Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects

Clemens Lutz
clemens.lutz@dfki.de
DFKI GmbH
Berlin, Germany

Sebastian Breß
sebastian.bress@tu-berlin.de
TU Berlin
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

GPUs have long been discussed as accelerators for database query processing because of their high processing power and memory bandwidth. However, two main challenges limit the utility of GPUs for large-scale data processing: (1) the on-board memory capacity is too small to store large data sets, yet (2) the interconnect bandwidth to CPU main-memory is insufficient for ad hoc data transfers. As a result, GPU-based systems and algorithms run into a transfer bottleneck and do not scale to large data sets. In practice, CPUs process large-scale data faster than GPUs with current technology.

In this paper, we investigate how a fast interconnect can resolve these scalability limitations using the example of NVLink 2.0. NVLink 2.0 is a new interconnect technology that links dedicated GPUs to a CPU. The high bandwidth of NVLink 2.0 enables us to overcome the transfer bottleneck and to efficiently process large data sets stored in main-memory on GPUs. We perform an in-depth analysis of NVLink 2.0 and show how we can scale a no-partitioning hash join beyond the limits of GPU memory. Our evaluation shows speed-ups of up to 18× over PCI-e 3.0 and up to 7.3× over an optimized CPU implementation. Fast GPU interconnects thus enable GPUs to efficiently accelerate query processing.

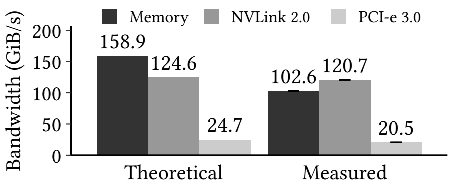


Figure 1: NVLink 2.0 eliminates the GPU's main-memory access disadvantage compared to the CPU.

ACM Reference Format:
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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3389705

1 INTRODUCTION

Over the past decade, co-processors such as GPUs, FPGAs, and ASICs have been gaining adoption in research [17, 35, 38, 56] and industry [88] to manage and process large data. Despite this growth, GPU-enabled databases occupy a niche [67] in the overall databases market [28]. In contrast, there is wide-spread adoption in the deep learning [21, 71] and high performance computing domains. For instance, 29% of the Top500 supercomputers support co-processors [92]. Database research points out that a data *transfer bottleneck* is the main reason behind the comparatively slow adoption of GPU-enabled databases [31, 100].

The transfer bottleneck exists because current GPU interconnects such as PCI-e 3.0 [1] provide significantly lower bandwidth than main-memory (i.e., *CPU memory*). We break down the transfer bottleneck into three fundamental limitations for GPU-enabled data processing:

L1: Low interconnect bandwidth. When the database decides to use the GPU for query processing, it must transfer data ad hoc from CPU memory to the GPU. With current interconnects, this transfer is slower than processing the data on the CPU [13, 31, 100]. Consequently, we can only speed up data processing on GPUs by increasing the interconnect bandwidth [16, 26, 51, 89, 95]. Although data compression [24, 85] and approximation [81] can reduce transfer volume, their effectiveness varies with the data and query.

L2: Small GPU memory capacity. To avoid transferring data, GPU-enabled databases cache data in GPU memory [13, 38, 50, 83]. However, GPUs have limited on-board *GPU memory* capacity (up to 32 GiB). In general, large data sets cannot be stored in GPU memory. The capacity limitation is intensified by database operators that need additional space for intermediate state, e.g., hash tables or sorted arrays. In sum, GPU co-processing does not scale to large data volumes.

L3: Coarse-grained cooperation of CPU and GPU. Using only a single processor for query execution leaves available resources unused [16]. However, co-processing on multiple, heterogeneous processors inherently leads to execution skew [22, 32], and can even cause slower execution than on a single processor [13]. Thus, CPU and GPU must cooperate to ensure that the CPU's execution time is the lower bound. Cooperation requires efficient synchronization between processors on shared data structures such as hash tables or B-trees, that is not possible with current interconnects [4].

In this work, we investigate the scalability limitations of GPU co-processing and analyze how a faster interconnect helps us to overcome them. A new class of *fast interconnects*, that currently includes NVLink, Infinity Fabric, and CXL, provides high bandwidth, and low latency. In Figure 1, we show that fast interconnects enable the GPU to access CPU memory with the full memory bandwidth. Furthermore, we propose a new co-processing strategy that takes advantage of the cache-coherence provided by fast interconnects for fine-grained CPU-GPU cooperation. Overall, fast interconnects integrate GPUs tightly with CPUs and significantly reduce the data transfer bottleneck.

Our contributions are as follows:

(1) We analyze NVLink 2.0 to understand its performance and new functionality in the context of data management (Section 3). NVLink 2.0 is one representative of the new generation of fast interconnects.
(2) We investigate how fast interconnects allow us to perform efficient ad hoc data transfers. We experimentally determine the best data transfer strategy (Section 4).
(3) We scale queries to large data volumes while considering the new trade-offs of fast interconnects. We use a no-partitioning hash join as an example (Section 5).
(4) We propose a new cooperative and robust co-processing approach that enables CPU-GPU scale-up on a shared, mutable data structure (Section 6).
(5) We evaluate joins as well as a selection-aggregation query using a fast interconnect (Section 7).

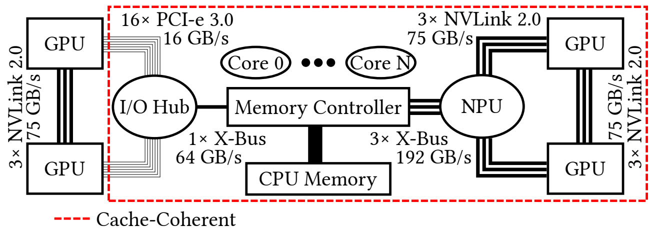


Figure 2: Architecture and cache-coherence domains of GPU interconnects, with their electrical bandwidths annotated.

The remainder of the paper is structured as follows. In Section 2, we briefly explain the hash join algorithm and highlight NVLink 2.0. We present our contributions in Sections 3–6. Then, we present our experimental results in Section 7 and discuss our insights in Section 8. Finally, we review related work in Section 9 and conclude in Section 10.

2 BACKGROUND

In this section, we provide an overview of the no-partitioning hash join, and the PCIe 3.0 and NVLink 2.0 interconnects.

2.1 No-Partitioning Hash Join

In this work, we focus on the no-partitioning hash join algorithm as proposed by Blanas et al. [10]. The no-partitioning hash join algorithm is a parallel version of the canonical hash join [8]. We focus on this algorithm because it is simple and well-understood. Loading base relations from CPU memory requires high bandwidth, scaling the hash table beyond GPU memory requires low latency, and sharing the hash table between multiple processors requires cache-coherence. Thus, the no-partitioning hash join is a useful instrument to investigate fast GPU interconnects.

The anatomy of a no-partitioning hash join consists of two phases, the build and the probe phase. The build phase takes as input the smaller of the two join relations, which we denote as the inner relation R. In the build phase, we populate the hash table with all tuples in R. After the build phase is complete, we run the probe phase. The probe phase reads the second, larger input relation as input. We name this relation the outer relation S. For each tuple in S, we probe the hash table to find matching tuples from R. When executing the hash join in parallel on a system with p cores, its time complexity observes $O(1/p(|R| + |S|))$.

2.2 GPU Interconnects

Discrete GPUs are connected to the system using an interconnect bus. In Figure 2, we contrast the architectures of PCI-e 3.0 and NVLink 2.0.

2.2.1 *PCI-e 3.0.*
State-of-the-art systems connect GPUs with a PCI Express 3.0 [1] interconnect.

Physical Layer. PCI-e connects devices point-to-point to bridges in a tree topology rooted at the CPU. All connections therefore share the available bandwidth. Each connection consists of multiple physical lanes. The lanes are full-duplex, which enables bidirectional transfers at full speed. GPUs bundle 16 lanes and aggregate their bandwidth (16 GB/s in total). However, the electrical bandwidth is not achievable for data transfers, due to packet, encoding, and checksum overheads. PCI-e specifies a packet-based communication protocol. Packets consist of a payload of up to 512 bytes and a 20–26 byte header. Although the header incurs little overhead for bulk transfers, the overhead is significant for the small payloads of irregular memory accesses [70].

Transfer Primitives. PCI-e provides two data transfer primitives: memory-mapped I/O (*MMIO*) and direct memory access (*DMA*). MMIO maps GPU memory into the CPU's address space. The CPU then initiates a PCI-e transfer by accessing the mapped GPU memory with normal load and store instructions. In contrast, DMA allows the GPU to directly access CPU memory. The key difference to MMIO is that DMA only allows access to a pre-determined range of *pinned memory*. Memory is "pinned" by locking the physical location of pages, which prevents the OS from moving them. DMA operations can thus be offloaded to copy engines. These are dedicated hardware units that facilitate both bidirectional transfers and overlapping transfers with computation.

Software APIs. CUDA [74] exposes these two primitives through three abstractions. `cudaMemcpyAsync` copies pageable (i.e., non-pinned) memory through MMIO, but copies pinned memory using DMA copy engines. In contrast, Unified Virtual Addressing exposes "zero-copy" pinned memory to the GPU via DMA. Finally, *Unified Memory* transparently moves CPU pages to GPU memory. The page migration is triggered by a memory access to a page not present in GPU memory. The operating system receives a page fault, and moves the requested page from CPU memory to GPU memory [102]. To avoid the page fault's latency, pages can be explicitly prefetched using `cudaMemPrefetchAsync`. Although Unified Memory is built on the aforementioned transfer primitives, CUDA hides the type of memory used internally.

2.2.2 *NVLink 2.0.*
Nvidia Volta GPUs and IBM POWER9 CPUs support NVLink 2.0 [15, 40, 41, 73], which is a new fast interconnect for GPUs.

