Supporting Massive DLRM Inference through Software Defined Memory

Ehsan K. Ardestani, Changkyu Kim, Seung Jae Lee, Luoshang Pan, Jens Axboe, Valmiki Rampersad
Banit Agrawal, Fuxun Yu, Ansha Yu, Trung Le, Hector Yuen, Dheevatsa Mudigere, Shishir Juluri
Akshat Nanda, Manoj Wodekar, Krishnakumar Nair, Maxim Naumov, Chris Petersen
Mikhail Smelyanskiy, Vijay Rao
Meta Platforms Inc.
Menlo Park, USA

***Abstract*—Deep Learning Recommendation Models (DLRM) are widespread, account for a considerable data center footprint, and grow by more than 1.5x per year. With model size soon to be in terabytes range, leveraging Storage Class Memory (SCM) for inference enables lower power consumption. This paper evaluates the major challenges in extending the memory hierarchy to SCM for DLRM, and presents different techniques to improve performance through a Software Defined Memory. We show how underlying technologies such as Nand Flash and 3DXP differentiate, and relate to *real world* scenarios, enabling from 5% to 29% power savings.**

I. Introduction

Recommendation models are ubiquitous across web companies [1]–[6], with ranking and click through rate (CTR) prediction [7]–[9] being among the widely deployed use cases. Such use cases account for a considerable demand in infrastructure resource [10]–[12] and rapid increase in datacenters footprint [13].

Deep learning recommendation models (DLRMs) are often composed of sets of *fully connected layers (MLPs)* and *embedding tables* [14], and tend to be very large with up to trillions of parameters. One of the main reasons for such high number of parameters is that more sparse features (materialized through embedding tables) usually result in better model quality [8]. Hence the model size is mainly dictated by the *embedding tables*, which could account for 100s of Gigabytes at the time of serving (inference), and increases rapidly year over year (e.g. 1.5x per year [15] or more).

The massive size of DLRM models requires considerable amount of memory capacity to serve. Relying on DRAM is expensive. Interestingly, not all such capacity is required at the same memory bandwidth (BW, i.e. bytes retrieved per query from an embedding table). There is a high variation in BW and Size among the embedding tables with some being accessed many times per query (e.g. 1000 accesses per query, hence requiring high memory BW), while others do not (e.g. 1 access to a row hence low BW requirement). The inherent difference between batched accessing in *user* related embedding tables and *item* related embedding tables (explained in Section II) further skews such BW requirement, resulting in majority of capacity to require much smaller BW compared to a subset (mainly *item* related ones) requiring high BW[1]. Figure I shows an example of such skew.

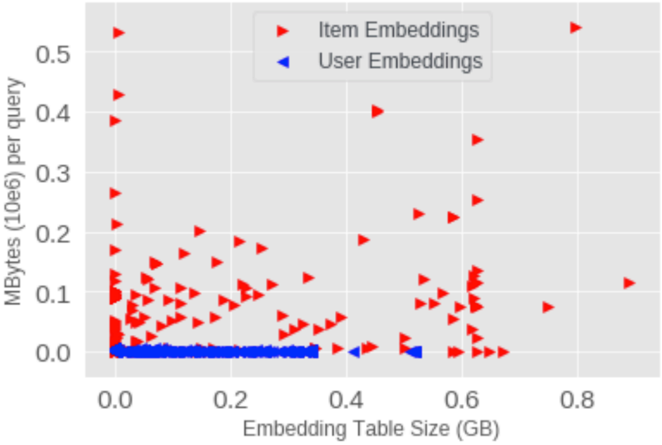


Fig. 1. **Embedding Table Size (x-axis) and Bytes per query (y-axis) in a 140GB model. The model has 734 tables, out of which 445 are user tables accounting for 100GB. Majority of tables, and hence model capacity, requires low BW.**

Presence of locality accessing the embedding tables (Section IV-B) would further allow for leveraging slower, but denser memory through caching [16].

Extending the memory hierarchy beyond DRAM to include slower memory technologies, such as Storage Class Memory (SCM), provides a cheaper and more power efficient approach to increase memory capacity per host. Considering the scale of deployment, the power saving could be in the order of 10s of Mega Watts. Given the importance of power in serving such models, it is becoming increasingly appealing to leverage a tiered memory. However, given the latency and BW requirement, and access granularity issues, deploying such a solution is challenging.

Corresponding author Ehsan K.Ardestani (ehsanardestani@fb.com). Fuxun Yu and Trung Le contributed while interning at Meta.

[1] For example, one user's inference query could access the user embeddings once for that user, while access the items' embeddings for a batch of items.

This paper presents a software defined memory system which extends the memory hierarchy to SCM to accommodate the ever increasing memory capacity needs of massive DLRM models at inference. To our knowledge, this is the first paper that not only entertains the possible solutions to some aspects of enabling such technology for inference, but also evaluates all the challenges that need to be addressed by pushing the solution all the way to *real world* datacenter deployment, and evaluating how the solution adds value for the end to end warehouse scale usecase.

The contributions of this paper are as follows:

- Extends memory available to DLRM using SCM through a Software Defined Memory stack, which can leverage different underlying technologies such as Nand Flash and Optane SSD.
- Enables smaller granularity of read access, down to dword, for NVMe devices, which saves latency and BW, and avoids read amplification.
- Evaluates pooled embedding cache to improve performance by bypassing dequantization and pooling when possible, and considers a range of trade offs with the cheaper capacity in slower memory to gain further performance when possible, namely de-quantization and depruning at load time.
- Presents end to end results for running the usecases, and discusses the added value of the solution in realistic warehouse scale deployment scenarios.

II. Background

A. DLRM Architecture

Recommendation models rank a set of items according to a user's preference. For example, Amazon uses the recommendation models for selecting items in its catalog [4]–[6], Netflix for showing movie options [1], Google for displaying personalized advertisements [3], and Facebook for ranking and click through rate (CTR) prediction [7], [12].

Deep Learning Recommendation Models [14] are often composed of two main components:

1) Embeddings, which map the categorical features (e.g. what subject a user has shown interest in) into dense representations. Different categorical features have varying cardinality, and hence require different size when materialized through embedding tables. The embeddings could be further divided into **user embeddings** (materializing categorical features for users) and **item embeddings** (materializing categorical features for items to be recommended such as news and movies). The embeddings are typically memory intensive.
2) Interaction, which aggregates continuous features and the dense representation of categorical features (e.g. by concatenation), and captures their complex interaction (e.g. by multi-layer perceptrons (MLP)). The interaction components are typically compute intensive.

Figure 2 depicts the high-level architecture of DLRM models. Bottom MLP reprojects the continuous features (e.g. age of the user) to dense ones. The embeddings components convert the categorical features to dense representation, and the top MLP captures the interaction of all the features.

