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

Xinyun Cao, Randy Collado-Cedeno, Meng-Heng Lee, Shaolin Xie

Goal: Scale GPU-based data processing to arbitrary data size

Three Fundamental Limitations

- Low interconnect bandwidth
- Small GPU memory capacity
- Coarse-grained cooperation of CPU and GPU

Solutions

- New fast Interconnects, mainly NVLink 2.0

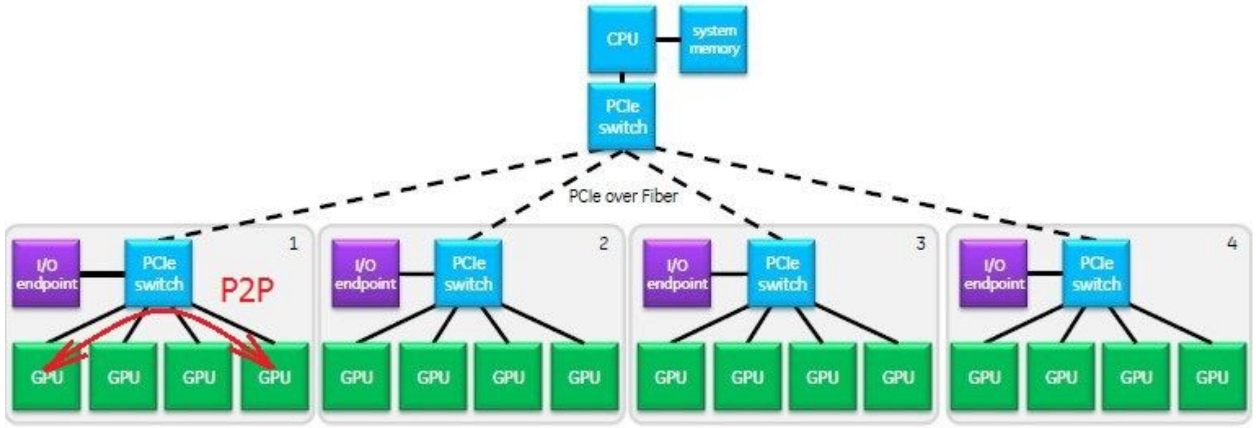
- New co-processing strategy

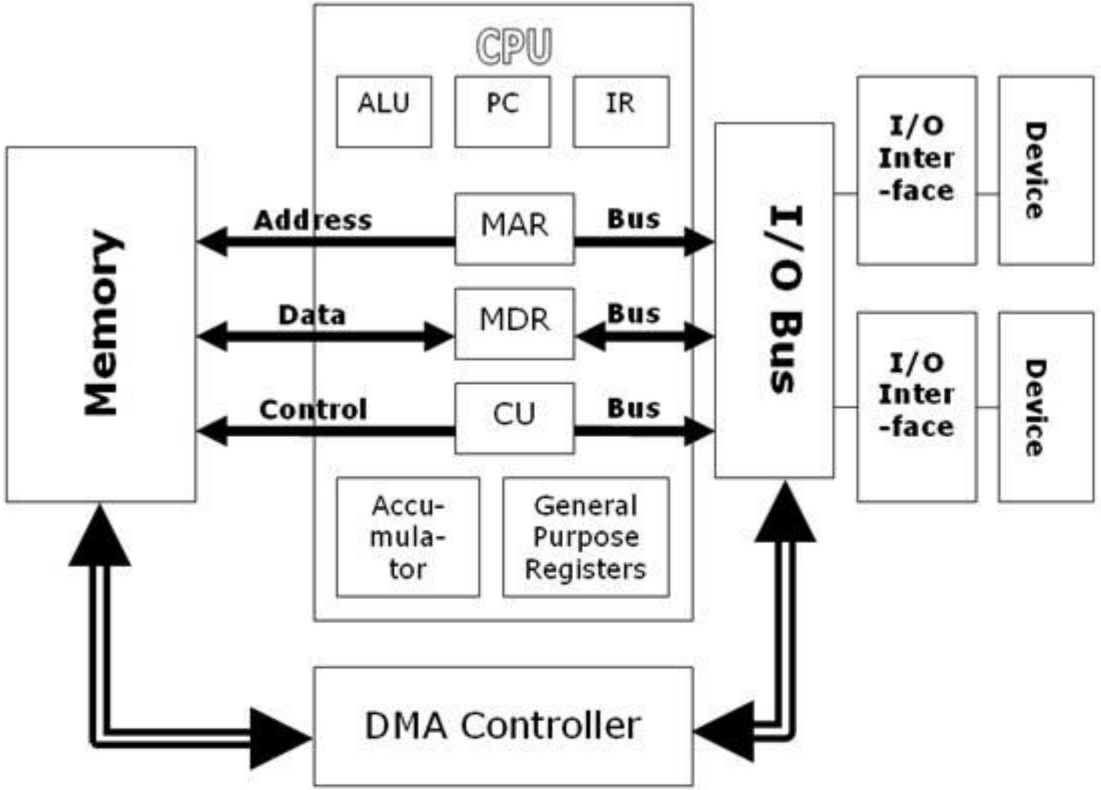
PCI-e 3.0

Transfer Primitives

- Memory-Mapped I/O (MMIO)

- Direct Memory Access (DMA)




CPU
ALU
PC
IR
