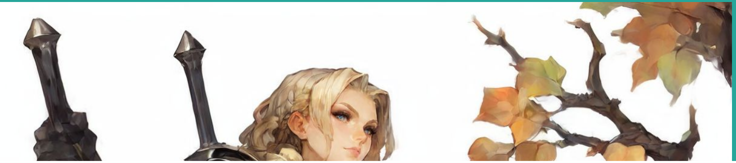
Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects

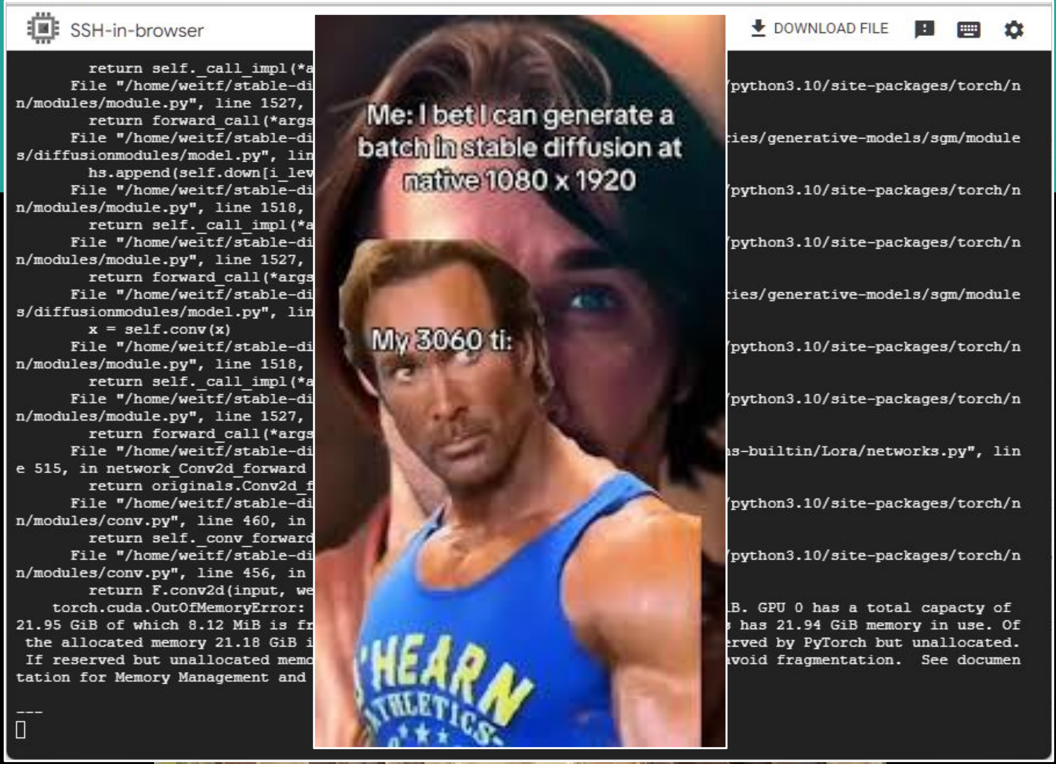
Teona Bagashvili, Tongfan Wei

Why should we process data on GPU?

1080

1920




SSH-in-browser
DOWNLOAD FILE
return self._call_impl(*a
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/module.py", line 1527,
return forward_call(*args
File "/home/weitf/stable-di ies/generative-models/sgm/module
s/diffusionmodules/model.py", lin
hs.append(self.down[i_lev
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/module.py", line 1518,
return self._call_impl(*a
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/module.py", line 1527,
return forward_call(*args
File "/home/weitf/stable-di ies/generative-models/sgm/module
s/diffusionmodules/model.py", lin
x = self.conv(x)
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/module.py", line 1518,
return self._call_impl(*a
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/module.py", line 1527,
return forward_call(*args
File "/home/weitf/stable-di s-builtin/Lora/networks.py", lin
e 515, in network_Conv2d_forward
return originals.Conv2d_f
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/conv.py", line 460, in
return self._conv_forward
File "/home/weitf/stable-di /python3.10/site-packages/torch/n
n/modules/conv.py", line 456, in
return F.conv2d(input, we
torch.cuda.OutOfMemoryError: iB. GPU 0 has a total capacity of
21.95 GiB of which 8.12 MiB is fr s has 21.94 GiB memory in use. Of
the allocated memory 21.18 GiB i erved by PyTorch but unallocated.
If reserved but unallocated memo avoid fragmentation. See documen
tation for Memory Management and

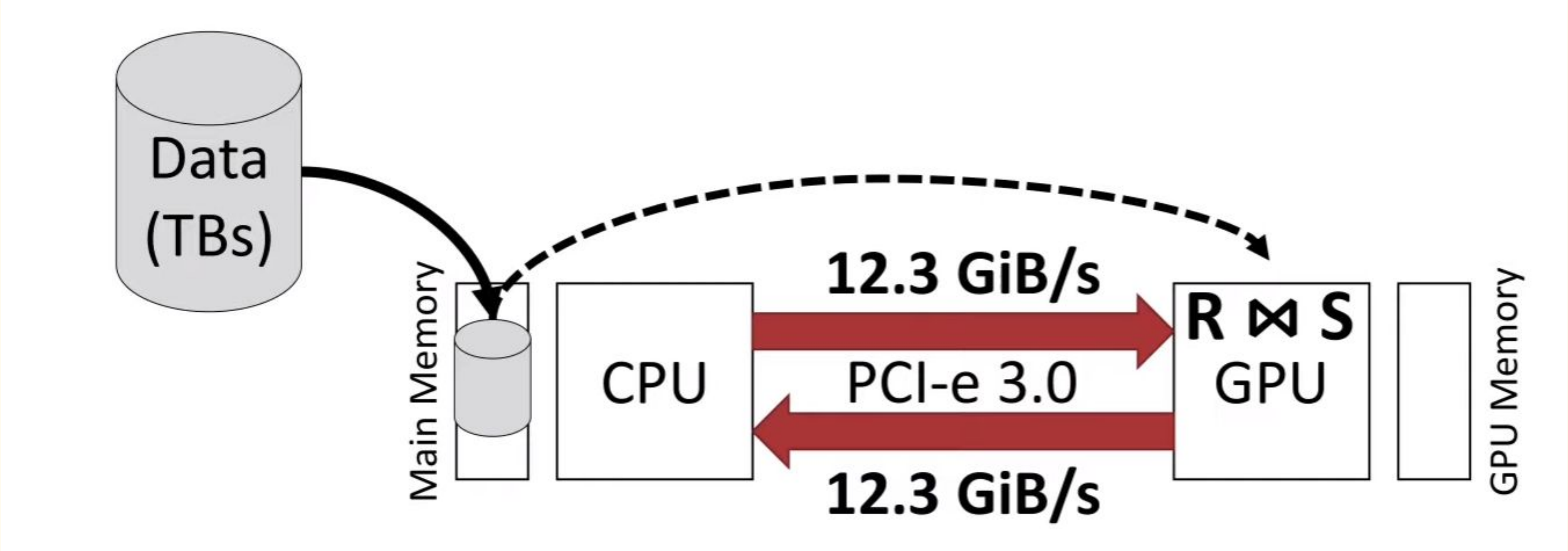
Me: I bet I can generate a batch in stable diffusion at native 1080 x 1920
My 3060 ti:

What is the goal?

