Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Review:

Training large-scale models with limited GPU memory: a survey

Yu TANG[§], Linbo QIAO[†§], Lujia YIN[‡], Peng LIANG, Ao SHEN, Zhilin YANG, Lizhi ZHANG, Dongsheng LI[†‡]

National University of Defense Technology, Changsha 410073, China

[†]E-mail: qiao.linbo@nudt.edu.cn; dsli@nudt.edu.cn

Received Oct. 17, 2023; Revision accepted Mar. 31, 2024; Crosschecked

Abstract: Large-scale models have gained significant attention within a wide range of fields, such as computer vision and natural language processing, due to their effectiveness across various applications. However, a notable hurdle in training these large-scale models is the limited memory capacity of GPUs. In this paper, we present a comprehensive survey focused on training large-scale models with limited GPU memory. The exploration commences by scrutinizing the factors that contribute to the consumption of GPU memory during the training process, namely model parameters, model states, and model activations. Following this analysis, we present an in-depth overview of the relevant research work that addresses these aspects individually. Finally, the paper concludes by presenting an outlook on the future of memory optimization in training large-scale language models, emphasizing the necessity for continued research and innovation in this area. This survey serves as a valuable resource for researchers and practitioners keen on comprehending the challenges and advancements in training large-scale language models with limited GPU memory.

Key words: Training techniques; Memory optimization; Model parameters; Model states; Model activations
https://doi.org/10.1631/FITEE.2300710 **CLC number:**

1 Introduction

With the advent of deep learning (LeCun et al., 2015), neural networks witnessed significant advancements across diverse domains, such as speech recognition (Ze et al., 2013; Povey et al., 2011; Cho et al., 2014), computer vision (Krizhevsky et al., 2012; Ji et al., 2013; Ren et al., 2015), and natural language processing (Chowdhury, 2003; Sutskever et al., 2014; Dong et al., 2019). The landscape changed dramatically in 2017 with the introduction of transformer (Vaswani et al., 2017), which surpassed conventional neural network models in language-translation tasks, capturing widespread attention from various fields (Kitaev et al., 2020; Liu et al., 2021; Han et al., 2023). Large-scale models, such as BERT (Devlin et al., 2019), ERNIE (Sun et al., 2019), T5 (Raffel et al., 2020), and GPT-3 (Brown et al., 2020), which are generally built on the transformer model, have demonstrated state-of-the-art performance in a wide range of applications, including reading comprehension, question answering (Rajpurkar et al., 2016), and adversarial generations (Zellers et al., 2018). Recent studies have consistently shown that larger models with increased parameter sizes exhibit improved capacity and representation capabilities (Qiu et al., 2020). Consequently, there has been a rapid escalation in the size of these models, necessitating the need for memory-optimization techniques during their training (Sun et al., 2021; Zeng et al., 2021).

The exponential growth of model parameters places high demands for GPU memory. Unfortu-

§ These two authors contributed equally to this work
‡ Corresponding authors
ORCID: Yu TANG, https://orcid.org/0000-0002-8595-1547

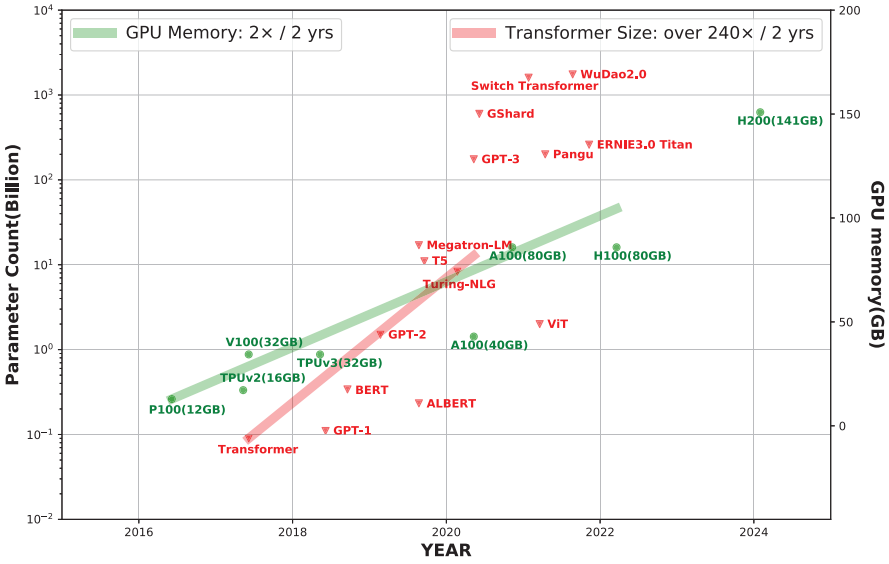

Fig. 1 The development of large-scale models' sizes and GPU memory capacity in recent years. It is obvious that the sizes of these models are increasing more and more rapidly and far beyond the capacity of GPU.

nately, the linear increase of GPU memory capacity occurring recently cannot meet the requirements of the exponential growth of model parameters. Therefore, the further advancement of large-scale models is substantially hindered by the limited GPU memory, i.e., the *GPU memory wall* problem (Gholami et al., 2021; Rajbhandari et al., 2021), as shown in Fig. 1, posing a significant impediment to progress in this domain. The demand for GPU memory remains a crucial factor when training large-scale models. For instance, training a model with more than one trillion parameters necessitates the usage of more than 180 A100 GPUs, each equipped with 80 GB of memory. Therefore, training large-scale models with limited GPU memory is a pressing research challenge that demands further advancements.