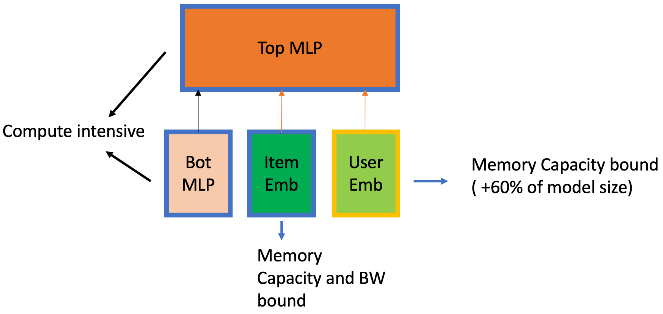

Top MLP
Compute intensive
Bot MLP
Item Emb
User Emb
Memory Capacity bound (+60% of model size)
Memory Capacity and BW bound

Fig. 2. **High level DLRM architecture.**

B. BW and Capacity Requirement

BW requirement for embeddings can be defined as

$$BW = QPS * \sum(p_i * d_i), \quad i \in T, \tag{1}$$

where **QPS** is Query Per Second, which is the rate at which the inference queries are expected to be processed in a given host, p_i is the **pooling factor** (number of embedding rows which needs to be looked up per query) for table i, and d_i is the **embedding dimension** for table i. T is the number of embedding tables in the model. Note that the number of rows in the tables does not impact BW.

For inference, several items will be evaluated (ranked) to arrive at the top items for recommendation. Hence an inference query could access the user embeddings once for that user, while accessing the items' embeddings for a batch of items. The user side embedding results could be broadcasted to all the items for Top MLP computation. It's worth nothing that this is different from training where one input sample consists of one user and one item. As a result, the *BW requirement per query for user embeddings is much lower than that of item embeddings*. We can rewrite Equation 1 as follows:

$$BW = QPS * (B_I \sum(p_i * d_i) + B_U \sum(p_j * d_j)), \\ i \in T_I, \quad j \in T_U, \tag{2}$$

where the batch size for Items and Users (B_I and B_U, respectively) is separated. T_I and T_U denote the number of item and user embedding tables, respectively. Given the latency sensitivity of inference queries, B_U is typically 1. B_I could in in order of 10s or 100s of items (depending on how fast they can be processed).

Our observation shows that more than 2/3 of the model capacity are contributed by the user embeddings. This could be due to the fact that there is a wider set of categorical features to describe the users, resulting in more user embeddings being used in the model. The implication is that *the bigger portion of model size have lower BW requirement.*

Another important observation is that the execution of user and item embeddings are independent, while the Top MLP has dependency on both. Assuming the user embeddings are the prime candidate to be accommodated by slower memory [2], *as long as the access time for user embedding is still smaller than that of item embeddings, the slower access due to slower memory is not exposed* in end to end latency. Equation 3 formulates, at high level, the time budget for the slower memory.

$$time(UserEmbeddings) = time(ObjectEmbeddings), \tag{3}$$

which could be elaborated further as follows:

$$BW_q(user)/BW_{SlowMem} = BW_q(items)/BW_{FastMem}, \tag{4}$$

$BW_q(user)$ refers to the BW requirement at a given query for user embeddings, and $BW_q(items)$ denotes that of item embeddings.

C. Hyper-Scale Deployment

Latency and Throughput: The inference of DLRMs are both latency and throughput sensitive. The latency sensitivity is derived from real time user interaction, requiring the latency in 10s of millisecond range for the ranking. At the same time, queries at Data Center level need to be processed within the expected throughput. Given QPS per host at a given target latency, the total throughput (e.g. in a DC region) will translate into a set number of hosts (Equation 5-7, $Comp_{HW}$ and BW_{HW} denote the compute and BW capability for a particular HW, $Comp_q$ and BW_q denotes compute and BW needed per query). Note that the latency requirement varies across different model/usecases. For example some models have strict p99[3] latency requirement with active load balancing to ensure the latency requirement across the fleet. Other models/usecases could have desired p95 latency which is achieved through static allocation of resources.

$$\begin{aligned} &QPS(HW, q) \propto \\ &min(BW_{HW})/BW_q,\ Comp_{HW}/Comp_q), \end{aligned} \tag{5}$$

$$\begin{aligned} &Latency(HW, q) \propto \\ &sum(BW_q/BW_{HW},\ Comp_q/Comp_{HW}), \end{aligned} \tag{6}$$

$$Resources(HW) \propto QPS_{Total}/QPS_{HW}, \tag{7}$$

Scale Up vs Scale Out: As the model size increases, either the memory per host needs to increase (scale up) or the model needs to be sharded to scale the memory by leveraging multiple hosts (scale out, e.g. see [17]). Extending the memory to SCM could be considered as a scale up only approach, or applied to the hosts involved in scale out to reduce the fan out. It needs to be mentioned that increase in model size usually is accompanied with increase in compute intensity of the model as well. So the relevant approach to serve the model would depend on the compute, memory BW and memory capacity requirement and their relative ratio.

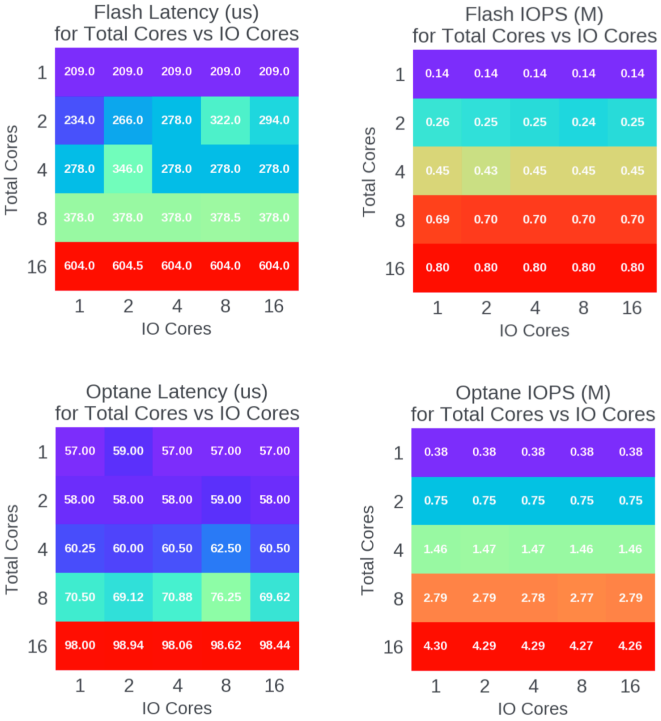


Fig. 3. **IOPS and latency for Nand Flash and Optane SSDs. Given each query to a table involves multiple lookups (pooling factor), we benchmark each device with average of 20 lookups per IO. The latency is for the batch of 20 lookups. As the results show, Optane SSD provides much lower latency and higher IOPS than the Nand Flash.**

Power Boundness: DLRM models keep increasing in their complexity and size faster than the rate new data centers could be developed (e.g. see [13]). Furthermore, DLRM applications account for a considerable portion of infrastructure resources [10], [11]. This leads to power boundness of the usecase, with query/watt at the acceptable latency being the primary metric to solve for at scale. Tiered memory directly helps with this top line metric by 1) leveraging more power efficient per GB memory when possible, and 2) allowing for better system solution, e.g. not scaling out.