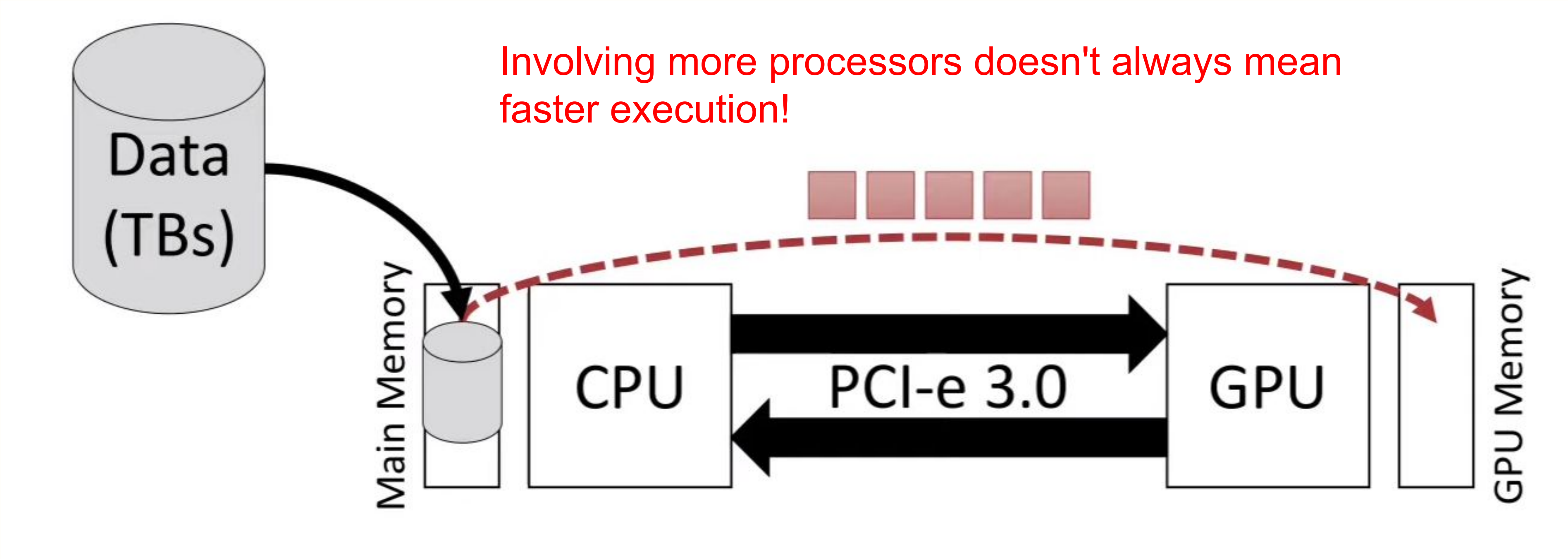
Scale GPU-accelerated data management to arbitrary data volumes!

Issues?

