Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models

presented by Jiaqi Lou and Divya Koya

Outline

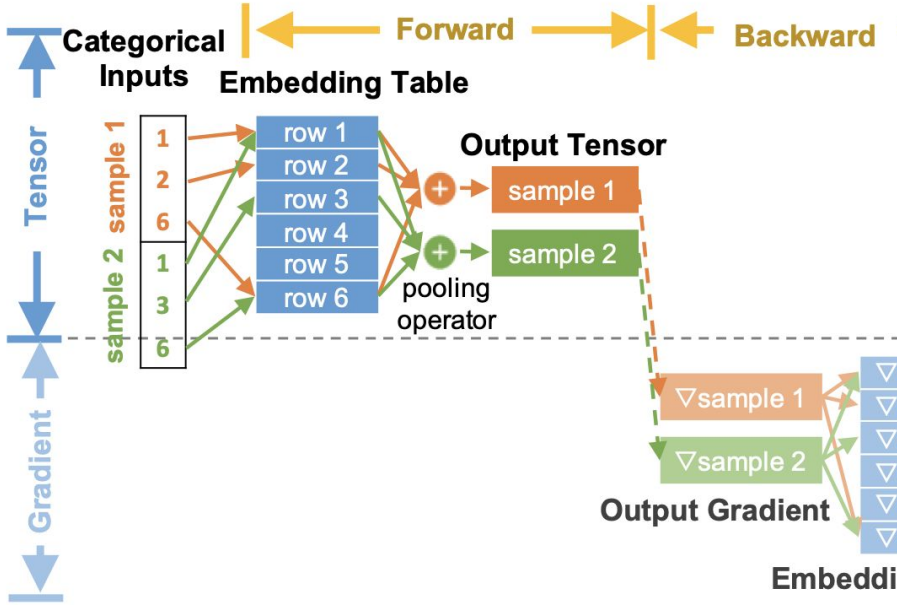
- Background
- Motivation
- Related Work
- Overview
- 4D Parallelism
- Embedding Optimizations
- Zion to ZionEx
- Implementation
- Evaluation
- Conclusion

Background

- **Deep learning recommendation models (DLRMs)** used in many companies and are single largest AI application in terms of infrastructure demand in data centers
- Conventional deep neural networks (DNNs) mainly compute-intensive, but DLRMs have **both compute-intensive components** and up to thousands of data-intensive **embedding operators**
- DNN training typically uses *data, model* or *pipeline parallelism* → not optimal for DLRMs
- Focus on **offline training**

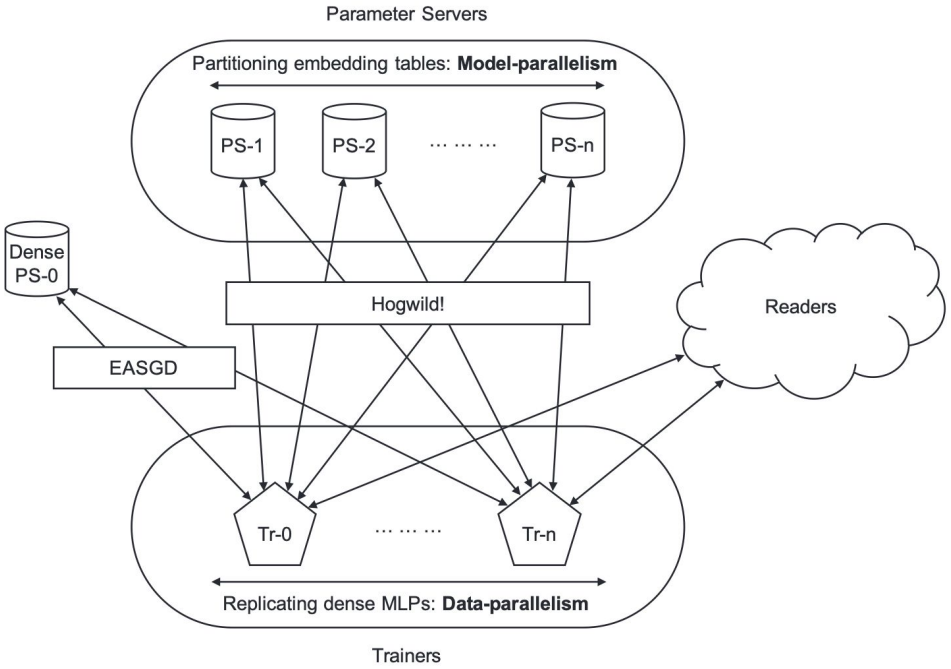
Background: Embedding Operators

- Each categorical feature has a dedicated **embedding operator**
- Takes multi-hot vector input, retrieves rows from embedding table, outputs pooled embedding vector



Background: Parallelization Strategies

- **Data parallelism** → dense parameters from MLP modules are duplicated
- **Model parallelism** → parameters from embedding tables are partitioned



Motivation

- Need to be able to support DLRMs with trillions of parameters, terabytes in size
  - PS-based systems cannot scale to this size → model accuracy decreases due to increased asynchronous updates
- Need:
  - A high-performance synchronous training solution
  - Memory-efficient computation
  - Scales while preserving accuracy of model

Related Work: Memory Optimization and Sharding

- **DeepSpeed** → fully shards model parameters, gradients and optimizer states across all nodes
- **GShard** → trains massive translation model with MoE, sharded across accelerators for tensor-level parallelization
- **FlexFlow** → automatic search to discover best operator parallelization strategy