Training large-scale models with limited GPU memory has been an urgent issue to be solved. Liang et al. presented a survey that introduced auto parallelism in detail (Liang et al., 2022). They gave a comprehensive understanding and analysis of parallel and distributed training of deep neural networks(DNNs). Gusak et al. introduced the techniques of large-scale model training, including memory-optimization methods (Gusak et al., 2022). They summarized those main strategies contributing to training large-scale models. However, their comparison is not enough. Different from them, this paper presents a comprehensive analysis aimed at identifying the key factors that consume GPU memory during the training of DNNs and providing a survey about how to train large-scale models with limited GPU memory. Drawing upon our analysis, a thorough study of the prevalent techniques that are employed in training large-scale models with limited GPU memory is presented. This study delves into the significant memory consumption caused by model parameters, model states, and model activations, all of which impose considerable demands on memory resources while training large-scale models. To be more specific, methodologies that decrease the memory consumption of model parameters within one GPU include multi-GPU training, mixed-precision training, and specific model designs. For example, Model Parallel (MP) partitions the entire model into submodels along layers and executes each sub-model on each worker, such as Megatron-LM (Shoeybi et al., 2020) from NVIDIA. On the other hand, Pipeline Parallel shards the model along the depth and executes each submodel in a pipeline manner, such as GPipe(Huang et al., 2019), PipeDream (Narayanan et al., 2019), and

XPipe (Guan et al., 2019). Moreover, mixed parallel, consisting of more than two parallelism, could also decrease the memory consumption of model parameters, such as HetPipe (Park et al., 2020), DAPPLE (Fan et al., 2021), Chimera (Li and Hoefler, 2021), and 3D Parallel (Rasley et al., 2020; Lai et al., 2023). In terms of reducing memory consumption from model states, this paper introduces some optimizers that have achieved success in training large-scale models, such as zero redundancy optimizer (ZeRO) (Rajbhandari et al., 2020), PatrickStar (Fang et al., 2023), Adafactor (Shazeer and Stern, 2018) and CAME (Luo et al., 2023). Last but not the least, given the substantial memory of model activations, we focus on those methods that alleviate the memory overhead of model activations. Representative work includes rematerialization and swapping methods, such as dynamic tensor rematerialization (DTR) (Kirisame et al., 2021), Checkmate (Jain et al., 2020), virtualized Deep Neural Network (vDNN) (Rhu et al., 2016), ZeRO-Offload (Rajbhandari et al., 2020), and DELTA (Tang et al., 2022). Training large-scale models with limited GPU memory has not only been a research hit in academia but also draws plenty of interest in the industry. For example, Megatron-LM achieves training large-scale models with Data Parallel, MP, and Pipeline Parallel. Besides, training PanGu-α with 20B parameters adopts rematerialization and zero optimizers beyond 3D parallel. The detailed information of these works is concluded in Table 1, which is an elaborate summary for future reference. In the end, we also provide our concerns and discussion of future research about training large-scale models with limited GPU memory.

This paper is organized as follows. We begin by addressing the question of GPU memory allocation in the training process in Section 2. Next, we delve into the examination of memory consumption from three distinct perspectives, discussed from Section 3 to Section 5. In Section 3, we explore various methods that effectively decrease the memory usage associated with model parameters. Section 4 elaborates on techniques employed to minimize the memory footprint of model states during the training process of large-scale models. Section 5 focuses on those popular strategies to mitigate the memory overhead attributed to model activations. Section 6 highlights potential avenues for future research in the realm of memory optimization. Finally, Section 7 presents a comprehensive summary of the key findings and contributions of this paper.

2 GPU memory consumption analysis

Fig. 1 illustrates the growth trends of various DNN model sizes and GPU memory capacities since 2017. The red dots represent DNN model sizes, the green dots represent NVIDIA GPU memory capacities, and the blue dots represent domestic GPU memory capacities. The DNN model sizes show a rapid growth trend of approximately 240 times every 2 years, while GPU memory capacity has only increased from 12GB in the NVIDIA P100 GPU released in 2016 to 141GB in the NVIDIA H200 GPU in 2023. Although a single GPU could barely support mainstream model training before 2019, a turning point quickly emerged after 2020, and GPT-3 reached an astonishing parameter count of 175 billion, surpassing the memory capacity of the most advanced NVIDIA A100 GPU (Choquette et al., 2021) by over 10 times. Therefore, the distributed storage/parallel acceleration mode for large-scale model training is inevitable. However, this brings about the serious issue of GPU device communication. Over the past 20 years, the peak computational performance of GPUs has increased by 90,000 times, but the memory/hardware interconnect bandwidth has only increased by 30 times (Jia et al., 2018). Under such conditions, the scalability of distributed parallelism is greatly restricted. Therefore, the *GPU memory wall* problem significantly impedes the further advancement of deep learning. To address the GPU memory wall issue, how to manage GPU memory effectively becomes an urgent problem that needs to be solved. Motivated by this dilemma, our investigation focuses on analyzing the memory consumption throughout the training process and seeks to answer a fundamental question: which part consumes GPU memory most? Besides, there still lacks a comprehensive survey about how to train large-scale models with limited memory.

