

Accelerating Presto with GPUs

Introducing GPU acceleration into a production SQL engine

Greg Kimball · Devavret Makkar · Todd Mostak · **Karthikeyan Natarajan** (NVIDIA)
Daniel Bauer · Luis Garcés-Erice · **Deepak Majeti** · Zoltán A. Nagy · Sean Rooney (IBM)

ADMS @ VLDB 2026

Exploiting GPU bandwidth: the three principles

Analytics is memory-bandwidth-bound, and GPUs have **>10× the bandwidth** — yet mainstream engines stay CPU-bound. Capturing it means one thing: **stop moving across the CPU bottleneck.**

1

Read INTO GPU memory

no CPU staging

2

KEEP it there

across every operator, whole query

3

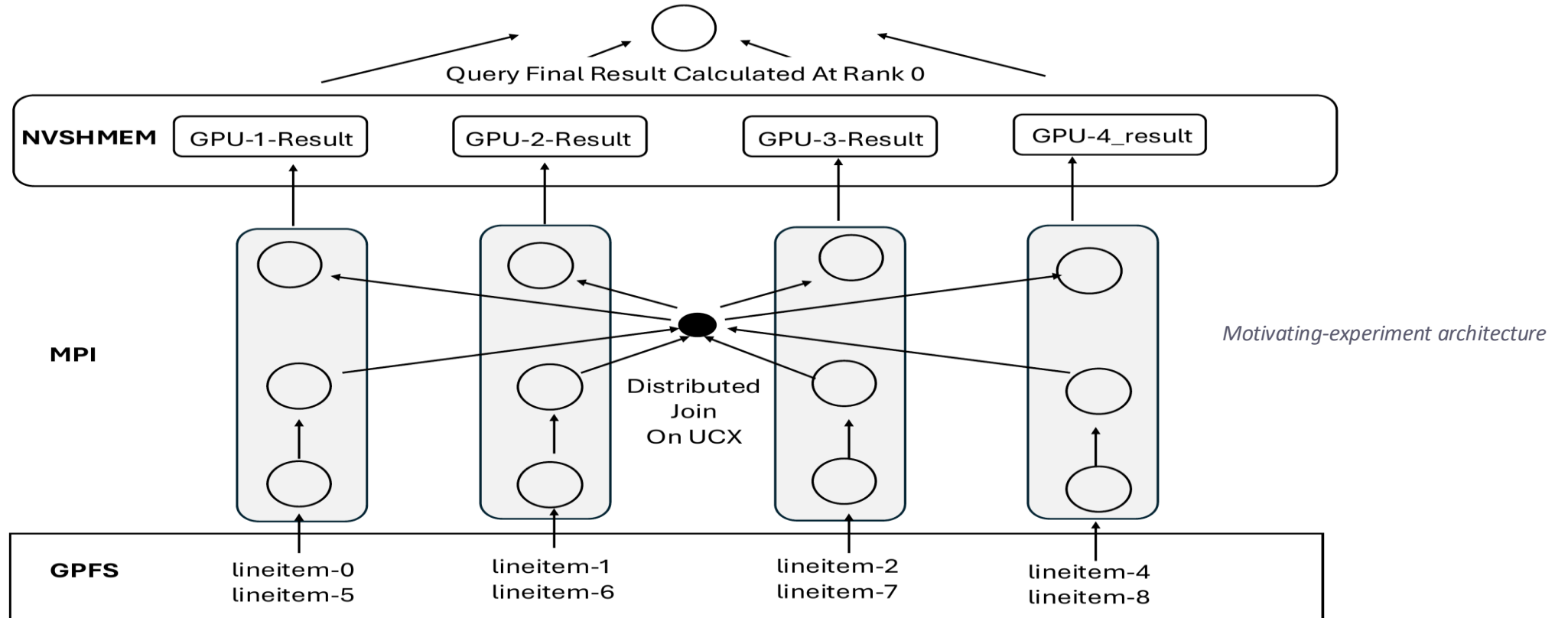
Exchange GPU→GPU

never through the CPU

The catch: do this **inside a production engine** — respecting Velox's async operator model, not bypassing it. That's why prior GPU databases were built from scratch; **we didn't.**

Roadmap: first validated with a bare-bones system → then built into Presto.