❌ General systems, not designed for highly sparse recommendation models

Related Work: Systems for Sparse Recommendation Models

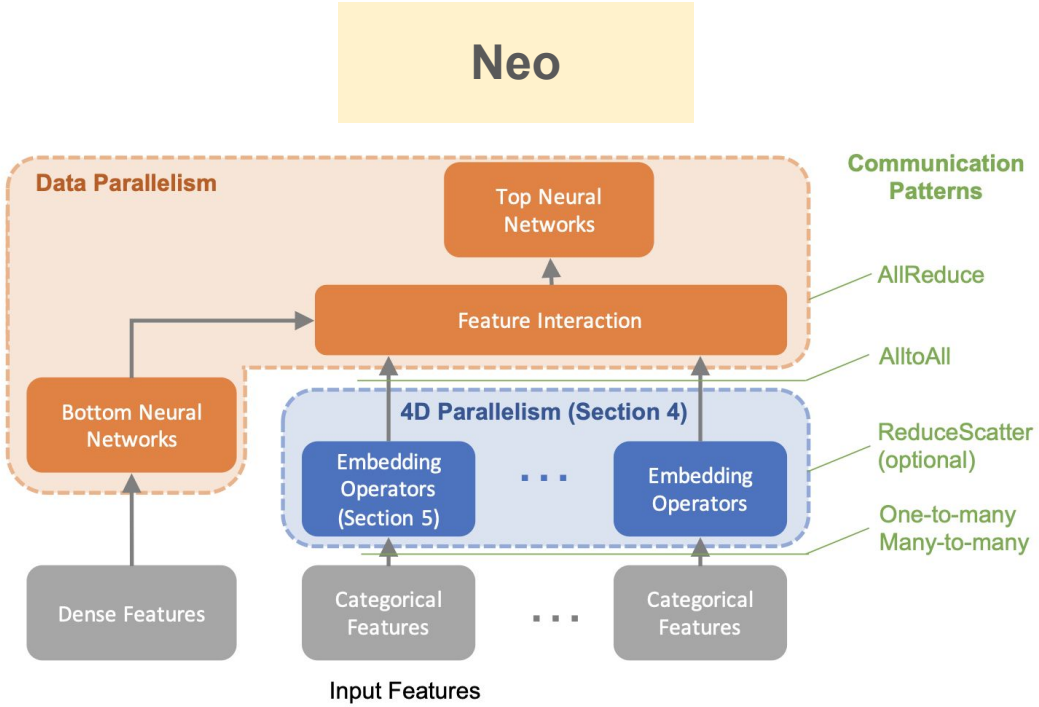
- **XDL** → hierarchical sample compression, workflow pipelining, zero copy and CPU binding
- **Kraken** (online training) → decoupled key-value fetching and embedding, codesigned cache eviction policy, memory efficient optimizers, non-colocated deployment model
- Work done in **CPU-based DLRM** training
- **Baidu's AIBox** → fits training into single node; pipelines network, disk and CPU/GPU tasks, hashing schemes

Related Work: Communication Performance

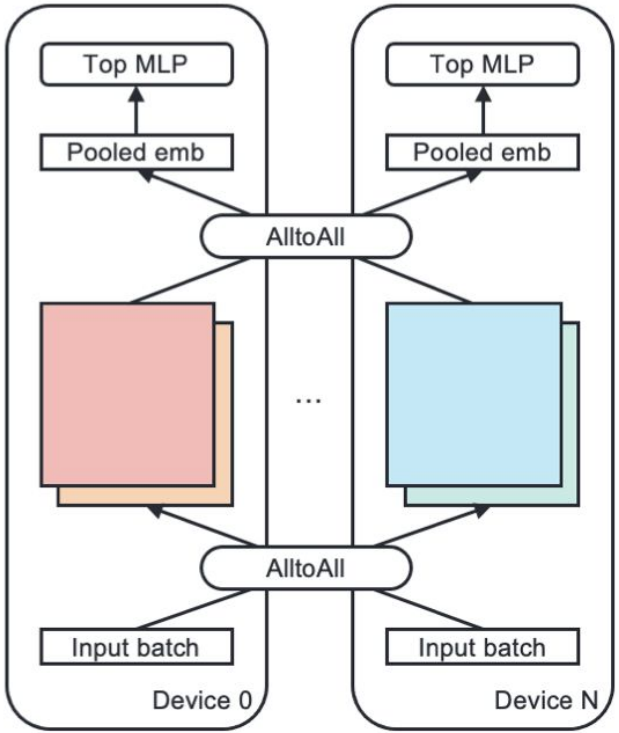
- **BytePS and ByteScheduler** → utilizes idle CPU and network resources with better communication scheduling
- **SwitchML and ATP** → utilizes programmable network switches for in-network aggregation in datacenter environments, cuts down cross-rack data transfer
- Exploiting datacenter **network locality**
- Utilizing **quantization schemes** to reduce communication volume

Overview

- **Data parallelism** → training DNN layers
- **4D parallelism** → training memory-intensive embedding operators
- ***ZionEx*** → hardware platform to optimize inter-node communications



4D Parallelism: Table-Wise Parallelism



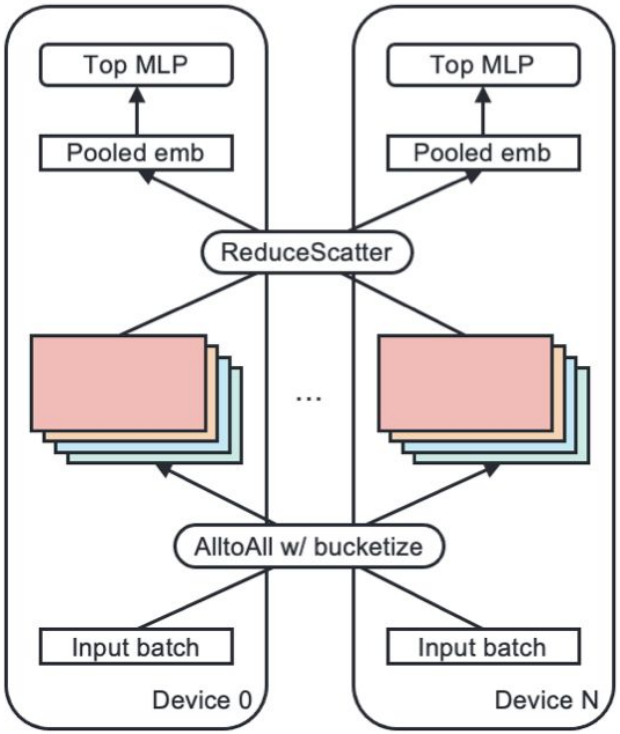
- Partitioning and parallelizing multiple embedding tables across GPUs

✅ Requires no handling of embedding table input indices or pooled embedding results

❌ Cannot handle large embedding tables that exceed the memory capacity of a single GPU