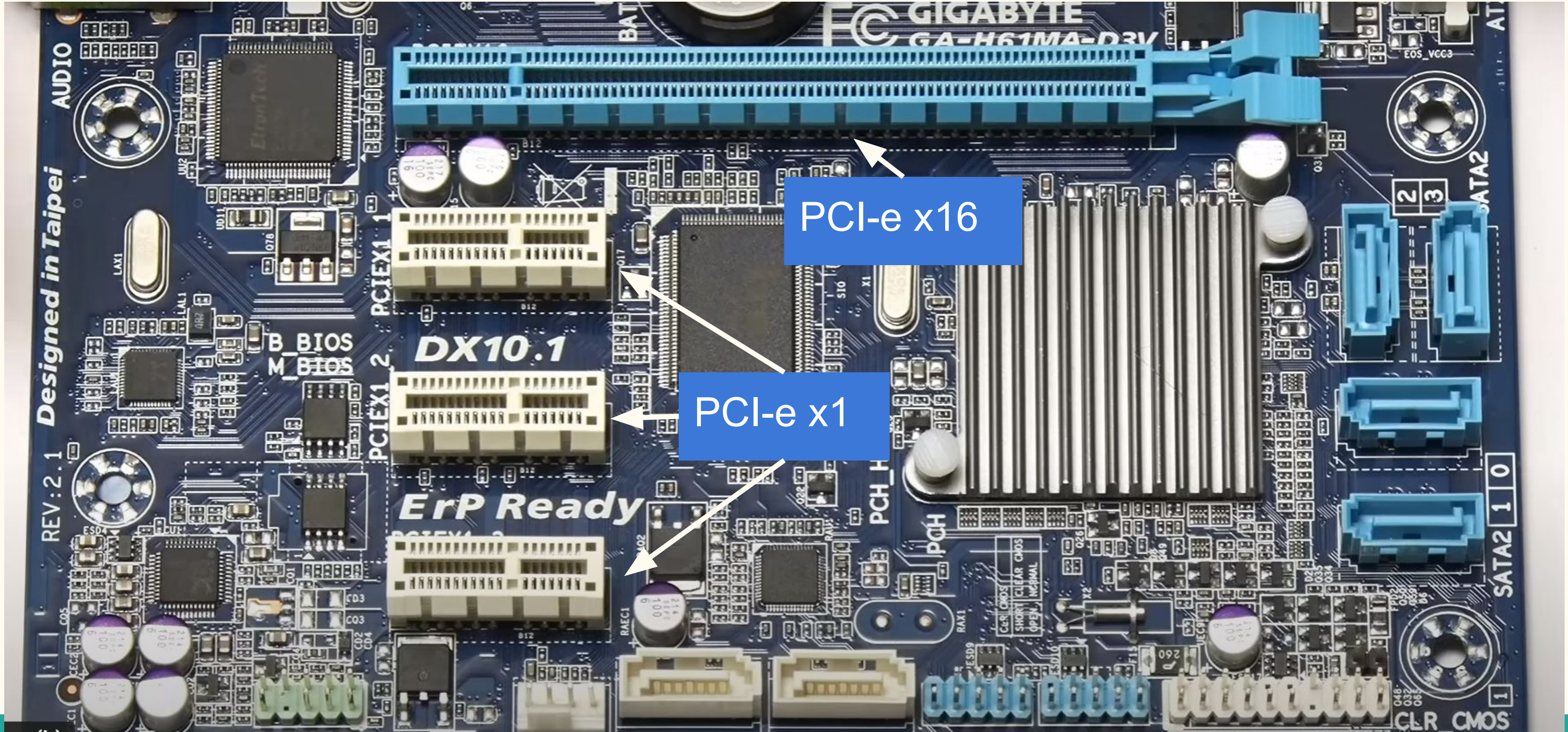
Transfer Bandwidth



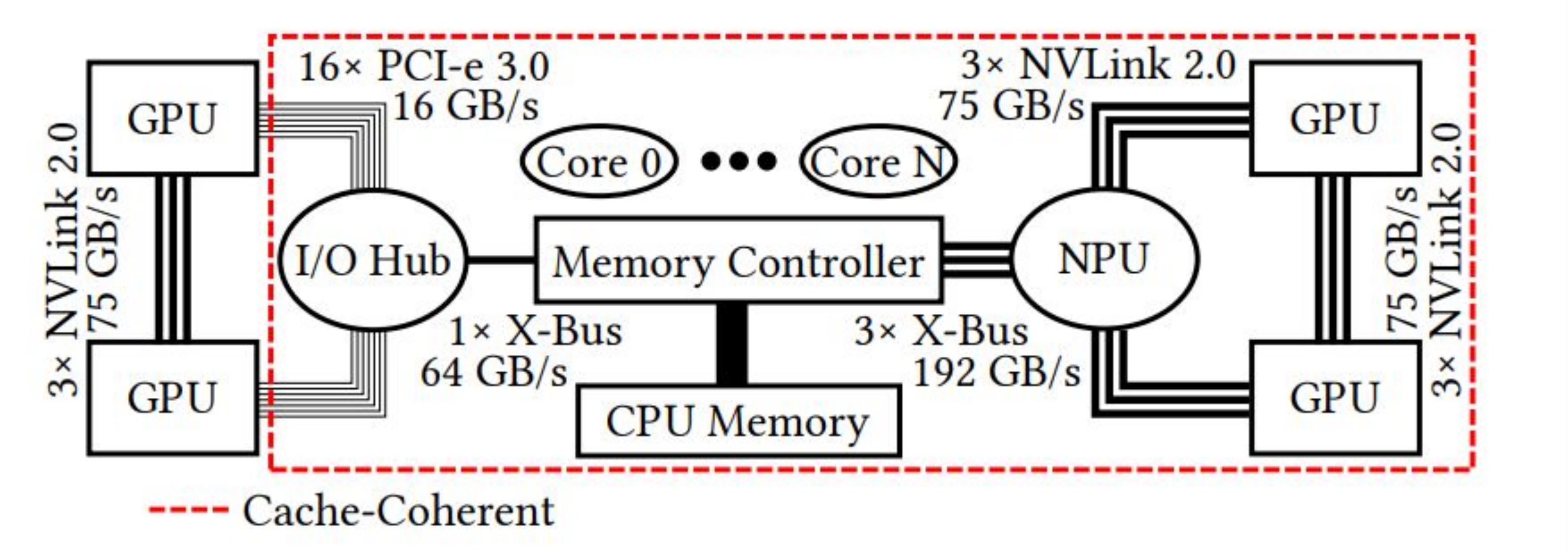
Coarse-grained Cooperation



PCI-e slots on motherboard



Overview of GPU interconnects



How can we improve?

