Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects

Clemens Lutz*
clemens.lutz@tu-berlin.de
TU Berlin
Berlin, Germany

Sebastian Breß†
sebastian.bress@snowflake.com
Snowflake
Berlin, Germany

Steffen Zeuch
steffen.zeuch@dfki.de
DFKI GmbH
Berlin, Germany

Tilmann Rabl
tilmann.rabl@hpi.de
HPI, University of Potsdam
Potsdam, Germany

Volker Markl
volker.markl@tu-berlin.de
DFKI GmbH, TU Berlin
Berlin, Germany

ABSTRACT

Database management systems are facing growing data volumes. Previous research suggests that GPUs are well-equipped to quickly process joins and similar stateful operators, as GPUs feature high-bandwidth on-board memory. However, GPUs cannot scale joins to large data volumes due to two limiting factors: (1) large state does not fit into the on-board memory, and (2) spilling state to main memory is constrained by the interconnect bandwidth. Thus, CPUs are often the better choice for scalable data processing.

In this paper, we propose a new join algorithm that scales to large data volumes by taking advantage of *fast interconnects*. Fast interconnects such as NVLink 2.0 are a new technology that connect the GPU to main memory at a high bandwidth, and thus enable us to design our join to efficiently spill its state. Our evaluation shows that our *Triton* join outperforms a no-partitioning hash join by more than 100× on the same GPU, and a radix-partitioned join on the CPU by up to 2.5×. As a result, GPU-enabled DBMSs are able to scale beyond the GPU memory capacity.

CCS CONCEPTS

• **Information systems** → **Database management system engines**; **Join algorithms**.

KEYWORDS

Modern hardware, GPU, data transfer bottleneck, out-of-core, TLB

ACM Reference Format:
Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton Join: Efficiently Scaling to a Large Join State on GPUs with Fast Interconnects. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3514221.3517911

*Research partially conducted while the author was employed by DFKI GmbH.
†Work done while the author was employed by TU Berlin.

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

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517911

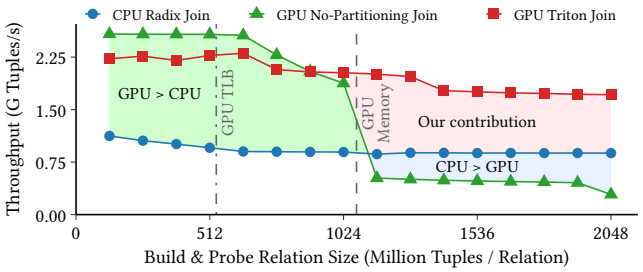


Figure 1: Out-of-core state results in a performance cliff and a slow-down, despite using a fast interconnect. In contrast, our Triton join gracefully scales to joins with a large state.

1 INTRODUCTION

GPUs are being commercially adopted to accelerate query processing [86]. They are available from all major cloud vendors, including Amazon EC2, Google Compute Engine, and Microsoft Azure, and are being integrated into academic [24, 29, 39, 51, 106, 122] and commercial GPU-enabled database management systems [78, 121]. These GPU-enabled DBMSs see the most benefit for join and group-by aggregation queries with an in-GPU state [23, 39, 43, 47, 132, 145]. However, database research suggests that GPUs cannot efficiently scale to large, out-of-core state due to the *data transfer bottleneck* [39, 43, 119, 132, 145].

The data transfer bottleneck is caused by the low bandwidth and high latency of the interconnect between the GPU and the CPU [81]. This hardware limitation leads to a narrow scope where DBMSs benefit from GPUs. However, as we illustrate in Figure 1, higher interconnect bandwidth is necessary, but not sufficient for high scalability. Even if the GPU is given a faster interconnect, the CPU outperforms the GPU when joining two large data sets. Therefore, we identify three fundamental challenges that need to be addressed to widen the applicability of GPUs:

Scalability. GPU joins store their state in GPU memory to increase throughput [48, 66, 92, 105, 132]. Due to the limited capacity of GPU memory, GPU joins cannot efficiently scale to a large state [81]. In contrast, CPUs [7, 59, 139] have two orders-of-magnitude higher memory capacity than GPUs [6, 97, 100]. Thus, we must adapt GPU joins to spill their state to CPU memory in order to achieve scalability.

Robustness. Spilling the join state to CPU memory results in a performance cliff [81]. These sharp performance drops are difficult to account for in query optimizers, because cardinality estimates can be significantly wrong [28, 87]. Thus, GPU-enabled DBMSs must gracefully scale to large data sizes for a consistent user experience.

Efficiency. State-of-the-art approaches reduce interconnect transfers by shifting computations from the GPU to the CPU [42, 44, 113, 134, 140]. However, both interconnect bandwidth and CPU cycles are scarce resources. DBMSs should use the GPU to offload computations from the CPU, while maximizing performance.

A newly emerging technology, *fast GPU interconnects*, has the potential to help us address the above challenges. Fast interconnects provide GPUs with high-bandwidth, cache-coherent access to main memory. Recent examples include NVLink 2.0 [97] and 4.0 [53], Infinity Fabric [6], and Compute Express Link 2.0 [31].

In this work, we investigate how fast interconnects can improve GPU join throughput. Our goal is to enable GPUs to process joins with a state that exceeds the GPU memory capacity. Thus, we consider joins smaller and larger than the GPU memory. For large joins, we partition data out-of-core in CPU memory using the fast interconnect to achieve data locality during the join. In contrast, small joins require us to cache all intermediate results in GPU memory to avoid unnecessary data transfers. We combine the GPU-based partitioning and the caching in our new, hierarchical hybrid hash join algorithm: ${}^{3}\mathrm{H}^{+} \equiv$ the *Triton join*.

Overall, our contributions are as follows:

(1) We investigate the requirements of an out-of-core GPU join in regard to fast interconnects, and identify hardware bottlenecks that limit scalability (Section 3).
(2) We propose a new GPU radix partitioning algorithm that takes advantage of fast interconnects to achieve a high bandwidth and scale to large data volumes (Section 4).
(3) We present our new Triton join algorithm, a scalable radix-partitioned GPU hash join that partitions data using the GPU and caches partitioned data in GPU memory (Section 5).

The further structure of this paper is as follows. In Section 2, we briefly introduce modern GPU hardware and joins. Next, we motivate our approach by revisiting out-of-core GPU joins in Section 3. After that, we demonstrate our out-of-core radix partitioning approach in Section 4, and then overcome these challenges with our Triton join in Section 5. In Section 6, we show our evaluation and discuss our insights. Finally, we review related work in Section 7 and conclude in Section 8.

2 BACKGROUND

In this section, we provide an overview of the hardware architecture of a fast-interconnect system, and of hardware-sensitive joins.

2.1 GPUs and Fast Interconnects

Fast interconnects are able to connect a GPU and a CPU with high bandwidth, a unified address space, and system-wide cache-coherence [6, 58, 97, 104]. As an example, we show the architecture of an IBM AC922 system [93] with an IBM POWER9 CPU and an Nvidia V100 GPU in Figure 2. Although we focus on NVLink 2.0, other fast interconnects such as Compute Express Link [31], Infinity Fabric [6], and OpenCAPI [103] specify a similar system architecture.

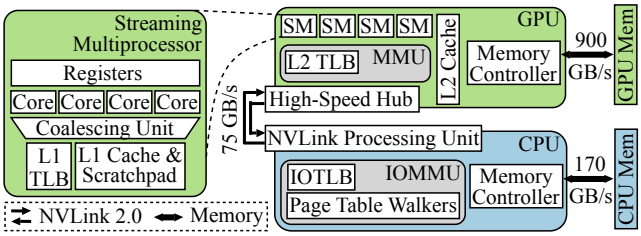


Figure 2: Hardware architecture of a system with a fast interconnect. The electrical bandwidths are annotated.

Overview. The system consists of one or more multi-core CPUs and discrete GPUs. Each GPU is attached to a CPU by NVLink 2.0, and has at least 16 GiB of on-board memory with 900 GB/s of bandwidth [97]. Each CPU comes with up to 4 TiB of memory attached at 170 GB/s [139].

GPU Architecture. GPUs such as Nvidia "Volta" [27, 97] and AMD "Vega" [3] execute threads in parallel on up to 84 *streaming multiprocessors (SM)*. Each SM schedules threads in hardware [79], and provides up to 65 thousand registers to hide memory latencies of up to 2 µs [38]. Each SM consists of 32–128 cores, on which 32 threads are physically executed together as a *warp* [101]. GPU programming languages abstract multiple warps as a *thread block* [5, 101]. Warps *coalesce* (i.e., group) adjacent memory accesses into a single *memory transaction* to improve memory transfer efficiency [32, 94] and to reduce the memory address translation request rate [118]. The GPU caches memory accesses in its L1 and L2 caches [27].

Address Translation. CPUs and GPUs share a single address space [27]. A program's virtual addresses are translated into physical addresses on a memory access [27]. Translations are cached in a hierarchy of *translation lookaside buffers (TLBs)* [126]. Although GPU vendors do not publish details, it is widely accepted that "Pascal" and newer GPUs have two TLB levels [61, 64, 65, 69, 75]. The L1 TLB is private to each SM, while multiple SMs share a L2 TLB [10, 69]. In addition, CPU memory accesses are translated by an *I/O memory management unit (IOMMU)* [21, 22, 27, 57]. The IOMMU is part of the CPU and contains an *IOTLB* and 12 parallel *page table walkers* [21, 56, 57]. On a IOTLB miss, the page table walkers fetch a translation from a page table stored as a radix tree in CPU memory [56, 62]. GPUs are able to coalesce page table walks [69, 110, 111], and thus the IOMMU returns up to 16 translations at a time [57]. Both the GPU and the IOMMU support 4 KiB, 64 KiB and 2 MiB pages [21, 56, 57, 98], and the IOMMU also supports 1 GiB pages [21, 56].

NVLink 2.0. The GPU connects to the CPU at 75 GB/s in both directions, for a total of 150 GB/s [93]. The connection is mediated by a high-speed hub on the GPU [96], and an NVLink processing unit on the CPU [57]. These units send and receive packets consisting of a 16-byte header and 1–256 bytes of payload [38, 57]. Small payloads incur additional overhead. Small reads are padded to a 32-byte payload [38]. Small writes require a 16-byte "byte enable" header extension [38], that specifies which payload bytes to write [103]. The GPU SMs support packets up to 128 bytes (i.e., an L1 cacheline [65, 71]), and direct memory access copy engines handle packets up to 256 bytes [57]. We calculate that the maximum effective bandwidth is 62–65.7 GiB/s per direction.

2.2 Hardware-Sensitive Joins

In this work, we extend the parallel radix-partitioned hash join algorithm, as introduced by Kim et al. [72]. The core idea of partitioned join algorithms is to increase data locality, such that we are able to store the hash table in the processor cache [133]. The low access latency of the cache improves the performance of random accesses to the hash table. This technique also applies to other hash-based relational operators, such as group-based aggregations [35, 133, 144] and duplicate elimination [35]. In contrast to other partitioning methods, radix partitioning has the advantage that it helps to reduce TLB misses [84].

Radix-partitioned hash joins have two constraints. First, the hash tables must fit into the cache. As the cache has a constant size, larger data volumes require a higher *fanout* (i.e., number of partitions) to keep the size of each partition constant. Second, a high fanout incurs frequent TLB misses if the fanout is higher than the number of available TLB entries. TLB misses are expensive, because resolving a miss involves between 1–6 memory accesses [15].

For large data sets, there exists a fundamental tension between these two constraints. On the one hand, a high fanout is necessary to efficiently look up hash table entries. On the other hand, a high fanout increases the cost of partitioning. Database literature addresses this trade-off by optimizing high-fanout partitioning on CPUs and GPUs as follows.

On CPUs, we reduce TLB misses through *software write-combining (SWWC)* [128]. SWWC reduces TLB misses by intermediately buffering tuples in the processor cache. Tuples are then written out to their final positions in batches. Thus, a batch size of N reduces the amount of TLB misses by a factor of N [130]. Flushing buffers can be optimized with *non-temporal stores*, that avoid polluting the cache [142]. Finally, storing partition offsets in a *micro-row layout* reduces cache misses and requires less cache space [11, 131].

Optimization techniques for GPUs differ from those for CPUs. Scattered writes can be coalesced by partially sorting tuples in scratchpad memory [127]. A thread block works together to sort tuples and flush them to memory. In contrast to SWWC on CPUs, all tuples are flushed at the same time. If the batch size is larger than the fanout, it follows that each memory transaction must write out multiple tuples per partition. However, although writes introduce coalescing opportunities, misalignment can still prevent coalescing, thereby reducing efficiency.

