High-performance, Distributed Training of Large-scale Deep Learning Recommendation Models

Dheevatsa Mudigere[†], Yuchen Hao[‡], Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie (Amy) Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, Vijay Rao

Facebook, Menlo Park, USA

ABSTRACT

Deep learning recommendation models (DLRMs) are used across many business-critical services at Facebookand are the single largest AI application in terms of infrastructure demand in its data-centers. In this paper we discuss the SW/HW co-designed solution for high-performance distributed training of large-scale DLRMs. We introduce a high-performance scalable software stack based on PyTorch and pair it with the new evolution of *Zion* platform, namely *ZionEX*. We demonstrate the capability to train very large DLRMs with up to *12 Trillion parameters* and show that we can attain 40× speedup in terms of time to solution over previous systems. We achieve this by (i) designing the *ZionEX* platform with dedicated scale-out network, provisioned with high bandwidth, optimal topology and efficient transport (ii) implementing an optimized PyTorch-based training stack supporting both model and data parallelism (iii) developing sharding algorithms capable of hierarchical partitioning of the embedding tables along row, column dimensions and load balancing them across multiple workers; (iv) adding high-performance core operators while retaining flexibility to support optimizers with fully deterministic updates (v) leveraging reduced precision communications, multi-level memory hierarchy (HBM+DDR+SSD) and pipelining. Furthermore, we develop and briefly comment on distributed data ingestion and other supporting services that are required for the robust and efficient end-to-end training in production environments.

1 INTRODUCTION

Recommendation models are ubiquitously used by many online companies, including Amazon for selecting items in its catalog [32, 34, 52], Netflix for showing movie options [10, 25] and Google for displaying personalized advertisements [4, 6, 15]. They have also been adopted by standard benchmarking organizations, such as MLCommons (MLPerf) [35, 46]. At Facebookwe recommendation models extensively for ranking and click through rate (CTR) prediction, including News Feed and search services [12, 13, 38, 41], and is the single largest AI application in terms of infrastructure demand in its data-centers.

[†]dheevatsa@fb.com, [‡]haoyc@fb.com
[*]https://openai.com/blog/ai-and-compute/
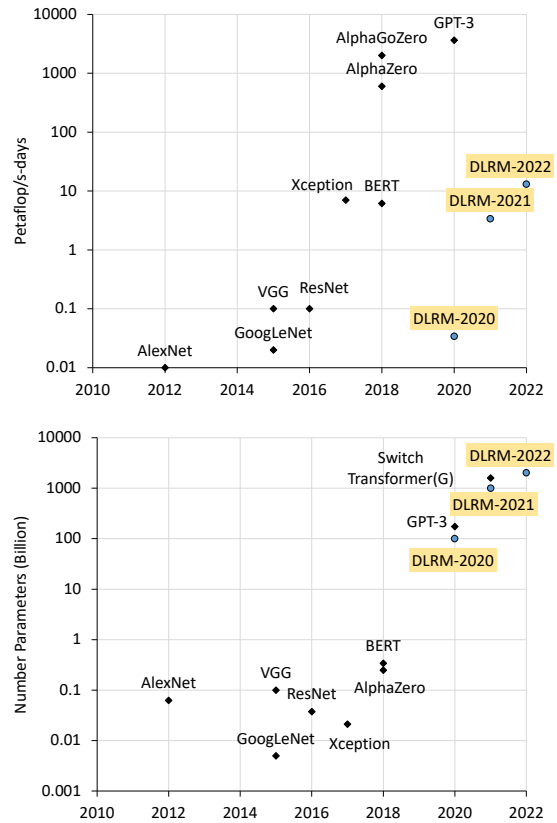
Figure 1: Comparison of deep learning models in terms of total amount of compute, in petaflop/s-days[*] (top) and model capacity (bottom)

Deep learning recommendation models (DLRMs) that are often composed of sets of fully connected layers (MLPs) and embedding tables [39], tend to be very large compared to their computer vision [5, 14, 26, 50, 53], natural language processing [2, 7, 55] and reinforcement learning counterparts [48, 49], with models having trillions of parameters being deployed in practice, see Fig. 1.

Table 1: Sample DLRM time to train latency resources demand

Total compute	1+ PF/s
Total memory capacity	1+ TB
Total memory BW	100+ TB/s
Network injection BW per worker	100+ GB/s
Network bisection BW	1+ TB/s

Furthermore, these models exhibit a more balanced workload profile, exercising memory capacity as well as memory and network bandwidths in addition to compute. Owning to these unique characteristics, efficient training of these models at scale is a significant challenge, necessitating a flexible combination of different forms of pipeline, model and data parallelism.

DLRMs typically have two modes of training - offline and online, each with varying requirements. The offline training can be viewed more as a pre-training, where a candidate model is trained on sufficiently large historical data, and expected to generalize when deployed to current/unseen samples. Once deployed, DLRMs continue to be trained online using the data that it has already served. Offline-training is throughput limited, fitting into the more conventional *"train as fast as possible on as much data as possible"* paradigm. Whereas online training is more latency sensitive, with the frequency of re-training and update being an important factor. For online training, the throughput requirement is lower hence it might be desired to use proportionally lower resources. This for instance creates an unique requirement of training very large models at smaller scales capable of tolerating lower throughput.

In this paper we focus on offline training, since it has more demanding training throughput needs - up to millions of samples (queries) per second resulting from processing through tens of petabytes of training data within a reasonable time. This drives the training platform requirements, as summarized in Table 1. Specifically, this paper makes the following contributions:

- Co-design state-of-art solution for end-to-end training of large scale DLRMs deployed for practical applications, including both hardware *ZionEX* training platform and co-designed high-performance scalable training software stack implemented in PyTorch.
- Enable flexible distributed training in PyTorch, combining model and data parallelism. In particular, supporting optimized sharding method with optimal partitioning and placement of model parameters among any dimension.
- Implement high performance embedding operators and the complete training pipeline including data ingestion and other supporting services.
- Demonstrate more than an order of magnitude speedup in end-to-end training throughput and training platform capability for production use cases. In particular, we show more than a 40× reduction in total training time over previously deployed distributed training solutions and efficient training of a *12 Trillion parameter* model on 128 GPUs.

2 BACKGROUND

Traditionally a disaggregated parameter-server (PS) based distributed CPU training system has been used for training DLRMs in a production setting [13, 38]. Specifically, the dense parameters from
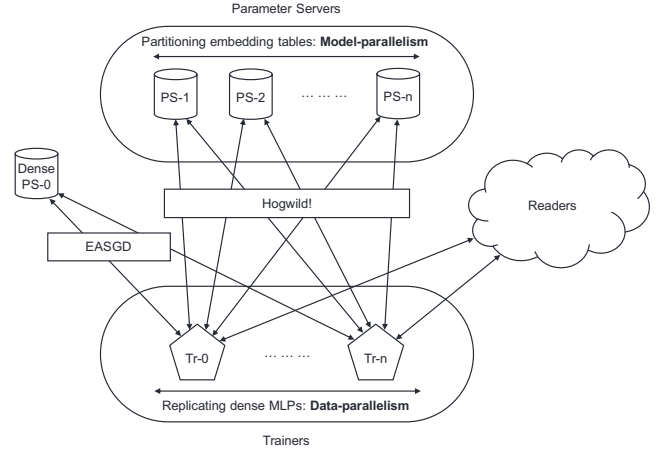
Figure 2: Disaggregated parameter-server based system

the MLP modules are duplicated between the trainers to exploit data-parallelism. Their weights are synchronized with a centralized dense parameter server using Elastic Averaging method SGD [61, 64]. On the other hand, The parameters from the embedding tables are partitioned and placed on multiple PS to exploit model-parallelism, since the size of embedding parameters simply prevents model replication. Then, to achieve higher performance the parallel updates of the embedding parameters are done with Hogwild! [45]. Also, the readers are deployed on a separate tier of machines to feed training batches to the trainers as illustrated in Fig. 2.

Such PS-based system is well suited for DLRMs allowing scaling different components separately and achieving a balanced resource utilization when training different models with different trainer, parameter server and reader configurations. Moreover, resources in the system are largely fungible, making it low-cost for datacenter operations. Specially designed training hardware such as *Zion* (Fig.3) has been previously proposed for training DLRMs [51].

However, the need for supporting DLRMs with trillions of parameters and therefore terabytes in size poses a serious challenge to the scalability of this approach. Necessitating steeply growing number of trainers and parameter-servers to meet the ever growing training requirements. This quickly becomes intractable, degrading model accuracy with staleness due to increased asynchronous updates across a very large number of workers. To tackle these issues, we build a high-performance synchronous training solution for large DLRMs, decoupling distributed scaling from statistical quality.
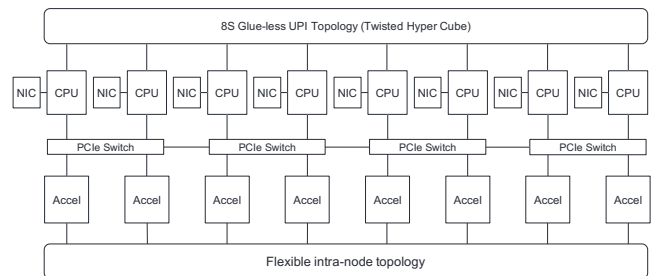
Figure 3: The system architecture of *Zion* platform [51]

The efficient design of the synchronous training system leads us to use a novel combination of model-parallelism for memory intensive embeddings tables, data-parallelism for compute intensive MLPs and pipelining across the different components. This hybrid parallelism requires `AlltoAll` communication for the embedding lookup results [38, 39], as well as embedding table input re-distribution if the inputs are streamed from database in batches, which is often the case. Unlike the `AllReduce` communication for the gradient synchronization, which can be overlapped, these `AlltoAll` communications are on the critical path due to the data dependency, stressing the performance of the interconnect and communication primitives. Furthermore DLRMs are typically trained on *very large* amounts of data, which corresponds to mostly unstructured and unlabeled interactions from a wide variety of applications. Typical data-set size are in the range of several petabytes, necessitating the use of common, distributed network store, such as the Tectonic filesystem [40]. For training, this data would need to be streamed-in, putting additional stress on the host network and host-to-device bandwidths.