Fast interconnects
NVLink 2.0, Infinity Fabric, CXL

High bandwidth (124 GiB/s total)
System-wide cache-coherence

Data-dependent memory access
Fine-grained CPU+GPU cooperation

Contributions

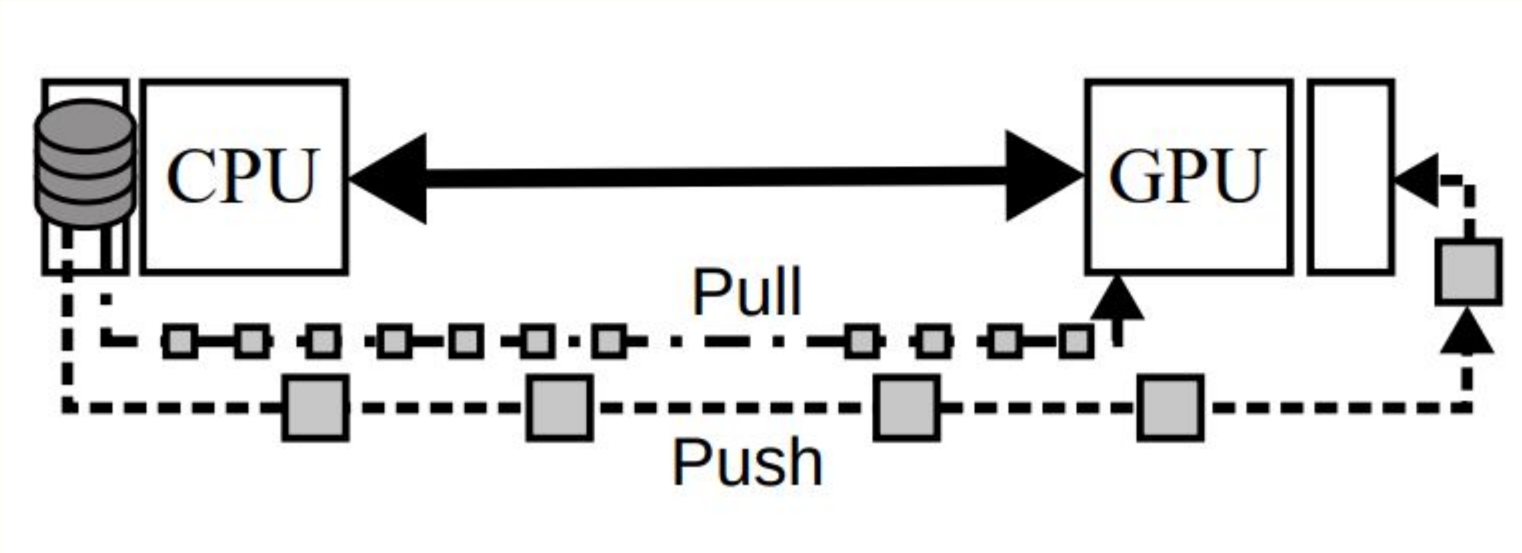
Hardware analysis

Data transfer strategy

Join operator

Cooperative co-processing approach

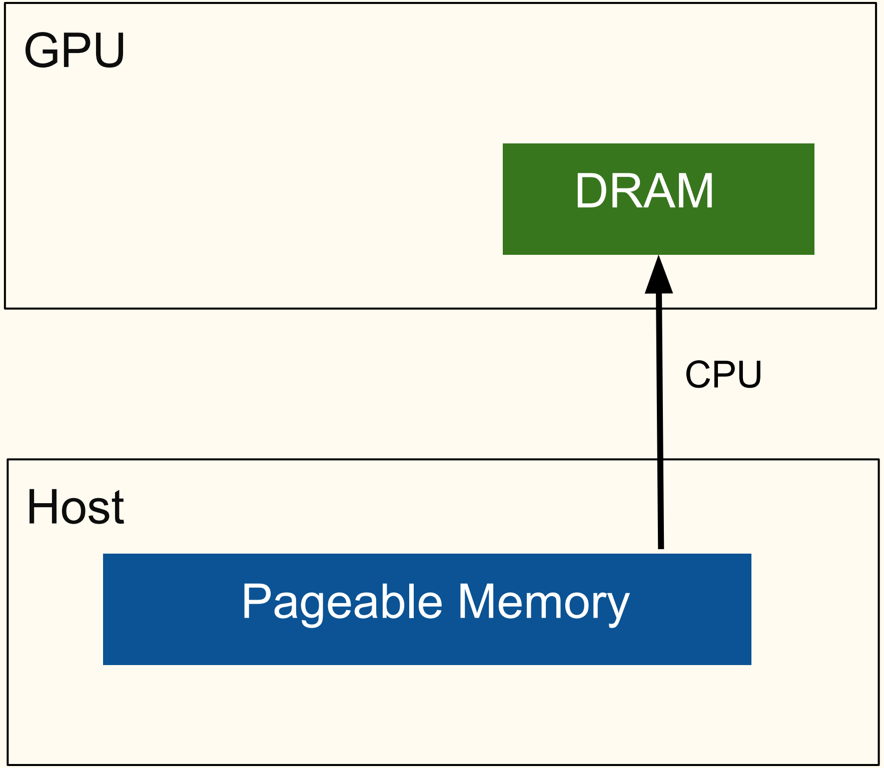
Push vs Pull based data transfer



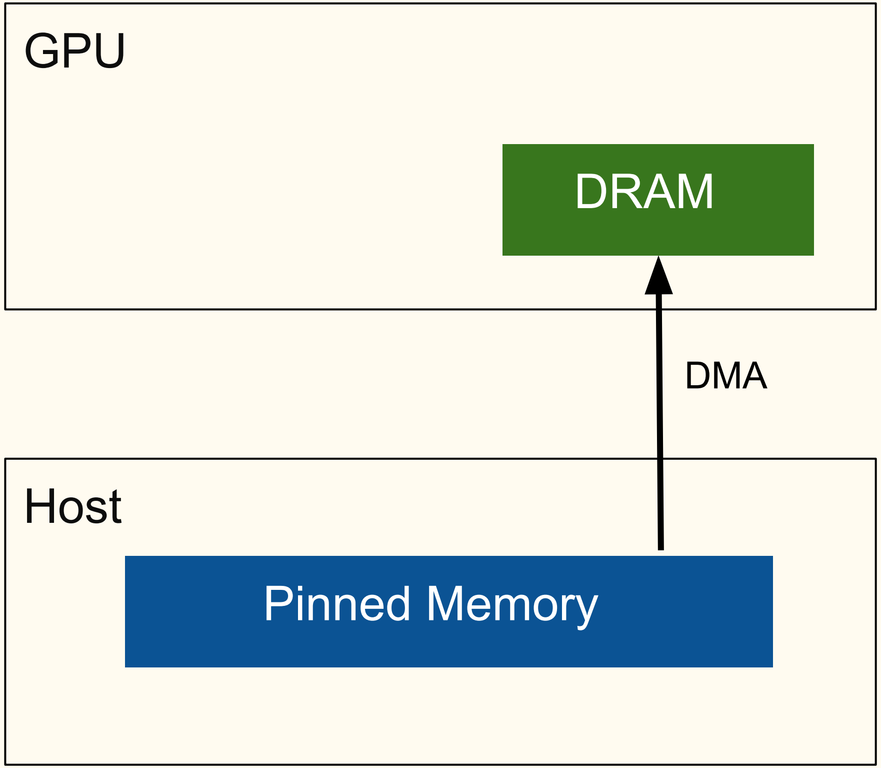
Data transfer methods

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

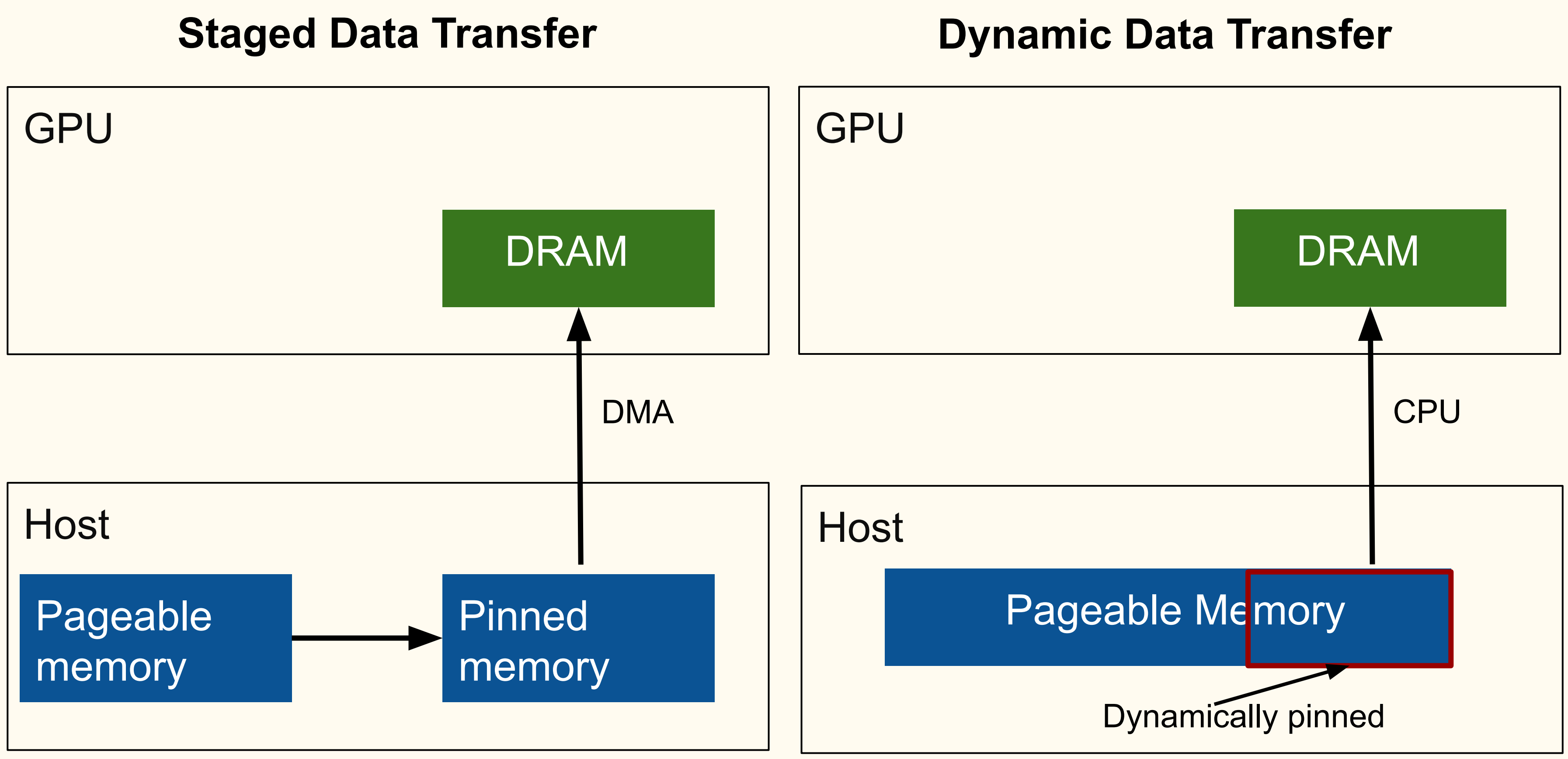
Pageable Data Transfer



Pinned Data Transfer



Physical location of the data matters!


Staged Data Transfer
Dynamic Data Transfer
GPU
DRAM
DMA
Host
Pageable memory
Pinned memory
GPU
DRAM
CPU
Host
Pageable Memory
Dynamically pinned