Physical Layer. NVLink 2.0 connects up to one CPU and six GPUs in a point-to-point mesh topology, which has the advantage of higher aggregate bandwidth compared to a tree. Connections consists of multiple full-duplex links that communicate at 25 GB/s per direction. A device has up to six links. Of these, up to three links can be bundled for a total of 75 GB/s. Thus, two GPUs can saturate CPU memory bandwidth, but adding a third reduces the per-GPU bandwidth by ⅓. Like PCI-e, NVLink transmits packets. However, packet headers incur less overhead for small packets, with a 16 byte header for up to 256 bytes of payload.

Transfer Primitives. Data transfers from CPU memory can use MMIO and DMA copy engines. However, in contrast to PCI-e, NVLink gives the GPU direct access to pageable CPU memory. GPU load, store, and atomic operations are translated into CPU interconnect commands (i.e., X-bus on POWER9) by the *NVLink Processing Unit (NPU)*. The NPU is connected by three X-Bus links, each capable of 64 GB/s.

Address Translation. The GPU is integrated into a system-wide address space. If a TLB miss occurs on a GPU access to CPU memory, the NVLink Processing Unit provides the address translation by walking the CPU's page table. Thus, in contrast to Unified Virtual Addressing and Unified Memory, address translations do not require OS intervention.

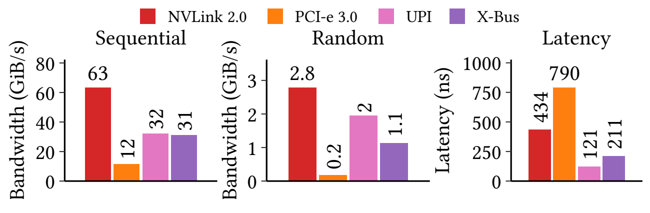
Cache-coherence. Memory accesses are cache-coherent on 128-byte cache-line boundaries. The CPU can thus cache GPU memory in its cache hierarchy, and the GPU can cache CPU memory in its L1 caches. Cache-coherence guarantees that writes performed by one processor are visible by any other processor. The observable order of memory operations depends on the memory consistency model. Intel CPUs guarantee that aligned reads and writes are atomic, and that writes are (nearly) sequentially consistent [42, vol.3A, §8.2]. In contrast, IBM CPUs and Nvidia GPUs have weaker memory consistency models [63].

Related Interconnects. In contrast to NVLink 1.0 [72], NVLink 2.0 provides higher bandwidth, cache-coherence, and more advanced address translation services. AMD Infinity Fabric [3] and Intel CXL [19, 44] offer similar features as NVLink 2.0, but are commercially not yet available for GPUs. ARM AXI [96], IBM OpenCAPI [2], and Intel QPI/UPI [43, 77] are comparable interconnects for FPGAs.

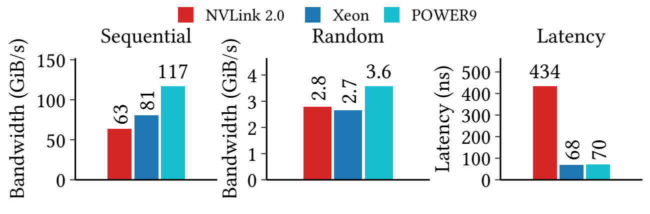
3 ANALYSIS OF A FAST INTERCONNECT

In this section, we analyze the class of fast interconnects by example of NVLink 2.0 to understand their performance and new functionality in the context of data management. The main improvements of fast interconnects compared to PCI-e 3.0 are higher bandwidth, lower latency, and cache-coherence. We investigate these properties and examine the benefits and challenges for scaling co-processing.

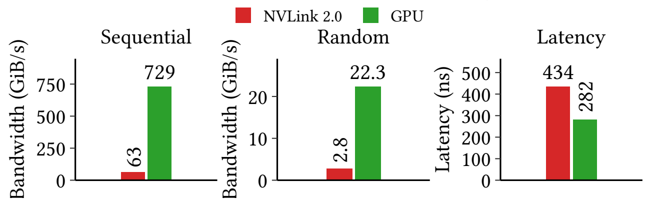
Bandwidth & Latency. We start by quantifying how much NVLink 2.0 improves the GPU's interconnect performance. We compare NVLink 2.0's ② performance to GPU (PCI-e 3.0 ①) and CPU interconnects (Intel Xeon Ultra Path Interconnect (UPI) ③, IBM POWER9 X-Bus ④), CPU memory (Intel Xeon ⑤, IBM POWER9 ⑥), and GPU memory (Nvidia V100



(a) NVLink 2.0 vs. CPU & GPU Interconnects.

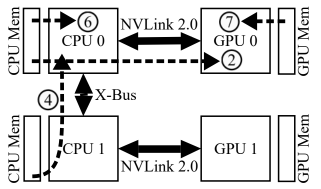


(b) NVLink 2.0 vs. CPU memory.

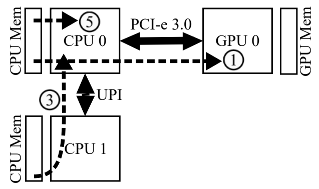


(c) NVLink 2.0 vs. GPU memory.

Figure 3: Bandwidth and latency of memory reads on IBM and Intel systems with Nvidia GPUs. Compare to data access paths shown in Figure 4.



(a) 2× IBM POWER9 with 2× Nvidia V100-SXM2.



(b) 2× Intel Xeon with 1× Nvidia V100-PCIE.

Figure 4: Data access paths on IBM and Intel systems.

⑦). We visualize these data access paths in Figure 4. In all our measurements we show 4-byte read accesses on 1 GiB of data. We defer giving further details on our measurement setup and methodology to Section 7.1.

We first compare NVLink 2.0 to the other GPU and CPU interconnects in Figure 3(a). Our measurements show that NVLink 2.0 has 5× more sequential bandwidth than PCI-e 3.0, and twice as much as UPI and X-Bus. Random accesses patterns are 14× faster than PCI-e 3.0, and 35% faster than UPI. However, while the latency of NVLink 2.0 is 45% lower than PCI-e 3.0, it is 3.6× higher than UPI and 2× higher than X-Bus. Overall, NVLink 2.0 is significantly faster than PCI-e 3.0, and more bandwidth-oriented than the CPU interconnects.

Next, we show the NVLink 2.0 vs. CPU memory in Figure 3(b). We note that the IBM CPU has 8 DDR4-2666 memory channels, while the Intel Xeon only has 6 channels of the same memory type. We see that for sequential accesses, the Intel Xeon and IBM POWER9 have 28% and 65% higher bandwidth than NVLink 2.0, respectively. For random accesses, NVLink 2.0 is on par with the Intel Xeon, but 30% slower than the IBM POWER9. The latency of NVLink 2.0 is 6× higher than the latency of CPU memory. We take away that, although NVLink 2.0 puts the GPU within a factor of two of the CPUs' bandwidth, it adds significant latency.

Finally, in Figure 3(c), we compare GPU accesses to CPU memory over NVLink 2.0 with GPU memory. We observe that both access patterns have an order-of-magnitude higher bandwidth in GPU memory, but that latency over NVLink 2.0 is only 54% higher. As GPUs are designed to handle such high-latency memory accesses [39, 94], they are well-equipped to cope with the additional latency of NVLink 2.0.

Cache-coherence. Cache-coherence simplifies the practical use of NVLink 2.0 for data processing. The advantages are three-fold. First, the GPU can directly access any location in CPU memory, therefore pinning memory becomes unnecessary. Second, allocating pageable memory is faster than allocating pinned memory [25, 68, 93]. Third, the operating system and database are able to perform background tasks that are important for long-running processes, such as memory defragmentation [18] and optimizing NUMA locality through page migration [61].

In contrast, the non-cache-coherence of PCI-e has two main drawbacks. First, data consistency must be managed in software instead of in hardware. The programmer either manually flushes the caches [74], or the OS migrates pages [72]. Second, system-wide atomics are unsupported. Instead, a work-around is provided by first migrating Unified Memory pages to GPU memory, and then performing the atomic operation in GPU memory [76].

Research shows that adding fine-grained cache-coherence to PCI-e is not feasible due to its high latency [27]. However, NVLink 2.0 removes these limitations [41] and thus is better-suited for data processing.

Benefits. We demonstrate three benefits of NVLink 2.0 for data processing with a no-partitioning hash join. First, we are able to scale the probe-side relation to arbitrary data volumes due to NVLink 2.0's high sequential bandwidth. With the hash table stored in GPU memory, we retain the GPU's performance advantage compared to a CPU join. Second, we provide build-side scalability to arbitrary data volumes using NVLink 2.0's low latency and high random access bandwidth. Thus, we are able to spill the hash table from GPU to CPU memory. Third, we employ the cache-coherence and system-wide atomics of NVLink 2.0 to share the hash table between a CPU and a GPU and scale-up data processing.

Challenges. Despite the benefits of NVLink 2.0 for data processing, translating high interconnect performance into high-performance query processing will require addressing the following challenges.

First, an out-of-core GPU join operator must perform both data access and computation efficiently. Early GPU join approaches cannot saturate the interconnect [34, 35]. More recent algorithms saturate the interconnect, and are optimized to access data over a low-bandwidth interconnect [47, 89]. This can involve additional partitioning steps on the CPU [89]. We investigate how a GPU join operator can take full advantage of the higher interconnect performance.

Second, scaling the build-side volume beyond the capacity of GPU memory in a NP-HJ requires spilling the hash table to CPU memory. However, spilling to CPU memory implies that the GPU performs irregular accesses to CPU memory, as, by design, hash functions map keys to uniformly distributed memory locations. Such irregular accesses are inefficient over high-latency interconnects. For this reason, previous approaches either cannot scale beyond GPU memory [34, 47], or are restricted to partitioning-based joins [89]. Higher interconnect performance requires us to reconsider how well a NP-HJ that spills to CPU memory performs on GPUs.

Third, fully exploiting a heterogeneous system consisting of CPUs and GPUs requires them to cooperatively process the join. We must take into account data locality, synchronization costs, and the differences in hardware architectures to achieve efficiency.

4 EFFICIENT DATA TRANSFER BETWEEN CPU AND GPU

In order to process data, the GPU needs to read input data from CPU memory. Since the GPU memory is limited to tens of gigabytes, we cannot store a large amount of data on the GPU. As a consequence, any involvement of the GPU in data processing requires ad hoc data transfer, which makes interconnect bandwidth the most critical resource (L1).