❌ Achieved load balance often limited due to skew in table sizes

4D Parallelism: Row-Wise Parallelism

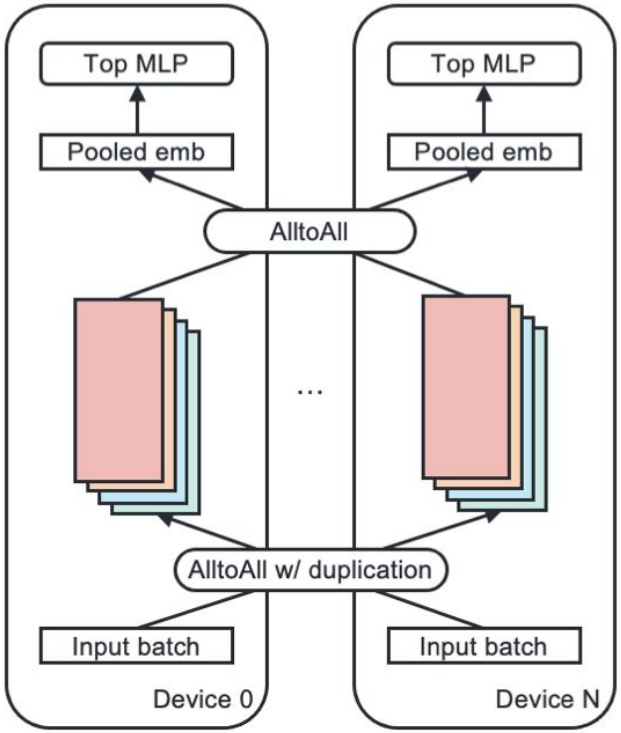


- Parallelizing by rows and assigning different table shards to different trainers
- Inputs *bucketized*
- Partial results need to be reduced then scattered

✅ Handles large tables well and leads to better load balance

❌ Communication cost scales linearly with the number of trainers

4D Parallelism: Column-Wise Parallelism



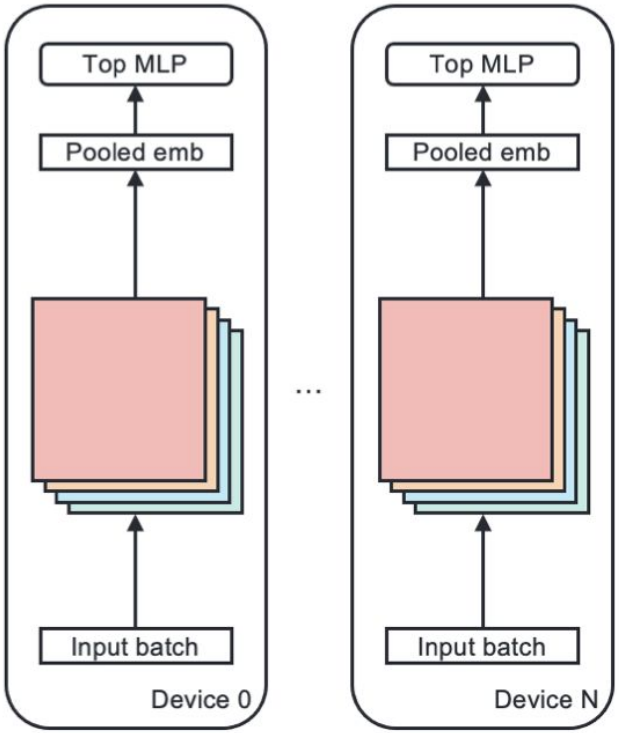
- Partitioning the embedding tables along the embedding dimensions
- Input indices duplicated

✅ Enables finer-grained parallelism

❌ Only works well with large embedding dimensions, increases payload for input indices

❌ Row-wise updates introduce additional parameters, one for each shard of the row

4D Parallelism: Data Parallelism



- Treating embedding tables as dense parameters and replicating them across all trainers
- Works best on small embedding tables with fewer rows

✅ No communication in the forward pass

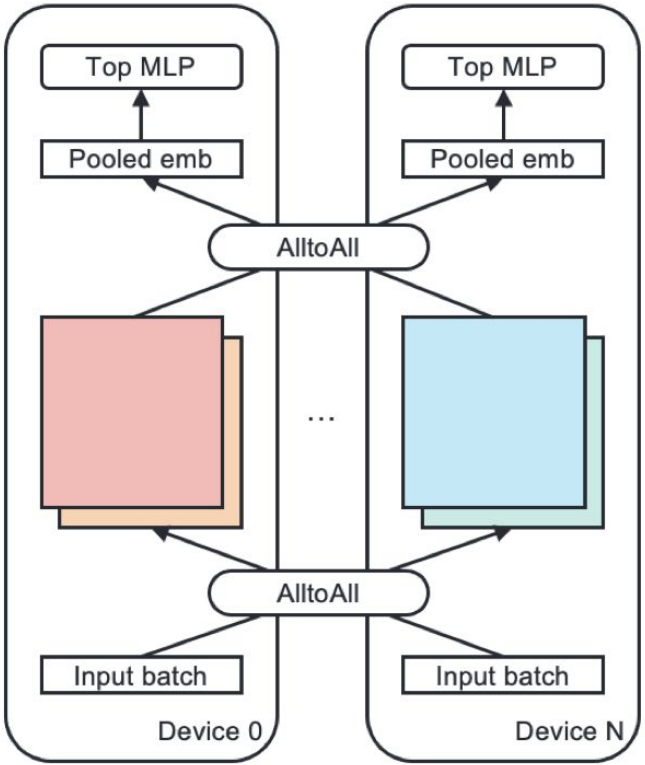
⚠️ Trade-off: *AlltoAll* of pooled embeddings v.s. *AllReduce* on entire table

Parallelization Algorithms

- Neo allows users to mix and match parallelization primitives
- Supports partitioning embedding operators in a **recursive** manner
    - E.g. “table-wise then row-wise” scheme assigns a set of tables to a particular node, then within that node, tables partitioned row-wise
- Can explore placement algorithms for each parallelization scheme by minimizing **“cost function”**
    - combination of communication overhead and load imbalance between the trainers

Pipelining

- When batch i is being evaluated, the same GPUs start receiving and distributing batch $i + 1$ using a separate stream
- Overlap input *AlltoAll* of batch i + 1 with forward propagation of top MLP of batch i
- Overlap pooled embedding *AlltoAll* with forward propagation of bottom MLP



