

The first step of calculating the logical distribution of buckets across the g GPUs is to exchange the histograms between the GPUs via P2P transfers. We reserve enough memory buffers for each GPU to store g separate histograms. After each GPU receives the histograms of all other $g-1$ GPUs, it computes the multi-GPU-striped histogram – a histogram data structure that combines the histograms of all g GPUs in the following way: We reserve the memory space for an array of size $g \cdot 2^c$. We do not add together the bucket counts of each GPU but rather align the bucket counts from every histogram in a specific order. First, we order by the radix value that the bucket represents (from 0 to 2^c) and as a tie-breaker, we sort by the GPU identifier $i \in \{0, ..., g\}$. Thus, the first bucket count stores how many keys belong to bucket `[0]` and reside on GPU 0. The second bucket count stores the number of keys belonging to bucket `[0]` that are in the device memory of GPU 1. The bucket count at index g stores the number of keys of GPU 0 that belong to bucket `[1]`, and so on.

Calculating the prefix sum on the multi-GPU-striped histograms returns the starting write offsets for each bucket of each GPU. However, since the prefix sum continuously adds up the bucket counts, the write offsets are computed as if all GPUs share the same device memory. Once we perform the actual P2P bucket exchange, we adjust the memory offsets to the destination GPU's chunk-local memory address space for each bucket b by subtracting the number of keys that we distribute onto all the preceding GPUs.

Using the prefix sum of the multi-GPU-striped histogram, we also determine the mapping of each bucket to its destination GPU. The computation of this mapping is the last step of each partitioning pass. In the case that each bucket maps to exactly one GPU only, there are no spanning buckets and the radix partitioning phase is completed.

3.2.2 Load-Balancing

To reduce the number of partitioning passes, we do not enforce perfect load-balancing. Instead, we allow for certain GPUs to handle slightly more keys than other GPUs. The first partitioning pass very rarely results in a bucket distribution that is perfectly aligned with the chunk size. The sum of the bucket sizes that belong to the same GPU would need to be exactly equal to the chunk size $\lceil n/g \rceil$ for every GPU. Even for uniform data distributions, this is highly unlikely. For example, the radix partitioning phase shown in Figure 5 could already be completed after two rounds if we allowed for distributing the entire bucket $[0{:}2]_3$ to GPU 2. Then, GPU 2 sorts slightly more keys than GPU 1, but we avoid computing another histogram and scattering the keys for that one spanning bucket on GPU 3 while all the other GPUs idle.

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 to avoid slightly overflowing buckets to be treated as spanning buckets. Thus, whenever a bucket overflows into a GPU by a number of keys $\sigma \leq \epsilon$, these otherwise overflowing σ keys are assigned to the adjacent GPU that already holds more than σ 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, result in avoiding this spanning bucket completely. In that case, the entire bucket fits onto a single GPU and we avoid an entire partitioning pass, given that there are no spanning buckets on other GPUs. 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 over by up to two. This is because each GPU employs the ϵ-padding at the start (spanning bucket is *left-reducable*?) and the end (spanning bucket is *right-reducable*?) of its chunk buffer.

3.2.3 Last-Pass Spanning Buckets

In cases of extremely skewed data distributions, it can happen that spanning buckets occur even after the last partitioning pass on the least significant c bits. The simplest example of such a case is an input data set whose n keys all consist of one single value. Having considered all k bits after the radix partitioning phase, there will still be a spanning bucket with n keys that spans over all g GPUs. The important insight here is that any spanning bucket that remains after the last partitioning pass is completed, can only consist of keys of the same single value. Since all keys of one last-pass spanning bucket are of the same value, we can choose arbitrary borders for where to split the spanning bucket, and consequently, how to distribute its parts onto the different GPUs. We simply distribute the keys of each such last-pass spanning bucket across the spanning GPUs in such a way

that we achieve perfect load balancing on the involved GPUs. Thus, this so-called last-pass spanning bucket poses an exception to the rule that no spanning buckets are allowed before the P2P key swap.

3.2.4 Upper Bucket Bound

Since our algorithm employs MSB radix partitioning, the number of buckets that we need to manage grows continuously with each partitioning pass. It is therefore necessary to analyze the upper bound for the number of buckets that our radix partitioning phase generates. Since we consider c bits at a time, each partitioning of a spanning bucket divides that spanning bucket into 2^c sub-buckets. Thus, the upper bound for the number of possible spanning buckets determines the maximum possible number of buckets that we need to manage, and sort in the final bucket sorting phase. In this context, we view the initial input data of n keys that is divided onto the g GPUs as the initial spanning bucket (spanning over g GPUs).

Given that a GPU can only be involved in at most two spanning buckets (one on each side of its GPU chunk buffer), the maximum possible number of spanning buckets that result from a partitioning pass is $g - 1$. In that case, the spanning buckets are spanning over two GPUs each: One spans from GPU 0 to GPU 1, the next spans over GPUs 1 and 2, and so on.

We perform a maximum of p partitioning passes in our radix partitioning phase, with $p = \lceil k/c \rceil$ (see Table 1), while in the first partitioning pass, we partition the initial input data as the only spanning bucket of that first pass. Thus, the upper bound for spanning buckets is $s_{max} = (g - 1) \cdot (p - 1) + 1$. As a result, the total number of buckets can not exceed $2^c \cdot s_{max}$. Given a reasonable number of bits to consider at a time, e.g. $c = 8$, this is equal to $256 \cdot s_{max}$.

For each spanning bucket, we perform the following operations on each GPU that contains keys of that spanning bucket: We compute and store the histogram, we perform the local partitioning of keys into their respective buckets (i.e. key scattering), we exchange the histograms between the involved GPUs, and we determine the logical distribution of buckets. The latter step includes computing the multi-GPU-striped histogram and the bucket-to-destination-GPU mapping.

The number of spanning buckets also influences the memory allocations necessary. For each spanning bucket, we store many different data structures on each GPU; including the g individual histograms of every GPU and the multi-GPU-striped histogram. We explain how exactly the upper bound on the number of spanning buckets translates into the memory overhead of our algorithm implementation in Section 4.5.

3.3 On-GPU Bucket Sorting

After the P2P key swap, each GPU contains complete buckets with keys of distinct value ranges. While the keys of different buckets are correctly sorted given that the buckets are ordered by their MSB radix values, the keys within each individual bucket are still unsorted. Depending on the number of partitioning passes performed before the key swap, we have already examined a certain number of most significant bits of each key. We use this information to accelerate the sorting computation. We use a single-GPU radix sorting algorithm to sort each bucket locally on its respective GPU. For each bucket to sort, we specify the bit range that the radix sorting algorithm sorts on. For example, when sorting 32-bit keys with $c = 8$, and a radix partitioning phase that required only one partitioning pass, we use a single-GPU radix sort algorithm to sort the final buckets on the remaining least significant 24 bits: [24..0].

Since many spanning buckets can occur on different GPUs in different partitioning passes, the final bucket partitioning level is heterogeneous in the following sense: Some buckets are sufficiently partitioned after the first partitioning pass already while others are much more refined through multiple partitioning passes. This becomes clear in the example in Figure 5, where the complete buckets that will be distributed to GPU 3 stay untouched after the first partitioning pass while the spanning bucket `[0]` is subject to refining partitioning rounds. The heterogeneous and also recursive nature of the partitioning is also indicated by the bucket notation that we employ: $\texttt{[b}_0\texttt{:b}_1\texttt{:(...):b}_{p-1}\texttt{]}$. It represents a bucket's partitioning history by enumerating the identifiers of *parent*-buckets that the bucket was a part of, colon-separated from the first to the last partitioning pass.

As a consequence, for each complete bucket that we sort, we have taken a different number of most significant bits into account already during the radix 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 in the radix partitioning phase, we do not need to sort the bucket at all because all of its bits have been taken into account already. In contrast to sorting on the entire bit range that includes all k bits per key, specifying a reduced bit range improves the sorting performance of the local radix sort algorithm significantly. We evaluate the impact of the reduced bit range on the sorting duration in Section 4.4.

Many buckets result in many radix sort kernel launches. The overhead of these kernel launches, however insignificant for a single launch, can add up to have a negative performance impact for too large numbers of buckets, especially when the buckets contain very small numbers of keys. In order to reduce the total number of

buckets and mitigate the associated overhead, we fuse neighbouring buckets whose number of keys is below a certain threshold SBT (small bucket threshold). We can only fuse neighbouring buckets and still preserve the buckets' global sort order.

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 fuse buckets of the same partitioning pass only. As a result, the combined buckets share their initial bit range [*endbit*..0] which we extend by the necessary minimum, depending on the bits in which the two buckets' radix values differ. For example, when we fuse bucket `[33]`, representing keys that start with `00100001`, and bucket `[36]`, representing keys with `00100100` as their most significant bit, we extend the bit range by three bits. This is the case because the third least significant bit of the two bucket's radix values is the most significant bit in which they differ.

In general, the bit range is extended by the most significant bit position in which the two bucket values differ, starting from the least significant bit to minimize the bit range extension. We compute the bit range extension by performing an XOR operation on the two buckets' radix values (from 0 to 2^c) and counting the left shift operations that we perform on the result until it becomes equal to zero.

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, effectively overlapping the sorting computation with the device-to-host copy operation.

4 Implementation

In this section, we give detailed information about how we implement our radix-partitioning-based multi-GPU sorting algorithm (RMG sort). It is designed to scale inherently well to increasing numbers of GPUs. Still, each kernel function needs to be fine-tuned and its performance highly optimized to achieve a competitive end-to-end sort duration. In this context, we explain the performance optimizations that we employ in order to leverage the GPU's hardware capabilities. Across this section, we reference important variables and constants that we use in our implementation. We give an overview of our relevant variables' notations and values in Table 2.

Table 2: RMG sort variables and constants

Symbol	Description	Value
c	Number of bits considered at a time	8
NUM_B	Number of buckets per spanning bucket	$2^c = 256$
KPT	Keys per thread	12 for 32-bit keys, and 6 for 64-bit keys
NUM_{TPB}	Number of threads per thread block	1024
NUM_{TB}	Number of thread blocks	$\lceil \lceil n/g \rceil / (\text{KPT} \cdot \text{NUM}_{TPB}) \rceil$
NUM_{TPW}	Number of threads per warp	32
NUM_{BHA}	Number of consecutive, block-local histograms aggregated per thread block	256
ϵ	GPU chunk buffer padding	$0.5\% \cdot \lceil n/g \rceil$
SBT	Small bucket threshold	$1\% \cdot \lceil n/g \rceil$
MAX_{BRS}	Maximum number of buckets for the reduced-sorting optimization	128
MIN_{SCO}	Minimum number of buckets for the sort-copy-overlap	4

4.1 Histogram Computation

After the input data has been copied to the GPUs in chunks, the first step of the radix partitioning phase is for each GPU to compute the histogram on its keys on the most significant c bits. To compute the histogram, we need to read all $\lceil n/g \rceil$ keys of the GPU's chunk. For each key, we increment one of the 2^c bucket counters in our histogram array depending on the radix value of the key's most significant c bits. When parallelizing the computation with thousands of thread blocks, we assign each thread block an equal number of keys to process: KPT $\cdot$ NUM_{TPB} (see Table 2). The involved arithmetic operations to find out the correct bucket of a key are cheap to perform (logical AND operations and bit shifting). The main performance-related challenge of the histogram computation is to find a way to efficiently orchestrate the many atomic operations that are performed concurrently by all threads on the small histogram array of size 2^c.

When each thread block performs globally atomic increments on a single global histogram that is stored in device memory and accessed by all threads concurrently, the atomic load is too high. At any time during the execution, nearly all threads would be stalled, waiting for their atomic operation to finish. To achieve peak performance instead, it is crucial to utilize each thread block's shared memory. We split the histogram computation into two kernel functions. In the first kernel (`ComputeHistogram`), each thread block first computes a thread-block-local histogram that is stored in its shared memory. Atomic operations on shared memory are significantly faster than on global device 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.

The second kernel (`AggregateHistogram`) aggregates the block-local histograms and therefore needs to perform global atomic operations. To reduce the number of performed global atomics, we divide the NUM_{TB} block-local histograms into subsequent groups of size NUM_{BHA} (see Table 2). We launch just enough thread blocks so that each block pre-aggregates one such group of NUM_{BHA} block-local histograms before performing the global atomic operation.

To simplify the parallelization within each thread block, we utilize only the first 2^c out of the NUM_{TPB} threads. This allows us to assign each thread of a thread block to aggregate one bucket counter only – the one that corresponds to its thread index. Each thread iterates over and sums up the corresponding bucket counters of its NUM_{BHA} block-local histograms using a single counter variable in the register. Then, the thread performs one atomic add operation to the respective bucket counter in the GPU-global histogram in global memory – incrementing the global

counter by the sum of its block-local counters. In that way, we reduce the number of global atomic operations by a factor of NUM_{BHA}.

Our employed memory read pattern to the block-local histograms, that are stored in the relatively slow global memory, is perfectly warp-aligned because the accesses are orchestrated using the thread index as the read offset. Consequently, the threads of a warp read from consecutive memory addresses, and the number of necessary memory transactions is minimal.

Handling Data Skew. By splitting the GPU-global histogram computation into two kernel functions (compute local histograms and aggregate), we significantly reduce the global atomic operations, and efficiently utilize shared memory. For very skewed distributions, the shared memory atomic operations on the block-local histogram are under increased pressure. In the most extreme case, where all keys that a thread block processes are the same in the currently considered c bits, all NUM_{TPB} threads increase the same bucket counter of the block-local histogram. This results in high contention for the atomic add operations between the threads of that thread block.

To mitigate the performance degradation for skewed data distributions, we employ the following lightweight optimization: Each thread stores its first key's bucket value and holds back the atomic increment for that first bucket. Instead, we initialize an additional counter variable (`start_bucket_count`) in the register to be equal to one. For every following key, we first check if its bucket is equal to the first key's bucket and if so, we increment the `start_bucket_count` by one instead of performing an atomic increment. After the thread finished iterating over its KPT keys, we perform the postponed shared atomic add operation: We atomically increment the first key's bucket counter by `start_bucket_count`. This optimization heavily mitigates the performance drop that skewed distributions would otherwise cause. Even if the data is moderately skewed, the probability of reducing the number of performed atomic operations is still high enough. After this optimization, we measure the execution time of the histogram computation on skewed data to be equal to that on uniformly distributed keys. The performance optimization is lightweight in the sense that it does not introduce a measurable overhead for those cases where the optimization does not improve the performance (i.e. for uniform distributions where `start_bucket_count` does not exceed one).

4.2 Key Scattering

After the histogram computation, the key scattering is the second step of each partitioning pass. Each GPU locally partitions its keys in its device memory

based on their radix values in the most significant c bits so that all keys of bucket i precede all keys of bucket $i+1$, for all $i \geq 0$ and $i \leq 2^c$.

In contrast to the histogram computation, we not only need to read the n keys of the GPU chunk, but also write them back to global memory. To avoid synchronization between reading from and writing to the same memory buffer via many threads, we perform the key scattering step out-of-place. We allocate two buffers of size $n + 2 \cdot \epsilon$ and use these two buffers as alternating input/output buffers, keeping track of which buffer is currently considered as the input (i.e. the current key buffer) and which one is the output buffer (i.e. the buffer where the current partitioning pass writes the result into).

Similar to the histogram computation, we launch the `ScatterKeys` kernel with many thread blocks and assign a small distinct subset of the input keys KPT to each thread. The threads process their assigned keys one after another. For each key, we determine the bucket that it belongs to based on the currently considered c (most significant) bits. Depending on the bucket it belongs to, each key needs to be written to different positions in the output buffer in global memory. For data distributions with little skew, the keys that a thread processes are very likely to belong to different buckets. Therefore, writing keys back to global memory one by one results in random write patterns, and thus, poor performance. To avoid random write patterns, we first pre-scatter all the keys of a thread block into their respective buckets in shared memory. This allows each thread block to write its pre-scattered buckets back to global memory one after another. As a result, the global memory write pattern is sequential for the keys of the same bucket.

Both, pre-scattering the keys in shared memory, and writing them back to global memory, require knowledge of the exact memory offsets for where to write the keys of each bucket. We determine both using our computed histogram data structures. We launch the `ComputeHistogram` kernel with the same number of threads and thread blocks as the `ScatterKeys` kernel. In that way, we can re-use the block-local histograms from our histogram computation step, not only the GPU-global histogram.