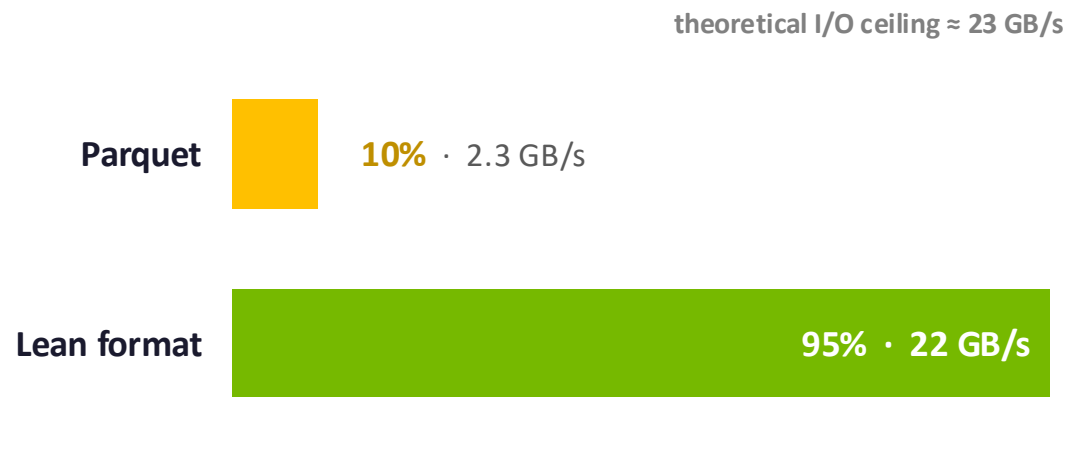
Understanding performance limits



- **Handcrafted bare-bones system: pure cuDF + MPI + NVSHMEM + UCX.**
 - No planner, no coordinator.
- **Goal: a lower bound on latency the hardware allows.**
- **Hardware: 8xA100 (80 GB) per box, GPFS @ 170 GB/s, GPU Direct Storage.**

Findings — the hardware floor

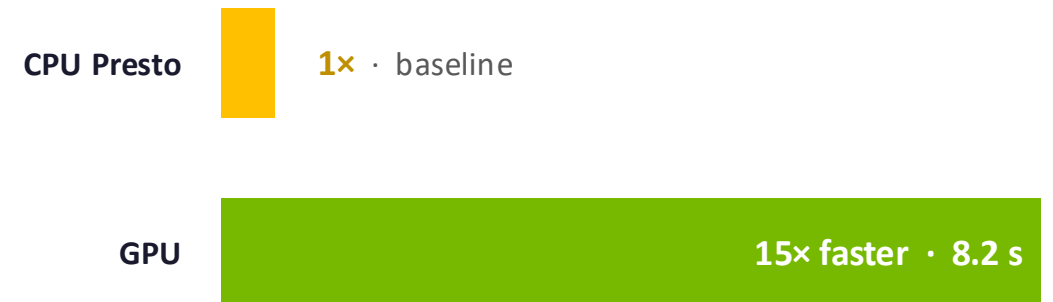
The bottleneck is I/O, not compute



Same compute, same hardware — only the format changed. The swap is the proof: I/O is the wall.

...and the floor is fast

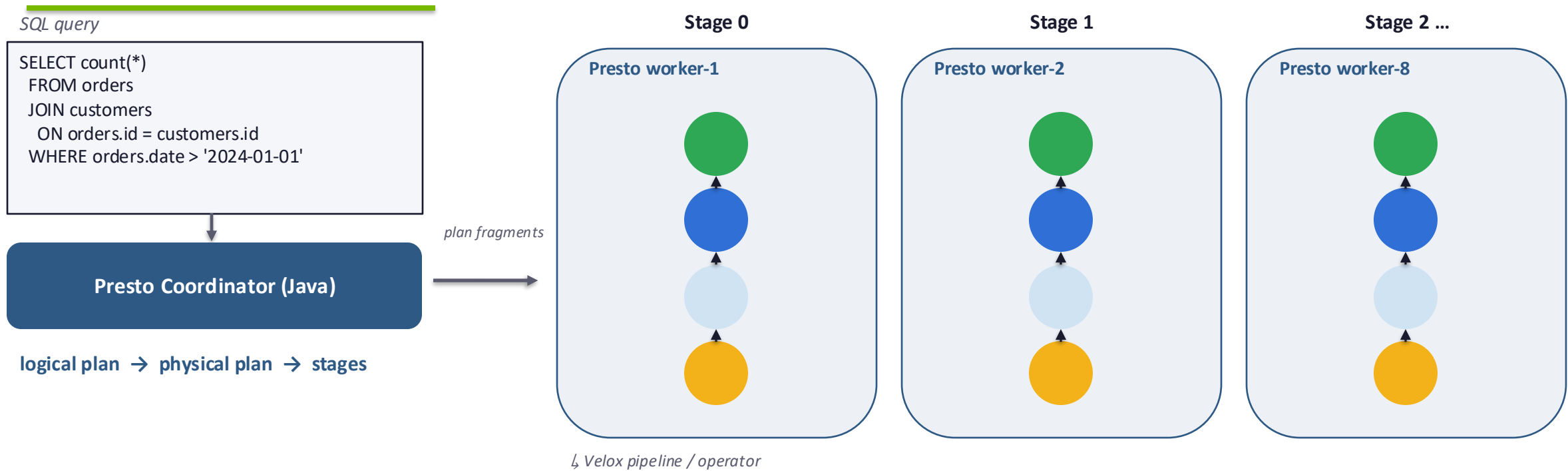
Q1 @ SF-10K (10 TB) — speedup



On par with purpose-built Theseus (230 s, 16 nodes) — the ceiling is real and worth chasing.