3 HIGH-PERFORMANCE SCALABLE TRAINING

Fig. 4 outlines the scalable synchronous distributed training solution for DLRMs. This also follows the same strategy of model-parallelism for the embedding tables and data-parallelism for the MLPs. However, this employs a decentralized approach unlike the parameter-server based approach used for asynchronous training described in Section 2. Here each worker performs computations on its resident portion of the model and sub-batch of samples.This avoids potential network bottlenecks compared to the centralized parameter-server based approaches [19, 20].

For the data-parallel MLPs, an `AllReduce` communication is performed in the backward pass to average the gradients computed on the multiple nodes for the different sub-batches of data. Whereas the model-parallel embedding tables require an `AlltoAll` communication both in the forward pass for pooled embeddings and in the backward pass for the updates. Generally the aggregated transfer size or `AllReduce` is up to several GBs, since the parameters for all the MLP layer are replicated and are reduced, hence requiring higher interconnect bandwidth than `AlltoAll`. Whereas the `AlltoAll` sizes are typically smaller with around 100s MB of total transfer size and individual messages sizes of up to 100s of KB, making it more sensitive to interconnect latency.

The salient characteristics are the following which span across software implementation and training platform hardware.

- Distributed non-parameter server based approach, with each worker processing a subset of the model for the embeddings and a replica of the MLP parameters.
- Fully synchronous update of both replicated data-parallel MLPs and model-parallel embeddings sharded across workers with flexible partitioning.
- Scalable communication layer with sufficiently provisioned network (high bandwidth / low latency), along with the optimized communication backend supporting overlap, prioritization in addition to high performance collectives to ensure efficient scaling.
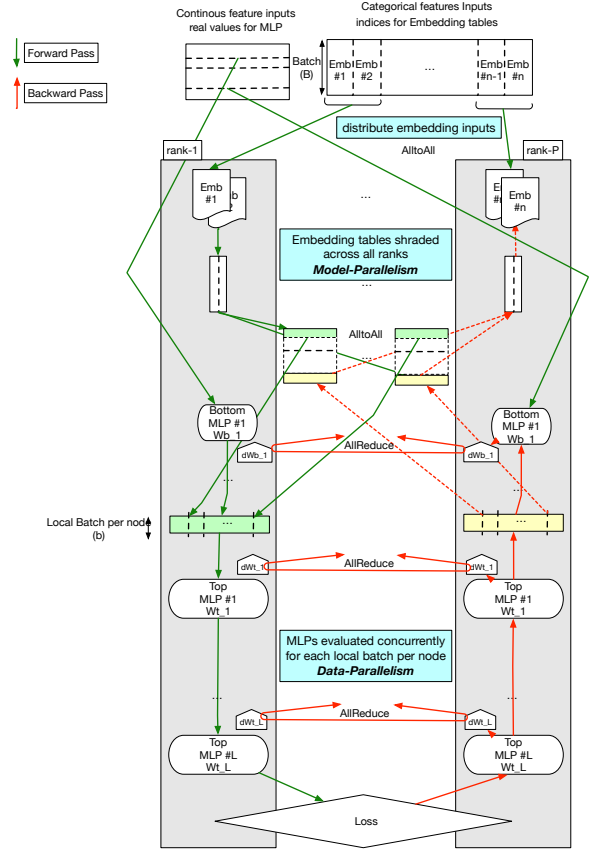

Figure 4: High-performance scalable training of DLRMs

- Optimized end-to-end training pipeline, the data-ingestion optimizations to support model distributed model + data parallel training and higher training throughput.

3.0.1 Partitioning. Similar to parameter-server based approaches, embedding tables are partitioned and placed on multiple workers to exploit model parallelism due to the sheer size. As studied in [1], embedding tables in DLRMs exhibit diverse shapes and dynamic costs. It is critical to achieve a balanced partitioning between the workers since it directly impacts the performance of embedding look-ups and collective communications.

Specifically, for an embedding table of shape $\{H, D\}$ with an average pooling size L, the cost of distributing the pooling input is proportional to L, the cost of embedding pooling is roughly proportional to $L \times D$ (with H having impact on caching effects from data reuse), and the cost of communicating the pooled embedding is proportional to D. While all three operations with corresponding hardware costs need to be considered in a cost function, distributing embedding tables to N workers maps to the well-known *partitioning problem*.

In practice, we found that algorithmic approaches alone could not solve all load balance issues when the model has too few embedding tables or tables have varying characteristics. To handle these scenarios and improve load balancing in general, we enable sharding an embedding table along all possible dimensions and
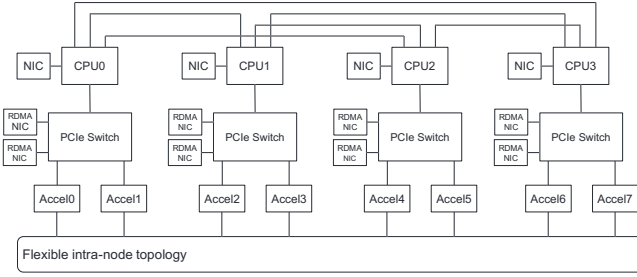
Figure 5: The system architecture of new *ZionEX* platform

Figure 6: The overall training system

supporting corresponding variants in collective communications, which will be detailed in Section 4.2.

3.0.2 Data I/O. The adoption of fully synchronous training and accelerators has a number of implications on data I/O. First, the host to device transfer should be non-blocking and fast enough not to limit the overall training throughput. Ideally overlapping the input transfer with training using double buffering or pipelining. Second, even though mapping input data distribution to collective communications between the trainers is potentially faster, this introduces additional challenges for the input and output data layout of the collective communication. Initial experiments have shown us that these could add significant latency to the critical path. We will illustrate how we overcome these practical challenges in Section 4.4.

3.1 Hardware and Systems

For training DLRMs, previously systems like *Zion* [37, 51] have been proposed. With each *Zion* node having 8-socket CPUs with 1.5 TB memory, 8 GPUs, and 8 network interface cards (NICs). It offers a much more powerful *super node* supporting CPU-GPU co-training, by (1) offloading the compute heavy layers of the model, such as MLPs, onto the GPUs; (2) leveraging CPUs for large embedding tables on the relatively cheaper DRAM instead of HBM, so that TB-scale models can be accommodated on a single node. However, such type of hybrid training introduces a number of challenges for the software design and performance. For example, it's critical to balance the work on CPUs and GPUs ensuring maximum overlap, which requires elaborate pipelining and fine-grained model partitioning with an accurate cost model. Hybrid co-training also introduces non-trivial overheads such as increased *CPU ↔ GPU* traffic, NUMA overheads from large number of CPUs and cross-socket communication. In addition, a critical missing piece of the original *Zion* design is that the NIC is attached to the CPU, instead of the directly to the GPUs. Resulting in inter-node communication (gradient synchronization etc.) going through the host NIC, with CPU intervention and over the shared network constrained to use more data center-friendly topologies/protocols (TCP/IP) which are sub-optimal for distributed training. Although each *Zion* node is equipped with 8x 100Gbps NIC bandwidth, in reality we have found it very difficult to scale out to multiple nodes due to networking overheads. With today's increasing demand on modeling size of DLRM models, *Zion* is not able to scale well and fully utilize the powerful hardware resources.

To address these shortcomings, we introduce the *ZionEX* platform which we have designed to be more scalable with improved
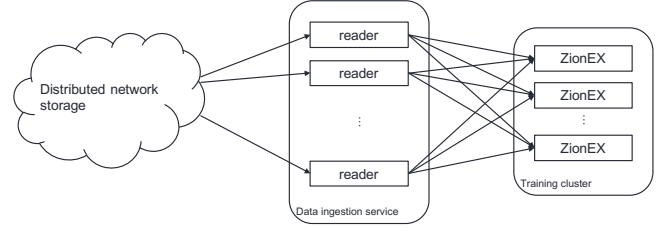
network capabilities. While retaining all of the goodness of *Zion* such as the OAM form factor, modular design [37, 51] flexible intra-node accelerator fabric [62]. Fig. 5 shows the overall system architecture. Each *ZionEX* node has 4-socket CPUs and 8 GPUs, with 4 frontend NICs connected to the host CPUs and a dedicated *RDMA over Converged Ethernet* (RoCE) NIC for each of the GPUs connected via PCIe switches to allow for a dedicated inter-node connectivity (isolated from common data-center network) and more importantly more efficient RDMA/GPUDirect protocols [38]. These *ZionEX* nodes can be connected with a dedicated backend network to form a cluster for distributed scalable training. The extensible design of *ZionEX* allows for scaling the backend network to interconnect many thousands of nodes, forming a data-center scale AI training cluster.

As a scale-out solution, we offload the entire DLRM to GPUs, fully leveraging the abundant parallelism and high memory bandwidth to accelerate MLPs and embedding operations alike. In exchanging activations and synchronizing gradients, each GPU can communicate with GPUs on a different node through the dedicated low-latency high-bandwidth RoCE NIC, without involving the host CPU. While the data ingestion goes through the regular frontend network and PCIe, without interfering with activations or gradients. The host CPUs are only used to setup input batches and training operators.

Fig. 6 shows the overall training platform, along with the disaggregated data-ingestion service that supports streaming input data from a network store such as Tectonic [40] and perform lightweight data pre-processing operations in a distributed fashion. So that the data-ingestion is not a bottleneck for the end-to-end training and to ensure sufficient throughput in feeding *ZionEX* trainers.

4 IMPLEMENTATION IN PYTORCH

In this section, we detail the implementation of high-performance scalable training for DLRMs described above. We built a high-performance training software stack for DLRMs using PyTorch [42], with efficient CUDA implementation for most deep learning operators via the `ATen` library, and automatic handling of parameter replication and gradient synchronization with overlapped back-propagation and `AllReduce` via the PyTorch `DistributedDataParallel` library [29]. In addition, we have enabled the following components in order to efficiently train DLRMs:

- High-performance embedding operators.
- A flexible sharding module to enable balanced model-parallel training.
- Pipelining across training iterations to increase GPU utilization.

- Data-ingestion optimizations to keep compute units highly utilized.
- Communication primitives and network tuning to improve collective operation performance.

We detail the design of the above components below.