4.2.1 Global and Shared Memory Write Offsets

Before launching the `ScatterKeys` kernel, we compute the prefix sum on the GPU-global histogram. For this, we use the `thrust::exclusive_scan` primitive provided by NVIDIA's parallel computing library Thrust [35, 49]. The prefix sum on the GPU-global histogram gives us the starting position for each full bucket in the global memory output buffer, which we refer to as the *starting global write offsets*.

They determine the global memory output buffer index of each bucket's first key for the 2^c buckets of the entire GPU chunk of $\lceil n/g \rceil$ keys.

Since we divide the key scattering workload among NUM_{TB} thread blocks, each thread block writes only a subset of the keys for each of the GPU-global 2^c buckets. All thread blocks concurrently scatter their share of keys for each bucket. We orchestrate the concurrent write access of NUM_{TB} thread blocks as follows. At the beginning of the `ScatterKeys` kernel, each thread block initializes its exact global write offsets. For each bucket b, the thread block atomically increments the *starting global write offset* of bucket b by the number of keys it will write to. The block-local histogram gives us exactly that information as it contains the number of keys per bucket for a given thread block. This first initialization step ensures that the global memory write-back phase of the kernel is performed without further synchronization.

Before the global memory write-back, we pre-scatter the keys in a shared memory buffer of size $\text{KPT} \cdot \text{NUM}_{TPB}$. As part of the kernel initialization, we also compute each bucket's starting position in the shared memory buffer, i.e. the *starting local write offsets*. For this, each thread block computes the prefix sum on its block-local histogram. We implement the prefix sum computation on the block-local histogram sequentially in a tight for-loop and measure an insignificant execution time. After the prefix sum computation, we continue with the pre-scattering of keys in shared memory.

4.2.2 Shared Memory Pre-Scattering

Each thread iterates over its assigned KPT keys. For each key, we check the currently considered c (most significant) bits to determine the bucket b that the key belongs to. We determine the exact shared memory write offset for the current key value to be written to by atomically reading the current *starting local write offset* of b, and incrementing it by one. We write the key's value to the shared memory offset that we read before the increment. Any subsequent writes by another or the same thread will be performed on the incremented offset, avoiding write conflicts. With this approach, we deliberately perform significantly more atomic operations on shared memory than on global memory because they are substantially faster in shared memory.

The limited shared memory size (128 KB on the NVIDIA Tesla V100, and 196 KB on the NVIDIA A100 GPU) sets an upper bound for the number of keys that each thread block can pre-scatter. We configure our algorithm implementation to process twelve 32-bit keys per thread, and each tread block to run 1024 threads.

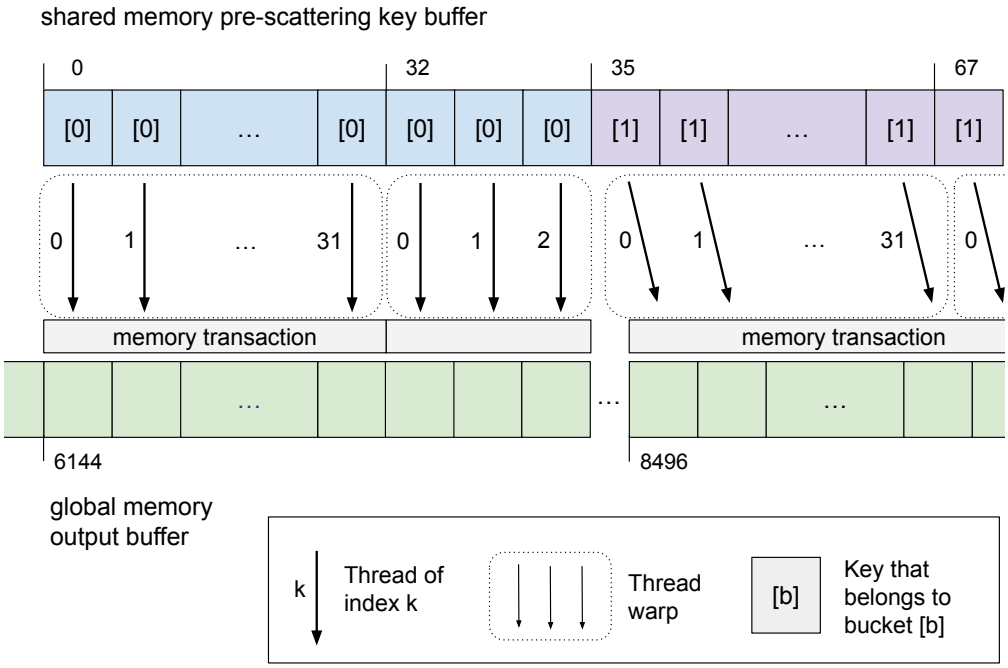


Figure 6: A thread warp writes back pre-scattered keys from shared to global memory.

For 64-bit keys, each thread processes six keys. This results in a shared memory usage of 12 keys · 4 Bytes ·1024 threads = 49.152 KB. The remaining memory is used as the L1 cache. We find that this configuration achieves the best performance on our evaluated GPUs.

4.2.3 Global Memory Write-Back

The pre-scattering step in shared memory substantially accelerates the global memory write-back because it allows us to write back keys of the same bucket in a sequential pattern rather than scattering individual keys to random addresses. We optimize the global memory throughput even further by implementing the write-back in a warp-aligned way.

We orchestrate the global write-back so that each thread warp is responsible for writing back a small, constant number of consecutive buckets, one after the other. Figure 6 illustrates how one thread warp of a thread block writes back the pre-scattered keys of buckets `[0]` and `[1]` as an example. In the example, other thread warps would concurrently write back the remaining buckets. For each bucket, the associated thread warp writes its keys in such a way that the number of necessary memory transactions is minimal (i.e. cache-optimal). To achieve this, each thread

warp's thread writes one of $\text{NUM}_{TPW} = 32$ consecutive keys of the current bucket, iterating over the bucket's keys in steps of 32. As explained in Section 2.1, 32 is the number of threads that run concurrently as part of one thread warp. Since all threads of the same warp execute the same instruction concurrently, and their memory accesses are grouped, all 32 threads of a warp write a consecutive block of 32 keys at the same time. This results in warp-aligned memory accesses, and peak GPU memory bandwidth utilization.

However, as indicated in Figure 6, the bucket size can negatively influence the achieved memory throughput. When a thread block's local bucket is not empty but contains less than 32 keys, some threads of the warp idle. If many buckets contain very small numbers of keys, the memory throughput drops considerably. Therefore, it is desirable that all non-empty buckets should contain enough keys to fill at least one full memory transaction of a thread warp.

Because the shared memory size is limited, we set the number of keys per thread block to be constant. Thus, the number of consecutive bits c that we consider per partitioning pass influences on how many keys can fall into each bucket. For example, if we chose to partition on $c = 16$ bits in each partitioning pass, the number of possible buckets $2^{16} = 65,536$ would be higher than the number of keys that a thread block processes ($\text{KPT} \cdot \text{NUM}_{TPB} = 12{,}288$). This would drastically reduce the achieved memory throughput of our global memory write-back because most of the buckets would contain very few keys – if any – for uniform distributions. If we set c to be too small, we increase the number of necessary partitioning passes, ultimately decreasing the total sort duration as well. We find that configuring $c = 8$ is an ideal trade-off between minimizing the number of partitioning passes and maximizing the achieved memory throughput of the `ScatterKeys` kernel. Thus, we confirm the findings of Stehle et al. who configure their single-GPU MSB radix sort to take 8 bits into account at a time as well [63].

We measure our global write-back to achieve a global memory throughput between 70% and 95% of the GPU's peak memory bandwidth on the NVIDIA A100 GPU. The total execution time of the `ScatterKeys` kernel depends not only on the global write-back but also on the shared memory atomic operations performed during the pre-scattering of keys. We evaluate the performance of our key scattering kernel, our histogram computation, and other steps of our algorithm in detail in Section 5.

4.2.4 Scatter and Swap

In the first partitioning pass, we scatter all keys of the GPU chunk based on the most significant c bits. While the partitioning phase proceeds, we only refine the

partitioning of spanning buckets. That is, we perform partitioning passes on those parts of a GPU chunk that belong to a spanning bucket. In the first partitioning pass, we view the entire input as the initial spanning bucket that is spanning all g GPUs, and the entire chunk of each GPU.

After the first partitioning pass, it depends on the input data distribution how many keys we scatter in subsequent partitioning passes. If a subsequent partitioning pass is performed on a spanning bucket that is not taking up the entire chunks of the GPUs that it spans, we only scatter parts of the chunks' keys. Since we perform the key scattering out-of-place, we write the scattered keys into the output buffer of our alternating double buffer. Thus, we need to copy all the remaining keys of the current input buffer to the output buffer as well. Thereby, the partitioning pass aligns its output keys in one complete buffer. We implement the key swap to be interleaved to the key scattering, i.e. we launch the `ScatterKeys` kernel to run concurrently to the issued device-local CUDA memory copy operations in separate CUDA streams. After both streams finish executing, we flip the alternating double buffer's input and output pointers, as the current output will be treated as the input for the next partitioning pass or, if sufficiently prepared, the P2P key swap.

We measure that the device-local CUDA copy operations execute orders of magnitude faster than the `ScatterKeys` kernel. Thus, the *scatter and swap* execution time is bound by the `ScatterKeys` kernel execution.

For skewed distributions, partitioning pass can generate only one bucket. For example, if all n 32-bit input keys have leading zeros in their most significant eight bits, the first partitioning pass will find that all keys on each GPU belong to bucket `[0]`. It is only in the second partitioning pass that the histogram computation will identify a difference in the keys' radix values. If, in any case, the histogram computation on a GPU determines that all keys of its part of a current spanning bucket belong to the same bucket, we skip the key scattering step completely. Since a GPU can be a part of two spanning buckets, we perform up to two histogram computations per GPU per partitioning pass. If during a partitioning pass, a GPU skipped all of its `ScatterKeys` kernel launches, we do not swap any keys between the input and the output buffer. Consequently, we do not flip the alternating double buffer's input and output pointers, as the input buffer still contains the unchanged keys.

4.3 P2P Key Swap

At the end of a partitioning pass, each GPU locally computes the distribution of buckets across the g GPUs. For this, the GPUs exchange their GPU-global histograms via P2P transfers. Once a GPU computed its histogram, we launch the CUDA copy operations that transfer its histogram to every other GPU via the P2P interconnects. During the histogram exchange, we concurrently run the `ScatterKeys` kernel. By interleaving these two operations, we hide the time duration of the histogram exchange. Since the histogram of one GPU is only $256 \cdot 8$ Bytes ≈ 2 KB in size, we measure the histogram exchange time to be negligible.

Once a GPU receives the histograms of all other GPUs, it constructs the multi-GPU-striped histogram and computes the prefix sum on it. Finally, each GPU determines the bucket distribution across the GPUs by computing the mapping of buckets to their destination GPUs. For both, the multi-GPU-striped histogram and the bucket mapping, we implement one kernel function respectively. For the prefix sum computation, we again use the `thrust::exclusive_scan` primitive. We interleave the execution of the three kernels with the `ScatterKeys` kernel. Similar to the histogram exchange, we measure insignificant execution times for all three kernels, with runtimes of up to 3μs when sorting a total of 2 billion keys on two GPUs.

As explained in Section 3.2.2, we employ nearly-perfect load balancing with regards to distributing the complete buckets across the g GPUs. Experimentally, we determine that setting our ϵ padding value to 0.5% of the initial GPU chunk size achieves the best performance. With this configuration, uniformly distributed keys need only one partitioning pass on their most significant c bits.

When the radix partitioning phase is completed, we perform the P2P key swap. For each complete bucket, we asynchronously launch the CUDA memory copy operation to the bucket's destination GPU. We perform the P2P key swap out-of-place, utilizing our alternating double buffer from the key scattering step. This allows us to utilize the P2P interconnects' bidirectional bandwidth without synchronization. We interleave the P2P transfers of all g GPUs, as each GPU transfers its buckets simultaneously. Thus, our P2P key swap benefits from modern P2P interconnects, such as NVLink-based NVSwitch interconnects because they allow for high-speed non-blocking all-to-all communication between all system-wide GPUs. For those buckets whose destination GPU is the same as their initial GPU, we perform device-local copies from the input to the output of our alternating double buffer.

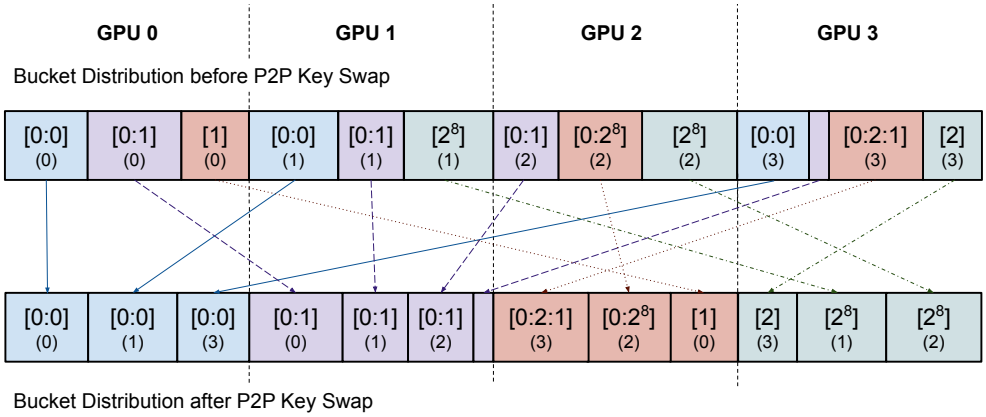


Figure 7: P2P Key Swap Example

In the example of a P2P key swap depicted in Figure 7, each GPU only contains a few buckets to swap. When sorting reasonably large data sets that are uniformly distributed, each GPU transfers 256 buckets. Then, a GPU issues multiple CUDA memory copy operations to the same destination GPU (one for each bucket that is transferred to that same GPU). We test the performance impact of a bucket batching approach where we, for each GPU, first locally align its buckets based on the destination GPU that they will be transferred to (i.e. the so-called *bucket batch*). For this, we copy all buckets' keys within each GPU's device memory locally from the input to the output buffer in such a way that we can afterwards reduce the number of issued CUDA copy operations: A GPU copies the entire batch of buckets for each GPU that it needs to transfer at least one bucket to.

We find that this strategy does not improve the performance. In fact, the overhead of an asynchronous CUDA copy operation, whether performed as a P2P transfer or a local device memory copy, is negligible even for high numbers of buckets. The CUDA driver appends each CUDA call into the specified CUDA stream's queue and is able to perform the issued copy operations one after the other at peak bandwidth, without a significant delay or latency. Therefore, we discard the bucket batching approach, and simply launch a CUDA copy for each bucket. This avoids the need to align the buckets locally according to their destination GPU prior to the P2P transfers, and thus, simplifies the code without any performance sacrifice.

Last-pass spanning buckets are the exception with regards to our nearly-perfect load balancing approach. For those spanning buckets, our kernel function that determines the bucket-to-destination-GPU mapping returns more than one destination GPU. For each spanning bucket that resulted from the last partitioning

pass, we divide it up into fractions so that we achieve perfect load balancing across its destination GPUs. We can choose arbitrary borders for splitting up a last-pass spanning bucket because such a bucket is filled with the same key value. We split up the last-pass spanning buckets into fractions right before the P2P key swap, and then transfer them just as we transfer the ordinary, complete buckets.

4.4 Bucket Sorting and Copy-Back

Once the P2P key swap is completed, each GPU sorts its buckets and copies them back to the CPU memory. For sorting each bucket, we employ the fastest state-of-the-art single-GPU sorting algorithm. Maltenberger et al. recently publish an evaluation study of merge-based multi-GPU sorting algorithms in which they also evaluate the performance of state-of-the-art single-GPU sorting algorithms [34]. They find that the LSB radix sort algorithm provided in NVIDIA's CUB library achieves the fastest performance [41], sorting one billion 32-bit integer keys in 36ms on the NVIDIA A100 GPU. Thus, we implement our bucket sorting step using `cub::DeviceRadixSort::SortKeys` as our sorting primitive. It uses the same underlying radix sort algorithm that is used within `thrust::sort` from NVIDIA's parallel algorithms library Thrust [44].

As explained in Section 3.3, we accelerate the sorting computation for each individual bucket by reducing the bit range that we sort the bucket's keys on, given that we already looked at a certain number of most significant bits in the radix partitioning phase. The `cub::DeviceRadixSort::SortKeys` function conveniently allows for passing a custom bit range as a parameter. The magnitude of the performance improvement of CUB's radix sort primitive for reduced bit ranges depends on how many bits still need to be considered. We perform a micro-benchmark of CUB's radix sort primitive to evaluate the performance impact of the reduced bit range and depict our results in Figure 8. We observe that specifying a reduced bit range, i.e. sorting on fewer bits per key, significantly improves the sorting performance.