Contributions

Hardware analysis

Data transfer strategy

Join operator

Cooperative co-processing approach

Environment

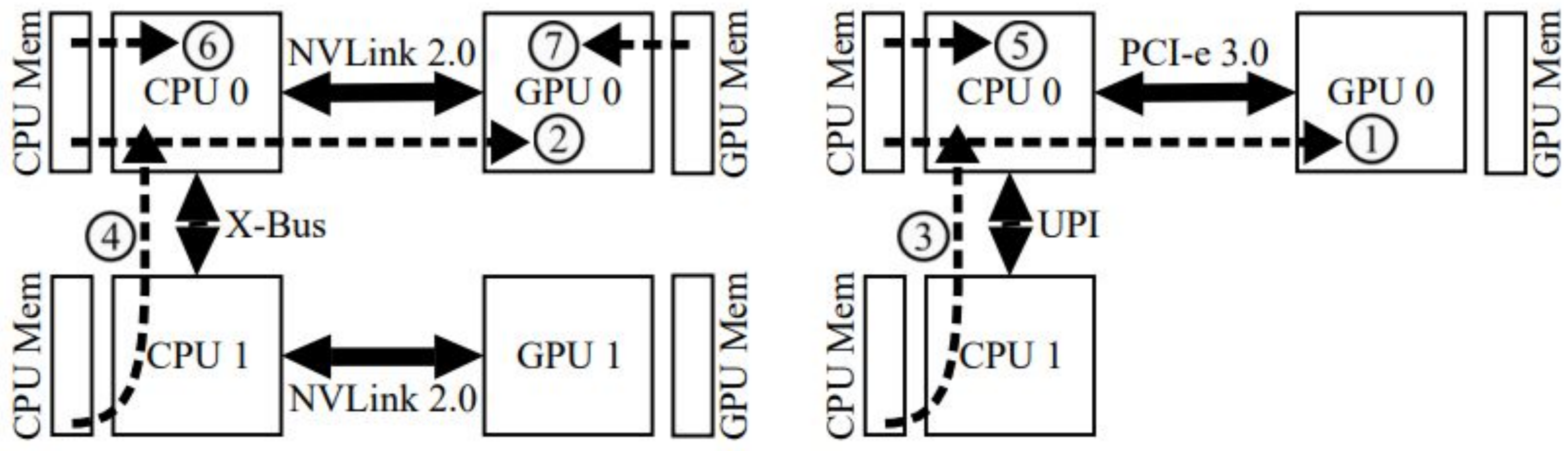
CPU Specifications:

- *IBM POWER9*
  - Configuration: Dual-socket
  - Clock Speed: 3.3 GHz
  - Cores: 32 (2 × 16)
  - Memory: 256 GB
- *Intel Xeon Gold 6126 ("Skylake-SP")*
  - Configuration: Dual-socket
  - Clock Speed: 2.6 GHz
  - Cores: 24 (2 × 12)
  - Memory: 1.5 TB

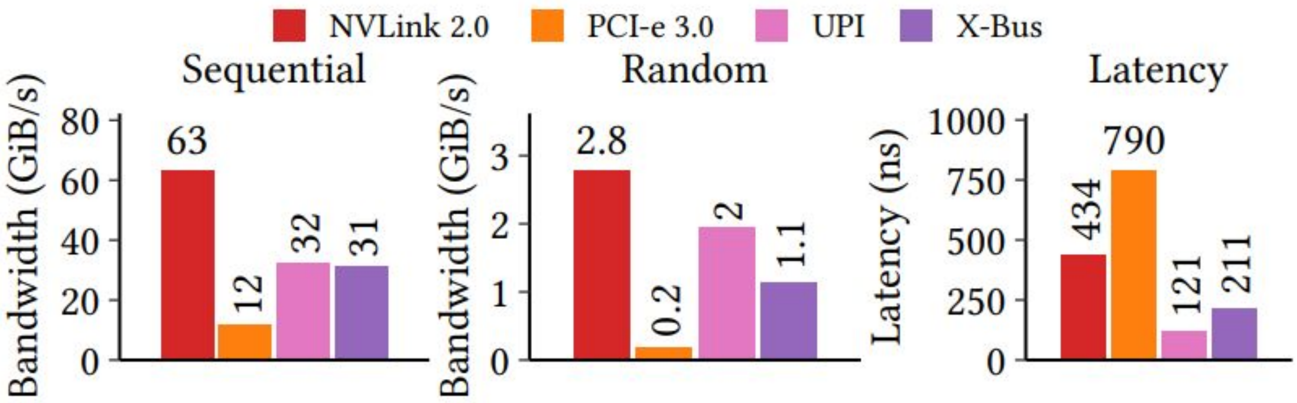
GPU Specifications:

- Nvidia Tesla V100-SXM2
- Nvidia V100-PCIE ("Volta")
- Memory: 16 GB for each GPU

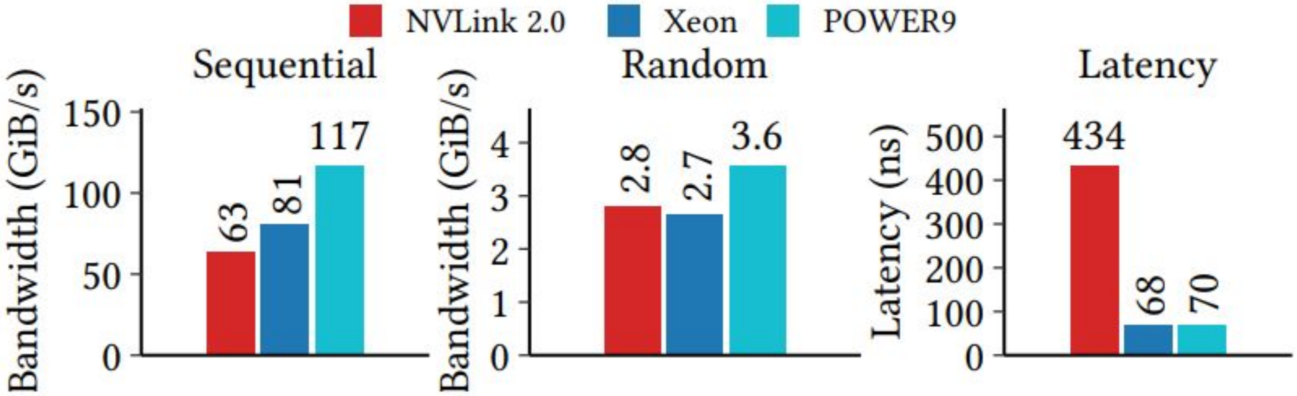
Access Paths



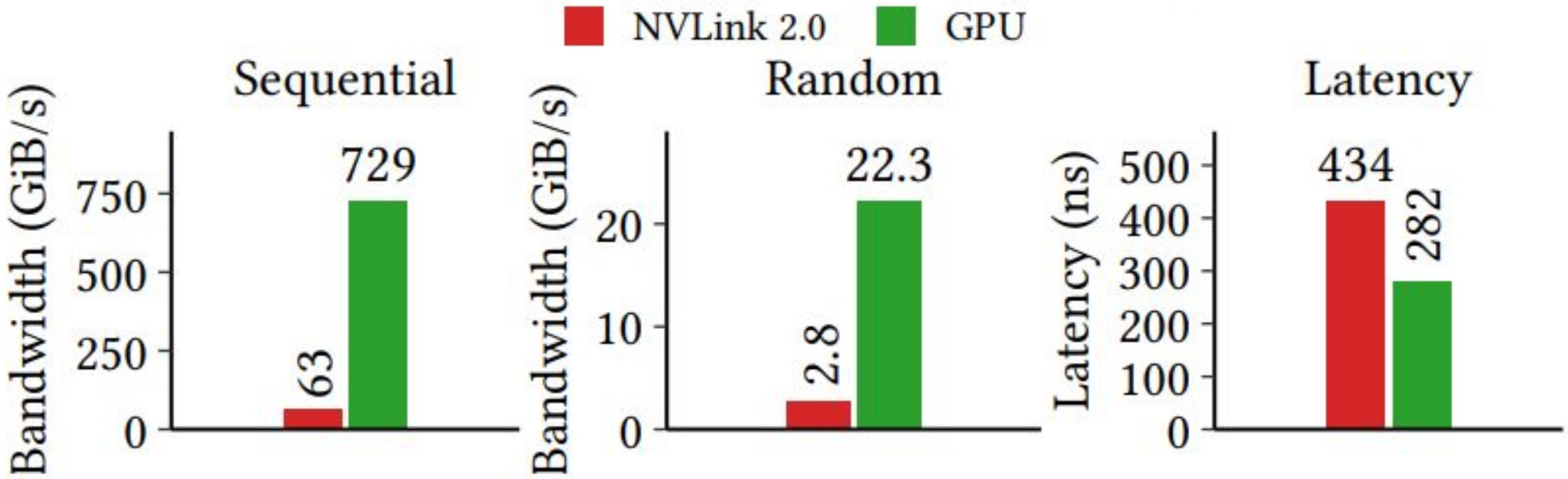
(a) 2× **IBM POWER9 with 2× Nvidia V100-SXM2.** (b) 2× **Intel Xeon with 1× Nvidia V100-PCIE.**



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



(b) NVLink 2.0 vs. CPU memory.



