Recommender Systems 101: Accelerated Training at Scale

Minseok Lee | 03-20-2023

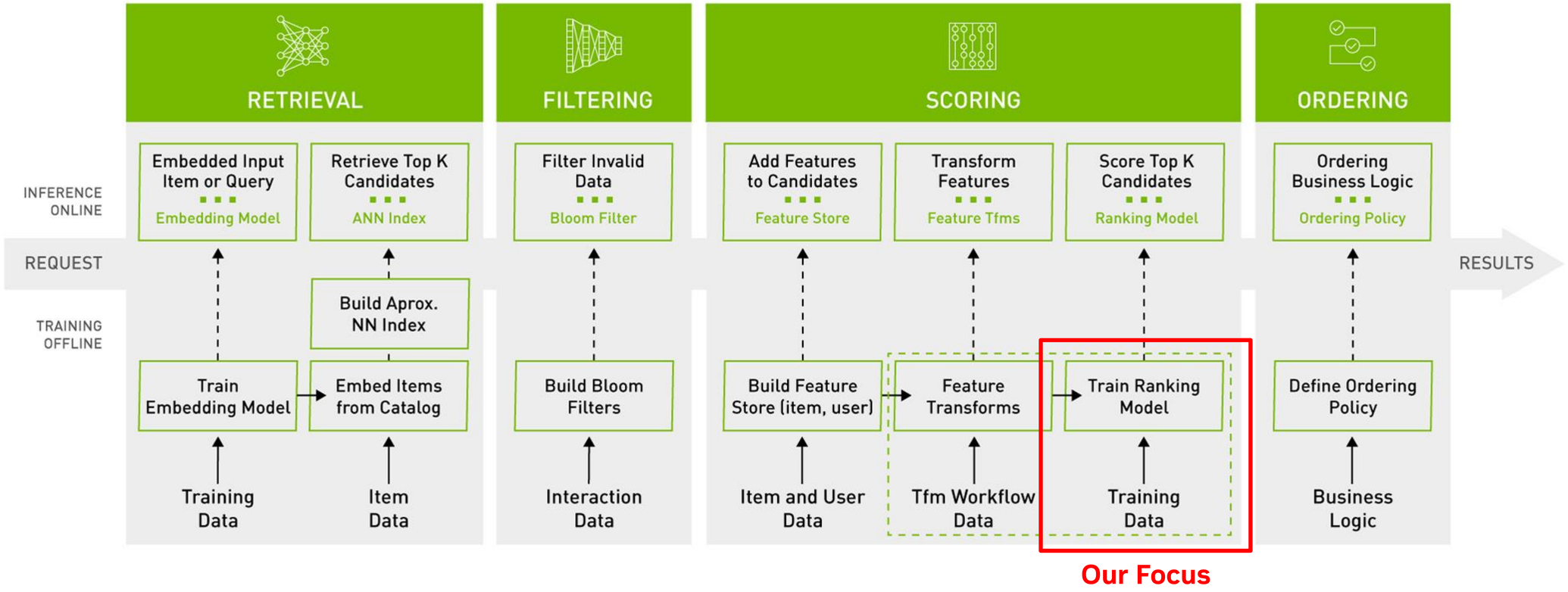
Recommender Systems

Personalization Engines of the Internet

"Already, 35 percent of what consumers purchase on Amazon and 75 percent of what they watch on Netflix come from product recommendations based on such algorithms." - McKinsey

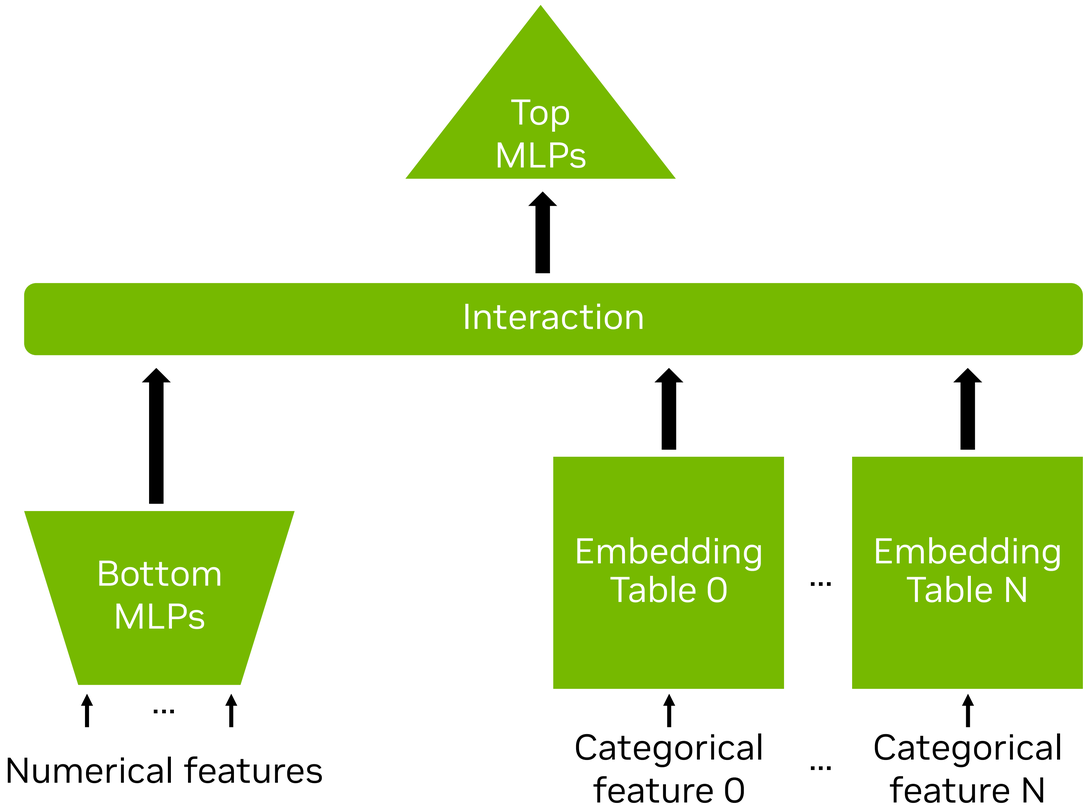
Multi-stage Recommender System Pipeline

and Scope of This Presentation



Deep Learning Recommendation Models (DLRM)

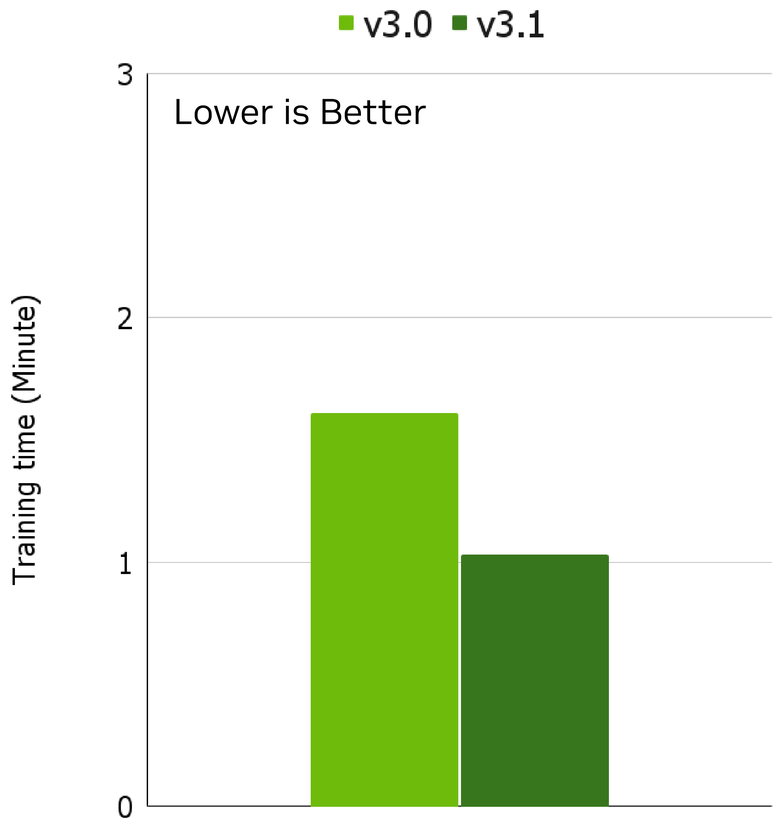
Simplified View



- Bottom MLPs
  - Process dense input features, often independent to the embedding lookups
- Embedding tables
  - Map one-hot/multi-hot categorical features into the embedding vectors
- Interaction
  - Learn effective feature crosses
- Top MLPs
  - Output the final recommendation result, e.g., click probability