Memory
Address
MAR
Bus
Data
MDR
Bus
Control
CU
Bus
Accu-mula-tor
General Purpose Registers
I/O Bus
I/O Inter-face
Device
I/O Inter-face
Device
DMA Controller

NVLink 2.0

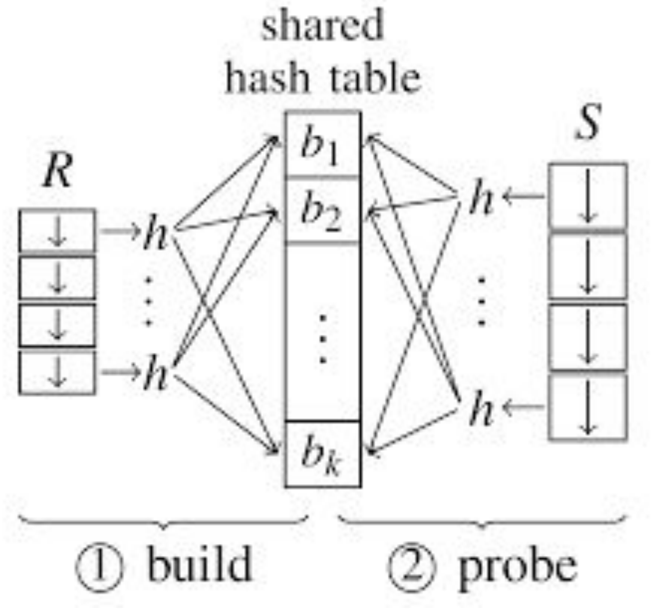
- Mesh Topology
- MMIO and DMA
- **Allows direct access to pageable CPU memory**

No-Partitioning Hash Join

- Loading **base relations** from **main memory** requires **high bandwidth**

- Scaling the **hash table beyond GPU memory** requires **low latency**

- Sharing the **hash table** between two or more processors requires **cache-coherence**



How does NVLink Compare?