The memory allocation primarily arises from the substantial storage requirements of the model parameters, also known as model weights. In the case of a model containing N parameters, when transferred to the GPU in a floating-point 32 (FP32) format, a total of $4N$ GPU memory is consumed to store the

Table 1 A conclusion of training methodologies regarding M_p, M_s, and M_a.

Category	Methods	Information			
		Work	Authors	Year	Feature
Reducing memory of model parameters	Multi-GPU training	DistBelief	Dean et al.	2012	Model parallel
		Megatron-LM	Shoeybi et al.	2019	
		Expert Parallel	Fedus et al.	2021	
		GPipe	Huang et al.	2019	Pipeline parallel
		PipeDream	Narayanan et al.	2019	
		Xpipe	Guan et al.	2019	
		TorchGpipe	Kim et al.	2020	
		HetPipe	Park et al.	2020	Mixed parallel
		DAPPLE	Fan et al.	2021	
		Chimera	Li and Hoefler	2021	
		PipeTransformer	He et al.	2021	
		Hanayo	Liu et al.	2023	
		Sequence Parallel	Li et al.	2021	Sequence parallel
		MQSP	Zhong et al.	2022	
	Mixed Precision training	DOREFA	Zhou et al.	2016	-
		APEX	Micikevicius et al.	2018	-
		ActNN	Chen et al.	2021	-
	Specific design	L2L	Pudipeddi et al.	2020	-
		ReZero	Bachlechner et al.	2021	-
Reducing memory of model states	Optimizer	ZeRO	Rajbhandari et al.	2020	DeepSpeed
			Li et al.	2021	ColossalAI
			Zhao et al.	2023	PyTorch FSDP
		PatrickStar	Fang et al.	2022	-
		Adafactor	Shazeer and Stern	2018	-
		CAME	Luo et al.	2023	-
		DeepZero	Chen et al.	2023	-
Reducing memory of model activations	Rematerialization	Checkpoint	Chen et al.	2016	Static
		DTR	Kirisame et al.	2020	Dynamic
		Checkmate	Jain et al.	2020	Static
		Optimal Checkpoint	Beaumont et al.	2020	Static
		Rockmate	Zhao et al.	2023	Static
		Coop	Zhang et al.	2023	Dynamic
		Moccasin	Bartan et al.	2023	Static
	Swapping	vDNN	Rhu et al.	2016	Static
		SwapAdvisor	Huang et al.	2020	Static
		AutoTM	Hildebrand et al.	2020	Static
		FlashNeuron	Bae et al.	2021	Static
		ZeRO-Offload	Ren et al.	2021	Static
		ZeRO-Infinity	Rajbhandari et al.	2021	Dynamic
	Combination	SuperNeurons	Wang et al.	2018	Static
		Capuchin	Peng et al.	2020	Static
		DELTA	Tang et al.	2022	Dynamic
		RecShard	Sethi et al.	2022	Static
		Tsplit	Nie et al.	2022	Static

entire model in the GPU.

Another significant memory consumption stems from the model states during the training process. In the context of the SGD optimization algorithm (Amari, 1993), it is necessary to retain the gradients employed in the backward process, resulting in an additional $4N$ GPU memory allocation. Furthermore, for the Adam optimizer (Kingma and Ba, 2015), the memory allocation expands to include the storage of gradient variances, requiring an additional

$4N$ GPU memory. Moreover, both the momentum of gradients and variances necessitate two separate $4N$ memory allocations. Consequently, the Adam optimizer requires a total of $16N$ GPU memory for the training process. A similar memory-allocation pattern can be observed for AdamW (Zhuang et al., 2022).

In addition to parameters and model states, a significant portion of GPU memory is consumed by the training activations. Assuming that b is the input batch size and s is the sequence length, for a Transformer model of l layers, assuming its hidden size is h and the vocab size is V, its parameters counts is about $l(12h^2 + 13h) + Vh$. When h is large enough, the parameters are about $12lh^2$. For these activations, M_a is counted by $(34bsh+5bs^2a)l$, where a is the number of the heads in the model (Korthikanti et al., 2023). M_a for GPT-4 (OpenAI et al., 2024) is represented as "-" because the settings is not public. In Table 2, we compute some typical large-scale models together with their parameters and the parameters' memory consumption M_p, and model states' memory consumption M_s on the condition that $b = 16$ and $s = 128$ and utilizing the Adam optimizer. With the Adam optimizer and mixed-precision training, M_s is nearly $3\times$ compared to M_p. We notice that GPT-3 will produce about 366.00 GB activations. However, GPT-3 is trained with a longer sequence 2048 and a large batch size such as 3.2M (Brown et al., 2020). Therefore, the memory consumption of activations in GPT-3 is much more than 366.00 GB. For example, Li et al. tried to train large-scale models with a sequence length of more than 2k, which will cost a large number of activations. With the increased batch size and sequence length, the memory consumption of model activations will also increase at a ratio.

Ideally, M_s accounts for the primary overhead, followed by M_p, and finally M_a. However, in non-ideal scenarios where only multi-GPU parallelism is available, M_s effectively becomes a portion easily amortized by optimizer parallel methods. Next is M_p, which can be efficiently partitioned across multiple GPUs through tensor parallelism to distribute tensor operator parameters within the model. The most challenging to optimize is M_a. Although using heterogeneous system parallelism and sequence parallelism methods can be effective, adopting data parallelism for acceleration can result in linear batch growth, significantly expanding M_a. Additionally, pipeline parallelism and mixed expert parallelism optimize all three types of memory consumption by allocating model layers to different GPUs, but the communication overhead for passing activations between GPUs is substantial. According to Amdahl's Law (Gustafson, 1988), the performance of parallel systems is determined by both the serial and parallel portions, where the sequence serial refers to the part that cannot be parallelized. This law also applies to the performance analysis of distributed parallel systems. For three different types of memory consumption, optimizing one type of memory usage on a single GPU to its limit using a certain parallel method becomes a serial portion that cannot be further parallelized. Therefore, it is unreasonable to emphasize M_s overly and underestimate M_a purely based on numerical values, because M_a is inherently more challenging to parallelize compared to M_s.