On recent GPUs that feature efficient atomic additions [64], partitioning can be improved by replacing prefix scan with a linear allocator for a single data pass within the scratchpad [125, 140]. A linear allocator tracks free array slots using an atomically incrementing counter. We refer to this approach as the *linear allocator software write-combining (Linear)* partitioning algorithm.

3 REVISITING OUT-OF-CORE GPU JOINS

Existing out-of-core join algorithms assume to varying degrees that interconnect bandwidth is the bottleneck, which fundamentally shapes the design strategy underpinning the algorithm. In this section, we examine how fast interconnects change this assumption, and study the impact of the interconnect on the join strategy.

We first discuss the CPU-partitioned join strategy (Section 3.1), as we find it an enlightening point in the design space due to its focus on the transfer volume. Then, we show that fast interconnects enable the GPU to process out-of-core data (Section 3.2). Based on this insight, we argue that fast interconnects open the door for a new high-level design, the GPU-partitioned join strategy (Section 3.3). Finally, we analyze the hardware capabilities and limitations to inform our detailed design choices (Section 3.4).

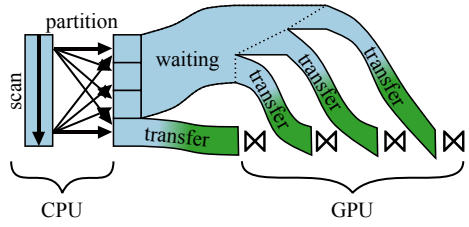


Figure 3: The CPU-partitioned join strategy splits state into small partitions before starting to process data on the GPU.

3.1 The CPU-Partitioned Join Strategy

A recent *CPU-partitioned join strategy* proposes to partition the data on the CPU before transferring it to the GPU [134]. The goals are to minimize data transfers across the interconnect, and to access the join's state efficiently in GPU memory.

We outline this join strategy in Figure 3. It consists of three phases. First, the CPU partitions the data into working sets that individually fit into GPU memory. Then, the strategy transfers a working set to the GPU. Third, the GPU joins the relations within the working set. Steps two and three are repeatedly executed in a pipeline to hide the transfer latency. Although the partitioning and transfer may overlap, at least one relation must be completely partitioned before starting the join.

The CPU can initially transfer only a fraction of the data to the GPU, as only one working set completely fits into GPU memory at a time. Let this fraction be $\alpha := \frac{|\text{working set}|}{|\text{data}|}$. To saturate the interconnect bandwidth, the CPU must partition at a rate higher than $1/\alpha \times$ [transfer bandwidth]. For example, with a 12 GiB/s transfer rate and $\alpha = 1/4$, the CPU must partition at $4 \cdot 12$ GiB/s = 48 GiB/s. However, the partitioning throughput must increase to 260 GiB/s in order to saturate a 65 GiB/s fast interconnect.

We argue that such a partitioning rate is unrealistically fast, as it would exceed the CPU memory bandwidth even when using multiple CPUs. As a result, the CPU-partitioned strategy underutilizes the GPU and the fast interconnect.

3.2 Fast Interconnects Outpace CPUs

Fast interconnects provide a new opportunity to utilize hardware resources efficiently by computing all join phases on the GPU. We show that if the join is optimized for a fast interconnect, then the GPU is able to outperform a CPU even for this data-intensive task.

We demonstrate our insight in Figure 4. We measure the partitioning throughput of a CPU and a GPU, and distinguish between the two extreme cases: (a) either all resulting partitions fit into GPU memory or (b) all partitions are stored to CPU memory. Both processors read the base relation from CPU memory, and split the data into 512 partitions. We observe that in both cases the GPU is

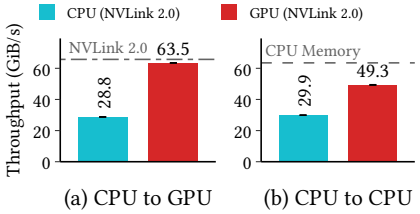


Figure 4: Data partitioning throughput of a CPU and a GPU for different source and destination locations.

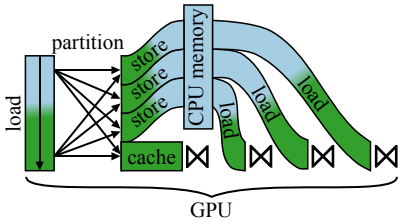


Figure 5: The GPU-partitioned join strategy processes both the partition and join phases on the GPU, spilling state to CPU memory if necessary.

faster than the CPU. Conversely, despite transferring all partitions at once ($\alpha = 1$), the CPU cannot saturate the fast interconnect.

Our take-away is that fast interconnects require a new approach for GPU joins to take full advantage of the hardware. The existing CPU-partitioned strategy underutilizes the GPU *and* the fast interconnect. Thus, a GPU-centric approach would be able to utilize the available hardware resources better.

3.3 The GPU-Partitioned Join Strategy

Our goal is to compute the join end-to-end on the GPU. For this reason, we propose a new, *GPU-partitioned join strategy* that is optimized for GPUs with fast interconnects.

We highlight our GPU-partitioned join strategy in Figure 5. Our strategy works as follows. In the partitioning phase, the GPU loads the data from CPU memory, and caches the resulting partitions in GPU memory. If this state exceeds the GPU memory capacity, the GPU spills the remainder to CPU memory. In the join phase, the GPU loads the spilled state from CPU memory again.

We overlap transfers and computations using two methods. For phases that consist of a single GPU kernel, we rely on the hardware cache-coherence [81]. In contrast, for phases consisting of multiple kernels, we describe a new transfer method in Section 5.2.

Overall, the advantages of our strategy are that (1) computation is offloaded to the GPU and that (2) the join gracefully scales to out-of-core state. The trade-off is a 1–2× higher transfer volume, depending on how many data are cached vs. spilled to CPU memory.

3.4 Capabilities of Fast Interconnects

To efficiently implement our GPU-partitioned join strategy in practice, we require an in-depth understanding of the interconnect hardware. Crucially, if data is spilled during the partitioning phase, the GPU performs random writes to CPU memory [84]. Thus, we

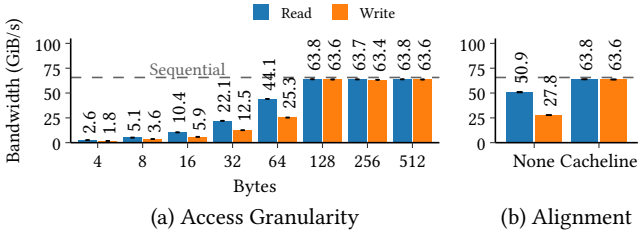


Figure 6: GPU interconnect bandwidth of a random access pattern to CPU memory with varying access granularities.

analyze the key metrics for random accesses: the interconnect bandwidth of fine-grained memory accesses, and the TLB miss latency.

3.4.1 Efficient Transfers with Fine Granularity. Ideally, a join running on the GPU achieves the full interconnect bandwidth when accessing CPU memory. However, the bandwidth achieved in practice depends on the access granularity, as the GPU executes memory accesses in units of *memory transactions* [101]. Memory transactions have a hardware-specific size. If accesses are fine-grained, i.e., smaller than the memory transaction size, then each memory transaction only carries a partial payload. This leads to a reduced bandwidth utilization. Although memory transactions in GPU memory have been researched [60, 71, 135], prior work does not consider the effect on the interconnect bandwidth.

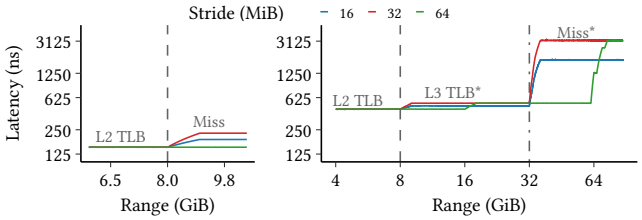
Setup. We experimentally determine the minimum required memory access granularity to achieve the full interconnect bandwidth in Figure 6(a). In the experiment, the GPU randomly accesses CPU memory on the nearest NUMA node. We first scale the access granularity from 4–16 bytes by increasing the integer type from 32–128 bits. Then, we continue to scale by coalescing 2–32 threads (i.e., up to a warp) for 32–512-byte accesses. We measure read and write accesses within a 1 GiB array, and efficiently generate the random access pattern via a linear congruential generator [73]. All accesses are aligned according to their granularity, i.e., a 512-byte access is aligned to 512 bytes.

Results. In the measurement, we observe that the interconnect bandwidth grows linearly with the access granularity. Small reads up to 64 bytes are 44–74% faster than writes. At 128 bytes, the bandwidth of random accesses equals the bandwidth of a coalesced sequential access pattern.

Furthermore, in Figure 6(b), we determine that misaligned accesses reduce the achieved interconnect bandwidth. We measure that misaligning a 512-byte memory access by 16 bytes reduces the bandwidth by 20% for reads and 56% for writes.

Analysis. From our results, we deduce that "Volta" GPUs coalesce CPU memory accesses via NVLink 2.0 into 128-byte memory transactions (or larger) instead of 32 bytes in GPU memory [71, 137]. These transactions are aligned to 128-byte cachelines. Our analysis is substantiated by vendor documentation on NVLink 2.0 [57, 101] and an investigation of PCI-e [90]. However, it remains unclear why small reads outperform small writes.

Our findings differ from GPU literature, which suggests that GPU random accesses to CPU memory are slower than sequential transfers [81], and that GPU programmers should coalesce memory accesses of warps with natural alignment on the data type [4, 101].



(a) GPU Memory (b) CPU Memory

Figure 7: TLB miss latency for GPU memory, and for CPU memory via NVLink 2.0.

Overall, if accesses are *perfectly coalesced* as described above, GPUs are able to achieve the full interconnect bandwidth for random CPU memory accesses at a 128-byte granularity.

3.4.2 TLB Miss Cost with Fast Interconnects. Fast interconnects give GPUs high-bandwidth access to terabytes of data in CPU memory. Due to the large data size, a join randomly accesses thousands of memory pages. As a result, virtual to physical memory address translations impact join throughput. We quantify the address translation cost for GPUs, and discover that TLB misses when using a fast interconnect are up to an order-of-magnitude more expensive than TLB misses in GPU memory.

Setup. In Figure 7, we compare the TLB miss costs of GPU accesses to GPU memory and to CPU memory. We measure the latency of individual memory accesses with fine-grained pointer chasing [88]. We perform 16, 32, and 64 MiB strides in a memory range of 6–10.7 GiB in GPU memory and a range of 1–87.5 GiB in CPU memory. We allocate 2 MiB huge pages in CPU memory on the NUMA node closest to the GPU. To avoid page fragmentation, we preallocate huge pages early at boot time [91]. To prevent the hardware from caching translations across runs, we flush the IOTLB before each run by calling the `mprotect` system call [41]. We observe that the GPU TLBs are flushed by the CUDA runtime before each kernel launch. As the L1 data cache is virtually tagged and thus does not incur address translations [65], we bypass the L1 cache with the `cg` PTX cache hint [102].

Results. In GPU memory, we observe that the GPU L2 TLB covers 8 GiB. We measure a L2 TLB hit latency of 151.9 ± 4.8 ns and a miss latency of 226.7 ± 4.8 ns. Our measurements match the results of Jia et al. [65], who state that "Volta" GPUs have a L1 TLB in addition to the L2 TLB.

In CPU memory, the L2 TLB also covers 8 GiB with a hit latency of 449.7 ± 32.4 ns. Beyond the L2 TLB, we notice two miss plateaus, one at 9.5–32 GiB and another above 37 GiB. For the first, we measure a latency of 532.9 ± 45.8 ns, and 3186.4 ± 154.0 ns for the second. We speculatively name the plateaus *L3 TLB** and *Miss**.

Analysis. We observe that the L2 TLB page size is 32 MiB not only in GPU memory [65, 69], but also in CPU memory. Thus, 16 physically adjacent 2 MiB pages are likely coalesced on a page table walk [33, 34, 57, 69, 110].

However, we lack evidence to fully explain the TLB misses that occur in CPU memory. The high miss latency (Miss*) indicates a GPU TLB miss, that results in an IOTLB or IOMMU lookup. In contrast, the L2 TLB miss penalty to the L3 TLB* is only 83 ns. On the one hand, this is likely too short to traverse the interconnect. On the other hand, the L2 TLB miss penalty in GPU memory is similar at 75 ns. As NVLink 2.0 enables a system-wide page table [57], we assume that the GPU does not duplicate the table in GPU memory. Thus, our results might indicate that another translation caching layer exists [18, 20, 61], that is distinct from the IOTLB. However, we leave a deeper investigation to future work.[1]

In conclusion, TLB misses are a hard problem to mitigate for out-of-core algorithms. However, we find that if an algorithm carefully manages its TLB misses and access granularity, then the GPU can achieve a high interconnect bandwidth even for random accesses.