III. Technology

Extending the memory hierarchy to SCM can be deployed regardless of the choice of accelerators (e.g. using GPU for inference), and the hierarchy of faster memories (e.g HBM + DRAM). Hence, we refer to the the last level memory with SCM as *SM* (for slow memory) and the first level(s) of memory as *FM* (for fast memory).

There is a range of technologies that could be used for *SM*. Table I lists some of the currently more readily options. We track the following key parameters for each technology:

IO Per Second (IOPS). The access patterns to the embedding tables are random. We track IOPS instead of GB/s, because the embedding rows, and hence the access granularity to the *SM*, is typically much smaller than 4KB block size (read amplification). The inference access is read only, with non-frequent writes only during model update.

[2] Large item tables with low pooling factor could be considered for placement on the slower memory. However, without lack of generality, in this work we primarily focus on placing user embeddings on slower memory.

[3] Here, p99 denotes 99 percents of queries needs to be processed within the latency requirement, similar for p95.

TABLE I
DIFFERENT OPTIONS FOR THE SLOW MEMORY (*SM*). THE VALUES ARE BASED ON PUBLIC INFORMATION. COST IS RELATIVE TO DRAM

Technology	IOPS (M)	Latency (us)	Endurance	Access Granularity	Cost	Sourcing
PCIe Nand Flash	0.5	O(100)	5	4K	1/30	multi
PCIe 3DXP (Optane)	4	O(10)	100	512	1/5	single
PCIe ZSSD	1	O(100)	5	4K	1/10	single
DIMM 3DXP (Optane)	-	O(0.1)	-	64	1/3	single
CXL 3DXP	>10	O(0.5)	-	64-128	-	single

Access Granularity. The quantized embedding rows, while growing, are in 128-256B range.[4] IO Read with higher granularity (e.g. 4KB) will result in read amplification and wasted BW.

Latency. This is the loaded access latency for a block of data. Different technologies show different curve as the load increase from low to high. Given latency sensitivity of the usecase, we need to operate on a latency region that is in order of up to a few 10s of us.

Write BW. The only write access happens during model update. In general more symmetric read and write BW becomes more important as the update frequency increase.

Endurance. The endurance can translate to model update interval. ($UpdateInterval = 365 * ModelSize/(pDWPD^{5} * SMCapacity)$

Cost. Relative cost per GB compared to DDR4 DRAM

Sourcing. How many vendors offer the technology. The higher, the better.

Nand Flash provides the cheapest option, with multiple vendors offering the technology. However, it suffers from two drawbacks. Low random IO per second (IOPS), and increased latency as the IOPS increases (Lower endurance can be offset by capacity). This limits the usecase to models with low BW requirement.

PCI3 3DXP (Optane) provides a good random IOPS (4M at 512B) and considerably better latency profile compared to Nand flash (O(10) usec). The endurance is also high enough to accommodate frequent updates. As a result, Optane SSD can enable tiered memory for the frontier of the models with high capacity and BW requirements. Figure 3 shows the IOPS and latency profile for Nand Flash and Optane SSD.

PCIe ZSSD offers better latency compared to Nand Flash, but does not offer high enough IOPS to set it considerably apart from Nand Flash. DIMM 3DXP impacts the available memory BW to the CPU which is point of concern. CXL 3DXP would provide the best performance in the set, without having the negative side effect of DIMM 3DXP. But still not as readily available as other technologies listed here.

The choice of technology for *SM* depends on specific usecase and model characteristics. As the models scale size and BW, the higher BW options become more relevant. We observe that some models (e.g. lower-end with less strict p99 latency) can leverage Nand Flash without performance implications. Optane SSD enable tiered memory for a wide range of DLRM models including higher-end of BW requirement.

TABLE II
USECASES

Inference	user batch size = 1, item batch size > 1 (O(100)), Inference is latency sensitive.
InferenceEval*	The goal is accuracy validation. user batch size == item batch size > 1.

*InferenceEval is similar to eval after training, but model has gone through inference specific transformations such as quantization.

In this work, we only consider Nand Flash and Optane SSD as technology choices for *SM*. We do not consider DIMM Optane due to implications on BW. However, as the model's capacity and BW scale overtime, CXL based solution would become more relevant.

IV. DESIGN AND IMPLEMENTATION

We evaluated several different design choices for the software stack. Given the scheme could be used for a wide range of model configuration and underlying HW, we evaluate the design choices, such as cache organization, by evaluating a wide range of target models beyond what presented in the results section. We also consider both Inference as well as Inference Eval (see Table II). This is to avoid over designing for a particular usecase. Several tuning options are provided such that the desired serving configuration could be decided at model deployment time (e.g. through an auto-tuning tool). Such tuning options are highlighted as *Tuning API* in each subsection.

A. Fast IO

Most of the relevant *SM* technologies currently are block devices with NVMe interface. As the BW requirements of the model grow, the IOPS requirement consequently grow (Equation 8). However, IO through NVMe stack is still an expensive operation. Performing multi-million IO per second could required prohibitive amount of computing resource (CPU). We have chosen to use *io-uring* [18] due to its lower overhead per IO as it allows for less system calls, enables async polled IO, to mention a few. Figure 3 shows the performance characteristics of PCIe Nand Flash and PCIe Optane using io-uring.

$$IOPS \propto QPS * \sum(pi), \quad i \in Tables(SM) \qquad (8)$$

[4] We primarily use row-wise quantization

[5] Physical Drive Write per Day

One particular design choice was $mmap$ vs $DIRECT-IO$. Due small access granularity and lack of considerable spacial locality (Section IV-B), we observed that $mmap$ would not provide the best use of *FM* space, and results in higher access latency (by 3x. e.g. reading in and maintaining 4KB into memory for a 128B request). Hence we opted for $DIRECT-IO$ with an application level cache.

Given different technologies could be used for *SM*, we realized some optimizations are technology specific. For example with Nand Flash, we need to smooth out the bursts by limiting the maximum outstanding requests to the SSD because SSD controllers typically try to serve all possible outstanding requests which results in extra latency.

Tuning API: Total number of outstanding IOs per table and total number of tables that can be processed at given time.

1) Enabling small access granularity: Sub-block (e.g. 4KB) reads is not normally supported by an operating system. The higher access granularity to the SCM device, given the lack of spacial locality (Section IV-B) has three adverse implications: 1) higher latency due to read amplification as more data need to be transferred from device to the host; 2) more pressure on the interconnect (PCIe) in the system, which might require provisioning more PCIe lanes, and hence increased system power; 3) requiring extra memory copy to handle extracting row data from block data and copying it into the cache. Given that majority of tables have embedding dimension smaller than 512B at inference (due to quantization [20]), we have enabled arbitrary access granularity, down to DWORD, with NVMe. A two legged approach is taken to achieve this goal.