Since there is a performance overhead associated with launching a kernel function (in this case the CUB's radix sort primitive), we only want to sort each bucket individually, when the total number of buckets on a GPU is below a certain threshold MAX_{BRS} (see Table 2). Otherwise, when there are more than MAX_{BRS} buckets, the kernel launch overhead adds up to a significant amount, and it becomes faster to simply call CUB's radix sort primitive on the entire GPU chunk, disregarding the information gained during the radix partitioning phase. To avoid losing the buckets' partitioning information, we fuse neighbouring buckets of the same partitioning pass that are too small, i.e. have less than SBT keys (see Table 2). We find that for an SBT equal to 1% of the initial GPU chunk size, we rarely

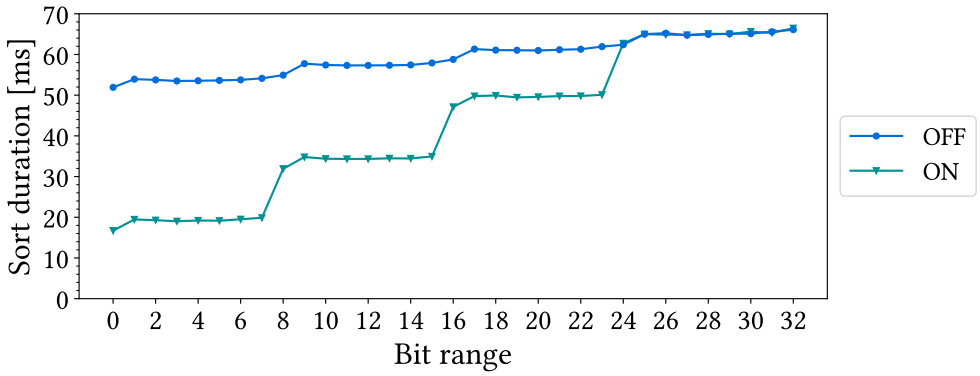


Figure 8: Sorting performance of `cub::DeviceRadixSort` for reduced bit ranges on the NVIDIA Tesla V100 GPU. (1 billion 32-bit unsigned integer keys)

end up with too many buckets. Consequently, we benefit from the reduced-sorting optimization in most cases.

The second optimization that we implement, given that we sort each GPU's buckets individually, is the overlap of the sorting computation and the device to host copy operation (as mentioned in Section 3.3). Starting from the first bucket of each GPU chunk, we sort up to two subsequent buckets concurrently. As soon as a bucket is sorted, we asynchronously launch the CUDA memory copy to transfer the latest sorted range of keys back to main memory.

Some buckets of the final sorting phase can be the result of the last partitioning pass. For these buckets, we already looked at all of their keys' k bits during the partitioning phase, and thus do not need to sort them anymore. We still, however, need to copy these buckets back to main memory. To account for this, we keep track of the memory address up to which we have copied the sorted keys back to the CPU. For each newly sorted bucket, we copy back all of the yet unreturned keys up to the latest sorted bucket's last key.

For current interconnects, the sort-copy-overlap optimization hides the time duration of the sorting computation because the device to host copy takes longer. This is the case, even when the GPUs are connected to main memory via NVLink 2.0 interconnects. Given that there are always sorted buckets ready to transfer in each loop iteration, the device to host copy operation can be viewed as a transfer of a total of x Bytes in chunks of size x/t Bytes. We find that there is an optimum t for which the achieved copy throughput of the chunked transfer is equal to the direct transfer of all x Bytes in one CUDA copy call. In our case, the factor t is

determined by the number of buckets and their size. We reduce the total number of buckets by fusing small, neighbouring buckets. We configure the small bucket threshold (SBT) to ensure an optimal number of buckets for data distributions that generate many buckets (e.g. uniform distributions). We find that a value of 1% of the initial GPU chunk size works well.

On the other hand, if the number of buckets is too small, overlapping the sorting computation and the device to host copy does not improve the performance, as each operation takes too long to achieve efficient overlap. Since we cannot split buckets without introducing the necessity of merging them later on, we turn off the sort-copy-overlap optimization if a GPU contains less than or equal to MIN_{SCO} buckets (see Table 2). We measure the device to host copy throughput to reach optimal throughput rates for most data distributions. We evaluate this and other performance characteristics and implications of RMG sort in-depth in Section 5.4.

The `cub::DeviceRadixSort::SortKeys` primitive works out-of-place and requires auxiliary memory in $O(n)$, plus a small additional memory overhead that depends on the number of streaming multiprocessors on the device. We re-use our alternating double buffer to avoid dynamic memory allocations during the algorithm execution. For each bucket to sort, we use a distinct block of memory from our alternating double buffer's output buffer as the auxiliary memory.

4.5 GPU Memory Implications

We avoid dynamic CUDA memory allocations, both on the device and the host, during our algorithm execution. They are comparatively slow and would increase the total end-to-end sort duration. Furthermore, dynamic CUDA memory allocations implicitly synchronize CUDA operations from all streams that could otherwise run concurrently to each other. Thus, to achieve peak performance, we pre-allocate the necessary device memory, as well as pinned host memory, assuming dedicated accelerators whose entire GPU memory is reserved for the database system to run.

In this section, we lay out the memory overhead that our radix-partitioning-based multi-GPU sorting algorithm implementation entails. First, we allocate the memory for the two buffers of our alternating double buffer. Any GPU-accelerated sorting algorithm that relies on CUB's out-of-place radix sort primitive has a space complexity lower bound of $O(2n)$.

For our nearly-perfect load balancing approach, each of the two buffers needs to store $n + 2\epsilon$ keys. Furthermore, we add 128 MB of memory to each buffer to ac-

count for the additional memory overhead that `cub::DeviceRadixSort::SortKeys` needs. The 128 MB of additional constant memory space is enough to sort any data set size that fits onto modern GPUs. When sorting two billion 32-bit integer keys per GPU, the double buffer allocates $2 \cdot 4$ Bytes per key $\cdot\ 2 \cdot 10^9 = 16$ GB of memory just for sorting exactly n keys. Then, the 2ϵ padding overhead makes up only 20 MB per buffer (i.e. 40 MB in total). Together with the constant sorting overhead of 128 MB per buffer, this constitutes less than 1.85% of the 16 GB that are required for sorting the input keys out-of-place.

Additionally, we allocate the necessary memory for the histogram and bucket data structures that we manage during our radix partitioning phase and beyond. In Section 3.2.4, we already defined the upper bound for the number of buckets that our algorithm needs to handle. The upper bound for spanning buckets is $s_{max} = (g-1) \cdot (p-1) + 1$. Since we cannot know in advance how many spanning buckets a given input key distribution will generate, and which GPUs the spanning buckets will span, we allocate enough data structures for s_{max} spanning buckets on each GPU. For each spanning bucket, each GPU stores the data structures that are shown in Table 3.

Table 3: Data structures per spanning bucket per GPU

Data structure	**Data type**	**Array size**	**Memory size**
GPU-global histogram	64-bit unsigned integer	2^c	2048 B
Prefix sum on the GPU-global histogram	64-bit unsigned integer	2^c	2048 B
Global key scatter write offsets	64-bit unsigned integer	2^c	2048 B
Thread block-local histograms	32-bit unsigned integer	$\text{NUM}_{TB} \cdot 2^c$	$\text{NUM}_{TB} \cdot$ 1024 B
g GPU-global histograms (histogram exchange)	64-bit unsigned integer	$g \cdot 2^c$	$g \cdot 2048$ B
Multi-GPU-striped histogram	64-bit unsigned integer	$g \cdot 2^c$	$g \cdot 2048$ B
Bucket-to-destination-GPU map	32-bit integer	$g \cdot 2^c$	$g \cdot 1024$ B

When sorting two billion 32-bit integer keys on each GPU with $g = 8$ GPUs for a total input size of 16 billion keys, the number of thread blocks that we launch for the histogram computation and the key scattering kernels is $\text{NUM}_{TB} = 162.761$. Thus, the most significant data structure in terms of the memory overhead is the thread block-local histograms. For 32-bit keys, the maximum number of partitioning passes is $p = 4$. Thus, the upper bound for the number of spanning buckets is $s_{max} = 22$. The total memory overhead that results from allocating all of our histogram and bucket data structures listed in Table 3 for 22 potential spanning buckets is $22 \cdot 166,7$ MB = 3,668 GB – 22% of the 16 GB that are required for sorting the two billion input keys out-of-place.

The thread block-local histograms are not used after the associated `ScatterKeys` kernel finishes execution. Thus, all block-local histograms only need to be stored during their respective partitioning pass. Their memory buffers can therefore be re-used, which significantly reduces the number of block-local histogram buffers that we need to allocate on each GPU from s_{max} down to $g - 1$. Given the example from above, where we sort a total of 16 billion keys with $g = 8$ GPUs, the total memory overhead of our histogram and bucket data structures would be reduced to 7% of the 16 GB allocated by the double buffer. We propose to implement this memory overhead reduction as part of future work.

5 Evaluation

In this section, we evaluate the performance of our radix-partitioning-based multi-GPU sorting algorithm (**RMG sort**). In Section 5.2, we compare its sorting performance with state-of-the-art parallel CPU-only sorting algorithms. In Section 5.3, we compare our radix-partitioning-based algorithm with two state-of-the-art merge-based multi-GPU sorting algorithms from the literature. Finally, we analyze the performance of RMG sort for different data distributions, including skewed data, in-depth in Section 5.4.

5.1 Experimental Setup

In this section, we explain our experimental setup. This includes the hardware systems that we evaluate our multi-GPU sorting algorithm on, our employed experimental methodology, as well as defining our sorting algorithm baselines.

5.1.1 Hardware Systems

For our experimental evaluation, we use two modern multi-GPU accelerator platforms that are equipped with state-of-the-art interconnect technology. We provide the basic hardware information for both systems in Table 4.

The first hardware system that we evaluate on is the IBM Power System AC922. It is a two-socket NUMA system that attaches two NVIDIA Tesla V100 GPUs to each NUMA node [37]. The interconnect topology of the IBM AC922 is shown in Figure 9. We report the bandwidth rates of the different interconnects per direction. Since all involved interconnects are bidirectional, their aggregated bidirectional bandwidth is twice as high. On the IBM AC922, data transfers between one NUMA node and its two local GPUs are powered by the high-speed NVLink 2.0 interconnect for a theoretical bandwidth of 75 GB/s per direction. The IBM AC922 is currently the only system that uses NVLink 2.0 for CPU-GPU interconnects. Furthermore, the two GPUs of each CPU-local GPU pair achieve a high P2P copy throughput as they are interconnected with NVLink 2.0 as well. The system's data transfer bottleneck lies in the NUMA-interconnect as the X-Bus bandwidth is comparatively low, compared to the CPU-GPU connections.

Our second evaluation system is the NVIDIA DGX A100 [42]. It comes with a total of eight NVIDIA A100 GPUs which are interconnected with NVLink 3.0-based NVSwitch for high-speed inter-GPU communication. We show the system's interconnect topology in Figure 10. The NVIDIA A100 GPU is currently NVIDIA's latest server-grade high-performance GPU, while NVLink 3.0 is their latest interconnect. Connected via NVSwitch, the DGX A100 allows full non-blocking

Table 4: Hardware systems overview

	(a) IBM Power System AC922	(b) NVIDIA DGX A100
CPU	2x IBM POWER9 16 × 2.7 GHz	2x AMD EPYC 7742 64 × 2.3 GHz
GPUs	4x NVIDIA Tesla V100 SXM2 32 GB	8x NVIDIA A100 SXM4 40 GB
RAM	2x 256 GB DDR4	2x 512 GB DDR4
OS	RHEL 8.2	Ubuntu 20.04
ISA	ppc64le	x86_64
Tools	CUDA 11.2 GCC 10.2.1	CUDA 11.4 GCC 9.3.0

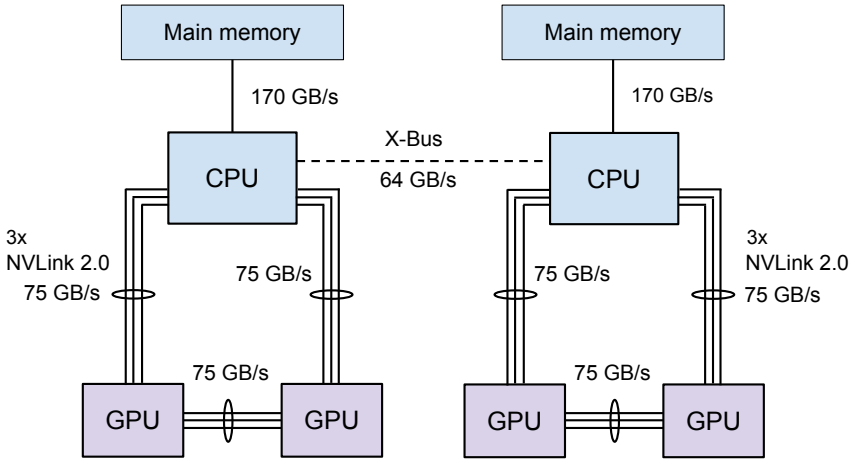


Figure 9: Interconnect topology for the IBM Power System AC922

all-to-all P2P transfers between the eight GPUs at 300 GB/s per direction. For this system, the PCIe 4.0 CPU-GPU interconnects are the data transfer bottleneck. They limit the host-to-device and device-to-host copy throughput, especially for neighbouring pairs of GPUs that share a PCIe switch, as seen in the system's interconnect topology depicted in Figure 10. For example, when we perform a parallel data transfer from CPU 0 to GPUs 0 and 1 concurrently, the throughput can

not exceed the theoretical bandwidth rate of one PCIe 4.0 instance (i.e. 32 GB/s) because the two GPUs share a PCIe switch.

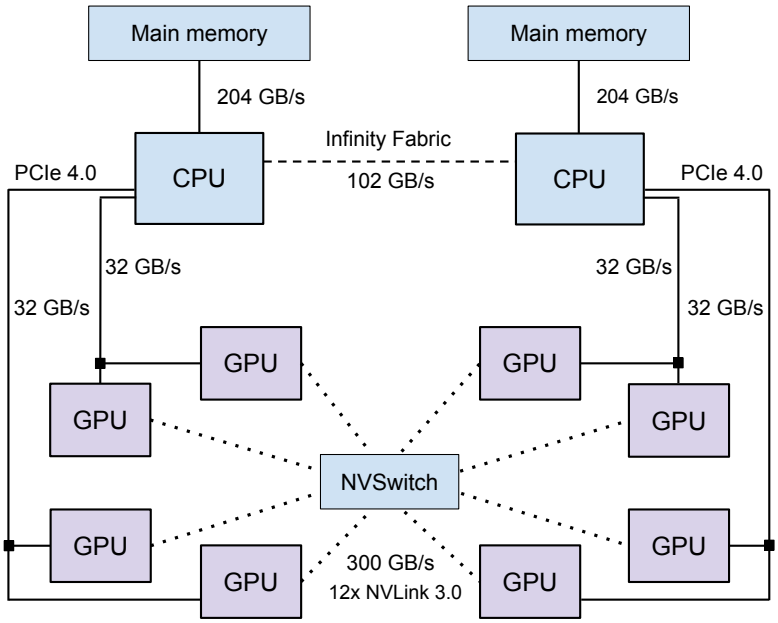


Figure 10: Interconnect topology for the NVIDIA DGX A100

5.1.2 Experimental Methodology

For all experiments, we measure the total end-to-end sort duration which includes the time durations for transferring the data from main memory to the GPUs and copying the output data back to the CPU. We run every experiment five times and report the arithmetic mean across all runs. Our experiments results are stable as we measure that the standard error across all runs is less than 4% from the mean.

In all experiments, we generate the input data in the main memory of NUMA node 0 and return the output to that same key buffer. Our standard data distribution is the uniform distribution, and by default, we sort unsigned 32-bit integer keys. However, we clarify exactly which data sets we sort for each experiment across the entire section.