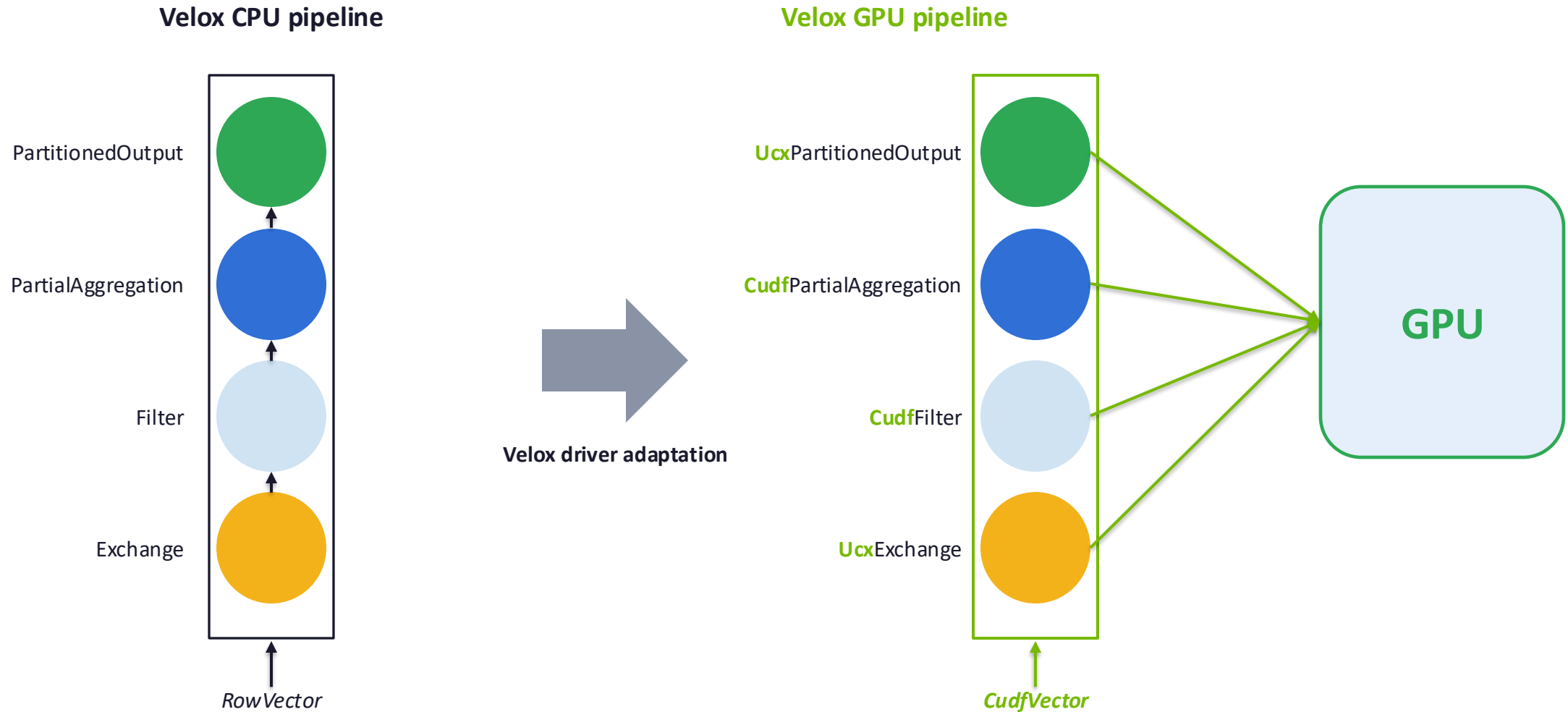
Not proposing to replace Parquet — just quantifying the gap. Best chunk size depends on scale + resources $\Rightarrow$ motivates a smarter planner (Part 2).

Integrating into Presto: how a query runs



Stage	PlanNode	Workers	Input
0	Aggregation	1	Stage=1
1	HashJoin (build=Orders, probe=Customers)	1..8	Stage=2,3
N	TableScan	7,8	s3://orders

Axis 1 — swap operators, keep data on the GPU (principles 1 & 2)