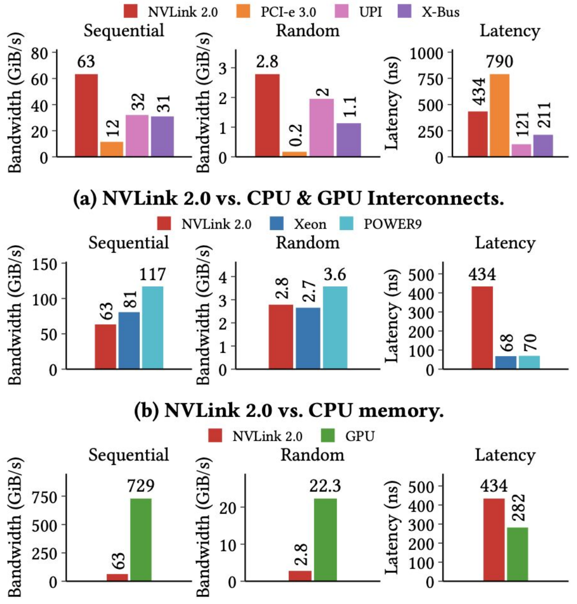
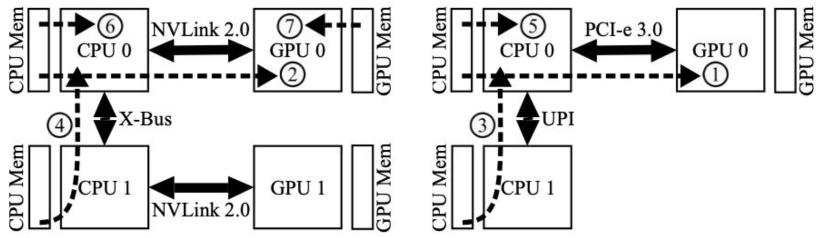


Figure 3: Bandwidth and latency of memory reads on IBM and Intel systems with Nvidia GPUs. Compare to data access paths shown in Figure 4.

- GPU Interconnects
  - PCI-e 3.0 (1)
  - NVLink 2.0 (2)
- CPU Interconnects
  - Intel Xeon Ultra Path Interconnect (UPI) (3)
  - IBM POWER9 X-Bus (4)
- CPU Memory
  - Intel XEON (5)
  - IBM POWER9 (6)
- GPU Memory
  - Nvidia V100 (7)



(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.

Efficient Data Transfer Between CPU and GPU

Push based transfer methods

- **pageable copy** - transfers data directly from any location in pageable memory via the CUDA API's (`cudaMemcpyAsync`)

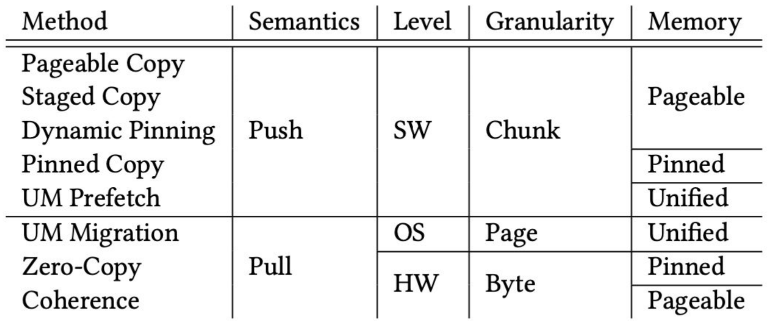
- **pinned copy** - similar to pageable copy except data is constrained to **pinned** memory

- **staged copy** - copy chunk of data from pageable memory into pinned memory buffer

- **dynamic pinning** - pinning pre-existing pageable memory ad hoc - avoids copying operation in CPU memory

- **UM Prefetch** - prefetch region of unified memory if we know data access pattern beforehand

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

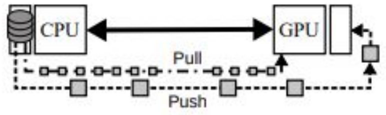


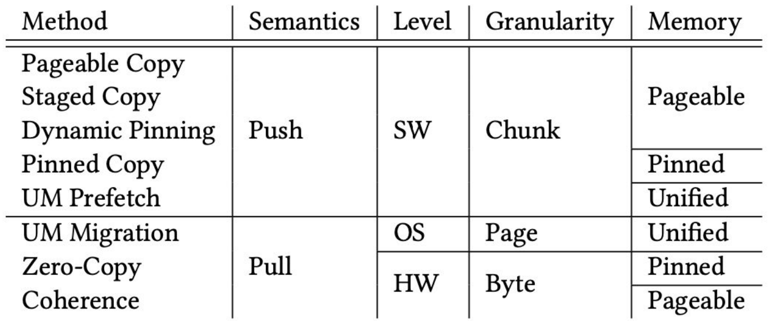
Figure 5: Push- vs. pull-based data transfer methods.