Table 1: Partitioning design goals.

Algorithm	Space Efficient	Perfect Coalescing	High Fanout
SWWC	✗	✗	✗
Linear	✓	✗	✗
Shared	✓	✓	✗
Hierarchical	✓	✓	✓

4 EFFICIENTLY PARTITIONING DATA OVER A FAST INTERCONNECT

In order to join large data efficiently, the GPU first needs to partition the data out-of-core. We transform our hardware insights from Section 3.4 into concrete design goals (Section 4.1), on which we base two new radix partitioning algorithms for GPUs. First, we increase the interconnect utilization of random writes in our *shared software write-combining (Shared)* algorithm (Section 4.2). In a next step, we reduce GPU TLB misses for high fanouts in our *hierarchical software write-combining (Hierarchical)* algorithm (Section 4.3).

4.1 Design Goals

Achieving high partitioning throughput requires us to consider both the GPU architecture and the fast interconnect. To this end we formulate three design goals. First, the algorithm should be space efficient, due to the small scratchpad capacity. As a thread block shares the scratchpad, all accesses to the scratchpad must be thread-safe. Second, random memory accesses over the interconnect should adhere to perfect coalescing (see Section 3.4.1). Finally, large data sets require a high fanout to reduce the size of each partition. This incurs TLB misses, which should be avoided (see Section 3.4.2).

State-of-the-art algorithms do not achieve these goals, as we summarize in Table 1. The SWWC algorithm allocates thread-private buffers, as CPUs have large caches. Linear is designed for in-GPU partitioning, and opportunistically coalesces writes by sorting batches of tuples. Thus, we devise a new partitioning approach optimized for out-of-core partitioning.

4.2 Shared: High-Throughput Partitioning

We design our *shared software write-combining (Shared)* algorithm for space-efficiency and perfect coalescing. We first provide an overview of Shared, and then examine the buffer and flush phases in detail. Finally, we discuss how Shared achieves our design goals.

Description. In Figure 8, we show the execution flow of our algorithm in seven steps. On a high level, Steps 1–3 make up the

[1] According to Nvidia, this information is currently not publicly available.

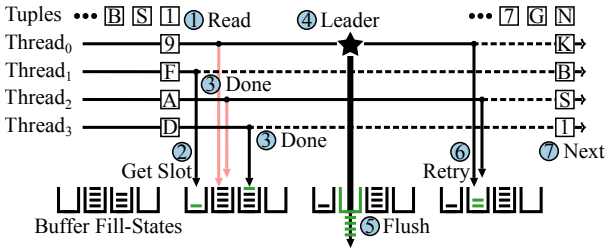


Figure 8: A thread block shares scratchpad buffers, and flushing is coalesced. Shown is one warp with four threads.

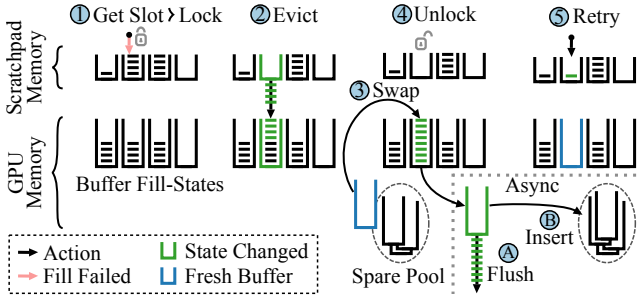


Figure 9: Buffer tuples in a two-level SWWC hierarchy for high fanouts. The 2nd level provides space for more buffers.

fill phase, and Steps 4–6 constitute the *flush phase*. Step 7 begins a new fill phase. Before execution begins, the input is divided into equally-sized chunks which are assigned to thread blocks.

Fill Phase. Execution proceeds in warps. In Step ①, each thread reads a tuple into a register and hashes the key. Then (②), each thread tries to acquire a free slot in the buffer indicated by the hash. Threads acquire slots atomically, as the buffers are shared among all warps. If a thread successfully acquires an empty slot (⟶) in Step ③, the thread stores its tuple into the buffer and marks itself "done". If all threads in a warp are "done", the warp proceeds to the next fill phase. Else, if any thread encountered a full buffer (⟶), the warp proceeds to the flush phase.

Flush Phase. The flush phase begins with Step ④. All active threads (i.e., not "done") of the warp participate in a leader ballot, and elect a thread as the warp leader. We define the first invalid slot (i.e., the buffer size) as a lock on the buffer. Thus, the full buffers are locked since Step two. All active threads except the leader immediately release their lock to enable parallel flushes by other warps. Next (⑤), the warp flushes the leader's buffer. Then, in Step ⑥, the active threads retry acquiring a slot. If at least one thread fails to acquire a slot, the warp repeats the flush phase until all threads have buffered their tuple and are marked "done". Finally, all threads start a new fill phase in Step ⑦.

Design Discussion. In our design, two aspects are important to efficiently share buffers and perfectly coalesce writes. First, filling the buffers is thread-safe but lock-free. Only flushing a buffer requires a lock, which we assign to a warp instead of spinning on the lock. Second, each flush is a multiple of the memory transaction size and also aligned to the transaction size. This ensures that optimally-sized writes are not split into two memory transactions.

4.3 Hierarchical: High-Fanout Partitioning

To reduce expensive GPU TLB misses (see Section 3.4.2), we introduce a new *hierarchical shared software write-combine (Hierarchical)* algorithm. Hierarchical extends the SWWC buffers in scratchpad memory with a second-level cache in GPU memory. By adding buffer capacity, Hierarchical incurs less TLB misses when writing to CPU memory, and we are thus able to increase the fanout.

Description. We derive the Hierarchical algorithm from Shared by extending the flush phase into a two-level hierarchy in Figure 9. The fill phase remains unchanged. The new flush phase consists of seven steps, that are executed by a warp.

L1 Eviction. The flush begins when the warp encounters a full buffer and obtains a lock on that buffer in Step ①. The lock is enforced by the fill-state counter when the buffer is full. In Step ②, the warp evicts the all tuples from the L1 buffer to its corresponding L2 buffer. If free space remains in the L2 buffer, the warp proceeds to the next fill phase after the eviction completes. Otherwise, if the buffer is full, the warp transitions to the L2 flush.

L2 Flush. The warp flushes the L2 buffer asynchronously to the execution of other warps as follows. In Step ③, the warp first swaps the full buffer with an empty buffer from a spare buffer pool. The swap is non-blocking, as the spare pool contains one spare buffer per warp (i.e., *double-buffering*). Then, the warp releases its lock on the buffer in Step ④. This allows other warps to fill the fresh buffer in parallel to the flush. Thus, the next two steps occur asynchronously to the main control flow. In Step Ⓐ, the warp flushes the full buffer's contents to CPU memory, and inserts the emptied buffer into the spare pool in Step Ⓑ. Finally, the warp proceeds to a new fill phase (⑤).

Design Discussion. A key aspect of Hierarchical is that L2 buffers are flushed asynchronously. This shortens the critical section, as we move the high-latency writes to CPU memory outside of the lock. Crucially, releasing only the L1 buffer is not enough. Instead, the L2 buffer must also be released via double-buffering. Inside the critical section, the buffer swap consists of a pointer update followed by a scratchpad memory fence and has a low overhead.

Overall, our Hierarchical algorithm enables us to efficiently partition large, out-of-core data in CPU memory with a high fanout.

5 SCALING THE STATE OF A GPU JOIN

Join algorithms are all limited by the GPU memory capacity and the interconnect bandwidth. For example, no-partitioning joins have poor data locality when the hash table spills to CPU memory, whereas partitioned joins are bandwidth-intensive due to their multiple data passes. The challenge is to achieve good data locality while at the same time reducing interconnect transfers.

In this section, we introduce our *Triton join algorithm*, which is based on the hybrid hash join [35] and aims to balance these two constraints. We optimize our Triton join for GPUs by performing multi-pass radix partitioning [84] (Section 5.1), overlapping transfer and compute (Section 5.2), and a new caching scheme for in-memory data (Section 5.3). By using the GPU to partition data with our Hierarchical algorithm and caching a working set in GPU memory, the Triton join puts into practice our GPU-partitioned join strategy that we describe in Section 3.3.

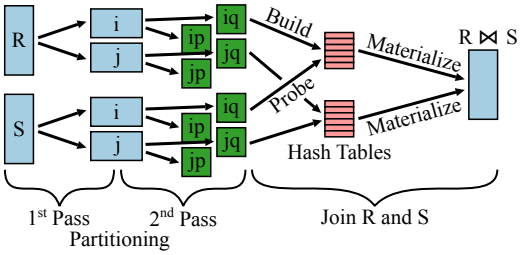


Figure 10: The Triton join is based on a parallel radix-partitioned hash join with three stages.

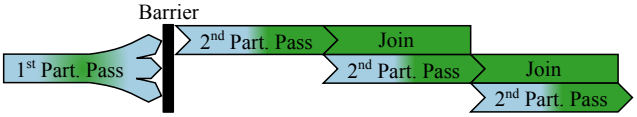


Figure 11: In the Triton join, the 2nd partitioning pass and the join are overlapped to optimize interconnect utilization.

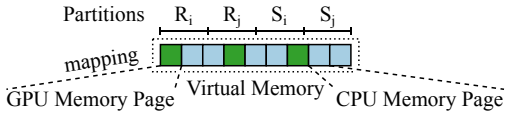


Figure 12: State is cached in GPU memory pages that are interleaved with CPU memory pages into a contiguous array.

5.1 The Triton Join Algorithm

The Triton join algorithm joins an inner relation R and an outer relation S using an equality predicate (i.e., an equi-join). We define the cardinality of R to be smaller or equal to the cardinality of S. We explicitly make no assumptions about the data volume $|R|$ and $|S|$, apart from that the system has enough total memory capacity to store both relations; either relation may be smaller or larger than the GPU memory capacity C.

We illustrate our Triton join algorithm in Figure 10. The algorithm consists of three stages:

1st Pass. The first pass radix-partitions R and S by the lower B_1 bits of the hashed join key. We choose B_1 such that two corresponding partition pairs of R and S fit into, e.g., half of the GPU memory, i.e., $|R_i| + |S_i| + |R_j| + |S_j| < \frac{C}{2}$. For example, 1 TiB of data requires $B_1 = 9$ radix bits to store each partition into a 2 GiB memory buffer. Two pairs, i and j, are necessary to pipeline the next algorithm stages. The first partitioning pass uses only a part of the GPU memory's capacity, e.g., $\frac{C}{2}$ to leave space for the results of the 2nd partitioning pass. The partitioning is executed in parallel on the GPU. At the end of this stage, all threads wait at a barrier before the join continues to the second partitioning pass.

2nd Pass. The second pass partitions each R_i and S_i partition by their next higher radix bits. Our choice of B_2 ensures that the resulting R_{ip} partitions fit into the scratchpad memory. For example, a 2 GiB partition requires $B_2 = 15$ radix bits, given a 64 KiB scratchpad. Optionally, the second pass processes only a subset of B_2, and a third pass handles the remainder. The second pass reads data from CPU memory and writes its results to GPU memory. Thus, the third pass and the join phase operate within GPU memory.

Join R and S. The join phase processes each R_{ip} and S_{ip} pair together. The join first builds a hash table in scratchpad memory with R_{ip}. Then, the join probes the hash table with S_{ip}. The join result is written to CPU memory, as, in the general case, the results are larger than the GPU memory capacity. The join requires only a single data pass to materialize results by using a linear allocator [39]. Alternatively, each thread aggregates values inside a register, and the total result of all threads is computed by, e.g., an atomic addition.

5.2 Overlapping Transfer and Compute

Pipeline parallelism is an integral part of our Triton join, as pipelining hides the data transfer time. In the partitioning stages, the GPU pulls data from pageable CPU memory on-demand using the cache-coherence [81]. This mechanism enables the hardware to transfer data implicitly and in parallel to computations. However, the Triton join requires multiple kernels to overlap with the data transfer (i.e., the second partitioning pass and the join). Multiple kernels can be overlapped with explicit transfers (e.g., `cudaMemcpyAsync`), but this would require pinned memory. Instead, we devise a new solution based on *concurrent kernel execution* [2, 95].

Concurrent kernel execution enables task parallelism on GPUs by running kernels on different SMs, and serves to increase GPU resource utilization [106]. In our Triton join, we configure each pipeline stage to occupy half of the available SMs and schedule the stages on multiple CUDA streams as shown in Figure 11. The GPU then executes the kernels in parallel. Thus, the transfer in the partitioning stage overlaps with the computation in the join stage.