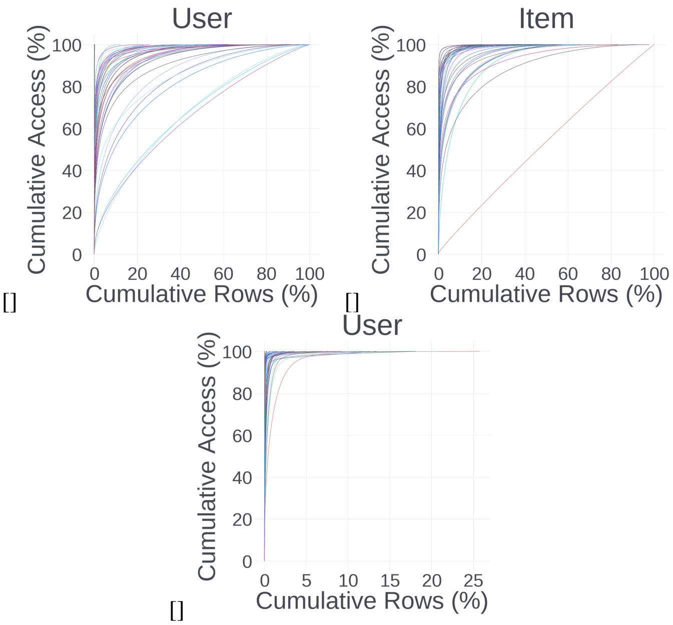
- Linux Kernel: Linux kernel is updated to allow a custom command over the *io-uring* [18] application transport that allows down to 4B granularity reads. Note that the device still accesses the data in the configured block (e.g. 512B) granularity internally.
- NVMe Driver: The NVMe Scatter Gather List (SGL) Bit Bucket is used to communicate the desired portion of a block. This allows full flexibility as to in which parts of a request the host is interested, hence only transferring the necessary parts of a read over the bus.

By only reading the parts of a block that is necessary, we save around 75% of the bus bandwidth and reduce the time needed to transfer this data. This reduces the observed latency of a given read by 3-5%. The savings at the application level are more given removal of the extra memcpy (see Section IV-C for more details).

Both of these features will be submitted for the upstream kernel, and will be publicly available.

B. Locality

Locality is an important characteristic of accessing embedding tables as it could allow for providing a higher effective BW for the data in *SM* by a cache in *FM*. Figure 4 captures *temporal locality* through the cumulative distribution of a range of categorical features. Majority of the features show a power law distribution, with a small subset of embedding rows accounting for majority of accesses, hence high temporal locality. We separate User and Item embeddings since we observe a meaningful difference in the distributions (item embeddings show more locality). This motivates the use of a Software Managed Cache in *FM* to cache the hot portion of embeddings placed in *SM*.



[]

[]

[]

Fig. 4. **Temporal Locality accessing User (top left) and Item (top right) embeddings. Access to majority of the tables demonstrate power law. For each plot, we track 50 tables at random, for data sampled post hash for 6 days. (bottom) shows temporal locality for the same set of user tables observed by one host during serving, indicating higher locality.**

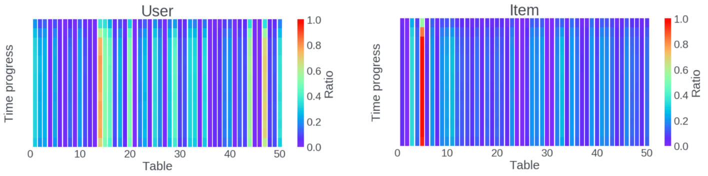


Fig. 5. **Spatial Locality accessing User and Item embeddings. Value 1.0 indicate 100% spatial locality. For each plot, we track 50 tables at random, for data sampled for 6 days. The average window is around 25M access per table**

Note that the temporal locality observed from a host also depends on the serving system. Inference queries will go through a scheduler/aggregator which routes a query to a specific host for ranking. Figure 4-(c) shows the temporal locality for the same set of user embedding tables, but observed from one host during serving, which shows higher locality. Enforcing a user-to-host sticky policy can help increase cache hit rate observed from a host.

Figure 5 demonstrates the degree of *spatial locality* accessing the table. It uses the average ratio of unique index to unique 4KB block size, normalized to the maximum unique index per block size per table, as proxy for spacial locality. The ratios are captured in intervals (average 25M access per table). Value 1.0

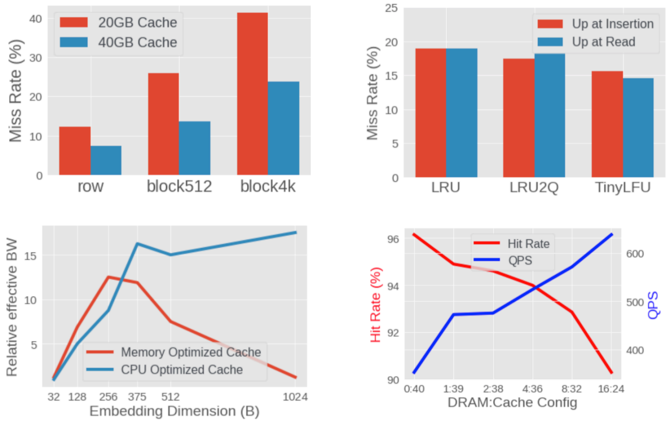


Fig. 6. **Performance implications of different cache organization choices. We opt for a unified row cache, which internally implements two caches optimized differently based on embedding size. The cache routes the requests to proper internal cache based on embedding dim (Embedding dim $<=$ 255 will be routed to memory optimized cache). The bottom right figure shows a case where direct placement on DRAM could have considerable impact on QPS.**

TABLE III
SUMMARY OF POOLED EMBEDDING CACHE PROFILING FOR 100M QUERIES. HIT RATE SHOWS THE NUMBER OF QUERIES WHICH GOT A HIT FOR AT LEAST ONE OF THE SUBSEQUENCES. GENERATED SEQUENCES SHOW THE NORMALIZED NUMBER OF SUBSEQUENCES GENERATED. WHILE CONSIDERING $0 < c <= P$ INCREASES THE CACHE OF A HIT FOR A SUBSEQUENCE, IN PRACTICE THE OVERHEAD OF FINDING SUCH SUBSEQUENCE IS PROHIBITIVE EXCEPT FOR c NEAR 1 OR NEAR P.

Scheme	Hit rate (%)	Generated sequences
c=10	26	O($\binom{avgP}{c}$)
c=10, top indices	19	O(100)
c=P	5	1

TABLE IV
AVERAGE LENGTH (NUMBER OF INDICES) SAVED FOR HIT IN POOLEDEMB CACHE WITH A 4GB CACHE SIZE.

LenThreshold	Hit Rate	Hit Avg Len
1	4.39%	11
4	4.58%	35
8	4.02%	40
16	4%	56
32	3.9%	76

indicates the same number of unique index and unique 4KB blocks, i.e. high spacial locality. The heat map and the cooler temperature overall indicates low spatial locality.