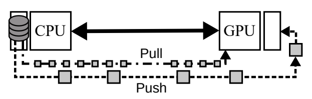


Figure 5: Push- vs. pull-based data transfer methods.

We can choose between different strategies to initiate data transfers between CPU and GPU. Each strategy shows different performance on the same interconnect. In this section, we discuss these data transfer strategies to identify the most efficient way for data transfer. We build on these insights in the following sections.

Recent versions of CUDA provide a rich set of APIs that abstract the MMIO and DMA transfer primitives described in Section 2.2. From these APIs, we derive eight transfer methods that we list in Table 1. We divide these methods into two categories based on their semantics, *push-based* and *pull-based.* On a high level, push-based methods perform course-grained transfers to GPU memory, whereas in pull-based methods the GPU directly accesses CPU memory. We depict these differences in Figure 5. We first describe push-based methods, and then pull-based methods.

Table 1: An overview of GPU transfer methods.

Method	Semantics	Level	Granularity	Memory
Pageable Copy	Push	SW	Chunk	Pageable
Staged Copy				
Dynamic Pinning				
Pinned Copy				Pinned
UM Prefetch				Unified
UM Migration	Pull	OS	Page	Unified
Zero-Copy		HW	Byte	Pinned
Coherence				Pageable

4.1 Push-based Transfer Methods

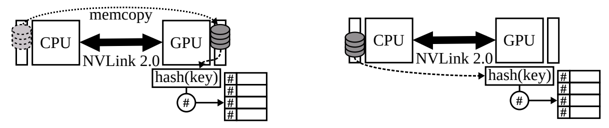
In order to transfer data, push-based methods rely on a pipeline to hide transfer latency. The pipeline is implemented in software and executed by the CPU. We describe the pipeline stages of each method and contrast their differences.

Pageable Copy. Pageable Copy is the most basic method to copy data to the GPU. It is exposed in the API via the `cudaMemcpyAsync` function, and transfers data directly from any location in pageable memory. As the API is called on the CPU, data are *pushed* to the GPU. Before we setup the pipeline, we split the data into chunks. Subsequently, we setup a two-stage pipeline by first transferring each chunk to the GPU, and then processing the chunk on the GPU. As both steps can be performed in parallel, the computation overlaps with the transfer.

Pinned Copy. As Nvidia recommends using pinned memory instead of pageable memory for data transfer [75], we apply the same technique as in Pageable Copy to pinned memory. Thus, the hardware can perform DMA using the copy engines instead of using a CPU thread to copy via memory-mapped I/O. Therefore, Pinned Copy is typically faster than Pageable Copy, but requires the database to store all data that is accessed by the GPU in pinned memory.

Staged Copy. However, storing all data in pinned memory violates Nvidia's recommendation to consider pinned memory as a scarce resource [75], and pinning large amounts of memory complicates memory management. Therefore, we setup a pinned staging buffer for the copy. In the pipeline, we first copy a chunk of data from pageable memory into the pinned memory buffer. Then, we perform the transfer and compute stages. We thus pipeline the transfer at the expense of an additional copy operation within CPU memory.

Dynamic Pinning. CUDA supports pinning pages of preexisting pageable memory. This allows us to pin pages ad hoc



(a) Data and hash table in GPU memory. **(b) Data in CPU memory and hash table in GPU memory.**

Figure 6: Scaling the probe side to any data size.

before we transfer data to the GPU, avoiding an additional copy operation in CPU memory. After that, we execute the copy and compute stages.

Unified Memory Prefetch. If we use Unified Memory and know the data access pattern beforehand, we can explicitly prefetch a region of unified memory to the GPU before the access takes place. This avoids a drawback of the Unified Memory Migration method that we describe next, namely that migrating pages on-demand has high latency and stalls the GPU [102]. We execute the transfer in a two-stage pipeline that consists of prefetching a chunk of data to the GPU, and then running the computation. Thus, prefetching requires a software pipeline in addition to using Unified Memory.

4.2 Pull-based Transfer Methods

Many database operators access memory irregularly, especially operators based on hashing. Hashed accesses are irregular, because hash functions are designed to generate uniform and randomly distributed output. These accesses are also data-dependent, as the hash function's input are attributes of a relation (e.g., the primary key).

Push-based transfer methods cannot handle these types of memory access. The CPU decides which data are transferred to the GPU. Thus, the GPU has no control over which data it processes, and cannot satisfy data-dependencies.

In contrast, pull-based methods are able to handle data-dependencies, as they intrinsically request data. In the following, we introduce three pull-based transfer methods.

Unified Memory Migration. Instead of dealing with pageable and pinned memory inside the database, Unified Memory allows us to delegate data transfer to the operating system. Internally, memory pages are migrated to the GPU on a page access [102] (4 KiB on Intel CPUs, 64 KiB on IBM CPUs [69]). Therefore, the GPU *pulls* data, and pipelining in software is unnecessary. However, the database must explicitly allocate Unified Memory to store data.

Zero-Copy. The previous approaches involve software or the operating system to manage transferring data. In contrast, we can use Unified Virtual Addressing to directly access data in CPU memory during GPU execution. We are able to load data with byte-wise granularity, but are restricted to accessing pinned memory. As Zero-Copy is managed entirely in hardware, software or operating system are not involved.

Coherence. NVLink 2.0 offers a new transfer method that is unavailable with previous interconnects. Using the hardware address translation services and cache-coherence, the GPU can directly access any CPU memory during execution. In contrast to Unified Memory Migration, NVLink 2.0 accesses memory with byte-wise granularity. In contrast to Unified Virtual Addressing, NVLink 2.0 does not require pinned memory. Instead, the GPU is able to directly access pageable CPU memory. Thus, NVLink 2.0 lifts previous constraints on the memory type and access granularity.

5 SCALING GPU HASH JOINS TO ARBITRARY DATA SIZES

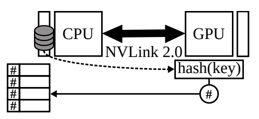
Current algorithms and systems for data processing on GPUs are all limited to some degree by the capacity of GPU memory (L2). Being limited by GPU memory capacity is the most fundamental problem in adopting GPU acceleration for data management in practice. In this section, we study how fast interconnects enable us to efficiently scale up data processing to arbitrary database sizes.

We study the impact of fast interconnects on the example of a no-partitioning hash join because of its unique requirements: (1) The build phase performs random memory accesses and thus requires either a low-latency interconnect to access the hash table in CPU memory, or enough GPU memory to store the hash table. The latter is a common scalability constraint. (2) The probe phase puts high demands on the interconnect's bandwidth. We discuss how we can scale up the probe side (Section 5.1) and the build side (Section 5.2), respectively, and propose our hybrid hash table approach to improve performance (Section 5.3).

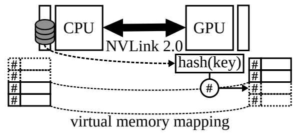
5.1 Scaling the Probe Side to Any Data Size

Transferring the inner and outer relations on-the-fly allows us to scale the relations' cardinalities regardless of GPU memory capacity. We begin by describing a simple, *baseline join* [29, 99] that is non-scalable. After that, we remove the probe-side cardinality limit by comparing the baseline to the *Zero-Copy pull-based join* introduced by Kaldewey et al. [47]. Based on the Zero-Copy join, we contribute our *Coherence join* that uses the Coherence transfer method. To simplify the discussion, we focus on pull-based methods. However, at the cost of additional complexity, we could instead use push-based pipelines to achieve probe-side scalability [34, 35].

First, in the baseline approach that we show in Figure 6a, we first copy the entire build-side relation R to GPU memory. When the copy is complete, we build the hash table in GPU memory. Following that, we evict R and copy the probe-side relation S to GPU memory. We probe the hash table



(a) Data and hash table in CPU memory.



(b) Data in CPU memory and hash table spills from GPU memory into CPU memory.

Figure 7: Scaling the build side to any data size.

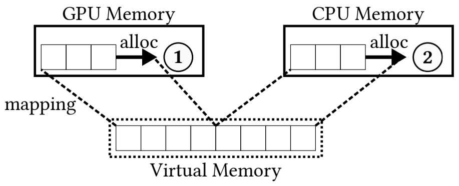


Figure 8: Allocating the hybrid hash table.

and emit the join result (i.e., an aggregate or a materialization). The benefit of this approach is that it only requires the hardware to support synchronous copying. However, this baseline doesn't scale to arbitrary data sizes, as it is limited by the GPU's memory capacity.

Next, in Figure 6b, we illustrate our probe-side scalable join. By using a pull-based transfer method, we are able to remove the scalability limitation. Zero-Copy and Coherence enable us to access CPU memory directly from the GPU (i.e., by dereferencing a pointer). Therefore, we build the hash table on the GPU by pulling R tuples on-demand from CPU memory. Behind the scenes, the hardware manages the data transfer. After we finish building the hash table, we pull S tuples on-demand and probe the hash table.

Finally, we replace the Zero-Copy transfer method with the Coherence transfer method in the Zero-Copy join. The Zero-Copy method requires the base relations to be stored in pinned memory. However, databases typically store data in pageable memory. We enable the GPU to access any memory location in pageable memory by replacing Zero-Copy with Coherence, which simplifies GPU data processing.

5.2 Scaling the Build Side to Any Data Size

We assume that the hash table is small enough to fit into GPU memory in Section 5.1. This limits the cardinality of R. We now lift this limitation and consider large hash tables.

We show our build-side scalable join in Figure 7a. The join is based on our probe-side scalable join, that we introduce in Section 5.1. However, in contrast to our probe-side scalable join, we store the hash table in CPU memory. By storing the hash table in CPU memory instead of in GPU memory, we are no longer constrained by the GPU's memory capacity.

In contrast to reading in base relations, hash table operations (insert and lookup) are data-dependent and have an irregular memory access pattern. Pull-based transfer methods (e.g., Coherence) enable us to perform these operations on the GPU in CPU memory. As we typically allocate memory specifically to build the hash table, we can allocate pinned memory or Unified Memory for use with the Zero-Copy or Unified Memory Migration methods. This flexibility allows us to choose the optimal transfer method for our hardware.