MLPerf DLRM-DCNv2 Training Performance

Train 4 Billion Samples in 1 Minute



- DLRM-DCNv2: a new MLPerf recommendation benchmark since the v3.0 submission with the following updates:
  - Criteo 4TB multi-hot dataset
  - 3 Cross layers
  - Adagrad optimizer
  - 26B parameters

- How could we achieve such training performance?
  - We will answer the question throughout this talk
  - But it is not limited to a specific benchmark

Goals of This Presentation

- Understand common challenges in training large-scale recommender systems
- Explore solutions and best practices to tackle them in a framework-agnostic manner
- Share the achieved multi-node training performance of our reference design on NVIDIA DGX H100s

GPU for Recommender System Training

- Embedding operations are memory bound
  - GPU has very high memory bandwidth, e.g,. H100 up to 3.35TB/s
  - GPU has highly parallel memory subsystem, good for scattered accesses
  - Caches do deduplication because categorical data follows power-law distribution

- Interaction and MLPs are highly dependent on linear algebra operations
  - GPU has proven its excellence in accelerating such operations, esp., GEMMs with Tensor Cores

Challenges of Large-Scale Recommender System Training

- Growing memory capacity demands
  - As content, users, and their interactions multiply, datasets and embedding tables expand
  - Embedding tables don't often fit into a single GPU

- Memory- and communication-bound embedding table operations
  - Embedding table lookups and related operations are memory bound
  - Distributing tables across multiple GPUs requires extensive inter-GPU communication

- Balancing workload distribution
  - different embedding tables with varying characteristics, e.g., different distribution, hotness, table size, vector size, etc.

Challenges of Large-Scale Recommender System Training

and Solutions

- Growing memory capacity demands → Employ distributed training with hybrid parallelism
  - As content, users, and their interactions multiply, datasets and embedding tables expand
  - Embedding tables don't often fit into a single GPU

- Memory- and communication-bound embedding table operations
  - Embedding table lookups and related operations are memory bound → Fuse their data structures and operations
  - Distributing tables across multiple GPUs requires extensive inter-GPU communication
    → Leverage a fast, scalable network fabric such as NVSwitch, compress the communication traffic, and hide their overhead

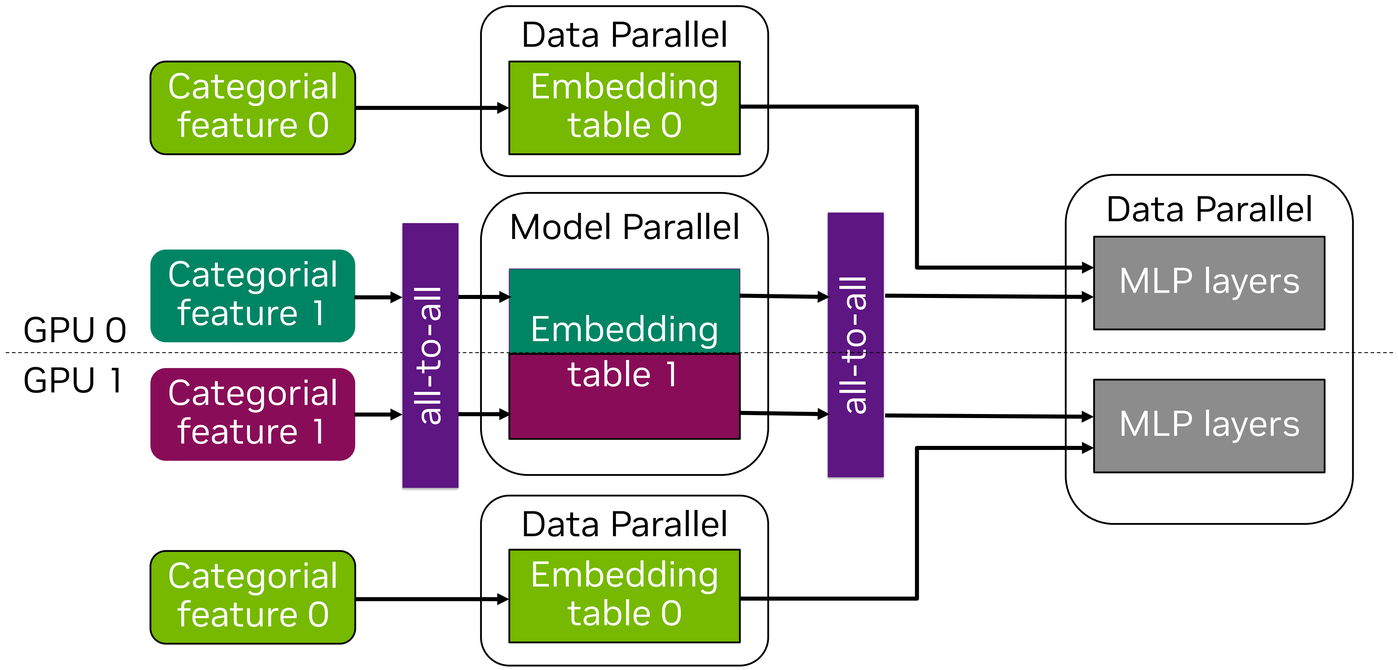
- Balancing workload distribution → Create per-table sharding strategies
  - different embedding tables with varying characteristics, e.g., different distribution, hotness, table size, vector size, etc.

Distributed Training with Hybrid Parallelism

Hybrid Parallelization Approach

- Large-scale recommender system training requires both model and data parallelisms:
  - Data parallelism for small embedding tables, MLP and interaction layers
  - Model parallelism for large embedding tables – tables are **sharded** on specific GPUs

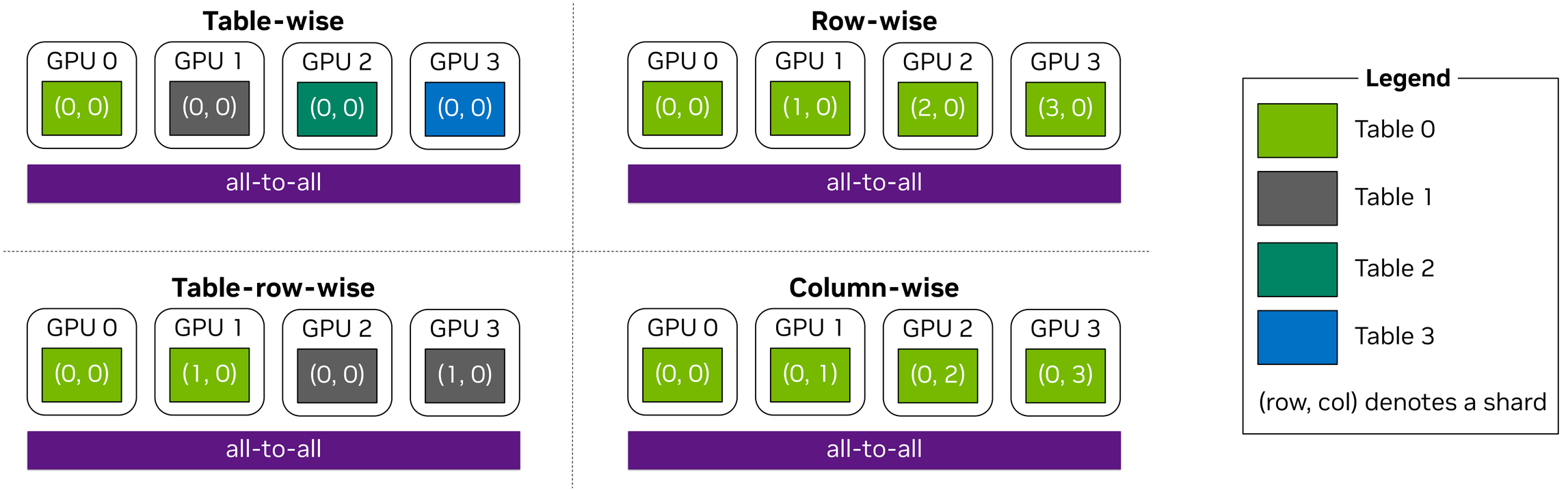
- Their judicious orchestration overcomes GPU memory capacity limit and maximizes the training throughput



Taxonomy of Model Parallelism

Embedding Table Sharding Schemes

- Embedding tables can be sharded at a variety of granularities:



* First introduced in: Mudigere et al., "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models", in ISCA'22.