In summary, model parameters, model states, and model activations are three key factors consuming GPU memory the most. There are various ways to reduce the memory consumption of model parameters, model states, and model activations. Then, we will discuss how to train large-scale models with limited GPU memory by reducing memory consumption of model parameters(Section 3), model states(Section 4), and model activations(Section 5), respectively. For example, researchers usually adopt multi-GPU training, and mixed-precision training to reduce the memory consumption of model parameters. Besides, they also design some specific models to reduce M_p. As for reducing M_s, researchers usually focus on optimizers and reduce the memory consumption of gradients. Last but not the least, M_a is reduced by rematerialization and swapping, which is introduced in detail in Section 5.

3 Reducing memory of model parameters

As the number of parameters grows, the efficient training of large-scale models emerges as a critical challenge that remains unresolved. Model parameters result in severe memory redundancy, especially in Data Parallel because the whole model parameters are replicated across all the workers. To overcome the memory redundancy of model parameters, assignment and reduction could be used. To be more

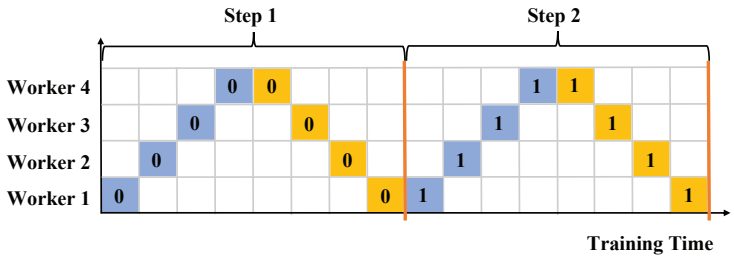

Fig. 2 MP. The entire model is split onto different GPUs and the training data batch flows from worker1 to worker4, which causes severe bubbles in the training process.

specific, assignment means sharding these model parameters and assigning them to different GPUs, relieving the memory pressure within one GPU. On the other hand, reduction stands for reducing the parameter count while keeping the performance of the model at the same time. In this section, we mainly introduce training large-scale models by decreasing the memory consumption from these two kinds of methods.

3.1 Multi-GPU training

Addressing the memory utilization associated with model parameters within a single GPU, distributed and parallel training try to assign the model parameter to different GPUs and have proven pivotal in mitigating parameter consumption. In this section, we mainly introduce distributed and parallel training which shards model parameters on multiple GPUs.

Before introducing distributed and parallel training which reduces the memory redundancy of model parameters, we introduce Data Parallel, which causes plenty of memory redundancy because of the model replica on each worker. To reduce the memory redundancy of training the whole dataset, Data Parallel splits the training samples into subsets and trains these subsets on each worker. This approach effectively mitigates memory redundancy by distributing the computational load across workers and reducing the burden associated with model parameter storage. Each worker independently engages in a training process driven by stochastic gradient algorithms. Data Parallel is commonly implemented using two architectural approaches: the centralized architecture and the decentralized architecture. In the centralized architecture, such as Parameter Server (Li et al., 2014), a central worker acts as the hub for sending and receiving gradients computed by each worker. The decentralized architecture, exemplified by Horovod (Sergeev and Del Balso, 2018), operates without a central entity. Each node in Horovod serves as both a server and a worker simultaneously, and communication is limited to neighboring workers. PyTorch Distributed Data

Table 2 Typical large-scale models and their parameter count together with M_p, M_s, and M_a.

Model	Parameters(B)	M_p	M_s	M_a
BERT-Large	0.34	1.36	4.08	8.25
T5-Small	0.06	0.24	0.72	0.62
T5-Base	0.22	0.88	2.64	6.18
T5-Large	0.7	2.8	8.4	16.50
T5-3B	3	12	36	20.25
T5-11B	11	44	132	42.75
Turing-NLG	17	68	204	96.78
GPT-3 6.7B	6.67	26.68	80.04	39.00
GPT-3 13B	12.85	51.4	154.2	61.15
GPT-3 175B	174.6	650.44	1951.32	366.00
GPT-4	≥1000	≥4000	≥12000	-

Parallel (PyTorch DDP) launches multi-progress as workers. Each worker stores a replica of the whole model (Li et al., 2020). They use AllReduce across these workers to synchronize the gradients. Data Parallel involves duplicating the whole model in each worker, leading to memory redundancy from the model, particularly for large-scale language models. The substantial number of parameters significantly occupies a significant portion of GPU memory. For example, BERT-LARGE with 340 million parameters requires approximately 5.07 GB of GPU memory per GPU to store these parameters, and this does not account for the memory consumed by activations during the training process.

To decrease model parameters within one single GPU, the parallel algorithms assign these model parameters to different GPUs, such as MP and Pipeline Parallel, which reduce the memory consumption of model parameters within one GPU and have been widely adopted to alleviate the burden on model parameter-memory consumption.

