



CONFERENCE

—NORTH AMERICA 2026—

PyTorch-Native Feature Transformation and Training Framework
for Uber Eats Recommendation

Ke Chen, Xandra Zhu, Peng Zhang, Peng Chen

Agenda

01 Context & Background

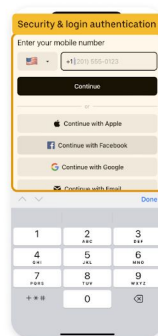
02 PyTorch Native Data Transformation

03 Data Loading Improvements

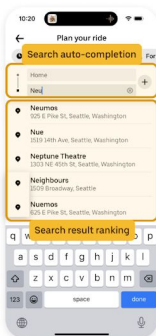
04 Model and Training Framework Co-design

05 Summary and Next Steps

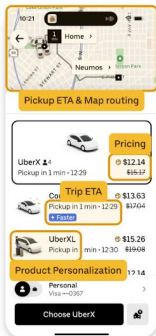
Deep Learning Use Cases in Uber



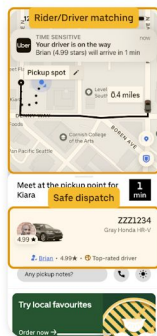
Login



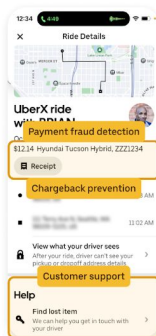
Search



Booking

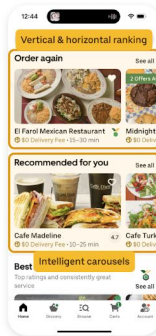


Trip

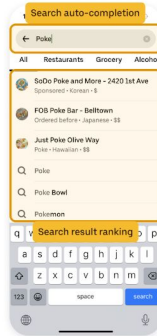


Post-trip

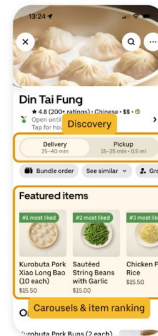
Uber Ride



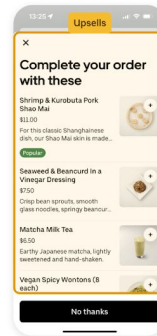
Home feed



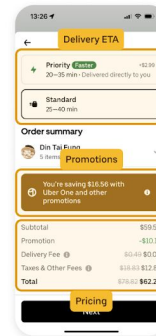
Search



Storefront



Pre-checkout



Checkout

Uber Eats

Deep Learning Model Evolution in Uber

Before 2019

2020-2024

2025+

Model parameters

- 10 M -100 M

- 100 M ~ 1 B

- 1 B - 30 B+

Training Approach

- Data parallelism
- GPU training

- Scale up data and model
- ONNX/TRT Packaging

- ZeRO / Model parallelism
- Incremental training
- Transfer learning

New Framework

- Spark ML
- Tensorflow / Horovod

- PyTorch / Lightning
- **Ray (Core/Train/Data)**

- **DeepSpeed / FSDP**