5.3 Caching the Working Set in GPU Memory

We transform the partitioned hash join into a hybrid hash join by caching part of the state in GPU memory. Caching state reduces data transfers for small data sets, while providing robustness against performance cliffs when scaling the data size. However, achieving these benefits requires us to consider how caching impacts transfers.

The Triton join keeps the interconnect busy by distributing the cache space evenly over the intermediate state. We implement the cache by allocating pages that are physically in GPU and CPU memory, and then mapping these pages into a contiguous array in virtual memory, which we illustrate in Figure 12. The pages are interleaved in intervals in proportion to the physical allocation sizes, e.g., one GPU page after every two CPU pages. During execution, the GPU accesses multiple pages in parallel, and consistently utilizes the interconnect due to the evenly spaced CPU memory pages.

This is different than the standard hybrid hash join [35], which only caches the hash table of the first partition R_0. After partitioning the data, the hybrid hash join directly joins the partitions, e.g., $R_0 \bowtie S_0$. In contrast, the Triton join performs multiple partitioning passes. Hypothetically, if the Triton join were to cache R_0 and S_0 to speed up the second partitioning pass, the interconnect would be idle while the GPU partitions and joins R_0 and S_0 in GPU memory. Consequently, caching would reduce the transfer-compute overlap and leave performance on the table.

6 EVALUATION

In this section, we evaluate how well our Triton join scales to large data volumes. We describe our experiment setup and configuration in Section 6.1, and then present our results in Section 6.2.

6.1 Setup and Configuration

We first detail our evaluation environment and methodology. Next, we give an overview of the data sets used in our evaluation. Finally, we outline our experiments.

Environment. We conduct our measurements on an IBM AC922 Power System 8335-GTH. The system consists of two IBM POWER9 ("Monza") CPUs and two Nvidia Tesla V100-SXM2 ("Volta") GPUs. Each GPU is connected to one CPU via NVLink 2.0. For PCI-e 3.0 measurements, we use an Nvidia V100-PCIE GPU. Each CPU has 16 cores clocked at 3.8 GHz, that support 4-way SMT and 128-bit VSX SIMD instructions. Each GPU consists of 80 SMs running at 1.53 GHz. The system contains 128 GiB of CPU memory per socket, and each GPU has 16 GiB of GPU memory. The machine runs Ubuntu 18.04 with Linux 5.0.0-25. Our experiments are implemented in C++ and CUDA. We compile our code with GCC 8.4.0 and CUDA 10.2 with the flags: "-O3 -mcpu=native -mtune=native".

Methodology. We measure the join throughput in billions of tuples per second (*G tuples/s*). As in recent works [81, 130, 134], we define the join throughput as the total input cardinality divided by the total runtime (i.e., $\frac{|R|+|S|}{\text{runtime}}$). We report the mean and standard error over 10 runs for all experiments. We note that our measurements are stable with a standard error below 5%.

Workloads. We specify our default workload similar to related works [12, 72, 81, 134]. We use two base relations, R and S, each consisting of 16-byte `<key,record-id>` tuples. We scale their cardinalities to $|R| = |S| \in \{128, 512, 2048\}$ million tuples (*M tuples*) each. R contains primary keys, and S references the primary keys of R. We randomly shuffle the unique primary keys, generate the foreign keys following a uniform random distribution in the range $s \in [1, |R|]$, and fill the `record-id`s with random values. We store the relations in a column-oriented layout. In summary, we define in-GPU and out-of-core scenarios with up to 61 GiB of data.

Settings. Unless mentioned otherwise, our measurements are configured with the following settings and optimizations. All base relations are stored in pageable CPU memory (i.e., non-pinned). We allocate memory as 2 MiB huge pages [12, 130, 131] on the NUMA node closest to the GPU, and preallocate the pages at boot time [91] to avoid page fragmentation. The GPU directly accesses CPU memory using cache-coherence [81]. We use our Hierarchical partitioning algorithm with 6–10 radix bits for the first pass, and our Shared variant with 9 radix bits for the second pass. For the Triton and radix joins, we use a bucket-chaining hash table [47, 134] with 2048 entries [134]. On the GPU, we store the hash table in the scratchpad cache. For the no-partitioning join, we configure a linear probing scheme with a 50% load factor [70, 120, 123]. In both hashing schemes we use a multiply-shift hash function [36, 120].

Baselines. We measure a radix-partitioned, multi-core hash join implementation [17]. We port all optimizations used by Balkesen et al. [12] to the POWER ISA as described below (our own implementation adds SIMD loads). We extend the code with an array join [130] (i.e., perfect hashing), and partition with 12–14 radix bits in a single pass.

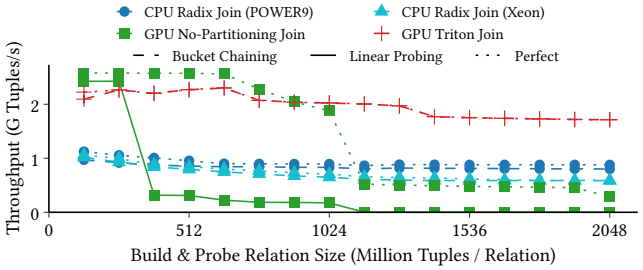


Figure 13: Scaling the build-side relation.

We optimize our CPU implementation (shown in Section 3) for the POWER9 architecture. We tune memory reads with SIMD instructions and by disabling stride-N prefetching (`DSCR = 0`) [54], as we observed that stride-N prefetching reduces sequential bandwidth. Note that sequential prefetching is enabled. We optimize the SWWC flush using SIMD stores to write 128-byte cachelines. We note that in contrast to x86_64, the POWER ISA does not support non-temporal stores that bypass the cache [37, 54]. We tested streaming store hints (`dcbtst` and `dcbz`), but these provided no speedup. In our prefix sum, each SIMD lane builds a private histogram to avoid read-after-write hazards [52]. We tune the SMT setting (16, 32, or 64 threads) for each data point.

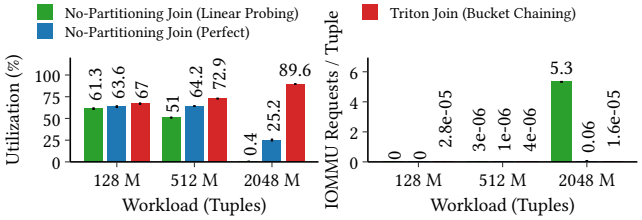
Experiments. We conduct twelve experiments. First, we evaluate how our Triton join speeds-up join throughput compared to a GPU no-partitioning join and a CPU radix-partitioned join. We then explain why the Triton join outperforms no-partitioning joins. After that, we profile the Triton join to account where time goes. As the partitioning phase has a large performance impact, we show how it is affected by the processor type (CPU vs. GPU). We analyze the GPU partitioning algorithms in-depth, and then measure the speedup gained by caching. Next, we explore computing the prefix sum on the CPU vs. the GPU. Furthermore, we analyze build-to-probe ratios and wide tuples. Finally, we investigate power efficiency and how future hardware might affect the Triton join.

6.2 Experiments

We conduct our experimental evaluation in this section.

6.2.1 Scaling the Triton Join vs. Baselines. In Figure 13, we scale the base relations from 128–2048 million tuples per relation. The relations have the same size, and consist of 16-byte tuples. The total data size is thus 3.8–61 GiB, and is up to 122 GiB large when considering the partitioned copy. This is close to the CPU memory capacity of one 128 GiB NUMA node. We compare the throughput of the Triton join to state-of-the-art join strategies on a CPU and a GPU. In addition to the IBM POWER9, we include a Intel Xeon Gold 6126 ("Skylake-SP") with 12 cores at 2.6 GHz. Note that Figure 1 is a simplified version of this experiment with only perfect hashing. Next, we compare the baselines.

CPU Radix Join. The performance of the POWER9 baseline declines by 22% from 1.1 G tuples/s to 0.9 G tuples/s, due to increasing the fanout from 2^{12} to 2^{14}. Perfect hashing is 6–16% faster



(a) Interconnect Utilization. (b) GPU TLB Misses

Figure 14: Interconnect usage of join algorithms.

than bucket chaining. In contrast, the Xeon is slower at 1.0–0.6 G tuples/s. Above 1408 M tuples, the SWWC buffers exceed the Xeon's 1.25 MiB L3 cache capacity (the POWER9 has 5 MiB/core). Thus, the Xeon switches to two-pass partitioning and a 2^{18} fanout.

GPU No-partitioning Join. The GPU baseline using perfect hashing achieves 2.5 G tuples/s up to a relation size of 640 M tuples. For larger relations, the throughput decreases to 0.5 G tuples/s for sizes above 1024 M tuples. This performance degradation occurs due to exceeding the GPU memory capacity. In contrast, linear probing reaches only 1.1 M tuples/s for large inputs due to exceeding the GPU TLB range, which we analyze in detail in Section 6.2.2. As a result, perfect hashing is up to 400× faster than linear probing.

GPU Triton Join. The Triton join performs within 85% of the GPU baseline for relations up to 896 M tuples. Then, the Triton join gracefully degrades from 2.3 to 1.7 G tuples/s. It retains 74% of its peak throughput for 2048 M tuples of data. Thus, the Triton join is 1.9–2.6× faster than the POWER9 baseline, and up to 3.9× faster than the GPU baseline with perfect hashing. The performance of bucket chaining remains within 0–2% of perfect hashing.

Summary. We draw three conclusions. First, a no-partitioning join does not scale well on GPUs with fast interconnects. Second, the hashing scheme has a large impact on the no-partitioning join, but only a small impact on the partitioned joins. Third, in all cases, our Triton join outperforms the baselines beyond 1024 M tuples.

6.2.2 Why the Triton Join Outperforms No-partitioning Joins. To better understand the join performance, we analyze the interconnect utilization and GPU TLB misses using hardware performance counters in Figure 14. We calculate the utilization as the measured bandwidth divided by the theoretical limit. We measure the bandwidth of CPU to GPU transfers including protocol overhead, for which the theoretical limit is 75 GB/s. We use a GPU prefix sum to obtain a full GPU profile. In addition, we count GPU TLB misses as the number of address translation requests received by the CPU's IOMMU [21, 55]. Note that GPU vendors do not expose GPU TLB hardware performance counters [8, 99, 138].

Interconnect Utilization. With an increasing data size, the Triton join caches a smaller proportion of the data in GPU memory. This increases interconnect utilization, as the join phase reads data from CPU memory more often. Closer inspection shows that the prefix sum and partitioning phases are at 90–100% utilization, but the join phase varies between 9–78%. In contrast, the no-partitioning join utilizes the interconnect at up to 63.6% for hash tables in GPU memory, but drops to 25.2% for out-of-core hash tables. With linear probing, utilization drops further to 0.4%.

GPU TLB Misses. GPU TLB misses are the main reason why the no-partitioning join with linear probing has a low interconnect utilization. The 50% load factor doubles the hash table size of linear probing compared to perfect hashing (64 GiB vs. 30.5 GiB for 2048 M tuples), and is rounded up to a power of two. Thus, the hash table exceeds the GPU TLB range of 32 GiB by 2× (see Section 3.4.2). As a result, the GPU issues a translation request to the IOMMU on nearly every memory access, i.e., 5.3 requests per tuple. In contrast, the Triton join issues an IOMMU request once per 10^5 tuples.

Overall, partitioning is effective at reducing TLB misses. Spilling leads to intensive interconnect utilization. However, caching and interconnect utilization are challenging to balance.

6.2.3 Time Accounting. As not all phases of the Triton join are interconnect bound, we account where time goes in Figure 15. We break down the execution time per kernel, and profile each kernel to find out whether the GPU is executing (instruction issued) or stalling (everything else). We configure a GPU prefix sum instead of a CPU prefix sum to obtain a full GPU profile.

Time Breakdown. Most of the time is spent in the first partitioning pass, which always read data from CPU memory. Partitioning (Part 1) takes 43.8–47.2% of the total time, and the prefix sum (PS 1) takes 18.9–23.4%. In contrast, the join phase reads data from GPU memory unless data is spilled to CPU memory. In our implementation, the join phase consists of four kernels: a prefix sum (PS 2) and a partitioner (Part 2) for the second pass, a join task scheduler (Sched), and the join (Join). Spilling increases the time spent in the prefix sum, as it copies the data into GPU memory to avoid redundant transfers by subsequent kernels.

Profiling. Both prefix sum passes and the first partitioning pass are mostly interconnect bound due to memory dependencies. In contrast, the second partitioning pass is mostly compute bound due to issuing instructions, as it runs in GPU memory. Only first partitioning pass and second prefix sum pass change with the workload, due to spilling and reloading data. Counterintuitively, large data sizes reduce memory stalls for partitioning, as high fanouts cause additional work.

