Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward not Backwards

KAIST

Youngeun Kwon and Minsoo Rhu
Research Scope
DNN based recommendation models

Provides “personalized” recommendation for ads/apps/contents/...
Dominant application in industry

Recommendation model is the most successful AI application in the industry

* Acun et. al., “Understanding Training Efficiency of Deep Learning Recommendation Models at Scale,” HPCA-2021
Model architecture overview

Embeddings are the key bottlenecks of deep learning recommendation models (DLRM)

Input data  Embeddings  DNNs

User feature  Embedding Table  DNN Layers  Prediction

Item feature  Embedding Table

etc.  Embedding Table

DNNs
Hybrid CPU-GPU system for DLRM
Memory capacity requirements make embeddings reside in the CPU.
Hybrid CPU-GPU system for DLRM

Memory capacity requirements make embeddings reside in the CPU

[Diagram showing CPU and GPU components with user feature, item feature, and etc. leading to Embedding Table and further to DNN Layers and Prediction with a Major Bottleneck indicated.]
Key primitives of embedding layer
Performs extremely memory bandwidth intensive operation

![Diagram of embedding layer](image)

- **Input data**
- **Embeddings**
- **DNNs**

- **CPU**
  - Table Indices
  - Embedding Table
  - Gathered Embedding

- **GPU**
  - DNN Layers
  - Prediction
Caching approach for embeddings
Caching frequently accessed entries within the fast GPU memory
Caching approach for embeddings
Caching frequently accessed entries within the fast GPU memory
Key Observations
Conventional cache policies

Best effort speculation on future accesses, based on past history

![Diagram showing conventional cache policies]
Conventional cache policies
Best effort speculation on future accesses, based on past history
For recommendation model training
All embedding accesses can be known in advance from training dataset

Generated from training “dataset”

Table Indices (Lookup addresses)

Embedding Table (Memory)

Embedding Cache (Cache)

Future Current Past

3 2 1
5 3 0
7 7 2
9 8 2
...

Which to evict?

fill

evict

E[0]
E[1]
E[2]
E[3]
E[4]
E[5]
E[6]
E[7]
E[8]
E[...]

E[6]
E[3]
E[7]
E[...]
E[...]
E[...]
E[...]

Past

Future

Current


Our solution: a cache that “always” hits
Proactively prefetch all required data into cache before the access happens

Gives an illusion of “GPU-only” training
(A Naïve) Straw-man Cache Architecture
Software-managed dynamic caching

How to fill/evict cache entries by software in runtime

CPU

Input Index
1 4 5 7 8

Embedding Table

GPU

DNN Layers

Embedding Cache

Prediction
Software-managed dynamic caching

How to fill/evict cache entries by software in runtime

[Diagram showing CPU and GPU processes with embedding tables and cache structures]
Query stage
Classify cache hit/miss and select victims for eviction

Hit
Missed

Input Index
1 4 5 7 8

Embedding Table

CPU

Query
Collect Exchange Insert Train

GPU

DNN Layers
Embedding Cache
Prediction
Collect stage

Collect embedding entries to be filled into GPU and evict from GPU
Exchange stage
Transfer collected embedding entries via PCIe channel

CPU

Embedding Table

<table>
<thead>
<tr>
<th>Input Index</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>E[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E[0]</td>
</tr>
<tr>
<td>E[…]</td>
<td>E[…]</td>
<td>E[…]</td>
<td>E[…]</td>
<td>E[…]</td>
<td>E[…]</td>
</tr>
</tbody>
</table>

GPU

DNN Layers

Embedding Cache

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>E[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E[0]</td>
<td>E[0]</td>
<td>E[0]</td>
<td>E[0]</td>
<td>E[0]</td>
</tr>
</tbody>
</table>

Input Index

1 4 5 7 8

1 4 5 7 8


Predcition

Query Collect Exchange Insert Train
Insert stage

Insert evicted/required entries to embedding table/cache
Train stage

Everything is ready in GPU, perform fwd&bwd propagation of training
Timeline of single training iteration

Caching steps include high-overhead CPU embedding operations

Major bottlenecks

Query | Collect | Exchange | Insert | Train

“GPU-only” speed
Timeline of single training iteration

Caching steps include high-overhead CPU embedding operations

Major bottlenecks

```
Query  Collect  Exchange  Insert  Train
```

“GPU-only” speed

“Prefetching future inputs”

=  

“Pre-computing caching operations”

=  

“Pipelined execution”
Pipelined execution of stages
Proactively prefetch using future access information

Batch$_i$
- Query
- Collect
- Exchange
- Insert
- Train

Batch$_{i+1}$
- Query
- Collect
- Exchange
- Insert
- Train

Batch$_{i+2}$
- Query
- Collect
- Exchange
- Insert
- Train

Batch$_{i+3}$
- Query
- Collect
- Exchange
- Insert
- Train

Batch$_{i+4}$
- Query
- Collect
- Exchange
- Insert
Data dependencies on embeddings

How to ensure fetching/evicting up-to-date values?
Data dependencies on embeddings

How to ensure fetching/evicting up-to-date values?