5.3 Optimizing the Hash Table Placement

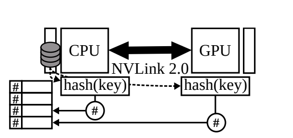
Although the Coherence transfer method enables the GPU to access any CPU memory location, access performance is non-uniform and varies with memory locality. CPU memory is an order-of-magnitude slower than GPU memory for random accesses (see Section 3). We combine the advantages of both memory types in our new *hybrid hash table*. We design the hybrid hash table such that access performance degrades gracefully when the hash table's size is increased.

In Figure 7b, we show that our hybrid hash table can replace a hash table in CPU memory without any modifications to the join algorithm. This is possible because the hybrid hash table uses virtual memory to abstract the physical location of memory pages. We use virtual memory to combine GPU pages and CPU pages into a single, contiguous array. Virtual memory has been available previously on GPUs [49]. However, fast interconnects integrate the GPU into a system-wide address space, which enables us to map physical CPU pages next to GPU pages in the address space.

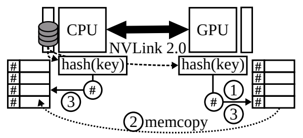
We allocate the hybrid hash table using a greedy algorithm, that we depict in Figure 8. By default, ① we allocate GPU memory. If the hash table is small enough, we allocate the entire hash table in GPU memory. Otherwise, ② if not enough GPU memory is available, we allocate memory on the CPU that is nearest to the GPU. Therefore, we spill the hash table to CPU memory. If that CPU has insufficient memory, we recursively search the next-nearest CPUs of a multi-socket NUMA system until we have allocated sufficient memory for the hash table. Overall, we allocate part of the hash table in GPU memory, and part in CPU memory.

The hybrid hash table is optimized for handling the worst case of a uniform join key distribution. We model this case as follows. We assume that the hash table consists of G_{mem} and C_{mem} bytes of GPU and CPU memory. We then expect that $A_{GPU} = \frac{G_{mem}}{G_{mem}+C_{mem}}$ of all accesses are to GPU memory. We estimate hash join throughput to be $J_{tput} = A_{GPU}G_{tput} + (1 - A_{GPU})C_{tput}$, where G_{tput} and C_{tput} are the hash join throughputs when the hash table resides in GPU and CPU memory, respectively. Overall, throughput is determined by the proportion of accesses to a given processor.

There are two additional benefits to our hybrid hash table that cannot be replicated without hardware support. First, the contiguous array underlying the hybrid hash table comes



(a) Cooperatively process join on CPU and GPU with hash table in CPU memory.



(b) Build hash table on GPU, copy the hash table to processor-local memories, and then cooperatively probe on CPU and GPU.

Figure 9: Scaling-up using CPU and GPU.

at zero additional cost, because processors perform virtual-to-physical address translation regardless of memory location. We could simulate a hybrid hash table on hardware without a system-wide address space by mapping together two non-contiguous arrays in software. However, the software indirection would add extra cycles on each access. Second, besides a change to the allocation logic, we leave the hash join algorithm unmodified. Thus, our hybrid hash table can easily be integrated into existing databases.

6 SCALING-UP USING CPU AND GPU

The third fundamental limitation (L3) of GPU co-processing is single processor execution. Without a way that enables CPUs and GPUs to collaborate in query processing, we leave available processing resources unused and cannot take full advantage of a heterogeneous CPU+GPU system.

In this section, our goal is to increase throughput by utilizing all available processors cooperatively, i.e., combining CPUs and GPUs. The main challenge is to guarantee that performance always improves when we schedule work on a GPU, even for the first query that is executed on the GPU. For this, the scheduling approach must be highly robust with respect to execution skew. As a consequence, truly scalable co-processing has the following three requirements. (a) We must process chunks of input data such that we can exploit data parallelism to use CPU and GPU for the same query. (b) At the same time, the task scheduling approach needs to avoid load imbalances. (c) The approach must avoid resource contention (e.g., of memory bandwidth) to prevent slowing down the overall execution time.

We first propose a heterogeneous task scheduling scheme. Following that, we optimize our hash table placement strategy for co-processing. Finally, we describe scaling up on multiple GPUs that are connected with a fast interconnect.

6.1 Task Scheduling

Load imbalances inherently occur on heterogeneous architectures due to the relative throughput differences of processors. As the throughput of a processor depends on many variable parameters that change over time (e.g., query, data, processor clock speeds), we cannot know the relative differences upfront. A task scheduler ensures that all processors deliver their highest possible throughput.

We adapt the CPU-oriented, morsel-driven approach [10, 57] for GPUs. In the CPU-oriented approach, all cores work concurrently on the same data and, in the case of joins, the same hash table. Cores balance load by requesting fixed-sized chunks of data (i.e., *morsels*) from a central dispatcher, that is implemented as a read cursor. Each core advances at its own processing rate.

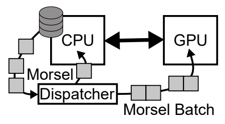


Figure 10: Dynamically scheduling tasks to CPU and GPU processors.

In Figure 10, we show our heterogeneous scheduling approach. In contrast to the CPU-oriented approach, we give each processor the right amount of work to minimize execution skew by considering the increased latency of scheduling work on a GPU, and the higher processing rate of the GPU. Instead of dispatching one morsel at-a-time, we dispatch batches of morsels to the GPU. Batching morsels amortizes the latency of launching a GPU kernel over more data. We empirically tune the batch size to our hardware.

6.2 Heterogeneous Hash Table Placement

Processors are fastest when accessing their local memories. Consequently, our hybrid hash table (Section 8) prefers data in GPU memory, and spills to CPU memory only when necessary. In our hybrid hash table, however, we consider only a single processor. In this section, we optimize for multiple, heterogeneous processors accessing the hash table via a fast interconnect. We consider two cases: one globally shared hash table, and multiple per-processor hash tables. We summarize the placement decision process in Figure 11.

In Figure 9a, we show the CPU and GPU processing a join using a globally shared hash table (*Het* strategy). Globally sharing a hash table retains the build-side scaling behavior

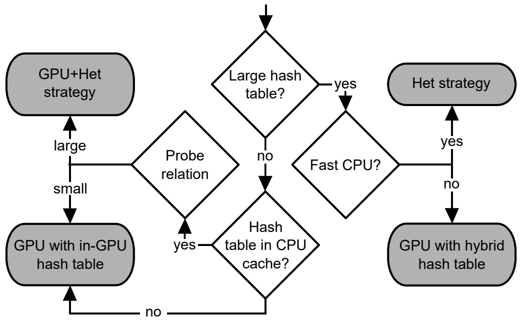


Figure 11: Hash table placement decision.

that we achieve in Section 5.2. However, we avoid our hybrid hash table optimization and store the hash table in CPU memory. We choose this design because we aim to always speed up processing when using a co-processor. Therefore, we avoid slowing down CPU processing through remote GPU memory accesses. In addition, the CPU has significantly lower performance when accessing GPU memory than the GPU accessing CPU memory, due to the CPU coping worse than the GPU with the high latencies of GPU memory and the interconnect [41].

We handle the special case of small build-side relations separately (*GPU + Het* strategy), because processors face contention when building the hash table. Furthermore, small hash tables allow us to optimize hash table locality. We show our small table optimization in Figure 9b. In a first step, ① one processor (e.g., the GPU) builds the hash table in processor-local memory (in this case, GPU memory). Following that, ② we copy the finished hash table to all other processors. By storing a local copy of the hash table on each processor, we ensure that all processors have high random-access bandwidth to the hash table. Finally, ③ we execute the probe phase on all processors using our heterogeneous scheduling strategy. Our strategy could be extended to multi-way joins (e.g., for a star schema) by building hash tables on a different processor in parallel, and then copying all hash tables to all processors. However, in this work, we focus on investigating fast interconnects using a single join.

6.3 Multi-GPU Hash Table Placement

Systems with multiple GPUs are connected in a mesh topology similar to multi-socket CPU systems. For small hash tables, we can use the GPU+Het execution strategy with multiple GPUs. However, for large hash tables, multi-GPU systems can distribute the hash table over multiple GPUs, as GPUs are latency insensitive [39, 94]. We distribute the table by interleaving the pages over all GPUs. This strategy is used in NUMA systems [57]. Fast interconnects enable us to use the strategy in multi-GPU systems.

In contrast to CPU+GPU execution, distributing computation over multiple GPUs provides three distinct advantages. First, using only GPUs avoids computational skew. Second, distributing large hash tables within GPU memory frees CPU memory bandwidth for loading the base relations. Finally, interleaving the hash table over multiple GPUs utilizes the full bi-directional bandwidth of fast interconnects, as opposed to the mostly uni-directional traffic of the Het strategy.

7 EVALUATION

In this section, we evaluate the impact of NVLink 2.0 on data processing. We describe our setup in Section 7.1. After that, we present our results in Section 7.2.

Table 2: Workload Overview.

Property	A (from [10])	B	C (from [54])
key / payload	8 / 8 bytes	8 / 8 bytes	4 / 4 bytes
cardinality of R	2^{27} tuples	2^{18} tuples	$1024 \cdot 10^6$ tuples
cardinality of S	2^{31} tuples	2^{31} tuples	$1024 \cdot 10^6$ tuples
total size of R	2 GiB	4 MiB	7.6 GiB
total size of S	32 GiB	32 GiB	7.6 GiB

7.1 Setup and Configuration

We first introduce our methodology and experimental setup. Then, we describe the data sets that we use in our evaluation. Finally, we introduce our experiments.

