

Building a Highly Efficient Inference System for Recommendations AI

Lu Fang, Ilina Mitra

Multi-Stage Ranking



01

Problem Statement

Provide a ranked list of Ads and content (stories, reels, posts, etc.) to show the user.



02

Latency Constraints

End-to-end ranking must complete within strict real-time budget of **$O(100\text{ms})$** .



03

Cost Efficiency

Expensive heavy-model computation is strictly reserved for the most promising candidates.



04

Funnel Architecture

Narrows candidate set with increasing model complexity:

Billions → Thousands → Hundreds → Final List

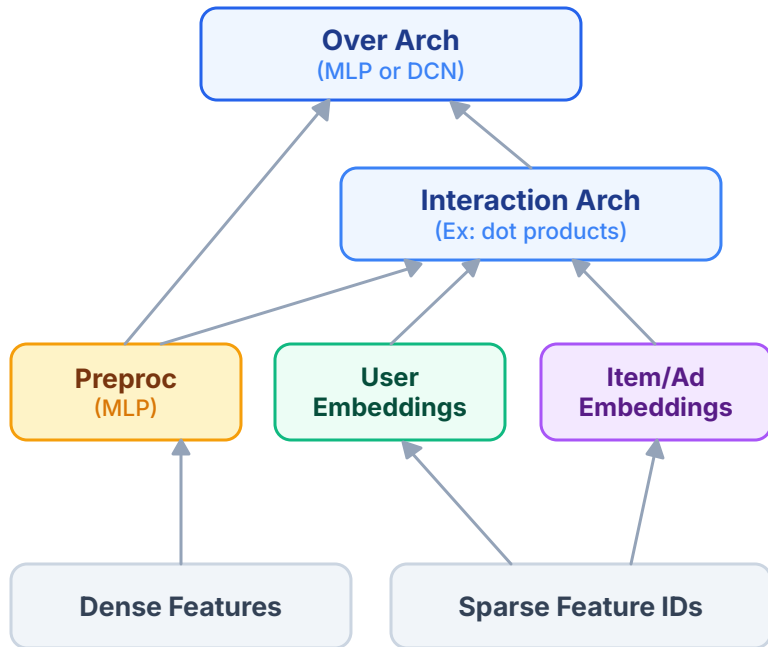


A high-performance inference system is critical to serve Meta's most sophisticated machine learning models for Recommendation Systems.

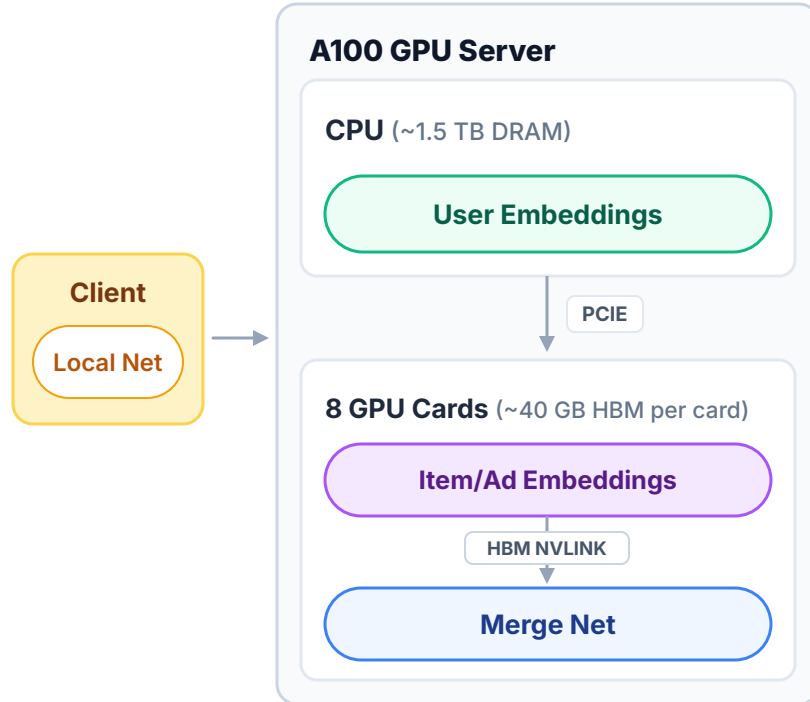
Mapping DLRM Model onto NVIDIA

Mapping high-dimensional embeddings and dense compute layers onto CPU & GPU memory hierarchy