Embedding Table Sharding Schemes

Pros and Cons

Scheme	Advantages	Disadvantages
Data-parallel	• No all-to-all communication during the forward pass • Can leverage NVLink Sharp (NVLS) to offload AllReduce for t he gradient calculation	• Larger tables may not fit in GPU memory • A table-sized AllReduce for the gradient calculation
Table-wise	• Simple and straightforward • Best communication efficiency with compressions	• Cannot handle embeddings larger than single GPU's memory • Very inefficient for cases where # GPUs > # tables • Bad at handling load imbalance among tables
Table-row-wise	• Help reduce lookup time for features with very high hotness	• Cannot handle embeddings too large to fit in a single node
Row-wise	• Not limited by the table size	• All-to-all message size per GPU is proportional to # GPUs • Have the intra-table load imbalance issue
Column-wise	• No need to compromise between load balance and compression ratio • Useful in increasing parallelism when # hotness > # GPUs	• The inputs to tables must be replicated • Only attractive for large embedding vector sizes

- A single scheme cannot satisfy all tables, models and systems of various characteristics

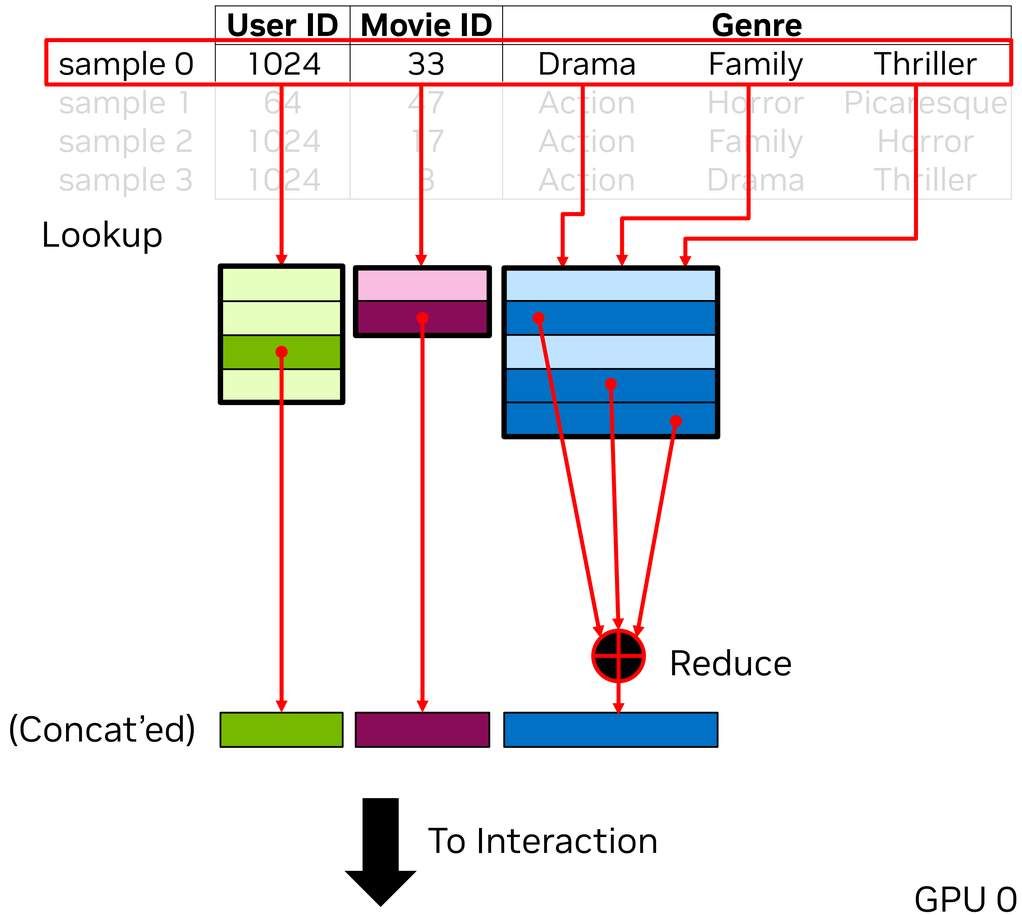
Sharding Strategy

- The combination of sharding schemes for all embedding tables of a given model
- A good sharding strategy should improve the training throughput by (1) achieving load balance and (2) minimizing their communication traffic among GPUs
- It is a NP-hard problem to find the optimal sharding strategy considering all given constraints including the properties of dataset, model, underlying hardware, etc.
- However, a heuristic or greedy algorithm still works decently for many cases

Optimizing Embedding Table Operations

Embedding Table Lookup

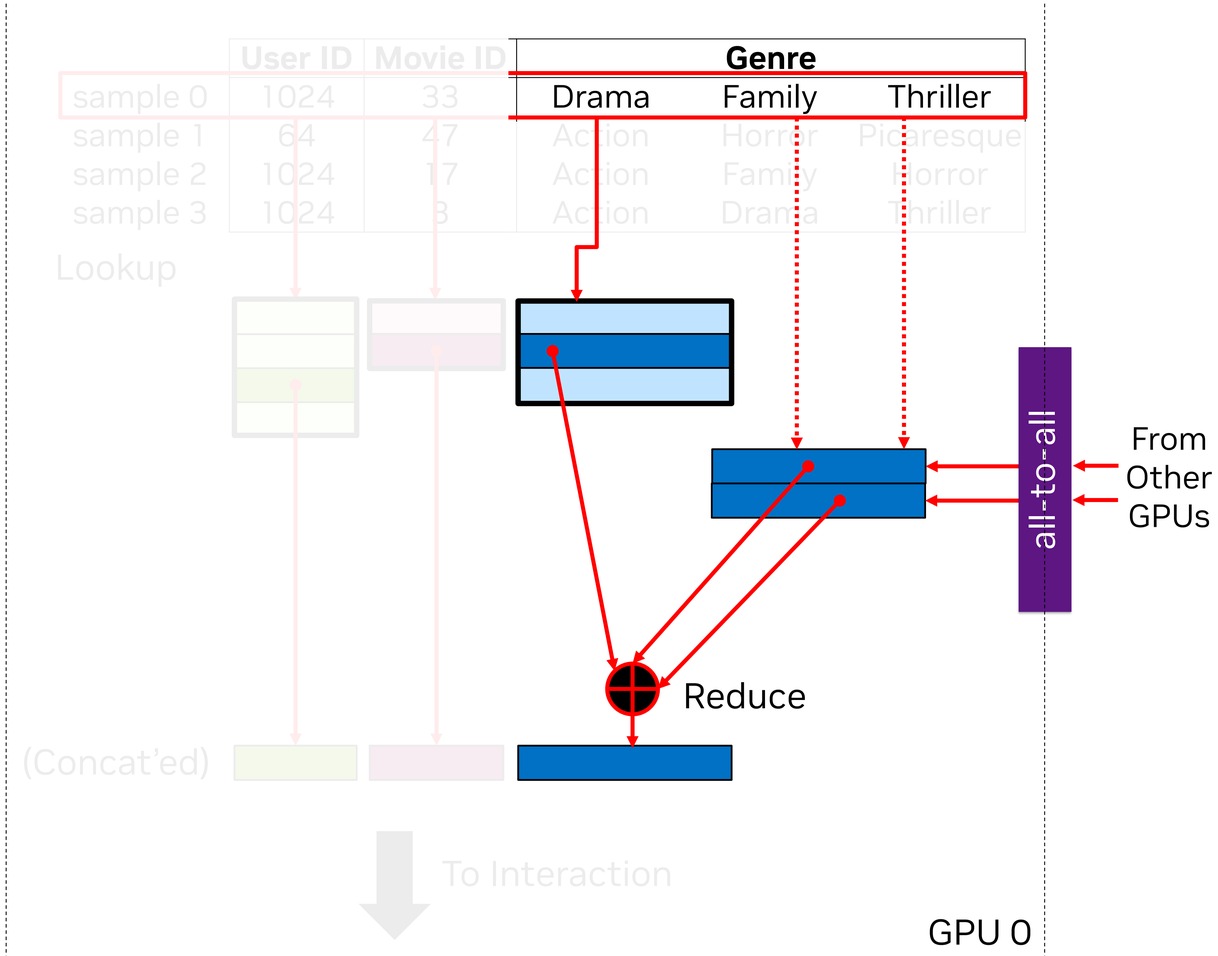
Single-GPU Case



- Each feature is mapped to its own embedding table
- Each table can have different # rows and # cols
- Each feature can have different hotness:
  - 1-hot: a single lookup per sample (User ID & Move ID)
  - M-hot: M lookups per sample (Location & Genre)
- Embedding vectors from a M-hot lookup are likely to be pooled into a single vector
  - Can be implemented as an in-register operation