C. Cache Organization

The design and organization of the cache also has impact on the overall performance.

Unified Row Cache: Given the observation from the locality study (Section IV-B), we opted for a a cache organization where once logical cache serves all the embedding tables, and caches at embedding table row granularity (rather than IO block granularity). Hence referred to as unified row cache. The unified cache allow for better utilization of the memory space compared to per table cache, and the lack of considerable spacial locality motivated avoiding block cache. We use CacheLib [19]. Without the small granularity access (Section IV-A) we need to copy a block of data into an aligned *FM* buffer, and then copy the desired portion into the cache. This means more than 2X *FM* BW needed for every X data pulled in from *SM*, and increased latency due to the memory copies. With small granularity access, we can directly copy data to the cache storage, and save on *FM* BW and improve latency.

Memory vs CPU overhead: CacheLib allocates 31B of meta data per item stored in the cache for lifetime, LRU and internal bookkeeping. This is a considerable memory overhead considering that majority of tables, now and in foreseeable future, have embedding dimension smaller than 256B. The advantage is that each item is directly addressable (We call this CPU Optimized). To reduce the memory overhead, we consider a configuration where several items (e.g 8) are packed into a bucket, hence reduce the memory overhead per item (e.g. 31/8 4B). The downside is that to lookup an item, first the bucket is decoded, and then the bucket is searched for the requested item, hence increased CPU utilization per look up (We call this Memory Optimized). Given the overhead and performance results shown in Figure 6, we opted for a dual cache where tables with embedding dim smaller than 256B are routed to a Memory Optimized Cache, and to CPU Optimized Cache otherwise. We also evaluated multi-level cache (row cache backed by a block cache) but did not observe any benefit.

Tuning API: Cache sizes and number of cache partitions.

D. Pooled Embedding Cache

The *SM* cache stores the raw quantized embeddings. For every embedding operator, there are p_i embeddings read out for $table_i$ from the cache or from *SM*, which then go through dequantization and pooling [23] to generate the output for Top MLP. If we had the resulting pooled embeddings already cached (or even partial pooled embeddings), we could (partially) save lookup, dequantization and pooling.

We profile queries to establish whether there is locality in sequence of indices that appear across queries. Table III shows the profiling result. There is $\binom{P}{c}$ choices for an embedding operation with P indices. For a subsequence of indices of size $c, 0 < c <= P$, the possibility of a repeating subsequence decreases as c increases. However, except for near the edges (e.g. $c = 1$ or $c = P$), the number of possible subsequences is too large. In our profiling we limit the length of subsequence to 10, and only profile most frequent indices. Nonetheless, our observation is that in the case of $c = P$, where we only cache the full sequence and only lookup for the full sequence of indices for each table when a request arrives, provides small enough overhead, and reasonably high enough hit rate to have a chance at improving performance . Algorithm 1 depicts the implementation. We observed around 5% hit rate for the pooled embeddings (Table IV). The average length of requests

Algorithm 1 Algorithm: Pooled Embedding Cache

```
Input: Table, Indices
doPooledEmbCache = len(indices) $>$ LenThreshold
if doPooledEmbCache then
  $sequenceKey = hash(indices)$
  if $e = lookup(t, sequencekey)$ then
    return e //pooled emb vector exists in the cache
  end if
end if
for i in Indices do
  if not E[i] = lookup(t, i) then
    $prepareIO(t, i)$
  end if
end for
$submitIOs(E)$
// dequantize and pool all the embedding vectors in the sequence
for e in E do
  output += dequant(e)
end for
if doPooledEmbCache then
  cache[sequenceKey] = output
end if
return output =0
```

hit in PooledEmbedding Cache increase as the $LenThreshold$ is increased.

Algorithm 1 shows the implementation. We use an order-invariant hash to create a key from the sequence of indices in a request.

Tuning API: The min sequence length which could be cached is configurable ($LenThreshold$).

E. SM vs FM capacity Tradeoff

Given the cheaper *SM* capacity, we evaluated a few approaches that reverse the schemes commonly used to reduce model size, namely *de-pruniung* explained here, and de-quantization in Appendix A.

Algorithm 2 Algorithm: De-pruning at load time

```
Input: Tables
for t in Tables do
  if t is pruned-table then
    nt = new Table(dim=[t.mapper.dim[0], t.dim[1])
    for i in t.mapper do
      if i is pruned-row then
        nt[i] = createZero(i.dim)
      else
        nt[i] = t[i]
      end if
    end for
    t = nt
  end if
  SaveToSM(t)
end for =0
```

Pruning embedding tables post training is commonly used to reduce inference model size (e.g. see [17]). At high level, the embeddings rows with values very close to 0 are heuristically removed. A new tensor is defined to map the indices in the un-pruned space to indices in pruned space. The size of a mapping tensor is $NumRow(Unpruned) * IdxType$, $IdxType\epsilon\{4,8\}Bytes$. To place pruned embedding tables on *SM*, we can either 1) save both the pruned table and mapping tensor to *SM*, which means two accesses to *SM* per embedding lookup; or 2) place the pruned table on *SM* and keep the mapping tensor in *FM*. Given the IOPS boundness with *SM*, and relatively smaller size of the mapping tensor, options 2 is a more desirable choice. However, as the model size increases, the aggregate size of the mapping tensors increases. The space taken by mapper tensors are the memory that is taken away from the *SM* cache.

TABLE V
PLACEMENT STRATEGIES.

Policy	Description
SM only with Cache	all the (user) tables are mapped to *SM*. rely on Cache in *FM* to keep the hot rows in faster memory
Fixed *FM*, *SM* with Cache	some tables could be directly mapped to *FM* based on a given policy. The rest will be placed on *SM*
per table cache enablement	For *SM* cache in *FM*. Low temporal locality tables will not use the cache.

To free up the memory used by mapping tensors, we can *de-prune* the embeddings at the time of loading. Algorithm 2 shows how de-pruning is done. Beside increased model foot-print on *SM*, de-pruning could lead to extra accesses to *SM*, and consequently cache pollution. This is because the pruned embeddings now will be accessed and cached. However, the intuition is that the pruned embeddings are also less frequently accessed, hence the impact would be minimal. Our experiments confirm the intuition by showing 2.5% increase in the total requests, while allowing for up to 2x cache size in some configurations in practice. We see up to 48% increase in performance for cases where performance is bounded by user embeddings in *SM*.

F. Placement

With a software defined cache in *FM*, there will be two choice to use the *FM* space; 1) use all the available space for the cache 2) use portion of *FM* to map tables directly, and portions for the cache.

In general, allocating all the tables to *SM* and relying on the cache to keep the hot rows in *FM* will perform well across the board. However, given the extra overhead of looking up an embedding row from the cache vs plane memory, there are possibilities to improve the performance further with more detailed placement. Table V lists different placement categories. Figure 6 shows the impact of placement with different budget for direct placement on DRAM on a 150GB model running inferenceEval (which is more sensitive to placement than inference because the user and item batch sizes are the same).

