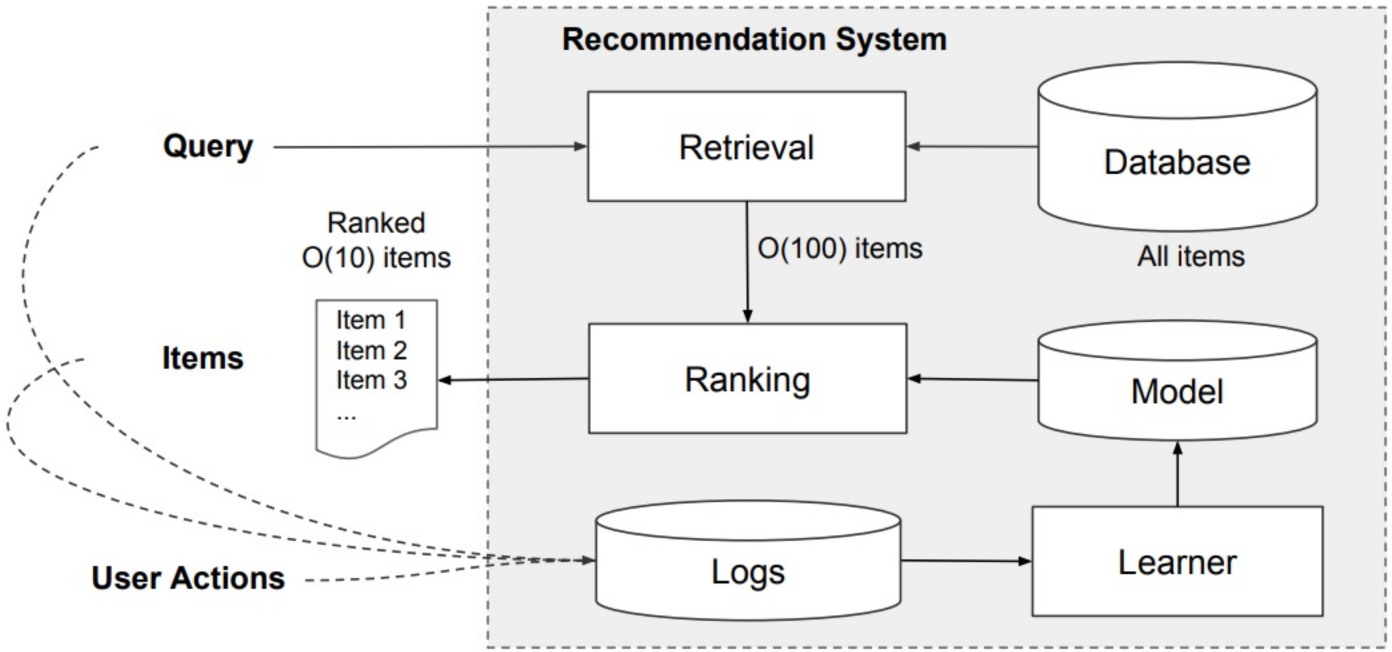
Accelerating recommendation model training using ByteCCL and UCX

Haibin Lin (Bytedance), Mikhail Brinskii (NVIDIA), Yimin Jiang (Bytedance),
Yulu Jia (Bytedance), Chengyu Dai (Bytedance), Yibo Zhu (Bytedance)

Recommendation models

- Ranking and recommendation
  - Application: ads, feed, search, etc.
  - Involves machine learning models that predict the probability of one or multiple events at the same time. Events ranked higher are shown to users