Embedding Optimizations

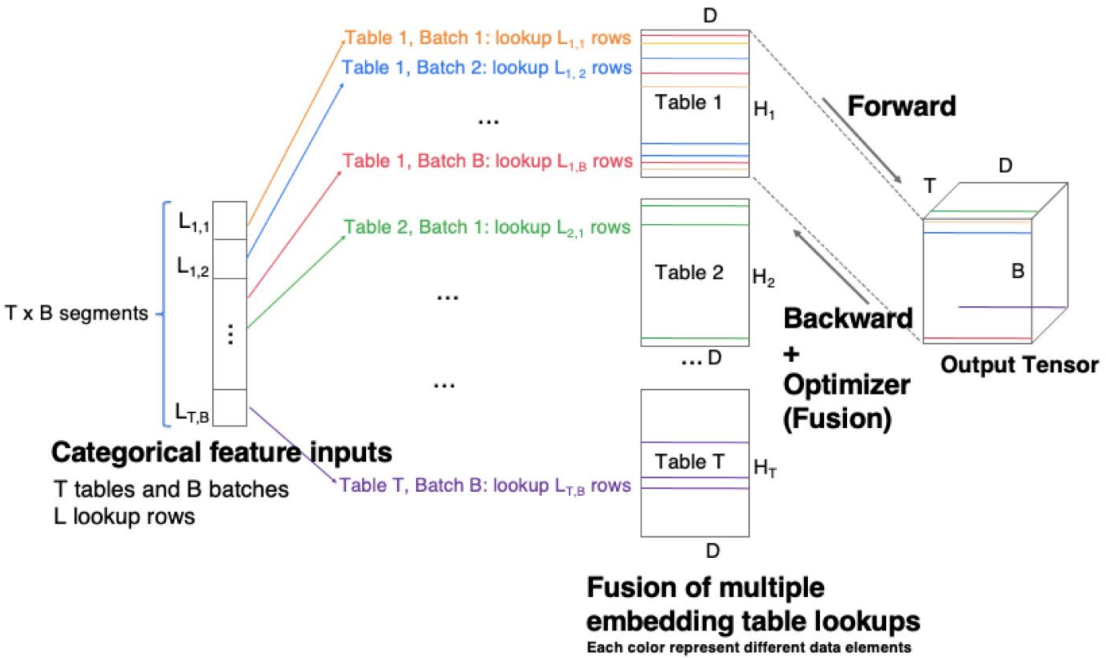
Two main challenges when optimizing runtime performance of DLRM's embedding operators:

1. Forward processing, backward propagation, and gradient updates for embedding operators require launching thousands of GPU kernels
2. Some embedding operators may include up to billions of parameters and do not fit on the device memory of a single GPU

Embedding Optimizations

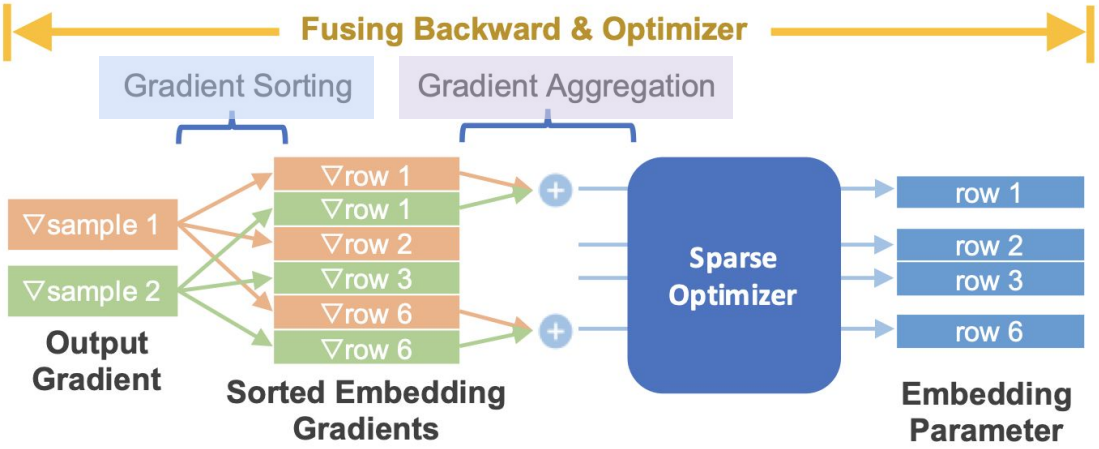
- Hybrid kernel fusion
- 4D parallelism
- Multi-level memory hierarchy
- *ZionEx*

Embedding Optimizations: Hybrid Kernel Fusion



- Fuses multiple embedding lookups on the same GPU into a single CUDA kernel
- Reduces overhead of launching multiple CUDA kernels (for each embedding lookup) on GPUs

Embedding Optimizations: Hybrid Kernel Fusion



- Fuses the backward pass with the sparse optimizer

⚠ Avoid race conditions!

Gradient sorting: updates to each row processed by a single CUDA thread block

Gradient aggregation: applied within each CUDA thread block using faster, smaller GPU shared memory

Embedding Optimizations: Hybrid Kernel Fusion

- Avoids allocating GPU device memory for embedding gradients
- Intermediate embedding gradients stored in GPU shared memory (not device memory)
- Improves overall performance of embedding computations by up to 7× compared to native implementation

Embedding Optimizations: Managing Memory Hierarchy

- DLRMs with up to trillions of parameters → embedding tables are too large to entirely fit on a single GPU

HBM (High Bandwidth Memory)	for very **fast, frequently accessed** data.
DRAM (Dynamic Random-Access Memory)	for **moderately fast, less frequently accessed** data.
SSDs (Solid-State Drives)	for **large, infrequently accessed** data storage.

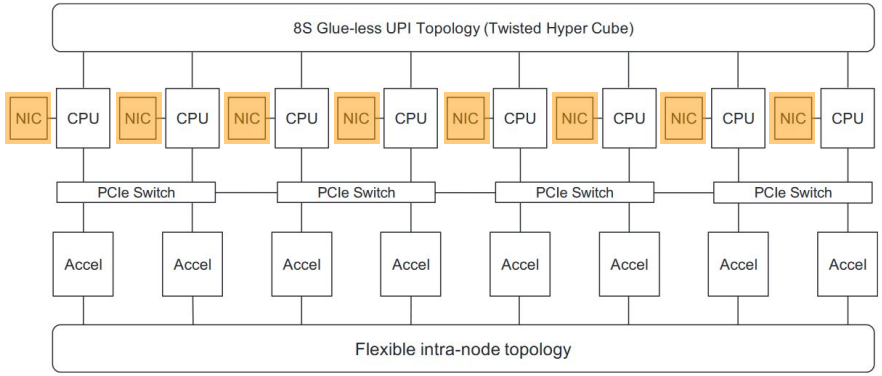
Embedding Optimizations: Managing Memory Hierarchy

- Faster memory serves as a **software cache** of the subsequent layer
- System also **scales out to multiple nodes** to increase aggregate capacity
- Useful for **online training** of DLRMs
    - Lower throughput requirements