Tuning API: Pre-defined placement policies based on table size and pooling factor can be enabled. We also implemented an option to providing a list of tables which should not be placed in *SM* (for more elaborate offline placement). All

TABLE VI
TARGET MODELS CONFIGURATION

Model	*M1*	*M2*	*M3*
Num parameters	143B	450B	5T
Size (GB)	143	150	1000
Num of user emb tables	61	450	1800
Emb table dim (B)	[90, 172]	[32, 288]	[32, 512]
(range [min, max], avg)	avg: 51	avg: 64	avg: 192
Avg pooling factor (PF)	42	25	26
User batch	1	1	1
Num of item emb tables	30	280	900
Emb table dim (B)	[90, 172]	[4, 320]	[32, 512]
(range [min, max], avg)	avg: 69	avg: 38	avg: 192
Avg pooling factor (PF)	9	14	26
Item batch	50	150	1000
Num MLP layers	31	43	35
Avg MLP size	300	735	6000

TABLE VII
HW CONFIGURATIONS. ALL THE CPUS ARE INTEL. FOR SSDS, N STANDS FOR NAND FLASH, AND O FOR OPTANE SSDS. SEE [24] FOR MORE INFORMATION ON THE ACCELERATORS.

Name	CPU	DRAM (GB)	SSD	Accelerator
HW-L	2xXeon	256	-	-
HW-S	1xXeon	64	-	-
HW-SS	1xXeon	64	2x2TB N	-
HW-AN	1xXeon	64	2x1TB N	Yes
HW-AO	1xXeon	64	2x0.4TB O	Yes

placement policies adhere to a configurable DRAM budget to place tables on DRAM directly.

V. RESULTS

This section evaluates the results for different models with different relevant underlying HW. Section V-A describes the models used and hardware platforms evaluated. While Section V-B and Section V-C present experimental results, Section V-D is based on mixture of experimental results on a current platform, and modeling and estimation to derive the performance on a future platform.

A. Experiment Setup

We consider 3 models with different characteristics which reflects models in use for different usecase. We use production traffic for the evaluation of the models.

We use a set of hardware platforms available in Data Center to evaluate different models, as listed in Table VII. The choice of platform among what is available in Data Center is driven by the usecase characteristics and requirements. Hence, some of the possible evaluation combinations are not feasible (e.g running the exact *M1* on *HW-SS* without *SM* as it would run out of memory).

B. Using simpler HW

In many occasions, a usecase has to select from a very limited set of available host types deployed in DC. Such different host types provide different CPU, DRAM, and Storage capabilities. Using *SM* for *M1* allows for lowering the DRAM capacity requirement per host to serve a model. This enables using single socket, 64GB DRAM *HW-SS* instead of dual socket 256GB DRAM *HW-L*. While each *HW-SS* can sustain lower QPS at the desired latency compared to *HW-L*, the more favorable compute to DRAM ratio of *HW-SS* plus having attached SSDs leads to 20% lower power consumption considering the full scale of the serving. Table VIII shows the results.

TABLE VIII
IMPACT OF USING SSD ON SAVING POWER CONSUMPTION. REPORTED POWER IS NORMALIZED. DEPLOYMENT WITH SSD CAN REACH THE SAME LATENCY AS DEPLOYMENT ON DRAM ONLY, RESULTING ON 20% POWER SAVING.

Scenario	QPS	Power	Total Hosts	Total Power
HW-L	240	1.0	1200	1200
HW-SS + SDM	120	0.4	2400	960

The IOPS required by the model at 120 QPS is around 246K ($120QPS \times 50Tables \times 42avgPF$). We observe cache hit rate of more than 96% in steady state which typically is reached within a few minutes after a full model update. This means less than 10K IOPS in steady state.

We observe higher p99 latency on *HW-SS* due to occasional long tail latency of Nand Flash. Nonetheless p95 is the metric of interest for this usecase, which is matched on *HW-SS*. Using *HW-SS* saves equivalent of 159.4 TB of DRAM for this particular model in production like settings.

C. Avoiding Scale-Out

M2 uses an accelerator enabled platform (*HW-AN*) due to its higher compute intensity [13]. The item embeddings as well as the dense part of the model is mapped to the accelerator. The user embeddings are mapped to the host CPU. *HW-AN* has adequate accelerator memory to host the item embeddings, however, the 64GB host DRAM is smaller than the 100GB memory required by the user embeddings. The extra memory required for the user embeddings is achieved through the scale out as presented in [17], using *HW-S* host types. A *HW-S* on average can serve 5 *HW-AN*.

For this usecase, using *SM* prevents scale out. However, given the accelerated QPS per host, a higher degree of IOPS is required from *SM* ($450QPS \times 450Tables \times 25AvgPF = 4.8MIOPS$). We observe more than 90% hit rate in the *SM* cache. So the average sustained IOPS required is around 480 kIOPS. As shown in Table IX, the two Nand flash on *HW-AN* provide aggregate minimal IOPS of around 1M. However, due to long latency accessing nand flash, we have to considerably underutilize the devices to keep the latency low. Hence in practice Nand Flash in this setup considerably impacts QPS. However, Optane SSD provide much higher IOPS and lower latency, keeping the user embedding processing out of the critical path. By removing the need to scale out, *HW-AO* reduces the power consumption by 5%.

At the same time, *HW-AO* simplifies the serving paradigm, as the scale out paradigm is more complex to operate, and more prone to failures given that many more hosts are involved

TABLE IX
IMPACT OF USING SSD ON POWER CONSUMPTION FOR *M2*

Scenario	QPS	Power	Total Hosts	Total Power
HW-AN + ScaleOut	450	1.0 + 0.25	1500 + 300	1575
HW-AN + SDM	230	1.4	2978	2978
HW-AO + SDM	450	1.0	1500	1500

in serving a single query. While the power saving is modest, it increases as the models grow.

D. Facilitate Multi-Tenancy

For *M3* we present the estimated results, as it is a future use case, with the chance to impact the design of the host type. *M3* represents a future model which could run on an updated accelerator-enabled platform(e.g. see [26]). The primary arguments for SDM in such platform is to limit the amount of DRAM deployed per host. The power savings come from allowing for increased accelerator utilization without becoming DRAM memory capacity bound through Multi-tenancy.

Multi-tenancy refers to running multiple models on the same host. This capability is becoming more important (e.g. see [15]) as it allows for co-locating models with different requirements, and balancing the utilization of different resources such as accelerator, CPU, and DRAM. The balanced utilization leads to increase overall host utilization, and power saving. As an example, at any given time, there are a large number of experimental models running, of which, a subset will eventually be promoted for full scale deployment. Our observation is that given the number of experimental models running per production models, and on average it consumes up to quarter of the allocated resourced. Such experimental models run on a small volume of traffic, and hence have low QPS requirement per model, which could leave the hosts underutilized. This becomes more important as more compute capability is packed into a single host with the advent of more powerful accelerators, increasing the computation and power cost of a model underutilizing a host. Co-locating more than one model on a given host increases utilization. Notably, the memory capacity requirement will scale with the number of models co-located together. Therefore, serving becomes memory capacity bound.