Efficient Data Transfer Between CPU and GPU

Pull based transfer methods

- **Unified Memory Migration** - migrate memory pages to GPU on page access. Database MUST explicitly allocate Unified Memory to store data to utilize this.
- **Zero-Copy** - use Unified Virtual Addressing to directly access CPU's pinned memory during GPU execution
- **Coherence** - new method offered by NVLink 2.0. GPU can access pageable CPU directly

Table 1: An overview of GPU transfer methods.



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

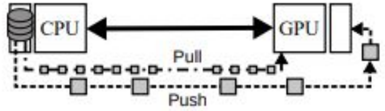


Figure 5: Push- vs. pull-based data transfer methods.

Overcome L2: Small GPU memory capacity

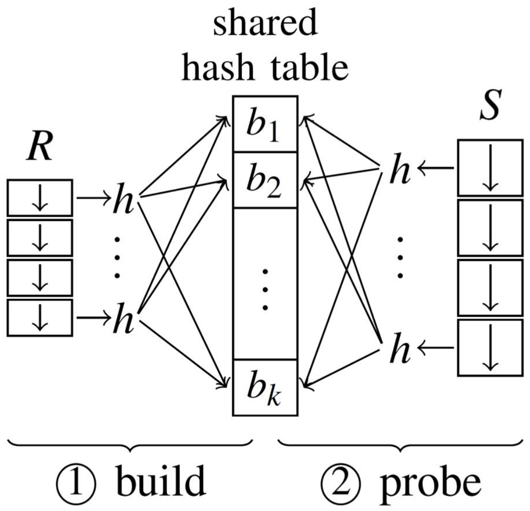
Tool: No-Partitioning Hash Join

- Build phase:
  - Build the shared hash table with build-side relation R

Random memory access - requires low-latency interconnect to access the hash table in CPU, or large enough GPU memory to store the table

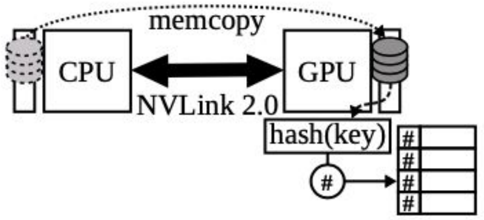
- Probe phase:
  - Probe the table with tuples from probe-side relation S

High demand on interconnects' bandwidth



Non-scalable Baseline Join vs Probe-side Scalable Join

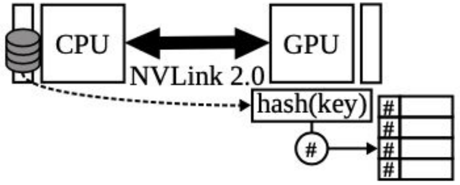
Baseline Join



- Copy build-side relation R to GPU
- Build hash table in GPU
- Evict R
- Copy probe-side relation S to GPU
- Probe the hash table and emit the join results

– Limited by GPU’s memory capacity

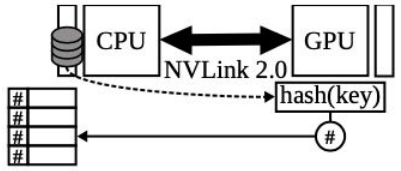
Probe-side Scalable Join



- Pull R tuples on-demand from CPU
- Build hash table in GPU
- Pull S tuples on-demand from CPU
- Probe the hash table

* Uses **pull-based transfer** with **Coherence** method instead of Zero-copy method to enable GPU to access any memory location in pageable memory (directly access CPU)