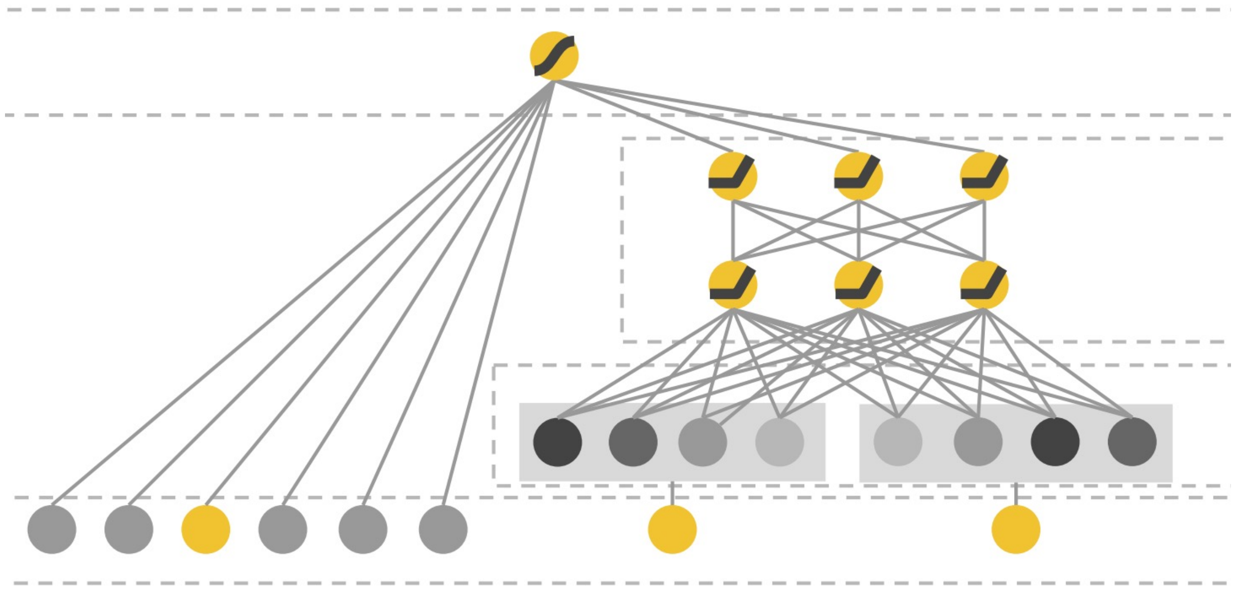


Cheng, Heng-Tze, et al. "Wide & deep learning for recommender systems." *Proceedings of the 1st workshop on deep learning for recommender systems*. 2016.

ByteDance

Recommendation model: wide and deep

- “wide and deep” is a ranking model applied successfully in practice [2]



Neural network layers

