Multi- GPU Parallelism and Large Sparse Data Movement in DL Models

Ujwal Pratap Krishna[1], **Aryaman Bahukhandi**[1]

1 Introduction

Recommendation systems have become the backbone of data driven e-commerce and many other profitable sectors. Up until 2013, Recommendation systems were driving 35% of Amazon's sales [12]. It can be safely said that this number has only increased, since over 80% of the consumers have reported their preference for personalized recommendations [4]. Leading this revolution at the forefront are recommendation systems that have a deep neural network based architecture. Of these recommendation systems, Facebook's Deep Learning Recommendation Model has recently established as the state of the art. [15]

Facebook's Deep Learning Recommendation Model (DLRM) [15] uses tabular data-based feature embeddings. It scales well and makes accurate, customized recommendations. The model architecture consists of very large embedding and multi-layer perceptron layers. And the tabular data consists of multiple dense as well as categorical features. Traditionally, tabular data-based embedding training regimes are used to train DLRM, because they can handle high-dimensional sparse data and capture non-linear relationships, generalization, transferability, and variable interactions[13, 15].

However, these tabular data based training regimes suffer from a highly exacerbated curse of dimensionality. Categorical columns have high cardinality; hence embeddings are high-dimensional. E.g. UserID and ZipCode columns should have several distinct values. This increases embedding table size, as seen in open-source datasets like the 23-day Criterio Dataset, which can exceed 400 GB[2]. The tabular data also has a lot of 0-sparsity because of the categorical features. Our work has observed that for Criteo Dataset, 96% of the data locations have 0-sparsity. [1] Table lookup and update operations in embedded systems require a lot of memory bandwidth, but they are also computationally expensive. The concurrent execution of multiple memory and compute operations during model training may burden the architecture's subsystems[5]. The embeddings, therefore, can't be stored in a single device because of their size. The DLRM model often exceeds the size of most throughput devices and is in order of 140GB at least[14].

Since AlexNet[11], most deep learning workloads employ the use of GPUs for computation. However, due to DLRM's large size, it can not fit in 1 GPU. The model size is also increasing with the advent of Transformer [17] based architecture and the same trend will continue in recommendation systems as well. Therefore, various model parallel and data parallelism solutions are applied, yet the scope of efficient training is still bounded by the memory available. Most popular are splitting and storing each neural network layer in individual devices and training [9, 14].

Computer systems have been bounded by the size of fast local memory like DRAM and VRAM in CPUs and GPUs, respectively. So various heterogeneous memory solutions have been employed to store the embeddings in NVRAM which have a comparatively larger memory capacity at the cost of increased latency. These heterogeneous memory solutions lose out on parallel model performance as CPU memory is again constrained [5, 9, 13].

In the context of training the Deep Learning Recommendation Model (DLRM) within a parallel architecture setup, another challenge arises. During the forward propagation process, GPUs encounter a situation where they need to wait often for the computation of intermediate outputs of the corresponding layers, resulting in a synchronization delay. This issue of busy waiting introduces a bottleneck that hampers the overall training efficiency of the model [14].

[1] • All authors are with University of California, Davis. E-mail: ujwkal, abahukhandi@ucdavis.edu.

This problem is of utmost importance because the size of the feature embeddings is growing exponentially with the increasing availability of digital platforms that gather extensive user data. Also, maximizing the usage of computing resources per node will decrease the financial cost of training DLRM over a large parameter space.

Various optimization techniques on data parallelism and intra-layer model parallelism have been employed in the context of Natural Language Processing. Yet, not many optimizations have been performed in the context of DLRM. Efficiently computing sparse matrices and optimizing them in distributed training settings has been used but not in the context of Heterogeneous Memory Systems with NVRAM.

We propose a distributed training regime that employs a multi-GPU intra-layer model parallel approach, where synchronization time between devices will be reduced. Furthermore, we will use NVRAM to store large embedding table data and distribute the embedding layer across a multi-GPU setup. To optimize the data movement across all these components, we will initialize a software cache, RecCacheEmbedding, to conserve data movement. This will reduce the number of access to NVRAM, which has a relatively higher read latency. [16, 7, 13, 9, 14].