As mentioned in Section 4.5, we pre-allocate the GPU's device memory buffers as well as pinned host memory because we assume dedicated accelerators that are reserved for the database system exclusively. For a fair comparison, we proceed in that manner for our proposed multi-GPU sorting algorithms as well as for the two state-of-the-art merge-based multi-GPU algorithms and the single-GPU

sorting algorithm baseline. Furthermore, for all experiments that include GPUs, we allocate pinned host memory because pinned memory accelerates the CPU-GPU data transfers compared to pageable memory buffers. Pinning the host memory significantly increases the achieved throughput rate because then, the GPU's copy engine accesses the host memory via direct memory access (DMA) [32, 34, 45, 51].

To enable reproducible evaluation results, we publish the source code of our multi-GPU sorting algorithm together with benchmarks scripts to automatically run the experiments and generate plots for the results[1].

5.1.3 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 multi-GPU sorting algorithm on the IBM AC922, the optimal two-GPU-sets are the CPU-local GPU pairs $(0, 1)$ and $(2, 3)$ because of the direct NVLink 2.0 P2P interconnects. Given that we run our experiments starting from NUMA node 0, the most optimal 2-GPU set is $(0, 1)$. Similarly, the optimal four-GPU-set on the NVIDIA DGX A100 is $(0, 2, 4, 6)$ as it includes only one GPU of each GPU pair that shares a PCIe switch. Thus, the CPU-GPU copy throughput is maximized. Across our evaluation section, when we depict the measured performance for sorting with a given number of GPUs g, we always use the system's optimal g-GPU, and we mention which GPUs the optimal g-GPU set consists of.

For some experiments, we only depict the overall best performing GPU set of a multi-GPU accelerator platform. Oftentimes, a multi-GPU system achieves the best performance when accelerating with all its g_{max} system-wide GPUs. However, this is not always the case. On the IBM AC922 for example, we find that sorting any given data set is slower with four than with two GPUs because the low X-Bus bandwidth reduces the CPU-GPU copy throughput for remote GPUs. Thus, on the IBM AC922, the best performing GPU set is the GPU pair $(0, 1)$, when the input data resides in the main memory of NUMA node 0. For this GPU pair, the CPU-GPU data transfers achieve the peak copy throughput of the NVLink 2.0 interconnects. For more detailed multi-GPU data transfer benchmarks and an evaluation of the impact of a system's interconnect topology on the multi-GPU sorting performance, we refer to the work by Maltenberger et al. [34].

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

5.1.4 CPU Baselines

To compare the performance of RMG sort to the CPU's sorting performance, we use state-of-the-art parallel radix sort PARADIS as our CPU baseline [9]. Maltenberger et al. recently benchmarked the performance of state-of-the-art parallel CPU sorting algorithms and find that PARADIS achieves the fastest execution [34]. It is also platform-independent as it does not rely on hardware-specific SIMD instruction sets. PARADIS sort is an in-place parallel radix sort that employs speculative permutation followed by a repair phase to maximize parallelization. In our experiments, we find that it scales very well with the number of physical CPU cores.

Maltenberger et al. also evaluate the parallel sorting primitives of modern libraries, including the GNU parallel algorithms extension [61, 14], Intel's Thread Building Blocks library [53, 12], and the parallel C++17 extension of `std::sort`. They find that `gnu-parallel::sort` from the GNU parallel algorithms extension, a parallel multiway merge sort algorithm [14], is the fastest library primitive. Thus, we use it as our second CPU baseline.

5.1.5 Single-GPU Baseline

Our single-GPU radix sort baseline is `thrust::sort` [44], because it achieves the fastest sorting performance on a single GPU [34]. As mentioned in Section 4.4, `thrust::sort` uses the same underlying LSB radix sort as CUB's sorting primitive `cub::DeviceRadixSort` [41]. For sorting individual buckets in the sorting phase of RMG sort, we use `cub::DeviceRadixSort` since its interface allows for reducing the bit range. As our single-GPU baseline, we use Thrust's sorting primitive because it provides a simpler interface.

5.1.6 Multi-GPU Baselines

We compare the performance of RMG sort to two state-of-the-art merge-based multi-GPU sorting algorithms. The algorithm by Tanasic et al. utilizes P2P interconnects to merge sorted chunks within GPU memory [64]. The heterogeneous multi-GPU sorting algorithm by Gowanlock et al. uses the CPU to merge chunks that have been sorted on the GPUs [18]. The first multi-GPU baseline allows for a direct comparison of the utilization of P2P interconnects. With the second one, we compare our GPU-only algorithm to an approach that involves CPU computations. We compare RMG sort to our multi-GPU sorting baselines in Section 5.3

5.2 CPU Comparison

In our first experiment, we sort two billion uniformly distributed 32-bit integer keys with our proposed RMG sort, our single-GPU radix sort baseline, and our two parallel CPU sorting algorithm baselines. We depict the results for the IBM AC922 in Figure 11. We compare the performance of RMG sort to that of the CPU for the best performing interconnect-optimal GPU set of the IBM AC922; the GPU pair $(0, 1)$.

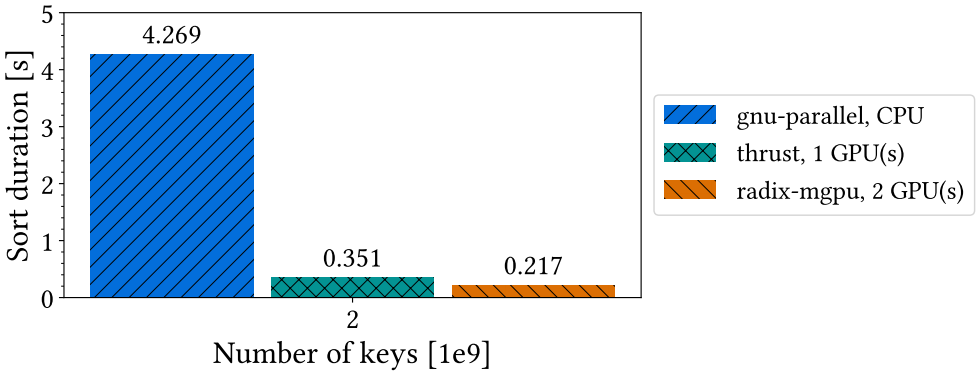


Figure 11: Sorting performance overview: CPU, single-GPU, and multi-GPU sorting algorithms on the IBM AC922

We observe that RMG sort (`radix-mgpu`) outperforms `gnu-parallel::sort` 19×. We do not depict the sort duration of `paradis::sort` in the plot because it takes 10 seconds to sort two billion keys on the IBM AC922. We measure `paradis::sort` to perform disproportionately worse on the POWER9 CPU of this system due to the low number of physical CPU cores (see Table 4). Given sufficient thread-level parallelism and sufficiently large input sizes, highly parallel CPU sorting algorithms are commonly main memory bandwidth-bound [34, 59]. The IBM AC022 has not enough physical CPU cores for `paradis::sort` to saturate the main memory bandwidth. It only does so for data sets with more than eight billion keys. With two GPUs and their limited memory capacity, we can sort up to six billion keys. Thus, on the IBM AC922, `gnu-parallel::sort` is the faster CPU sorting algorithm for all input sizes that the two NVIDIA Tesla V100 GPUs can sort. While the single-GPU radix sort `thrust::sort` already outperforms the CPU by a factor of 12×, RMG sort achieves a speedup of 1.6× over the single-GPU baseline.

In Figure 12, we depict the sort duration of RMG sort with GPU pair $(0, 1)$, and our two state-of-the-art CPU baselines for increasing numbers of keys. We measure that RMG sort scales linearly with the number of input keys n. It outperforms the

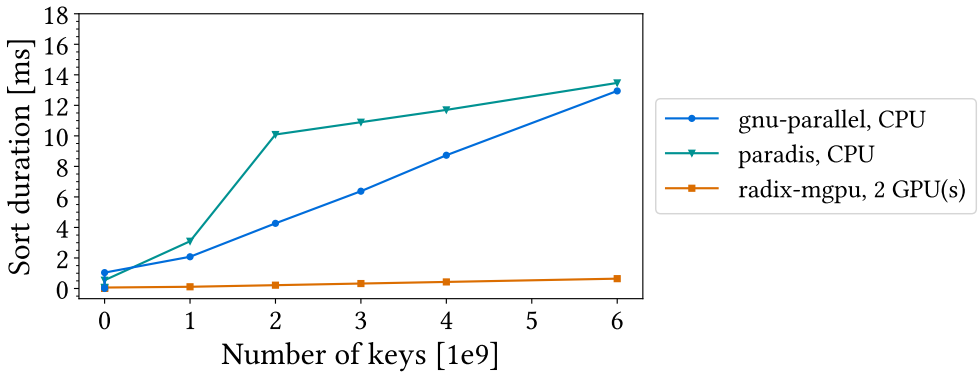


Figure 12: CPU-only vs. multi-GPU sorting performance for increasing numbers of keys on the IBM AC922

CPU 20× for six billion keys. Thus, we find that our multi-GPU sorting algorithm (RMG sort) considerably accelerates sorting on the IBM AC922.

In Figure 13, we depict the sorting performance overview for the NVIDIA DGX A100. We compare the sort duration of RMG sort with four and all eight GPUs against our two CPU baselines and the single-GPU radix sort baseline. The system's optimal four-GPU-set consists of GPUs $(0, 2, 4, 6)$ as each GPU of the set utilizes one PCIe switch exclusively, achieving peak CPU-GPU copy throughput. Again, we sort two billion uniformly distributed 32-bit unsigned integers.

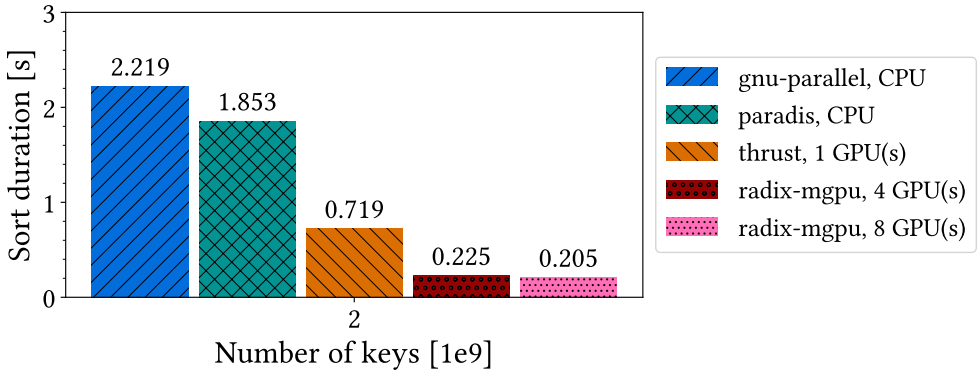


Figure 13: Sorting performance overview: CPU, single-GPU, and multi-GPU sorting algorithms on the NVIDIA DGX A100

We observe that both CPU sorting algorithms achieve shorter execution times on the DGX A100 compared to the IBM AC922. The DGX A100 is equipped with four times as many physical cores per CPU, twice as many threads, and a

memory subsystem of higher bandwidth. Moreover, `paradis::sort` outperforms `gnu-parallel::sort` on the DGX A100. When sorting large data set sizes with enough physical CPU cores to saturate the main memory bandwidth, the beneficial constant computational complexity of `paradis::sort`, a radix sort algorithm, becomes visible in comparison to `gnu-parallel::sort`, a merge sort algorithm. This explains why `paradis::sort` is the faster CPU sorting algorithm on this system.

With regards to GPU-accelerated sorting, we see that one GPU achieves a speedup of 3× over `gnu-parallel::sort` and 2.6× over `paradis::sort`. We observe that the total sort duration of our single-GPU radix sort is approximately twice as high on the DGX A100 as on the IBM AC922. The main reason for this significant difference is that the IBM AC922 connects the GPUs to the CPU via high-bandwidth NVLink 2.0 interconnects while the DGX A100 uses PCIe 4.0 as the CPU-GPU interconnects (see Figure 9 and Figure 10). Thus, the CPU-GPU data transfers take significantly longer on the DGX A100.

When sorting with multiple GPUs using RMG sort, we achieve speedups over `gnu-parallel::sort` of 9.8× for four GPUs and 10.8× for eight GPUs. Moreover, RMG sort outperforms the state-of-the-art CPU-based radix sort `paradis::sort` 8× with four GPUs, and 9× with eight GPUs. Thus, we find that all $g_{max} = 8$ system-wide GPUs reach the best sorting performance on the DGX A100.

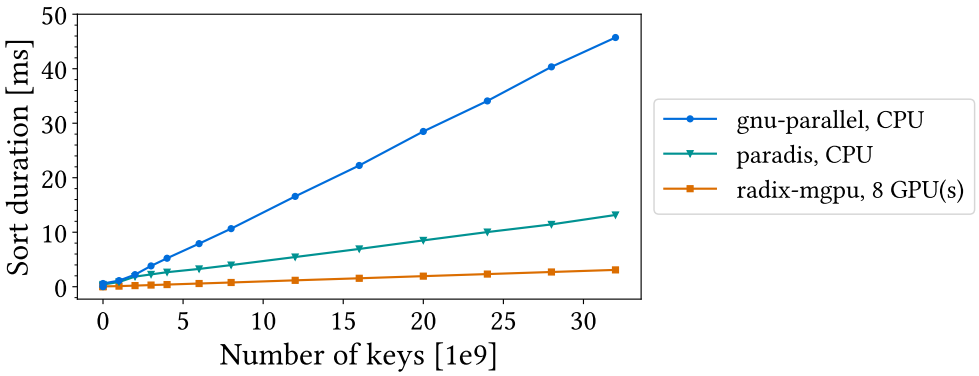


Figure 14: CPU-only vs. multi-GPU sorting performance for increasing numbers of keys on the NVIDIA DGX A100

In Figure 14, we depict the sorting performance of our two CPU baselines and RMG sort for sorting increasing numbers of 32-bit keys on the DGX A100. We again observe that RMG sort scales linearly. We also see that, with eight GPUs and a total combined GPU memory of 320 GB, we can sort significantly larger data sets. We measure RMG sort to sort the largest data set of 32 billion keys in

3.1s, outperforming `paradis::sort` 4.3×. Including the data transfers, RMG sort achieves an end-to-end sorting rate of over 10 billion keys per second.

With eight GPUs on the DGX A100, we can sort significantly larger data sets. Due to `paradis::sort`'s constant computational complexity, it sorts 32 billion keys almost 3.5× faster than `gnu-parallel::sort`. Similar to the IBM AC922, we measure that `paradis::sort` needs more than two billion keys to fully maximize its achieved memory throughput, and saturate the system's main memory bandwidth. This is why the speedup factor of RMG sort over `paradis::sort` is lower for 32 billion than for two billion keys. Still, the speedup of RMG sort over `paradis::sort` is significant.

5.3 Radix-Partitioning vs. Sort-Merge

For many previous single-GPU and multi-GPU sorting algorithms, researchers reported significant speedup factors compared to CPU-only algorithms. In the next section, we compare our radix-partitioning-based multi-GPU sorting algorithm to two state-of-the-art merge-based multi-GPU sorting algorithms from the literature. By comparison, we evaluate which algorithm achieves the best performance.

5.3.1 Sort-Merge-based Multi-GPU Sorting Algorithms

First, we explain the two state-of-the-art merge-based multi-GPU sorting algorithms that we use for comparison.

The P2P-based multi-GPU merge sort (**P2P merge sort**) by Tanasic et al. sorts chunks of data with multiple GPUs and merges the sorted chunks on the GPUs utilizing inter-GPU communication [64]. It benefits from high-bandwidth P2P interconnects, such as NVLink 2.0 and NVLink 3.0. By selecting a pivot element within the sorted chunks of a GPU pair, blocks of keys are swapped so that afterwards, the first GPU contains keys that are all less 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 the globally sorted output order. The algorithm sorts on more than two GPUs using multiple subsequent P2P key swaps and GPU-local merge steps. The number of P2P transfers scales linearly with the number of GPUs g. Due to the recursive nature of the merge phase, P2P merge sort can only sort on g GPUs when $g = 2^k$ for $k \in \mathbb{N}$. RMG sort can utilize any number of GPUs g.

The heterogeneous multi-GPU merge sort (**HET merge sort**) by Gowanlock et al. sorts chunks of data on multiple GPUs and merges the sorted chunks on the CPU in main memory [18]. After the sorted chunks are copied back to main memory,

a parallel multiway merge algorithm is used to produce the sorted output. Out of the three multi-GPU sorting algorithms, it is the only one that is not limited by the combined GPU memory capacity. Since our proposed algorithm only sorts data sets that fit into the combined GPU memory, we disregard the evaluation of large out-of-core data sets.