- Model non-linear functions and feature interactions

Embedding table lookup

- Lookup the corresponding one-hot embedding vector based on the category index
- Billions of parameters for lookup

Dense features

- continuous values
- e.g., age, num views

Sparse features

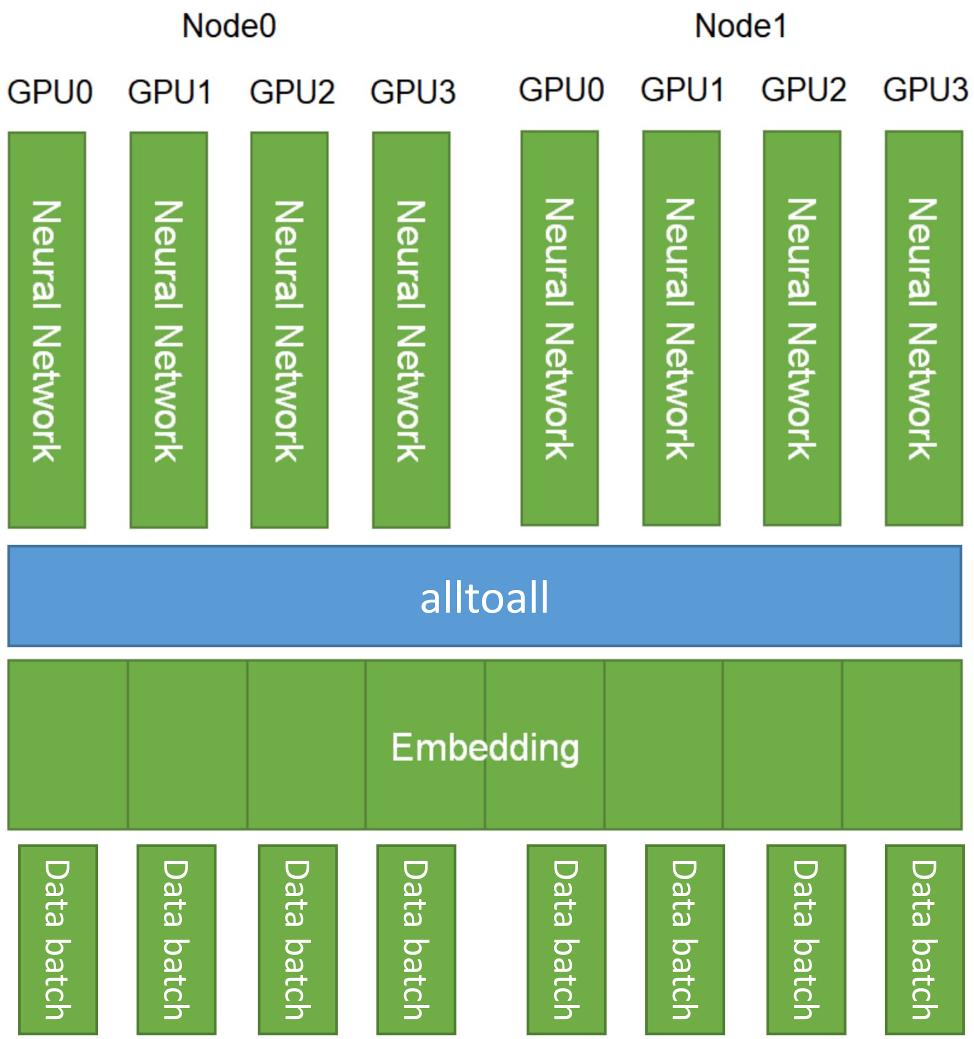
- category indices
- e.g., movie type, user demographics