3.1.1 Model parallel

MP is a widely utilized strategy in deep learning, which partitions the model into submodels and places each submodel on the GPU, such as DistBelief (Dean et al., 2012). It offers an alternative approach to mitigate memory redundancy within one GPU compared to Data Parallel. In MP, the model parameters are also distributed among different workers. Each worker is responsible for sending and receiving activations instead of gradients, focusing solely on updating its assigned parameters.

Tensor MP, a specific form of Model Parallel, encompasses Column Parallel and Row Parallel techniques. A notable example of tensor MP is showcased in the work of Megatron-LM (Shoeybi et al., 2020), where the MLP and self-attention modules within the transformer layers are divided using Column Parallel and Row Parallel strategies. Synchronization of data within the matrices is achieved through the utilization of AllReduce to ensure convergence.

Expert Parallel has been developed in the Mixture of Expert (MoE) models (Rajbhandari et al., 2022; Fedus et al., 2021). Expert Parallel is specifically designed for MoE models, where computation cores are allocated to handle specific experts. In essence, Expert Parallel can be considered as a variant of MP, but it can also be utilized in conjunction with data parallel and MP simultaneously. Furthermore, Fedus et al. have adopted a combination of parallel methods, including Data Parallel, MP, and Expert Parallel, to tackle models with over a trillion parameters(Fedus et al., 2021). In this setting, Data Parallel and MP operate conventionally, while each expert is assigned to a computation core in Expert Parallel, handling the corresponding allocated data efficiently.

It is well known that designing and implementing MP algorithms pose considerable challenges and MP faces limitations in terms of scalability and poses greater implementation challenges compared to Data Parallel. Moreover, achieving a suitable balance between model scaling capacity, flexibility, and efficiency remains a complex task. As a result, MP algorithms tend to be architecture-specific or task-specific, limiting their generalizability.

3.1.2 Pipeline Parallel

GPipe (Huang et al., 2019), brought out by Google, primarily focuses on Pipeline Parallel, as depicted in Fig. 3. GPipe overcomes these limitations by offering a flexible and training-efficient library. Initially, GPipe partitions the model into $\mathcal{N}$ parts and assigns each part to a specific accelerator. Based on this partitioning, GPipe further divides the training batch $\mathcal{B}$ into $\mathcal{M}$ micro-batches, which are pipelined across the $\mathcal{N}$ accelerators. During the back-propagation process, the gradients for each micro-batch are computed using the same model parameters employed during the forward pass. Ultimately, the gradients from all $\mathcal{M}$ micro-batches are accumulated and utilized to update the model parameters in the current mini-batch across all accelerators. An implementation of Gpipe on PyTorch is achieved on TorchGpipe (Kim et al., 2020). There exist severe bubbles in GPipe, as shown in Fig. 3.

Fig. 4 shows the architecture of PipeDream (Narayanan et al., 2019), which addresses the bubble problem present in GPipe by employing a "1F1B" (one forward one backward) strategy. This strategy ensures that the backward computation of each stage commences immediately after the completion of the forward stage. By reducing bubble overhead, PipeDream optimizes the utilization of GPU memory. The "1F1B" strategy is commonly used in recent pipeline paralleled works. Moreover, PipeDream maintains all parameters

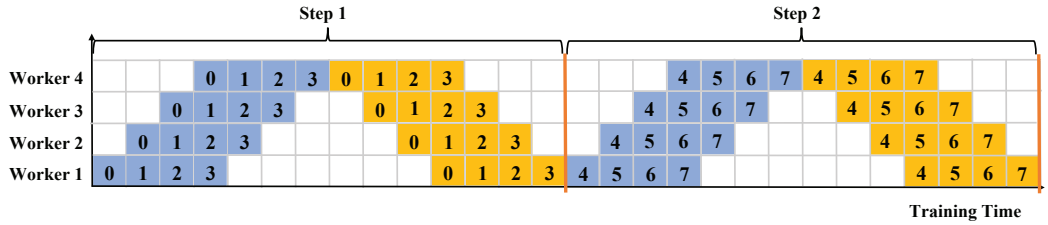

Fig. 3 GPipe. GPipe executes training DNNs in a pipeline. However, the bubbles result in low efficiency of GPU.

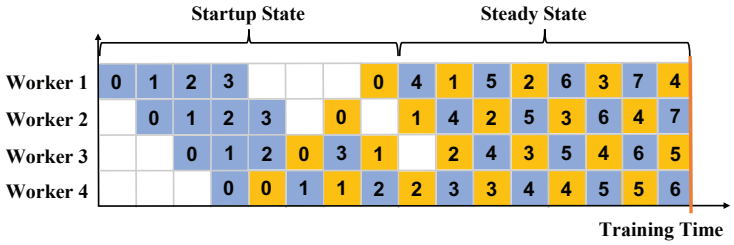

Fig. 4 PipeDream. PipeDream adopts the "1F1B" strategy and reduces the bubbles in GPipe. The blue blocks are the forward process in each worker and the orange ones are the corresponding backward process.

assigned to the stage in GPU memory, which reduces the memory consumption from parameters within one single GPU. In GPipe, the forward process and the backward process for one micro batch use different versions of parameters at each stage. However, in PipeDream, the weights for new micro batches will not update until all the micro batches' weights update before it, which incurs inconvergence. To solve this problem, PipeDream proposed weight stashing to avoid the mismatch between the version of weights. To be more specific, weight stashing keeps multiple versions of weights for every minibatch in GPU. PipeDream also supports checkpointing (Chen et al., 2016) on each stage for fault tolerance. PipeDream gains a comparable accuracy result and a better speedup compared to DP.