Our take away is that interconnect bandwidth limits the partitioning phase, but compute power limits the join phase. As bandwidth outweighs computation, we focus our optimization efforts on the interconnect in the following sections.

6.2.4 CPU-Partitioned vs. GPU-Partitioned Join. We evaluate the impact of the processor used for partitioning on the end-to-end join in Figure 16(a). Following that, we investigate the partitioning phase in Figure 16(b). For a fair comparison, we reimplement the strategy of Sioulas et al. [134] and optimize it for the POWER9 and NVLink 2.0 (see Section 6.1). The join overlaps the transfer and second partitioning pass over R with the first pass over S, and caches its working set in GPU memory. We compare this CPU-partitioned radix join to our Triton join, that is GPU-partitioned. We run the default workloads, and plot the throughput in G tuples/s for the join and GiB/s for the partitioning.

End-to-End Join. The CPU-partitioned join reaches a throughput of 1.3–1.8 G tuples/s. The 128 M tuple workload has a 38% higher throughput than the 2048 M workload, due to caching the working set. In contrast, the Triton join achieves a 1.2–1.3× speedup.

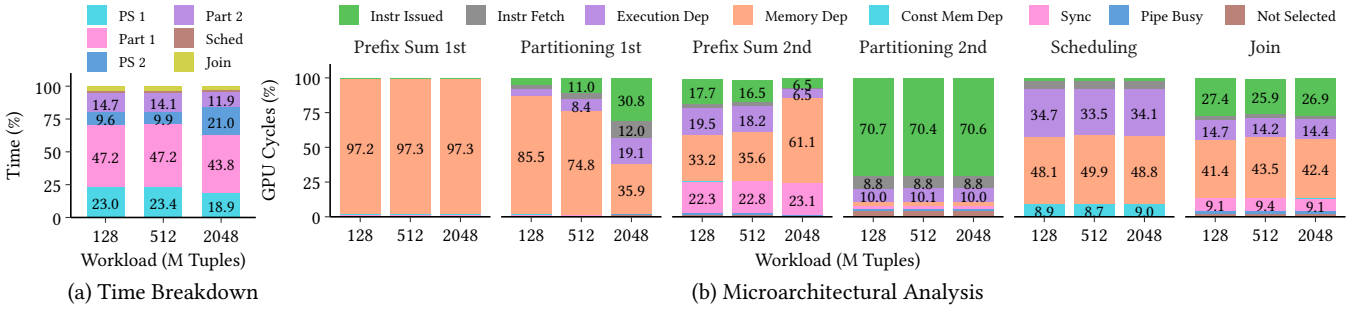


Figure 15: Time breakdown of the Triton join.

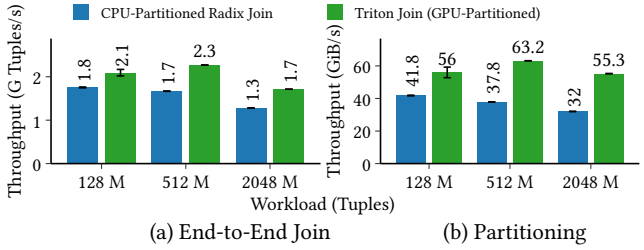


Figure 16: Partitioning data using the CPU vs. the GPU.

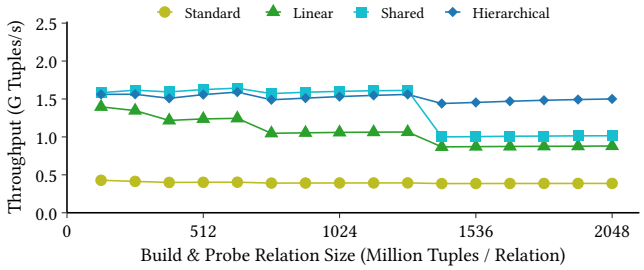


Figure 17: Effect of partitioning algorithms on a radix join.

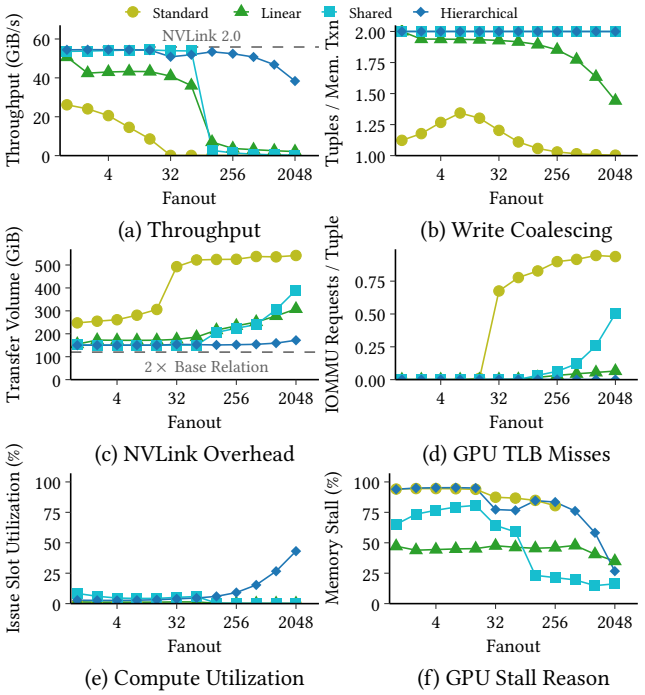


Figure 18: Profiling state-of-the-art partitioning algorithms.

Partitioning. A closer inspection reveals why the Triton join is faster. First, the GPU partitions data 1.5–1.7× faster than the CPU. Second, the Triton join caches intermediate results in GPU memory, leading to a lower transfer volume. In contrast, the CPU-partitioned join first has to write results to CPU memory and then read them again for the transfer to the GPU, which consumes memory bandwidth. However, the CPU-partitioned join overlaps the partitioning of the outer relation and the transfer of the inner relation. Thus, its join pipeline is 3–13% faster than that of the Triton join.

Overall, our Triton join is faster than the CPU-partitioned join due to partitioning data efficiently on the GPU, and the caching optimizations that this design enables.

6.2.5 Partitioning Algorithms. In Figure 17, we evaluate the impact of the partitioning algorithm on the join. We compare our Shared and Hierarchical to the Linear and *Standard* radix partitioning algorithms. We vary the algorithm used in the first pass and measure the end-to-end join throughput. We scale the base relations from 128 to 2048 M tuples. We disable caching to eliminate side-effects.

Observations. Our Shared algorithm achieves a throughput of 1.5–1.6 G tuples/s up to a size of 1280 M tuples. At this threshold, the flush granularity drops below 128 bytes due to the high fanout. For larger relation sizes, the throughput of Shared reduces to 0.9–1 G tuples/s. In contrast, our Hierarchical variant performs between 1.4–1.5 G tuples/s over the whole range, and degrades gracefully. Thus, Hierarchical achieves a speedup of 1.1–1.9× and 3.6–4× over the Linear and Standard algorithms, respectively.

Overall, the choice of the partitioning algorithm is important for scaling a GPU join. Most notably, our Hierarchical algorithm improves the scaling to large data sizes.

6.2.6 Why Hierarchical Outperforms the State-of-the-Art. To reveal the superior the partitioning throughput, we investigate all partitioning algorithms with hardware performance counters. We use 60 GiB of data, which are sufficiently large to incur TLB misses (see

Section 3.4.2). The GPU reads 16-byte tuples from CPU memory, and writes the results back to CPU memory.

Throughput. In Figure 18(a), we measure the partitioning throughput in isolation while increasing the fanout. We highlight three aspects. First, the Linear algorithm never achieves the bidirectional interconnect bandwidth of 55.9 GiB/s. It reaches 50.7 GiB/s for one partition (i.e., a memcopy), but then drops to 42.4 GiB/s for a fanout of 2. Second, Shared partitions at 54 GiB/s, but does not scale beyond a fanout of 64. In contrast, our Hierarchical algorithm achieves 38.3 GiB/s even at a fanout of 2048.

Write Coalescing. To reveal the reasons for the performance, we begin by recording the tuples per memory transaction in Figure 18(b). We find that Linear only partially coalesces writes. The reason is that sorting tuples by partition usually does not result in batches of exactly 128 bytes. High fanouts increase this effect as the tuples cached per partition decrease. In contrast, both of our algorithms perfectly coalesce writes by design.

NVLink Overhead. Ineffective coalescing leads to the high physical transfer volumes in Figure 18(c). This overhead results from the interconnect packet header attached to each payload, and is higher for small payload sizes. In the case of Linear, interconnect overhead accounts for up to 156% of the transfer volume. In contrast, our Hierarchical algorithm remains below 43%.

GPU TLB Misses. In Figure 18(d), we unmask the performance barrier at a fanout of 64 by measuring the IOMMU requests. Going from 64 to 128 partitions causes the TLB miss rate of Shared to increase by 33×, i.e., a miss on every second flush. In contrast, at a fanout of 2048 Hierarchical achieves a 1436×, 100×, and 771× lower miss rate compared to Standard, Linear, and Shared, respectively.

Compute Utilization. We inspect if computation limits throughput by reporting the "percentage of issue slots that issued at least one instruction" [99] in Figure 18(e). Typically, utilization remains below 5%. Only Hierarchical utilizes up to 43% of the GPU with high fanouts. The trend starts when the buffer size drops to 16 tuples at a fanout of 256, and flushing no longer occupies a full warp.

GPU Stall Reason. In Figure 18(f), we reveal why compute utilization is low. Shared and Hierarchical stall on memory dependencies 65–90% of the time. In contrast, Linear additionally stalls on synchronization and pipeline busy [99]. TLB misses manifest themselves as instruction latency, i.e., execution dependency and pipeline busy stalls. For Standard, the stall counters overflow for fanouts of 512–2048 due to its runtime of 10 minutes [136].

We conclude that the data access pattern and TLB miss tolerance of our Shared and Hierarchical algorithms are the main reasons they outperform the Standard and Linear approaches.

6.2.7 Caching the Working Set in GPU Memory. We explore the effect of caching on the throughput of the no-partitioning join and the Triton join in Figure 19. We scale the cache size in GPU memory from 0 to 14.9 GiB. For the no-partitioning join, we cache part of the hash table [81]. We note that the Triton join with no cache is effectively a two-pass radix join, and that a part of the GPU memory is required for the join pipeline.

GPU No-Partitioning Join. Caching the entire hash table instead of not caching anything increases throughput by 4.6–4.8× for the 128 M and 512 M workloads using perfect hashing. In contrast, caching has no effect on the 2048 M workload. The reason is the high cache miss rate of 50%. A miss rate of only 4% reduces the gain to 1.8× for 512 M with linear probing. In contrast, for 2048 M with linear probing the reason is that GPU TLB misses slow down the join (see Section 6.2.2).

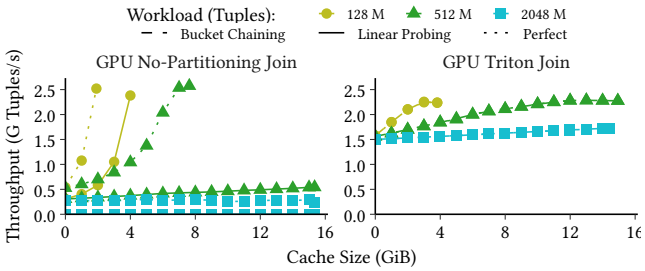


Figure 19: Scaling the GPU memory cache size.

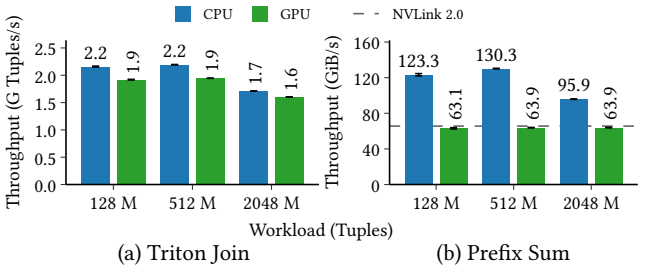


Figure 20: Prefix sum on the CPU vs. on the GPU.

GPU Triton Join. In contrast, the 128 M and 512 M workloads improve performance by 1.4×, and the 2048 M workload by 1.1×. However, the 128 M workload slows down by 1.5% when the whole working set is cached, instead of only 79% of the working set. This is because the GPU memory and interconnect together provide more bandwidth than GPU memory alone [1, 112].

Our take-away is that the Triton join robustly scales with the cache size, and avoids sharp performance cliffs caused by the TLB range and the GPU memory capacity.