[2] Cheng, Heng-Tze, et al. "Wide & deep learning for recommender systems." *Proceedings of the 1st workshop on deep learning for recommender systems*. 2016.

ByteDance

Distributed recommendation model training

- Synchronous parallel training
  - Neural network is replicated across workers
  - Embedding is sharded across workers
  - Each worker is responsible for embedding lookup for the shard it owns
  - Use **alltoall** to communicate embedding lookup results to form the embeddings of each data batch



Oldridge, Even, et al. "Merlin: A GPU Accelerated Recommendation Framework." *Proceedings of IRS*. 2020.

ByteDance

Design choices for distributed training

- Synchronous v.s. asynchronous parallel training
  - whether the workers iterate training steps with synchronization or not
  - sync training leads to better reproducibility, model convergence but worse system performance
  - requires different communication primitives
    - sync: alltoall, allreduce
    - async: push/pull, gather/scatter
- Embedding placement on devices: GPU v.s. CPU v.s. SSD
  - Each device type has its own capacity-bandwith characteristic
  - implications for communication: src and dst may be on different devices, opportunity for topology-aware optimization

ByteDance

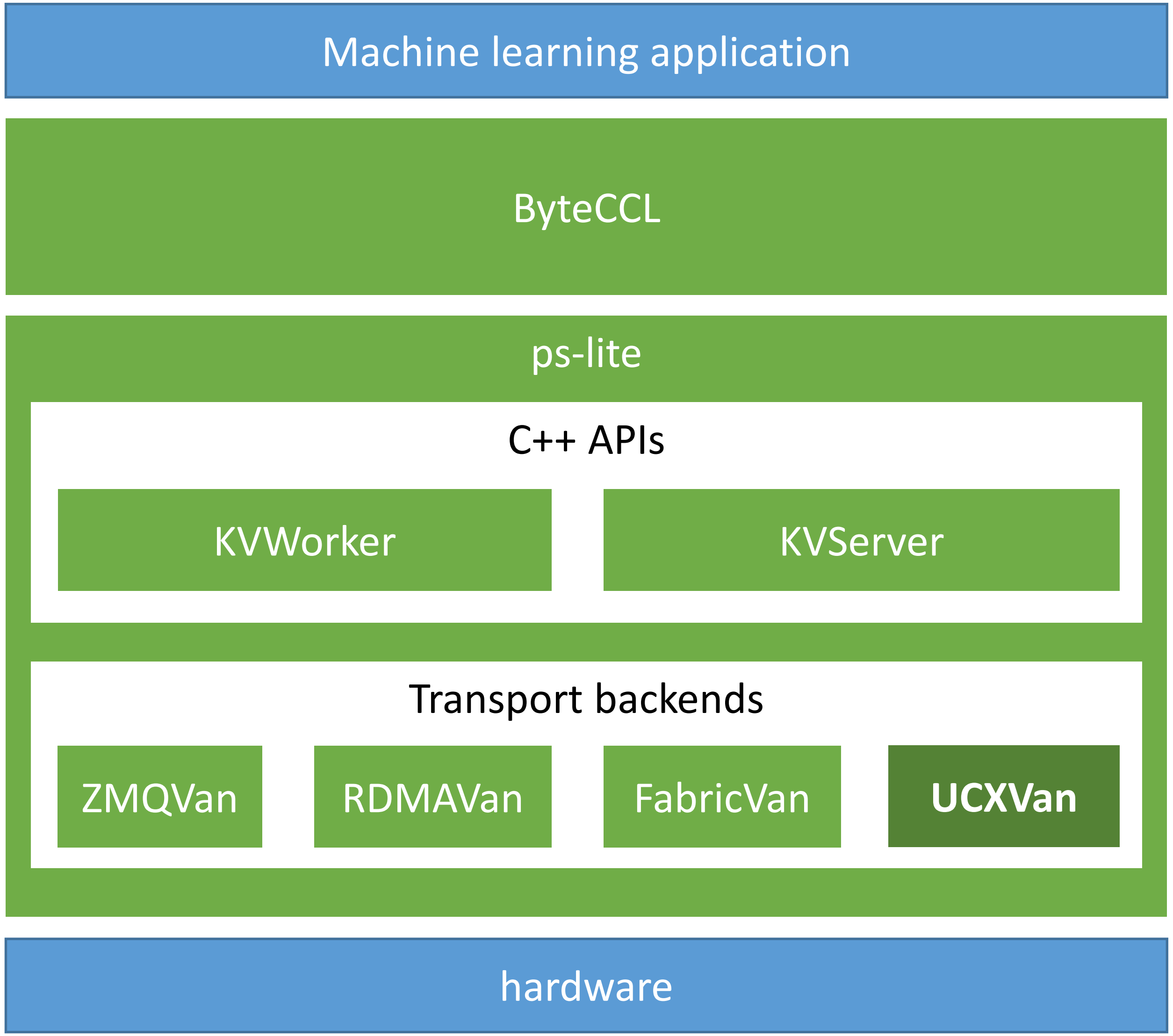
ByteCCL overview

- Developed on top of BytePS[3] with an extended set of communication primitives
  - allreduce, alltoall, gather, scatter, allgather, broadcast, send & recv
- Supports both sparse and dense neural network use cases
  - computer vision, natural language processing, speech, recommendation models
- Supports sync and async training
- Integrated with multiple machine learning frameworks
  - Tensorflow, PyTorch, MXNet
- Supports and optimized for multiple hardware (CPU & GPU)
  - e.g., alltoall variants: CPU to CPU, GPU to GPU, CPU to GPU, GPU to CPU

[3] Jiang, Yimin, et al. "A Unified Architecture for Accelerating Distributed {DNN} Training in Heterogeneous GPU/CPU Clusters." *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 2020.