2 Prior Work

Maxim Naumov et al. [15] emphasize the significance of personalization in recommendation systems and the need for further advancements in deep learning techniques. They discuss challenges such as the cold-start problem (where new users or items have limited data) and data sparsity, which require innovative solutions. Data sparsity and data movement has inspired much of the research in optimizing these large recommendation systems. In previous research endeavours, the storage and processing of large sparse embedding representations were predominantly carried out on central processing units (CPUs). In a noteworthy publication, Hildebrand et al. [6] introduced a novel approach for DLRM, by utilizing an NVMe solid-state drive (SSD) cache specifically designed for storing embeddings. Moreover, recent studies conducted by Adnan et al. [1] and Kassa et al. [10] presented an innovative software cache solution for GPUs, which proved to be effective in managing high-frequency data. Nevertheless, their implementations were limited to a single GPU, and they encountered challenges when dealing with scenarios that involved scarce high-frequency data.

Additionally, GPipe [7] and Megatron-LM [16] emerged as optimization techniques for model parallelism. GPipe primarily focused on leveraging instruction-cycle parallelism to enhance vertical slicing, while Megatron-LM adopted a different approach by prioritizing horizontal slicing. However, both GPipe and Megatron-LM exhibited a common limitation: the lack of efficient communication mechanisms for sparse matrix data. Furthermore, their exclusive emphasis on language modeling tasks hindered their applicability to broader domains.

3 Main Idea

Our hypothesis is that we are optimizing communication channels between CPU-GPU and GPU-GPU by :

1. Caching locations of frequently accessed values within data with high 0 sparsity and communicating only the locations. Since 95% of the locations of the datasets in our consideration had 0 sparsity, we expect a 19x improvement in memory bandwidth.
2. Optimizing memory utilization bottlenecks by distributing computations across GPUs uniformly through intra-layer model parallelism

To achieve the above hypothesis, we have 3 major components to our system as depicted in Figure 1:

1. Heterogeneous Memory Platform based Embedding Data NVRAM Storage
2. Cache Embedding

3. Multi GPU Intra Layer Parallelism

We use a Heterogeneous Memory Platform for storing large embedding data. Then we cache frequently accessed values in software cache in the DRAM, following which we use a multi-GPU intra-model parallelism technique to perform training-based computation.

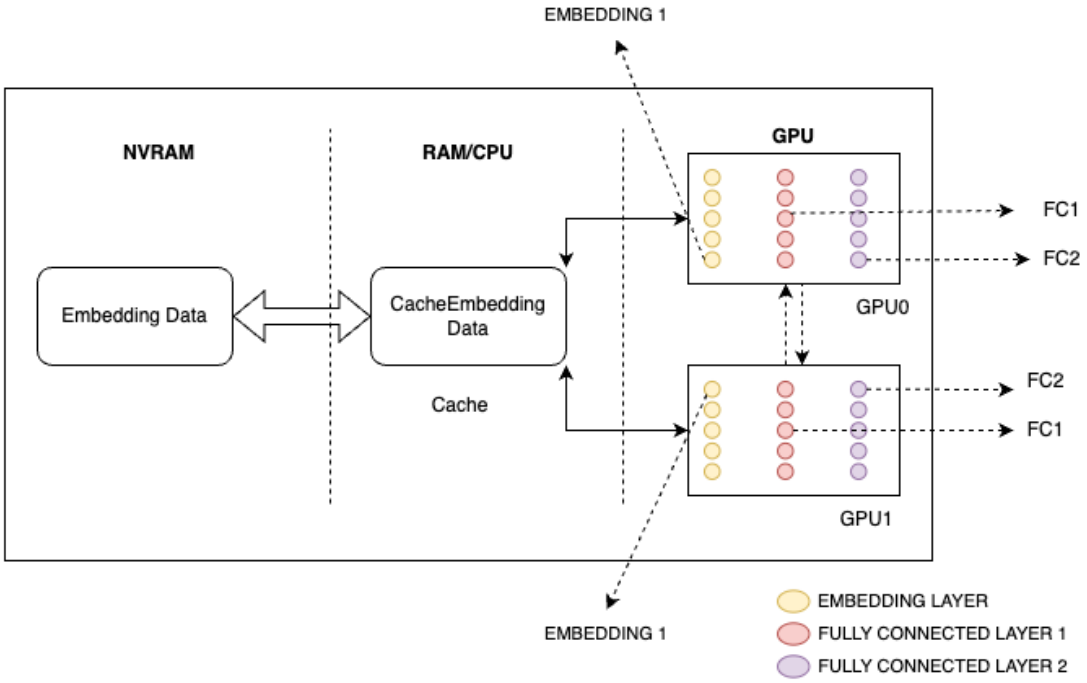

Figure 1: Data flow of embeddings from NVRAM to GPUs. Initially, the frequently accessed embedding data is cached in the CPU RAM. Subsequently, it is distributed across the horizontally sliced embedding layer, fully connected layer-1, and fully connected layer-2 during the training process.