Embedding Table Lookup

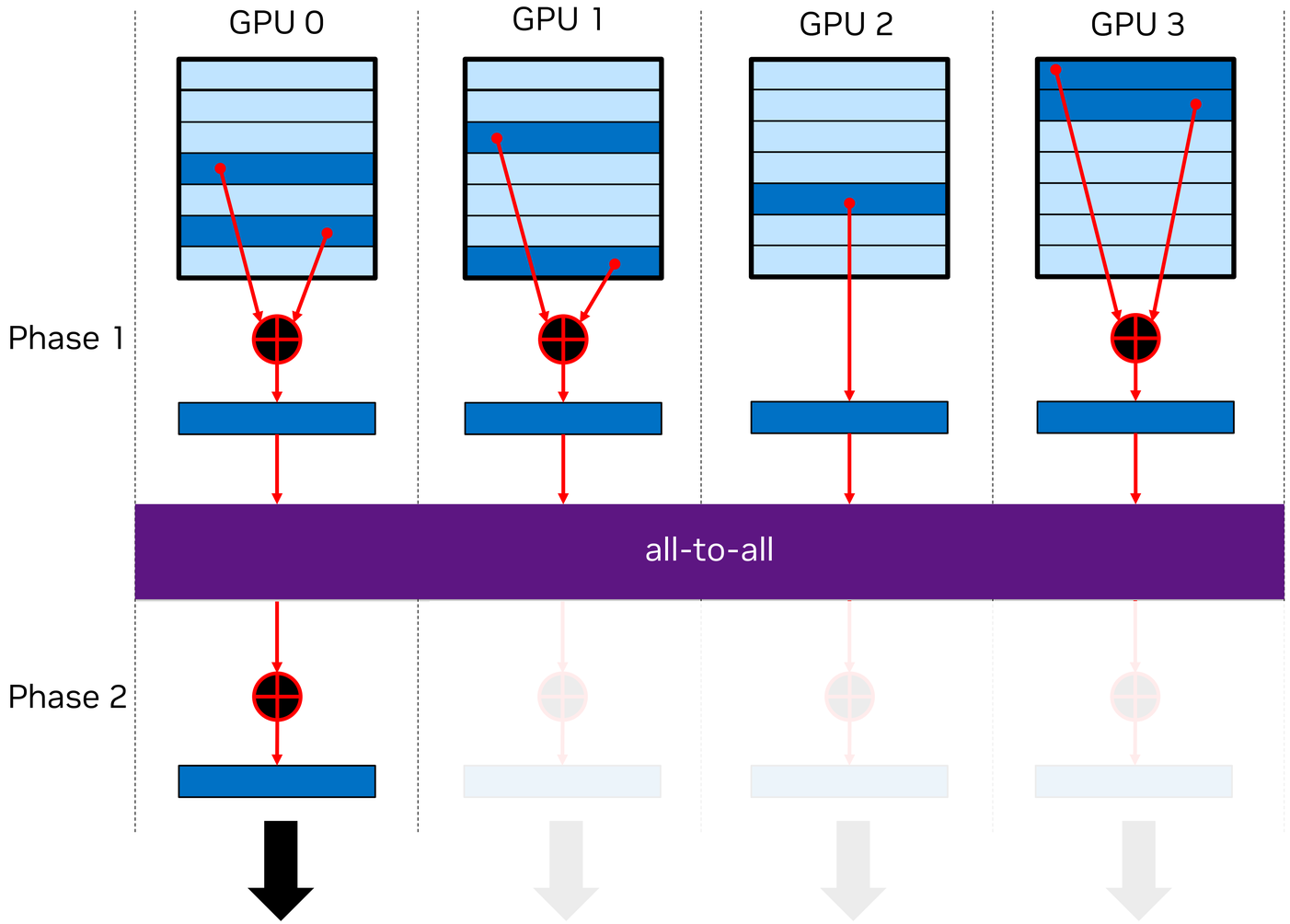
Multi-GPU Case



- Requested embedding vectors or tables may not reside in the requester GPU
- They must be received through the all-to-all communication (A2A) among GPUs
- The amount of A2A traffic is a function of embedding vector size, hotness, batch size, sharding and cluster size
- The A2A must be mitigated to better accelerate large-scale recommender training

A2A Traffic Reduction Technique 1

Hierarchical Reduce for Multi-Hot Features



- Partially reduce the vectors before A2A, and the n do the final reduction in the destination GPU
- Advantages:
  - Partial reduction can be done at the register-leve l effectively
  - Independent of batch size
- Disadvantages:
  - Compression ratio up to hotness of the feature
  - Compression ratio depends on the employed sharding scheme

A2A Traffic Reduction Technique 2

Unique Operation

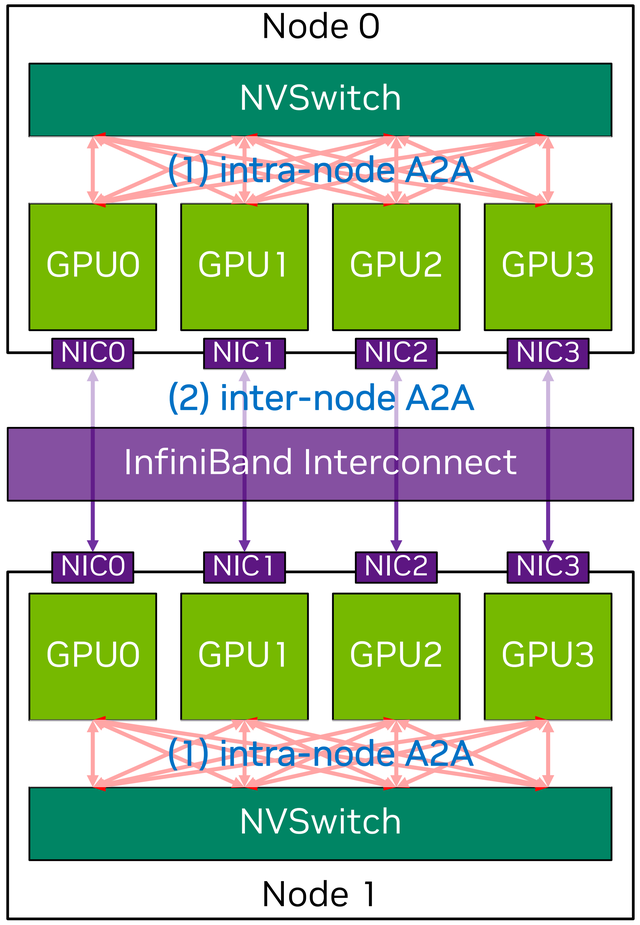
	User ID	Movie ID	Genre					
sample 0	1024	33	0	Drama	Family	0	0	Thriller
sample 1	64	47	Action	0	0	Horror	Picaresque	0
sample 2	1024	17	Action	0	Family	Horror	0	0
sample 3	1024	3	Action	Drama	0	0	0	Thriller

per-column unique across samples

User ID	Movie ID	Genre					
1024	33	Action	Drama	Family	Horror	Picaresque	Thriller
64	47						
	17						
	3						

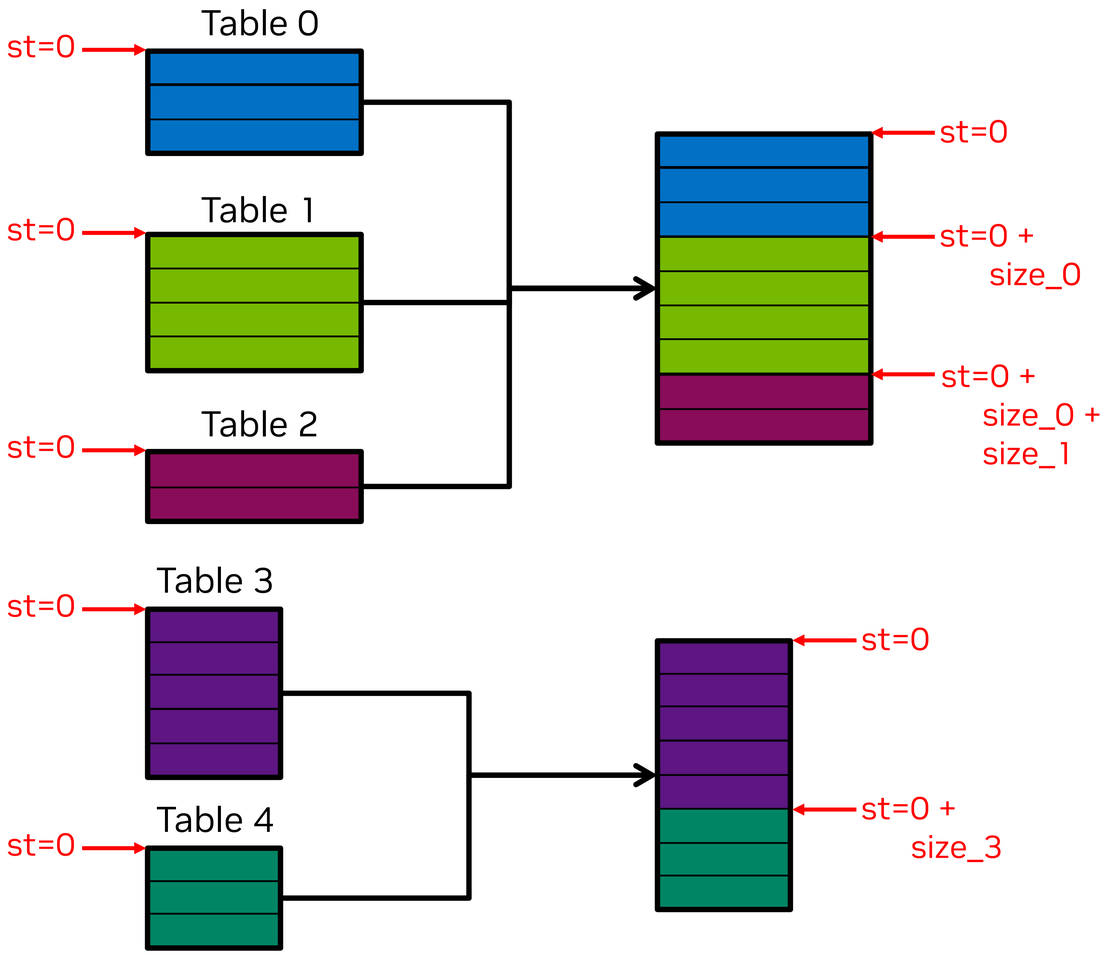
- Apply a unique op to each feature in a batch before any inter-GPU communication
- Advantages:
  - Independent of hotness
  - Less sensitive to the sharding scheme
- Disadvantages:
  - The compression ratio is highly dependent on input distribution and batch size