- Mix precision
- Kernel fusion

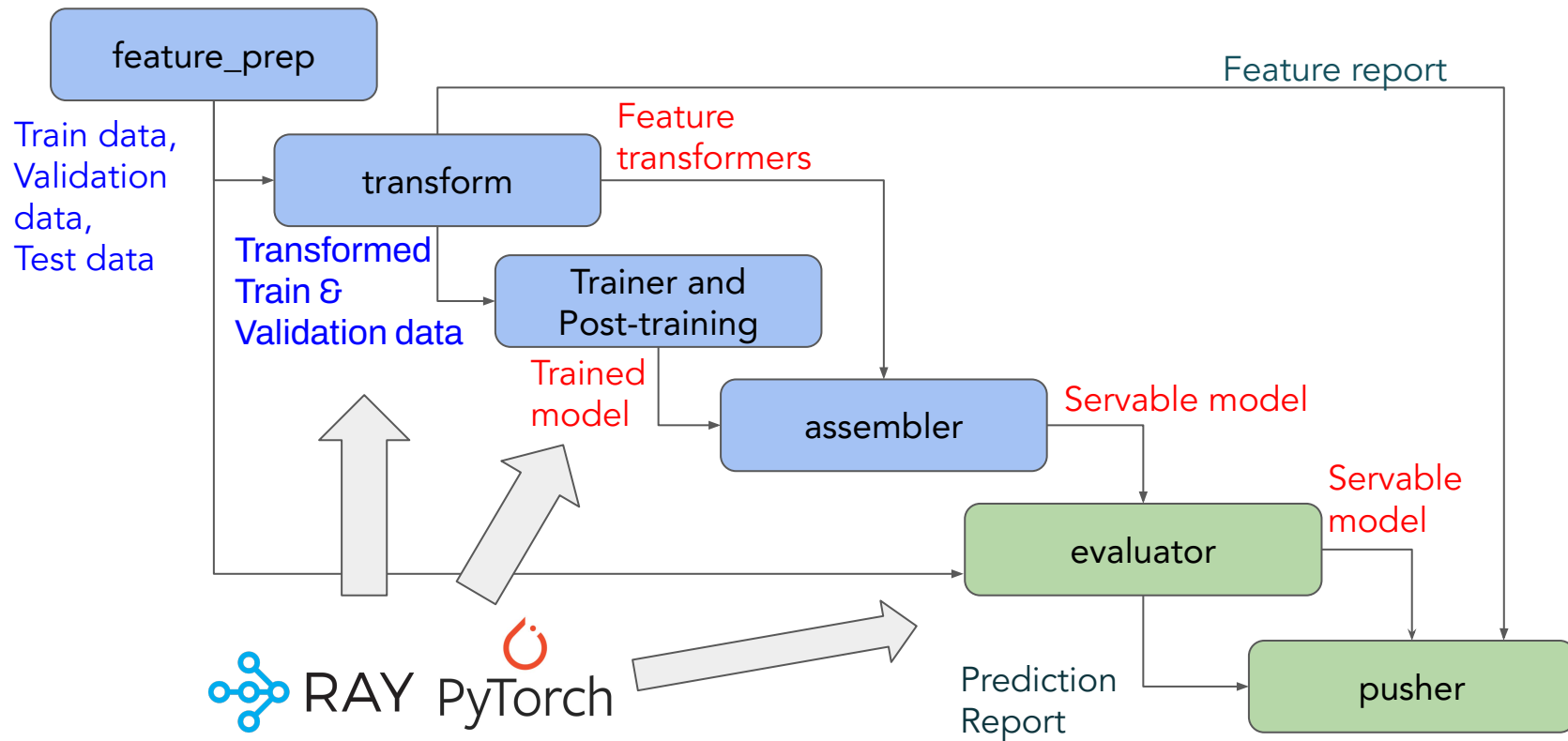
New HW

- NVidia 1080 Ti GPU
- NVidia P6000 GPU

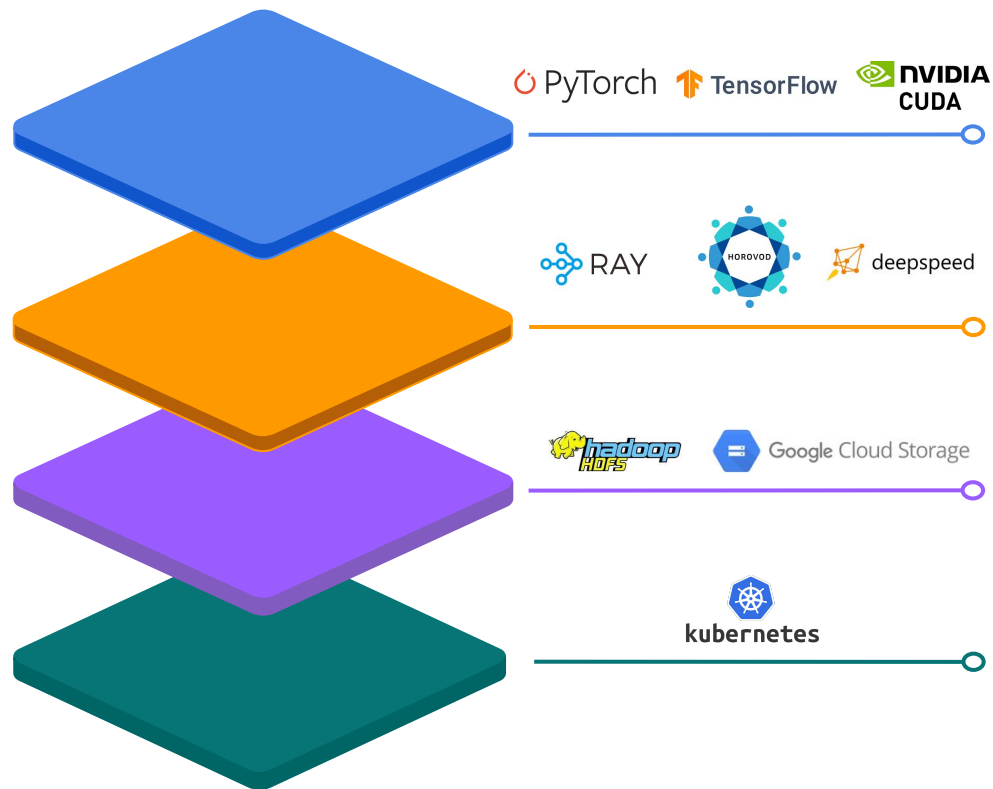
- NVidia RTX5000 GPU
- NVidia A100 GPU

- NVidia H100 GPU
- NVidia B200 GPU

Michelangelo's Core Training Flow



Michelangelo's Training Architecture Layers



DL framework

Flexible Model generation and prototyping

Distributed Training

Scalable and reliable multi-GPU cluster and data loading

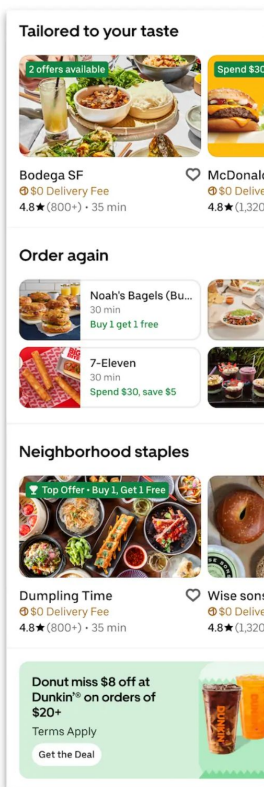
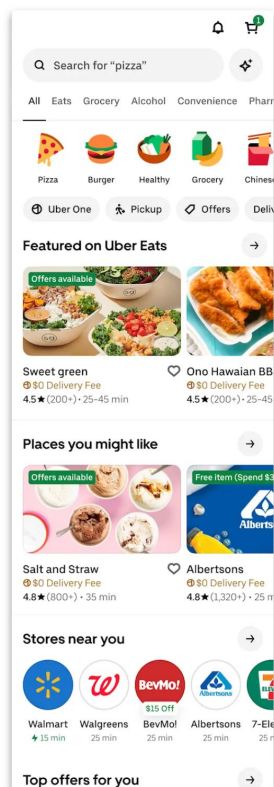
Storage Layer

Multi-region storage access and feature loading

Compute Layer

Multi-provider Multi-hardware Job scheduling

UberEats Recommendation improvements



From **static, pointwise ranking** to a **transformer-centered Generative Recommender** that uses behavioral sequences, near-real-time features, and scores multiple restaurants simultaneously.

This needs ML infrastructure to be more context-aware, scalable, and more computationally efficient.

Uber Blog: <https://www.uber.com/us/en/blog/next-gen-restaurant-recommendation/>

Overall Improvements

A training iteration has multiple steps: data transform → data loading → compute → communication.
To support large model training, we need optimizations for these components to improve training efficiency and GPU utilization.