5.3.2 Multi-GPU Sorting Algorithm Comparison

Both merge-based multi-GPU sorting algorithms have recently been evaluated on the IBM AC922 and the DGX A100 by Maltenberger et al. [34]. We perform benchmark experiments for our proposed RMG sort and compare its performance with the evaluation results reported by Maltenberger et al. Since we use the same experimental methodology, assumptions, and hardware systems as Maltenberger at al. (see Section 5.1), our algorithm performance comparison is as representative and as precise as an evaluation with measured results for all three algorithms.

In Figure 15, we depict the sort duration of RMG sort, P2P merge sort, and HET merge sort for the CPU-local GPU pair $(0, 1)$ and increasing numbers of 32-bit integer keys with uniform distribution on the IBM AC922. We observe that RMG sort scales best to the input size n, sorting faster than the two merge-based multi-GPU sorting algorithms. RMG sort outperforms P2P merge sort by 17%, and HET merge sort 1.7× for four billion keys.

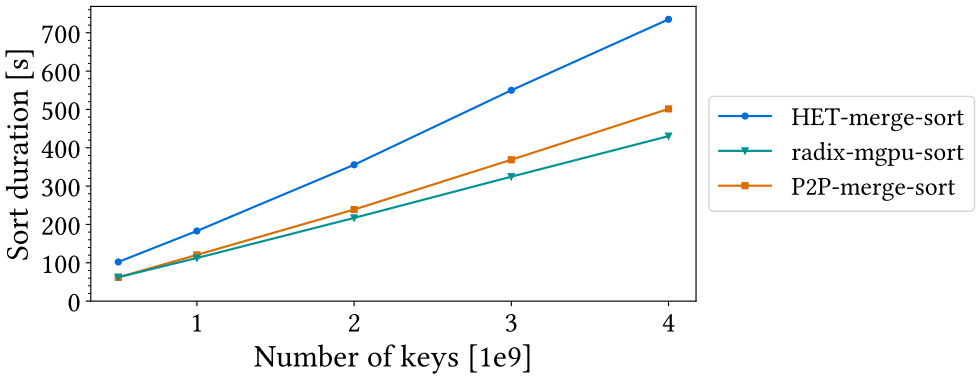


Figure 15: Algorithm comparison: 2 GPUs sorting increasing numbers of keys on the IBM AC922

In Figure 16, we show the sorting durations for the three multi-GPU sorting algorithms for increasing input sizes on eight GPUs on the DGX A100. We again find that RMG sort achieves the best sorting performance. Compared to P2P merge sort, RMG sort is faster up to 1.26× while outperforming HET merge sort

up to 1.8×. Moreover, we observe that RMG sort is the faster algorithm option for any number of input keys n, as is the case on the IBM AC922. In the following plots and experiments, we analyze why RMG sort outperforms the other two merge-based algorithms.

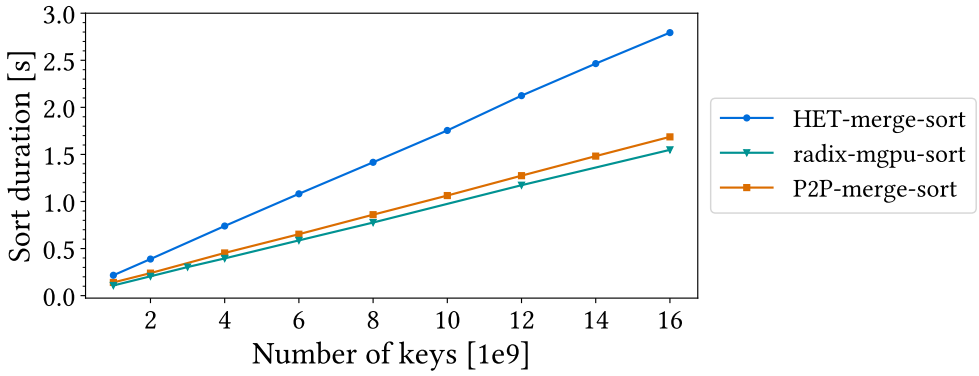


Figure 16: Algorithm comparison: 8 GPUs sorting increasing numbers of keys on the NVIDIA DGX A100

To explain the performance differences between RMG sort and the merge-based algorithms, we break down the total sort duration of each algorithm into its individual algorithm phases.

All three algorithms start with the initial host-to-device (HtoD) copy where chunks of the input data are distributed to the involved g GPUs. For RGM sort, the remaining phases are the radix partitioning phase, the P2P key swap, and the bucket sorting phase which is interleaved with copying the sorted buckets back to the host. For P2P merge sort, we break down the sort duration into the HtoD copy, the sort phase, the P2P merge phase on the GPUs, and the device-to-host copy (DtoH). In its sort phase, the entire chunk of each GPU is sorted using Thrust's single-GPU radix sort primitive. HET merge sort entails similar phases except for the merge phase. Instead of the GPU-based P2P merge phase, it uses a CPU-based multiway merge.

Sort Duration Breakdown on the IBM AC922. In Figure 17, we depict the sort duration breakdown of RMG sort for an input size of two billion uniformly distributed keys on the IBM AC922. We show the sorting time breakdown for the single-GPU baseline, the GPU pair $(0, 1)$, and all four GPUs of the system.

We observe that the radix partitioning phase achieves the shortest time duration out of all algorithm phases. On two GPUs it makes up only 11% of the total sort duration with 22.8ms, while it takes four GPUs half that time (11.4ms). Because

the total sort duration on four GPUs is disproportionately higher, this constitutes only 3% of the total sort duration. Nonetheless, we observe linear scaling of our radix partitioning phase to the number of keys (i.e. the GPU chunk size). Twice as many GPUs partition a fixed total input size twice as fast.

For the GPU pair $(0, 1)$, the second shortest time duration is the P2P key swap. Powered by NVLink 2.0 interconnects for a bandwidth rate of 75 GB/s, the P2P bucket transfers take 35.8ms, which is 16% of the total sort duration.

While the HtoD copy is halved compared to the single-GPU baseline, we observe that the "Sort Buckets & DtoH Copy" phase makes up 47% of the total sort duration. Maltenberger et al. find that the DtoH copy throughput decreases for parallel transfers from multiple GPUs to the same NUMA node from a peak throughput of 145 GB/s in the HtoD direction down to 110 GB/s [34].

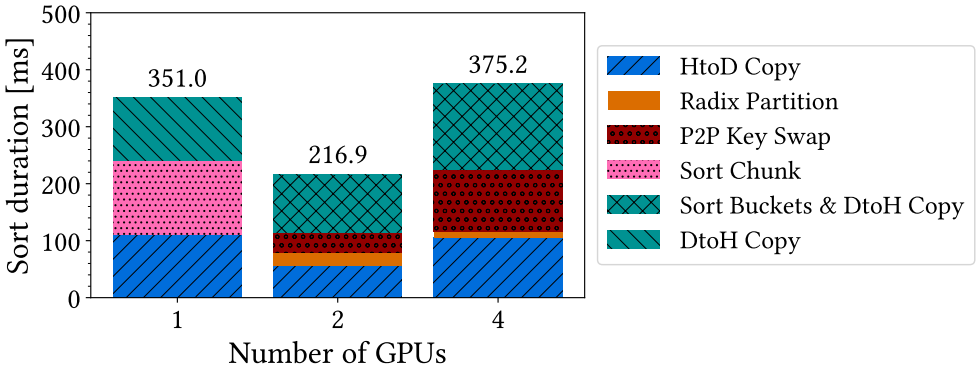


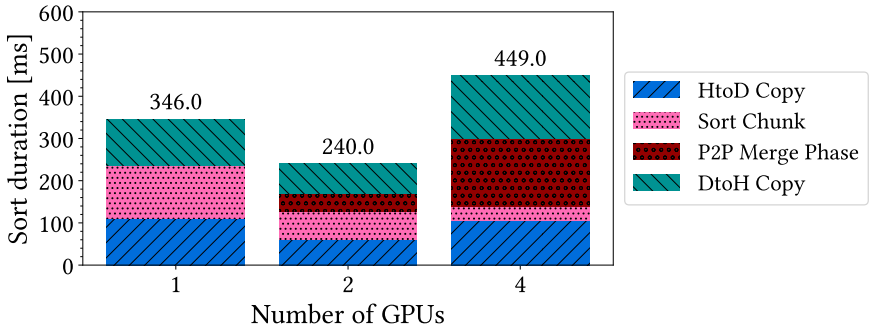
Figure 17: Sort duration breakdown for RMG sort: Sorting two billion 32-bit integer keys on the IBM AC922

In addition to this anomaly, we measure that the sort-copy-overlap comes with a slight performance overhead since the time duration of the "Sort Buckets & DtoH Copy" phase is not equal to the time duration of a simple parallel DtoH copy (i.e., what would be the case if the sorting computation was perfectly hidden). Instead, we measure it to take 43% longer. Still, overlapping the sorting computation with the DtoH copy improves the total sort duration, saving 50% of the bucket sorting time duration (32ms) when sorting with two GPUs on this system. This equals to 24% of the time it would take the GPUs to sort all buckets and copy them back without overlap.

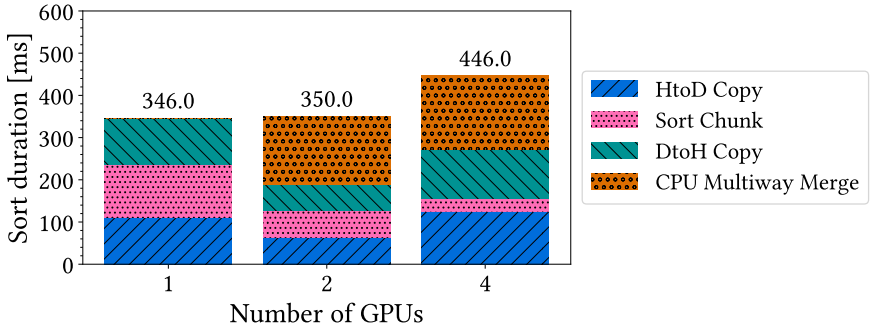
Figure 17 also shows why four GPUs perform worse than two on the IBM AC922. The CPU-interconnect X-Bus is the system's data transfer bottleneck. The X-Bus bandwidth has previously been evaluated to achieve only 41 GB/s of the theoretical

peak bandwidth of 64 GB/s [51, 34]. As a consequence, the slow CPU-GPU data transfers (HtoD and DtoH copies) to and from the remote GPUs 2 and 3 decrease the total sort duration for four GPUs. Also, the throughput of the P2P key swap suffers from the low X-Bus bandwidth, which is why the P2P key swap phase takes 3× longer when sorting with four GPUs compared to two.

In total, RMG sort achieves a speedup of 1.6× over the single-GPU baseline for sorting two billion integer keys with two GPUs on the IBM AC922. On four GPUs, RMG sort performs 7% slower compared to one GPU.



(a) P2P merge sort



(b) HET merge sort

Figure 18: Sort duration breakdown: Sorting two billion 32-bit integer keys on the IBM AC922

Figure 18 shows the sort duration breakdown of P2P merge sort and HET merge sort on the IBM AC922. We compare RMG sort's performance with the two merge-based algorithms.

In Figure 18a (P2P merge sort), we observe that the time duration for sorting each GPU chunk gets halved every time we double the number of GPUs. Since the number of P2P transfers performed in the GPU-based merge phase scales

linearly with the number of GPUs g, the merge phase overhead is minimal when sorting with two GPUs. In fact, on two GPUs, P2P merge sort requires only a single P2P key swap and is in that case similar to RMG sort. For two GPUs, we measure the P2P transfer times to be nearly identical between RMG sort and P2P merge sort.

We find that, compared to the combined time duration of the radix partitioning phase and the P2P key swap of RMG sort, the merge phase of P2P merge sort is 33% faster (=14.5ms) on two GPUs. Given that the P2P-based merge sort algorithm swaps one consecutive block of sorted keys between the GPUs, the algorithm's final step is to locally merge on each GPU. For this, P2P merge sort utilizes `thrust::merge` which, on the Tesla V100 GPU, takes less time (11ms) than our histogram computation and the key scattering step combined (22.8ms). However, since we overlap the sorting computation with the DtoH copy in our algorithm, RMG sort outperforms P2P merge sort on two GPUs by 11% for an input size of two billion 32-bit integer keys.

On four GPUs, we measure that the combined time duration of the radix partitioning phase and the P2P key swap of RMG sort executes 33% faster than the P2P merge phase. This is because the P2P merge phase takes longer the more GPUs are involved, scaling linearly with the number of GPUs g. This explains why the speedup of RMG sort over P2P merge sort increases from two to four GPUs (20%).

When comparing RMG sort with HET merge sort (see Figure 18b), we observe that the CPU-based multiway merge phase is significantly slower compared to our GPU-based partitioning approach. On two GPUs, it takes RMG sort to partition and swap the keys via P2P transfers 2.7× less time than it takes the CPU to merge the two sorted GPU chunks. On four GPUs, the P2P key swap throughput of RMG sort is significantly reduced. Still, the combined time duration for the radix partitioning phase and the P2P key swap is 44% lower than the CPU-based merge phase. In total, RMG sort outperforms HET merge sort 1.6× on two GPUs, and 1.2× on four GPUs when sorting two billion keys on the IBM AC922.

We conclude that on GPUs with high-bandwidth P2P interconnects, GPU-only sorting approaches are superior to the CPU-based multiway merge. Both, RMG sort and P2P merge sort heavily rely on high-bandwidth P2P interconnects. Because the two-GPU-set $(0, 1)$ is the system's optimal GPU set, RMG sort does not benefit from its reduced inter-GPU communication which only applies for more than two GPUs. Still, RMG sort outperforms P2P merge sort because we overlap the sorting computation of buckets with the DtoH copy – an algorithmic optimiza-

tion that results from our MSB radix partitioning approach. The P2P merge sort algorithm is required to wait for each GPU's last key to be merged until the DtoH copy can start.

Sort Duration Breakdown on the DGX A100. Figure 19 shows the sort duration breakdown of RMG sort for sorting two billion uniformly distributed keys on the NVIDIA DGX A100. We evaluate the optimal GPU sets $(0, 2)$, $(0, 2, 4, 6)$, and $(0, 1, 2, 3, 4, 5, 6, 7)$.

First, we note that the HtoD and DtoH copy phases take significantly longer on this system due to the low bandwidth of PCIe 4.0, compared to the three links of NVLink 2.0. Furthermore, we observe that our radix partitioning phase scales very well to increasing numbers of GPUs. The time duration of the radix partitioning phase equals 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.

Figure 19 also shows that the P2P key swap time duration stays constant, independent of the number of GPUs g, because our radix partitioning phase is designed to enable a single all-to-all bucket exchange step. Also, the P2P key swap benefits from the NVLink 3.0-based NVSwitch P2P interconnects that allow the DGX A100 to achieve fast transfers between all eight GPUs. We measure the P2P key swap phase to take from 14-17ms for sorting with two, four, and eight GPUs, not exceeding 8% of the total sort duration. Similar to our radix partitioning phase, the time duration of the sorting computation gets halved whenever the number of GPUs is doubled. However, the time it takes an NVIDIA A100 GPU to sort the buckets of its chunk is rather insignificant compared to the HtoD and DtoH copy phases combined. On the DGX A100, we furthermore observe that the sort-copy-overlap introduces close to no overhead as the sorting computation is almost completely hidden.

On this system, the CPU-GPU transfers are the only performance bottleneck. Still, we observe RMG sort to scale comparatively well from one to eight GPUs. Compared to the single-GPU baseline, RMG sort achieves speedups of 1.9× with two GPUs, 3.2× with four GPUs, and 3.5× with eight GPUs. We can not expect the speedup factor to become significantly higher on eight GPUs, because of the shared bandwidth effects that result from the system's limited number of PCIe switches. When performing parallel CPU-GPU transfers, each pair of neighbouring GPUs shares the respective bandwidth of the PCIe 4.0 interconnects (see Table 4). We cannot avoid this hardware limitation when using all GPUs of the system.