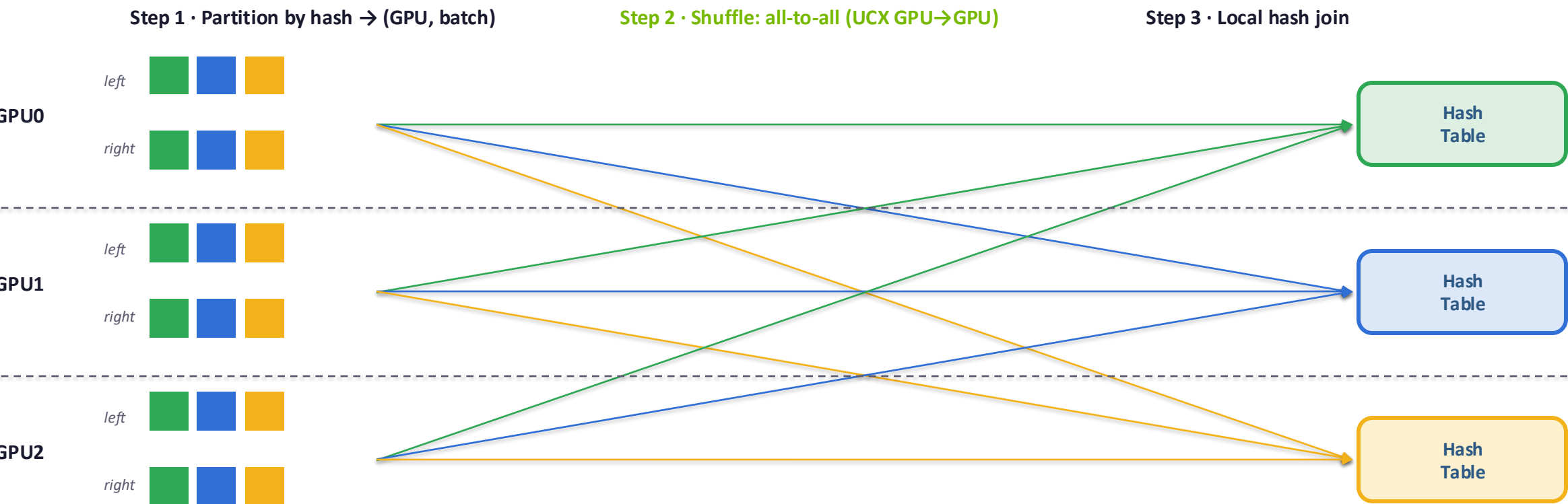


~50% of operators GPU-aware — yet enough for all 22 TPC-H + all 99 TPC-DS, no CPU fallback

CudfVector preserves Velox's async futures · a CPU fallback (CudfToVelox / CudfFromVelox) costs a GPU↔CPU round-trip ⇒ coverage matters

Why the exchange dominates — a shuffle moves the whole dataset

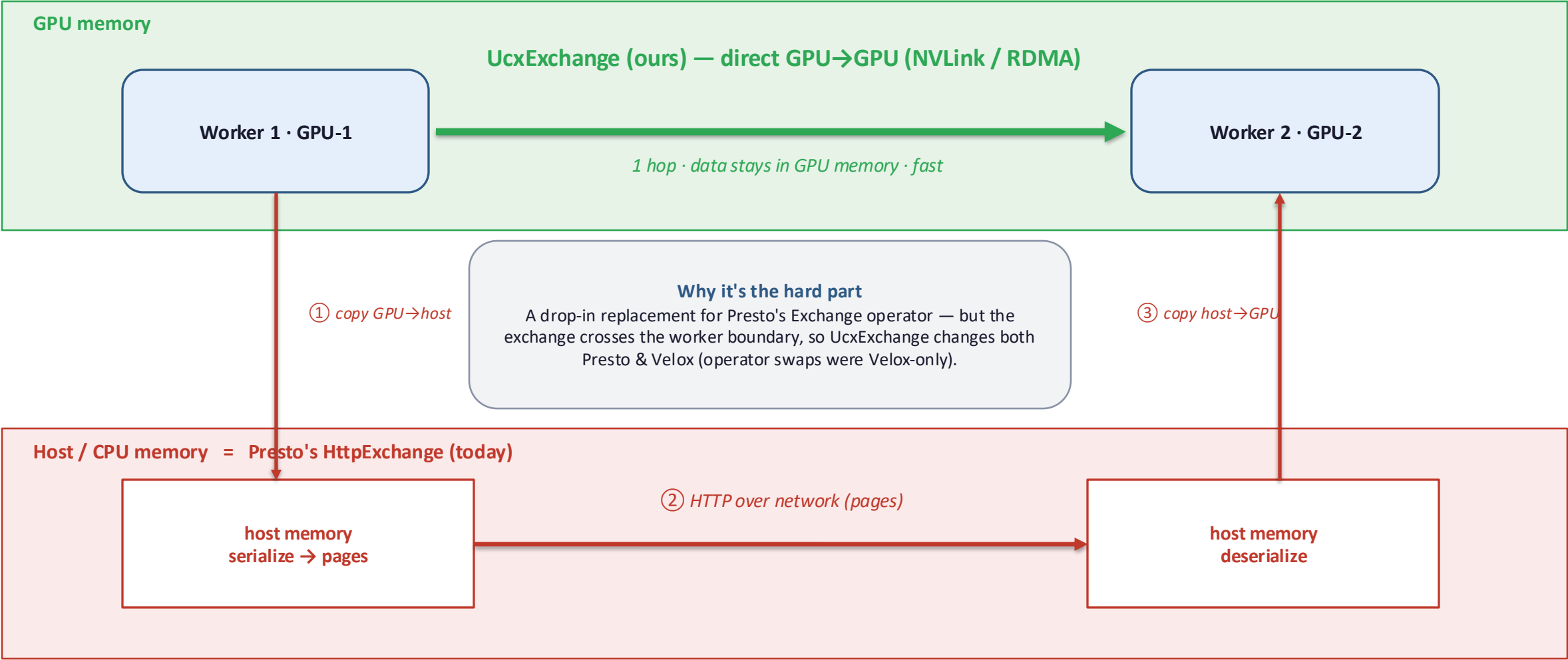
An exchange repartitions data across workers — joins, aggregations, sorts. Example below: a distributed hash join.



Every batch crosses to every GPU → the exchange is bandwidth-bound.