DATA TRANSFORM / LOADING

- PyTorch Native Data Transformation
- Petastorm → Ray data loader
- Dynamic sequential features (ragged inputs)

COMPUTATION

- Model Parallelism (e.g. FSDP)
- Mixed-precision training
- Flash Attention
- Multi-hash embeddings
- High-end GPU SKUs (e.g. B200)
- GPU kernel fusion (e.g. torch.compile)
- Large batch training
- Activation checkpointing

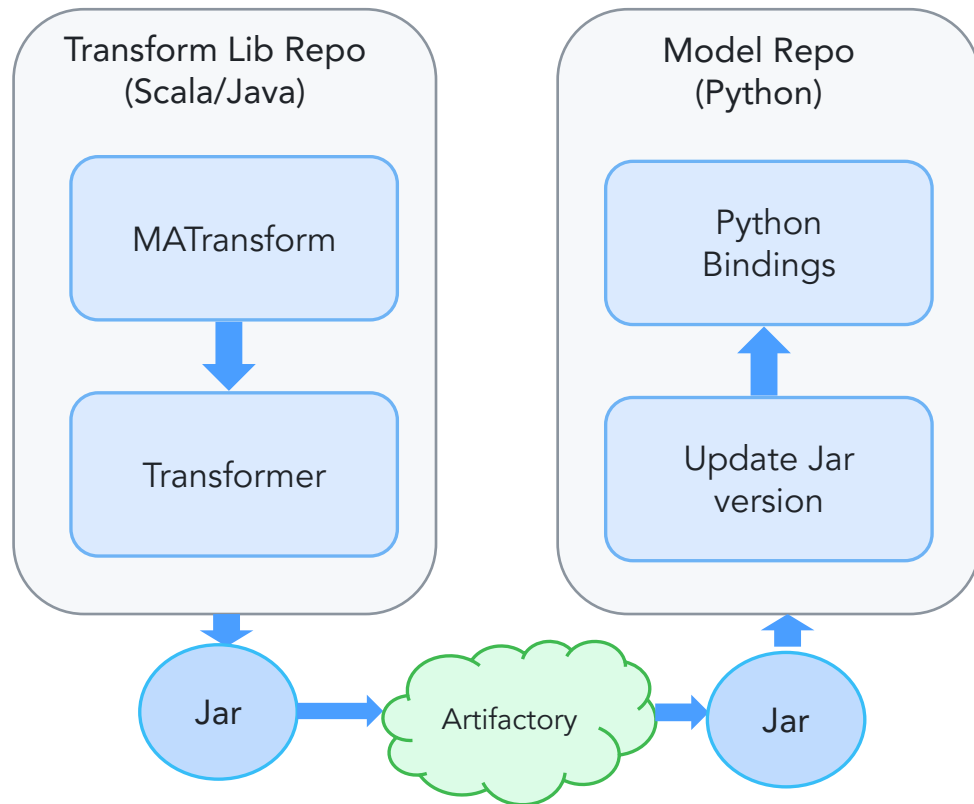
COMMUNICATION

- Data parallelism (DDP)
- High-speed interconnects (NVLink ≫ PCIe ≫ NIC)
- B200 RDMA
- Hybrid parallelism (e.g. HSDP)

PyTorch Native Data Transformation

Spark Data Pre-processing Pain Points

- Complex to add: Scala/Java + JAR release.
- Hard to debug
- Additional latency for serving.
 - Online: sequential processing over stages.
 - Offline: data conversion (Dataframe, Neuropod).
- Inconsistency: separate implementations for online (Java) & offline (Python).
- Prevents serving the whole model on GPU.



PyTorch Native Transform

- Uses native PyTorch framework layers to preprocess DL model input features
 - Leverages high tensor throughput
 - Unifies with the **main deep learning model**
 - Enables GPU serving (online and offline) for low latency
 - Ensures online / offline consistency

TRANSFORM LAYERS

STATEFUL

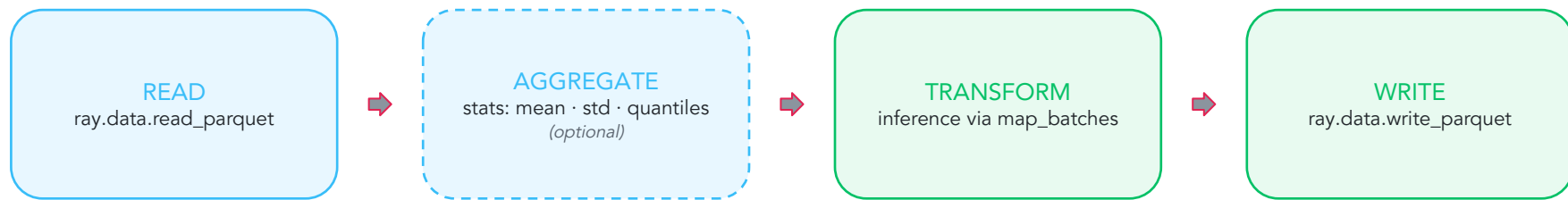
- Normalization
- Min-Max scaling
- Bucketization
- ID / vocab indexing

STATELESS

- Clip · Floor · Ceil
- Scale · Divide · Log
- Concat · Stack · Tile
- Pad/Crop · Fill-None

PyTorch Native Transform on Ray Data

- Python-native, Arrow-based framework for distributed, streaming batch processing.
- Splits datasets into blocks across a Ray cluster and executes parallel operations.
- Uses aggregate for statistics computation.
- Uses map_batches for parallel data transformation.



Why Ray Data for Feature Preprocessing

PYTHON-NATIVE API

Transform logic written in Python/PyTorch (same as model training).
No JVM boundary — runs entirely in Python process.
No Py4J serialization overhead on every batch.

ZERO-COPY ARROW FORMAT

Ray Data uses Apache Arrow natively. Arrow buffers share memory with NumPy and PyTorch for compatible numeric tensors.
No intermediate Pandas round-trip.

PIPELINE FUSION

Multiple transform layers fused into one Ray task.
Read → Transform → Write streamed E2E — no materialization.
Single continuous Ray Data pipeline — no separate job per stage.