DLRM MODEL ARCH

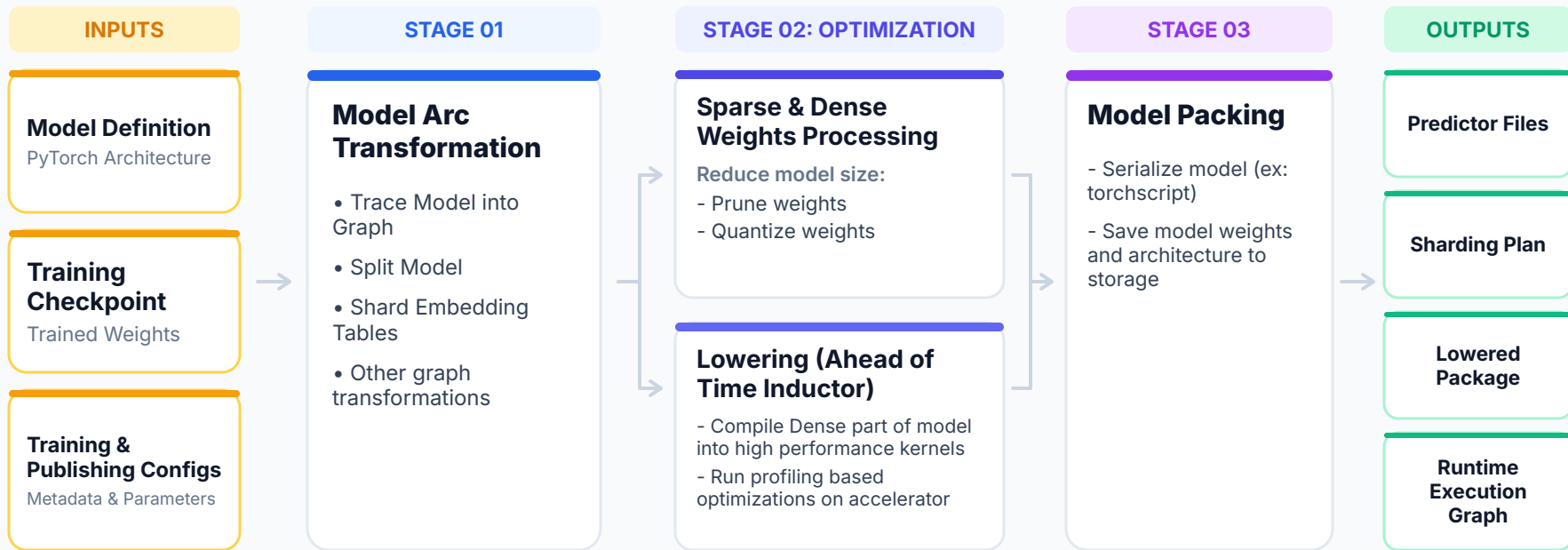


HARDWARE MAPPING



Inference Model Processing

Automated transformation pipeline converting trained PyTorch checkpoints into production-ready runtime artifacts



Pipeline Objective: Converts raw training models into sharded, lowered, and serialized runtime artifacts optimized for high-throughput accelerator serving.

Model Loading and Execution



01

C++ High-Performance Server

Engineered in C++ for maximum execution speed and ultra-low latency inference.



02

DAG Representation & Executor

- DAG nodes deployed across distributed hosts
- Efficient communication library between components



03

Zero-Copy In-Place Model Updates

Seamless transition between model versions without requiring significant extra capacity during transition.



04

Decoupled Structure & Weights

Separate loading of model structure and weights optimized for extremely large-scale models.