4.1 Embedding ops

Two of the most important operators for DLRMs are fully-connected operators (FC) and embedding operators. While FC operators are highly optimized with the GEMM routine in cuBLAS library, we developed the efficient CUDA implementation of embedding operators on GPUs. We have open sourced these optimized implementation as part of the *PyTorch FBGEMM high-performance operator library*[†].

4.1.1 Kernel fusion. Typical DLRM models can have up to $\approx 1000s$ of categorical features with each feature corresponding to an embedding table. Instead of applying a separate embedding lookup for each embedding table, we fuse multiple embedding lookups into a single CUDA kernel (Figure 7), which improves the parallelism and bandwidth utilization, and reduces the overhead to launch multiple kernels on GPUs. Besides batching multiple embedding tables together, we also fuse the backward pass with the sparse optimizer, which saves the additional memory for the gradients (by a factor of pooling size L) and reduces the memory traffic. The overall performance speedup of the fused kernel is up to 7× compared to the native nn.EmbeddingBag implementation at the operator level.

4.1.2 Embedding updates with exact sparse optimizers. Large batch synchronous training for embedding parameters required optimizing for parallelized sparse updates to ensure high HBM bandwidth utilization on the GPUs. This introduces challenges such as avoiding potential race-condition across different updates and handling non-linearity in the advanced optimizers like AdaGrad [8], LAMB [60], and Adam [23]. The exact sparse optimizers are implemented to guarantee the determinism and accuracy with a minimal performance overhead. This requires essentially transposing of the sparse update matrix, done via sorting row indices in each mini-batch and merging gradient updates for the same rows into one update. Assume one input in the mini-batch accesses rows 1 and 2 with gradient g_1, another input accesses rows 2 and 3 with gradient g_2. We sort by the rows, accumulate gradients (e.g., $g_1 + g_2$ for row 2), and then apply the accumulated gradients. Deterministic updates coupled with synchronous training, we are able achieve bit-wise reproducibility across runs with different number of workers, which helps greatly for debugging and enabling new models, algorithms at scale.

4.1.3 Memory hierarchy. For the DLRM model with up to trillions of parameters, the embedding tables are too large to entirely fit on a single GPU. We leverage multiple levels of memory hierarchy of the *ZionEX* platform, including HBM, DRAM and SSDs to ensure sufficient memory capacity for the models, with the faster memory serving as a software cache of the subsequent layer. Hierarchical memory training is also useful for applications such as online training, which warrants using fewer nodes for training the same
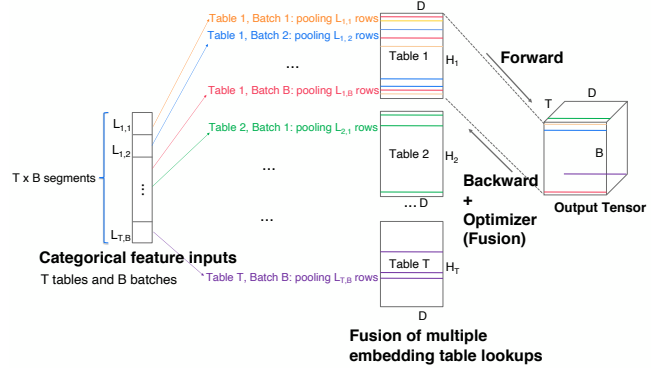

Figure 7: Kernel fusion in the embedding ops. Note that the pooling size L for each table and each batch can be different.

model (as described above). One option is to use the mechanism like unified memory (UVM)[‡]. However, UVM replaces and evicts the unused parameters in large pages instead of finer granularity like embedding rows. We instead implement our own customized 32-way set-associative software cache [57] using least recently used (LRU) or least frequently used (LFU) cache replacement policies, where the associativity matches the warp size of GPUs. This enables fine grain control of caching and replacement, allowing it to be tuned for target model characteristics. Note that UVM is bounded by PCIe bandwidth, while the software cache can bridge the gap for the bandwidth between PCIe and HBM (50 × difference). It generally brings 15% end-to-end performance improvements utilizing the software cache compared to UVM for the typical DLRM workload (related to cache size, input data reuse pattern, pooling size, and cache hit rates).

4.1.4 Memory saving techniques. Due to the large embedding table size, we also applied different embedding compression techniques like row-wise sparse AdaGrad; low/mixed-precision training using a high-precision cache backed by low precision embedding tables [57]; and advanced factorization techniques like Tensor Train compression [59].

The row-wise sparse AdaGrad was first introduced in [11], and then further elaborated in [56]. Each element of the moment estimation is applied to the entire embedding row. For each row it is a single scaling factor and this factor is updated by adding the average squared sum of gradients across the row. For example, $m_i' = m_i + (1/D)(\sum_{j=1}^{D} g_{ij}^2)$ where i is the row index. In this way, we keep the momentum as a 1D tensor with H elements instead of $H \times D$ 2D tensor, which saves the total memory by up to 50%.

4.2 Hybrid Embedding Table Sharding

As stated in 3.0.1, achieving load balance and minimizing communication cost are crucial for performance. We combine the following embedding table sharding schemes with placement algorithms to balance the load between workers.

[†]The embedding op implementations are open sourced as part of FBGEMM_GPU: https://github.com/pytorch/FBGEMM/tree/master/fbgemm_gpu

[‡]https://developer.nvidia.com/blog/unified-memory-in-cuda-6/
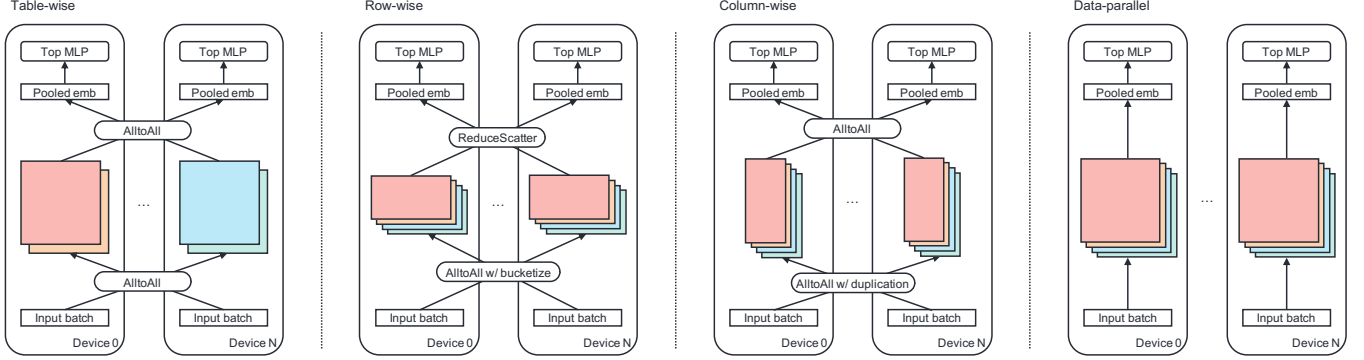
Figure 8: Four different embedding table sharding schemes with different implications on the communication cost, load balancing and memory requirement. Bottom MLP is omitted in this figure for simplicity of illustration.

4.2.1 Table-wise sharding. The most straightforward sharding scheme is to partition the multiple embedding tables. Since table-wise sharding does not further split tables, this scheme requires no additional handling of embedding table input indices or pooled embedding results, leading to the optimal communication efficiency. However, this is unable to handle massive embedding tables that exceed the memory capacity of a single worker, and the achieved load balance is often limited, due to the skew in table sizes.

4.2.2 Row-wise sharding. Row-wise sharding partitions expensive tables by rows and placing different table shards on different workers. Since the embedding table inputs index tables by rows, they need to be *bucketized* based on the row-wise sharding decision and distributed to the respective workers. Moreover, partial results on multiple trainers need to be reduced and then scattered to all trainers for interaction, which maps to a ReduceScatter communication pattern in the forward pass. This scheme handles large tables well and leads to better balance. However, its communication cost scales linearly with the number of trainers.

4.2.3 Column-wise sharding. Column-wise sharding partitions the embedding tables along the embedding dimensions and treat the table with smaller embedding dimensions as the regular tables. This scheme requires duplication of the input indices for the partitioned tables. Comparing with table-wise partition, it preserves the same flow and communication pattern (AlltoAll). The advantage is that this allows for more fine-grained partitioning, especially for large tables. However, it only works well only with larger embedding dimensions and increases the payload for in the input indices AlltoAll. Furthermore, since the rows of column-wise sharded tables are split across different trainers, using an independent row-wise update for these tables introduces additional parameters one for each shard of the row instead of just a single value for the entire row with using optimizers such as row-wise Adagrad (Sec 4.1.4).

4.2.4 Data-parallel sharding. DLRM models tend to have a wide range of table sizes, while table-, row-, column- wise sharding are efficient for relatively larger tables which are prohibitive to replicate. For the smaller tables, data-parallelism with replication is a viable and more optimal since this does not have the overhead of communication in the forward pass. For such cases, we also enable support for data-parallel sharding. In this scheme, embedding tables are treated as dense parameters and thus are replicated to all workers. With this, AlltoAll is no longer needed for the pooled embeddings of data-parallel embedding tables. Instead, AllReduce is required to synchronize between all the replicas. Therefore, this depends on the trade-off between the cost of AlltoAll of the pooled embeddings versus the cost of AllReduce on the entire table. In general, small embedding tables with fewer rows are good candidates for data-parallel sharding. Input indices for theses tables are passed through as data-parallel inputs and no longer require re-distribution.

4.2.5 Partitioning algorithms. Sharding schemes can be applied at each embedding table level for maximum flexibility. Practitioners can mix-and-match the above primitives to determine the best strategy to shard a group of embedding tables. Additionally, an embedding table can be sharded in a recursive manner at different levels of hardware hierarchy to further improve load balance and hardware efficiency. For example, the table-wise then row-wise scheme can choose to first assign a set of tables to a particular node, and within that node the tables are sharded row-wise. This family of hierarchical sharding schemes improve hardware locality by fully exploiting the fast GPU interconnect and reduce inter-node communication messages.