MEMORY EFFICIENCY

Python processes — no JVM heap or GC pauses.
Shared object store — one copy per node, shared by all workers.
Streaming backpressure — minimal disk spill, reduced OOM.

TRAIN = SERVING

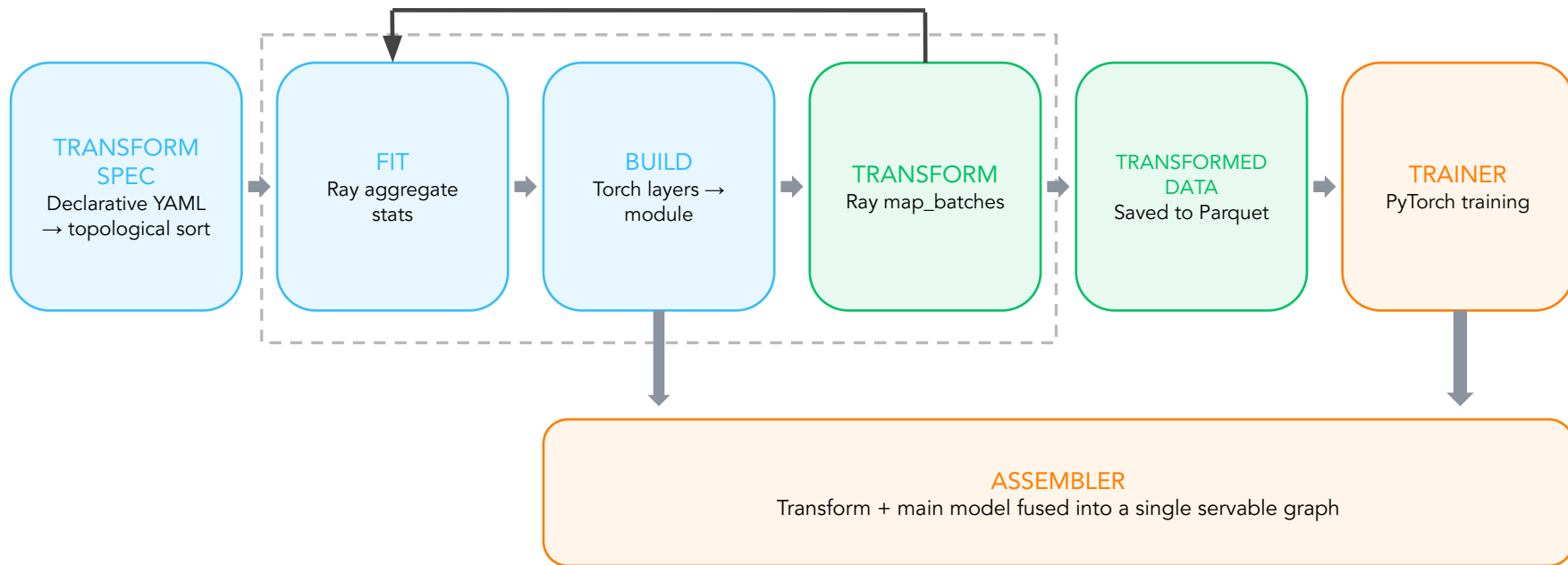
Runs PyTorch module directly for offline inference — no bridge, no reimplementation.
Identical transform logic and weights in offline and online serving.

DYNAMIC SCHEDULING

Concurrent tasks/actors placed dynamically across Kubernetes workers.
No fixed Spark executor allocation, no shuffle service bottleneck.

PyTorch Native Transform Workflow

FIT, BUILD, and TRANSFORM repeat once per topological level; the resulting layers are combined with the trained model for serving.



PyTorch Native Transform Performance Tuning & Results

RAY PERFORMANCE OPTIMIZATIONS

4.2× Scheduling-loop optimization

Batch metadata fetch + incremental usage updates

2.1× Larger read blocks

Set block size to 512 MB to reduce scheduling overhead

1.4× Increase rows per output file

Lower file count to reduce CFS metadata overhead

1.2× PyArrow batch format + fused tasks

Reduced data conversion and task execution overhead

CUMULATIVE RESULT

15×

23 h → 1.5 h

feature processing runtime

PRODUCTION PIPELINE

4.5B rows · 102 TB

full-scale input data

≈11 hours

100 workers (fewer resources)

19× less CPU-time

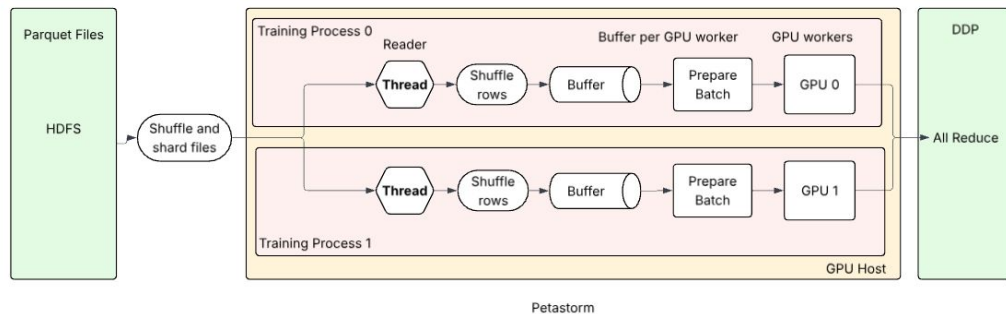
3× less memory vs. Spark

Data Loading Improvements

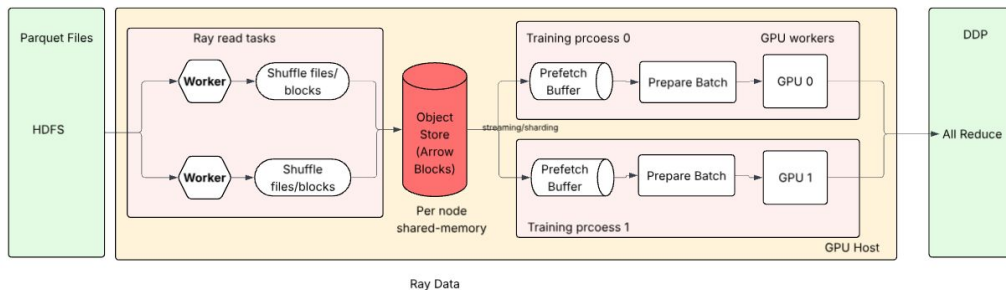
Data Loading: Why Ray Data for PyTorch

- Petastorm loads, shuffles, and batches data inside each training worker
- Ray Data reads as separate tasks and writes blocks into a shared object store
- I/O parallelism is no longer tied to GPU count
- Shuffle: File/buffer/block/global (row) level
- Reproducibility: `preserve_order` for deterministic ordering
- Improved debugging and observability