Hierarchical Communication



- To mitigate the cost of multi-node training and scale it out, leverage the full connectivity and high bandwidth of NVSwitch:
  1. Within the node, move the embedding vectors to a GPU that share the same InfiniBand rail as the destination GPU
  2. Send them as an aggregated message to the destination GPU without crossing rails
- Benefits:
  - Maximize effective message rate and minimize the transmission latency
  - Reduce traffic flow through the second-tier spine switches and optimizes network traffic
- The optimization is available in the name of PXN in NCCL

Fusing Embedding Tables



- Consider to fuse and co-optimize different embedding tables which share the characteristics, e.g., embedding vector size
- Such a fused embedding table can be implemented with the smaller number of operations
  - More efficient in terms of CUDA kernel launch overhead and GPU utilization
- If the original tables share the same vector size, it can be realized by applying offsets to their lookup indices
- Even if they have the same vector size, do not fuse them if their best sharding schemes are different, e.g., Data-Parallel vs. Row-Wise

MLP Layers

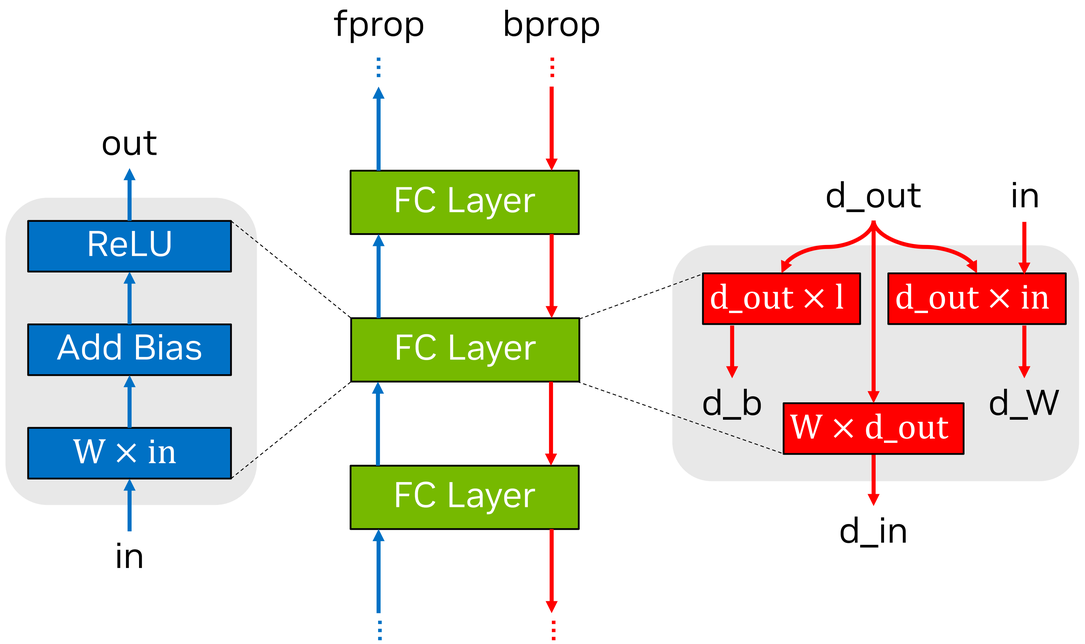
MLPs

Why Do We Care?

- MLPs can take a smaller portion of time than that those of CNN or NLP models
- But they still represent the significant portion of training time in a RecSys model
  - If we incorporate transformer layers beyond simple fully connected layers, it takes more time
- MLP optimization strategies
  - Lower precision
  - Kernel fusions
  - CUDA Graph
  - Pipelining

Kernel Fusion Example

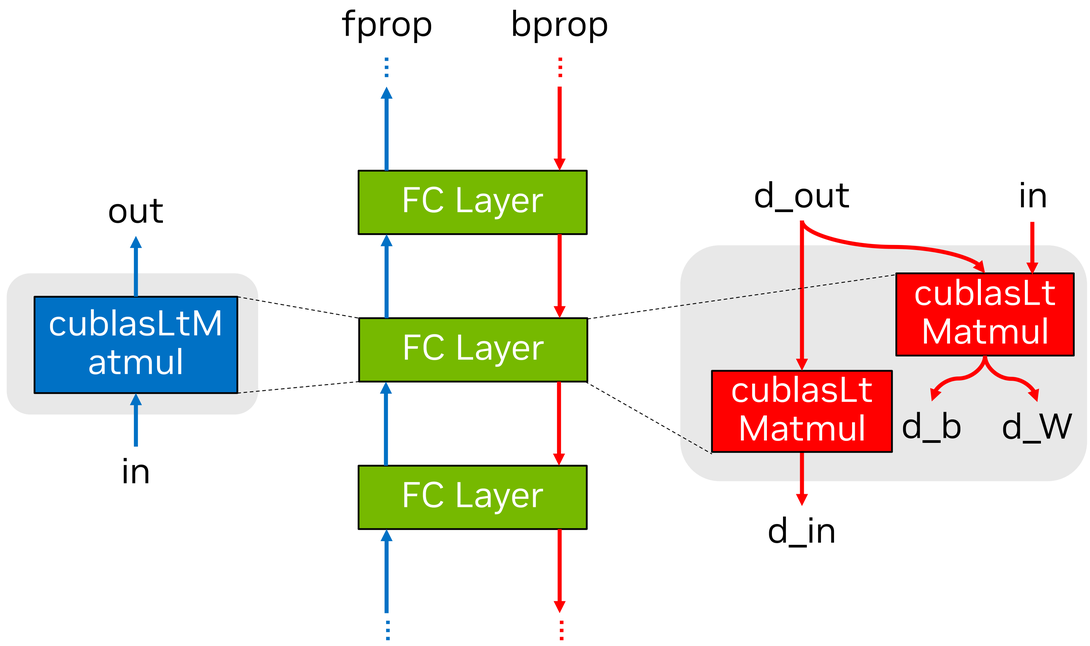
Fully Connected (FC) Layers



- Simple but common pattern across various RecSys models
- cuBLAS GEMMs are already highly optimized
  - Well-designed kernels
  - Efficient TensorCore utilization
- But there is room for improvements:
  - Redundant reads/writes between kernels
  - bprop kernels share the same input data (d_out)
  - Weight gradients (d_W and d_b) and data gradient (d_in) are computed in the same CUDA stream even if they are independent