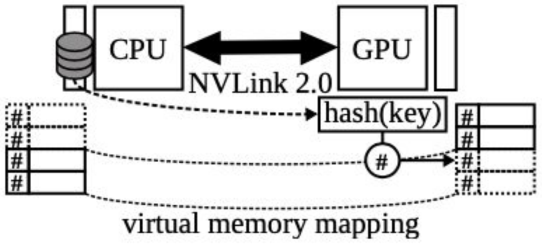
Scaling the Build-side of the Join



- Pull R tuples on-demand from CPU memory
- Build hash table in GPU
- **Store the hash table in CPU**
- Pull S tuples on-demand from CPU
- Probe the hash table

+ Not constrained by GPU memory

New Design: Hybrid Hash Table



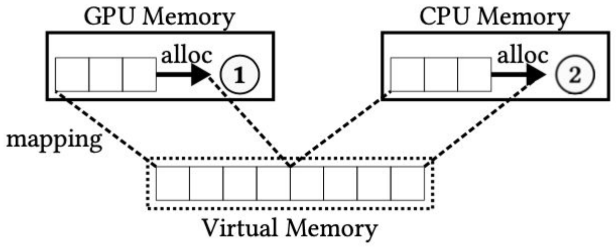
Advantages:

+ Access performance degrades gracefully when the hash table's size is increased
+ Easily integrated into existing databases - hash join algorithms remains the same

Replaces hash table in CPU memory:

- Data in CPU memory and hash table spills from GPU memory into CPU memory
- Use virtual memory to combine CPU and GPU into single, continuous array - fast interconnect integrate the GPU into a system-wide address space
- Map physical CPU pages next to GPU pages in the address space

Algorithm: Allocating Hybrid Hash Table



ALGORITHM

By default:

if hash table is smaller than available GPU memory, allocate the entire hash table in GPU memory

Else:

allocate in the CPU that is the nearest to the GPU; if CPU doesn’t have sufficient memory, search for the next-nearest CPU until we fit the entire hash table

Overcome L3: Coarse-grained Cooperation of CPU and GPU

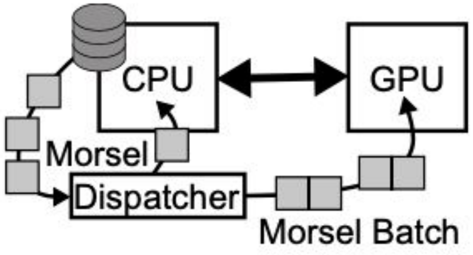
Task Scheduling

Task scheduler

- ensure all processors deliver their highest throughput

CPU-oriented approach for GPUs:

- All cores work concurrently on the same hash table
- Cores request fixed sized morsels from a central dispatcher
- Dispatches one morsel at a time

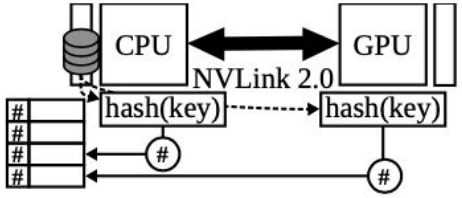


CPU+GPU heterogeneous scheduling approach:

- Dispatch batches of morsels to the GPU
  - GPU has higher processing rate but also higher latency in scheduling work
  - Amortizing the latency of launching a GPU kernel over more data

Hash Table Placement

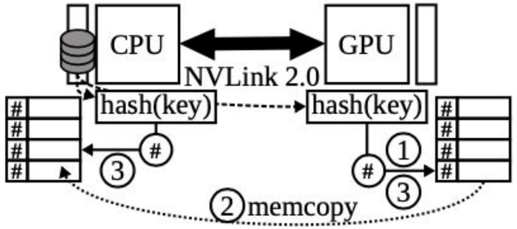
Het Strategy



One globally shared hash table:

- Same as build-side scaling method - stores hash table in CPU
- Why?
  - Avoid slowing down CPU processing through remote GPU memory accesses
  - GPU accesses CPU much faster than CPU accessing GPU

GPU+Het Strategy



Multiple per-processor hash tables(for small hash tables):

- One processor builds the hash table in local processor memory
- Copy the table to all other processors
- Execute probe phase using heterogeneous scheduling strategy

Multiple GPU Hash Table Placement

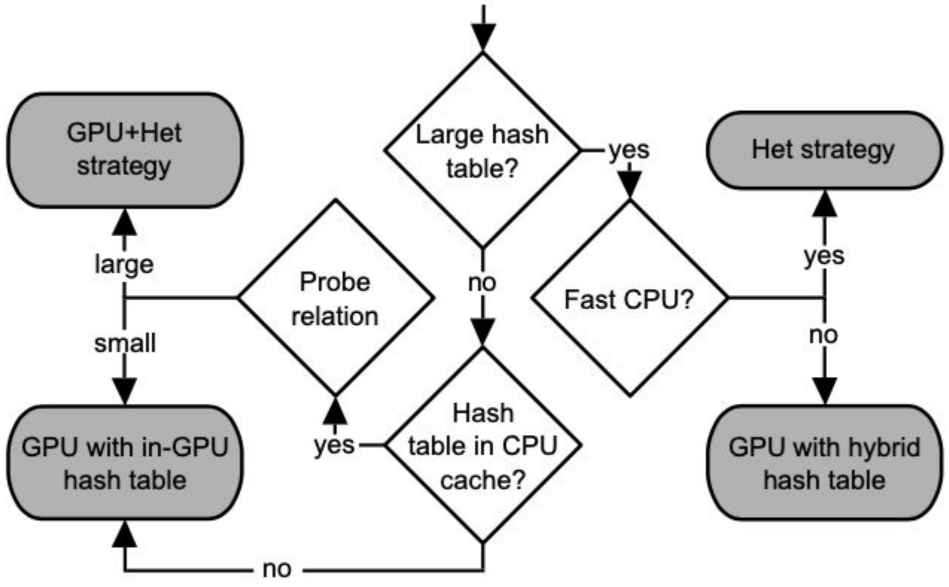
For large hash tables, we distribute the table by interleaving the pages over all GPUs

- Multi-GPU systems can distribute the hash tables over multiple GPUs

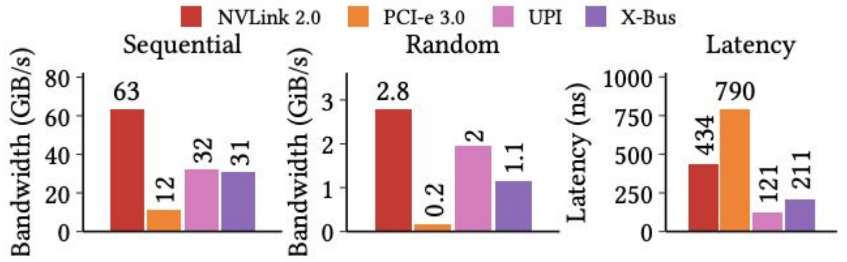
Advantages:

+ Avoids computational skew
+ Frees CPU memory bandwidth by loading the base relation
+ Utilizes the full bi-directional bandwidth of fast interconnects vs uni-directional bandwidth in Het Strategy