6.2.8 CPU vs. GPU Prefix Sum. We determine which processor computes the prefix sum faster. We first assess the effect of the prefix sum on the end-to-end join (Figure 20(a)), and then measure the prefix sum throughput achieved by the CPU and GPU (Figure 20(b)). We run the experiment using our Triton join on the default working sets with 128, 512, and 2048 M tuples. We show the prefix sum throughput in GiB/s to enable a comparison with the memory bandwidth. We highlight that the prefix sum reads a single column per relation, due to the columnar layout.

Triton Join. We observe that when using the CPU, the Triton join achieves a throughput of 2.2 G tuples/s for the 128 M and 512 M workloads, and 1.6 G tuples/s for the 2048 M workload. These results are 1.1× faster than when computing the prefix sum on the GPU.

Prefix Sum. The CPU achieves up to 129.6 GiB/s, and is able to nearly saturate the CPU memory bandwidth. For the 2048 M tuples workload, the throughput decreases to 96 GiB/s. In contrast, the throughput of the GPU is constant at 63 GiB/s. The reason is that

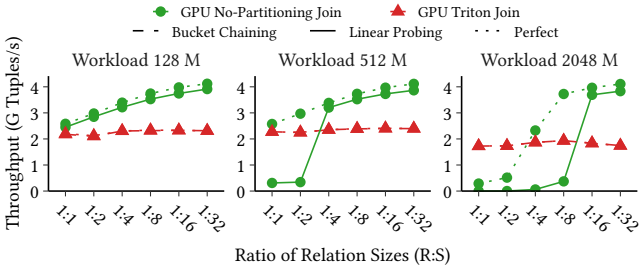


Figure 21: Varying build-to-probe ratios with the Triton join.

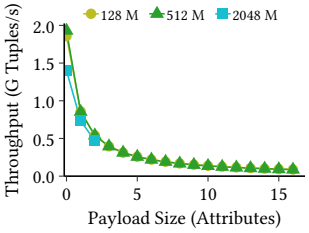


Figure 22: Scaling the number of payload attributes.

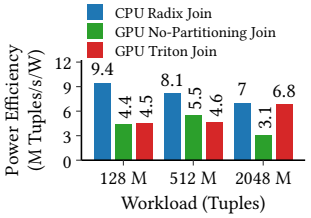


Figure 23: Performance/Watt of CPU vs. GPU.

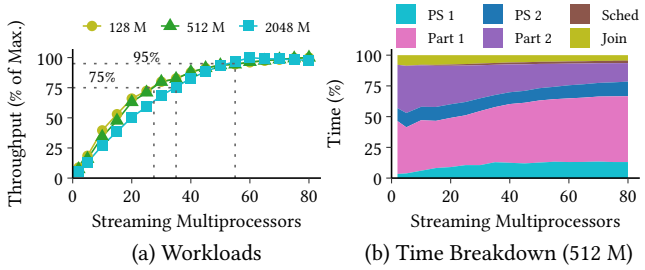


Figure 24: Compute power required for high throughput.

reads are unidirectional transfers, and thus the GPU is constrained by the interconnect bandwidth.

Overall, the CPU is able to sequentially scan data faster than the GPU, and thus computes the prefix sum 1.6–2.2× faster. However, the prefix sum has a small impact on the overall join throughput.

6.2.9 Build-to-probe Ratios. In Figure 21, we measure the throughput of the Triton join for different build-to-probe ratios. For each workload, we scale the ratio from 1:1 to 1:32 while keeping the data constant at 61 GiB. For example, for the 2048 M workload 1:1 means 2048:2048 M tuples, and 1:32 means 124:3972 M tuples.

Observations. The no-partitioning join is subject to two effects. First, the GPU memory capacity causes an abrupt performance cliff. The extreme case is linear probing, for which a 1:32 ratio is 3414× faster than 1:1 in the 2048 M workload. Second, reducing the build size within GPU memory causes a 60% speedup. Dissecting the perfect hashing variant shows that the probe throughput is 4.3 G tuples/s, whereas the build throughput is only 1.8 G tuples/s. In a deeper investigation, we find that random GPU memory reads are 3.2–6× faster than writes. In contrast, the throughput of the Triton join remains stable between 1.66-1.88 G tuples/s. This increase results from reducing the fanout from 1024 to 64 partitions, which increases partitioning throughput.

We conclude that the Triton join is insensitive to the build-to-probe ratio, due to partitioning the large outer relation. Thus, a no-partitioning join should be preferred for high ratios.

6.2.10 Tuple Width. The relation size is determined both by the number and width of tuples. In Figure 22, we investigate how materializing wide tuples affects the Triton join. Instead of reading two attributes, we partition only the join key and generate row IDs on-the-fly in the first pass. Thus, the join results in a join index, with which we materialize and aggregate the out-of-core, 8-byte payload attributes of the outer relation.

Observations. At 2 G tuples/s and 1.5 G tuples/s, constructing a join index (i.e., no payload) achieves a similar throughput as our default setup, which early-materializes one payload attribute. In contrast, late materialization incurs a random CPU memory access per attribute, which causes performance to degrade to 86–88 M tuples/s for 16 payloads. The 2048 M workload stops at 2 payloads due to reaching the CPU memory capacity.

In conclusion, partitioning leads to expensive random accesses during late materialization. Our results indicate that materializing wide, out-of-core tuples requires further investigation.

6.2.11 Power Efficiency. In Figure 23, we compare the performance per Watt of the CPU and the GPU. We measure the energy consumed by the system, and calculate the normalized throughput per power unit averaged over 50 joins. We subtract the idle power of both GPUs for the CPU radix join ($2 \times 32W$) to simulate a CPU-only system. At idle the AC922 system consumes 290 W. The joins are configured with perfect hashing.

Observations. The CPU turns out to be the most power-efficient processor at 7–9.4 M tuples/s/W. However, the GPU joins are not competitive due to the CPU's high idle power of 58–62 W. Under load the GPU consumes 62–80 W, while the CPU consumes 178–206 W. We must consider that the GPU use the CPU's I/O facilities for interconnect transfers, which takes 10-11 W.

Overall, the CPU join is more power-efficient than our Triton join, as the GPU is hosted by a CPU.

6.2.12 Compute Power Scaling. We explore how future hardware might affect the Triton join by scaling number of streaming multiprocessors in Figure 24. We measure the throughput as a percentage of the maximum, and explain the scaling behavior by examining where the join spends time in the 512 M tuples workload.

Workloads. We observe that 28 SMs suffice to achieve 75% of the peak throughput for the 128 M and 512 M workloads. The 95% mark is passed for all our workloads with 55 SMs.

Time Breakdown. The Triton join scales quickly at first, as the first and the second partitioning passes are compute bound below 25 SMs. With more than 25 SMs, profiling shows that the first pass becomes interconnect bound and stops scaling. In contrast, the second pass remains compute bound and continues to scale, but with diminishing returns. As a result, the overall scaling levels out.

We conclude that the Triton join is interconnect bound. A faster interconnect would increase join throughput, whereas a faster GPU would not yield significant gains.

6.3 Discussion

In this paper, we have investigated how fast interconnects can resolve the memory capacity limitation to scale the state of a GPU join, and have gained the following key insights.

GPUs with fast interconnects scale to a large join state. Fast interconnects provide sufficient bandwidth to spill large state to CPU memory. A 2× speedup over a strong CPU baseline is possible even when the state size exceeds the GPU memory capacity.

GPUs robustly spill state to CPU memory. We learned that partitioning and caching can be combined to gracefully degrade throughput. Thus, we are able to avoid performance cliffs caused by the TLB range and the GPU memory capacity.

GPUs are able to process tasks end-to-end. Fast interconnects obviate CPU involvement. For example, the CPU no longer must partition data or manage transfer pipelines. This enables DBMSs to efficiently use heterogeneous hardware.

Interconnect-awareness enables fast random accesses. Perfect coalescing saturates a fast interconnect. Thus, adapting the access pattern to the interconnect makes new use-cases possible.

Concurrent kernel execution is a versatile replacement for DMA copy engines. In addition to overlapping computation and transfers from pageable memory, kernels are able to directly compute or reshape the data. Thus, concurrent kernel execution helps to reduce GPU memory traffic and improve data access patterns.

Interconnect bandwidth is no longer the main bottleneck. In some cases, the high interconnect bandwidth shifts the bottleneck to other resources, such as random access bandwidth, TLB misses, and computation. Optimization becomes challenging, as multiple constraints can simultaneously affect different parts of the program.

Summary. Fast interconnects enable GPUs to cover a broader spectrum of database use-cases, but we require new algorithms to fully exploit the performance potential of fast interconnects.

7 RELATED WORK

In this section, we contrast our paper to related work.

Scalable Co-Processing. Recent GPU-enabled DBMSs [23, 24, 29, 30, 39, 47, 51] and machine learning frameworks [9, 82] are able to process data sets larger than GPU memory. Our work complements these systems by scaling the operator state, thus enabling large data sizes.

Relational and ML operators stream data from CPU memory to the GPU to transfer data efficiently across the interconnect [66, 70, 80, 123]. In contrast to these works, we scale operator state in addition to scaling the data size.

Join Co-Processing. Speeding up joins on co-processors has been of particular interest for database research [47–50, 66, 89, 92, 112, 143]. Recent works investigate radix-partitioned joins on GPUs [105, 125], MICs [26, 63, 114], and FPGAs [25, 45, 46, 76]. However, these approaches limit the join state to the co-processor's on-board memory, or assume a coupled architecture in which the co-processor has direct CPU memory access. In contrast, our Triton join handles large state on a discrete GPU.

Radix Partitioning on Co-Processors. Radix partitioning has been investigated on GPUs, MICs, and FPGAs. Early GPU works suggest a binary divide-and-conquer approach [127, 128], that requires a data pass per radix bit. More recently, GPUs with atomic additions enable a single-pass approach that sorts data in scratch-pad memory [125, 140]. In contrast, our Shared algorithm extends software write-combining [128] to fully coalesce writes on GPUs.

SWWC partitioning has been ported to MICs by SIMD vectorization [115–117]. Our Shared is structurally similar to vectorized SWWC. However, in contrast to SIMD partitioning, Shared saves cache space by sharing buffers among warps. In analogous terms, in our design, SMT threads share buffers in the L1 cache, in addition to SIMD vectorization. To the best of our knowledge, no prior work considers such a design on any processor architecture.

On FPGAs, write-combining can be implemented in hardware instead of in software [67, 141]. However, previous studies have been limited by slow interconnects that incur a data transfer bottleneck.

End-to-End Join Queries. Join state compression [16, 19, 40, 77], filtering [13, 44, 129] and pipelining [14, 147] the outer relation, and efficient tuple materialization [85, 116, 117, 146] have been proposed to speed-up joins. These optimizations complement our work and remain open challenges for GPUs with fast interconnects.

Transfer Bottleneck. Previous works consider scaling operator state for joins [44, 113, 134], sorting [140], and the primitives underlying these operators [42]. However, these works assume that PCI-e causes a transfer bottleneck. In contrast, we take advantage of fast interconnects by proposing a new approach that eliminates CPU pre-processing steps.

Fast Interconnects. GPUs with NVLink have been explored to speed up query processing. Recent works investigate the data transfer bottleneck [81], lazy transfers and scan sharing for HTAP DBMSs [119], multi-GPU joins [40, 107, 124], CSV loading [74], and sorting [83]. FPGAs with OpenCAPI have been exploited to scale the outer relation of a join [68] and data loading [108, 109]. In contrast, we show that by carefully designing algorithms for fast interconnects, GPUs efficiently accelerate joins with a large state.

8 CONCLUSION

Fast interconnects are not a silver bullet for large-scale hash joins. Our analysis of NVLink 2.0 reveals that interconnect overhead and TLB misses reduce performance. We propose our Triton join to overcome these challenges. Our hardware insights lead to a GPU-partitioned join strategy based on a new GPU partitioning algorithm. Overall, our Triton join scales to large data volumes at up to 400× faster performance by being aware of the fast interconnect.

We provide our source code for future research at:
https://github.com/TU-Berlin-DIMA/fast-interconnects

ACKNOWLEDGMENTS

We thank Jonas Pfefferle (IBM Research), Nikolay Sakharnykh (Nvidia), and Panagiotis Sioulas (EPFL) for the insightful discussions.

This work was funded by the EU Horizon 2020 programme as E2Data (780245), the German Ministry for Education and Research as BIFOLD — "Berlin Institute for the Foundations of Learning and Data" (01IS18025A and 01IS18037A), and the German Federal Ministry for Economic Affairs and Energy as Project ExDra (01MD19002B).