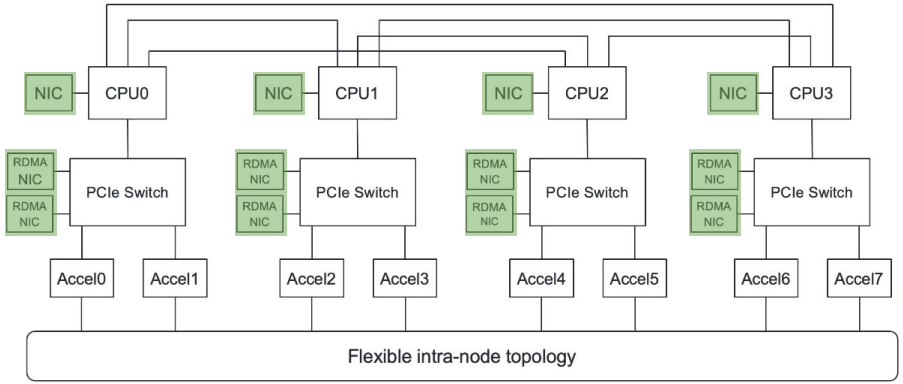
Embedding Optimizations: Managing Memory Hierarchy

- Customized 32-way set-associative **software cache** using least recently used (LRU) or least frequently used (LFU) cache replacement policies
- **Compression techniques**
  - Row-wise sparse optimizer
  - Low/mixed-precision training → high-precision cache backed by low precision embedding tables
  - Advanced factorization techniques

Zion to ZionEX: Evolution of Hardware Platform (1)



(a) *Zion*

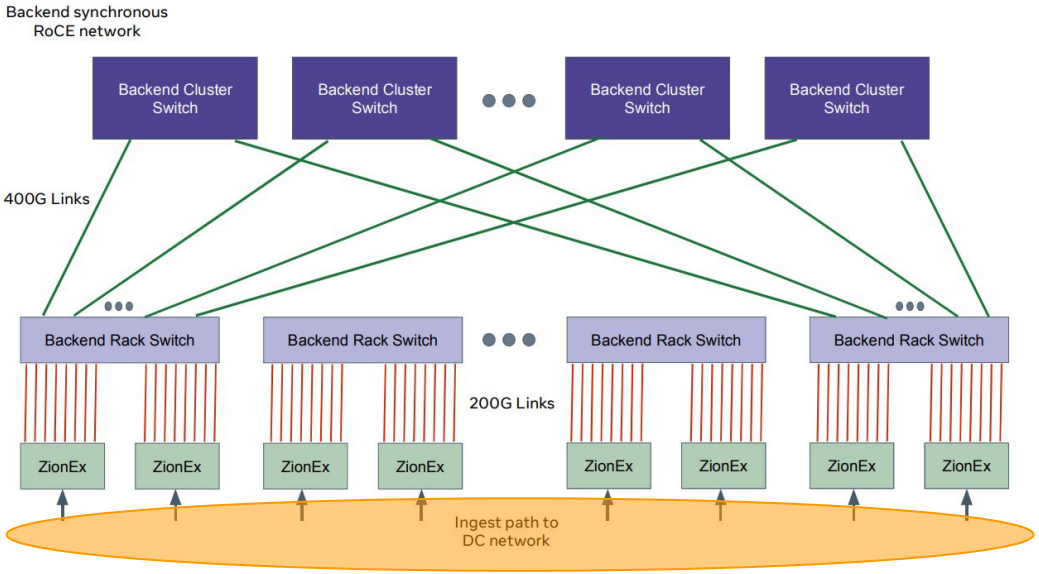


(b) *ZionEX*

Key improvements: Inter-node network

- TCP (100G regular NIC)
  -> RDMA over RoCE (200G RDMA NIC)
  - Significantly less network stack overhead.
  - Offload network stack from CPU to RDMA NIC.
  - Use dedicated inter-node connectivity.

ZionEX: A Separate Backend Network

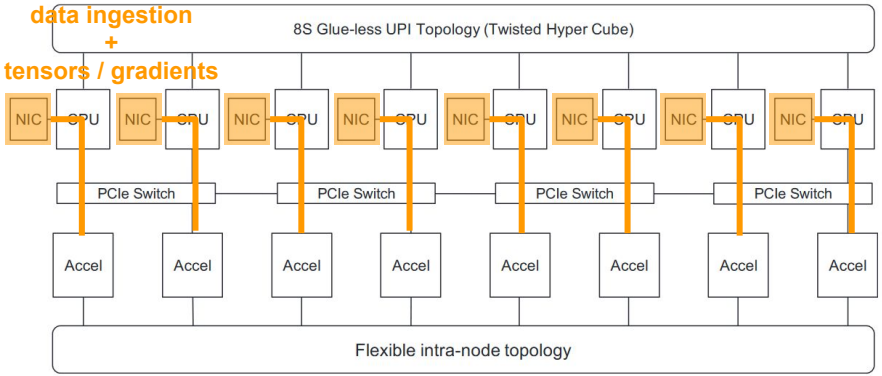


Ingestion to Regular
NICs attached to CPUs

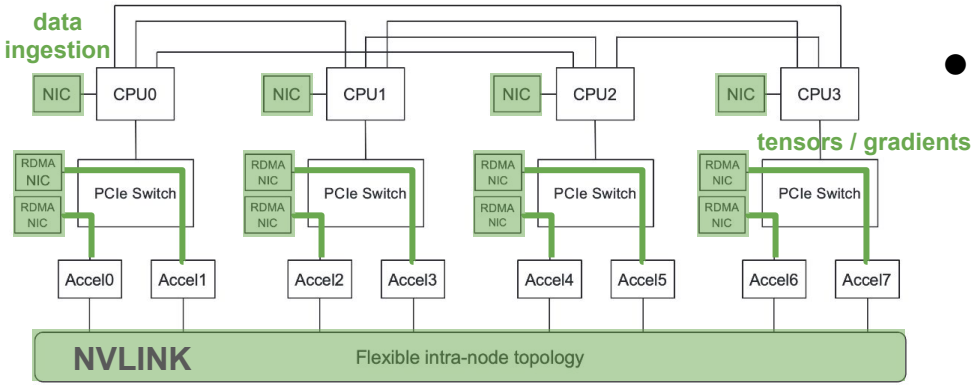
- Utilize multi-node low latency transport with direct data placement to serve AlltoAll and AllReduce collectives

✅ Scalability

Zion to ZionEX: Evolution of Hardware Platform (2)



(a) *Zion*



(b) *ZionEX*

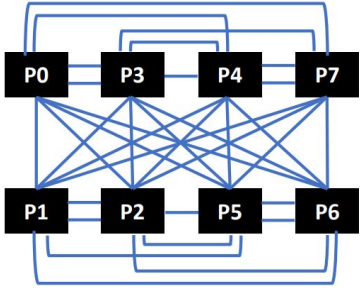
Key improvements: Inter-/intra-node network

- TCP (100G regular NIC)
  -> RDMA over RoCE (200G RDMA NIC)
  - Significantly less network stack overhead.
  - Offload network stack from CPU to RDMA NIC.
  - Use dedicated inter-node connectivity.
- Frequent CPU-GPU communication -> GPUDirect / RDMA
  - Heterogeneous on CPUs/GPUs -> entirely on GPUs.
  - Bypass CPUs: require less CPUs (8->4)
  - Better scalability.