PETASTORM · loading inside each training worker



RAY DATA · separate read tasks → object store



Data Loading: From Migration to Training Gains

PERFORMANCE OPTIMIZATIONS

Remove CPU batch preparation from the critical path

Producer thread prefetches batches in parallel with GPU training

Workload-specific configuration tuning

Read block size, task count, concurrency, and prefetching

Shadow validation

Model and performance parity before migration

PERFORMANCE GAIN

8–66%

faster training vs. Petastorm
after targeted optimization across
models/pipelines

Resume Training from Checkpoints

Resume from the latest useful checkpoint (instead of restart training)
Reduce wasted compute on transient training failures & preemptions

PER-EPOCH

Checkpoint after every epoch
Automatic intra-run retry
Tracked in ArtifactStore
Useful for multi-epoch training job

ASYNC

Overlaps I/O with training
CPU clone + background write
Atomic save to CFS
Next epoch starts immediately (no time overhead)

MID-EPOCH

Checkpoint at chunk boundaries in each epoch
Deterministic file-group chunks
Resume next unfinished chunk
Fault tolerant for long training epoch (large model & data)

Failure / preemption



Find latest recovery checkpoint



Resume training

Platform Reproducibility Improvements

- Achieved reliable model evaluation metrics across pipelines using the same data set ($\ll 1e-05$)
 - Reproducible experiments
 - Resuming from earlier data loading states
- Consistent distributed training convergence curves
 - Identical model weights and structures
 - Seed everything in PyTorch Lightning
 - Seed Ray Train
- Reproducible Ray datasets
 - Seed distributed data loader
 - Input file ordering and shuffling
 - Ray data block preserve_order
 - Seed data batch shuffling



Two (red and purple) distributed training pipelines have consistent training loss metrics at every step

Model and Training Framework Co-design

Migrating Uber Eats Modeling From TensorFlow to PyTorch

Motivation: Pytorch is better in "Generative Recommender" (GenRec) support

Value:

- GenRec learns from historical business actions for user intent predictions¹
 - More accurate than using human-engineered features
- Native distributed training & LLM tooling (FSDP/DeepSpeed, FlashAttention)
- Flexible inference framework (TorchScript, Onnx, TensorRT)

Innovation: Efficient large-model training and serving

- Increased training throughput by 65x
- Without increasing infrastructure costs

ROI: Turned computing investment directly into improved recommendations

1. behavioral features are processed under Uber's Privacy Policy and data retention standards; no individually identifying data is used outside Uber's approved purposes

From Deep Learning Recommendation Model to GenRec

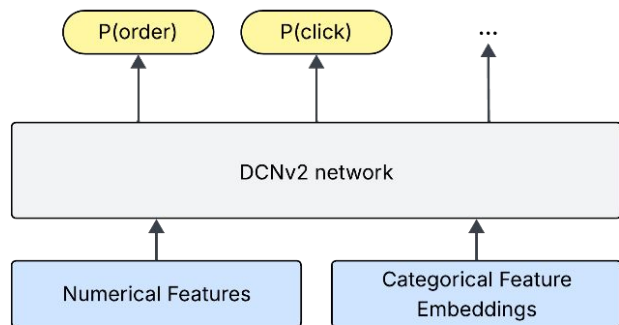


Image (a): Traditional Pointwise Architecture

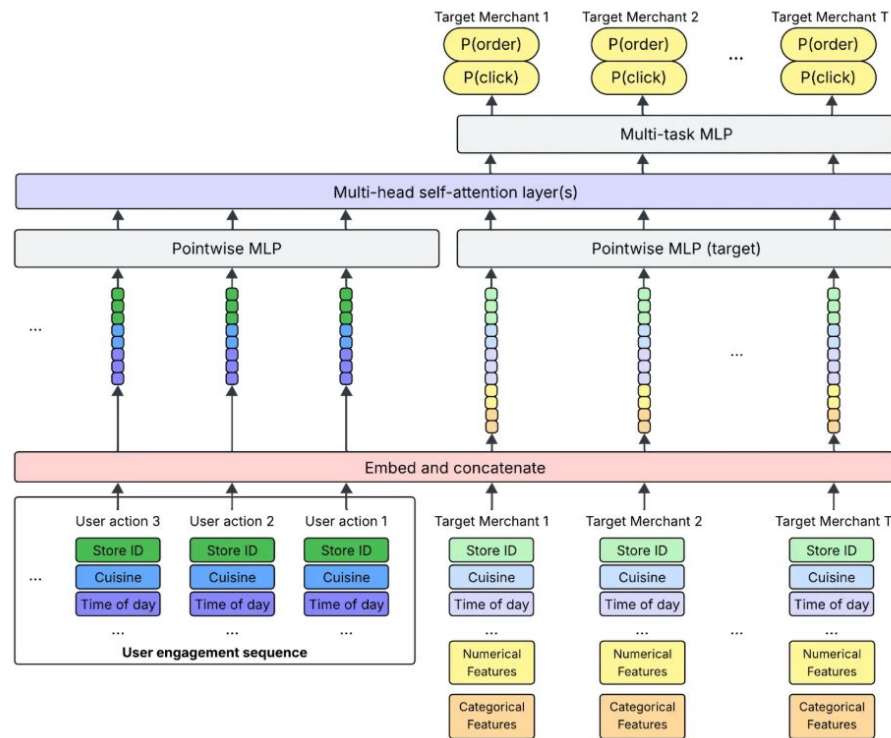


Image (b): Listwise GenRec Architecture

Uber Blog: <https://www.uber.com/us/en/blog/next-gen-restaurant-recommendation/>

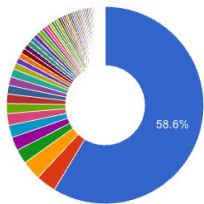
Dynamic Sequential Feature Padding in Training and Serving