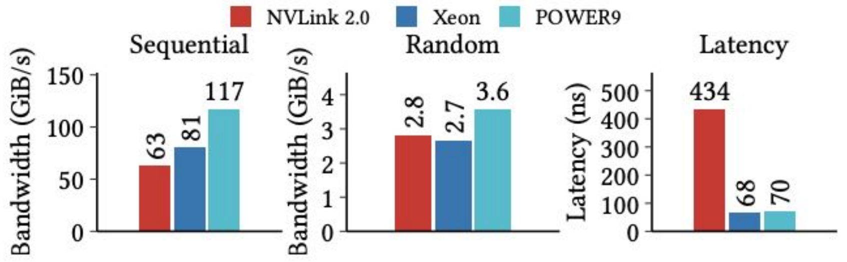
Hash Table Placement Strategy



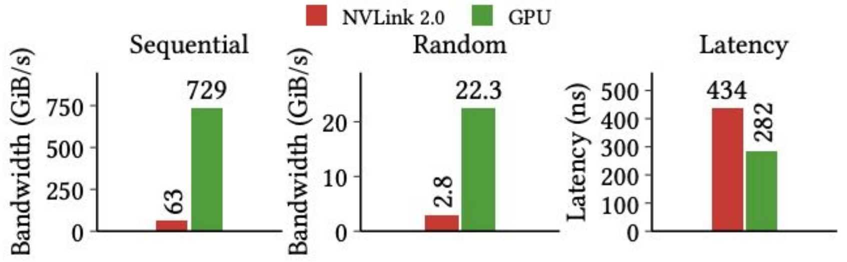
Evaluation and Experiments



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



(b) NVLink 2.0 vs. CPU memory.



(c) NVLink 2.0 vs. GPU memory.

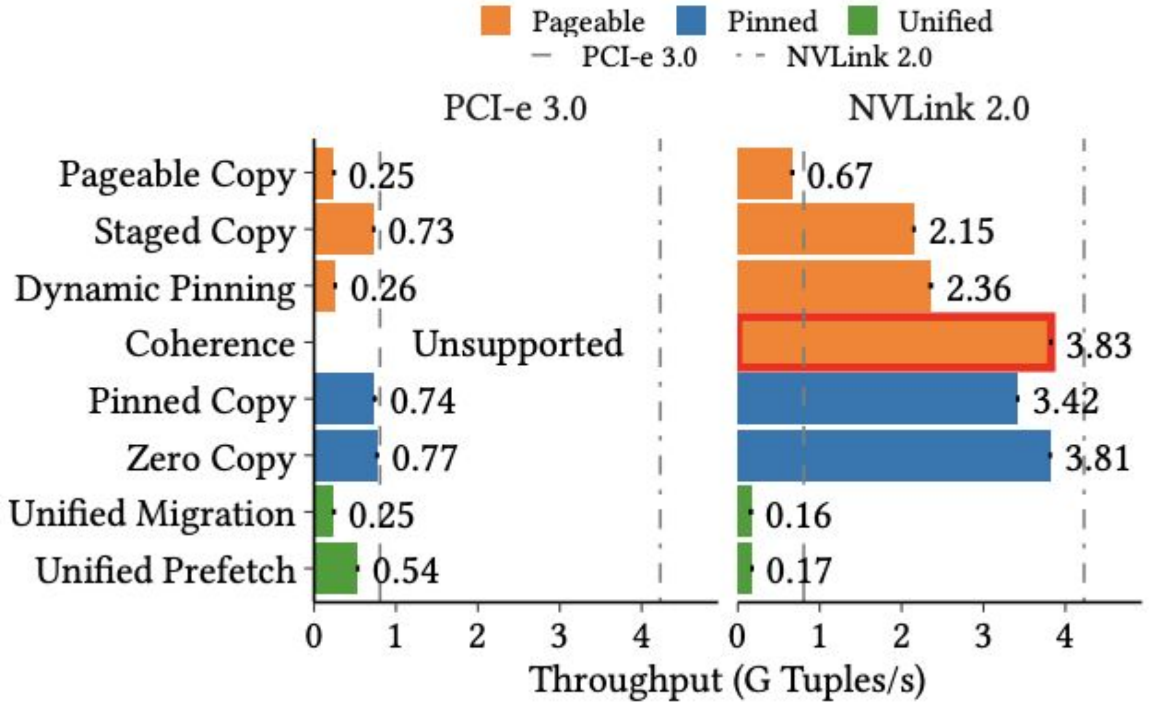
Measurements are conducted on 4-byte read accesses on 1 GiB of data.