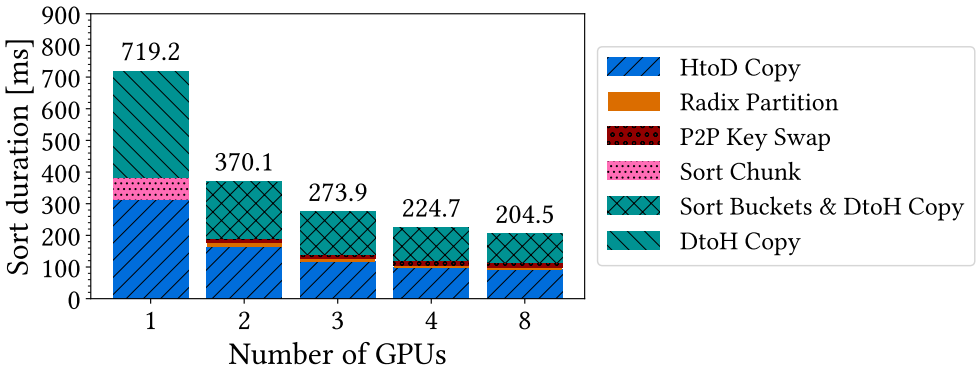


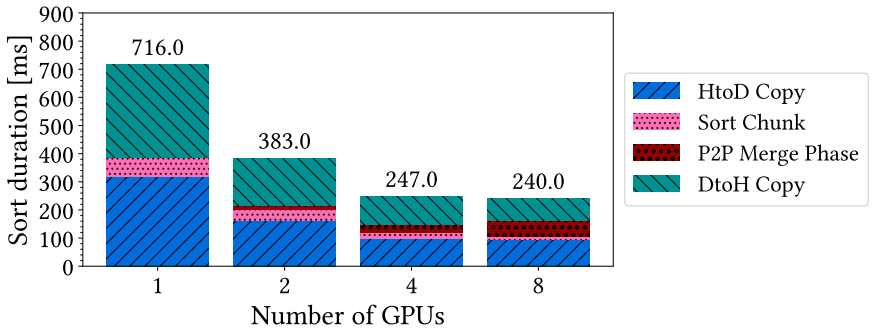
Figure 19: Sort duration breakdown for RMG sort: Sorting two billion 32-bit integer keys on the DGX A100

The reason why RMG sort is about 20ms faster with eight than with four GPUs despite the shared PCIe switches is that 1) the radix partitioning time duration is halved and, 2) the HtoD and DtoH copy throughput does increase slightly compared to the CPU-GPU transfers on four GPUs (see [34]).

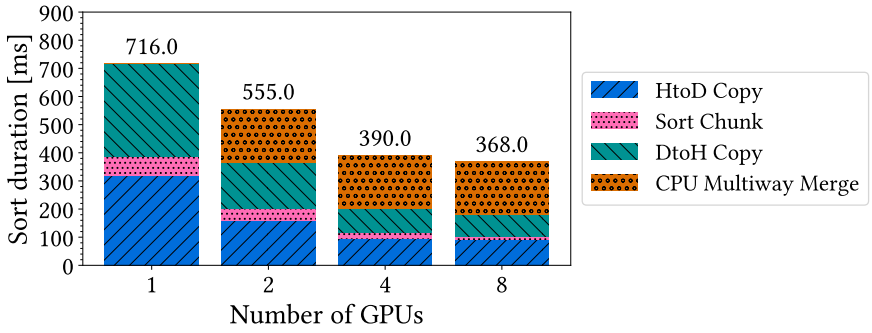
In Figure 20, we show the sort duration breakdown for the two merge-based sorting algorithms for the NVIDIA DGX A100. For consistency, we again sort a total of two billion uniformly distributed 32-bit integer keys.

In contrast to the time duration of RMG sort's P2P key swap phase, the P2P merge phase time of P2P merge sort (depicted in Figure 20a) increases with the number of GPUs. We measure it to take almost 4× longer when eight GPUs merge their chunks compared to two GPUs. This explains why RMG sort's speedup factor over P2P merge sort increases with the number of GPUs: $g = 2$ GPUs: 3%, $g = 4$ GPUs: 10%, $g = 8$ GPUs: 17%. When sorting with RMG sort on eight GPUs, the radix partitioning phase and the P2P key swap combine for a time duration of 20ms, which is 2.7× less than the P2P merge phase takes. The comparatively slow HtoD and DtoH copy phases reduce the relative impact of those algorithm phases on the total sort duration, limiting RMG sort's speedup factors over P2P merge sort. Still, we demonstrate the potential speedup that RMG sort could achieve if the system included high-bandwidth CPU-GPU interconnects.

We depict the sort duration breakdown of HET merge sort for the DGX A100 in Figure 20b. Similar to the IBM AC922, we observe that the limiting factor of HET merge sort is the CPU's merging performance. Compared to the combined time duration of RMG sort's radix partitioning phase and its P2P key swap, the CPU multiway merge phase takes 6.6× longer on two GPUs, 7.9× longer on four



(a) P2P merge sort



(b) HET merge sort

Figure 20: Sort duration breakdown: Sorting two billion 32-bit integer keys on the DGX A100

GPUs, and 9.2× longer on eight GPUs. Consequently, RMG sort outperforms HET merge sort significantly; up to 1.8× for eight GPUs.

Having evaluated RMG sort and two state-of-the-art merge-based multi-GPU sorting algorithms on the DGX A100, we confirm that, on modern accelerator platforms, P2P-based multi-GPU approaches sort significantly faster than the heterogeneous CPU-based merging approach.

We conclude that, compared to P2P merge sort, RMG sort more efficiently utilizes the NVLink-based NVSwitch interconnects of the DGX A100. RMG sort scales linearly with the number of GPUs g in the radix partitioning phase and keeps a constant P2P key swap time duration, independent of how many GPUs we sort with. RMG sort uses the non-blocking all-to-all P2P transfer capability of NVSwitch systems to the fullest extent.

5.4 RMG Sort Performance Analysis

In this section, we evaluate the performance of RMG sort for varying data and distribution types, especially for skewed data distributions, to analyze the performance characteristics and bottlenecks of RMG sort.

5.4.1 Sorting Different Data Distributions

In Figure 21, we show the sorting duration of RMG sort for different data distributions of the input keys. We sort 32-bit unsigned integer keys using the optimal GPU set $(0, 1)$ on the IBM AC922. We observe significant performance differences for different distribution types. We use the sort duration for uniformly distributed keys as this experiment's baseline since we performed all previous experiments of our evaluation using the uniform distribution (see Figures 11, and 17).

In the best case, all keys of the input data set are of the same value which we refer to as the `zero` entropy distribution. We measure the sort duration of the zero distribution to be reduced by 30% (165.9ms) compared to the uniform distribution (217ms) because of multiple reasons. First, no P2P transfers are necessary because all keys belong to the same last pass spanning bucket. Since we distribute each last pass spanning bucket in such a way that we achieve perfect load balancing, the distribution of keys exactly corresponds to the initial data chunks of each GPU.

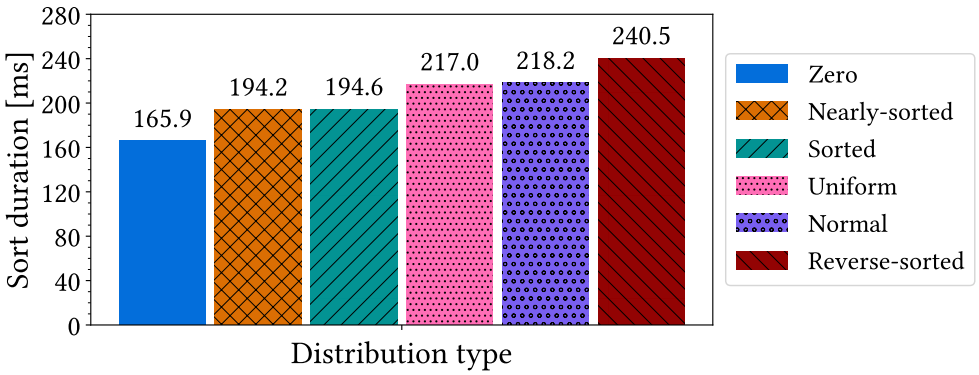


Figure 21: RMG sort's performance for different data distributions (2 billion keys, 2 GPUs, IBM AC922)

Moreover, we do not scatter any keys during the entire radix partitioning phase. Given that all n keys form one big spanning bucket that spans all g GPUs, we need all $p = 4$ partitioning passes. In each partitioning pass, we only compute the histogram and skip the `ScatterKeys` kernel launch because, for each GPU, all of its keys belong to the same bucket. The histogram computation takes only

6ms on the Tesla V100 GPU, while the key scattering step takes 16ms. Thus, skipping the `ScatterKeys` kernel for skewed distributions like the zero distribution improves RMG sort's performance significantly. Furthermore, RMG sort does not sort any bucket since we have already taken all keys' $k = 32$ bits into account during the radix partitioning phase. Thus, the DtoH copy is performed without the performance overhead that comes with overlapping the sorting computation with the data transfer.

For nearly-sorted distributions, we generate a uniformly distributed data set, sort its keys, and add a small, normally distributed (Gaussian) noise $e \sim N(0, \sigma^2)$ to each value to introduce keys that break the sort order. For both nearly-sorted and fully sorted input data, we observe that no P2P key swaps are necessary. Since we generate both distributions from uniform data sets, one partitioning pass is sufficient, before each GPU sorts its buckets and copies them back to the host. Thus, the performance gain corresponds to the time duration of the skipped P2P key swap, resulting in a performance improvement of 12%.

We observe that RMG sort sorts normally distributed data equally as fast as uniform distributions with two GPUs on the IBM AC922. The sort duration difference is less than 1%.

For reverse-sorted data, the P2P key swap time duration increases because we implement our bucket-to-destination-GPU mapping in such a way that the small buckets are transferred to GPU 0. In the case of reverse-sorted data, the input keys are copied (HtoD) to the GPUs in the mirrored way, i.e. the biggest key values reside on GPU 0. Thus, the two GPUs swap all of their buckets between each other in the P2P key swap, and the total sort duration increases by 11% up to 240.5ms.

We propose to optimize RMG sort for these kinds of distributions as part of future work. A check could determine whether our computed bucket-to-destination-GPU mapping would swap all buckets of each GPU i with the buckets of its *mirrored* GPU $j = g - i$. If so, we can completely skip the P2P key swap as we do for sorted and nearly-sorted distributions. We simply adjust the DtoH copies from the GPUs to main memory to correctly return the chunks in the globally sorted order.

In Figure 22, we show the sorting duration of RMG sort for different data distributions on the DGX A100 with eight GPUs. We find that the performance differences between input keys of varying distribution types are less significant than on the IBM AC922. This is because the CPU-GPU data transfers, which are independent of the data distribution, make up the majority of the total sort dura-

tion on the DGX A100, as evaluated in our sort duration breakdown experiments in Section 5.3.

We find that beneficial distributions reduce the overall sorting time by 13% for zero distributions, 8% for sorted data, and 7% for nearly-sorted distributions.

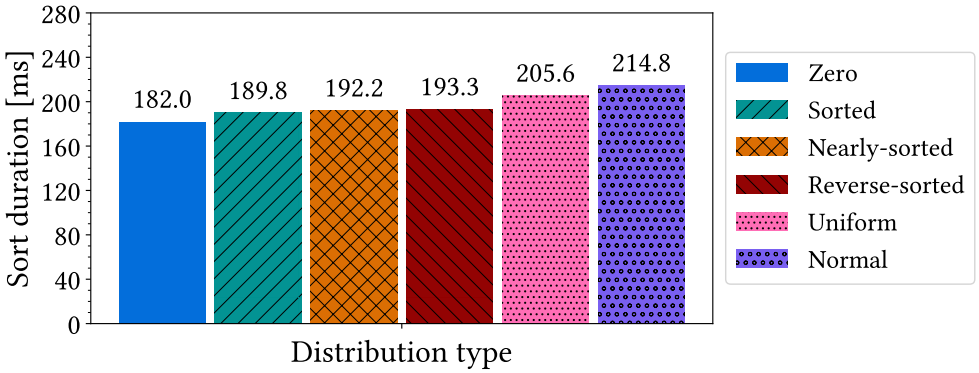


Figure 22: RMG sort's performance for different data distributions (2 billion keys, 8 GPUs, DGX A100)

Interestingly, reverse-sorted key distributions benefit the sort duration of RMG sort on the DGX A100. The total amount of data transferred between the GPUs is the same on the IBM AC922 and the DGX A100 given that we sort two billion keys on both systems. However, the reduced chunk size when sorting with eight GPUs decreases the P2P key swap duration because the system's P2P interconnects support multiple, concurrent, bidirectional transfers.

Additionally, we observe that the mirrored bucket distribution across eight GPUs is well suited for the NVLink-based NVSwitch interconnect topology of the DGX A100. For uniform distributions, every GPU contains buckets that need to be transferred to every other GPU. Given reverse-sorted data, the P2P key swaps are performed only between pairs of mirrored GPUs. We measure that this copy pattern allows each bucket swap to achieve a significantly higher P2P copy throughput ($1.3-4.7\times$). With the two effects combined, we sort reverse-sorted data 6% faster than uniformly distributed data with eight GPUs on this system.

We further find that eight GPUs sort normal distributions significantly slower than uniform distributions. Also, normally distributed input data requires two partitioning passes when sorting with eight GPUs. This results in a slightly increased P2P key swap time and explains the sort duration increase of 4%.

Bit Entropy. We evaluate RMG sort's performance for a skewed, parameterized data distribution, whose skew is determined by the bit entropy. We define the bit entropy to specify how many least significant bits of the entire bit range of the 32-bit keys we fill randomly. For a bit entropy value of 32, all $k = 32$ bits of the keys follow a uniform distribution, i.e. the data set is uniformly distributed. For a bit entropy value b of $0 < b < k$, the most significant $k - b$ bits are zero for all n keys. If the bit entropy value is zero, the data distribution is the same as the zero distribution from Figure 21 where all keys are of the same value.

In Figure 23, we show the sort duration of RMG sort, depending on the bit entropy, when sorting with the two GPUs $(0, 1)$ on the IBM AC922.

We observe that the sort duration peaks at those bit entropy values that correspond to a low total number of buckets after the partitioning phase. Since our MSB radix partitioning phase considers $c = 8$ bits per partitioning pass, this case occurs at regular intervals, i.e. for bit entropy values $b \in L = \{26, 25, 18, 17, 10, 9\}$. If the bit entropy value is equal to 26, the first partitioning pass considers the most significant bits [32..24), and finds that all keys of each GPU chunk fall into $2^{8-(32-26)} = 4$ possible buckets: `[0]` $\sim$ `00000000`, `[1]` $\sim$ `00000001`, `[2]` $\sim$ `00000010`, `[3]` $\sim$ `00000011`.

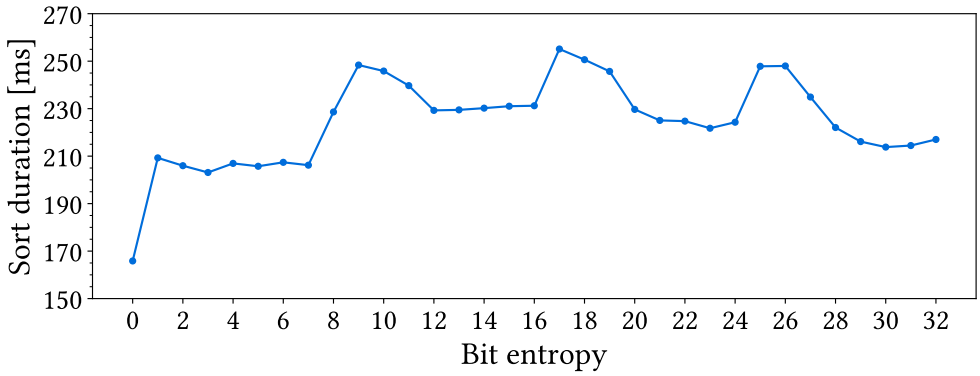


Figure 23: RMG sort's performance for a varying bit entropy (2 billion keys, 2GPUs, IBM AC922)

Similarly, a bit entropy of 25 results in only two possible buckets. For a bit entropy of 10, the first two partitioning passes on the bit ranges [32..24) and [24..16) determine spanning buckets across all g GPUs, until the third and completing partitioning pass on the bits [16..8) generates four buckets. In all cases where $b \in L$, the number of buckets is less than or equal to MIN_{SCO}, which is too low to efficiently overlap the sorting computation with the DtoH copy (see Table 2

and Section 4.4). This explains the performance decrease for these skewed data distributions.

We also observe that the sort duration gradually increases up to each peak (x-axis read from right to left) for the bit entropy values $b \in K = \{28, 27, 20, 19, 11\}$. This is because even though the number of buckets is greater than MIN_{SCO}, and we turn on the sort-copy-overlap optimization, the achieved DtoH copy throughput decreases compared to higher numbers of buckets (e.g. for uniform distributions).