- User actions are of varied lengths
- Padding to a global length can throttle especially inference traffic
- Padding to the longest sequence per batch for TensorRT serving
 - Saves memory for highly skewed seq_length and user_count distribution
 - Avoids wasted attention computations over padding tokens
 - Improves data loading efficiency for loading features with padding tokens
 - Training throughput improvement: **1.4x** (samples/GPU/seconds)

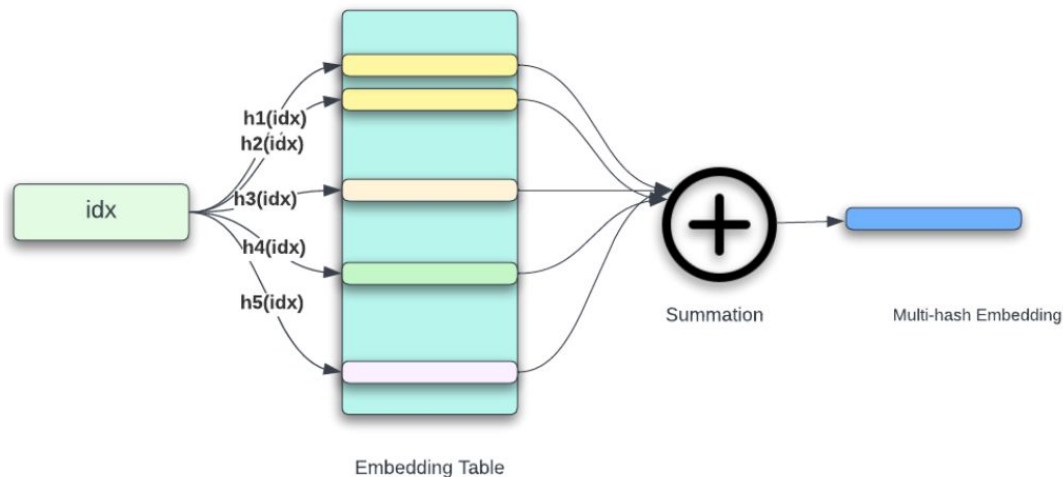
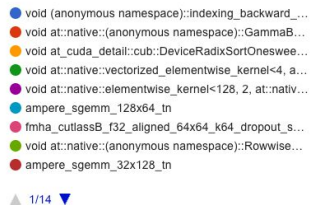


Modeling Optimizations: Multi-hash Embedding

- Faster in indexing an embedding for sequential feature tokens
 - Reduce embedding lookup/assembly time with comparable model metric performance
 - Training throughput improvement: **2.8x**



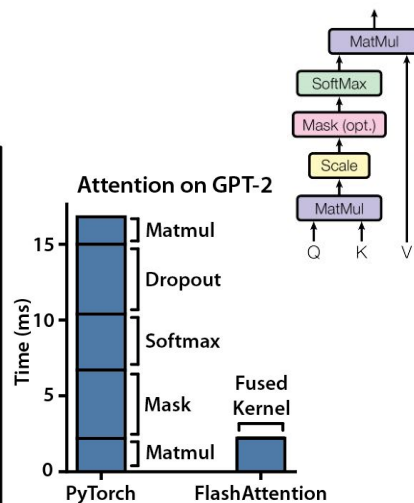
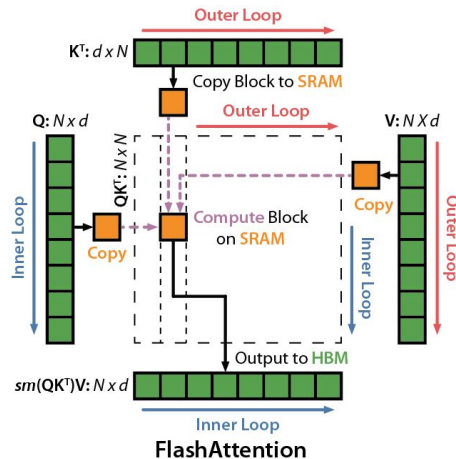
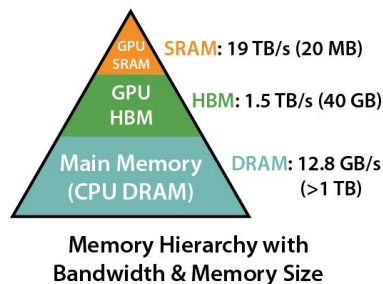
Direct indexing was over 50% Cuda kernel time



Multi-hash Embedding of token idx using 5 hash functions for parameter sharing

Modeling and Memory Bandwidth Optimizations: FlashAttention

- FlashAttention v2 in Sequential Transformer Modeling
 - Tiling: keep the computation within fast SRAM (~10x faster than HBM)
 - Kernel fusion: compute Ops (query-key dot, softmax, dropout, weight-sum) in one CUDA kernel
 - Training throughput improvement: 1.4x



$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

Images from <https://github.com/dao-ailab/flash-attention>
And <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>