Batch $i$
- Query
- Collect
- Exchange
- Insert
- Train

Batch $i+1$
- Query
- Collect
- Exchange
- Insert
- Train

Batch $i+2$
- Query
- Collect
- Exchange
- Insert
- Train

Batch $i+3$
- Query
- Collect
- Exchange
- Insert
- Train

Batch $i+4$
- Query
- Collect
- Exchange
- Insert
- Train
Our Proposal: ScratchPipe Architecture
How to resolve dependencies

Knowing future accesses applies the same to the victim selection

Which to evict?

Always containing the latest value

Embedding Table

Embedding Cache
ScratchPipe’s execution model

Future-aware cache management mechanism for embedding training
What actually happens in “Plan” stage

How to choose victims wisely?

Algorithm 1 ScratchPipe Pipeline Controller

1: Key data structures for ScratchPipe control
2: HitMap = {key,value) storage to index cached embeddings in storage
3: HoldMask = [circular queue tracking cached locations]

4: HoldMaskWidth ← 3

5: Step A: A new mini-batch enters the [Plan] stage
6: Each asynchronous HoldMask ← PipelineMiniBatchQueue[“Plan”]

7: 1: Advance HoldMask by one cycle
8: for i = 0 to CacheSize do
9: 2: HitMask[i] ← HoldMask[i] >> 1
10: end for

11: for i = 0 to NumberOfSparseIDsWithinMiniBatch do
12: 3: If a sparse ID is found in HitMap, set the corresponding HoldMask
13: if HitMap[hitIdx] is found in HitMap then
14: HitMask[hitIdx] ← HitMap[MiniBatch[i]]
15: HoldMask[hitIdx][HoldMask[hitIdx]] ← 1
16: end if
17: else
18: victimIdx ← CHOOSEVICTIM(HoldMask)
19: HitMap[MiniBatch[i]] ← victimIdx
20: HoldMask[victimIdx] ← HoldMask[victimIdx] | HoldMaskWidth−1
21: end if
22: end for
Evaluation
Evaluation methodology

Environment and implementations

- Software implementation on NVIDIA DGX-1V system
  - Pytorch C++ API
  - NVIDIA cuDNN / cuBLAS
  - NVIDIA’s NVLabs CUB
Evaluation methodology

Environment and implementations

- Software implementation on NVIDIA DGX-1V system
  - Pytorch C++ API
  - NVIDIA cuDNN / cuBLAS
  - NVIDIA’s NVLabs CUB

- Evaluated systems
  - Hybrid CPU-GPU
  - Straw-man
  - ScratchPipe
Evaluation methodology

Benchmarked model and input dataset

- Based on DLRM MLPerf benchmark
  - 8 embedding tables with 10M entries per table (total 40 GB model size)
  - Cache size: 2% – 10% of total embedding table

- Input dataset
  - Synthetically generated using real-world dataset’s access patterns
  - Random, Low, Medium, and High locality benchmarks
Overall performance

Achieves avg. 5.1x speedup compared to the hybrid CPU-GPU
TCO (Total Cost of Ownership)
Achieves 2-6x cost reduction compared to multi-GPU system

<table>
<thead>
<tr>
<th></th>
<th>8 GPU (GPU-only)</th>
<th>1 GPU (ScratchPipe)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AWS Price: $ 24.48/hr</td>
<td>AWS Price: $ 3.06/hr</td>
</tr>
<tr>
<td>Iteration Time</td>
<td>1M Iteration Cost</td>
<td>Iteration Time</td>
</tr>
<tr>
<td>Random</td>
<td>16.22 ms</td>
<td>$ 110.3</td>
</tr>
<tr>
<td>Low</td>
<td>16.12 ms</td>
<td>$ 110.2</td>
</tr>
<tr>
<td>Medium</td>
<td>17.82 ms</td>
<td>$ 121.2</td>
</tr>
<tr>
<td>High</td>
<td>18.61 ms</td>
<td>$ 126.6</td>
</tr>
</tbody>
</table>

* Comparing our single-GPU ScratchPipe system with the model parallel GPU-only system using 8-GPUs
Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward not Backwards

The “first” application specialized performance-optimized caching solution for sparse embedding layers

A “software-only” solution that significantly improves import AI workloads performance

Average “5.1x” performance improvement on prior hybrid CPU-GPU system
Questions?