(c) NVLink 2.0 vs. GPU memory.

Contributions

Hardware analysis

Data transfer strategy

Join operator

Cooperative co-processing approach

Why the old method is not so good

Loading base relations from CPU memory requires high bandwidth, scaling the hash table beyond GPU memory requires low latency

sharing the hash table between multiple processors requires cache-coherence

Our new Join operator

The no-partitioning hash join algorithm is a parallel version of the canonical hash join

2 phases

1: build phase, takes inner relation R

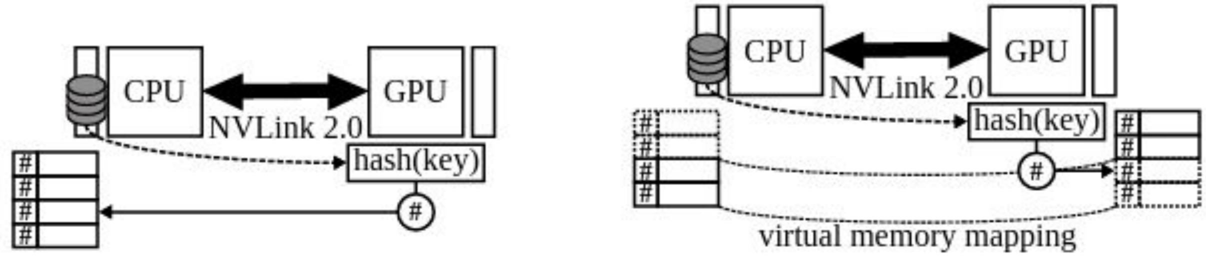
2:probe phase, takes outer relation S

executing the hash join in parallel on a system with p cores

Time complexity: $O(1/p(|R| + |S|))$.

Scale up build phase

store the hash table in CPU memory for bigger memory capacity



(a) Data and hash table in CPU memory.

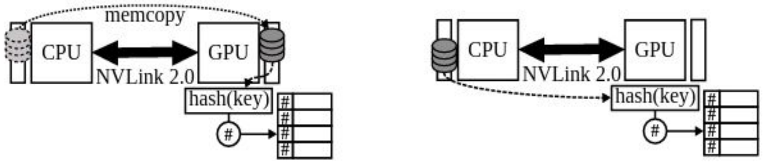
(b) Data in CPU memory and hash table spills from GPU memory into CPU memory.

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

Scale up probe phase

Simple baseline join first

Then we remove the probe-side cardinality limit by comparing the baseline to the Zero-Copy pull-based join



(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.

Finally, we replace the Zero-Copy transfer method with the Coherence transfer method in the Zero-Copy join

Optimizing the Hash Table Placement

Replace by hybrid hash table by greedy

Using the visual memory

It has zero additional cost

It can easily be integrated into existing databases

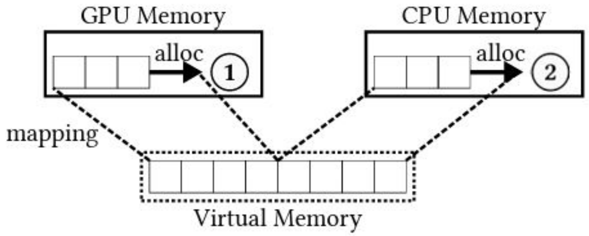


Figure 8: Allocating the hybrid hash table.

Contributions

Hardware analysis

Data transfer strategy

Join operator

Cooperative co-processing approach

Cooperative co-processing approach

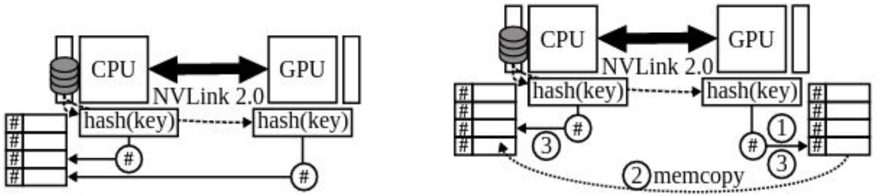
Make full use of CPU+GPU system

3 parts

Task schedule

Optimize hashtable placement strategy

Optimize on multiple GPUs



(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.

Task schedule

A task scheduler ensures that all processors deliver their highest possible throughput.

We adapt the traditional cpu based scheduler, make all processors can scheduling.

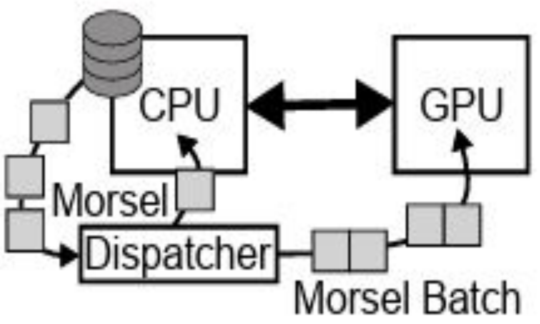


Figure 10: Dynamically scheduling tasks to CPU and GPU processors.

Optimize hashtable placement strategy

Processors are fastest when accessing

their local memories.

So we want to optimize multi processor placement

strategy so they can access closest data.

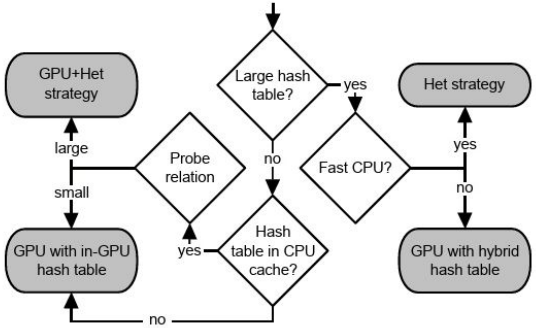


Figure 11: Hash table placement decision.

We also consider a special case of small build-side relations separately, so we can optimize hashtable locally.