3.1 Embedding Data Storage on Heterogeneous Memory Platform

Recreating results from [6, 3], we use a system having DRAM and non-volatile memory(NVM) and refer to it as a heterogeneous memory system, e.g., Intel's Optane Persistent Memory based on 3DXpoint non-volatile memory technology, which is called Persistent Memory(PM). [8]. In addition, we also add multiple GPUs to this system. We then leverage this platform's speedups between NVRAM and DRAM, being on the same memory channels. This system comprised of higher capacity NVRAM with DRAM serving as a cache to it is called a 2LM mode configuration[8]. We store the categorical one-hot encoded data (embedding table data) in NVRAM and use CacheEmbeddingData stored in DRAM to act as a cache for the embedding table data.

3.2 Cache Embedding Data

We draw inspiration from CacheEmbedding [6] and support CUDA Kernel-based operations for establishing high bandwidth memory channels between DRAM and multiple GPUs connected to the system.

CacheEmbedding [6] is mainly used for storing embedding weights and our CacheEmbeddingData will hold the value of the categorical one hot encoded values, and the embedding layer will be horizontally sliced and stored in multiple GPUs.

The access granularity of the embedding tables is based on feature vectors, rather than individual elements. This optimization technique decreases the number of memory accesses needed, thereby enhancing system performance. The presence of spatial locality in accesses cannot be assumed due to the typically large and sparse nature of the embedding tables. Consequently, conventional caching

methodologies may prove to be ineffective.

The phenomenon of the temporal locality may be observed in the accesses, whereby feature vectors that have been accessed recently are more probable to be re-accessed in the immediate future.

The CachedEmbeddings technique introduces an additional layer of indirection to the retrieval of feature vectors, enabling the caching of individual feature vectors in DRAM while being stored in PM. The caching of frequently accessed feature vectors enables expedited access, while those accessed infrequently can persist in the PM.

A caching mechanism for embedding table entries facilitates the implementation of customized policies specifically designed to align with the observed distribution in embedding table accesses. In summary, the utilization of CachedEmbeddings presents a viable and effective approach towards integrating extensive DLRM models within heterogeneous memory systems, capitalizing on the advantages offered by both DRAM and PM.

3.3 Multi GPU-based Intra-Layer Model Parallelism

Horizontal splicing of the layers in the model efficiently distributes the very large embedding and matrix computation between the GPUs and makes training synchronously and asynchronously possible.

The work of MEGATRON-LM [16] in language modelling shows that training large transformer models advances NLP applications. We draw inspiration from the communication models present in MEGATRON LM [16] and then initialize our communication models, which will communicate with the CacheEmbedding-Data. We implement DLRM using intra-layer model parallelism inspired from Hildebrand et al. [6].

For the forward pass, we split() the layers across GPUs. Then during backward propagation, when we want to sync the gradients across all models, we perform an AllReduceGradients(). These communication models are implemented in PyTorch MPI functionality.

4 Methodology

We implemented a deep learning model using PyTorch, incorporating horizontal slicing and caching of non-sparse indices. Our objective was to evaluate this approach against a state-of-the-art pipeline parallelized model, which served as our baseline. To conduct the evaluation, we employed a hardware setup consisting of two Nvidia L40 GPUs, each equipped with 32 vCPUs, 128 GB RAM, and a 250 GB SSD. This configuration ensured an appropriate environment with sufficient computational power and memory capacity for training large-scale recommendation models.

To represent realistic training workloads, we selected two datasets: the Criterio RecSys Challenge 2020 Dataset[sss] and the MovieLens Dataset[ss]. These datasets are widely used in recommendation system research and exhibit nearly 94% sparsity in the categorical data, enabling a comprehensive evaluation of the parallelization approaches.