With a cost function defined for each of the above sharding schemes, placement algorithms can be explored to minimize the cost differences between workers. We implement and evaluate two polynomial time heuristics as a proof of concept. The first one is the simple greedy heuristic. We first sort the costs in descending order and allocate the largest shards first, one per worker. Then, iterate through the remaining shards and assign the top cost to the node with the smallest sum of costs. The second heuristic is the largest differencing method (LDM), also called the Karmarker–Karp algorithm [22]. The main idea is to take the two largest numbers from the input and replace them by their difference. It directly reduces the difference of sums and usually works better than the greedy heuristic.

4.3 Pipelining

Although using GPUs as the single compute resource in the system does not provide pipelining opportunities within model evaluation, we improve GPU utilization by pipelining inter-batch data movement and overlapping communication with computation.

One straightforward idea is when batch i is being evaluated, the same GPUs can start receiving and distributing batch $i + 1$ using a separate stream. To minimize the interference, we overlap the input AlltoAll of batch $i + 1$ with the forward propagation of top MLP of batch i where no communication is involved. In addition, we overlap the pooled embedding AlltoAll with the forward propagation of bottom MLP to hide latency. In general, pipelining and overlapping is easy to achieve with PyTorch's imperative style programming model.

4.4 Data ingestion

Data ingestion is a key component to ensure end-to-end training performance especially for DLRMs which typically process through order(s) of magnitude larger amount of data than other typical deep learning models. We found data ingestion, if left unoptimized, can incur significant latency and introduce overheads for pipelining.

Originally designed for a distributed asynchronous CPU setup, our readers and data pre-processing module stores the offsets and indices [§] of each sparse feature in separate tensors per embedding table. As a result, a DLRM with hundreds of embedding tables can easily get a thousand input tensors per iteration, which translates into significant overheads from $CPU \leftrightarrow GPU$ transfers and was one of the key bottlenecks for the previous Zion platform as detailed in Sec. 2.

To overcome this practical challenge, we co-designed the data pre-processing module to use a combined format where lengths rather than offsets are used and inputs to different embedding tables are simply concatenated. The benefits of using the combined format are two-fold: (1) it optimizes CPU-GPU transfer by consolidating small transfers; (2) it can be directly consumed by the embedding kernel without additional layout transformations. We further optimized input data transfer by using pinned memory to avoid the extra copy.

With the combined format, we developed a module to efficiently distribute embedding table inputs based on the sharding strategy. In the case of table-wise sharding (shown in Fig. 8), an AlltoAll is needed to distribute the global batch for local tables to each worker. Since the size of indices is dependent on the values of the lengths, the communication is actually implemented as an AlltoAll for lengths followed by an AlltoAll for indices. In a setup with W workers, T local tables and B local batch size, this gives us indices in the order of (W, T, B), which needs to be further permuted to (T, W, B) for embedding kernel consumption. We have developed custom GPU kernels for permute, bucketize and replicate to achieve maximum throughput on embedding input indices distribution for table-wise, row-wise and column-wise sharding schemes. Checkpointing the model has similar challenges, requiring to be sufficiently frequency be able to write-out such larger model whilst not becoming an overhead for training, as outlined in this recent paper [9].

[§] Please refer to the interface of nn.EmbeddingBag https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html

4.5 Comms/NW

High-performance collective communication is key to enabling and scaling DLRM. PyTorch provides the Process Group (PG) interface for collectives - an abstract platform / collectives library agnostic API. DLRM uses this API directly (for Alltoall) or indirectly via DDP (for Allreduce) [29]. We use the *NVIDIA's Collective Communication Library* (NCCL) as our primary collective communication library since it efficiently uses RDMA and NVLINK for best performance. We extended PyTorch NCCL PG implementation to support Alltoall/Alltoallv collectives using NCCL Send/Recv primitives (requires NCCL 2.7.3 or later).

5 RESULTS

We provide results for end-to-end training of prod models, operator-wise performance breakdown and a performance roof line to establish the upper bound of achievable performance.

5.1 Performance modeling and benchmarking

In order to identify performance gaps, to see how far we are from fully utilizing the platform capabilities - we establish the upper bound for achievable performance using an analytical roofline model. DLRMs can be broken down into the following major components - 1) bottom MLP; 2) embedding lookup and update; 3) AlltoAll communication of the model-parallel pooled embeddings; 4) interaction and Top MLP; 5) AllReduce communication for the data-parallel MLP gradient synchronization. The execution depeendcy between there differetn components are outlined in Fig.9. As discussed above, individually each of these have different characteristics and latency/performance for each component is dependent on different parts of the system, for instance the embedding ops performance depends on the achievable HBM bandwidth, whereas the MLP performance is bounded by achievable compute flops. Even between the two collective communication primitives - AllReduce performance depends on both the scale-out and scale-up bandwidths, whereas the AlltoAll performance primarily depends
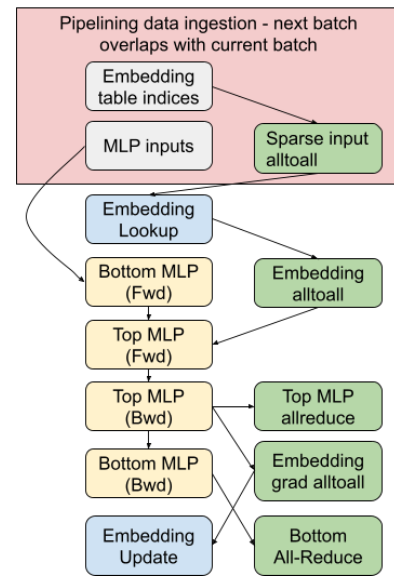

Figure 9: DLRM dependency graph

on the scale-out bandwidth. With estimates for latencies for these individual components, the overall per-iteration latency can be estimated as shown in Eq. 1

$$T_{fwd} = \max[BotMLP_{fwd}, (Embedding\_lookup + alltoall_{fwd})]$$
$$+ Interaction_{fwd} + TopMLP_{fwd}$$

$$T_{bwd} = \max[TopMLP_{bwd} + Interaction_{bwd} +$$
$$\max\{alltoall_{bwd} + Embedding\_update, BotMLP_{bwd}\},$$
$$(TopMLP\_Allreduce + BotMLP\_Allreduce)]$$

$$T_{total} = T_{fwd} + Tbwd \tag{1}$$

To better estimate the performance and latencies for each of these components, we use operator-level benchmarks which allow evaluation target operator shapes/sizes on candidate HW platforms. We benchmark[¶] the 1) embedding operators, 2) typical MLP sizes, 3) communication primitives. With these benchmarks we are able to establish the max achievable HBM bandwidth to be 850 GB/s for V100 and 1300 GB/s on A100 GPUs, and for the MLP sizes of interest, achievable compute efficiencies to be up to 78.6% (V100) and 70.5%. (A100). Furthermore, we achieve 7GB/s for 256MB `AlltoAll` and 60GB/s for 256MB `AllReduce`. Since, `AllReduce` utilizes both scale-out and NVLINKs and hence able to achieve higher bandwidth. We provide detailed benchmarking results and configuration used in appendix-A.

5.2 Experimental setup

Since the *ZionEX* cluster is not yet fully operationalwe report results on a prototype cluster using off-the-shelf NVIDIA HGX-2 based systems[‖]. Specifically, each node hosts dual-socket CPUs, 8 V100 GPUs with fully-connected using NvSwitch, 2 front-end host NICs, and 8 back-end RoCE NICs to allow direct RDMA between GPUs in separate nodes.

Table 2 summarizes the per node system capabilities. Additionally we include benchmark performance on a *ZionEX* node with A100 GPUs, since the full *ZionEX* data-center cluster is still under provisioning and deployment.

5.3 End-to-end training

Building on the operator-level benchmarking, in this section we discuss training performance for production DLRM models (Table 5) - end-to-end training throughput, scaling performance and bottlenecks.

[¶]https://github.com/facebookresearch/param
[‖]https://images.nvidia.com/content/pdf/hgx2-datasheet.pdf

Table 2: Per node system configuration for the prototype hardware

Compute (TFLOPS)	120 (FP32)/ 1000 (FP16)
HBM	256 GB, 7.2 TB/s
DDR	1.5 TB, 200 GB/s
Scale-up bandwidth	1.2 TB/s (uni-directional)
Scale-out bandwidth	800 Gbps (uni-directional)
Host NW	2 × 100 Gbps

Table 3: Target models configuration

Model	A1	A2	A3	F1
Num parameters	95B	793B	845B	12T
MFLOPS per sample	89	638	784	5
Num of emb tables	≈ 100s	≈ 1000s	≈ 1000s	≈ 10s
Emb table dim (range [min, max], avg)	[4, 192] avg: 68	[4, 384] avg: 93	[4, 960] avg: 231	[256, 256] avg: 256
Avg pooling size	27	15	17	20
Num MLP layers	26	20	26	7
Avg MLP size	914	3375	3210	490

We report results for four production DLRM models - A1, A2, A3 and F1. Table 5 lists high-level characteristics of these candidate models. Models A1 is created with moderate FLOPS per sample and overall size that can be trained with reasonable efficiency using a distributed CPU platform (previous-generation). Models A2 extends A1, targeting complex DLRMs that stress the compute capability with significantly higher FLOPS per sample, the memory bandwidth with higher embedding lookups, and the communication bandwidth to transfer a large number of embeddings. Models A3 take a step further in having wider embedding dimensions and MLP shapes. While models A2 and A3 also challenge load balancing with many embedding tables in diverse shapes, model F1 presents a different practical challenge where despite having low FLOPS per sample and small number of embedding tables, it has a single massive table that cannot fit in the memory of a single device. These target models are trained on up to 16 nodes (128 GPUs) of the prototype hardware. The model quality is evaluated in normalized entropy [16] and the performance is measured in queries per second (QPS).

Firstly, we use model A1 to demonstrate the training quality since this can also be trained on the distributed CPU platform. As seen from Figure 10, despite using significantly larger batch size (64K vs. ~150), synchronous large batch training on the prototype platform is able to provide on-par or better model quality (both using tuned hyperparameters).
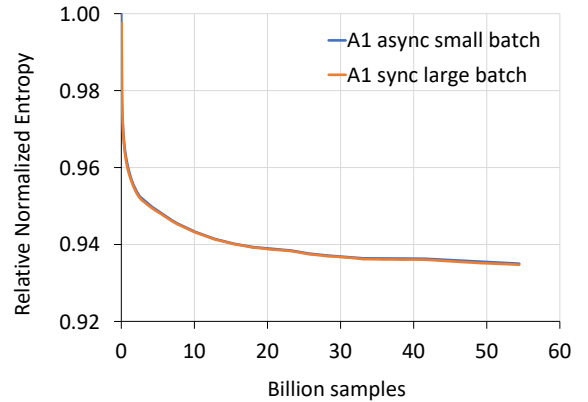
Figure 10: Training quality comparison between asynchronous small batch on a distributed CPU platform and synchronous large batch on the proposed platform, measured in relative normalized entropy[16].