Generally, we observe that with more partitioning passes performed during the radix partitioning phase, the total sort duration increases. However, since we skip the key scattering step for those partitioning passes that find one non-empty bucket only (see Section 4.2), we keep the resulting performance overhead to a minimum. For example, when the bit entropy is equal to 10, the first two partitioning passes execute the histogram computation for a combined runtime of 12ms.

On the other hand, with more partitioning passes, the sorting computation is accelerated because we reduce the bit range by $c = 8$ for each pass (see Section 4.4). For the same example as above, where the bit entropy is 10 and three partitioning passes are required, the bucket sorting phase is almost 2× as fast as it takes the GPU to sort on all 32 bits. The two optimizations combined (i.e skipping the key scattering step and reducing the bit range) mitigate the sorting performance decrease for increasing numbers of partitioning passes. As a result, the observed sort duration increases are most significant for low numbers of buckets generated by highly skewed data, i.e. for bit entropy values $b \in L \cup K$, which do not allow the sort-copy-overlap to be efficient.

We measure the total sort duration to increase up to 18% across all sub-optimal bit entropy values. Compared to P2P merge sort by Tanasic et al. [64], RMG sort is up to 6% slower for worst-case bit entropy values. Still, RMG sort outperforms HET sort 1.4× for worst-case bit entropy values. We evaluate both merge-based multi-GPU sorting algorithms to be stable for varying bit entropy values, with no significant performance difference.

As part of future work, we propose to mitigate the negative performance impact of these skewed distributions by dynamically adjusting the bit range on which we partition the keys in our radix partitioning phase. When the completing partitioning pass finds that there are $\leq \text{MIN}_{SCO}$ buckets, performing another partitioning pass on a small number of subsequent bits might show promising results, given that it would increase the total number of buckets, and thereby the throughput of the interleaved sorting and copy operations.

For bit entropy values below 8, we do not sort any buckets given that the partitioning phase took all 32 bits into account already. For bit entropy values $b \in [8..1]$, the sort duration reduces by up to 9%. As explained for Figure 21 for the zero distribution, i.e. a bit entropy value of $b = 0$, we additionally do not need any P2P key swaps, which explains the steep sort duration decrease by 30% compared to the uniform distribution.

In Figure 24, we show the results of our bit entropy experiment on the DGX A100 when sorting with eight GPUs.

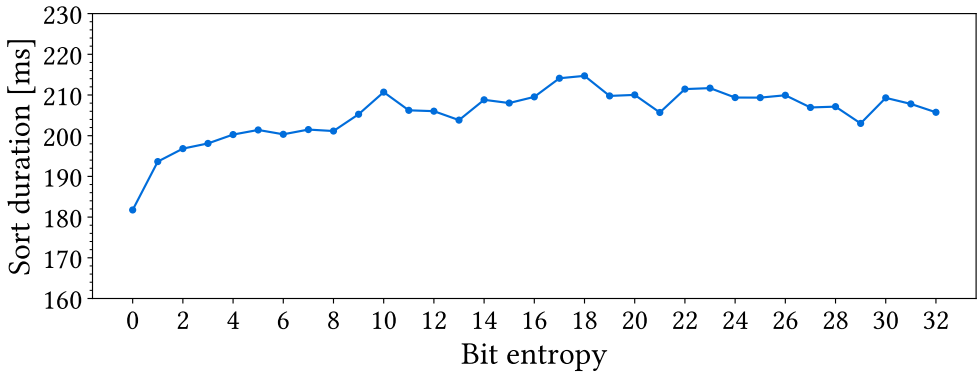


Figure 24: RMG sort's performance for a varying bit entropy (2 billion keys, 8 GPUs, DGX A100)

We note that the performance difference between different bit entropy values is much less significant. Compared to sorting uniformly distributed data, the sort duration of RMG sort varies between +5% and −2% for bit entropy values $b \in [32..9)$. For bit entropy values $b \in [8..1]$, the sort duration of RMG sort is reduced by up to 6% because we do not perform the bucket sorting step.

The main reason why the performance impact of the bit entropy is less significant on eight GPUs on the DGX A100, compared to two GPUs on the IBM AC922, is again the low CPU-GPU interconnect bandwidth. The comparatively slow HtoD and DtoH copies via PCIe 4.0 make up the majority of the total sort duration. Moreover, the time duration of the bucket sorting computation is significantly smaller on the DGX A100 because 1) each GPU sorts fewer keys in total as the input data is divided onto eight GPUs, and 2) the NVIDIA A100 sorts almost twice as fast as its predecessor, the NVIDIA Tesla V100. Thus, not being able to overlap the sorting computation with the DtoH copy has less of a negative performance impact on this hardware system.

Consequently, RMG sort outperforms P2P merge sort with eight GPUs on the

DGX A100, even for skewed bit entropy distributions. In the worst case, RMG sort still achieves a speedup of 12.5% over P2P merge sort, while being 1.7× faster than HET merge sort.

Zipfian Distributions. To further evaluate the performance impact of data skew, we analyze the sort duration of RMG sort for zipfian distributions, i.e. distributions with varying zipf exponents. A zipf exponent of 0 generates the default uniform distribution. For increasing zipf exponents, the probability of a key belonging to one of only a few highly frequent keys increases. Given two billion keys and a zipf exponent of 1.0, the probability that a key's value is equal to one of the top-1000 most occurring key values is 34%. The same probability is at 97.5% for a zipf exponent of 1.5.

At the same time, the remaining less frequent keys are scattered across increasing numbers of increasingly small buckets. Thus, zipfian distributions help us to analyze how RMG sort performs for distributions where certain GPUs end up with very few big buckets, while others handle large numbers of small buckets.

In Figure 25, we depict the sort duration of RMG sort, depending on the zipf exponent, for the two-GPU set $(0, 1)$ on the IBM AC922. We sort two billion 32-bit integer keys.

We observe that the sort duration increases for zipf exponents greater than zero, measuring the peak sort duration for an exponent of 1.5. At the peak, the sort duration reaches 273ms, which is an increase of 26% over the sorting time of the uniform distribution. The sort duration steadily decreases for zipf exponents greater than 1.5, almost dropping down to the initial, uniform sort duration.

For a zipf exponent of 0.5, one partitioning pass is sufficient for the radix partitioning phase to complete. We measure average execution times for the key scattering and the histogram computation kernels. However, we find that the number of buckets on the second GPU is greater than MAX_{BRS} (see Table 2), despite our optimization to fuse small neighbouring buckets. For numbers of buckets greater than MAX_{BRS}, we sort the entire GPU chunk at once, instead of sorting individual buckets to avoid the overhead of too many kernel launches. As a result, we cannot overlap the sorting computation with the DtoH copy, which explains why the sort duration increases by about 30ms.

Interestingly, for zipf exponents e, with $0.5 < e \leq 1.5$, the many buckets that result from the less frequent key values of the zipfian distribution become so small that our approach of fusing neighbouring buckets achieves a significant reduction. For some cases, we reduce the total number of buckets from up to 575 down to 90.

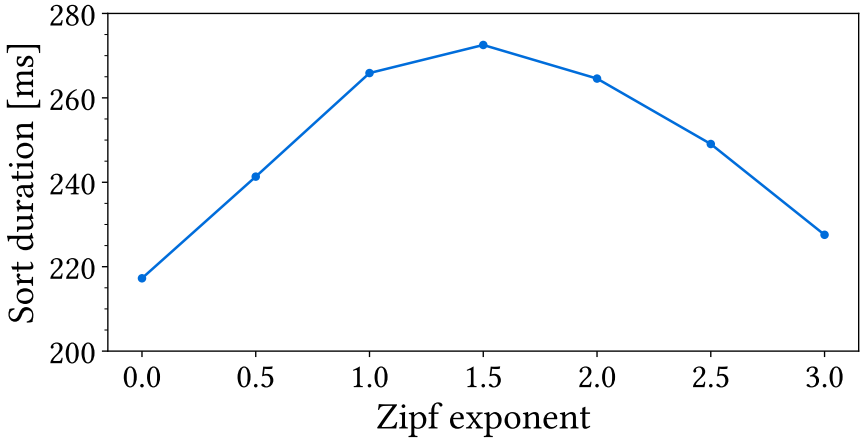


Figure 25: RMG sort's performance for a varying zipf exponent (2 billion keys, 2 GPUs, IBM AC922)

Thus, for zipf exponents $e \geq 0.75$, our optimization of fusing small buckets allows us to perform the sort-copy-overlap at peak DtoH copy throughput rates. With this, the bottleneck shifts to the radix partitioning phase.

For zipf exponents e, with $0.5 < e \leq 1.5$, the data distribution becomes more skewed, and more spanning buckets need to be resolved. For a zipf exponent of 1.0, three partitioning passes are necessary, while the exponent 1.5 requires all $p = 4$ partitioning passes to be performed. For zipfian distributions, and in contrast to the bit entropy experiments, each partitioning pass finds more than one non-empty bucket. Consequently, we have to execute the `ScatterKeys` kernel in each partitioning pass. As a result, we have little to no sorting computation left to perform in those cases, given that the radix partitioning phase considered (almost) all bits. However, the time duration of multiple `ScatterKeys` kernel executions adds up to a significant performance overhead on each of the two GPUs of this system, especially because the shard memory atomic operations load of the `ScatterKeys` kernel increases for skewed data. 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 zipf exponents of 1.0-1.5.

For zipf exponents beyond 1.5, the sort duration decreases. With such high exponents, almost all keys belong to the same few buckets, and the zipfian distribution approaches the zero bit entropy distribution. For these zipf exponents, we can increasingly skip the `ScatterKeys` kernel executions during the partitioning passes as the number of total buckets decreases.

We evaluate the sort duration of P2P merge sort and HET merge sort to be independent of the zipf exponent. We evaluate RMG sort to be up to 13% slower than P2P merge sort on the IBM AC922. Compared to HET merge sort, we still achieve a speedup of 1.3× for the wort case zipf exponent of 1.5.

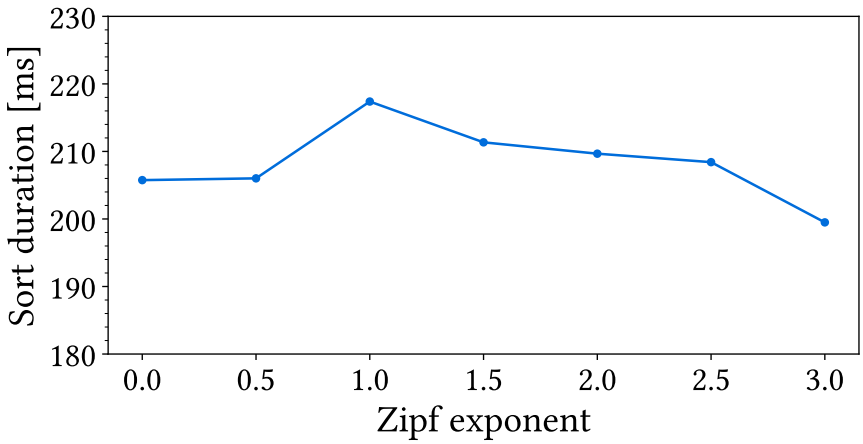


Figure 26: RMG sort's performance for a varying zipf exponent (2 billion keys, 8 GPUs, DGX A100)

In Figure 26, we depict the sort duration of RMG sort for varying zipf exponents on the DGX A100, when sorting with all eight GPUs of the system.

Similar to the bit entropy experiments, we measure significantly fewer differences in the sort duration of varying zipf exponents on the DGX A100. We observe the peak sort duration to be at 217ms for zipf exponents of 1.0. This constitutes a 6% increase compared to the sort duration of uniformly distributed keys. The reason for the performance drop is again the increased time duration of the radix partitioning phase as a result of multiple partitioning passes.

Overall, 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, compared to sorting on two GPUs on the IBM AC922. Furthermore, the total number of buckets that each GPU sorts after the P2P key swap is less likely to exceed MAX_{BRS}. Because we distribute the buckets across eight GPUs, each GPU handles fewer buckets. Similarly, the accumulated time duration of all histogram computations and key scattering kernel executions during the radix partitioning phase is less than it is on the AC922 because 1) the NVIDIA A100 GPU has a higher global memory bandwidth and improved atomic operations performance, and 2) each individual GPU processes fewer keys.

We conclude that, for highly skewed zipfian data distributions, RMG sort still outperforms P2P merge sort by at least 11%, and HET merge sort at least 1.7×, using eight GPUs on the DGX A100.

5.4.2 Sorting Varying Data Types

We evaluate RMG sort for different data types. We analyze its sorting performance for unsigned integer (`uint`) and floating-point (`float`) keys in their 32-bit and 64-bit variant. As outlined in Section 1.2, we only sort positive key values. Even though there is no dedicated unsigned floating-point data type, we only generate positive values for any data distribution.

We explain how the binary representation of floating-point numbers affects radix sort algorithms in Section 2.3. We sort uniformly distributed integer keys and floating-point keys whose values follow a zipfian distribution with an exponent of 1.0 on the entire floating-point value range. In that way, the k bits of the floating-point keys are distributed similarly to the uniform integer distribution.

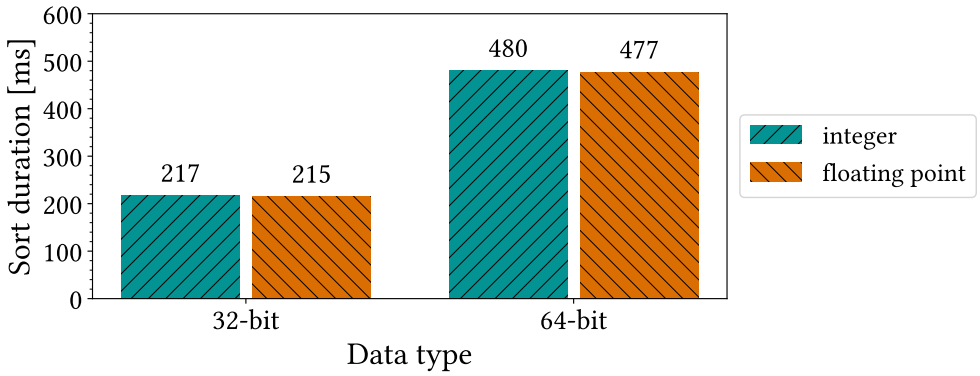


Figure 27: RMG sort's performance for varying data types (2 billion keys, 2 GPUs, IBM AC922)

In Figure 27, we show the sort duration of RMG sort for different data types when sorting 2 billion keys with two GPUs on the IBM AC922. As expected, we observe that the sort duration of 32-bit types is approximately the same, given that RMG sort's total sort duration depends on the number of bits per key k.

However, on the IBM AC922, the sort duration of 64-bit data types is 2.2× higher than for 32-bit types. Similar to the finding of Maltenberger et al. [34], we measure that our single-GPU radix sorting primitive `cub::DeviceRadixSort` performs disproportionately better for 32-bit than for 64-bit keys on the NVIDIA Tesla V100 GPU.

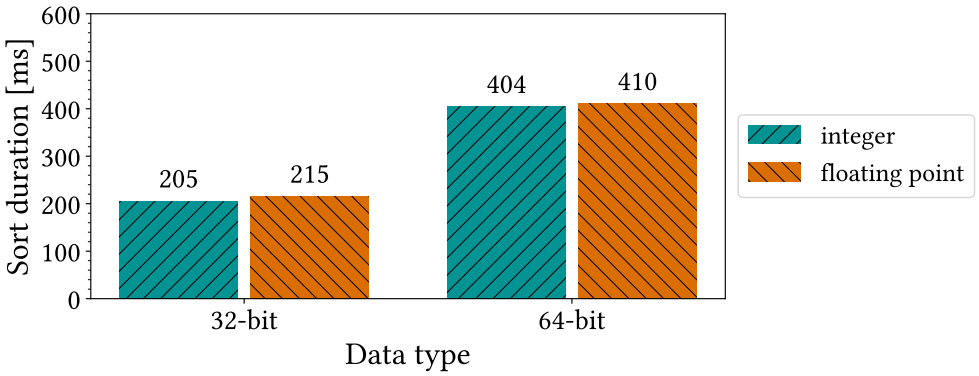


Figure 28: RMG sort's performance for varying data types (2 billion keys, 8 GPUs, DGX A100)

In Figure 28, we show the results of the same experiment performed on the DGX A100, with eight GPUs. We do not observe the performance discrepancy of `cub::DeviceRadixSort` on the latest NVIDIA A100 GPU. Sorting 64-bit data types takes exactly 2× longer than sorting their 32-bit counterparts, which is expected for any radix sort algorithm.