REFERENCES

[1] Neha Agarwal, David W. Nellans, Mike O'Connor, Stephen W. Keckler, and Thomas F. Wenisch. 2015. Unlocking bandwidth for GPUs in CC-NUMA systems. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 354–365. https://doi.org/10.1109/HPCA.2015.7056046

[2] AMD 2015. *Asynchronous shaders: Unlocking the full potential of the GPU.* AMD. https://developer.amd.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf

[3] AMD 2017. *Radeon's next-generation Vega architecture.* AMD. https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf

[4] AMD 2020. *OpenCL optimization.* AMD. https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-optimization.html Git Revision 1f057816.

[5] AMD 2020. *OpenCL programming guide.* AMD. https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html Git Revision 611e249.

[6] AMD 2021. *AMD CDNA 2 architecture.* AMD. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf

[7] AMD 2021. *AMD EPYC 7003 series processors.* AMD. https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf LE-77202-00 02/21.

[8] AMD 2021. *GPU performance API.* AMD. https://gpuperfapi.readthedocs.io/en/latest/ Git Revision 3642849d.

[9] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P. Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. In *PPoPP*. ACM, New York, NY, USA, 173–182. https://doi.org/10.1145/2688500.2688521

[10] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU memory manager with application-transparent support for multiple page sizes. In *MICRO*. ACM, New York, NY, USA, 136–150. https://doi.org/10.1145/3123939.3123975

[11] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB* 7, 1 (2013), 85–96. https://doi.org/10.14778/2732219.2732227

[12] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[13] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To partition, or not to partition, that is the join question in a real system. In *SIGMOD*. ACM, New York, NY, USA, 168–180. https://doi.org/10.1145/3448016.3452831

[14] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *PVLDB* 8, 4 (2014), 353–364. https://doi.org/10.14778/2735496.2735499

[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation caching: Skip, don't walk (the page table). In *ISCA*. ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/1815961.1815970

[16] Claude Barthels, Gustavo Alonso, Torsten Hoefler, Timo Schneider, and Ingo Müller. 2017. Distributed join algorithms on thousands of cores. *PVLDB* 10, 5 (2017), 517–528. https://doi.org/10.14778/3055540.3055545

[17] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *SIGMOD*. ACM, New York, NY, USA, 1463–1475. https://doi.org/10.1145/2723372.2750547

[18] Trinayan Baruah, Yifan Sun, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David R. Kaeli. 2020. Valkyrie: Leveraging inter-TLB locality to enhance GPU performance. In *PACT*. ACM, New York, NY, USA, 455–466. https://doi.org/10.1145/3410463.3414639

[19] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. 2012. MCJoin: A memory-constrained join for column-store main-memory databases. In *SIGMOD*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2213836.2213851

[20] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. 2018. Scalable distributed last-level TLBs using low-latency interconnects. In *MICRO*. IEEE Computer Society, Los Alamitos, CA, USA, 271–284. https://doi.org/10.1109/MICRO.2018.00030

[21] Bartholomew Blaner, Jay G Heaslip, Robert D Herzl, and Jody B Joyner. 2020. Maintaining consistency between address translations in a data processing system. Patent No. US10599569B2, Filed Jun. 23th., 2016, Issued Mar. 24th., 2020.

[22] Bartholomew Blaner, Jody B Joyner, Ronald N Kalla, Jon K Kriegel, and Charles D Wait. 2018. Maintaining agent inclusivity within a distributed MMU. Patent App. No. US20180300256A1, Filed Apr. 13th., 2017.

[23] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust query processing in co-processor-accelerated databases. In *SIGMOD*. ACM, New York, NY, USA, 1891–1906. https://doi.org/10.1145/2882903.2882936

[24] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.* 27, 6 (2018), 797–822. https://doi.org/10.1007/s00778-018-0512-y

[25] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2020. Is FPGA useful for hash joins?. In *CIDR*. www.cidrdb.org, 1–9.

[26] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A study of main-memory hash joins on many-core processor: A case with Intel Knights Landing architecture. In *CIKM*. ACM, New York, NY, USA, 657–666. https://doi.org/10.1145/3132847.3132916

[27] Jack Choquette, Olivier Giroux, and Denis Foley. 2018. Volta: Performance and programmability. *IEEE Micro* 38, 2 (2018), 42–52. https://doi.org/10.1109/MM.2018.022071134

[28] Stavros Christodoulakis. 1984. Implications of certain assumptions in database performance evaluation. *TODS* 9, 2 (1984), 163–186. https://doi.org/10.1145/329.318578

[29] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating heterogeneous CPU–GPU parallelism in JIT compiled engines. *PVLDB* 12, 5 (2019), 544–556. https://doi.org/10.14778/3303753.3303760

[30] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious query processing in GPU-accelerated analytical engines. In *CIDR*. www.cidrdb.org, 1–9.

[31] CXL 2020. *Compute Express Link specification, Revision 2.0.* CXL. https://www.computeexpresslink.org

[32] Jack W. Davidson and Sanjay Jinturkar. 1994. Memory access coalescing: A technique for eliminating redundant memory accesses. In *PLDI*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/178243.178259

[33] James Leroy Deming, Mark Allen Mosley, and William Craig McKnight. 2017. Method for automatic page table compression. Patent No. US9569348B1, Filed Jul. 23rd., 2010, Issued Feb. 14th., 2011.

[34] James Leroy Deming, Mark Allen Mosley, William Craig McKnight, Emmett M. Kilgrariff, Steven E. Molnar, and Colyn Scott Case. 2011. Efficient memory translator with variable size cache line coverage. Patent No. US20110072235A1, Filed Oct. 8th., 2005, Issued Mar. 24th., 2011.

[35] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *SIGMOD*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/602259.602261

[36] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms* 25, 1 (1997), 19–51. https://doi.org/10.1006/jagm.1997.0873

[37] Ulrich Drepper. 2007. *Memory part 5: What programmers can do.* LWN. Retrieved Jun 23, 2021 from https://lwn.net/Articles/255364/

[38] Denis Foley and John Danskin. 2017. Ultra-performance Pascal GPU and NVLink interconnect. *IEEE Micro* 37, 2 (2017), 7–17. https://doi.org/10.1109/MM.2017.37

[39] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined query processing in coprocessor environments. In *SIGMOD*. ACM, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[40] Hao Gao and Nikolai Sakharnykh. 2021. Scaling joins to a thousand GPUs. In *ADMS*. 55–64.

[41] Mel Gorman. 2004. *Understanding the Linux virtual memory manager.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[42] Michael Gowanlock, Ben Karsin, Zane Fink, and Jordan Wright. 2019. Accelerating the unacceleratable: Hybrid CPU/GPU algorithms for memory-bound database primitives. In *DaMoN*. ACM, New York, NY, USA, 7:1–7:11. https://doi.org/10.1145/3329785.3329926

[43] Chris Gregg and Kim M. Hazelwood. 2011. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *ISPASS*. IEEE Computer Society, Los Alamitos, CA, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[44] Tim Gubner, Diego G. Tomé, Harald Lang, and Peter A. Boncz. 2019. Fluid co-processing: GPU Bloom-filters for CPU joins. In *DaMoN*. ACM, New York, NY, USA, 9:1–9:10. https://doi.org/10.1145/3329785.3329934

[45] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. 2015. FPGA-based multithreading for in-memory hash joins. In *CIDR*. www.cidrdb.org, 1–9.

[46] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh W. Asaad, and Balakrishna Iyer. 2013. Accelerating join operation for relational databases with FPGAs. In *FCCM*. IEEE Computer Society, Los Alamitos, CA, USA, 17–20. https://doi.org/10.1109/FCCM.2013.17

[47] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. *TODS* 34, 4, Article 21 (2009), 39 pages. https://doi.org/10.1145/1620585.1620588

[48] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2008. Relational joins on graphics processors. In *SIGMOD*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[49] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB* 6, 10 (2013), 889–900. https://doi.org/10.14778/2536206.2536216

[50] Jiong He, Shuhao Zhang, and Bingsheng He. 2014. In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB* 8, 4 (2014), 329–340. https://doi.org/10.14778/2735496.2735497

[51] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* 6, 9 (2013), 709–720. https://doi.org/10.14778/2536360.2536370

[52] John L. Hennessy and David A. Patterson. 2017. *Computer architecture — A quantitative approach* (6th ed.). Morgan Kaufmann, Cambridge, MA, USA.

[53] Jensen Huang. 2021. *GTC '21 keynote*. Nvidia. Retrieved May 31, 2021 from https://images.nvidia.com/gtc/keynote/GTC21-Jensen-Huang-Keynote_04.pdf

[54] IBM 2017. *POWER ISA version 3.0B*. IBM.

[55] IBM 2018. *POWER9 performance monitor unit user's guide*. IBM. Version 1.2.

[56] IBM 2018. *POWER9 processor user's manual*. IBM. Version 2.0.

[57] IBM POWER9 NPU team. 2018. Functionality and performance of NVLink with IBM POWER9 processors. *IBM Journal of Research and Development* 62, 4/5 (2018), 9:1–9:10. https://doi.org/10.1147/JRD.2018.2846978

[58] Intel. 2019. *Intel unveils new GPU architecture with high-performance computing and AI acceleration, and oneAPI software stack with unified and scalable abstraction for heterogeneous architectures*. Intel. Retrieved Jun 10, 2021 from https://newsroom.intel.com/news-releases/intel-unveils-new-gpu-architecture-optimized-for-hpc-ai-oneapi

[59] Intel 2021. *Intel Xeon Gold 6338 processor*. Intel. https://ark.intel.com/content/www/us/en/ark/products/212285/intel-xeon-gold-6338-processor-48m-cache-2-00-ghz.html

[60] Akshay Jain, Mahmoud Khairy, and Timothy G. Rogers. 2018. A quantitative evaluation of contemporary GPU simulation methodology. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2 (2018), 35:1–35:28. https://doi.org/10.1145/3224430

[61] Aamer Jaleel, Eiman Ebrahimi, and Sam Duncan. 2019. DUCATI: high-performance address translation by extending TLB reach of GPU-accelerated systems. *ACM Trans. Archit. Code Optim.* 16, 1 (2019), 6:1–6:24. https://doi.org/10.1145/3309710

[62] Joefon Jann, Paul Mackerras, John Ludden, Michael Gschwind, Wade Ouren, Stuart Jacobs, Brian F. Veale, and David Edelsohn. 2018. IBM POWER9 system software. *IBM Journal of Research and Development* 62, 4/5 (2018), 6:1–6:10. https://doi.org/10.1147/JRD.2018.2846959

[63] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB* 8, 6 (2015), 642–653. https://doi.org/10.14778/2735703.2735704

[64] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the Nvidia Turing T4 GPU via microbenchmarking. arXiv:1903.07486 [cs.DC]

[65] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the Nvidia Volta GPU architecture via microbenchmarking. arXiv:1804.06826 [cs.DC]

[66] Tim Kaldewey, Guy M. Lohman, René Müller, and Peter Benjamin Volk. 2012. GPU join processing revisited. In *DaMoN*. ACM, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[67] Kaan Kara, Jana Giceva, and Gustavo Alonso. 2017. FPGA-based data partitioning. In *SIGMOD*. ACM, New York, NY, USA, 433–445. https://doi.org/10.1145/3035918.3035946

[68] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. 2020. High bandwidth memory on FPGAs: A data analytics perspective. arXiv:2004.01635 [cs.DC]

[69] Tomas Karnagel, Tal Ben-Nun, Matthias Werner, Dirk Habich, and Wolfgang Lehner. 2017. Big data causing big (TLB) problems: Taming random memory accesses on the GPU. In *DaMoN*. ACM, New York, NY, USA, 6:1–6:10. https://doi.org/10.1145/3076113.3076115

[70] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated group-by and aggregation. In *ADMS*. 13–24.

[71] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *ISCA*. IEEE, Washington, DC, USA, 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[72] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB* 2, 2 (2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[73] Donald E. Knuth. 1981. *Seminumerical algorithms* (2nd ed.). The art of computer programming, Vol. 2. Addison-Wesley, Reading, MA, USA.