Kernel Fusion Example

Fully Connected (FC) Layers

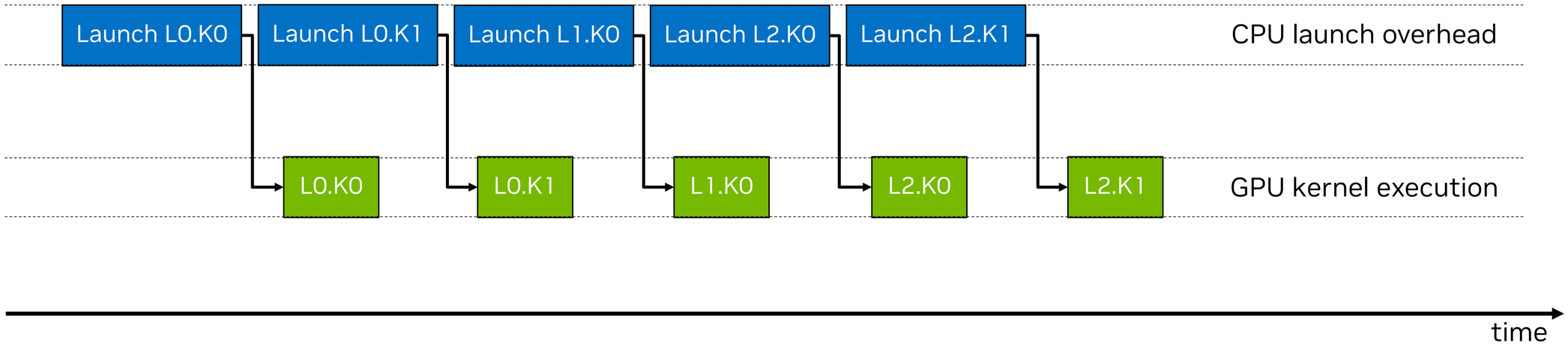


- cuBLASLt, a flexible lightweight GEMM library, offers several fusion options through cublasLtEpilogue_t:
  - CUBLASLT_EPILOGUE_DEFAULT
  - CUBLASLT_EPILOGUE_RELU_AUX_BIAS
  - CUBLASLT_EPILOGUE_DRELU_BGRAD
- d_b and d_W are computed in parallel with d_in
  - They are launched into different CUDA streams

CUDA Graph

Motivation

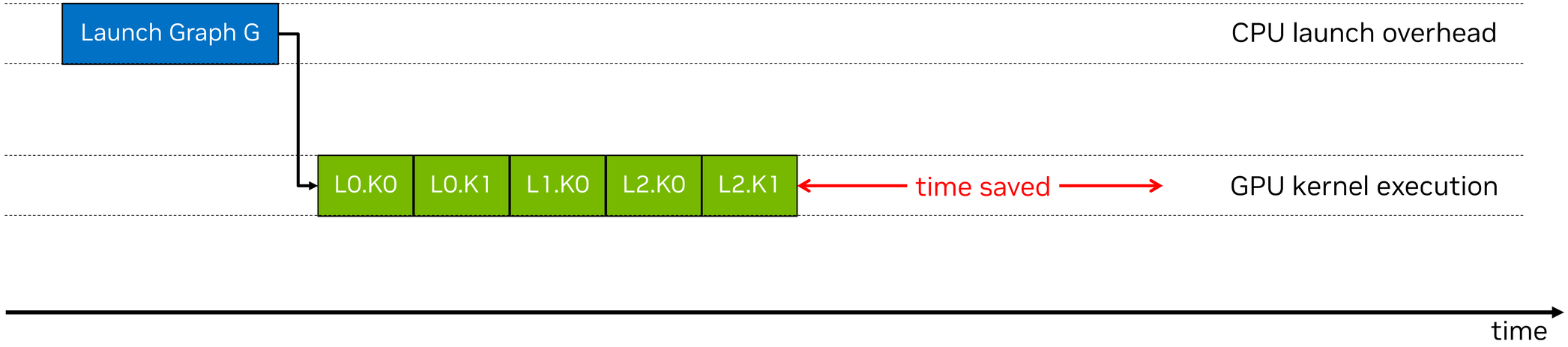
- CUDA kernel launch is not free
  - Overhead is aggregated with many small kernels



CUDA Graph

With CUDA Graph

- Encapsulate N kernel launches into a single CUDA graph launch
  - Construct an explicit CUDA graph or do a stream capture
  - CUDA library calls including NCCL are supported, e.g., AllReduce at the end of each backward pass



Parallelism and Concurrency

Trade-Offs of Embedding Vector Size and Batch Size

- Embedding vector size (or embedding width)
  - Too small
    - Cannot capture all information
    - Lead to computational inefficiency
  - Too large
    - may make your model overfit
    - increase the memory footprint
  - Choose the size as a multiple of 64B or 128B to achieve better memory utilization and parallel execution efficiency

- Batch size
  - Bigger batch size normally leads to more parallelism on GPUs, e.g., batched table lookups, batched MLPs, etc.
  - Cannot just increase it because it can affect the memory footprint and model accuracy

Pipelining

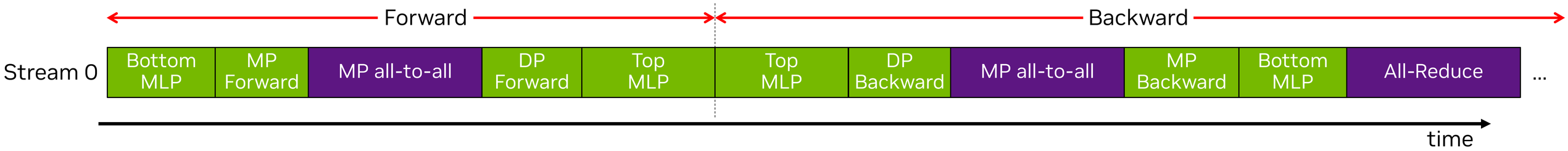
Motivation

- In large-scale, multi-node DLRM training, per-GPU batch size might not be full
  - Some operations may make GPU underutilized, e.g., SMs may not have enough thread blocks
- Multi-node DLRM training requires heavy all-to-all communications for model parallel embeddings and all-reduce communication for the data-parallel parts
  - During the communication, GPU SMs are not fully utilized
- Some operations are independent with one another
  - No dependency on input data between iterations
  - Embeddings don’t rely on Bottom MLPs
  - Data-parallel and model-parallel embedding table operations are independent
  - …

Pipelining

(A Simplified) Example: Before

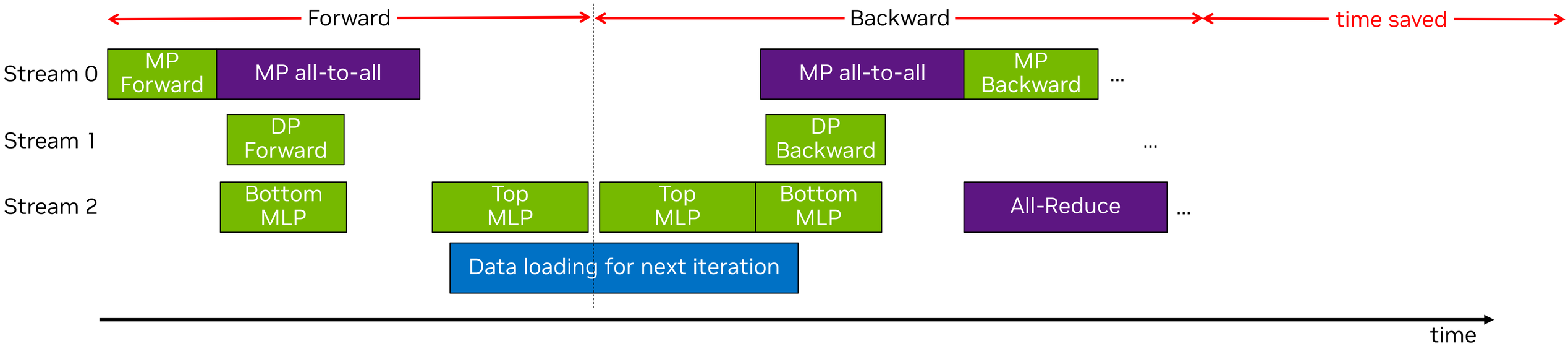
- Communications and computations are serialized in the same CUDA stream



Pipelining

(A Simplified) Example: After

- Overlap between computations and communications using different CUDA streams
- Prefetch next iteration data



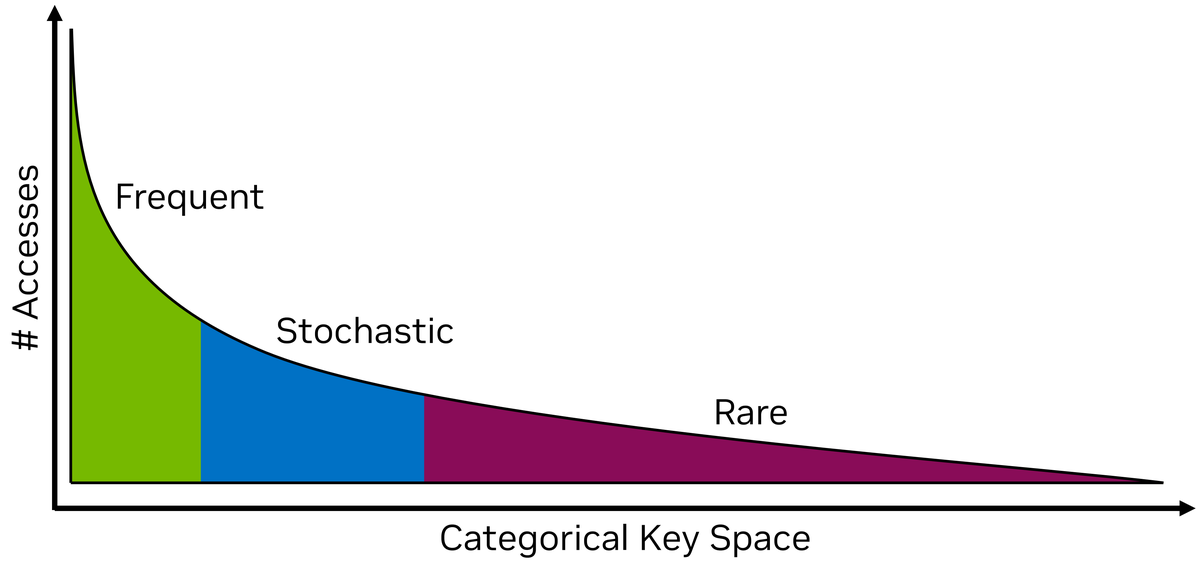
Dynamic Embedding Table

Dynamic Embedding Table

Motivation

- Embedding table size can grow rapidly and dynamically
  - and it is sometimes hard to predict its size in advance → Hard to preallocate