Developer Flow

End-to-end model creation, iterative validation loop, A/B testing, and production deployment pipeline

STAGE 01

Generate Model

Build & compile initial candidate inference model.

Debug Loop

Receives model fixes when validation fails.

STAGE 02: VALIDATION LOOP

Test Performance

Evaluate throughput & latency metrics.

Test Accuracy

Verify precision, recall & model outputs.

Tight Loop Iteration

Repeat until BOTH perf & accuracy pass

STAGE 03

Enter A/B Testing

Deploy candidate model to subset of live production traffic.

Validation Goal

Top-line metrics & real-world system performance stay green.

STAGE 04: RELEASE

Launch!

Full production rollout across serving cluster with zero-copy updates.

Production Ready

Continuous monitoring & automated fallback active.

Pipeline Feedback Policy: If performance or accuracy fails during Stage 2, loop back to Stage 1 to debug generation issues. Proceed to A/B testing only when top-line metrics and validation gates pass.

Key Optimizations



01

Distributed Inference

Sharding across GPU and CPU parameter servers for scale



02

Table Batched Embedding

Single fused kernel execution to eliminate launch overhead



03

More Efficient Computation

Custom kernel optimizations and streamlined compute graphs



04

AI Compiler

Automated model structure tuning, graph lowering, and targeted code generation



05

Freshness

In-place zero-copy APIs for seamless live updates without capacity overhead

Distributed Inference

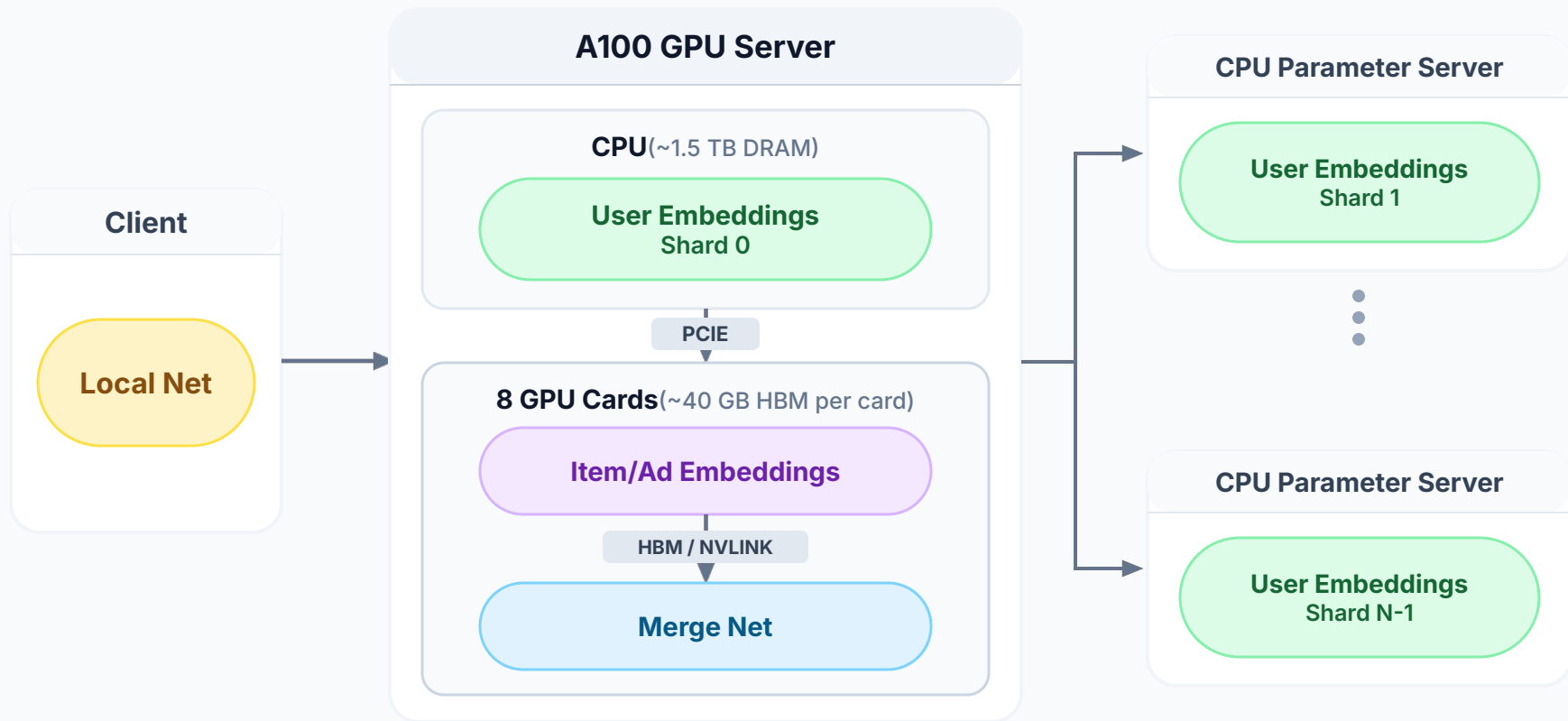


Table Batched Embedding

The Problem

Kernel Launch Overhead

Querying embedding tables individually incurs significant latency and performance bottlenecks.

Scalability Bottleneck:

Impacts performance exponentially when managing tens, hundreds, or thousands of tables simultaneously.

The Solution

Group All Queries Together

Data Transfer Optimization

Embedding lookups are data transfer heavy (effectively hashmap queries).

Single Fused Kernel Execution

Handles N embedding table lookups in one launch to maximize efficiency.

HBM Memory Caching

HBM serves as high-speed Cache for CPU DRAM when storing large tables.

Make Computation More Efficient

The Problem