ByteDance

ByteCCL Architecture

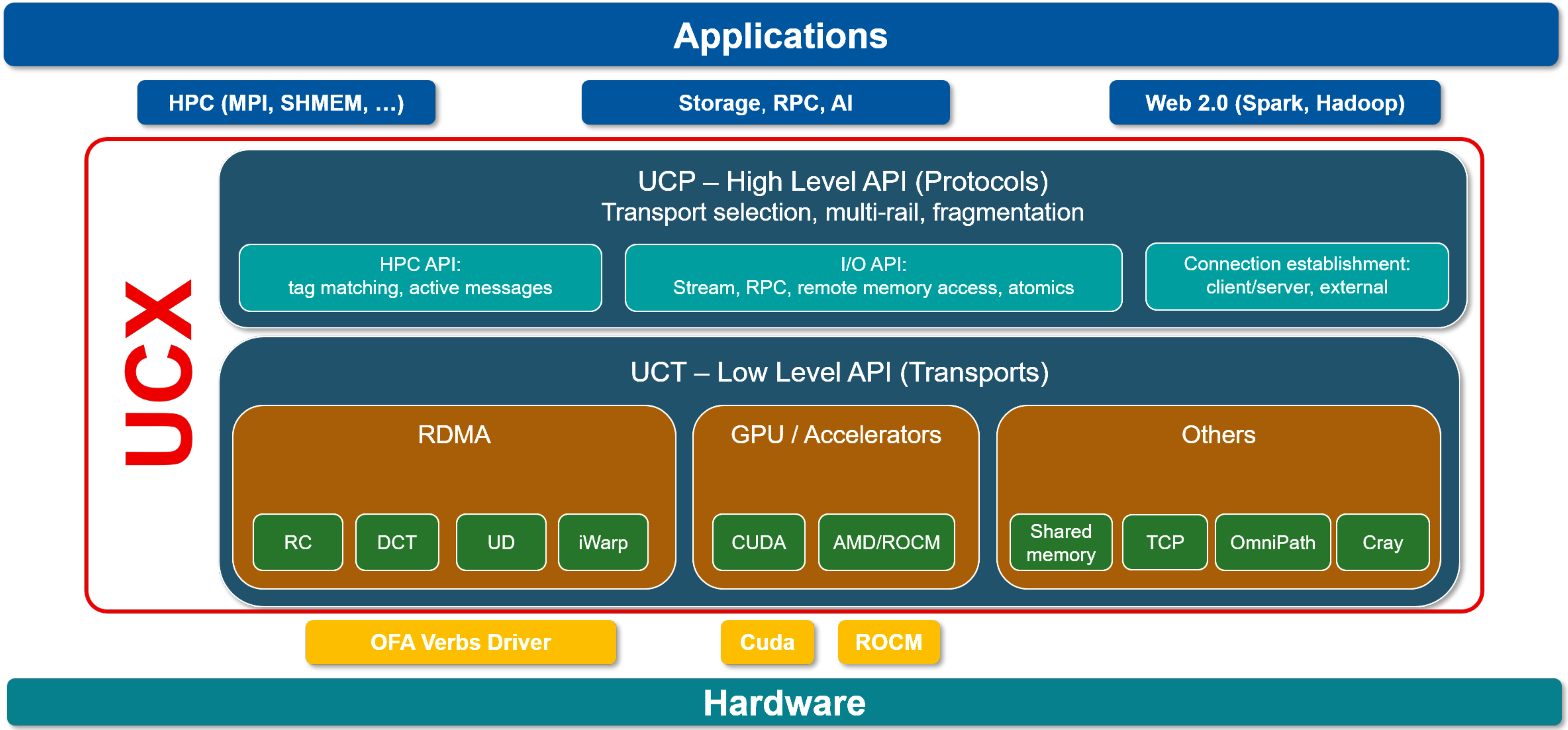


- ByteCCL
  - Tensorflow/PyTorch framework integration
  - Memory management
  - Collective algorithms and scheduling
  - Provides python APIs (allreduce, alltoall, etc)
- ps-lite C++ interface
  - Control plane - ps::StartPS, ps::Finalize
  - Data plane - ps::KVWorker::ZPush, KVWorker::ZPull
- ps-lite transport backends
  - ZMQVan: TCP
  - RDMAVan: RDMA, shared memory IPC
  - FabricVan: AWS EFA
  - UCXVan: tcp, rc_x, cuda, shm, etc

Why UCX

- Supports various platforms and networks
- Simple yet rich API, masking low-level details of RDMA programming
- Advanced protocols for transfer messages of different sizes
- Automatic selection of best available transports and devices
- Multi-rail support
- DC support
- GPU memory support
- Zero-copy with registration cache

Why UCX


Applications
HPC (MPI, SHMEM, …)
Storage, RPC, AI
Web 2.0 (Spark, Hadoop)
UCX
UCP – High Level API (Protocols)
Transport selection, multi-rail, fragmentation
HPC API:
tag matching, active messages
I/O API:
Stream, RPC, remote memory access, atomics
Connection establishment:
client/server, external
UCT – Low Level API (Transports)
RDMA
RC
DCT
UD
iWarp
GPU / Accelerators
CUDA
AMD/ROCM
Others
Shared memory
TCP
OmniPath
Cray
OFA Verbs Driver
Cuda
ROCM
Hardware