Optimize on multiple GPUs

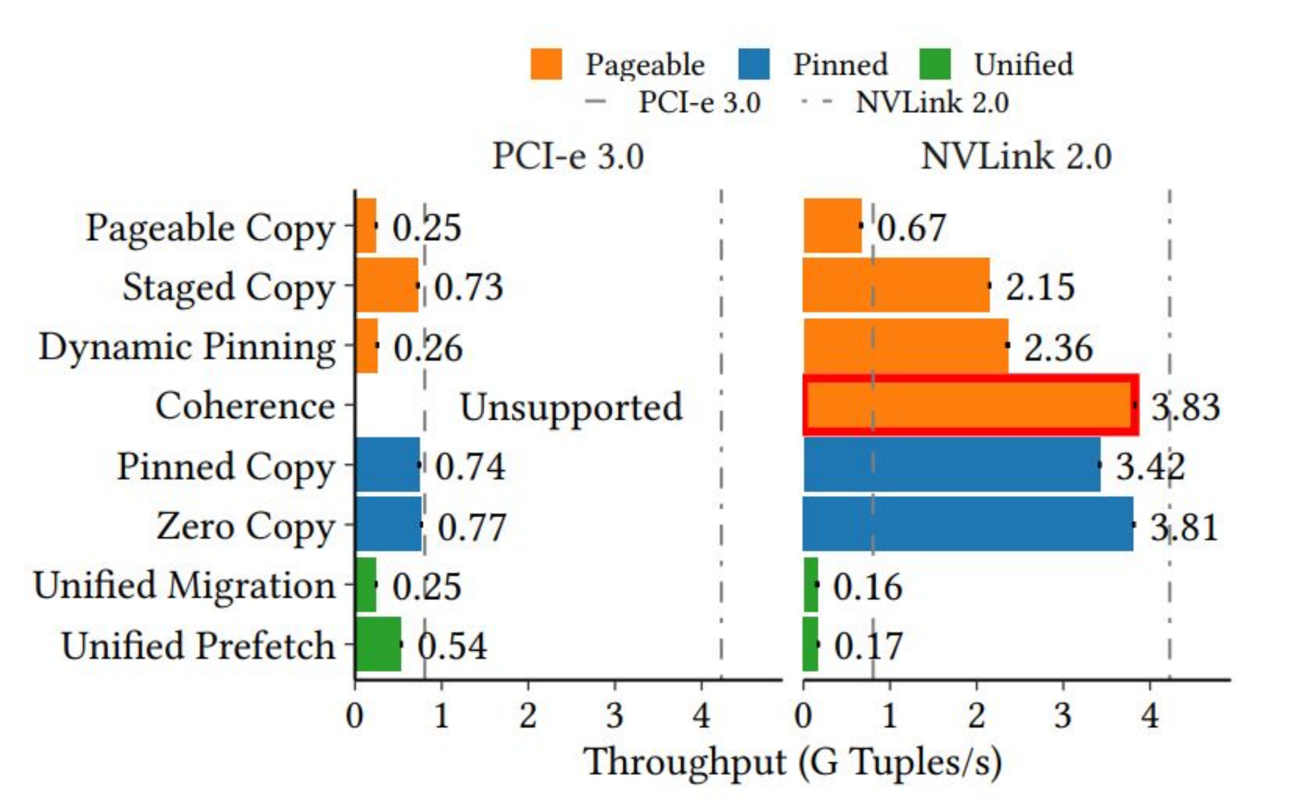
for large hash tables, multi-GPU systems can distribute the hash table over multiple GPUs, as GPUs are latency insensitive.

We use multiple GPUs instead of CPU+GPU to avoid computational skew, free CPU memory bandwidth and utilize the full bi-directional bandwidth of fast interconnects

Experiments: Setup and Configuration

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

Experiments: GPU Transfer Methods



HashTable Locality

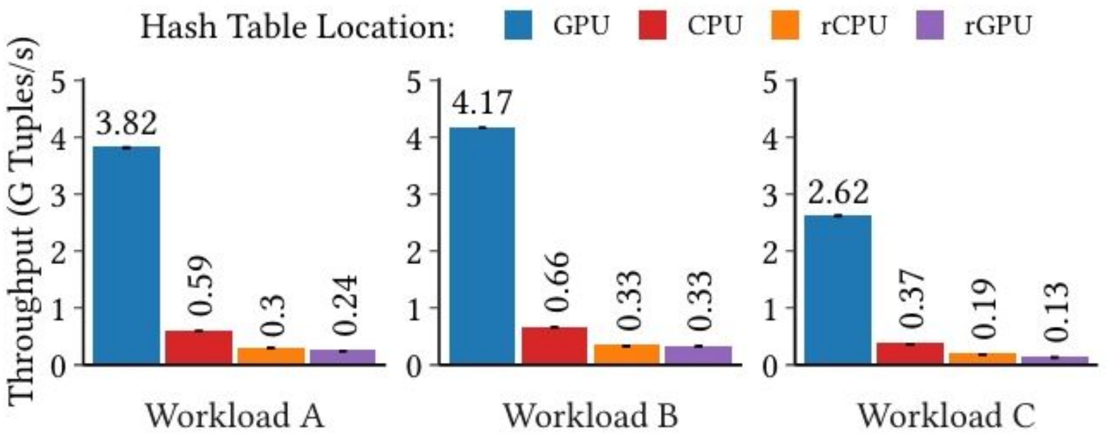


Figure 14: Join performance of the GPU when the hash table is located on different processors, increasing the number of interconnect hops from 0 to 3.

Selection and Aggregation Scaling

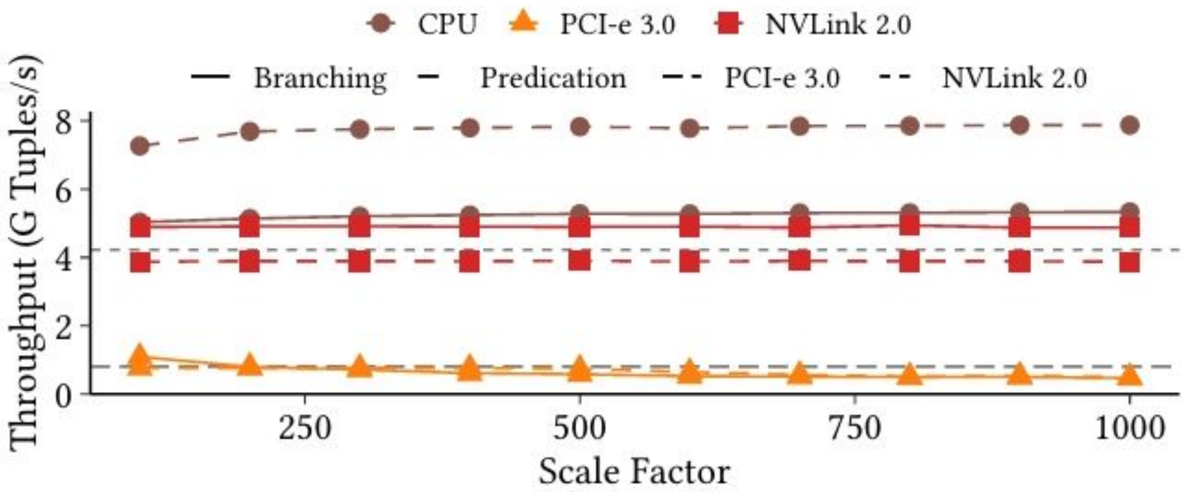


Figure 15: Scaling the data size of TPC-H query 6.