Model computation becomes increasingly heavy, yet single requests may not fully saturate the hardware capacity.

Solution 01

Quantization

Embedding Tables

Handles bf16 & int8, with int4 on specific tables. Quantization is customized per-table based on numerical sensitivities.

Dense Layers

Higher sensitivity; typically requires fp16 or bf16. Requires balancing dynamic range vs. accuracy regressions.

Solution 02

Batching

Execution Efficiency

Reduces overall number of kernel launches and significantly improves hardware throughput.

Input Semantics

Requires understanding input dimensions to determine dynamic batch size so requests concatenate appropriately.

AI Compiler & High-Performance Kernels

The Problem

Overhead & Inefficiency

We need to remove unnecessary overhead and improve hardware efficiency.

High Execution Bottlenecks:

Python-dependent JIT workflows introduce runtime latency and preclude pure C++ deployments.

Manual Kernel Tuning Effort:

Writing hardware-specific kernels for NVIDIA CUDA or AMD AITER requires significant engineering effort.

The Solution

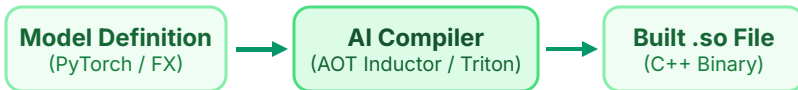
AI Compiler Pipeline

- **torch.compile / AOT Inductor:** Precompiles models into static runtime artifacts for C++ deployment.
- **Triton & HW Kernels:** Translates custom ops into C++ wrappers and utilizes native CUTLASS/CUDA or AITER.

Insight:

Actually, today LLMs serve as the ultimate AI compiler.

Compilation Flow:



Freshness

The Problem

Ranking models require real-time information, but full version updates take ~1–2 hours.

Solution 01

Weight-Only Updates

Architecture Preservation

If model architecture doesn't change, weight-only updates execute much faster.

Serving Protection

Requires careful tuning of weight loading pace to avoid disrupting serving.

Solution 02

Delta Streaming

Interim Updates

Delta and streaming updates happen continuously between full model version updates.

Real-Time Freshness

Keeps ranking signals fresh without relying solely on full version releases.

Best Practices

Module Sensitivity

Embedding Tables

More tolerant of partial updates.

Dense Modules

Highly sensitive; use buffer modules for full swaps rather than updating individual weights.



Questions & Answers

Thank you for your attention. We are ready to take your questions.