Guan et al. proposed XPipe (Guan et al., 2019) as an approach to enhance the efficiency of Gpipe and simultaneously tackle the challenges posed by synchronous training and asynchronous model training. PipeDream, on the other hand, utilizes different versions of model weights during iterations, resulting in memory redundancy. To address this issue, XPipe employs weight sharing within the complete mini-batch while managing weight prediction for each mini-batch. Furthermore, to alleviate memory redundancy, they introduce the concept of "weight difference" as a measure of the number of weight updates.

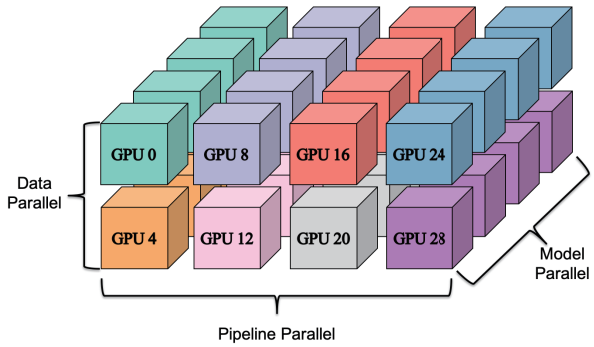

Fig. 5 Overview of 3D Parallel. 3D Parallel consists of Data Parallel, MP, and Pipeline Parallel. These parallel dimensions shards the GPUs as their degrees in the training system.

3.1.3 Sequence parallel

For transformer-based language models, their input is typically a sequence. However, for a long input sequence, the time and space complexity of the self-attention mechanism increases quadratically, leading to increased memory overhead. In certain specific scenarios, such as medical image processing, the input sequence length may be too large for the GPU's memory to meet training requirements. Sequence Parallel divides the input sequence into multiple chunks, with each subchunk assigned to a

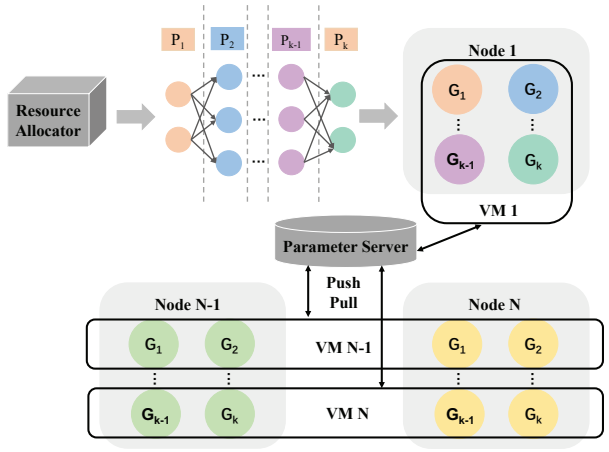

Fig. 6 The HetPipe architecture. Data parallelism is executed among each node, a.k.a. *virtual worker*. The pipeline parallelism is executed from node to node.

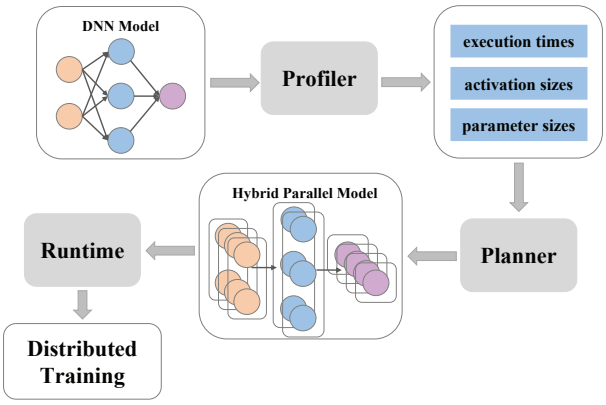

Fig. 7 The DAPPLE architecture. The Profiler takes the DNN model as input and profiles the execution time, activations, and parameter sizes for the Planner. Then the Planner finds the hybrid parallelism for this model through dynamic programming. The Runtime launches distributed training based on the hybrid parallel model.

GPU, and each GPU retains only a subsequence of the complete sequence (Li et al., 2022). To enable the self-attention mechanism to be applied to different chunks, they designed the ring self-attention algorithm.

Based on Sequence Parallel, Zhong et al. proposed MQSP to solve the difficulty for Transformer-based models of scaling up to a long sequence (Zhong et al., 2023). They also split the input sequence into several sub-sequences. They also project the local queries, keys, and values in self-attention. Different from the vanilla Sequence Parallel, they design a distributed self-attention by synchronizing the queries across all the devices. A distributed softmax is also used, which incurs negligible communication. They designed Micro-Q, which means the subqueries split from the local query, ensuring the linear scaling of the input sequence to a long sequence.

3.1.4 Mixed Parallel

Mixed Parallel is a collective term encompassing the combination of Data Parallel, MP, and Pipeline Parallel techniques. It introduces a novel paradigm known as 3D Parallel, where three axes, namely the x axis (Pipeline Parallel), the y axis (MP), and the z axis (Data Parallel), are represented as depicted in Fig. 5. This section primarily focuses on methods that leverage mixed parallelism.