Table 4: Achieved training throughput

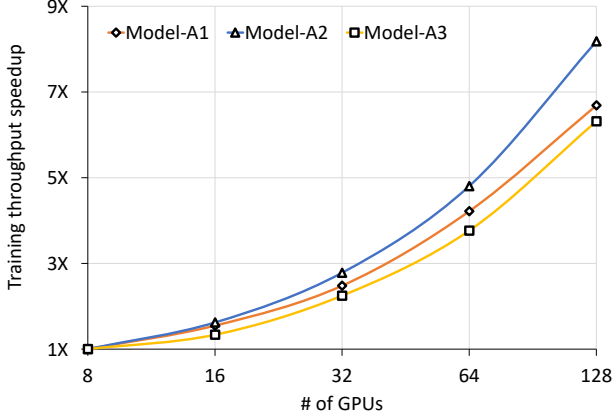| Model	A1		A2	A3	F1
	16 GPUs	128 GPUs	128 GPUs	128 GPUs	128 GPUs
QPS	273K	1047K	622K	360K	970K

Figure 11: Training throughput scaling for models A1, A2 and A3, relative to 8 GPUs (1 node).

Table 4 summarizes the achieved training throughput in QPS on the target models. For model A1, we are able to achieve 273KQPS using 2 nodes with 16 GPUs, a 3X speedup compared to our previous generation distributed CPU asynchronous training platform using ~16 parameter servers and ~16 trainers. The throughput increases to 1047KQPS using 16 nodes with 128 GPUs with reasonable scaling efficiency (more on scalability in section 5.3.1). With $\approx 10X$ larger and more complex models A2 and A3, we are able to achieve 622KQPS and 360KQPS. We will discuss in more details the results and challenges for training model F1 in section 5.3.3.

5.3.1 Scaling performance. Figure 11 shows the normalized training throughput with model A1, A2 and A3 using 1 to 16 nodes, with keeping the per-GPU batch size the same. While the per-GPU batch size remains the same, the numbers of embedding tables per GPU reduces with scaling. However, since these tables are model-parallel each GPU processes the entire global minibatch for each of its local tables and this increases commensurately with scale and compensating for the reduced tables, making this still a weak scaling experiment. To be able to run on the smaller node (GPU) counts we shrink the embedding table cardinality while hashing inputs to be within the reduced number of rows. This *shrunk version of the model* effectively reduces the model sizes with minimal/no impact on the performance characteristics, hence is used for studying scaling performance. As seen from the figure, on 16 nodes with 128 GPUs, the scaling efficiency is around 50% for model A2 and around 40% for model A1s and A3. The achievable scaling efficiency for DLRMs are limited mostly by AlltoAll, from the model-parallel embeddings which is in the critical path with fully exposed overheads for both forward and backward passes.

Furthermore, model A2 can more effectively saturate and load-balance 128 GPUs with ~1000 embedding tables. Whereas model

A1 with fewer tables suffers from higher load imbalance, in addition lower overall utilization since the number of embedding tables is comparable to the number of GPUs. With model A3, even with larger embedding has lower scaling efficiency since there is significantly higher AlltoAll cost due to the wider embedding dimensions.

To better understand the scaling performance, we provide a breakdown of serialized and exposed training iteration latency of model A2 in Figure 12. Comparing between serialized and exposed latency, the CPU to GPU transfer (HtoD) is completely hidden and the exposed communication latency is much less than serialized AlltoAll and AllReduce latency combined, which shows the impact of pipelining optimization and PyTorch DDP communication-compute overlap. As node count increases, we see increased AlltoAll and AllReduce latencies. Since most of the AlltoAll is on the critical path, increased AlltoAll cost has direct impact on the exposed communication and overall training latency. While AllReduce is mostly hidden up to 16 nodes, the increased AllReduce latency and unchanged MLP computation latency signifies that AllReduce can become the bottleneck once the slack in backward pass is completely used up with higher node counts and/or faster compute.

5.3.2 Training throughput optimizations. Using model A2 as a case study, we detail the various optimizations and their contribution in getting to 622KQPS, see Fig. 13. The baseline performance for model A2 on 128 GPUs is below 400KQPS, with large latency disparities between embedding lookup on different GPUs, signifying severe load imbalance. This is mitigated with optimized sharding using a combination of table-wise + column-wise + data-parallel sharding for the $\approx 1000s$ of embedding tables to partition them across the 128 GPUs. It is interesting to note that even while column-wise sharding introduces additional cost of the input AlltoAll, the benefit from better load-balance outweighs the overheads and results in overall QPS improvement by 20%. However, the scaling efficiency is still about 30% lower than ideal linear scaling.

As discussed previously, the two main issues limiting scaling efficiency are: (1) load imbalance and (2) increased AlltoAll latency. For model A2, further balancing the load using only HBM is
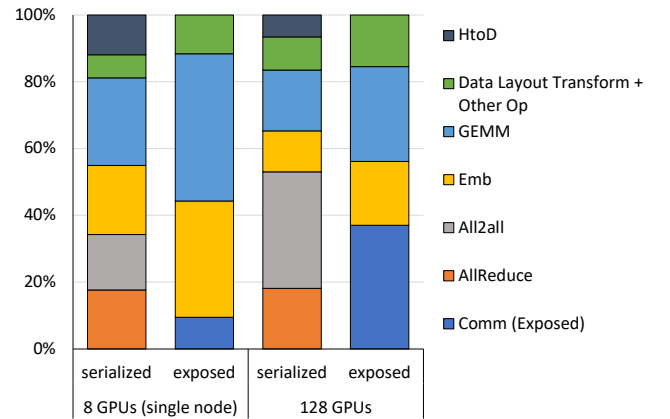
Figure 12: Model-A2 (with local batch size per GPU = 512) results and dominant operator time breakdown (serialized and exposed time) per GPU, after optimizations.
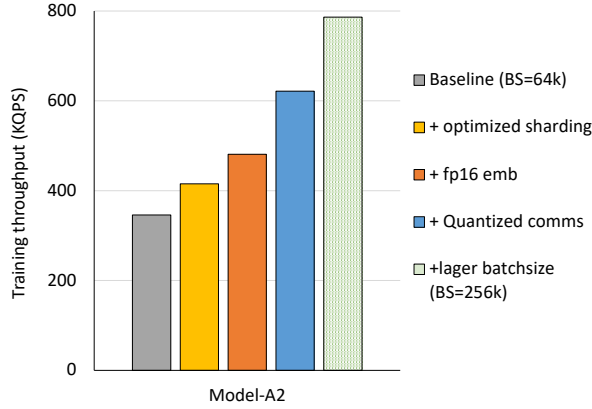
Figure 13: Training throughput improvements enabled by reduced precision embeddings, quantized communications and larger batch sizes.

particularly challenging because the 3TB model size (FP32) is very close to the 4TB aggregate HBM capacity on 128 GPUs. The sharder has very little room to explore placement strategies, after discounting for memory reserved by PyTorch framework and NCCL. To mitigate this issue, we use lower precision (FP16) embedding tables, reducing the model size by up to a factor of 2. While this alone does not provide direct throughput benefit, the sharder now can leverage the head room to strike a better balance. As a consequence, the training throughput is increased by another 20% due to improved load balancing.

Next, to address the increased `AlltoAll` latency, we incorporate quantized collective communications proposed in [58] which directly reduces the communication volume. For model A2, we validate that using FP16 in forward `AlltoAll` and BF16 for backward `AlltoAll` provide significant speedup without training quality loss.

Lastly, we further increase the global batch size, from 64K to 256K. Larger batch sizes help better saturate GPUs, improved overlap for communication, while complimentary to all of the other optimizations. With appropriately tuned optimizer/hyper-parameters we are able to achieve on-par training quality, however more comprehensive experimentation in warranted since large batch training of DLRMs is not as well studied and will be part of follow up work. Collectively, these techniques unlock an 87% improvement on training throughput compared to FP32 training with 64K global batch size.

5.3.3 Model capacity limit study. We use model F1 as an example to push the model capacity on the prototype system. Unlike model A1/A2/A3, efficiently training model F1 presents 2 different challenges. First, with 12T parameters, model F1 can easily require up to 96TB of memory using a naive training approach, far exceeding the total memory available on a 16-node cluster **. Second, the model has only a few massive embedding tables with ~10B rows and 256 columns, each requiring multi-node worth of GPU and host memory to train.

** Considering FP32 precision and doubled size for optimizer states $12e12 \times 4 \times 2 = 96e12$. The prototype cluster has in total 4TB HBM and 24TB DRAM

To fit the model onto 16 nodes, we first apply row-wise sparse AdaGrad optimizer to embedding tables which reduces optimizer states from per element to per embedding row. Then we use FP16 precision on embedding tables. These two optimizations collectively bring model memory footprint from 96TB down to 24TB, just fitting under the 4TB HBM + 24TB DRAM memory hierarchy. On the massive embedding tables, we enable row-wise sharding to distribute the tables to multiple nodes and adjust the training flow to use `AlltoAll` with bucketization and `ReduceScatter` as shown in Figure 8. With UVM enabled and HBM used as a cache, we are able to train model F1 with throughput as high as 970KQPS, demonstrating capability of our HW/SW co-designed solution to push beyond the current state-of-the-art.

6 RELATED WORK

Researchers have proposed various system level innovations to tackle the challenges brought by extremely large models. Deep-Speed [44] fully shards model parameters, gradients and optimizer states across all nodes, and reconstructs necessary states on the fly using checkpoint partitioning and rematerialization [17, 24] to drastically reduce memory usage. GShard [28] trains a massive translation model with mixture of experts that are sharded across accelerators through annotation of parallelization strategy at tensor level. FlexFlow [18] uses automatic search to discover the best parallelization strategy of operators in the graph. Building on this direction of auto-parallelization, these recent papers [36, 54] use optimal synthesis and reinforcement learning to find optimized device placement to further improve parallelization without the need for manual intervention. However, these general systems are not specifically designed for highly sparse recommendation models.