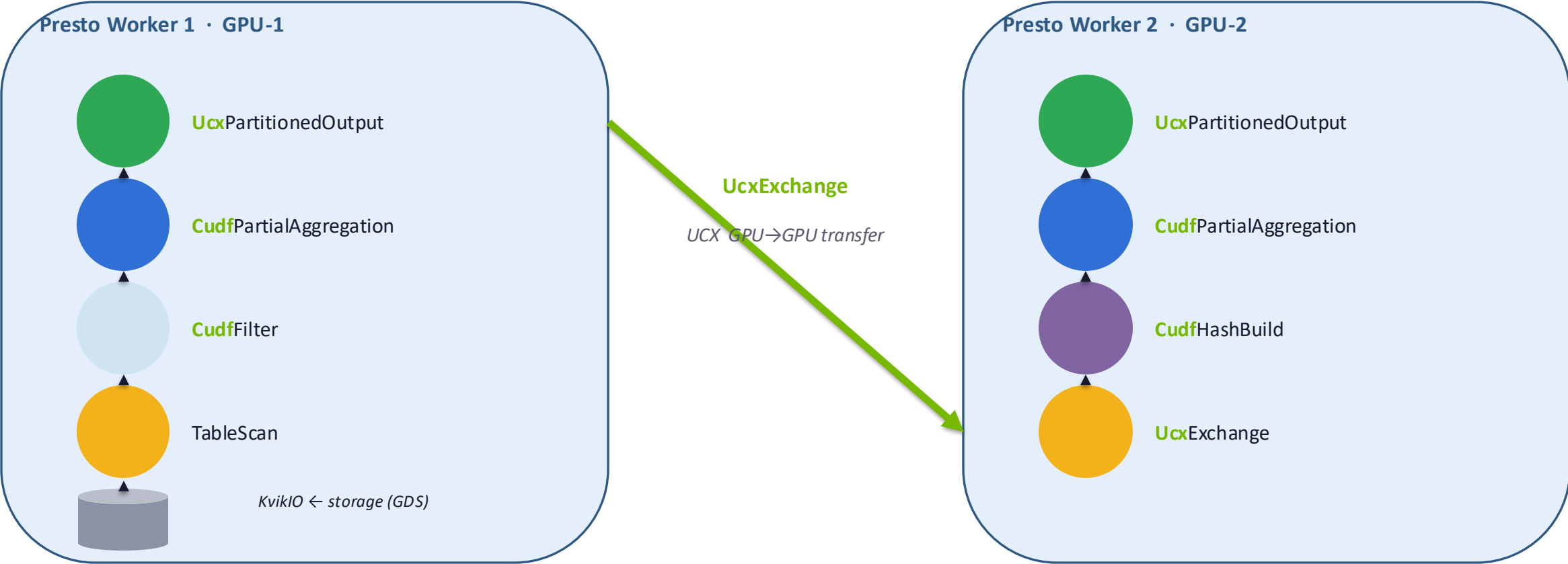
Presto's default HttpExchange shuffles through host / CPU memory (serialize → pages → HTTP) — not GPU-aware.

UcxExchange — the core contribution (principle 3)



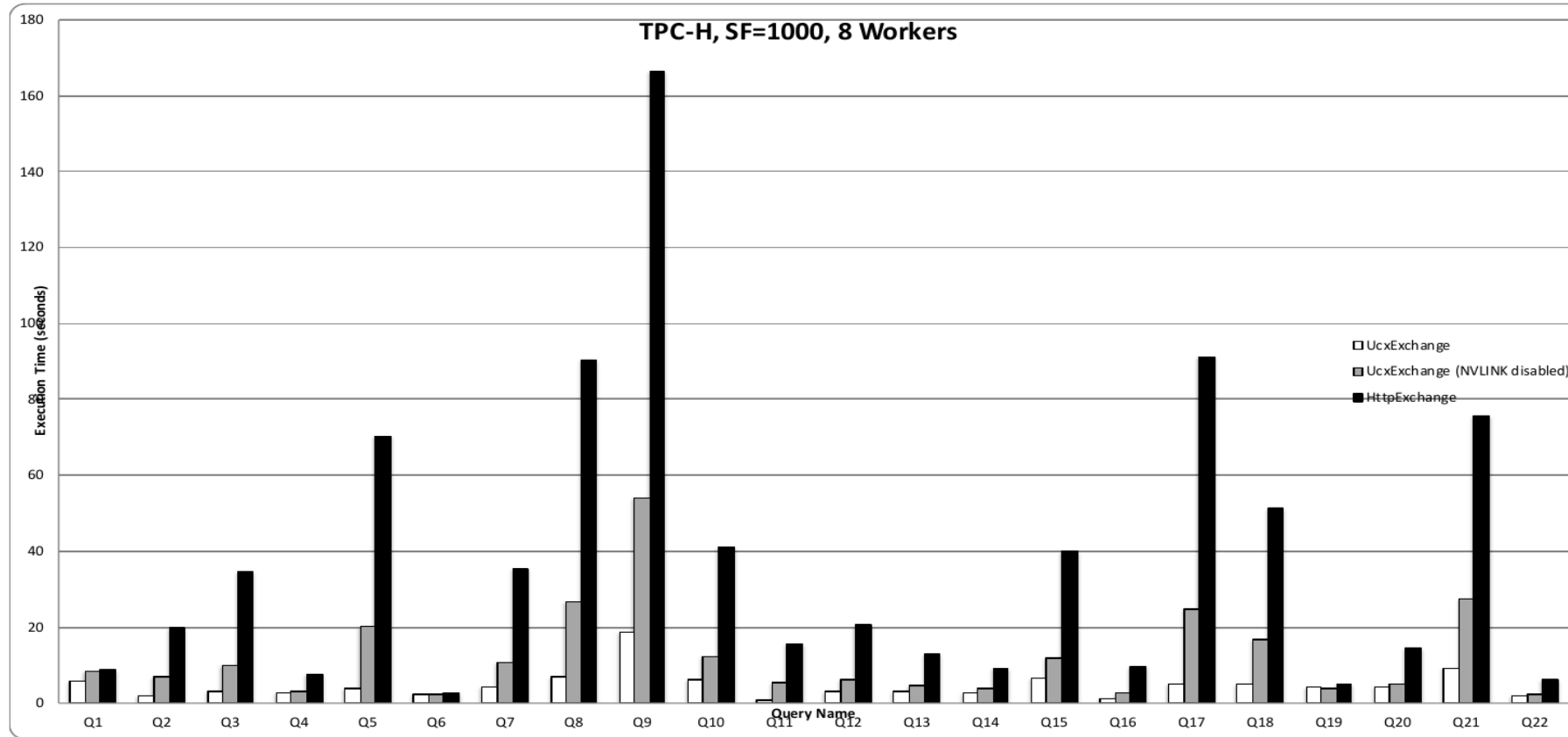
Putting it together — data never leaves GPU memory