We evaluated two parallelization approaches: horizontal data and model parallelism with cached non-sparse indices, and state-of-the-art data and pipeline model parallelism. The evaluation primarily focused on assessing training performance and resource utilization of these approaches. By comparing the two methods, our aim was to identify the approach that achieves superior results in terms of training time and memory efficiency.

To quantify the performance of the parallelization approaches, we monitored key benchmark metrics using the PyTorch Profiler. These metrics included the average CUDA time in milliseconds, GPU-GPU memory utilization in megabytes, and CPU-GPU memory utilization in Megabytes. By analyzing these metrics, we obtained insights into the computational efficiency and memory utilization of the different parallelization approaches.

The chosen hardware setup, dataset selection, and benchmark metrics monitoring formed a rigorous methodology for comparing the parallelization approaches in terms of large sparse datasets. The hardware configuration provided an appropriate environment, while the selected datasets represented realistic training workloads of recommendation systems. The benchmark metrics offered quantitative measures to assess training performance and resource utilization accurately. Through this evaluation, we aimed to gain valuable insights that would contribute to the advancement of efficient parallelization techniques for training large-scale recommendation models.

In conclusion, our evaluation plan encompassed all the necessary components to determine the success of the parallelization approaches. By analyzing the benchmark results and comparing the performance and resource utilization of the two approaches, we aimed to contribute to the development of horizontal parallelization techniques with caching non-sparse data for training recommendation models at scale.

5 Preliminary Results

In our preliminary work, we conducted benchmarking experiments to compare the training process on both Multi-GPU and GPU-CPU setups. The objective was to evaluate the optimizations achieved through horizontal slicing of the model and caching of non-sparse indices in terms of GPU compute and utilization.

Our findings demonstrated that our proposed approach, which utilizes horizontal slicing and caching, offered a significant speedup of approximately 500ms compared to pipeline parallelism during the forward propagation of training as shown in Figure 2. This speedup was primarily attributed to the optimized data movement in the horizontally sliced model, which required fewer synchronization points for weight and bias calculations. In contrast, the vertically sliced model experienced delays as it had to wait for each layer to complete its computation, resulting in several synchronization points for data fetching and offloading across the GPUs.