[74] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV loading using GPUs and RDMA for in-memory data processing. In *BTW (LNI, Vol. P-311)*. Gesellschaft für Informatik, Bonn, Germany, 19–38. https://doi.org/10.18420/btw2021-01

[75] Zhuohang Lai, Qiong Luo, and Xiaoying Jia. 2018. Revisiting multi-pass scatter and gather on GPUs. In *ICPP*. ACM, New York, NY, USA, 25:1–25:11. https://doi.org/10.1145/3225058.3225095

[76] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. 2022. Bandwidth-optimal relational joins on FPGAs. In *EDBT*. OpenProceedings.org, Konstanz, Germany, 1–13. https://doi.org/10.5441/002/edbt.2022.03

[77] Jae-Gil Lee, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy M. Lohman, Konstantinos Morfonios, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Vincent Kulandai Samy, Richard Sidle, Knut Stolze, and Liping Zhang. 2014. Joins on encoded and partitioned data. *PVLDB* 7, 13 (2014), 1355–1366. https://doi.org/10.14778/2733004.2733008

[78] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: A RateupDB experience of building a CPU/GPU hybrid database product. *PVLDB* 14, 12 (2021), 2999–3013. https://doi.org/10.14778/3476311.3476378

[79] Erik Lindholm, John Nickolls, Stuart F. Oberman, and John Montrym. 2008. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (2008), 39–55. https://doi.org/10.1109/MM.2008.31

[80] Clemens Lutz, Sebastian Breß, Tilmann Rabl, Steffen Zeuch, and Volker Markl. 2018. Efficient and scalable k-means on GPUs. *Datenbank-Spektrum* 18, 3 (2018), 157–169. https://doi.org/10.1007/s13222-018-0293-x

[81] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump up the volume: Processing large data on GPUs with fast interconnects. In *SIGMOD*. ACM, New York, NY, USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[82] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. 2018. In-RDBMS hardware acceleration of advanced analytics. *PVLDB* 11, 11 (2018), 1317–1331. https://doi.org/10.14778/3236187.3236188

[83] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating multi-GPU sorting with modern interconnects. In *SIGMOD*. ACM, New York, NY, USA, 15 pages. To appear.

[84] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[85] Stefan Manegold, Peter A. Boncz, and Niels Nes. 2004. Cache-conscious radix-decluster projections. In *PVLDB*. Morgan Kaufmann, St. Louis, MO, USA, 684–695. https://doi.org/10.1016/B978-012088469-8.50061-9

[86] MarketsandMarkets Research. 2018. *GPU Database Market*. MarketsandMarkets Research. Retrieved Oct 1, 2019 from https://www.marketsandmarkets.com/Market-Reports/gpu-database-market-259046335.html

[87] Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam Minh Tran. 2007. Consistent selectivity estimation via maximum entropy. *VLDB J.* 16, 1 (2007), 55–76. https://doi.org/10.1007/s00778-006-0030-1

[88] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distributed Syst.* 28, 1 (2017), 72–86. https://doi.org/10.1109/TPDS.2016.2549523

[89] Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An energy-efficient stream join for the Internet of Things. In *DaMoN*. ACM, New York, NY, USA, 8:1–8:6. https://doi.org/10.1145/3465998.3466005

[90] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. 2020. EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs. *PVLDB* 14, 2 (2020), 114–127. https://doi.org/10.14778/3425879.3425883

[91] Milan Navrátil, Laura Bailey, and Charlie Boyle. 2018. *Red Hat Enterprise Linux 7 Performance Tuning Guide*. Red Hat. Revision 10.13-59.

[92] Anh Nguyen, Masato Edahiro, and Shinpei Kato. 2018. GPU-accelerated VoltDB: A case for indexed nested loop join. In *HPCS*. IEEE, Washington, DC, USA, 204–212. https://doi.org/10.1109/HPCS.2018.00046

[93] Ritesh Nohria, Gustavo Santos, and Volker Haug. 2019. *IBM power system AC922 introduction and technical overview* (1st ed.). IBM International Technical Support Organization, Poughkeepsie, NY, USA. REDP-5494-00.

[94] Bryon S Nordquist and Stephen D Lew. 2009. Apparatus, system, and method for coalescing parallel memory requests. Patent No. US7492368B1, Filed Jan. 24th., 2006, Issued Feb. 17th., 2009.

[95] Nvidia 2009. *Nvidia's next generation CUDA compute architecture: Fermi*. Nvidia. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf v1.1.

[96] Nvidia 2016. *Nvidia Tesla P100*. Nvidia. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf WP-08019-001_v01.1.

[97] Nvidia 2017. *Nvidia Tesla V100 GPU architecture*. Nvidia. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf WP-08608-001_v1.1.

[98] Nvidia 2019. *Pascal MMU format changes*. Nvidia. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf Git Revision 60b67c3.

[99] Nvidia 2020. *CUPTI user's guide*. Nvidia. https://docs.nvidia.com/cupti/pdf/Cupti.pdf DA-05679-001_v11.1.

[100] Nvidia 2020. *Nvidia A100 tensore core GPU architecture*. Nvidia. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-

ampere-architecture-whitepaper.pdf v1.0.
[101] Nvidia 2021. *CUDA C programming guide*. Nvidia. http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf PG-02829-001_v11.3.
[102] Nvidia 2021. *Parallel thread execution ISA*. Nvidia. https://docs.nvidia.com/cuda/pdf/ptx_isa_7.3.pdf v7.3.
[103] OpenCAPI Consortium 2020. *OpenCAPI 4.0 transaction layer specification*. OpenCAPI Consortium. Version 1.0.
[104] Mark Papermaster. 2020. *AMD Financial Analyst Day*. AMD. Retrieved Jun 10, 2021 from https://ir.amd.com/news-events/analyst-day
[105] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2019. Revisiting hash join on graphics processors: A decade later. In *ICDEW*. IEEE, Washington, DC, USA, 294–299. https://doi.org/10.1109/ICDEW.2019.00008
[106] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based pipelined query processing engine. In *SIGMOD*. ACM, New York, NY, USA, 1935–1950. https://doi.org/10.1145/2882903.2915224
[107] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A scalable join for massively parallel multi-GPU architectures. In *SIGMOD*. ACM, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254
[108] Johan Peltenburg, Ákos Hadnagy, Matthijs Brobbel, Robert Morrow, and Zaid Al-Ars. 2021. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In *FPT*. IEEE, Washington, DC, USA, 1–9. https://doi.org/10.1109/ICFPT52863.2021.9609833
[109] Johan Peltenburg, Lars T. J. van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H. Peter Hofstee. 2020. Battling the CPU bottleneck in Apache Parquet to Arrow conversion using FPGA. In *FPT*. IEEE, Washington, DC, USA, 281–286. https://doi.org/10.1109/ICFPT51103.2020.00048
[110] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB reach by exploiting clustering in page translations. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 558–567. https://doi.org/10.1109/HPCA.2014.6835964
[111] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *ASPLOS*. ACM, New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942
[112] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2011. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS*. 27–35.
[113] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2014. Waste not…Efficient co-processing of relational data. In *ICDE*. IEEE Computer Society, Los Alamitos, CA, USA, 508–519. https://doi.org/10.1109/ICDE.2014.6816677
[114] Constantin Pohl and Kai-Uwe Sattler. 2018. Joins in a heterogeneous memory hierarchy: Exploiting high-bandwidth memory. In *DaMoN*. ACM, New York, NY, USA, 8:1–8:10. https://doi.org/10.1145/3211922.3211929
[115] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*. ACM, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645
[116] Orestis Polychroniou and Kenneth A. Ross. 2019. Towards practical vectorized analytical query engines. In *DaMoN*. ACM, New York, NY, USA, 10:1–10:7. https://doi.org/10.1145/3329785.3329928
[117] Orestis Polychroniou and Kenneth A Ross. 2020. VIP: A SIMD vectorized analytical query engine. *VLDB J.* 29, 6 (2020), 1243–1261. https://doi.org/10.1007/s00778-020-00621-w
[118] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA*. IEEE Computer Society, Los Alamitos, CA, USA, 568–578. https://doi.org/10.1109/HPCA.2014.6835965
[119] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR*. www.cidrdb.org, 1–11.
[120] Stefan Richter, Victor Alvarez, and Jens Dittrich. 2015. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB* 9, 3 (2015), 96–107. https://doi.org/10.14778/2850583.2850585
[121] Christopher Root and Todd Mostak. 2016. MapD: A GPU-powered big data analytics and visualization platform. In *SIGGRAPH*. ACM, New York, NY, USA, 73:1–73:2. https://doi.org/10.1145/2897839.2927468
[122] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query processing on heterogeneous CPU/GPU systems. *ACM Comput. Surv.* 55, 1, Article 11 (Jan. 2022), 38 pages. https://doi.org/10.1145/3485126
[123] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2019. Performance analysis and automatic tuning of hash aggregation on GPUs. In *DaMoN*. ACM, New York, NY, USA, 8. https://doi.org/10.1145/3329785.3329922
[124] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. *PVLDB* 14, 4 (2020), 708–720. https://doi.org/10.14778/3436905.3436927
[125] Ran Rui and Yi-Cheng Tu. 2017. Fast equi-join algorithms on GPUs: Design and implementation. In *SSDBM*. ACM, New York, NY, USA, 17:1–17:12. https://doi.org/10.1145/3085504.3085521
[126] Nikolay Sakharnykh. 2017. *Maximizing unified memory performance in CUDA*. Nvidia. Retrieved Jun 10, 2021 from https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda
[127] Nadathur Satish, Mark J. Harris, and Michael Garland. 2009. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*. IEEE, Washington, DC, USA, 1–10. https://doi.org/10.1109/IPDPS.2009.5161005
[128] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *SIGMOD*. ACM, New York, NY, USA, 351–362. https://doi.org/10.1145/1807167.1807207
[129] Tobias Schmidt, Maximilian Bandle, and Jana Giceva. 2021. A four-dimensional analysis of partitioned approximate filters. *PVLDB* 14, 11 (2021), 2355–2368. https://doi.org/10.14778/3476249.3476286
[130] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*. ACM, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917
[131] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the surprising difficulty of simple things: The case of radix partitioning. *PVLDB* 8, 9 (2015), 934–937. https://doi.org/10.14778/2777598.2777602
[132] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *SIGMOD*. ACM, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595
[133] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache conscious algorithms for relational query processing. In *PVLDB*. Morgan Kaufmann, San Francisco, CA, USA, 510–521. https://dl.acm.org/doi/10.5555/645920.758363
[134] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on GPUs. In *ICDE*. IEEE, Washington, DC, USA, 698–709. https://doi.org/10.1109/ICDE.2019.00068
[135] Evangelia A. Sitaridi and Kenneth A. Ross. 2013. Optimizing select conditions on GPUs. In *DaMoN*. ACM, New York, NY, USA, 4. https://doi.org/10.1145/2485278.2485282
[136] Greg Smith. 2014. *Stack Overflow*. Nvidia. Retrieved Nov 8, 2021 from https://stackoverflow.com/questions/27253434/what-does-overflow-mean-during-cuda-profiling/27262797#27262797
[137] Greg Smith. 2019. *Nvidia Developer Forums*. Nvidia. Retrieved Oct 13, 2020 from https://forums.developer.nvidia.com/t/pascal-l1-cache/49571/14
[138] Greg Smith. 2019. *Nvidia Developer Forums*. Nvidia. Retrieved Oct 13, 2020 from https://forums.developer.nvidia.com/t/performance-counters-similar-to-cpu/74949/2
[139] William J. Starke, J. S. Dodson, Jeffrey Stuecheli, Eric Retter, Brad W. Michael, Stephen J. Powell, and James A. Marcella. 2018. IBM POWER9 memory architectures for optimized systems. *IBM Journal of Research and Development* 62, 4/5 (2018), 3:1–3:13. https://doi.org/10.1147/JRD.2018.2846159
[140] Elias Stehle and Hans-Arno Jacobsen. 2017. A memory bandwidth-efficient hybrid radix sort on GPUs. In *SIGMOD*. ACM, New York, NY, USA, 417–432. https://doi.org/10.1145/3035918.3064043
[141] Ze-ke Wang, Bingsheng He, and Wei Zhang. 2015. A study of data partitioning on OpenCL-based FPGAs. In *FPL*. IEEE, Washington, DC, USA, 1–8. https://doi.org/10.1109/FPL.2015.7293941
[142] Jan Wassenberg and Peter Sanders. 2011. Engineering a multi-core radix sort. In *Euro-Par (Lecture Notes in Computer Science, Vol. 6853)*. Springer-Verlag, Berlin, Germany, 160–169. https://doi.org/10.1007/978-3-642-23397-5_16
[143] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational joins on GPUs: A closer look. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2663–2673. https://doi.org/10.1109/TPDS.2017.2677451
[144] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. 2011. Scalable aggregation on multicore processors. In *DaMoN*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/1995441.1995442
[145] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The yin and yang of processing data warehousing queries on GPU devices. *PVLDB* 6, 10 (2013), 817–828. https://doi.org/10.14778/2536206.2536210
[146] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*. www.cidrdb.org, 1–8.
[147] Marcin Zukowski, Sándor Héman, and Peter A. Boncz. 2006. Architecture-conscious hashing. In *DaMoN*. ACM, New York, NY, USA, 6. https://doi.org/10.1145/1140402.1140410