The data never leaves the GPU cluster's memory



GPU libraries: KvikIO (storage→GPU) · cuDF (operators) · UCX (GPU→GPU)

UcxExchange vs HttpExchange · TPC-H @ SF=1000, 8×A100

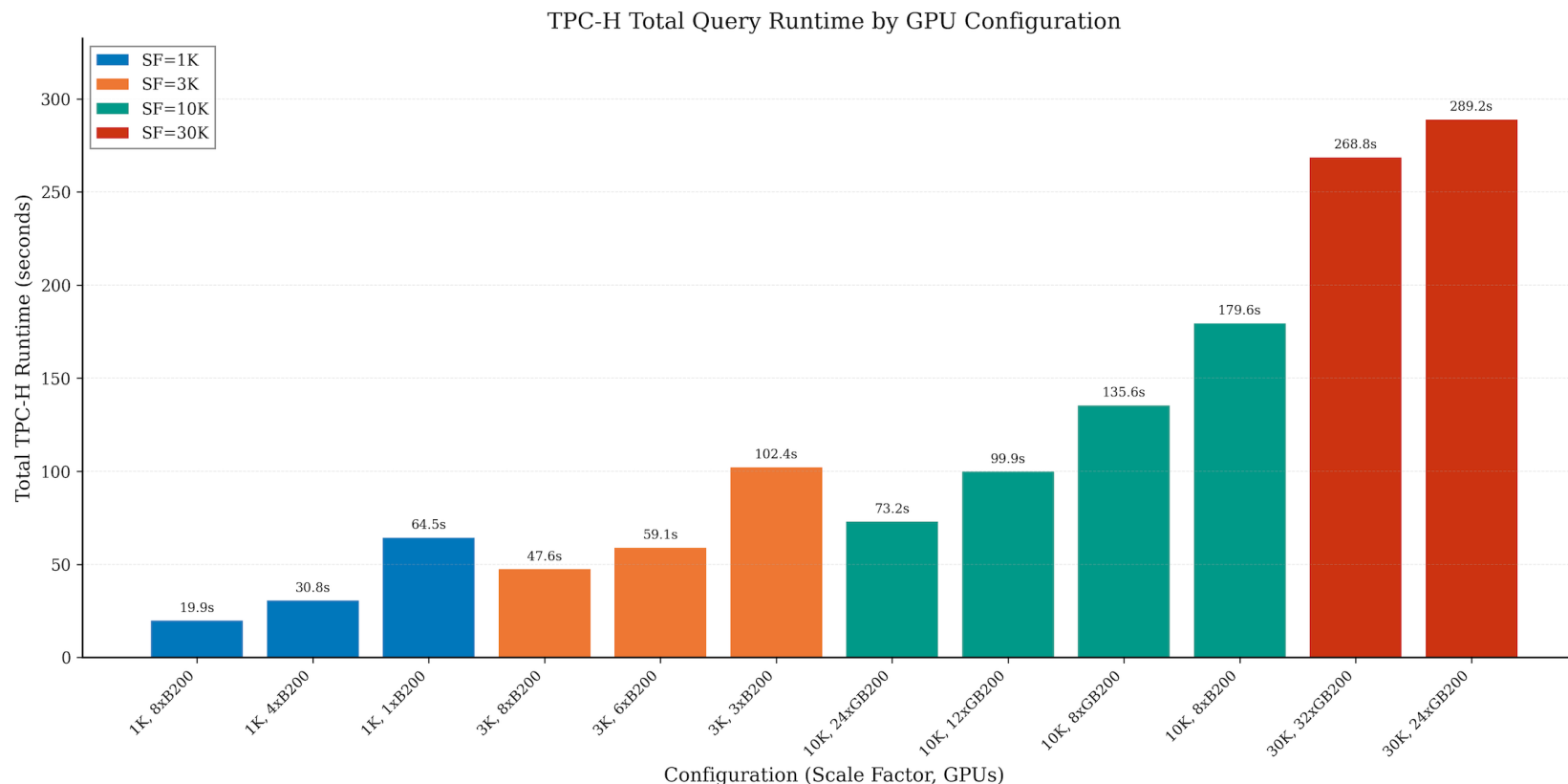


- All 22 queries: 93 s vs 828 s HttpExchange — >8×.
- Geomean/query: 22.2 s → 3.8 s (6×).
- Exchange-heavy Q9: >20× faster.