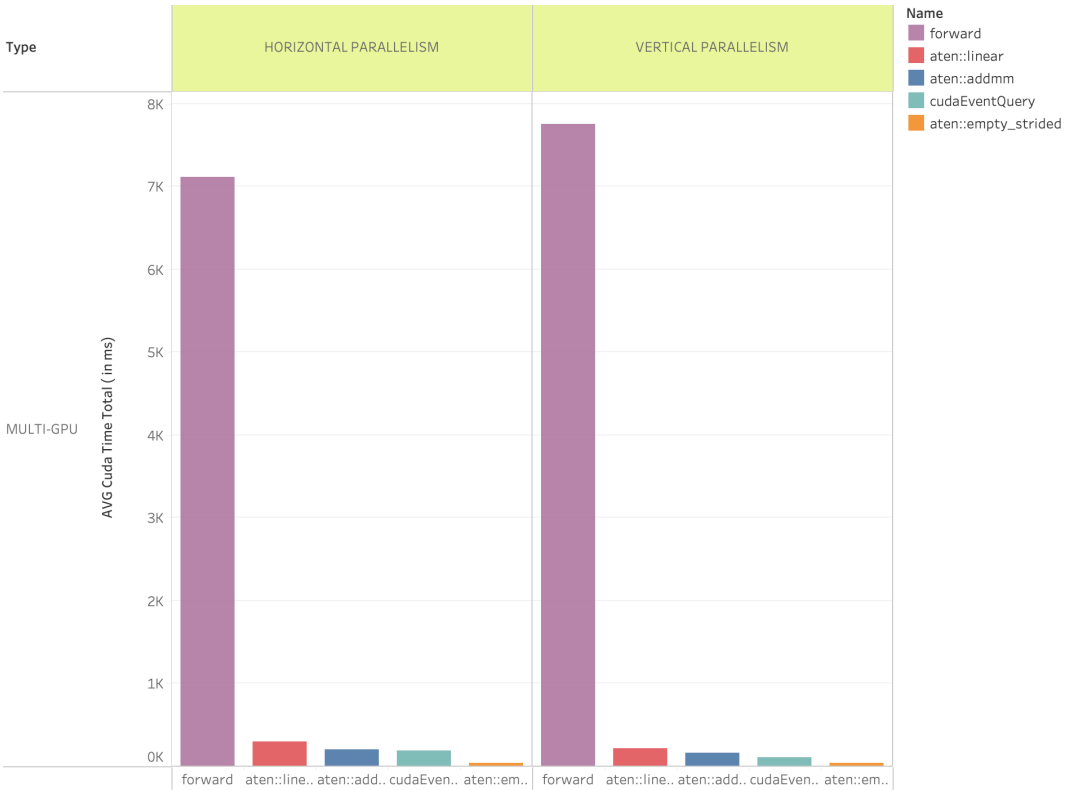

Figure 2: Average CUDA Time Total during 5 repeated training runs for horizontal parallelism vs vertical parallelism.

Further analysis of the time savings provided by our approach as represented in Figure 3 revealed notable improvements at each operation level within the forward propagation. Specifically, we observed a reduction of nearly 1800Mb in GPU memory utilization for the "aten::to" operation, which is responsible for transferring input data, weights, and bias between CPU and GPU, or GPU to GPU. This sucessfuly emphasizes and demonstrates the merits of our optimized data movement algorithm powered by our SparseEmbedding implementation. Additionally, significant memory savings were observed for the "atten::resize" operation, which is frequently used in state-of-the-art pipeline parallelized models to resize tensors in sequential and embedding layers. This is a testament to the success of our optimized memory utilization facilitated by intra-layer model parallelism.

Parallelism	Name	GPU0 MEMORY (in Mb)	GPU1 MEMORY (in Mb)
horizontal	aten::mm	15,260	15,240
	aten::linear	757	879
	aten::resize_	7,630	7,630
	aten::to	148	123
	cudaEventQuery	47	642
vertical	aten::mm	15,260	15,260
	aten::linear	581	581
	aten::resize_	15,130	15,130
	aten::to	1,950	1,950
	cudaEventQuery	213	587

Figure 3: Candidate application selection flowchart

We also investigated the CPU and CUDA memory utilization of the two models under test. Our findings revealed insightful optimizations in the horizontal model parallelism with cached non-sparse indices, shown in Figure4. We observed that the operation "atten::_to_copy," used to copy the large sparse matrix training data to the GPU, solely utilized CUDA memory. Horizontal slicing and caching minimized the need for frequent data retrieval from the CPU, as the data was efficiently sliced and transferred to each GPU, ensuring that the data required for a particular slice of the deep learning model resided within the same GPU. Data was only copied to subsequent GPUs during the forward propagation when needed for computation. In contrast, vertical slicing of the model necessitated significant data copying between the CPU and GPU, involving loading input data, offloading intermediate layer outputs and weights to the CPU, and transferring this intermediate output from the CPU back to the next GPU.

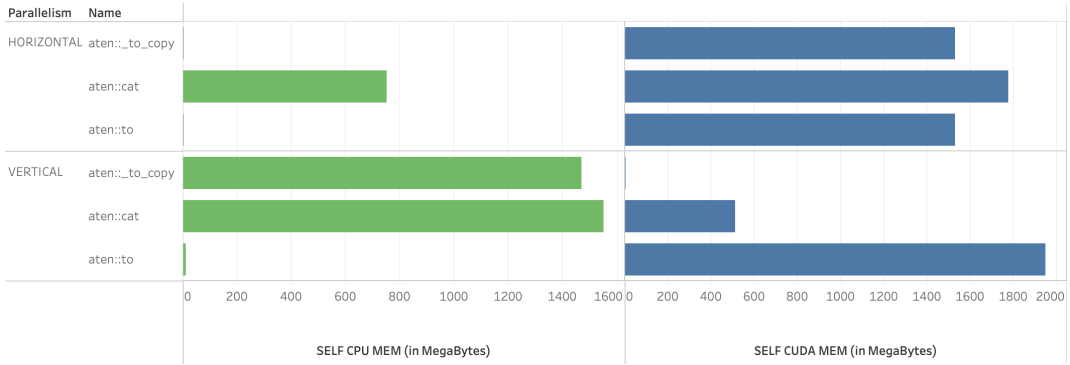

Figure 4: Comparison of Self CPU Memory and Self CUDA Memory in Mb for different operations within forward propagation for horizontal and vertical parallelism.