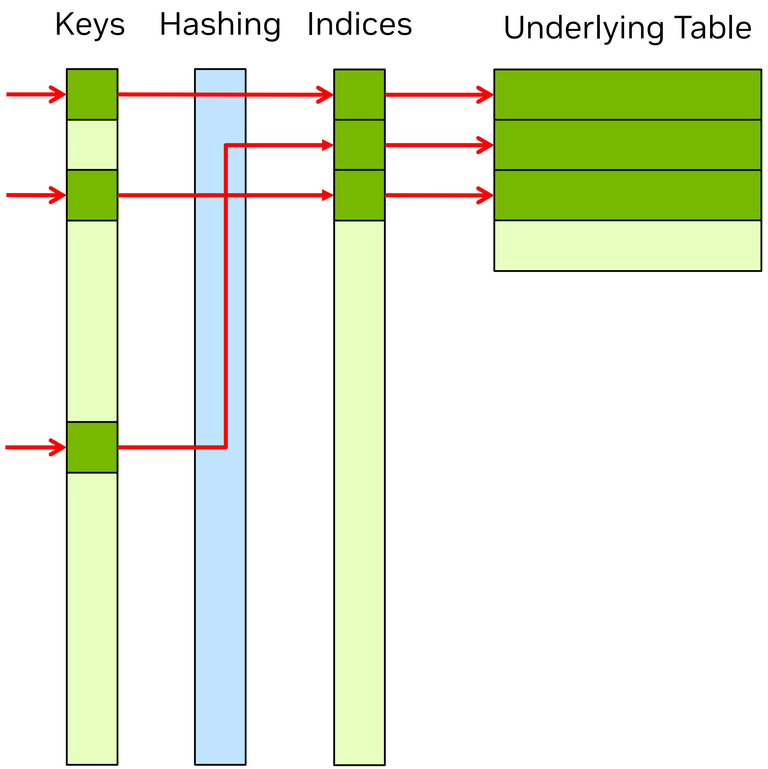
- Even if some categorical features can have large cardinalities, a large portion of the keys might be rarely used
  - It often follows a power-law-like distributions → No need to preallocate everything



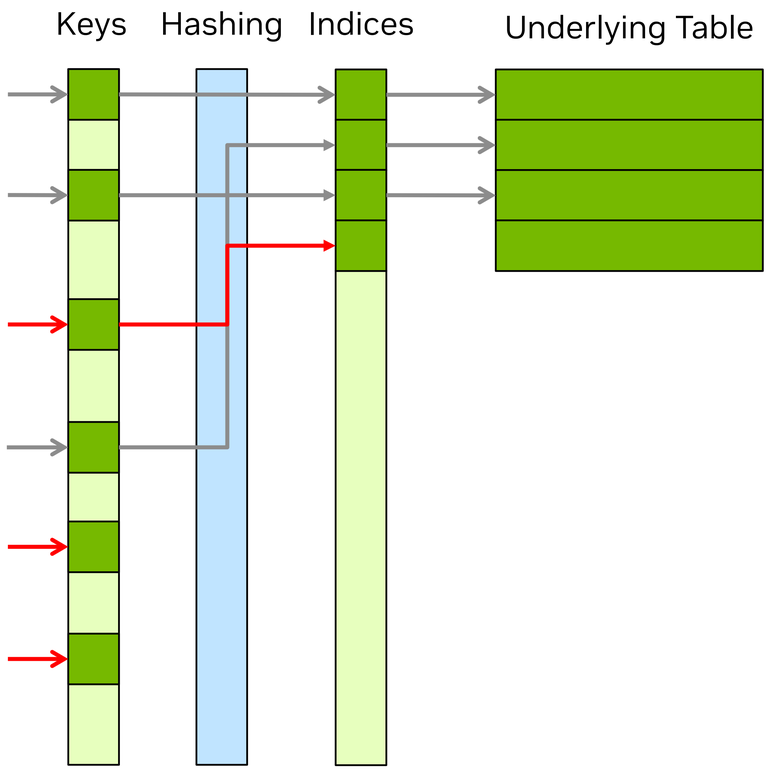
Dynamic Embedding Table

Grow on Demand

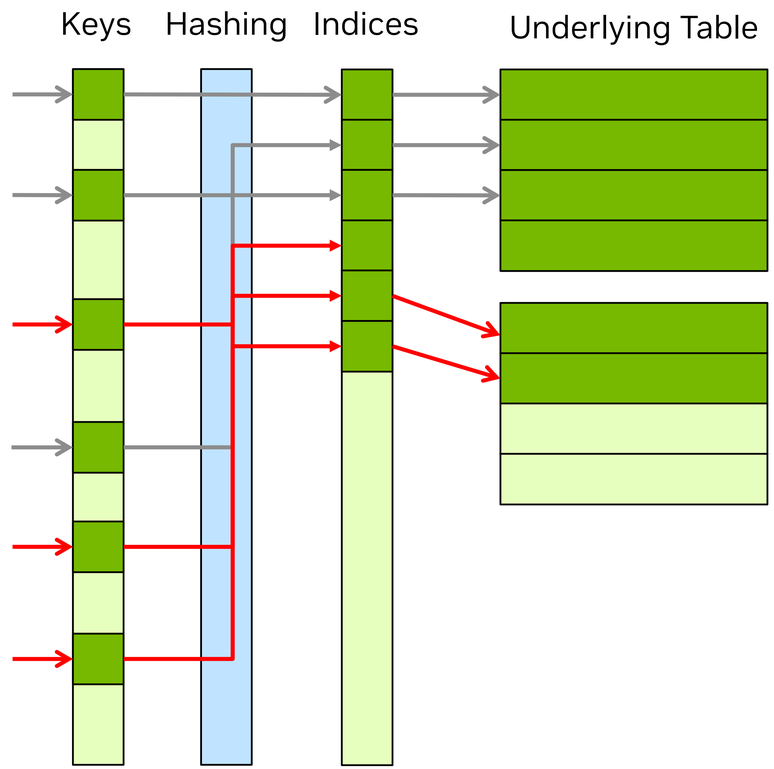
- Start from a small, predefined capacity, and then grow the capacity on demand



(1) Initial setting



(2) Capacity is full



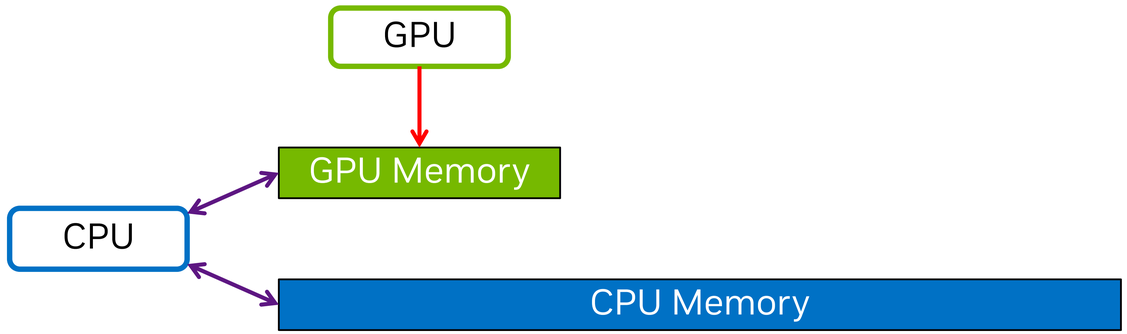
(3) Grown

* Note: this is a very simplified view. The implementation details can be more complex.