Environment. We evaluate our experiments on one GPU and two CPU architectures. We conduct our GPU measurements using an Nvidia Tesla V100-SXM2 and a V100-PCIE ("Volta"), on IBM and Intel systems, respectively. Both GPUs have 16 GB memory. We conduct our CPU measurements on a dual-socket IBM POWER9 at 3.3 GHz with 2×16 cores and 256 GB memory, and on a dual-socket Intel Xeon Gold 6126 ("Skylake-SP") at 2.6 GHz with 2×12 cores and 1.5 TB memory. The Intel system runs Ubuntu 16.04, and the IBM POWER9 system runs Ubuntu 18.04. We implement our experiments in C++ and CUDA. We use CUDA 10.1 and GCC 8.3.0 on all systems, and compile all code with "-O3" and native optimization flags.

Methodology. We measure throughput of the end-to-end join. We define join throughput as the sum of input tuples divided by the total runtime (i.e., $\frac{|R|+|S|}{\text{runtime}}$) [86, 89]. For each experiment, we report the mean and standard error over 10 runs. We note that our measurements are stable with a standard error less than 5% from the mean.

Workloads. In Table 2, we give an overview of our workloads. We specify workloads A and C similar to related work [8, 10, 54]. We scale these workloads 8× to create an out-of-core scenario. We define workload B as a modified workload A with a relation R that fits into the CPU L3 and GPU L2 caches and represents small dimension tables. All workloads assume narrow 8- or 16-byte <key, value> tuples. We generate tuples assuming a uniform distribution, and a foreign-key relationship between R and S. Unless noted otherwise, each tuple in S has exactly one match in R. We store the relations in a column-oriented storage model.

Settings. In the following experiments, we use the Coherence transfer method for NVLink 2.0 and the Zero Copy method for PCI-e 3.0, unless noted otherwise. We set up our no-partitioning hash join with perfect hashing, i.e., we assume no hash conflicts occur due to the uniqueness of primary keys. Our join is equivalent to the NOPA join described by Schuh et al. [86].

Baseline. As a CPU baseline, we use the radix partitioned, multi-core hash join implementation ("PRO") provided by

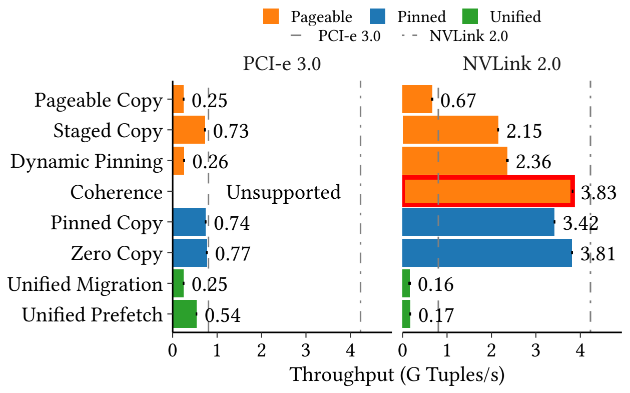


Figure 12: No-partitioning hash join using different transfer methods for PCI-e 3.0 and NVLink 2.0.

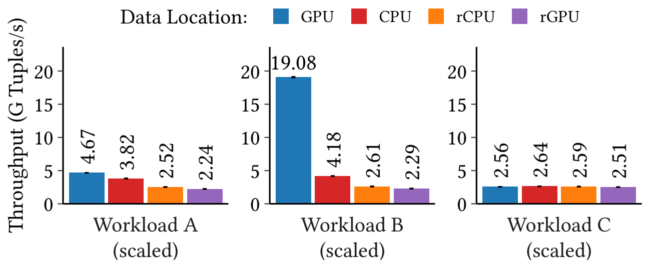


Figure 13: Join performance of the GPU when the base relations are located on different processors, increasing the number of interconnect hops from 0 to 3.

Barthels et al. [9]. We modify the baseline to use our perfect hash function, thus transforming the PRO join into a PRA join [86]. Furthermore, we tune our baseline to use the best radix bits (12 bits), page size (huge pages), SMT (enabled), software write-combine buffers (enabled) and NUMA locality parameters for our hardware. As our experiments run on one GPU, we run the baseline on one CPU.

Experiments. We conduct ten experiments. First, we evaluate the impact of transfer methods on data processing when using PCI-e 3.0 and NVLink 2.0. Then, we show the impact of NUMA locality considering the base relations and the hash table. Next, we explore out-of-core scalability when exceeding the GPU memory capacity with TPC-H query 6, the probe-side relation S, and the build-side relation R. Furthermore, we investigate the performance impact of different build-to-probe ratios, as well as skewed data and varying the join selectivity. Lastly, we investigate heterogeneous cooperation between a CPU and a GPU that share a hash table.

7.2 Experiments

In this section, we present our experimental results and describe our observations.

7.2.1 GPU Transfer Methods. In Figure 12, we show the join throughput of each transfer method with PCI-e 3.0 and NVLink 2.0 for workload A (2 GiB ⋈ 32 GiB). The outer relation is thus larger than GPU memory. We load both relations from CPU memory, and build the hash table in GPU memory.

PCI-e 3.0. We observe that pinning the memory is necessary to reach the peak transfer bandwidth of 12 GB/s. The Staged Copy method is within 5% of Zero Copy, despite copying from pageable memory. The hidden cost of using pageable memory is that we fully utilize 4 CPU cores to stage the data into pinned buffers. In contrast, Unified Migration and Unified Prefetch are 68% and 30% slower than Zero Copy. Although prefetching avoids the cost of demand-paging, we observe the overheads of evicting cached pages and mapping new pages in GPU memory. Pageable Copy and Dynamic Pinning are both significantly slower than Zero Copy. We note that the Coherence method is unsupported by PCI-e 3.0, due to PCI-e being non-cache-coherent.

NVLink 2.0. In contrast to PCI-e 3.0, NVLink 2.0 achieves up to 5× higher bandwidth. The Coherence method is within 14% of the maximum possible throughput. The throughput of Zero Copy matches that of Coherence, despite using pinned memory instead of pageable memory. In contrast, the DMA transfer from pinned memory with Pinned Copy is 11% slower. Transfers from pageable memory without using cache-coherence (i.e., Pageable Copy, Staged Copy, Dynamic Pinning) all achieve less throughput than Coherence. NVLink underperforms PCI-e in only two cases, when using either Unified Memory method[1]. Overall, the Coherence and Zero Copy methods are fastest, and NVLink 2.0 shows significantly higher throughput than PCI-e 3.0.

7.2.2 Data Locality. We measure the impact of base relation locality in Figure 13. We process the workloads from Table 2, and scale them down to fit into GPU memory (13 GiB, 12 GiB, and 10 GiB). We store R and S in GPU memory, CPU memory, remote CPU memory, and remote GPU memory (compare to Figure 4(a)). Each step increases the number of interconnect hops to load the data. In all measurements, we store the hash table in GPU memory.

Workload A. We observe that join throughput decreases by 32–46% as we increase the number of hops. We see that going from 1 to 2 hops has a larger effect than from 2 to 3 hops, because the X-Bus interconnect has lower throughput than NVLink 2.0 (cf. Figure 3(a)).

Workload B. We notice that storing the memory in GPU memory has 5.8× higher throughput than a single hop over NVLink 2.0. The reason is that the hash table is cached in the GPU's L2 cache, which has higher random access bandwidth

[1]We speculate that this is due to the POWER9 driver implementation receiving less optimization than on x86-64.

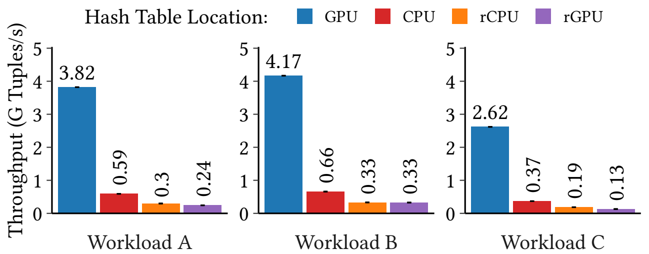


Figure 14: Join performance of the GPU when the hash table is located on different processors, increasing the number of interconnect hops from 0 to 3.

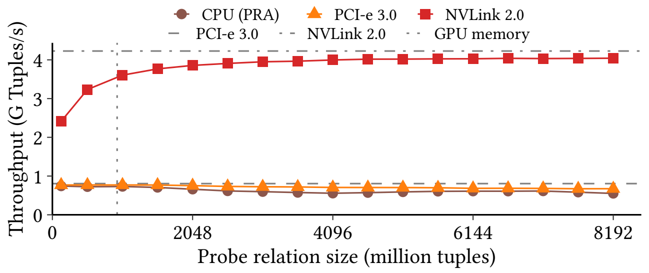


Figure 16: Scaling the probe-side relation.

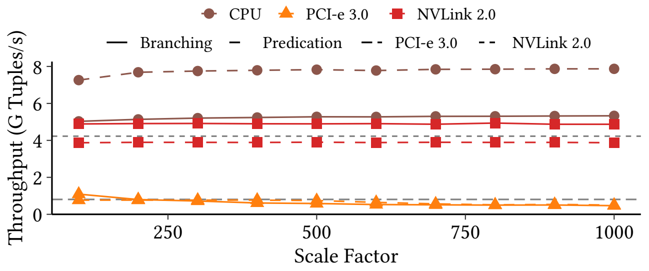


Figure 15: Scaling the data size of TPC-H query 6.

than GPU memory. For this workload, there is a 23% penalty for traversing three interconnects instead of one.

Workload C. In contrast to the best-case scenario represented by B, C is a worst-case scenario, as the relations have equal cardinalities (i.e., $|R| = |S|$). As a result, random memory accesses to GPU memory dominate the workload. Overall, NVLink 2.0 is not the bottleneck for hash joins that randomly access GPU memory. In addition, increasing the number of hops is mainly limited by the X-Bus' bandwidth.

7.2.3 Hash Table Locality. In Figure 14, we measure the influence of hash table locality on join performance. We process workloads A–C that have up to 34 GiB of data, and increase the interconnect hops to the hash table. In all measurements, we store the base relations in local CPU memory that is one hop away over NVLink 2.0.

Workloads A and C. We see that a single NVLink 2.0 hop causes an 75–82% throughput decrease. Adding a second hop and third hop effects another 50% and 17–33%, respectively.