Distributed Training Infrastructure Optimization: B200 + RDMA

Why it Helps

NVIDIA Blackwell B200 GPU dual-die compute

192 GB HBM3e for larger local batches

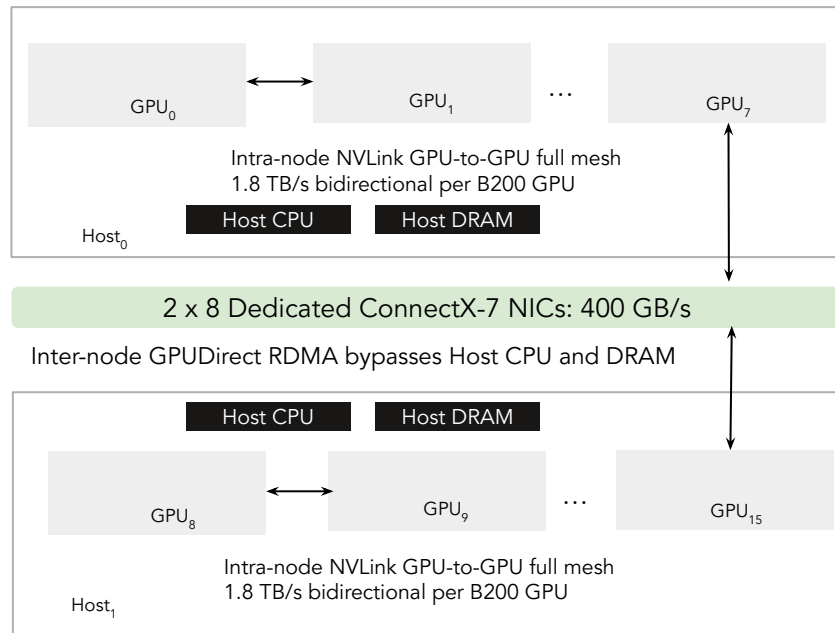
5th Generation NVLink

RDMA (Remote Direct Memory Access) for efficient inter-node communication

Training Throughput Improvement

~4x

v.s. H100 without RDMA



NVLink for intra-node, RDMA for inter-node GPU-to-GPU communication

Training Operator Optimization: Torch.compile and Kernel Optimization

What Changed

SDPA Kernel fusion + operator scheduling + CUDA graph optimization

Reduced memory ops and CPU kernel-launch overhead

Workflow Improvement

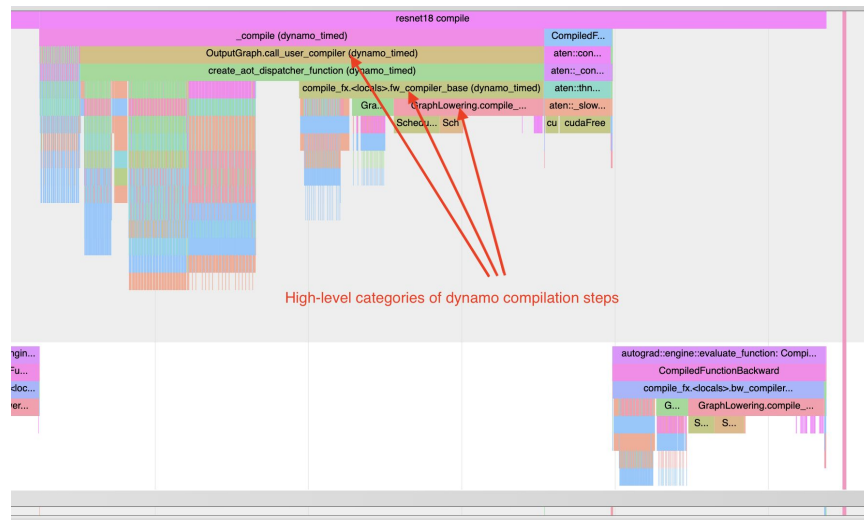
Model compilation for enhanced (distributed) training throughput

Eager model for TorchScript and TensorRT inference compatibility

Training Throughput Improvement up to

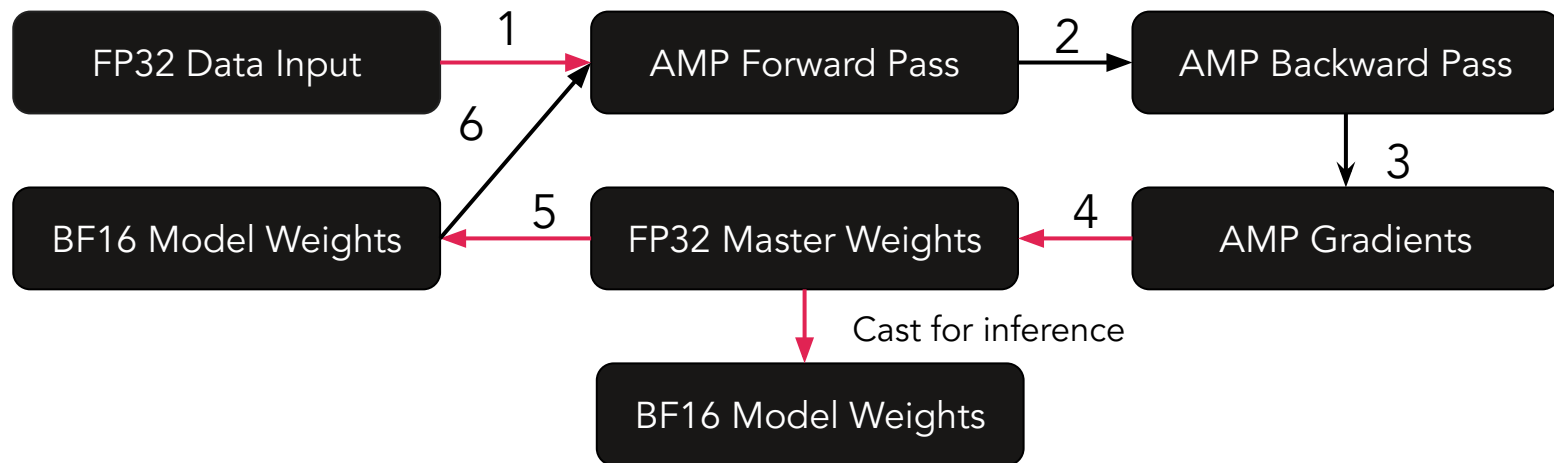
~3x

v.s. the raw Torch graph



Training Memory and Compute Optimizations: Mixed/low Precision Training

- Automatic Mixed Precision (AMP) training and bf16 serving
 - Forward and backward passes use AMP for speed and lower memory
 - BF16 for Fast / Safe Ops (Linear, MatMul, Activation)
 - FP32 for Precision-Sensitive Ops (Softmax, LayerNorm)
 - Master weights and optimizers use FP32 for precision
 - Training throughput improvement: **1.6x**