Furthermore, we observed that the "atten:cat" operation, responsible for concatenating weights computed during forward propagation, exhibited significantly higher GPU utilization in our proposed approach. This is

attributed to the manageable size of the concatenated data within the GPU's virtual memory (vRAM) due to the horizontal slicing of the model.

In summary, our results demonstrate the advantages of our proposed approach in terms of GPU compute and utilization. The horizontal model parallelism with cached non-sparse indices significantly reduced synchronization points, optimized data movement, and improved memory utilization compared to state-of-the-art pipeline parallelism. These findings validate the effectiveness of our approach in enhancing the training performance of deep learning models.

6 Conclusion

Our evaluation highlights the benefits of horizontal data and model parallelism with cached non-sparse indices for training deep learning recommendation models. This approach improves training performance by reducing synchronization points and optimizing data movement, resulting in faster forward propagation and reduced GPU memory consumption. While our findings are promising, it is important to consider the risks and limitations associated with our analysis, such as generalizability and implementation complexity. Further research is needed to address these concerns and ensure the practical applicability of our proposed solution. Nonetheless, our evaluation provides valuable insights for the development of efficient parallelization techniques in training large-scale recommendation models.

References

[1] Muhammad Adnan et al. "Accelerating Recommendation System Training by Leveraging Popular Choices". In: *Proc. VLDB Endow.* 15.1 (Sept. 2021), pp. 127–140. ISSN: 2150-8097. DOI: 10.14778/3485450.3485462. URL: https://doi.org/10.14778/3485450.3485462.

[2] Aditya Desai and Anshumali Shrivastava. "The trade-offs of model size in large recommendation models : 100GB to 10MB Criteo-tb DLRM model". In: *Advances in Neural Information Processing Systems*. Ed. by Alice H. Oh et al. 2022. URL: https://openreview.net/forum?id=c9I_NArDIjD.

[3] Assaf Eisenman et al. *Bandana: Using Non-volatile Memory for Storing Deep Learning Models*. 2018. arXiv: 1811.05922 [cs.LG].

[4] Epsilon. "New Epsilon research indicates 80% of consumers are more likely to make a purchase when brands offer personalized experiences". In: (2018).

[5] Udit Gupta et al. "The Architectural Implications of Facebook's DNN-based Personalized Recommendation". In: *CoRR* abs/1906.03109 (2019). arXiv: 1906.03109. URL: http://arxiv.org/abs/1906.03109.

[6] Mark Hildebrand, Jason Lowe-Power, and Venkatesh Akella. "Efficient Large Scale DLRM Implementation on Heterogeneous Memory Systems". In: *High Performance Computing*. Ed. by Abhinav Bhatele et al. Cham: Springer Nature Switzerland, 2023, pp. 42–61. ISBN: 978-3-031-32041-5.

[7] Yanping Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *CoRR* abs/1811.06965 (2018). arXiv: 1811.06965. URL: http://arxiv.org/abs/1811.06965.

[8] Joseph Izraelevitz et al. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. 2019. arXiv: 1903.05714 [cs.DC].

[9] Dhiraj Kalamkar et al. *Optimizing Deep Learning Recommender Systems' Training On CPU Cluster Architectures*. 2020. arXiv: 2005.04680 [cs.DC].

[10] Hiwot Tadese Kassa et al. *MTrainS: Improving DLRM training efficiency using heterogeneous memories*. 2023. arXiv: 2305.01515 [cs.IR].

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[12] Ian MacKenzie, Chris Meyer, and Steve Noble. "How retailers can keep up with consumers". In: *McKinsey & Company* 18.1 (2013).

[13] Dheevatsa Mudigere et al. *Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models*. 2023. arXiv: 2104.05158 [cs.DC].

[14] Kabir Nagrecha. *Systems for Parallel and Distributed Large-Model Deep Learning Training*. 2023. arXiv: 2301.02691 [cs.DC].

[15] Maxim Naumov et al. "Deep learning recommendation model for personalization and recommendation systems". In: *arXiv preprint arXiv:1906.00091* (2019).

[16] Mohammad Shoeybi et al. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism". In: *CoRR* abs/1909.08053 (2019). arXiv: 1909.08053. URL: http://arxiv.org/abs/1909.08053.

[17] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).