Workload B. We observe that, in contrast to GPU memory, the small hash table is not cached in the GPU's L2 cache for NVLink 2.0. The L2 cache is memory-side [101], and cannot cache remote data. We conclude that reducing random access bandwidth and increasing latency has a significant impact on join throughput.

7.2.4 Selection and Aggregation Scaling. We scale TPC-H query 6 from scale factor 100 to 1000 in Figure 15. This constitutes a working set of 8.9–89.4 GiB. We assume that no data are cached in GPU memory, thus all data are read from CPU memory. We run branching and predicated variants. The CPU is an IBM POWER9 and we ensure that predication uses SIMD instructions.

Interconnects. The CPU achieves the highest throughput, and outperforms NVLink 2.0 by up to 67% and PCI-e 3.0 by up to 15.8×. However, NVLink 2.0 achieves a speedup of up to 9.8× over PCI-e 3.0, thus considerably closing the gap between the GPU and the CPU.

Branching vs. Predication. Counterintuitively, branching performs better than predication on the GPU with NVLink 2.0. This is caused by the query's low selectivity of only 1.3%, that enables us to skip transferring parts of the input data. In contrast, predication loads all data and is thus bounded by the interconnect bandwidth.

Overall, NVLink 2.0 significantly narrows the gap between the CPU and the GPU for computationally light workloads, and enables the GPU to process large data volumes.

7.2.5 Probe-side Scaling. We analyze the effect of scaling the probe-side relation on join throughput in Figure 16. We use workload C with 16-byte tuples, and increase the probe-side's cardinality from 128–8196 million tuples (1.9–122 GiB). We store the base relations in CPU memory, and the hash table in GPU memory.

Observations. We notice that the throughput of NVLink 2.0 is 3–6× faster than PCI-e 3.0 and 3.2–7.3× faster than the CPU baseline. Throughput improves with larger data due to the changing build-to-probe ratio, that we investigate in detail in Section 7.2.7. In contrast, the throughput of PCI-e 3.0 remains constant, because of the transfer bottleneck. Thus, PCI-e 3.0 cannot outperform the CPU baseline.

Overall, we are able to process data volumes larger than the GPU's memory capacity at a faster rate than the CPU.

7.2.6 Build-side Scaling. In Figure 17, we scale the hash table size up to 2× the GPU memory capacity. The total data size reaches up to 91.5 GiB, counting both base relations plus the hash table. While scaling, we examine the effect of hash table placement strategies (see Section 5.3). We use

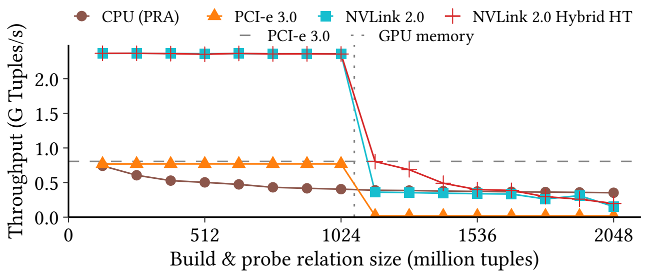


Figure 17: Scaling the build-side relation.

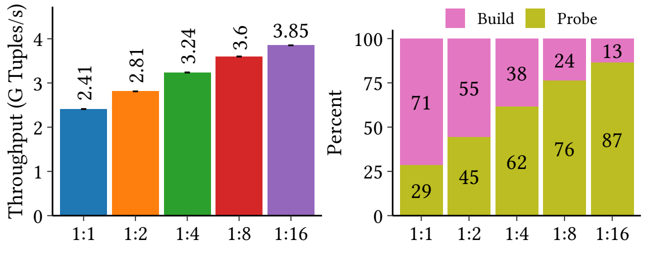
workload C with 16-bytes tuples and increase the cardinality of both base relations.

PCI-e 3.0. We note that throughput reaches 0.77 G Tuples/s as long as the hash table can be stored in GPU memory, which is up to 1.9× faster than the CPU baseline. For hash tables that are larger, throughput declines by 97% to 0.02 G Tuples/s, which is 20× slower than the CPU baseline.

NVLink 2.0. Our first observation is that throughput is 3× higher than PCI-e 3.0 and 3–5.8× higher than the CPU baseline for in-GPU hash tables. Although throughput declines by 85% for out-of-core hash tables, performance remains 8–18× higher than PCI-e 3.0. Although NVLink 2.0 is slower than the CPU baseline for the out-of-core hash table, NVLink 2.0 remains within 13% of the CPU.

NVLink 2.0 with Hybrid Hash Table. We notice that storing parts of the hash table in GPU memory achieves a speedup of 1–2.2× over only NVLink 2.0, despite facing a uniform foreign key distribution. We summarize that NVLink 2.0 helps to achieve higher out-of-core throughput than PCI-e 2.0, and that throughput degrades gracefully, instead of riding over a performance cliff when the hash table is larger than the GPU's memory capacity.

7.2.7 Build-to-probe Ratios. In Figure 18, we quantify the impact of different build-to-probe ratios on join throughput. We use workload C with 16-byte tuples, and increase S with a $|R|$-to-$|S|$ ratio from 1:1 up to 1:16 (up to 2 GiB $\bowtie$ 30.5 GiB). We store the base relations in CPU memory, and the hash table in GPU memory.



(a) Performance. **(b) Time Breakdown.**

Figure 18: Different build-to-probe ratios on NVLink.

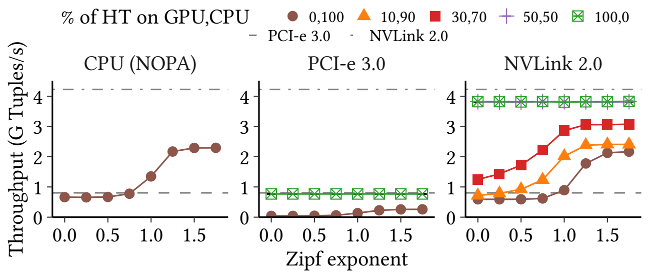


Figure 19: Join performance when the probe relation follows a Zipf distribution.

Observations. The build phase takes 71% of the time, and is thus 45% slower than the probe phase. The impact is most visible for 1:1 ratios. For larger ratios, the build-side takes up a smaller proportion of time, which makes the join faster. We are able to observe these differences because NVLink 2.0 eliminates the transfer bottleneck for this use-case.

7.2.8 Data Skew. We explore a join on data skewed with a Zipf distribution in Figure 19. We use workload A (34 GiB), but skew S with Zipf exponents between 0–1.75. With an exponent of 1.5, there is a 97.5% chance of hitting one of the top-1000 tuples. Thus, increasing data skew tends to increase cache hits. To show the effect of caching, we place the hash table in CPU memory, in GPU memory, and in a hybrid hash table with a varying CPU-to-GPU memory ratio.

Observations. We observe that higher skew leads to a higher throughput of 3.5×, 3.6×, and 6.1× for the CPU, NVLink 2.0, and PCI-e 3.0, respectively. This effect is not present for hash tables in GPU memory, as transferring the base relations from CPU memory is the bottleneck. Thus, we see throughput increase with the hybrid hash table.

7.2.9 Join Selectivity. We evaluate the effect of join selectivity on join throughput in Figure 20. We vary the selectivity of Workload A (34 GiB) from 0–100% by randomly selecting a subset of R. We show the performance of in-GPU and out-of-core hash table placement, and compare the GPU against an IBM POWER9 CPU running the same NOPA join variant.

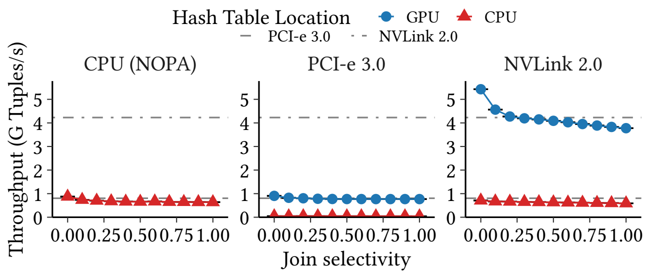
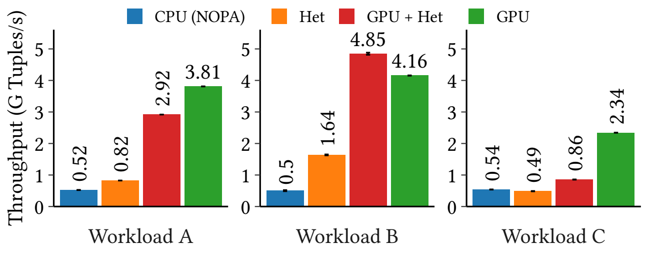
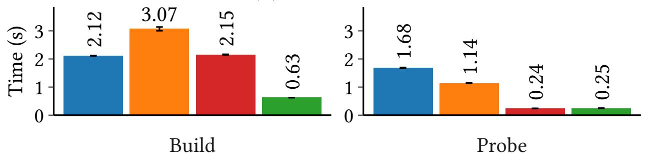


Figure 20: The effect of join selectivity on throughput.



(a) Performance.



(b) Time per join phase in workload C (scaled).

Figure 21: Cooperative CPU and GPU join. *Het* uses a shared hash table in CPU memory, whereas *GPU + Het* uses private hash tables in processor-local memory.

Observations. Our measurements show that join throughput decreases with higher selectivity. The decrease is largest at 30% for NVLink 2.0 with a GPU memory hash table. In contrast, PCI-e 3.0 slows down by only 7% with a hash table in CPU memory. We notice that both interconnects achieve throughput higher than the calculated bandwidth would suggest. This is because only the join key is necessary to establish a match. If there is no match, the value is not accessed. However, if there are one or more matches, the whole cache line is loaded. In effect, at 10% selectivity, 81.5% of values are loaded, resulting in a throughput drop.

Overall, NVLink 2.0 with an out-of-core hash table achieves similar performance to the CPU, and up to 6× better performance with a GPU-local hash table.