OAM and Accelerator Interconnect Topology

- OAM: OCP (open compute project) Accelerator Module
  - Define vendor-agnostic common form factor

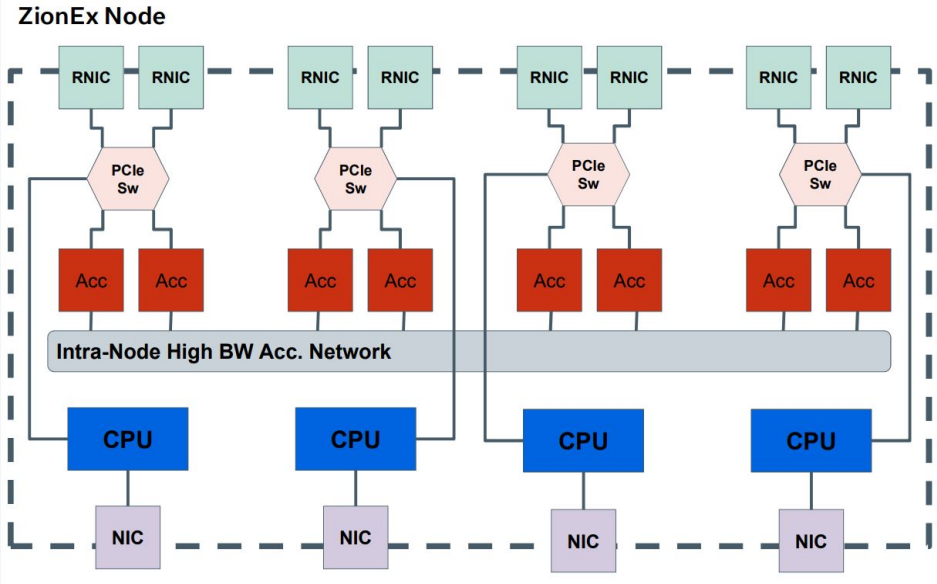
- Define superset physical topology
- RDMA and NVLINK



✅ Compatibility

✅ High performance

Zion to ZionEX: Evolution of Hardware Platform (3)



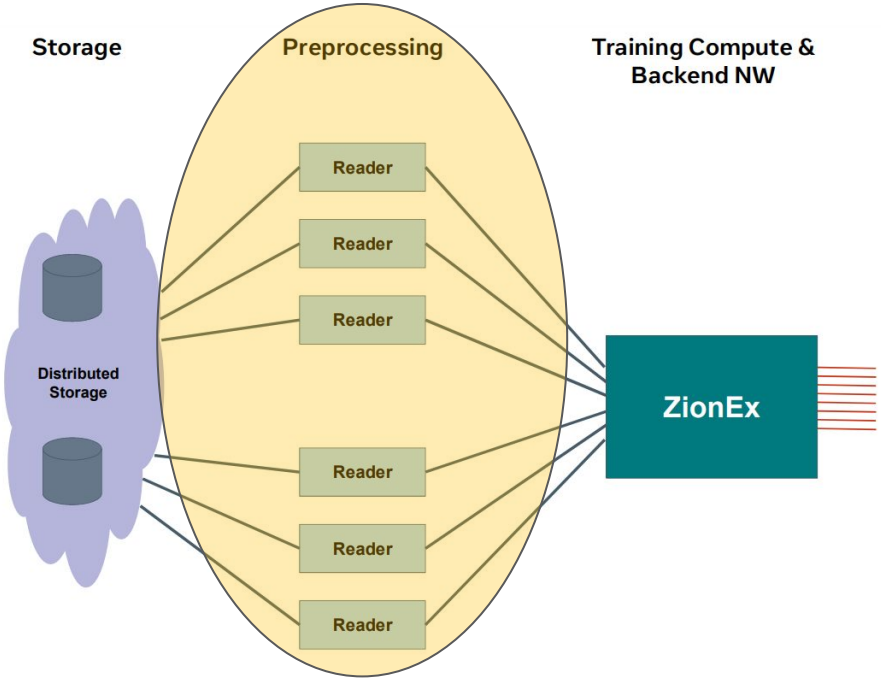
	Zion	ZionEX
Inter-node	TCP	RDMA
CPU-GPU comm.	Heavy	Light
Complexity	High due to heterogeneity	Low
Scalability	Low	High
Throughput	**63 TFLOPS/s**	**766 TFLOPS/s**

✅ More than 10x improvement, latency reduction can also be expected.

Implementation: Data Ingestion

❌ Significant latency overhead if unoptimized.
❌ CPU<->GPU transfer due to input tensors.

- Co-designed the data pre-processing module to use a combined format: lengths rather than offsets are used.
- Inputs to different embedding tables are simply concatenated.