To that end, Alibaba introduced XDL [19], an industry-scale training system designed for high-dimensional sparse data. XDL incorporates optimizations such as hierarchical sample compression, workflow pipelining, zero copy and CPU binding to improve training efficiency of the sparse part of the model. Kraken [56] targets at more efficient online training with decoupled key-value fetching and embedding, codesigned cache eviction policy with ML domain knowledge for the embedding tables, memory efficient optimizers for the sparse and dense part of the model, and a non-colocated deployment model that allows the inference servers and parameter servers to grow independently. [21] optimizes CPU-based DLRM training through lock-free embedding table update, tuned loop tiling for dense MLP, the `AlltoAll` communication primitive and a new split-SGD implementation that takes advantage of the bits aliasing in FP32 and BFloat16 to reduce memory footprint. Baidu's AIBox [63] takes a different approach to horizontal scaling and focuses on fitting training of large recommendation models in a single node. AIBox hides serving latency by pipelining network, disk and CPU/GPU tasks, and reduces model update overhead and improves SSD life span through a grouped hashing scheme and a multi-level in-memory hashing system.

Much attention is given to communication performance as it has become a major bottleneck in distributed training at cluster and datacenter scale. BytePS and ByteScheduler [20, 43] harnesses idle CPU and network resources and better communication scheduling to improve parameter exchange efficiency. However, in a homogeneous

training cluster where each job spans multiple nodes, there are reduced opportunities for finding and exploiting spare network resources, resulting in a sub-optimal use of such approach. SwitchML and ATP [27, 47] leverages programmable network switches to perform in-network aggregation for cross-rack bandwidth reduction in datacenter environments. [3, 33] discovers and exploits datacenter network locality and forms optimized and dynamic aggregation routes through learning and optimal synthesis. Alternatively, these papers [30, 31] address the communication overheads by using various quantization schemes to reduce communication volume.

7 CONCLUSION

DLRMs are an important class of models widely used by many internet companies for a wide range of applications. They can often be the single largest AI application in terms of infrastructure demand in data-centers. These models have atypical requirements compared to other types of deep learning models, but they still follow a similar trend of rapid rate of growth that is common across all deep learning-based applications. This growth constantly pushes the performance boundary required of the underlying software stack and hardware platform.

Although synchronous distributed training has become common practice for many deep learning models, enabling this for DLRMs involves a number of technical challenges. In particular, we need to address the data-parallelism required for compute intensive MLPs as well as model-parallelism needed for memory intensive embeddings tables, as well as ensuring high performance embedding lookup and update operation which have far lower arithmetic intensity. This requires tightly coupling the SW stack with HW platform to ensure the highest performance.

In this paper we co-design a solution that enables us to run models with trillions of parameters, while attaining 40× faster total training time for production recommendation models. On the SW-side, features such as hybrid model partitioning, collective communication allows for continued scaling and hierarchical memory enables sustaining support for growing model sizes. On the HW-side, the extensible *ZionEX* platform allows for scaling up to the full data center with thousands of nodes, thus enabling a data center-scale AI training cluster to continue catering to the growing demands of the deep learning models. While at the node-level, the modular design enables swapping accelerators to better suit application-specific requirements.

Finally, the system also enables us to explore co-designing models and algorithms to make them more amenable to the training cluster, for instance model architectures that reduce global `AlltoAll` communication for better scaling efficiency. With this solution successfully deployed in production, we intend to continue working on these future directions to further push the capability for large scale deep learning training.

ACKNOWLEDGEMENTS

REFERENCES

[1] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim Hazelwood. 2020. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. arXiv:2011.05497 [cs.AR]

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[3] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 62–75.

[4] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide and Deep Learning for Recommender Systems. *arXiv:1606.07792* (2016). http://arxiv.org/abs/1606.07792

[5] François Chollet. 2017. Xception: Deep Learning with Depthwise Separable Convolutions. arXiv:1610.02357 [cs.CV]

[6] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for YouTube recommendations. In *Proc. 10th ACM Conf. Recommender Systems*. 191–198.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]

[8] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research* 12, 7 (2011).

[9] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Murali Annavaram, Krishnakumar Nair, and Misha Smelyanskiy. 2020. Check-N-Run: A Checkpointing System for Training Recommendation Models. arXiv:2010.08679 [cs.IR]

[10] Carlos A. Gomez-Uribe and Neil Hunt. 2016. The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Trans. Manage. Inf. Syst.* 6, 4, Article 13 (Dec. 2016), 19 pages. https://doi.org/10.1145/2843948

[11] Maya R Gupta, Samy Bengio, and Jason Weston. 2014. Training highly multiclass classifiers. *The Journal of Machine Learning Research* 15, 1 (2014), 1461–1492.

[12] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. 2020. The Architectural Implications of Facebook's DNN-Based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 488–501. https://doi.org/10.1109/HPCA47549.2020.00047

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

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385 [cs.CV]

[15] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proc. 26th Int. Conf. World Wide Web*. 173–182.

[16] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising* (New York, NY, USA) *(ADKDD'14)*.

[17] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. 2019. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *arXiv preprint arXiv:1910.02653* (2019).

[18] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358* (2018).

[19] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. 2019. XDL: an industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data.* 1–9.

[20] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[21] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. 2020. Optimizing Deep Learning Recommender Systems Training on CPU Cluster Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20).* IEEE Press, Article 43, 15 pages.

[22] Narenda Karmarker and Richard M. Karp. 1983. *The Differencing Method of Set Partitioning.* Technical Report. USA.

[23] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).

[25] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 8 (2009), 30–37.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12).* Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[27] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. [n.d.]. ATP: In-network Aggregation for Multi-tenant Learning. ([n.d.]).

[28] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[29] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3005–3018.

[30] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2018. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. *arXiv preprint arXiv:1802.07389* (2018).

[31] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887* (2017).

[32] Romain Lopez, Inderjit Dhillon S., and Michael Jordan I. 2021. Learning from eXtreme Bandit Feedback. In *Proc. Association for the Advancement of Artificial Intelligence.*

[33] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. 2020. PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training.

[34] Yifei Ma, Balakrishnan (Murali) Narayanaswamy, Haibin Lin, and Hao Ding. 2020. Temporal-Contextual Recommendation in Real-Time *(KDD '20).* Association for Computing Machinery, New York, NY, USA, 2291–2299.

[35] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. arXiv:1910.01500 [cs.LG]

[36] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. https://arxiv.org/abs/1706.04972

[37] Dheevatsa Mudigere and Whitney Zhao. 2019. HW/SW Co-design for future AI platforms - Large memory unified training platform (Zion). In *2019 OCP Regional Summit, Amsterdam.* https://2019ocpregionalsummit.sched.com/event/Qyge

[38] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, Krishnakumar Nair, Isabel Gao, Bor-Yiing Su, Jiyan Yang, and Mikhail Smelyanskiy. 2020. Deep Learning Training in Facebook Data Centers: Design of Scale-up and Scale-out Systems. arXiv:2003.09518 [cs.DC]

[39] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[40] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21).* USENIX Association, 217–231. https://www.usenix.org/conference/fast21/presentation/pan

[41] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. 2018. Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications. arXiv:1811.09886 [cs.LG]

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems.* 8026–8037.

[43] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* 16–29.

[44] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–16.

[45] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011), 693–701.

[46] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. 2020. MLPerf Inference Benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).* 446–459.

[47] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[48] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815 [cs.AI]

[49] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (10 2017), 354–359. https://doi.org/10.1038/nature24270

[50] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV]

[51] M. Smelyanskiy. 2019. Zion: Facebook Next-Generation Large Memory Training Platform. In *2019 IEEE Hot Chips 31 Symposium (HCS).* https://doi.org/10.1109/HOTCHIPS.2019.8875650

[52] Brent Smith and Greg Linden. 2017. Two Decades of Recommender Systems at Amazon.Com. *IEEE Internet Computing* 21, 3 (May 2017), 12–18. https://doi.org/10.1109/MIC.2017.72

[53] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. arXiv:1409.4842 [cs.CV]

[54] Jakub Tarnawski, Amar Phanishayee, Nikhil R Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of dnn graph operators. *arXiv preprint arXiv:2006.16423* (2020).

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[56] M. Xie, K. Ren, Y. Lu, G. Yang, Q. Xu, B. Wu, J. Lin, H. Ao, W. Xu, and J. Shu. 2020. Kraken: Memory-Efficient Continual Learning for Large-Scale Real-Time Recommendations. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–17. https://doi.org/10.1109/SC41405.2020.00025

[57] Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. 2020. Mixed-Precision Embedding Using a Cache. arXiv:2010.11305 [cs.LG]

[58] Jie Amy Yang, Jongsoo Park, Srinivas Sridharan, and Ping Tak Peter Tang. 2020. Training Deep Learning Recommendation Model with Quantized Collective Communications. (2020).

[59] Chunxing Yin, Bilge Acun, Xing Liu, and Carole-Jean Wu. 2021. TT-Rec: Tensor Train Compression for Deep Learning Recommendation Models. arXiv:2101.11714 [cs.LG]

[60] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).

[61] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. *Advances in neural information processing systems* 28 (2015), 685–693.

[62] Whiteny Zhao, Dheevatsa Mudigere, Xiaodong Wang, Jongsoo Park, John Kim, and Mikhail Smelyanskiy. 2019. Accelerator Fabric in Facebook Zion Training System. In *2019 IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*.

[63] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 319–328.

[64] Qinqing Zheng, Bor-Yiing Su, Jiyan Yang, Alisson Azzolini, Qiang Wu, Ou Jin, Shri Karandikar, Hagay Lupesko, Liang Xiong, and Eric Zhou. 2020. ShadowSync: Performing Synchronization in the Background for Highly Scalable Distributed Training. *CoRR* 2003.03477 (2020).

APPENDIX-A

We collected and developed a set of operator-level benchmarks which we have also open sourced as part of *PARAM bench*[††], to evaluate the representative problem sizes and shapes on the candidate hardware platforms and to better understand the throughput and latency in compute, memory, and communications.