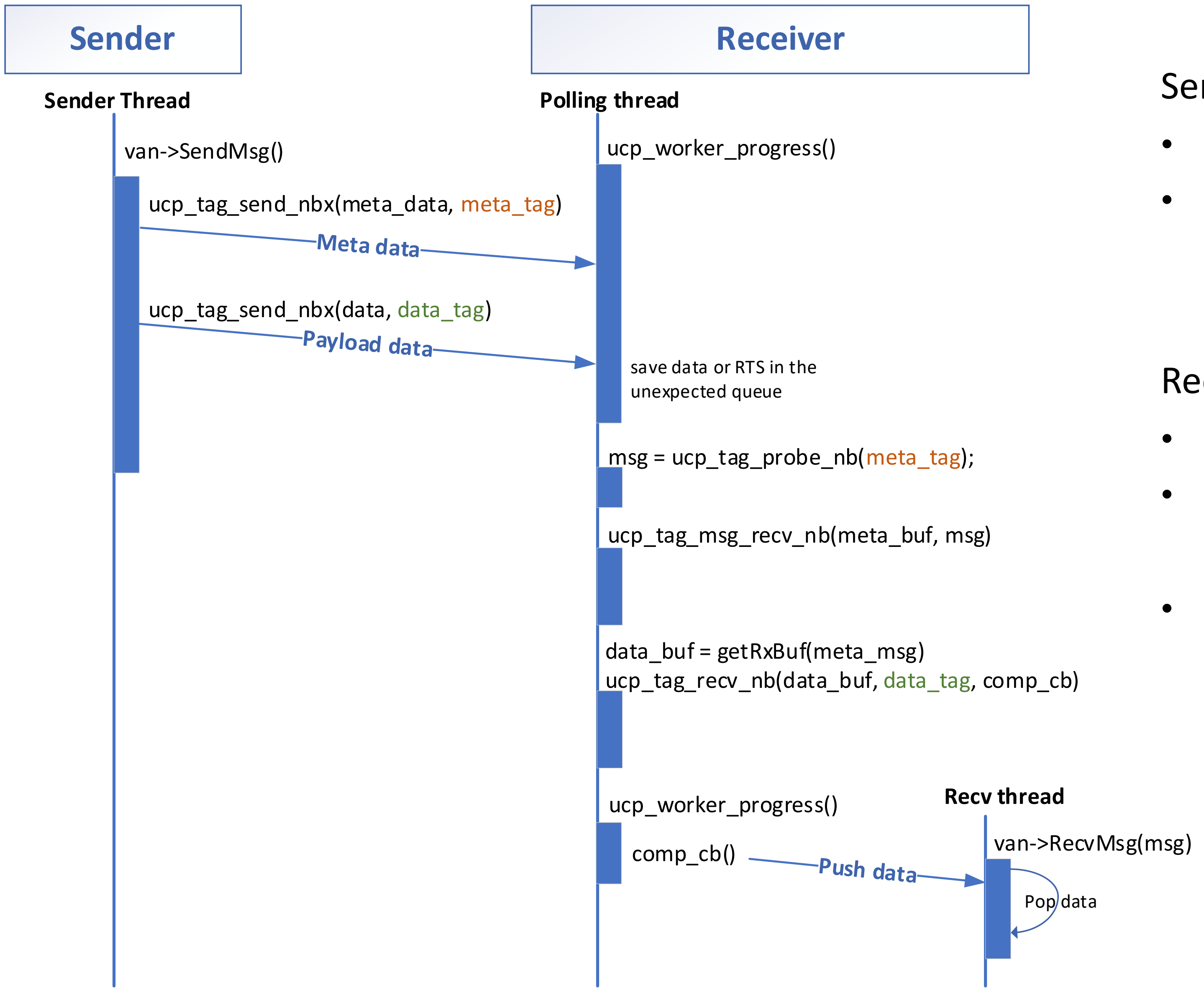
ByteDance

UCX integration for ps-lite (overview)

- UCX van implements Van abstraction
- Client-server connection establishment mode
- Several threads, two multi-thread support schemes:
  - Common worker for all threads
  - Send and receive threads use different workers
- Communication is based on TAG API:
  - Send data with ucp_tag_send_nb()
  - Poll for incoming data with ucp_tag_probe_nb()
  - Receive data with ucp_tag_recv_nb()

UCX integration for ps-lite (overview)

ByteDance



Sender (sender thread)

- Sends meta data for every message with meta_tag
- Send message itself with data_tag

Receiver (polling thread)

- Probes for incoming packets with meta_tag
- Receives meta data and posts receive operation for the packets with data_tag
- Pushes received message to thread safe queue