7.2.10 CPU/GPU Co-processing Scale-up. In Figure 21(a), we show the join throughput when scaling up the join to a CPU and a GPU using the cooperative Het and GPU + Het strategies described in Section 6.2. We use the workloads in Table 2, that have a size up to 34 GiB. We drill down into the individual join phases of workload C in Figure 21(b) to gain more insights. As the CPU we use an IBM POWER9, and execute the same NOPA algorithm that we run on the GPU. We store the hash table in CPU memory for the CPU and Het execution strategies, and in GPU memory for the GPU and GPU + Het strategy. Note that in GPU + Het, we copy the hash table to CPU memory for the probe phase. We store the base relations in CPU memory for all strategies.

Workload A. We observe that the Het, Het + GPU, and GPU execution strategies run faster than the CPU strategy by 1.5×, 5.6×, and 7.3×, respectively. Adding a GPU always increases throughput, and the GPU without the CPU achieves the highest throughput. The GPU-only strategy is faster than both heterogeneous strategies.

Workload B. We see that Het achieves 3.2×, Het + GPU 9.7×, and GPU 8.32× higher throughput than the CPU-only strategy. Like in workload A, all strategies that use a GPU achieve higher throughput than the CPU-only strategy. The cooperative GPU + Het strategy outperforms the GPU-only strategy by 16%.

Workload C. We notice that the Het strategy is within 10% of the CPU strategy, whereas GPU + Het and GPU are 1.6× and 4.3× faster.

Time per Join Phase in Workload C. To understand why the GPU-only strategy often outperforms the heterogeneous strategies, we investigate the join phases individually.

In the build phase, we observe that two processors (Het) are slower than one processor (all others). The GPU-only strategy is faster than GPU + Het, because the latter first builds the hash table on the GPU, and then synchronously copies it to CPU memory.

In the probe phase, we notice that adding a GPU to the CPU increases performance, but that the GPU by itself is fastest. We observe that a processor-local hash table increases throughput (Het vs. GPU + Het), and that transitioning from a CPU-only to a CPU/GPU solution (Het and GPU + Het) decreases processing time.

Overall, using a GPU always achieves the same or better throughput than the CPU-only strategy, and never decreases throughput. However, the GPU-only strategy achieves the best throughput for most of our workloads.

8 DISCUSSION

In this section, we discuss the key insights that we obtained from our fast interconnects characterization (Section 3) and data processing evaluation (Section 7).

(1) GPUs have high-bandwidth access to CPU memory. We observed that GPUs can load data from CPU memory with bandwidth similar to the CPU. Thus, offloading data processing on GPUs becomes viable even when the data is stored in CPU memory.

(2) GPUs can efficiently process large, out-of-core data. A direct consequence of (1) is that transfer is no longer a bottleneck for complex operators. We have shown speedups of up to 6× over PCI-e 3.0 for hash joins operating on a data structure in GPU memory. In these cases, performance is limited by other factors, e.g., computation or GPU memory.

(3) GPUs are able to operate on out-of-core data structures, but should use GPU memory if possible. In our evaluation, we showed up to 18× higher throughput with NVLink 2.0 than with PCI-e 3.0. Despite this speedup, operating within GPU memory is still 6.5× faster compared to transferring data over NVLink 2.0. However, for hash tables

up to 1.8× larger than GPU memory, we achieved competitive or better performance than an optimized CPU radix join by caching parts of the hash table in GPU memory.
(4) Scaling-up co-processors with CPU + GPU makes performance more robust. A cache-coherent interconnect enables processors to work together efficiently. Processors that cooperate avoid worst-case performance, thus making the overall performance more robust.
(5) Due to cache-coherence, memory pinning is no longer necessary to achieve high transfer bandwidth. We evaluated eight transfer methods, and discovered that fast interconnects enable convenient access to pageable memory without any performance penalty. The benefit is that memory management becomes much simpler because we no longer need staging areas in pinned memory.
(6) Fair performance comparisons between GPUs vs. CPUs have become practical. As a final point, in this paper, we have studied the performance of a GPU and a CPU that load data from the same location (i.e., CPU memory). Fast interconnects enabled us to observe speedups without caching data in GPU memory, although the CPU remains faster in some cases.
Summary. With fast interconnects, GPU acceleration becomes an attractive scale-up alternative that promises large speedups for databases.

9 RELATED WORK

We contrast our paper to related work in this section.
Transfer Bottleneck. The realization that growing data sets do not fit into the co-processor's memory [31, 100] has led recent works to take data transfer costs into account. GPU-enabled databases such as GDB [34], Ocelot [38], CoGaDB/Hawk/HorseQC [13, 14, 26], and HAPE [16, 17], as well as accelerated machine learning frameworks such as SystemML [5] and DAnA [66], are all capable of streaming data from CPU memory onto the co-processor. HippogriffDB [60] and Karnagel et al. [51] take out-of-core processing one step further by loading data from SSDs. The effect of data transfers has also been researched for individual relational operators on GPUs [30, 47, 51, 64, 65, 84, 89, 90]. All of these works observe that transferring data over a PCI-e interconnect is a significant bottleneck when processing data out-of-core. In this paper, we investigate how out-of-core data processing can be accelerated using a faster interconnect.
Transfer Optimization. The success of previous attempts to resolve the transfer bottleneck in software heavily depends on the data and query. Caching data in the co-processor's memory [13, 38, 50] assumes that data are reused, and is most effective for small data sets or skewed access distributions. Data compression schemes [24, 85] must match the data to be effective [20, 23], and trade off computation vs. transfer time. Approximate-and-refine [81] and join pruning using Bloom filters [32] depend on the query's selectivity, and process most of the query pipeline on the CPU. In contrast to these approaches, we show that fast, cache-coherent interconnects enable new acceleration opportunities by improving bandwidth, latency, as well as synchronization cost.
Transfer Avoidance. Another approach is to avoid the transfer bottleneck altogether by using a hybrid CPU-GPU or CPU-FPGA architecture [36, 37, 48, 62, 78]. Hybrid architectures integrate the CPU cores and accelerator into a single chip or package, whereby the accelerator has direct access to CPU memory over the on-chip interconnect [12, 33]. In contrast to these works, we consider systems with discrete GPUs, because discrete co-processors provide more computational power and feature high-bandwidth, on-board memory.
Out-of-core GPU Data Structures. Hash tables [11, 52], B-trees [7, 46, 87, 98], log-structured merge trees [6], and binary trees [53] have been proposed to efficiently access data using GPUs. In contrast, we investigate hash tables in the data management context. We demonstrate concurrent CPU and GPU writes to a shared data structure, and perform locality optimizations. In addition, our approach is more space-efficient than previous shared hash tables [11].
Fast Interconnects. NVLink 1.0 and 2.0 have been investigated previously in microbenchmarks [45, 58, 59, 79, 80] and for deep learning [55, 91, 97]. In contrast to these works, we investigate fast interconnects in the database context. To the best of our knowledge, we are the first to evaluate CPU memory latency and random CPU memory accesses via NVLink 1.0 or 2.0. Raza et al. [82] study lazy transfers and scan sharing for HTAP with NVLink 2.0. In contrast, we conduct an in-depth analysis of fast interconnects.

10 CONCLUSION

We conclude that, on the one hand, fast interconnects enable new use-cases that were previously not worthwhile to accelerate on GPUs. On the other hand, currently NVLink 2.0 represents a specialized technology that has yet to arrive in commodity hardware. Overall, in this work we have made the case that future database research should consider fast interconnects for accelerating workloads on co-processors.

ACKNOWLEDGMENTS

This work was funded by the EU Horizon 2020 programme as E2Data (780245), the DFG priority programme "Scalable Data Management for Future Hardware" (MA4662-5), the German Ministry for Education and Research as BBDC (01IS14013A) and 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] Jasmin Ajanovic. 2009. PCI express 3.0 overview. In *HCS*, Vol. 69. IEEE, New York, NY, USA, 143. https://doi.org/10.1109/HOTCHIPS.2009.7478337

[2] Brian Allison. 2018. Introduction to the OpenCAPI Interface. https://openpowerfoundation.org/wp-content/uploads/2018/10/Brian-Allison.OPF_OpenCAPI_FPGA_Overview_V1-1.pdf. In *OpenPOWER Summit Europe*.

[3] AMD. 2019. AMD EPYC CPUs, AMD Radeon Instinct GPUs and ROCm Open Source Software to Power World's Fastest Supercomputer at Oak Ridge National Laboratory. Retrieved July 5, 2019 from https://www.amd.com/en/press-releases/2019-05-07-amd-epyc-cpus-radeon-instinct-gpus-and-rocm-open-source-software-to-power

[4] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. In *CIDR*.

[5] 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*. 173–182. https://doi.org/10.1145/2688500.2688521

[6] Saman Ashkiani, Shengren Li, Martin Farach-Colton, Nina Amenta, and John D. Owens. 2018. GPU LSM: A Dynamic Dictionary Data Structure for the GPU. In *IPDPS*. 430–440. https://doi.org/10.1109/IPDPS.2018.00053

[7] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martin Farach-Colton, and John D. Owens. 2019. Engineering a high-performance GPU B-Tree. In *PPoPP*. 145–157. https://doi.org/10.1145/3293883.3295706

[8] 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, New York, NY, USA, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[9] 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

[10] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[11] Rajesh Bordawekar and Pidad Gasfer D'Souza. 2018. Evaluation of hybrid cache-coherent concurrent hash table on IBM POWER9 AC922 system with NVLink2. http://on-demand.gputechconf.com/gtc/2018/video/S8172/. In *GTC*. Nvidia.

[12] Dan Bouvier and Ben Sander. 2014. Applying AMD's Kaveri APU for heterogeneous computing. In *HCS*. IEEE, New York, NY, USA, 1–42. https://doi.org/10.1109/HOTCHIPS.2014.7478810

[13] 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

[14] 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

[15] Alexandre Bicas Caldeira. 2018. *IBM power system AC922 introduction and technical overview*. IBM, International Technical Support Organization.

[16] 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

[17] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. In *CIDR*.