Compute Benchmark

GEMM benchmark. This benchmark calls cuBLAS GemmEx routine to compute matrix multiplications on configurable problem sizes with multiple precision choices. On the V100 GPU, this benchmark supports FP32 GEMM on the CUDA core and FP16 mixed-precision GEMM on Tensor Core. On the A100 GPU, it additionally supports TF32 GEMM and BF16 GEMM on the Tensor Core.

The benchmark results are shown in Figures 14 and 15.

7.0.1   MLP benchmark. This benchmark implements the following multilayer perceptron (MLP) layers:

- Batch size = 128, 256, 512, 1024, 2048, 4096;
- 20 MLP layers, where each layer is 1K×1K , 2K×2K and 4K×4K;
- Each layer has ReLU and final layers has SoftMax;
- Both backward and forward passes, including SGD update as the optimizer after the backward pass;
- Precision support: FP16, BF16, TF32, FP32.

The batch size, layer dimension, and number of layers can be configured to the customized number. We implemented this MLP benchmark using C++, directly implementing FC and FCGradients in the MLP layer using cuBLAS SGEMM/GemmEx function, ReLU with cuDNN cudnnActivationForward/ cudnnActivationBackward function, SoftMax with cudnnSoftmaxForward in the forward a customized CUDA kernel for the backward pass, and SGD optimizer with cuBLAS axpy function. This benchmark can be used to project the performance of V100/A100 GPUs using a minimal MLP network without the framework overhead in PyTorch. The benchmark results are shown in Figures 16 and 17.

[††] https://github.com/facebookresearch/param

Table 5: Supported precisions in GEMM benchmark.

A/B Precision	C Precision	Accumulation Precision	CUDA/Tensor Core	Notes
FP32	FP32		CUDA	V100/A100
FP16	FP32	FP32	Tensor	
	FP16			
TF32	TF32			A100 only
BF16	FP32			
	BF16			
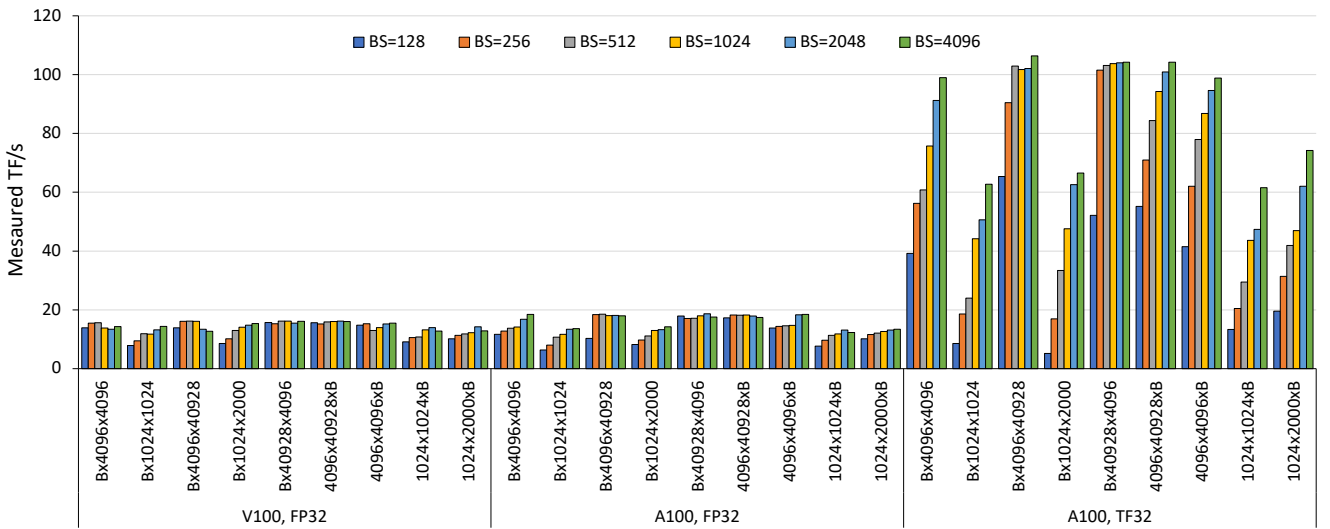
Figure 14: GEMM performance (TF/s) for V100 FP32 vs. A100 FP32/TF32.
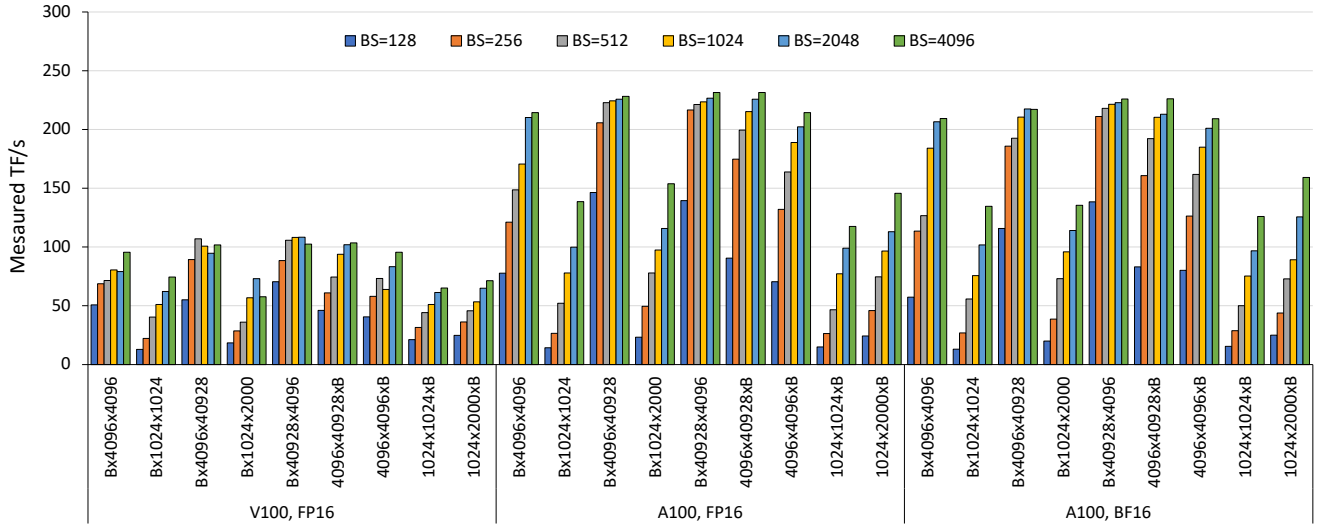
Figure 15: GEMM performance (TF/s) for V100 FP16 vs. A100 FP16/BF16.

Memory Benchmark

This benchmark evaluates the achieved memory bandwidth of the embedding kernels described in Section 4.1. To eliminate the L2 cache effects, a random tensor with 40 MB data (A100 L2 cache size) is allocated to flush the cache.

- Support the evaluation forward and backward pass (the backward pass is fused with optimizer);
- Precision Support: FP32 and FP16;
- Number of rows: 1000000, Number of tables: 64, Embedding dimension: 128, Pooling size: 32, rows per thread block: 32.

The benchmark results are shown in Figures 18 and 19.

Communication Benchmark

Low-level collective communication benchmarks, e.g. NVIDIA's NCCL tests or OSU MPI benchmarks, have the following limitations:

- Do not capture the behavior of actual workloads, i.e. exact message sizes, sequence of collective operations, etc. Instead these benchmarks support power-of-two message sizes - helpful to detect network trends.
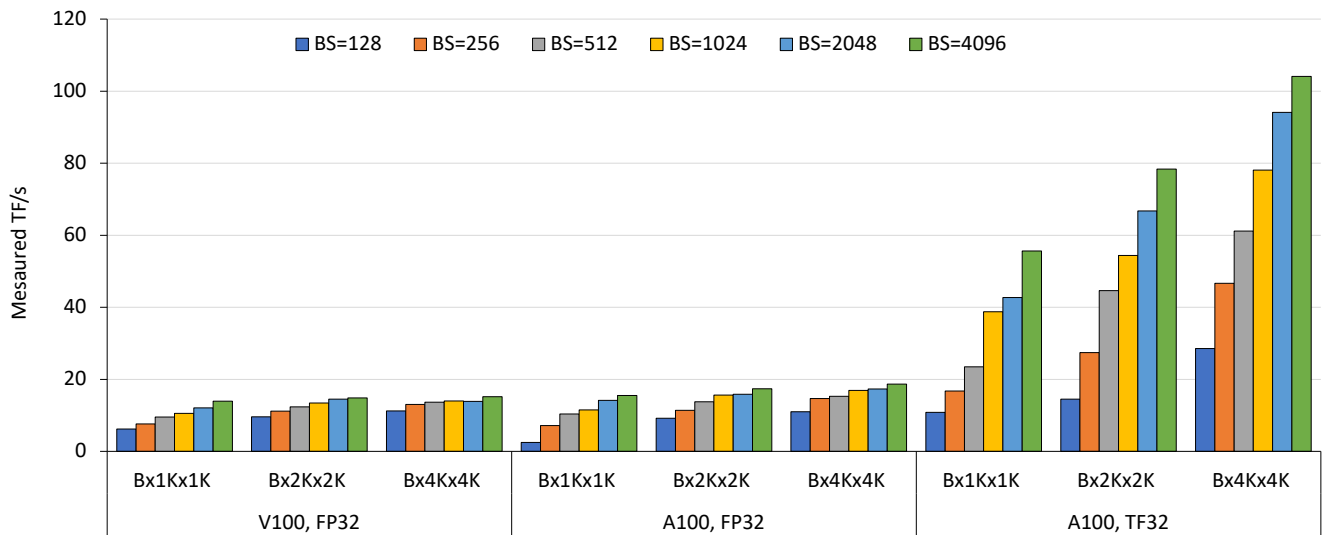