- Even NVLink-disabled (RDMA): still 3×.
- Small-exchange queries (Q1 = 4 rows) see no benefit ⇒ motivates a scheduler.

Scaling up and out

Get faster two ways — newer GPUs and more of them — and scale out to **30 TB in ~4.5 min.**



Faster with newer GPUs

- SF=1K, 8 GPUs: 93 s (A100) → 19.9 s (B200) = 4.7×
- Even 1xB200 (64.9 s) beats 8xA100 (93 s) — one new GPU > eight old ones.

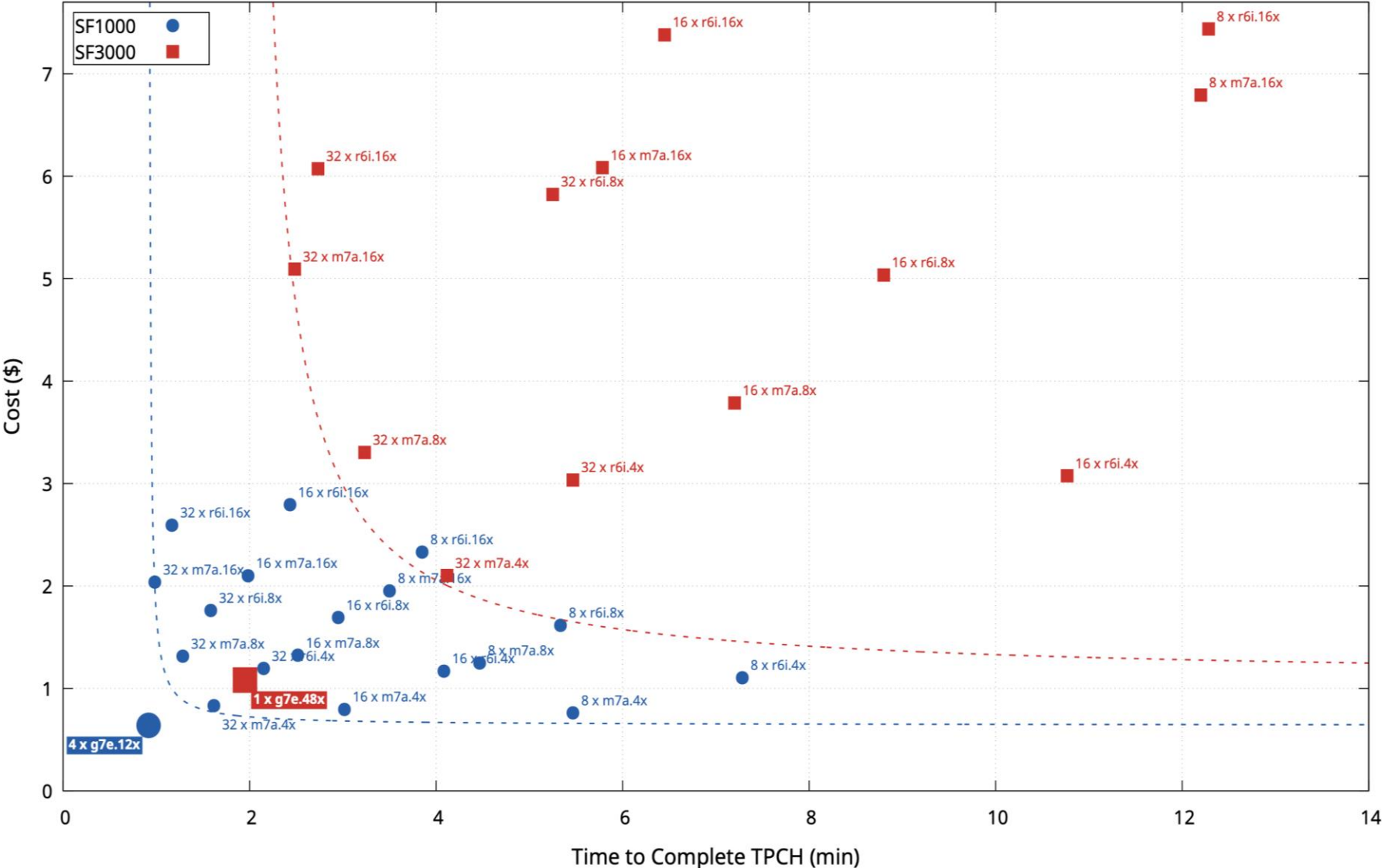
Faster with more GPUs

- Every scale factor gets faster with more GPUs.
- SF=1K: 8 / 4 / 1xB200 = 19.9 / 30.8 / 64.5 s.

Cost / performance vs CPU

• each point = a GPU or CPU cluster on AWS (cost vs. time)

Presto GPU price-performance advantage over CPU on AWS



- Fair metric: cost × time (lower is better), priced on AWS.
- SF=1K: 2.3× GPU advantage.
- SF=3K: 6.3× advantage — grows with scale.
- Consistent with Theseus (4× at cost parity) and Sirius (7–8×).
- Gap likely widens as older GPUs get cheaper.