Training Memory Optimization Toolkit: Activation Checkpointing

The Motivation for Memory-bound Workloads

Activation memory can dominate HBM in addition to weights + optimizer + gradients

How it Helps

Selectly discard intermediate activations during a forward pass

Recompute during the backward pass

Trade additional computation for memory saving

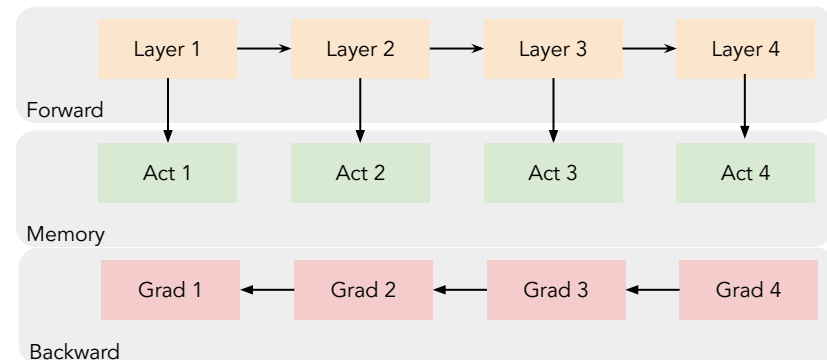
No numerical convergence loss

Reduced HBM (Peak) Memory by up to

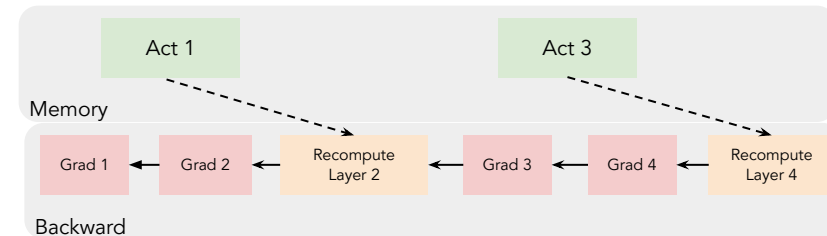
40%

at 5% extra compute

Standard Forward & Backward Pass



Activation Checkpointed Act1 and Act 3 Forward & Backward Pass



Distributed Training Toolkits: FSDP2/DeepSpeed and Hybrid Sharing

Intra-node Modeling Sharding

Shard parameters, gradients, optimizer states within one node

All-Gather for parameters via NVLink

Reduce-Scatter for gradients within each node via NVLink

Inter-node Data Parallelism & Model Replication

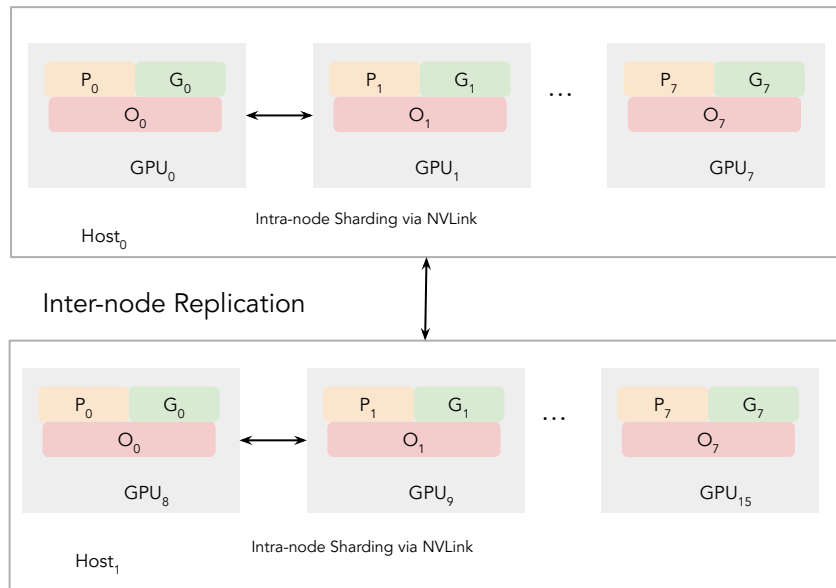
Replicates the shared model across nodes handling separate data batches

All-Reduce for gradients across nodes (via RDMA)

Composing with AC

(Selective) AC + FSDP2 or DeepSpeed

Watch out model metric performance and workflow compatibility



FSDP 2 / DeepSpeed hybrid sharing workflow

P, G, O correspond to parameters, gradients, and optimizer states

Key Takeaways

- Tune Ray Data loading, network, Ray Trainer, checkpointing and logging configs based on the actual workload and profiling for better efficiency
- Explore the combination of DDP, Activation Checkpointing, FSDP/DeepSpeed strategies, and various batch sizes for steady production system configs
 - A larger batch size does not always indicate higher throughput
 - High GPU utilization does not always indicate healthy system performance
- Monitor data loading, scheduling time, failure recovery, GPU utilization, goodput, and MFU (Model FLOPs Utilization) metrics for system performance in addition to training throughput of individual jobs

Acknowledgement

Contributors: Yicheng Chen, Peng Ye, Kyle Koon, Arushi Arora, Evelyn Liu, Nolan Foster,
Frank Chen, Bekzat Abdulla, Jiapei Huang, Bo Ling, Di Yu, Haoyao Chen, Yun Zhang,
Anatoli Papirovski, Ashwinkumar Badanidiyuru Varadaraja, Himaanshu Gupta, Chris Harris,
Sally Mihyoung Lee, Zhitao Li, Eric Chen, Viv Keswani
& many other Uber team members

Thank You!!

Stay tuned for Uber's innovations from our open-source end-to-end
ML platform



Michelangelo AI

<https://michelangelo-ai.org/>