Dynamic Embedding Table

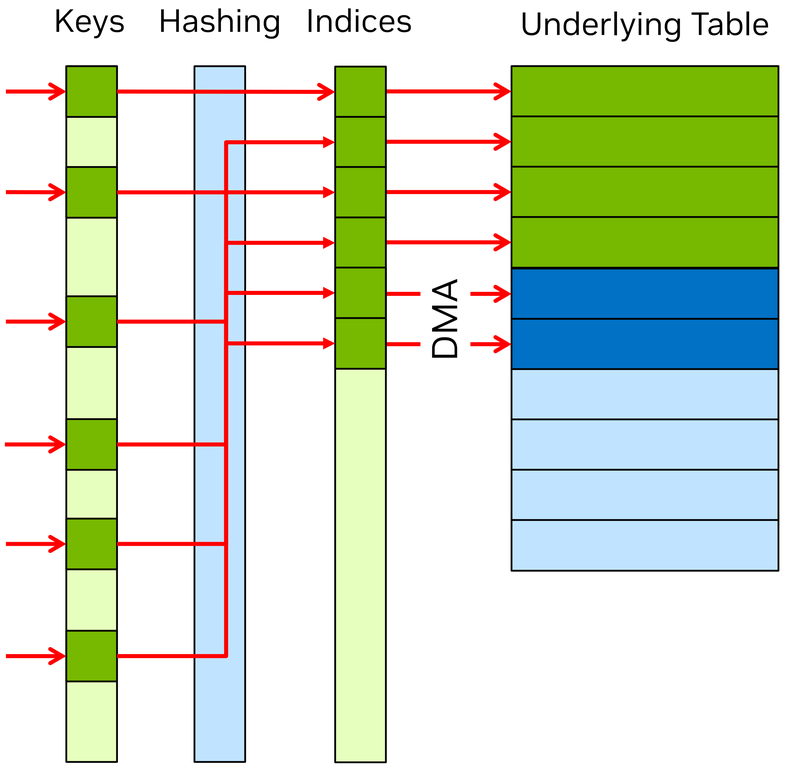
Handling “True” Full Capacity

- The capacity of dynamic embedding table cannot grow beyond the GPU memory capacity
- If new keys occur, the least frequent embedding vectors must be removed from the embedding table
- How should the victim vectors be handled?
  - Naïve approach - just discard it
    - Simple but affects accuracy
  - Conventional approach – use CPU memory as the secondary storage
    - More complex to implement, possibly poorer performance



Dynamic Embedding Table

Alternative Approach



- Treat CPU memory as the extension of GPU memory based on DMA
- Performance is limited by the data distribution and PCIe bandwidth
- More suitable for GH200 Grace Hopper Superchip
  - NVLink-C2C provides 900GB/s bidirectional bandwidth between CPU and GPU
- Example implementation - https://github.com/NVIDIA-Merlin/HierarchicalKV

Performance Evaluation

Evaluation Setup

- Training System - A cluster of NVIDIA DGX H100
  - Each DGX H100 has 8xH100 Tensor Core GPUs and 4xNVSwitchs

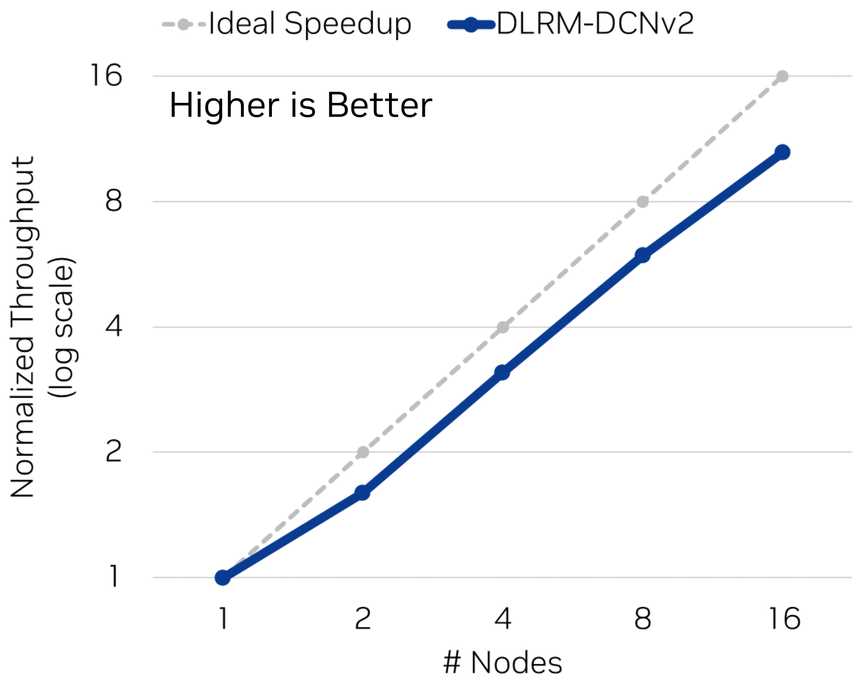
- Model

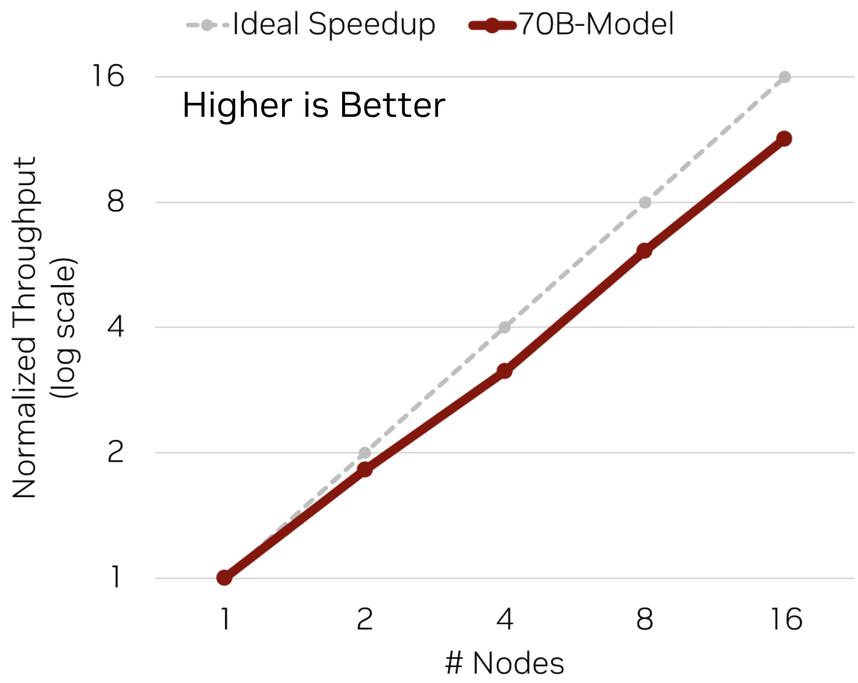
Model	Source	# Emb. Tables	Avg. Hotness	Emb. vector size
DLRM-DCNv2	MLPerf Training 3.0	26	10	128
70B-Model	Synthetic	180	80	{32, 128}

- Framework - NVIDA HugeCTR
  - But the introduced optimizations and practices are not limited to a specific framework

Scale-out Performance

Weak-Scaling Speedup





- Employed optimizations: better sharding + fused embedding tables + A2A traffic reduction + Better MLP + Pipelining
- A bigger model tends to achieve better scalability
- Observed the similar tendency in a DGX A100 based cluster

Takeaways

- A GPU based cluster is an attractive choice in training a large-scale recommender model if its high bandwidth memory and interconnect are well leveraged
- Considering diversities of models, datasets, and platforms, the best suite of optimizations and practices must be selected and implemented carefully
  - Sharding strategy
  - Embedding table fusion
  - A2A traffic compression
  - MLP optimizations
  - Pipelining
- Dynamic embedding also should be a good option to be explored, esp., if you don’t have enough number of GPUs and/or it is hard to predict embedding table sizes

More Resources on Recommender System Training

- At GTC 2024:
  - S61278 - Accelerate Recommender Systems and Increase GPU Utilization With Multiple CUDA Streams
  - S62277 - Implementing Lock-Free Linked-Lists Hash Table with a High-performance Memory Pool on GPU

- NVIDIA Blogs:
  - Fast, Terabyte-Scale Recommender Training Made Easy with NVIDIA Merlin Distributed-Embeddings
  - Accelerating Embedding with the HugeCTR TensorFlow Embedding Plugin
  - Merlin HugeCTR Sparse Operation Kit – Part 1 [CN]
  - Merlin HugeCTR Sparse Operation Kit – Part 2 [CN]
  - Breaking MLPerf Training Records with NVIDIA H100 GPUs

- NVIDIA Repositories:
  - Distributed Embeddings
  - HugeCTR & SOK
  - HiearachicalKV