Using *SM* in this case prevents the memory capacity boundness due to the multi-tenancy, by increasing memory capacity available to the models per host.

To drive the *SM* capacity and BW requirement per host, we use *M3* as the representative model. We estimate the QPS on the target hardware by 1) measuring the QPS on an available similar hardware, and 2) extrapolating the QPS based on the expected increase in compute and BW of the future HW. The BW needed from *SM* could be calculated according to Equation 2. Table X shows the need for 36 MIOPS which could be satisfied by 9 OptaneSSD, each providing 4 MIOPS.

Given the number of experimental models and their required QPS, we observe 63% utilization of the hosts at the scale.

TABLE X
SDM-BASED HW CONFIGURATION FOR FUTURE MODELS (*M3*). NUMBER OF SSDS ARE DERIVED BASED ON BW, AND HENCE IOPS NEEDED FOR THE USER EMBEDDING TABLES (IOPS BOUND).

Model	QPS	User Tables	PF	Emb dim	Hit Rate	MIOPS	numSSDs
M3	3150	2000	30	512	80%	36	9

TABLE XI
USING SDM WITH *SM* ALLOWS FOR PLACING MULTIPLE EXPERIMENTAL MODELS ON THE SAME HOST, HENCE INCREASING AGGREGATE UTILIZATION OF THE FLEET. THIS RESULTS IN BETTER PERF/WATT, AND POWER SAVING FOR THE GIVEN MODEL CLASS. THE INCREASE IN THE POWER CONSUMPTION OF THE HOST IS BASED ON THE OPTANESSDS NEEDED.

Scenario	Power	Utilization	fleet power
HW-FA	1.0	0.63	1.0
HW-FAO + SDM	1.01	0.90	0.71

Table XI shows the roofline estimation for power saving with multi-tenancy enabled through leveraging slower memory. The modeling shows up to 29% power saving.

VI. RELATED WORK

SSDs have been used to extend memory in different applications. For example [22] use SSD to increase memory capacity for search applications. [28] develops a distributed training system using SSD in a hierarchical fashion to increase available memory capacity. Inference, however, is more latency sensitive compared to training, which makes it harder to leverage SSD.

[16] is among the pioneers tackling the challenges in using SSD for inference. It groups the embedding vectors of a given tables to increase the possibility that the grouped embeddings could be accessed together. This helps reduce the read amplification due to large device block size read, which is considerably bigger than embedding dimension. Our work does not follow this path due to the implication of grouping on the latency between model updates. Nonetheless, Bandana and our approach are orthogonal. In [27] authors leverage the limited compute and DRAM in the SSD controller to collect and pack requested embeddings across different pages, hence making the data transfer over PCIe more efficient. In our work, we pursue techniques to reduce the access granularity which addresses read amplification and inefficient use of the BW by reading large block size. [25] present a recommendation system which can leverage SSD for embeddings. While they mention the implication of using SSD on latency, they do not further discuss how to remedy such increase in latency, or exact implication of using SSD vs memory. [17] use scale out and shard the model across multiple servers to scale the memory capacity available to the usecase.

Non Volatile Memory (NVM) has been studied and used to enable tiered memory system. For example [29] present a tired memory system evaluating per-application memory management at user level. [30] update the OS page tracking to enable transparent tiered memory paging. None of these works consider block memory behind NVMe, which is the focus of our work.

VII. Conclusion

Rapid increase in Deep Learning Recommendation Model (DLRM) size makes it more expensive in terms of power to serve such models. Power, is particularly among the most important metrics at DC scale. Companies operating DCs are willing to pay for extra compute, but the rate of growth is limited by the rate at which the power could be provisioned. We leverage the inherit skew in BW among different embedding tables in DLRM to deploy a Software Defined tiered memory increasing memory capacity per host by leveraging Storage Class Memory. We evaluate and address a range of challenges, such as fast IO, capturing locality, trade-off of capacity vs compute and BW. We discuss the value of such technology under different deployment scenarios. We observe 20% power saving serving a large model while using a *simpler hardware* with Nand Flash, 5% power saving using another compute heavy model by *avoiding scale-out*, and projected 29% improvement in perf/watt by increasing utilization of Accelerator enabled platforms through *multi-tenancy* using Optane SSD. Such power and perf/watt optimizations are considerable given the power boundness of serving such models at DC scale.

References

[1] Gomez-Uribe, C. A. and Hunt, N. "The netflix recommender system: Algorithms, business value, and innovation.", ACM Trans. Manage. Inf. Syst., 6(4), December 2016. ISSN 2158-656X. 10.1145/2843948. URL https://doi.org/10.1145/2843948.

[2] Covington, P., Adams, J., and Sargin, E. Deep neural networks for YouTube recommendations. In *Proc. 10th ACM Conf. Recommender Systems*, pp. 191–198, 2016.

[3] Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H. Wide and deep learning for recommender systems. *arXiv:1606.07792*, 2016. URL http://arxiv.org/abs/1606.07792.

[4] Smith, B. and Linden, G. Two decades of recommender systems at amazon.com. *IEEE Internet Computing*, 21(3):12–18, May 2017. ISSN 1089-7801. 10.1109/MIC.2017.72. URL https://doi.org/10.1109/MIC.2017.72.

[5] Ma, Y., Narayanaswamy, B. M., Lin, H., and Ding, H. Temporal-contextual recommendation in real-time. KDD '20, pp. 2291–2299, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Lopez, R., Dhillon S., I., and Jordan I., M. Learning from extreme bandit feedback. In *Proc. Association for the Advancement of Artificial Intelligence*, 2021.

[7] , "Applied machine learning at facebook: A datacenter infrastructure perspective", Hazelwood, Kim and Bird, Sarah and Brooks, David and Chintala, Soumith and Diril, Utku and Dzhulgakov, Dmytro and Fawzy, Mohamed and Jia, Bill and Jia, Yangqing and Kalro, Aditya, et. al., 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 620–629, 2018,IEEE

[8] Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., Law, J., Malani, P., Malevich, A., Nadathur, S., Pino, J., Schatz, M., Sidorov, A., Sivakumar, V., Tulloch, A., Wang, X., Wu, Y., Yuen, H., Diril, U., Dzhulgakov, D., Hazelwood, K., Jia, B., Jia, Y., Qiao, L., Rao, V., Rotem, N., Yoo, S., and Smelyanskiy, M. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications, 2018.

[9] Gupta, U., Wu, C., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., Lee, H. S., Malevich, A., Mudigere, D., Smelyanskiy, M., Xiong, L., and Zhang, X. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501, 2020. 10.1109/HPCA47549.2020.00047.