Evaluation & Experiments: Memory Access Throughput

- NVLink 2.0 is assessed by conducting sequential and random 4-byte memory reads on 1GiB of data and measuring the effective bandwidth versus other system interconnects such as Intel UPI and IBM XPower (CPU-to-CPU) and PCI-e 3.0 (CPU-to-GPU)

- Results showed a 5x increase in bandwidth vs. PCI-e 3.0 in sequential workloads, and 2x faster native CPU interconnects.

- In random workloads, NVLink 2.0 showed a 14x increase in bandwidth compared to PCI-e 3.0, and a max of 35% improvement over native CPU interconnects



Throughput measured as (|R| + |S|) / runtime

Evaluation & Experiments: Cache-Coherence

- 8 distinct methods of data transfer between CPU and GPU were determined by the authors, ranging from fetching pages directly from the CPU memory to preemptive memory moves

- NVLink 2.0 is proven to drastically increase throughput with all transfer methods tested, except for access methods based on Unified Memory
    - It is thought that this discrepancy might be due to an unoptimized driver in IBM’s Power9 architecture.

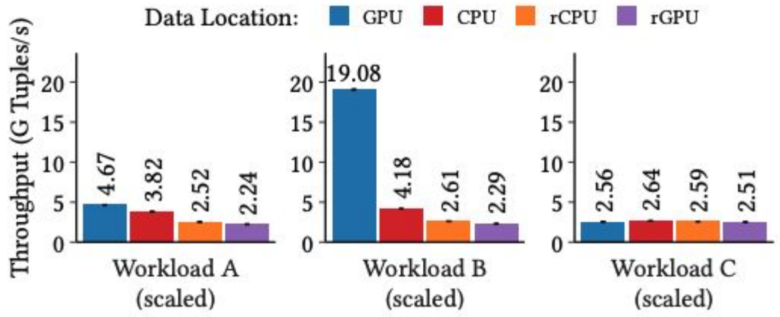


Figure 13: Join performance of the GPU when the base relations are located on different processors, increasing the number of interconnect hops from 0 to 3.

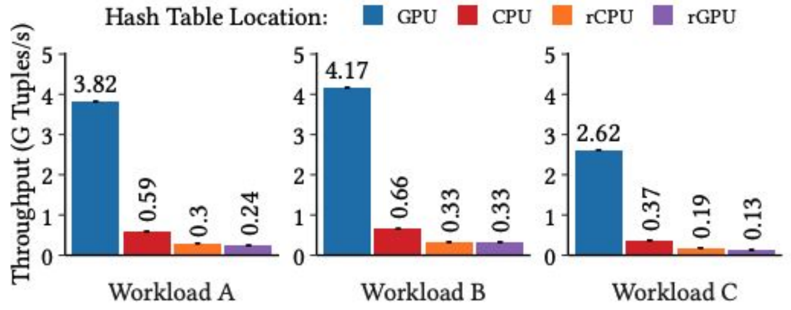


Figure 14: Join performance of the GPU when the hash table is located on different processors, increasing the number of interconnect hops from 0 to 3.

Evaluation & Experiments: Join Performance

- Join performance was measured considering relation locality and hash-table locality, as so to measure the effects of multiple layers of indirection on NVLink 2.0.

- Both figures show that the majority of the loss of throughput comes from the transition from 0 interconnect hops to 1, remaining stable with little relative performance loss as the number of interconnect hops increases.

- This is due to the lower throughput of the X-Bus interconnect, creating a bottleneck, showing the stability of NVLink 2.0 over multiple hops.

Conclusions

Fast interconnects integrate GPUs tightly with CPUs and significantly reduce the data transfer bottleneck

With fast interconnects, GPU acceleration becomes an attractive scale-up alternative that promises large speedups for databases.

Technical Question:

What are the disadvantages of current interconnect and the benefits of NVLink 2.0 besides speed?