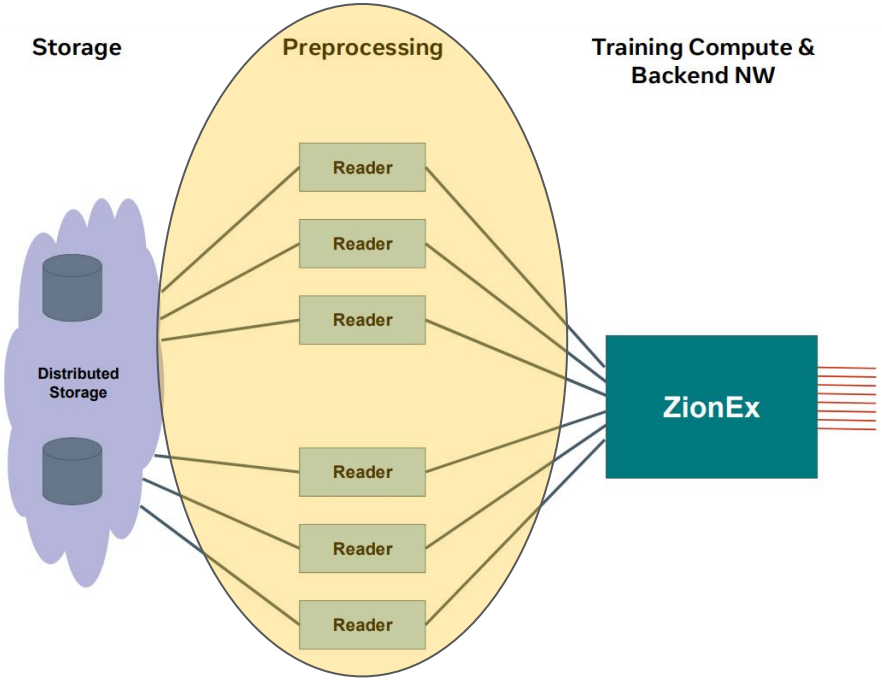


Implementation: Data Ingestion

❌ Significant latency overhead if unoptimized.
❌ CPU<->GPU transfer due to input tensors.

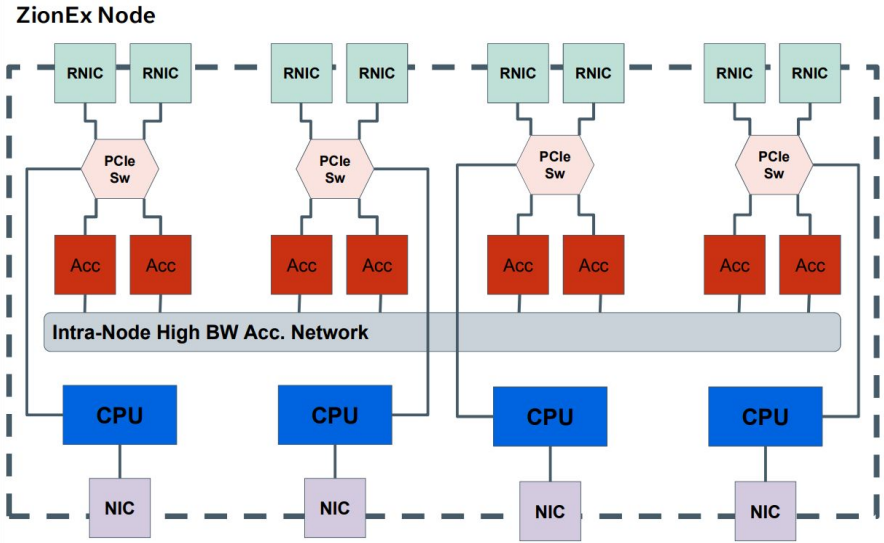
✅ No additional layout transformations.
✅ Consolidate small transfers.

- Co-designed the data pre-processing module to use a combined format: lengths rather than offsets are used.
- Inputs to different embedding tables are simply concatenated.



Implementation: Communication primitives

- Nvidia’s Collective Communication Library (NCCL)
  - Extend PyTorch NCCL process group to support Alltoall/Alltoallv collective with NCCL Send/Recv primitives



Evaluation Setup

- 16 ZionEX nodes
- On each node
  - 8 NVIDIA A100 GPUs with 320 GB HBM (12.4 TB/s)
  - 4-socket CPU with 1.5TB memory (320 GB/s)
  - Each GPU has a dedicated 200 Gbps RDMA NIC

End-to-End Training: Tested Models

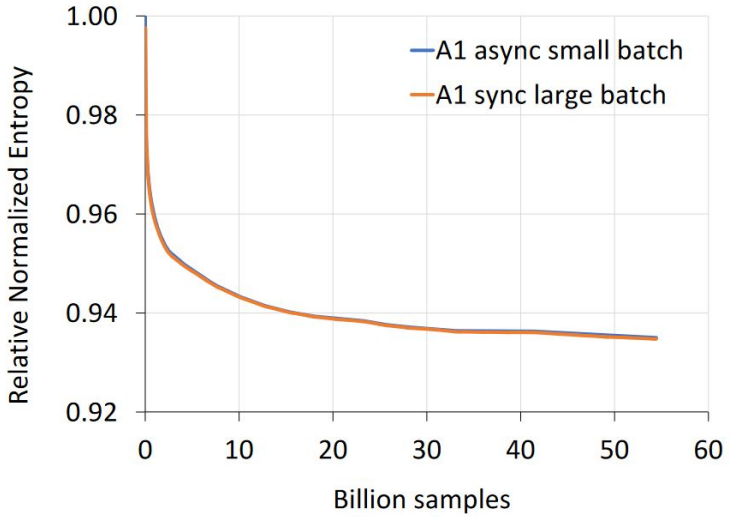
- Model-A: large and complex DLRMs -> computation and communication intensive.
- Model-F: small scale DLRM, has a single massive table that cannot fit in single GPU's memory.
- Model-I: moderate scale DLRMs -> memory intensive and high embedding pooling sizes.

Table 3: Target models configuration

Model	model-A	model-F	model-I
Num parameters	793B	12T	332B
MFLOPS per sample	638	5	60
Num of emb tables	$\approx$ 1000*s*	$\approx$ 10*s*	$\approx$ 100*s*
Embedding table dims (range [min, max], avg)	[4, 384] avg: 93	[256, 256] avg: 256	[92, 92] avg: 92
Avg pooling size	15	20	70
Num MLP layers	20	7	43
Avg MLP size	3375	490	682
Target local batch size	512	512	2048
Achieved QPS	1.2M	1.7M	3.4M

Training Quality (Model-A)