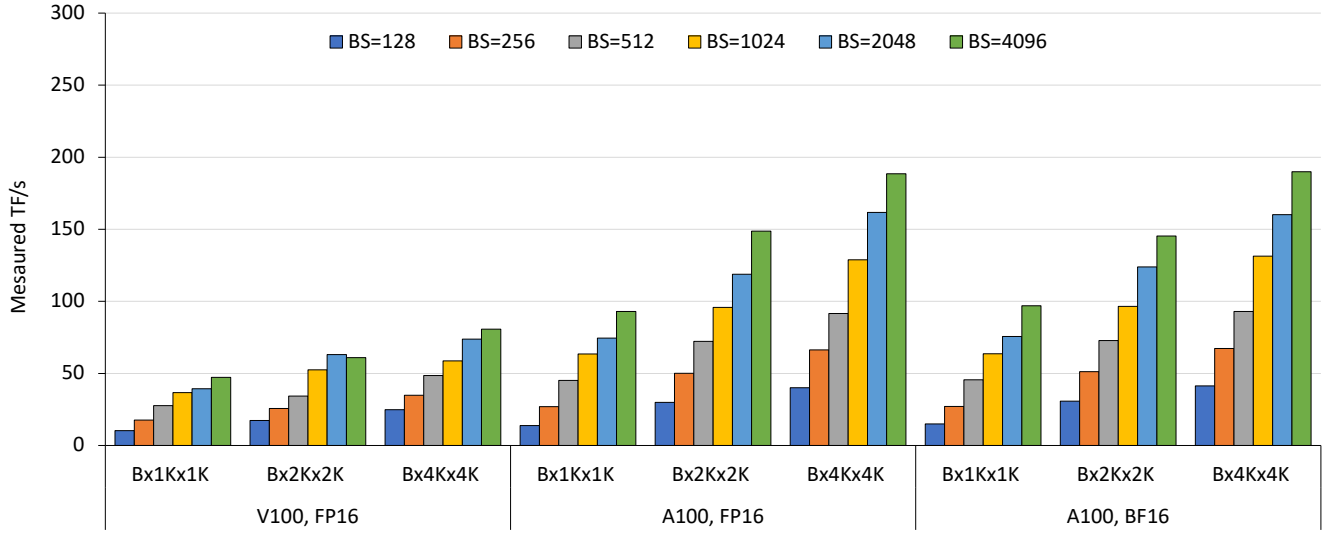**Figure 16: MLP performance for V100 FP32 vs. A100 FP32/TF32.**

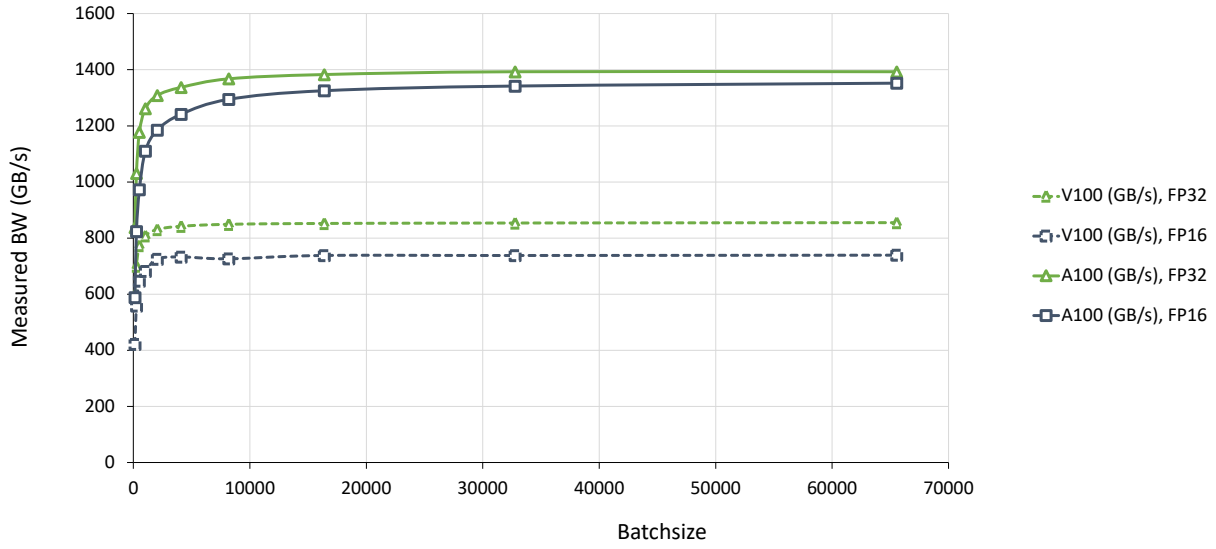**Figure 17: MLP performance for V100 FP16 vs. A100 FP16/BF16.**

Figure 18: Achieved embeddding lookup forward bandwidth using FP32 vs. FP16 on V100 vs. A100.

- Limited to one specific communication library. As the name suggests, NCCL tests works only with NCCL and OSU MPI benchmarks is limited to MPI.

The PARAM comms benchmarks addresses these gaps by:

- Creating common abstractions across platforms (e.g. NVIDIA GPUs, x86 CPUs, Google TPU etc.) to help standardize the benchmarking logic.
- Using PyTorch Process Group APIs to provide a portable interface across different communication libraries (e.g. NCCL, MPI, and UCC).

PARAM comms benchmarks supports two types of collective benchmarks:

- Bench mode: Simplest mode of operation similar to NCCL tests. Run single collective in blocking or non-blocking manner across fixed set of message sizes (e.g. power of 2 message sizes). This is mainly used for low-level HW testing
- Replay mode: Replays a trace of collective communication calls to mimic exact workload behavior in terms of collective sizes.
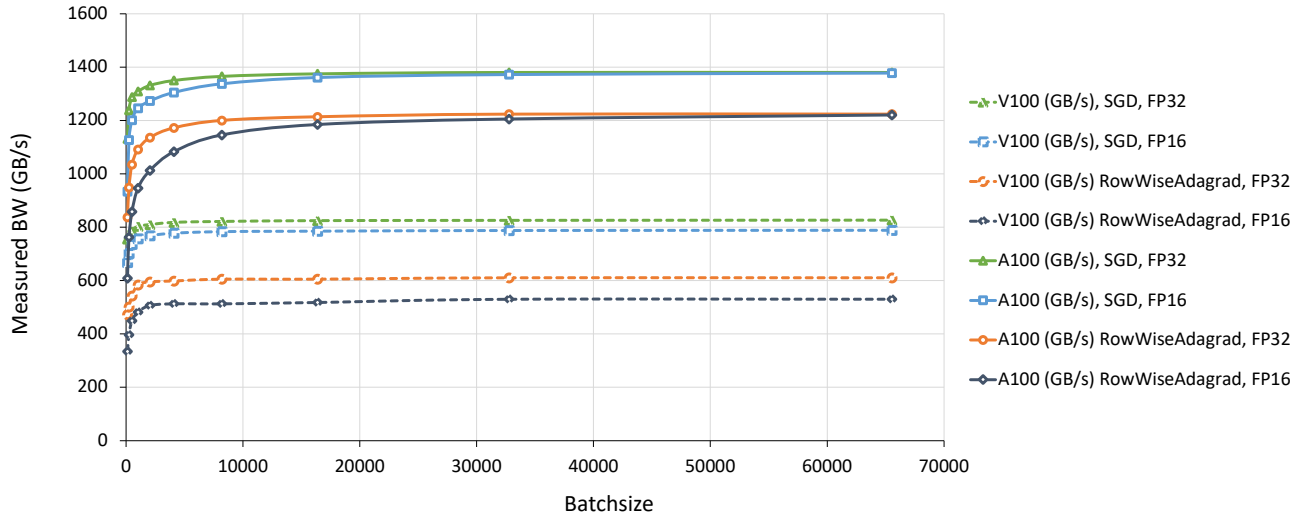
Figure 19: Achieved embeddding lookup backward+optimizer bandwidth using FP32 vs. FP16 on V100 vs. A100.
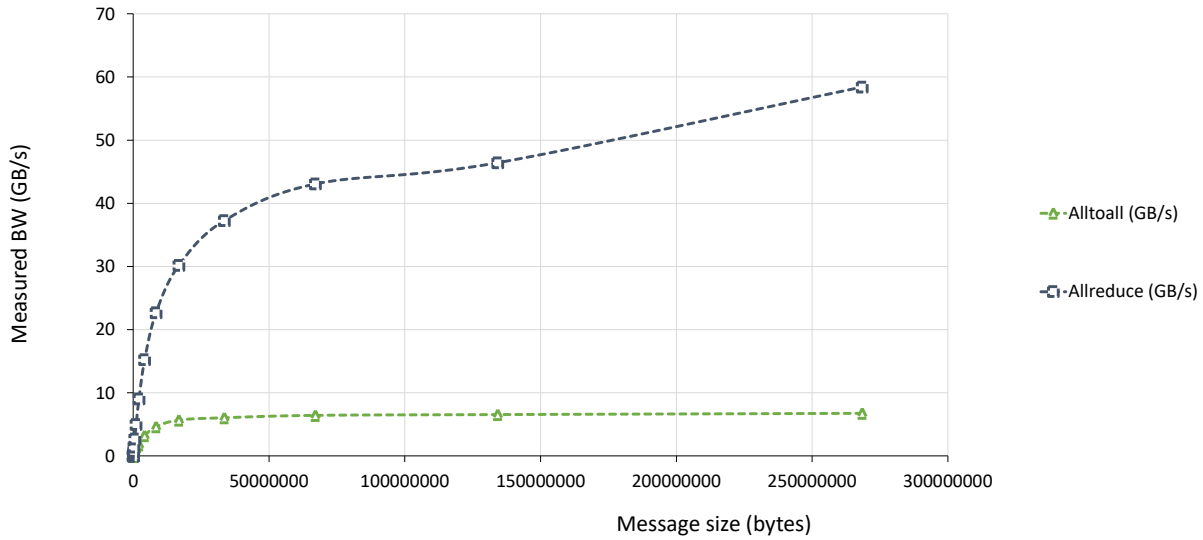

Figure 20: Achieved Alltoall and Allreduce bandwidth at 128GPUs

Figure 20 presents Alltoall and Allreduce benchmark scaling for power-of-two message sizes on 128 GPUs. Alltoall achieves 7GB/s and is primarily limited by scale-out bandwidth (12.5 GB/s peak; 10.5 GB/s achievable on V100). Allreduce achieves higher bandwidth since it uses NVLINK more effectively.