Limitations & future work

- **Working set must fit in GPU memory — else OOM. Spilling:**
 - *Explicit — Velox-style per-operator spill; more control, more operator complexity.*
 - *Automatic — CUDA managed memory; no changes to operators, bypasses Velox's manager.*
- **Planner is not GPU-aware — suboptimal plans & placement.**
 - *Assumes uniform workers on all stages — ignores NVLink $\gg$ RDMA.*
 - *Future: expose hardware; pick a worker subset per stage (e.g. favor NVLink).*
- **Cloud-storage I/O impedance — S3: no GDS, high latency.**
 - *Future: fan-in from many cheap VM/CPU's to fill the GPU pipe*
 - *Parallel S3 reads, Cache in Velox AsyncCache, Using UcxExchange to feed GPUs*

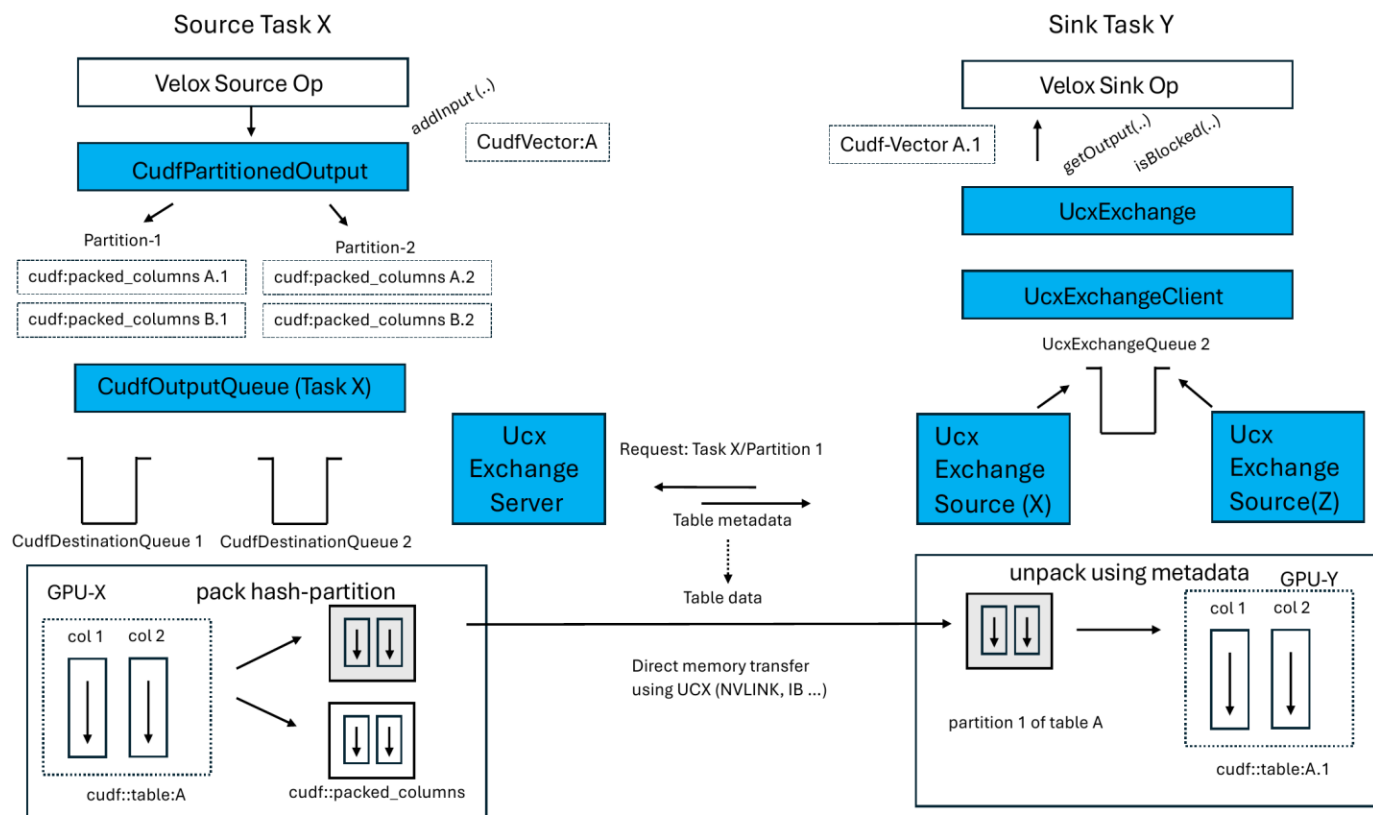
Conclusion

- **Three principles to avoid the CPU bottleneck:**
 - Read directly into GPU memory.
 - In a Velox pipeline, replace CPU operators with GPU-aware
 - At stage boundaries, exchange GPU-to-GPU with UCX.
- **Built on open-source Presto/Velox — existing workloads run on GPUs out of the box.**
- **Measured cost/performance advantage over CPU — up to 6× (grows with scale).**
- **Evidence for scaling up (newer + more GPUs) and out — validated to 30 TB.**

Thank you — questions?

Backup

UcxExchange Architecture



- CudfPartitionedOutput packs cuDF columns into contiguous GPU memory, per-partition queues.
- Receiver: active-message handshake → unique rendezvous tag; tagRecv metadata, then packed GPU data.
- One UCX connection per worker-pair per direction; reused across exchanges (setup is costly).
- Single-threaded communicator drives UCX; transfers run in UCX independently (not the bottleneck).
- Optional vector compaction merges tiny vectors from skewed joins before transmission.

Paper Figure 4 — UcxExchange architecture