HetPipe (Park et al., 2020) is a training framework that combines pipeline parallelism and data parallelism for heterogeneous GPU clusters, as illustrated in Fig. 6. In the training system, data parallelism is facilitated through the concept of a virtual worker (VM), which comprises multiple GPUs. The virtual worker allows for data parallelism by aggregating resources from multiple GPUs, even when individual GPUs may be resource-limited. Furthermore, each virtual worker processes each mini-batch based on pipeline parallelism, maximizing GPU utilization. HetPipe also introduces the "Wave Synchronous Parallel" model to synchronize model weights and ensure model convergence. A "wave" represents a sequence of mini-batches processed simultaneously within a virtual worker. Instead of transmitting weight updates for every mini-batch, the virtual worker aggregates and pushes the accumulated weight updates in a wave to the parameter server, as shown in Fig. 6.

DAPPLE (Fan et al., 2021) is another synchronous training method that integrates pipeline parallelism and data parallelism. DAPPLE consists of three components: Profiler, Planner, and Runtime, as depicted in Fig. 7. The Profiler takes the DNN model as input and profiles information such as the execution time, activations, and the parameters for each layer. The Planner aims to minimize pipeline latency between pipeline stages, leveraging dynamic programming to find optimal solutions, including the device assignment. The Runtime constructs forward and backward graphs for each pipeline stage, inserting *split* and *concat* operations for activation communications between adjacent stages. The Runtime also builds a subgraph for

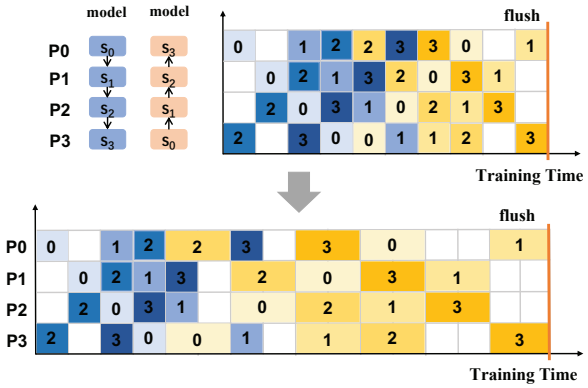

Fig. 8 The Chimera architecture. Chimera incorporates two pipeline directions which are in the opposite order.

synchronous training. To optimize GPU memory usage, backward stages are scheduled earlier to release activations. In addition, instead of injecting all M micro-batches at once, only $K < M$ micro-batches are inserted initially. After a backward stage finishes, a new forward stage is scheduled promptly to enable earlier backward stage execution.

Chimera (Li and Hoefler, 2021) is a synchronous method that combines data parallelism and pipeline parallelism, as depicted in Fig. 8. Different from DAPPLE, Chimera incorporates two pipelines in different directions, referred to as the *down* pipeline and the *up* pipeline. These pipelines are mapped in the opposite order. For M micro-batches, each pipeline schedules $M/2$ micro-batches using a 1F1B strategy. This approach effectively reduces the number of bubbles to $S/2-1$ during the forward and backward processes, where S represents the number of pipeline stages. However, Chimera increases the training efficiency at a cost of memory because each worker maintains two copies of submodels.

PipeTransformer (He et al., 2021) introduces an elastic Pipeline Parallel training framework specifically designed for transformer models. It consists of four modules: Freeze Algorithm, AutoPipe, AutoDP, and AutoCache. The Freeze Algorithm dynamically selects layers to freeze over different iterations adaptively. AutoPipe optimizes the allocation of active layers across GPUs, dynamically partitioning the training pipeline to minimize the number of pipeline devices. It employs a greedy algorithm to allocate all frozen and active layers across K GPU devices. AutoDP spawns pipeline replicas on unused GPUs to increase the degree of data parallelism. AutoCache shares activations across all Data Parallel processes and automatically replaces stale caches.

Liu et al. analyzed the drawbacks of Chimera and DAPPLE, resulting in a high bubble rate and communication overhead. Then they introduced a wave-like pipeline scheme named Hanayo(Liu et al., 2023). Instead of training a complete pipeline on all devices in one direction, they performed a direction reversal in the middle for either of the bidirectional pipelines. Through this direction reversal, the original bidirectional pipeline transformed into two single-directional pipelines in a wave-like fashion. In this pattern, Hanayo employed the same model replication and obtained at least as good as, if not as good as Chimera.

In addition to mixed parallelism, there has been significant research on automatic parallelism, aiming to identify the optimal parallel algorithm given large-scale models and the device mesh (Zheng et al., 2022; Unger et al., 2022; Jia et al., 2019; Yuan et al., 2021; Lin et al., 2023; Chen et al., 2023). However, these topics are beyond the scope of this paper, and readers can refer to the survey by Liang et al. for more details. Furthermore, addressing the dilemma of overlapping communication and computation to accelerate the training of large-scale models remains an ongoing challenge.