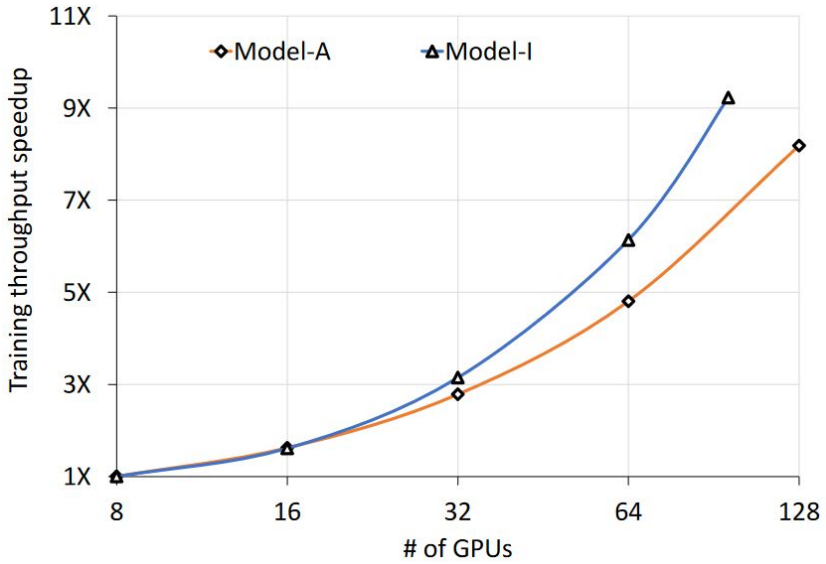
- Baseline: small batch (~150) asynchronous training on a distributed CPU platform (45 parameter servers and 15 trainers)
  ☹️ Unable to scale further
- Neo uses large (64K) batch size and synchronous training.



✅ Training on ZionEX provides on-par or better model quality
✅ Neo achieves 1.2MQPS -> 40x speedup
✅ ZionEX can further scale to more than 16 nodes.

Scaling Performance

- Using up to 16 ZionEX nodes, while keeping the per-GPU batch size constant.



Scaling efficiency:

- Model-A: 50%
- Model-I: 75%

What Factors Limit the Scaling Efficiency?

- Model-A has larger fully exposed All-to-All latency.
- More embedding tables -> larger All-to-All payload size -> consumes more bandwidth -> interferences
- Harder to balance the embedding computations and communications at the same time

What Factors Help the Scaling Efficiency?

- CPU-GPU (HtoD) transfer is completely hidden.
- The exposed communication latency is much lower than the sum of All2ALL and AllReduce latency.

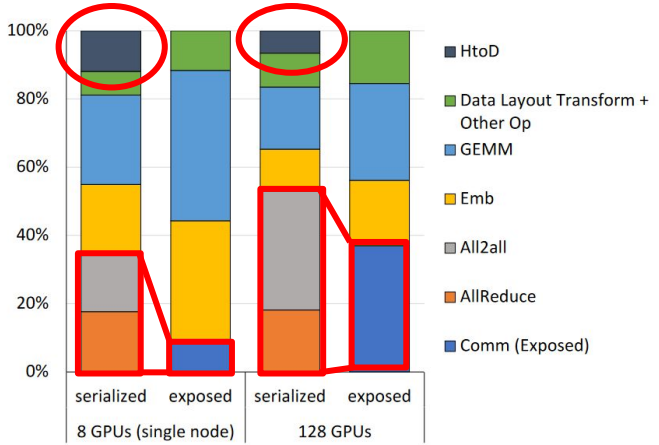


Figure 10: Model-A (with local batch size per GPU = 512) results and dominant operator time breakdown (serialized and exposed time) per GPU, after optimizations.

Training Throughput Optimizations

- Throughput within 15% theoretical estimates with roofline modeling.

Better load balance due to embedding table partition
Reduce model size and improve load balancing
Reduce communication volume and address the increased AlltoAll latency
Saturate GPUs and communication bandwidth better

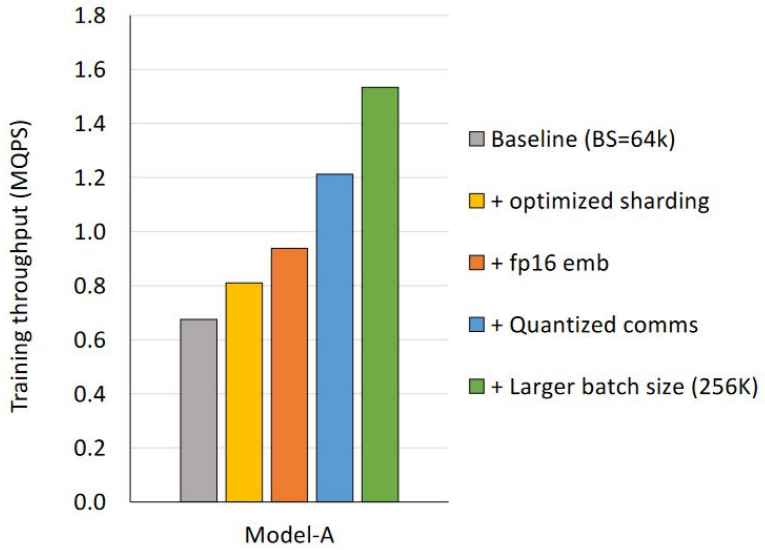


Figure 11: Training throughput improvements enabled by optimized sharding, reduced precision embeddings, quantized communications and larger batch sizes.

Model Capacity Limitation Study

- Model-F:
  - 12T parameters requires 96TB memory > total memory on 16 nodes.
  - A few massive embedding tables require multi-node memory.

- Use FP16 precision on embedding tables => 96TB → 24TB
- Apply row-wise sparse AdaGrad optimizer to embedding tables => distribute tables to multiple node.

Strength and Weakness

- Strength
  - Achieve significant performance improvement (upto 40x speedup).
  - Neo and ZionEX together improve the scalability of DLRMs training.
  - Good flexibility and compatibility.
  - A practical solution at data-center scale.

- Weakness
  - Communication primitives are not super clearly stated in the paper (e.g. one-sided/two-sided RDMA).
  - Scalability to thousands of ZionEX nodes is still doutable.
  - Evaluations only on three representative models => would be better to evaluate more models.

Conclusion

- SW/HW co-design solution that enables training models with trillions of parameters.
- Impressive performance improvement: 40× faster total training time for production recommendation models.
- **Neo** with novel software optimizations: 4D parallelism, embedding kernels, hybrid kernel fusion, hierarchical memory management.
- **ZionEX** enhances the scalability of training cluster and provides 10x performance improvements.
- **Successfully deployed in real production environment.**

Discussions

- As model size keeps growing, communication can further dominate the training time. How to further optimize the communication?
  - RDMA/NVLINK
  - Direct data placement
  - Customized data format
  - Quantization and compression (loss-less), sending only deltas
  - Multi-path data routing, programmable network switches
- Power budget
  - Customized hardware can definitely help reduce power consumption
  - Flexibility to accommodate workload changes is also critical
- Use SmartNIC to coordinate intra-node load balancing