[18] Jonathan Corbet. 2015. Making kernel pages movable. *LWN.net* (July 2015). https://lwn.net/Articles/650917/

[19] CXL 2019. *Compute Express Link Specification Revision 1.1*. CXL. https://www.computeexpresslink.org

[20] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *Trans. Database Syst.* 44, 3 (2019), 9:1–9:46. https://doi.org/10.1145/3323991

[21] Deloitte. 2017. Hitting the accelerator: the next generation of machine-learning chips. Retrieved Oct 1, 2019 from https://www2.deloitte.com/content/dam/Deloitte/global/Images/infographics/technologymediatelecommunications/gx-deloitte-tmt-2018-nextgen-machine-learning-report.pdf

[22] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *PVLDB* 12, 12 (2019), 2218–2229.

[23] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB* 9, 12 (2016), 960–971. https://doi.org/10.14778/2994509.2994515

[24] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *PVLDB* 3, 1 (2010), 670–680. https://doi.org/10.14778/1920841.1920927

[25] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the Hidden Cost of RDMA. In *ICDCS*. 553–560. https://doi.org/10.1109/ICDCS.2009.32

[26] 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

[27] Victor Garcia-Flores, Eduard Ayguadé, and Antonio J. Peña. 2017. Efficient Data Sharing on Heterogeneous Systems. In *ICPP*. 121–130. https://doi.org/10.1109/ICPP.2017.21

[28] Gartner. 2019. Gartner Says the Future of the Database Market Is the Cloud. Retrieved Oct 1, 2019 from https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the

[29] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. 2004. Fast Computation of Database Operations using Graphics Processors. In *SIGMOD*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/1007568.1007594

[30] 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

[31] 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, New York, NY, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[32] 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

[33] Prabhat K. Gupta. 2016. Accelerating Datacenter Workloads. In *FPL*. 1–27.

[34] Bingsheng He et al. 2009. Relational query coprocessing on graphics processors. *TODS* 34, 4 (2009).

[35] 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

[36] 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

[37] 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

[38] Max Heimel et al. 2013. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB* 6, 9 (2013), 709–720.

[39] Joel Hestness, Stephen W. Keckler, and David A. Wood. 2014. A comparative analysis of microarchitecture effects on CPU and GPU memory system behavior. In *IISWC*. IEEE, New York, NY, USA, 150–160. https://doi.org/10.1109/IISWC.2014.6983054

[40] IBM 2018. *POWER9 Processor User's Manual Version 2.0*. IBM.

[41] 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.

[42] Intel 2018. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel.

[43] Intel. 2019. Intel Stratix 10 DX FPGA Product Brief. Retrieved Accessed: Oct 2, 2019 from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/solution-sheets/stratix-10-dx-product-brief.pdf

[44] 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. Retrieved Jan 29, 2020 from https://newsroom.intel.com/news-releases/intel-unveils-new-gpu-architecture-optimized-for-hpc-ai-oneapi

[45] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *CoRR* abs/1804.06826 (2018). arXiv:1804.06826

[46] Krzysztof Kaczmarski. 2012. B $^+$ -Tree Optimized for GPGPU. In *OTM*. 843–854. https://doi.org/10.1007/978-3-642-33615-7_27

[47] 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

[48] Kaan Kara, Ken Eguro, Ce Zhang, and Gustavo Alonso. 2018. ColumnML: Column-Store Machine Learning with On-The-Fly Data Transformation. *PVLDB* 12, 4 (2018), 348–361.

[49] 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

[50] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB* 10, 7 (2017), 733–744. https://doi.org/10.14778/3067421.3067423

[51] Tomas Karnagel, René Müller, and Guy M. Lohman. 2015. Optimizing GPU-accelerated group-by and aggregation. In *ADMS*. ACM, New York, NY, USA, 13–24.

[52] Farzad Khorasani, Mehmet E. Belviranli, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *PACT*. IEEE, New York, NY, USA, 63–74. https://doi.org/10.1109/PACT.2015.13

[53] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*. 339–350. https://doi.org/10.1145/1807167.1807206

[54] 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

[55] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter R. Pietzuch. 2019. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB* 12, 11 (2019), 1399–1413. https://doi.org/10.14778/3342263.3342276

[56] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *SIGMOD*. ACM, New York, NY, USA, 555–569. https://doi.org/10.1145/2882903.2882906

[57] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. ACM, New York, NY, USA, 743–754. https://doi.org/10.1145/2588555.2610507

[58] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2019. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *CoRR* abs/1903.04611 (2019). arXiv:1903.04611

[59] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *IISWC*. IEEE, New York, NY, USA, 191–202. https://doi.org/10.1109/IISWC.2018.8573483

[60] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *PVLDB* 9, 14 (2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[61] Yinan Li, Ippokratis Pandis, René Müller, Vijayshankar Raman, and Guy M. Lohman. 2013. NUMA-aware algorithms: the case of data shuffling. In *CIDR*.

[62] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan D. A. Nguyen, and Akash Kumar. 2018. Column Scan Acceleration in Hybrid CPU-FPGA Systems. In *ADMS*. ACM, New York, NY, USA, 22–33.

[63] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *ASPLOS*. ACM, New York, NY, USA, 257–270. https://doi.org/10.1145/3297858.3304043

[64] Clemens Lutz et al. 2018. Efficient k-means on GPUs. In *DaMoN*. ACM, New York, NY, USA, 3:1–3:3. https://doi.org/10.1145/3211922.3211925

[65] 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

[66] 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

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

[68] Frank Mietke, Robert Rex, Robert Baumgartl, Torsten Mehlan, Torsten Hoefler, and Wolfgang Rehm. 2006. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Euro-Par*. 124–133. https://doi.org/10.1007/11823285_13

[69] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. 2014. *Unified memory in CUDA 6.0: a brief overview of related data access and transfer issues*. University of Wisconsin-Madison. TR-2014–09.

[70] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *SIGCOMM*. ACM, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[71] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet D. Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchý. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artif. Intell. Rev.* 52, 1 (2019), 77–124. https://doi.org/10.1007/s10462-018-09679-z

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

[73] 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.

[74] Nvidia 2018. *CUDA C Programming Guide*. Nvidia. http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf PG-02829-001_v10.0.

[75] Nvidia 2019. *CUDA C Best Practices Guide*. Nvidia. https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf DG-05603-001_v10.1.

[76] Nvidia 2019. *Tuning CUDA Applications for Pascal*. Nvidia. https://docs.nvidia.com/cuda/pdf/Pascal_Tuning_Guide.pdf DA-08134-001_-v10.1.

[77] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. 2011. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *ReConFig*. IEEE, New York, NY, USA, 80–85. https://doi.org/10.1109/ReConFig.2011.4

[78] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the Latency of Data Processing Pipelines Through FPGA based Hardware Acceleration. *PVLDB* 13, 1 (2019), 71–85. http://www.vldb.org/pvldb/vol13/p71-owaida.pdf

[79] C. Pearson, I. Chung, Z. Sura, W. Hwu, and J. Xiong. 2018. NUMA-aware Data-transfer Measurements for Power/NVLink Multi-GPU Systems. In *IWOPH*. Springer, Heidelberg, Germany.

[80] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2019. Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects. In *ICPE*. ACM, New York, NY, USA, 209–218. https://doi.org/10.1145/3297663.3310299

[81] Holger Pirk, Stefan Manegold, and Martin L. Kersten. 2014. Waste not…Efficient co-processing of relational data. In *ICDE*. IEEE, New York, NY, USA, 508–519.

[82] 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*.

[83] 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

[84] 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

[85] Eyal Rozenberg and Peter A. Boncz. 2017. Faster across the PCIe bus: a GPU library for lightweight decompression: including support for patched compression schemes. In *DaMoN*. ACM, New York, NY, USA, 8:1–8:5. https://doi.org/10.1145/3076113.3076122

[86] 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

[87] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2016. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD*. 1523–1538. https://doi.org/10.1145/2882903.2882918

[88] Arnon Shimoni. 2017. Which GPU database is right for me? Retrieved Oct 1, 2019 from https://hackernoon.com/which-gpu-database-is-right-for-me-6ceef6a17505

[89] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious Hash-Joins on GPUs. In *ICDE*. IEEE, New York, NY, USA.

[90] 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.

[91] Nathan R. Tallent, Nitin A. Gawande, Charles Siegel, Abhinav Vishnu, and Adolfy Hoisie. 2017. Evaluating On-Node GPU Interconnects for Deep Learning Workloads. In *PMBS@SC*. 3–21. https://doi.org/10.1007/978-3-319-72971-8_1

[92] Top500. 2019. Top500 Highlights. Retrieved Mar 16, 2020 from https://www.top500.org/lists/2019/11/highs/

[93] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R. Gross. 2015. RStore: A Direct-Access DRAM-based Data Store. In *ICDCS*. 674–685. https://doi.org/10.1109/ICDCS.2015.74

[94] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. Ph.D. Dissertation. UC Berkeley.

[95] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *MICRO*. IEEE/ACM, New York, NY, USA, 107–118. https://doi.org/10.1109/MICRO.2012.19

[96] Xilinx 2017. *Vivado Design Suite: AXI Reference Guide*. Xilinx. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf UG1037 (v4.0).

[97] Rengan Xu, Frank Han, and Quy Ta. 2018. Deep Learning at Scale on Nvidia V100 Accelerators. In *PMBS@SC*. 23–32. https://doi.org/10.1109/PMBS.2018.8641600

[98] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. 2019. Harmonia: A high throughput B+tree for GPUs. In *PPoPP*. 133–144. https://doi.org/10.1145/3293883.3295704

[99] Ke Yang, Bingsheng He, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, Pedro V. Sander, and Jiaoying Shi. 2007. In-memory grid files on graphics processors. In *DaMoN*. ACM, New York, NY, USA, 5. https://doi.org/10.1145/1363189.1363196

[100] 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

[101] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. 2019. Adaptive memory-side last-level GPU caching. In *ISCA*. ISCA, Winona, MN, USA, 411–423. https://doi.org/10.1145/3307650.3322235

[102] Tianhao Zheng, David W. Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *HPCA*. IEEE, New York, NY, USA, 345–357. https://doi.org/10.1109/HPCA.2016.7446077