6 Discussion

In this section, we discuss the main findings of our experimental evaluation. We show that RMG sort significantly outperforms state-of-the-art parallel CPU-only sorting algorithms; up to 20× with two GPUs on the IBM AC922, and up to 9× with eight GPUs on the DGX A100. RMG sort considerably accelerates sorting for data sets of up to 32 billion keys.

Best Scaling Multi-GPU Sorting Algorithm. We demonstrate that, compared to two state-of-the-art merge-based multi-GPU sorting algorithms, RMG sort scales best to increasing numbers of GPUs. The radix partitioning phase of RMG sort scales linearly with the number of keys n. The number of P2P transfers performed in the P2P key swap phase is constant, as RMG sort requires only a single all-to-all P2P exchange independent of the number of GPUs g. In contrast, the time duration of the P2P-based merge phase of P2P merge sort grows linearly when scaling to increasing numbers of GPUs. When sorting with eight GPUs, we measure RMG sort's radix partitioning phase combined with the following P2P key swap to be 2.7× faster than the entire P2P-based merge phase of P2P merge sort. Moreover, we find that the CPU's merging performance of HET merge sort can not compete with our P2P-based radix partitioning approach, as the main memory bandwidth is orders of magnitude lower than that of the GPUs.

With RMG sort, we employ an MSB radix partitioning strategy which allows us to interleave the sorting computation with the device-to-host copy. We sort individual buckets while copying the sorted buckets back to main memory. As a result, we already outperform the two merge-based multi-GPU sorting algorithms for two GPUs, even though the benefit of RMG sort's improved scaling does not apply for $g = 2$. On two modern multi-GPU accelerator platforms that include up to eight GPUs, we evaluate RMG sort to outperform P2P merge sort up to 1.26× and HET merge up to 1.8×.

Performance Impact of Data Skew. Since the performance of radix-based algorithms typically decreases when processing skewed data, we analyze RMG sort's performance for zipfian distributions and input keys of varying bit entropy values. When sorting with two GPUs on the IBM AC922, we see that RMG sort's execution time increases considerably for zipfian distributions. Compared to P2P merge sort, RMG sort becomes up to 13% slower for highly skewed input keys, i.e. for zipf exponents of 1.5. However, we observe that scaling to increasing numbers of GPUs mitigates this performance bottleneck because the time duration of the radix partitioning phase reduces linearly. Thus, the performance of RMG sort is significantly less affected by data skew when sorting with eight GPUs. On

the DGX A100, RMG sort remains the fastest multi-GPU sorting algorithm for all distribution types, with speedups of at least 11% compared to P2P merge sort, and at least 1.7× over HET merge sort.

Future Work. Still, to mitigate the performance decrease for skewed data on systems like the IBM AC922, we propose the following optimizations to be implemented as part of future work: To ensures optimal numbers of buckets, the bit range on which we partition the keys during the radix partitioning phase should be dynamically adjusted. Thereby, the interleaved sorting and copy operations achieve optimal throughput rates for any bit entropy. To mitigate the performance overhead that results from zipfian distributions we propose to further optimize the kernel functions of the radix partitioning phase. Both for the histogram computation and the key scattering, a reduction of the performed shared memory atomic operations promises even further speedups.

To sort large out-of-core data, we suggest extending our multi-GPU sorting algorithm by a preliminary partitioning step on the CPU which divides the input data into chunks that are no bigger than the combined GPU memory. The partitioning step also needs to ensure that the chunks contain keys of distinct value ranges. Then, the entire data set can be sorted through subsequent, independent sorting rounds using RMG sort as the in-core algorithm.

7 Related Work

To the best of our knowledge, we propose the first multi-GPU sorting algorithm that is based on radix-partitioning (RMG sort). Since we build upon prior research on single-GPU sorting approaches, we discuss the recent research concerning single-GPU sorting algorithms. With regards to multi-GPU acceleration, we discuss related work not only for sorting but for joins as well. Both research areas contain valuable insights in terms of inter-GPU communication, multi-GPU algorithm design, and performance optimizations. Finally, we give an overview of today's landscape of commercially available GPU-accelerated database management systems.

7.1 GPU Sorting Algorithms

Various single-GPU sorting algorithms have been proposed [5, 8, 13, 17, 29, 30, 57, 58, 62]. The best performing approaches are radix sort algorithms [34, 23, 35, 44].

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 [22]. Merrill et al. design an LSB radix sort algorithm that dynamically adjusts the number of keys that a thread processes [36]. They also implement an analytical performance model that determines the optimal radix length (i.e. number of bits per partitioning pass c) to reduce the memory workload for any given target architecture. Their approach has been integrated into NVIDIA's high-performance CUB library [41]. Stehle et al. publish an MSB radix sort algorithm that increases the number of bits considered at a time to $c = 8$ [63]. In subsequent partitioning passes, they partition the keys into smaller and smaller buckets until a local sort algorithm sorts the buckets in on-chip memory.

Recently, Adinets et al. improve the performance of the LSB radix sort algorithm provided in NVIDIA's CUB library in release version 1.11.0 [41], making it the fastest single-GPU sorting algorithm today [34]. Similar to Stehle et al., they increase the number of bits per pass to $c = 8$. They further improve the alignment of the global memory write-back phase and reduce the overall memory workload from $3n$ to $2n$ by optimizing the parallel prefix scan computation [1].

These sorting algorithms are single-GPU approaches. We use the fastest single-GPU sorting primitive, the LSB radix sort from the CUB library [41], as part of our novel multi-GPU sorting algorithm implementation.

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 and merges

large-out-of-core data, i.e. data set sizes that exceed the combined GPU memory capacity [52]. The authors use multiple GPUs to sort chunks that fit into the GPU's memory. After copying the sorted chunks back to main memory, the CPU finds splitter elements across the sorted chunks to form disjoint data sets that individual GPUs can merge independently. To mitigate the overhead of transferring data back and forth between the CPU and the GPUs, the authors overlap memory transfers and computation. Gowanlock et al. publish a heterogeneous multi-GPU sorting algorithm for large data as well. They sort data chunks using multiple GPUs and merge them in one final multiway-merge phase on the CPU [18]. Similar to the approach by Stehle et al. [63], the authors overlap the GPU computation with the data transfers when sorting out-of-core data in multiple sorting rounds. They simultaneously sort the chunks of round i on the GPUs, copy the sorted chunks of round $i-1$ back to main memory, and transfer the chunks of round $i+1$ to the GPUs. While both algorithms sort large-out-of-core data, neither one utilizes inter-GPU communication. RMG sort is limited by the combined memory capacity of the GPUs. We sort in-memory data with g GPUs cooperatively by exchanging keys between the GPUs via P2P interconnects. Thus, RMG sort could improve the performance of an out-of-core sorting algorithm by reducing the number of sorted chunks that need to be merged by a factor of g.

Tanasic et al. propose a merge-based multi-GPU sorting algorithm for data sets that fit into the combined GPU memory [64]. After each GPU sorts its data chunk, the GPUs cooperatively merge all chunks using P2P interconnects. The authors design a multi-stage merge algorithm. In each stage, a pivot selection determines which blocks of consecutive keys to exchange between pairs of GPUs. Because their pivot selection can only merge two arrays at a time, multiple merge stages become necessary for more than two GPUs. Thus, the merging workload and the P2P transfers of their algorithm scale linearly with the number of GPUs. In contrast, RMG sort requires only one P2P key swap between all GPUs, independent of the number of GPUs.

7.2 Multi-GPU Query Processing

Recent work utilizes multiple GPUs with state-of-the-art interconnects to accelerate query processing operators. Mostly, multi-GPU join algorithms have been proposed and evaluated. They contain valuable information about the efficient design of multi-GPU algorithms.

Rui et al. implement and evaluate three multi-GPU join algorithms for large out-of-core data on systems with NVLink interconnects. They analyze a nested-loop join, a sort-merge join, and a hybrid join that uses radix partitioning [56]. The

first step of their sort-merge join algorithm is to sort both relations using multiple GPUs. For this, the authors first sort chunks of data that fit into the GPU memory. The sorted chunks are copied back to main memory, where a merge path partitioning algorithm partitions the chunks into subsets that are cooperatively merged on the GPUs. Similar to the approach by Peters et al., the input data is transferred over the CPU-GPU interconnects more than two times. To mitigate the performance overhead of the data transfers, they overlap the copy and compute operations. Their hybrid join uses radix partitioning to partition the input tuples into disjoint buckets that the GPUs process using the sort-merge join as the in-memory join algorithm. Since the authors evaluate their hybrid join to perform best, we suggest that future research should evaluate the performance benefit of a hybrid multi-GPU sorting algorithm. For example, if the radix partitioning phase of RMG sort generates only a few big buckets due to data skew, these could be split into multiple sub-buckets to increase the throughput of the sort-copy overlap. Afterwards, the CPU can merge the sub-buckets in a repair phase.

Paul et al. publish a partitioned hash join for multiple GPUs and evaluate their approach on NVIDIA's DGX-1 platform [50]. Specific to their system's interconnect topology, they employ a multi-hop routing strategy where P2P data transfers between GPUs are adaptively redirected based on the most efficient route. They evaluate their join to achieve a high P2P interconnect bandwidth utilization. Gao et al. design a multi-GPU hash join for GPU clusters with up to 1024 GPUs. They harness NVLink interconnects within a node of eight GPUs and GPUDirect RDMA for efficient out-of-node communication over Infiniband [15]. To reduce the inter-GPU communication, they employ a GPU data compression strategy that might benefit P2P-based multi-GPU sorting algorithms, such as RMG sort.

7.3 GPU-Accelerated Database Systems

Today, commercially available GPU-accelerated database systems are emerging. BrytlytDB is a PostgreSQL-based database system that accelerates large-data analytics with GPUs [7]. BlazingSQL offers a distributed SQL engine in Python that processes raw data, scaling to thousands of GPUs [6]. Heavy.AI (formerly OmniSciDB) provides data analytics and data visualization products and accelerates both with multiple GPUs [24]. SQream DB is a GPU-based SQL database system that leverages the massive parallelism of multi-GPU servers to accelerate data analytics workloads [65]. We focus on accelerating a single database-relevant operation, namely sorting, with modern multi-GPU platforms. GPU-accelerated database systems might benefit from integrating a high-performance multi-GPU sorting primitive, either as a stand-alone operation or as part of a sort-merge join.

8 Conclusion

In this thesis, we designed, implemented, and evaluated a novel radix-partitioning-based multi-GPU sorting algorithm (RMG sort). To the best of our knowledge, RMG sort is the first multi-GPU sorting algorithm that employs radix partitioning, while all previous approaches are sort-merge approaches. Our presented algorithm sorts in-memory data, i.e. data sets that fit into the combined device memory of the GPUs. We sort the input keys across the GPUs by utilizing modern, non-blocking, all-to-all P2P interconnects, such as NVLink and NVSwitch, to their fullest extent. For this, we designed an MSB radix partitioning strategy that scales linearly with the input size and reduces the inter-GPU communication compared to prior merge-based algorithms. By exchanging the radix partitions between all GPUs in parallel, RMG sort requires only one all-to-all P2P key swap, independent of the number of GPUs.

We implemented our multi-GPU sorting algorithm and presented our performance optimizations in detail. To enable further research and reproducible evaluation results, we published our source code and our automated benchmark scripts. We evaluated RMG sort on modern accelerator platforms with up to eight GPUs and high-bandwidth interconnects. We compared its performance to highly optimized parallel CPU sorting algorithms and two state-of-the-art, merge-based, multi-GPU sorting algorithms.

Our evaluation shows that RMG sort utilizes state-of-the-art high-speed P2P interconnects, such as NVLink 2.0, NVLink 3.0, and NVSwitch, more efficiently than prior merge-based algorithms, especially for more than two GPUs. By overlapping the sorting computation and the device-to-host transfers, we also outperform the merge-based algorithms on two GPUs. We find that RMG sort significantly outperforms parallel CPU-based sorting algorithms up to 20× and two state-of-the-art merge-based multi-GPU sorting algorithms up to 1.26× and 1.8×. RMG sort scales better with increasing numbers of GPUs compared to the merge-based sorting algorithms. Thus, RMG sort further benefits from future accelerator platforms given that hardware vendors continue to increase the number of GPUs and the P2P interconnect bandwidth [47, 43, 48].

References

[1] A. Adinets. 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.

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

[3] 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.

[4] 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.

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

[6] Inc. BlazingSQL. Open-source sql in python. https://blazingsql.com/, December 2021. Last accessed: 2022-04-26.

[7] Brytlyt. The fastest and most advanced GPU database in the world. https://www.brytlyt.com/what-we-do/brytlytdb/, April 2022. Last accessed: 2022-04-26.

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

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

[10] Maria Colgan and Oracle Database Insider. 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.

[11] Intel Corporation. An Introduction to the Intel® QuickPath Interconnect. https://www.intel.ca/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf, January 2009. Last accessed: 2022-04-26.

[12] Intel Corporation. Oneapi threading building blocks. `https://github.com/oneapi-src/oneTBB`, June 2021. Last accessed: 2022-04-26.

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

[14] 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.

[15] Hao Gao and Nikolay Sakharnykh. Scaling Joins to a Thousand GPUs. In *Proceedings of the 12th International Workshop on Accelerating Analytics and Data Management Systems*. Association for Computing Machinery, August 2021.

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

[17] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 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, page 325–336. Association for Computing Machinery, 2006.

[18] Michael Gowanlock and Ben Karsin. Sorting Large Datasets with Heterogeneous CPU/GPU Architectures. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 560–569. Institute of Electrical and Electronics Engineers, 2018.

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

[20] Khronos OpenCL Working Group. The OpenCL Specification. `https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`, November 2021. Last accessed: 2022-04-26.

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

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

[23] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[24] HEAVY.AI. HEAVY.AI Product Overview. https://www.heavy.ai/product/overview, April 2022. Last accessed: 2022-04-26.

[25] Michael Herf. Radix Tricks. http://stereopsis.com/radix.html, December 2001. Last accessed: 2022-04-26.

[26] Floating-Point Working Group IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[27] AMD Inc. AMD EPYC 7002 Series Processors: A New Standard for the Modern Data Center. https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf, April 2020. Last accessed: 2022-04-26.

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

[29] Peter Kipfer and Rüdiger Westermann. 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.

[30] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU Sample Sort. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, 2010.

[31] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 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.

[32] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 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*, pages 1633–1649. Association for Computing Machinery, 2020.

[33] Z. Majo and T. R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, pages 1–10. Association for Computing Machinery, 2011.

[34] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. Evaluating Multi-GPU Sorting with Modern Interconnects. In *2022 ACM SIGMOD International Conference on Management of Data (SIGMOD '22)*. Association

for Computing Machinery, 2022.

[35] Duane Merrill and Michael Garland. 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.

[36] Duane Merrill and Andrew Grimshaw. 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.

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

[38] 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.

[39] 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.

[40] 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.

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

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

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

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

[45] 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.

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

[47] NVIDIA. NVIDIA DGX H100: The Gold Standard for AI In-

frastructure. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-h100-datasheet.pdf, March 2022. Last accessed: 2022-04-26.

[48] 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.

[49] NVIDIA. Thrust Documentation: thrust::exclusive_scan. https://thrust.github.io/doc/group__prefixsums_ga7be5451c96d8f649c8c43208fcebb8c3.html, April 2022. Last accessed: 2022-04-26.

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

[51] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu. Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, pages 209–218. Association for Computing Machinery, 2019.

[52] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 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)*, pages 1–8. Institute of Electrical and Electronics Engineers, 2010.

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

[54] Orestis Polychroniou and Kenneth A. Ross. 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*, pages 755–766. Association for Computing Machinery, 2014.

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

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

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

[58] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 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*, pages 351–362. Association for Computing Machinery, 2010.

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

[60] Debendra Das Sharma and Siamak Tavallaei. 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.

[61] J. Singler and B. Konsik. The GNU libstdc++ Parallel Mode: Software Engineering Considerations. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, pages 15–22. Association for Computing Machinery, 2008.

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

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

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

[65] SQream Technologies. Bringing the Power of the GPU to the Era of Massive Data. `https://sqream.com/product/data-acceleration-platform/sql-gpu-database/`, April 2022. Last accessed: 2022-04-26.

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

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

[68] Keliang Zhang and Baifeng Wu. 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*, pages 989–994, 2012.

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Die selbstständige und eigenständige Anfertigung versichert an Eides statt:

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used. The independent and unaided completion of this thesis is affirmed by affidavit:

Potsdam, 01.05.2022

Ivan Ilić