[10] Zhao, Z., Hong, L., Wei, L., Chen, J., Nath, A., Andrews, S., Kumthekar, A., Sathiamoorthy, M., Yi, X., and Chi, E. Recommending what video to watch next: a multitask ranking system. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pp. 43–51, 2019.

[11] Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501. IEEE, 2020.

[12] Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., Nair, K., Gao, I., Su, B.-Y., Yang, J., and Smelyanskiy, M. Deep learning training in facebook data centers: Design of scale-up and scale-out systems, 2020.

[13] Anderson, M., Chen, B., Chen, S., Deng, S., Fix, J., Gschwind, M., Kalaiah, A., Kim, C., Lee, J., Liang, J., et al. "First-generation inference accelerator deployment at facebook." *arXiv preprint arXiv:2107.04140*, 2021.

[14] Naumov, M., Mudigere, D., Shi, H. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019. URL https://arxiv.org/abs/1906.00091.

[15] Jouppi, N. P., Yoon, D. H., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P., Ma, X., et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14. IEEE, 2021.

[16] Eisenman, A., Naumov, M., Gardner, D., Smelyanskiy, M., Pupyrev, S., Hazelwood, K., Cidon, A., and Katti, S. Bandana: Using non-volatile memory for storing deep learning models. *arXiv preprint arXiv:1811.05922*, 2018.

[17] Lui, M., Yetim, Y., Özkan, Ö., Zhao, Z., Tsai, S.-Y., Wu, C.-J., and Hempstead, M. Understanding capacity-driven scale-out neural recommendation inference. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 162–171. IEEE, 2021.

[18] Axboe, J. Efficient io with io-uring. URL https://kernel.dk/io$_{u}ring.pdf$.

[19] Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M., et al. The cachelib caching engine: Design and experiences at scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 753–768, 2020.

[20] Guan, H., Malevich, A., Yang, J., Park, J., and Yuen, H. Post-training 4-bit quantization on embedding tables. *arXiv preprint arXiv:1911.02079*, 2019.

[21] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[22] Heo, J., Lee, S. Y., Min, S., Park, Y., Jung, S. J., Ham, T. J., and Lee, J. W. Boss: Bandwidth-optimized search accelerator for storage-class memory. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 279–291. IEEE, 2021.

[23] Khudia, D., Huang, J., Basu, P., Deng, S., Liu, H., Park, J., and Smelyanskiy, M. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*, 2021.

[24] Lee, K., Rao, V., and Arnold, W. Accelerating facebook's infrastructure with application-specific hardware. URL https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/.

[25] Liu, H., Gao, Q., Li, J., Liao, X., Xiong, H., Chen, G., Wang, W., Yang, G., Zha, Z., Dong, D., et al. Jizhi: A fast and cost-effective model-as-a-service system for web-scale online inference at baidu. *arXiv preprint arXiv:2106.01674*, 2021.

[26] Smelyanskiy, M. Zion: Facebook next-generation large memory training

platform. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019. 10.1109/HOTCHIPS.2019.8875650.

[27] Wilkening, M., Gupta, U., Hsia, S., Trippel, C., Wu, C.-J., Brooks, D., and Wei, G.-Y. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 717–729, 2021.

[28] Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *arXiv preprint arXiv:2003.05622*, 2020.

[29] Raybuck, Amanda and Stamler, Tim and Zhang, Wei and Erez, Mattan and Peter, Simon, HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM, Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, pp 392–407,2021

[30] "Nimble page management for tiered memory systems", Yan, Zi and Lustig, Daniel and Nellans, David and Bhattacharjee, Abhishek, Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp 331–345, 2019

Appendix

A. *CPU cost of high IOPS*

The CPU cost of extracting IO is considerable when the memory extension has IO semantics, and could be prohibitive. Newer technologies to expose SSD space to the CPU might alleviate this problem. Ideally standardization of such solution (e.g. through CXL) would make the option more adoptable.

B. *Inter-Op Parallelism*

The SparseLengthSum operator in Caffe2 or EmbeddingBag operator in Pytorch could involve IO when the embedding tables associated with the operator are placed on *SM*. Hence it becomes important to not only enable async IO for access to embeddings for a given table, but also provide async execution of the operators in such Deep Learning platforms. Such inter-Op parallelism allows for more efficient discovery of IOs that need to be issued, and enable IO and computation overlap. Therefore, the inter-Op parallelism reduces latency per query. In a latency sensitive usecase, higher latency per query could result in under utilization of the host, and hence lower throughput. Therefor, inter-Op parallelism also improves throughput. For example we have observed 20% reduction in latency per query through inter-Op parallelism, resulting in 20% more QPS per host at the desired latency for model *M1* (Table VI).

C. *Model update*

Models are refreshed frequently, with a desire for more frequent updates (e.g updates every few minutes), to keep the models as up to date as possible. However, given the large model size, updates could be separated to updating dense parameters and updating embedding tables, which could happen with different frequency (embedding updates being less frequent). Incremental update is another path to increasing update frequency for the model. Given the need to save the embeddings into *SM*, the time it takes to update the model will be increased. Hence, incremental updates are considered to minimize the amount of data that needs to be updated.

As new weights stream in, the host could be offline or still online serving traffic. The former prevent mixture of read and write BW which would considerably impact performance of Nand flash. The latter would allow for better utilization of the resources. Given the software defined cache, we can update the cache first and allow for dirty write backs to update the SM.

Section III discusses how endurance could limit model update frequency.

D. *Warmup*

Cold *SM* cache in *FM* could impact the performance right after a full model update. We observe that caches warmup in order of a few minutes. But the perf impact need to be compensated by over-provisioning the capacity. For example if 1) r=10% of the hosts serving a model are being updated in a given time (rolling update), and 2) the performance during warmup is p=50% of steady state, 3) update every t=30 minutes, 4) warmup in w=5 minutes, we need $(r*w)/(p*t) = (10\% * 30)/(50\% * 5) = 1.2\%$ more capacity to offset the slowdown.

E. *de-quantization*

Quantization is a widely use technique [21], and for embedding tables it helps to reduce model size as well as memory BW. Given the higher *SM* capacity, we can dequantize the embedding table at loading time into the *SM*, and save the dequantization at run time. It will consume more memory space in *SM*, which typically is not memory capacity bound. It will not add to BW consumption of *SM* in some of the systems due to higher access granularity to *SM* (e.g. 72B int8 qunatized embedding with 64 embedding elements and 8 byte quantization parameters per row expand to 256B, still smaller than access granularity for Nand Flash). However, dequantization at loading leads to less efficient use of *FM* space for cache. This is because less number of embedding rows could be stored in a given cache size when each row enlarges due to dequantization. We observe that while under very CPU bound usecases dequantization could help, but for most of the usecases the impact on cache is dominant and does not lead to benefit. Pooled embedding cache (Section IV-D) provides a more fine tuned solution which can leverage dequantized (and pooled) embeddings in a more selective manner.