Probe-side Scaling

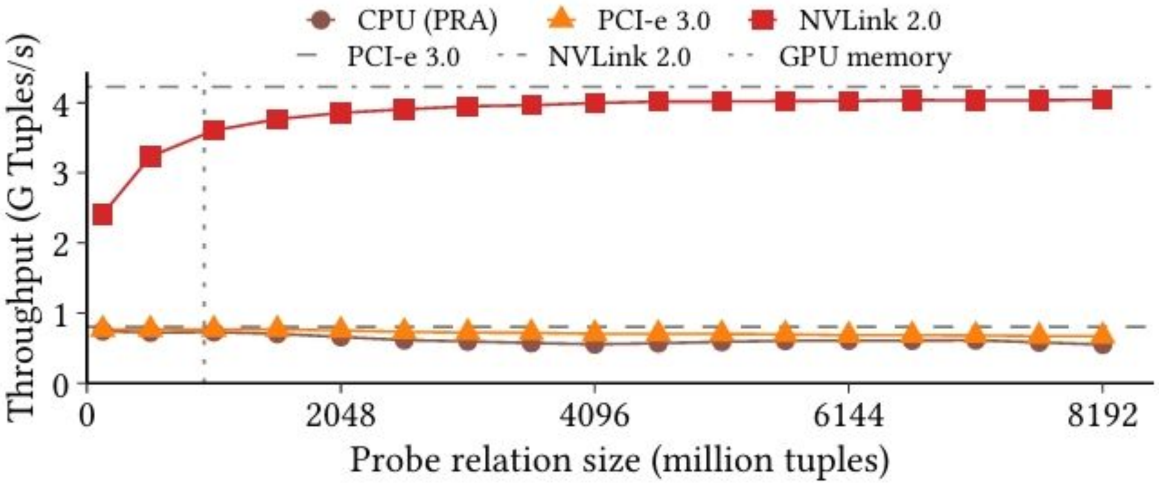


Figure 16: Scaling the probe-side relation.

Build-side Relation

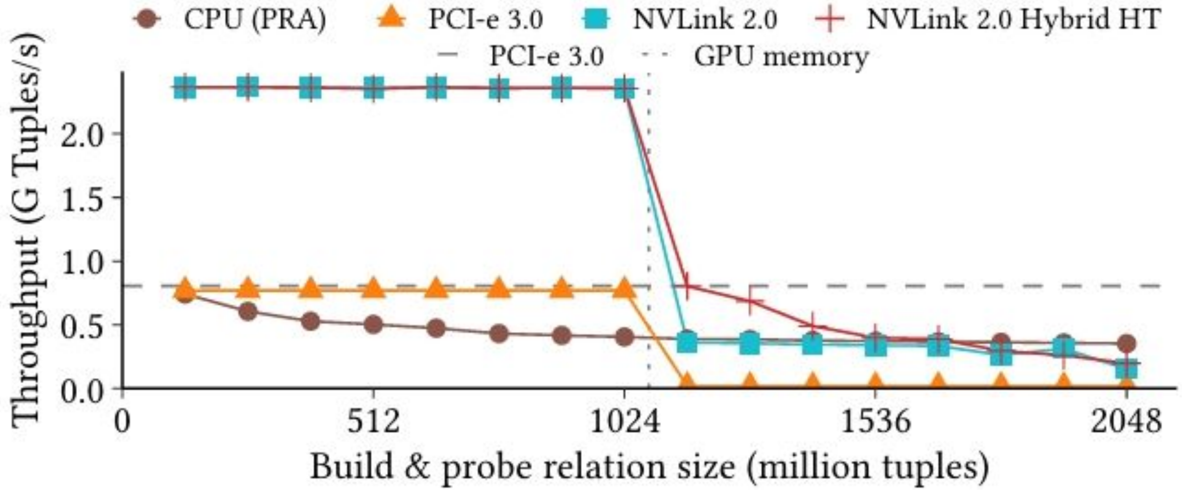
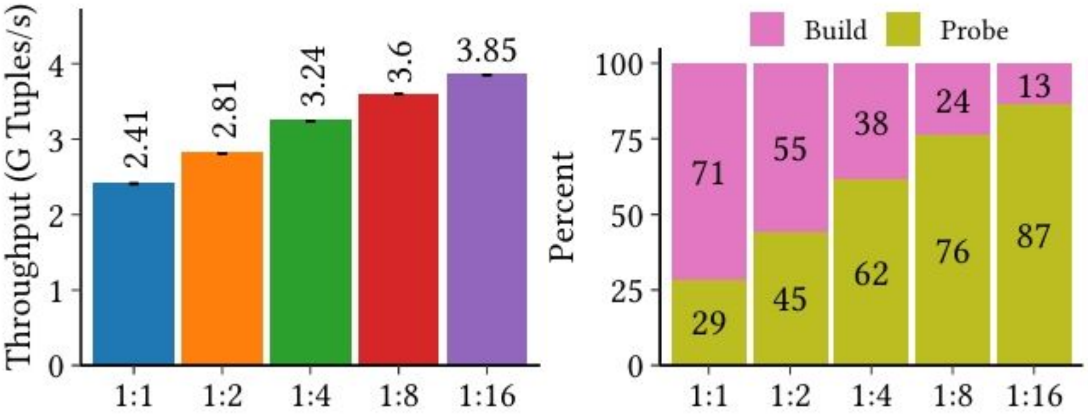


Figure 17: Scaling the build-side relation.

Build-to-probeRatios



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

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

Join Selectivity

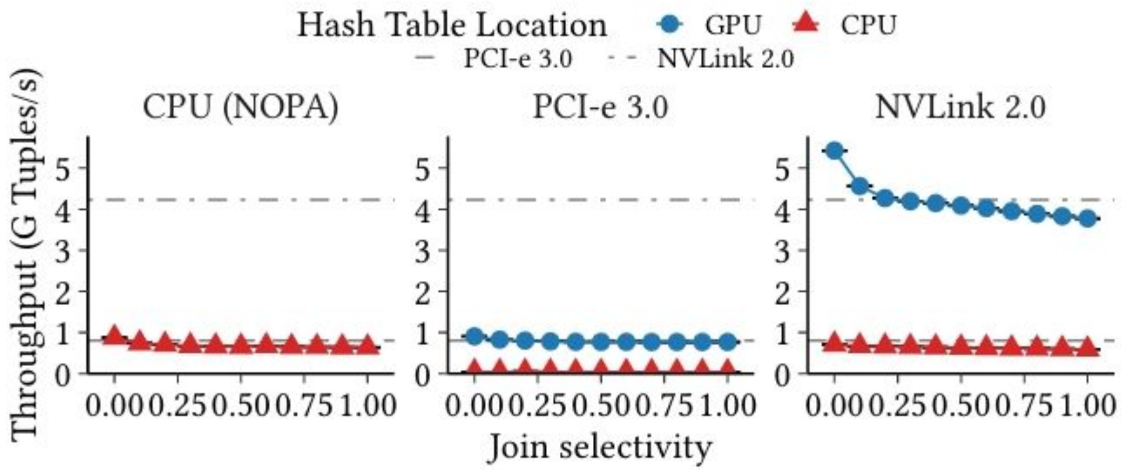


Figure 20: The effect of join selectivity on throughput.

CPU/GPU Co-processing Scale-up

Related works

Transfer Bottleneck

Previous Solutions: GPU databases (GDB, Ocelot, CoGaDB) and machine learning frameworks (SystemML, DAnA) stream data from CPU to co-processor.

Transfer Optimization

caching in co-processor memory, employing data compression

Transfer Avoidance

An on-chip interconnect

Out-of-core GPU Data Structures

GPU-efficient data structures like hash tables, B-trees, and binary trees.

Conclusion

- NVLink 2.0 boosts large-scale data processing in databases by resolving GPU memory constraints and slow data transfer rates.
- The fast interconnect system facilitates swift and efficient data exchange between CPU and GPU, enhancing the processing of larger datasets.
- Empirical results show marked performance enhancements in critical database operations, particularly hash joins, with the adoption of NVLink 2.0.
- NVLink 2.0's advancements make GPUs increasingly viable for managing extensive data volumes in modern database management systems.

Thank You