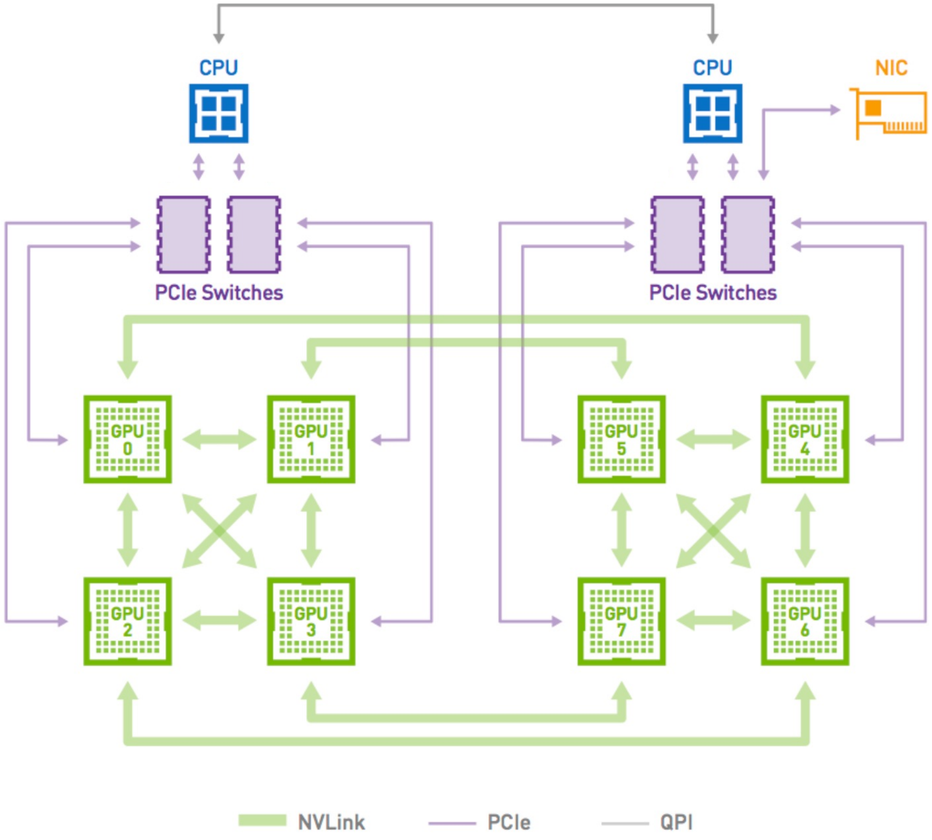
UCX integration for ps-lite (optimizations)

- Short protocol for small data
- Separate communication context per GPU device
- Pre-pinning of memory
- Separate UCX workers for send and receive threads
- Lockless thread-safe queue

ByteDance

Optimizations in ByteCCL

- Topology-aware optimizations with GDR and IPC
  - Imbalanced NIC affinity: turn on GDR if the GPU and the NIC shares the PCIe switch
  - Intra-node transfers: use shared memory
- Concurrent primitive supports
  - Multiple collective operations can happen concurrently
  - e.g., concurrent allreduce & alltoall
- More asynchronous
  - async requests are tracked and processed via UCX callbacks
  - collective operations do not require explicit synchronization



ByteDance

Performance results

- CPU recommendation model training
  - RDMA cluster with 64 CPU nodes
  - Each node with CX-6 200Gb/s NIC x 1
  - Recommendation model A, model size: ~1 TB
  - **12%** end-to-end speedup compared to horovod with HPCX

- GPU recommendation model training
  - RDMA cluster with 16 GPU nodes
  - Each node with CX-6 200Gb/s NIC x 4, NVLink A100 x 8
  - Recommendation model B, model size: ~4 TB
  - **8.6%** end-to-end throughput increase compared to NCCL

ByteDance

Pain points and future work

- Intra-node performance is low
  - For both cpu-cpu transfer and cpu-gpu / gpu-cpu transfers
- Unclear configuration prefix support
  - e.g., PREFIX_UCX_RDMA_CM_SOURCE_ADDRESS
- Multi-rail traffic load balancing
  - Some NIC have more (non-UCX) TCP workload than others, how to let UCX know this?

ByteDance

Acknowledgements

- Special thanks for supports from NVIDIA/MLNX
  - Yong Zhuang, Marina Varshaver, Akshay Venkatesh, Devendar Bureddy, Bin Lei, Yossi Itigin, Tal Nada Shalabi, Miao Wang, and many others