3.2 Mixed-precision training

Mixed-precision training, such as those introduced by NVIDIA APEX (Micikevicius et al., 2018) have been developed to train DNNs using half-precision floating point numbers. By utilizing lower precision, the memory and computing requirements of the training process can be significantly reduced. Additionally, researchers present several key ideas to enable 8-bit floating point (FP8) training including (a) Novel Hybrid FP8 formats to represent weights, activations, and gradients (b) chunk-based hierarchical accumulations to minimize low-precision accumulation errors and (c) selective precision rules (for first, last and depthwise convolutional layers) (Wang et al., 2018b; Sun et al., 2019). Banner et al. presented low-precision training results with 8-bit fixed-point integers for the forward phase as well as a subset of the computations in the backward phase. Sun et al. explores a novel adaptive Gradient Scaling technique (GradScale) that addresses the challenges of insufficient range and resolution in quantized gra-

dients and trains DNNs using a 4-bit floating point. Xi et al. propose a training method for transformers with all matrix multiplications implemented with the INT4 arithmetic and the numerical formats are supported by contemporary hardware. Furthermore, Ma et al. propose a layer-wise mixed-precision strategy that optimizes the training process without compromising convergence. Although mixed-precision training typically involves maintaining an additional copy of weights, resulting in increased memory requirements, the overall impact on memory usage is relatively small due to the low-precision representation of activations (which dominate the training memory consumption).

Quantized training (Chen et al., 2020; Zhou et al., 2018; Zhang et al., 2018) aims to reduce computational costs by performing quantization during inference or training. As a beneficial side effect, quantization can also reduce the memory footprint during training. During training, all layers' activations must be stored in memory to compute gradients. Activation compressed training techniques (Chen et al., 2021; Fu et al., 2020) achieve memory savings by compressing activations to lower numerical precision through quantization. Moreover, weight-compression techniques (Han et al., 2016) and gradient-compression techniques (Lin et al., 2018) compress weights and gradients, respectively, to reduce storage and communication overhead.

3.3 Specific model designs

Specific designs for the transformer architecture have been proposed to effectively reduce the memory footprint resulting from model parameters. In this section, we present some notable works related to this field.

In 2020, Pudipeddi et al. trained large-scale language models using a new execution algorithm that achieved constant memory consumption. As large-scale language models are built upon numerical transformer models, they introduced the Layer-to-Layer (L2L) algorithm for training large-scale models on a single device. Instead of keeping the entire encoder-decoder architecture on the GPU, they introduced a host called "Eager Param-Server" (EPS). The weights and gradients are managed by *EPS*. Except for the embedding layer and the output layer, only one encoder-decoder layer stays in the device. This layer gets weights from *EPS* and outputs gradients to *EPS*. Therefore, the device only stores three layers.

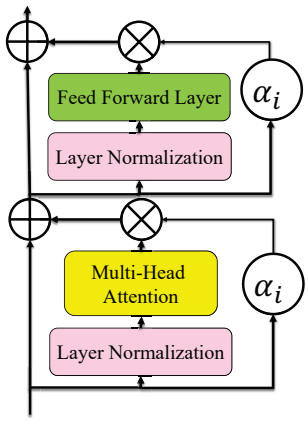

Fig. 9 The architecture of the ReZero transformer. The parameter α determines whether the layer is calculated in the process.

Another significant work related to specific design is ReZero (Bachlechner et al., 2021). Drawing inspiration from the residual connection (He et al., 2016) in the transformer architecture, they paid attention to the residual connection in the transformer architecture. First of all, they represent the transformation as a function $F(x)$. Then they introduce a residual connection for the input x and another parameter $\alpha \in [0, 1]$ which adjusts the transformation layer $F(x)$, which is represented as Eq. 1.

$$x_{i+1} = x_i + \alpha_i F(x_i). \tag{1}$$

The parameter α_i is initialized as 0 and is trainable from iteration to iteration. The ReZero transformer architecture is shown in Fig. 9. When $\alpha_i = 0$, the $i-$th layer is effectively bypassed, resulting in a reduction in model parameters. Comparisons between ReZero and the vanilla transformer, as well as Pre-LN transformer (Xiong et al., 2020), validate that ReZero enables training with deeper transformer models with saved GPU memory. The experimental results verify that ReZero makes it possible to train a deeper transformer model.

These specific designs rely on researchers' domain expertise regarding model architectures. By leveraging this knowledge, these methods can be developed and applied effectively.

4 Reducing memory of model states

In general, model states include the optimizer states, gradients, and parameters (Rajbhan-

dari et al., 2020). In this paper, the model states only include the optimizer states and the gradients because we introduce those methods reducing the memory consumption of model parameters. Therefore, in this section, we mainly introduce how to overcome the memory redundancy of model states.

4.1 ZeRO

One of the major challenges in training large-scale models is the significant memory requirement due to the vast number of model states. Loading such models into GPU memory is highly demanding, even more so considering the computational aspects. For instance, training GPT-3 with 175 billion parameters requires approximately 2.1TB of GPU memory for the model states when employing mixed-precision training together with Adam.

Rajbhandari et al. identified two main components consuming memory during the training process: model state and residual state. The model state includes optimizer states, gradients, and parameters. Taking the Adam optimizer as an example, storing the momentum and variance of each gradient, in addition to the model parameters, incurs an additional $12\times$ memory cost through mixed-precision training. To alleviate this memory limitation, ZeRO introduces a key insight: partitioning optimizer states, gradients, and parameters specifically designed for data parallelism. In the case of a data-parallel degree of N, the optimizer states, gradients, and parameters are partitioned into N groups, with each GPU or process responsible for storing and updating $1/N$ of these values. This partitioning strategy reduces the memory redundancy associated with data parallelism.

The residual state-consuming parts consist of activations, commonly observed in model parallelism, temporary buffers, and memory fragmentation. For activations checkpointing, ZeRO also employs a partitioning approach. In model parallelism, the redundancy arises from repeated activations checkpointing. Hence, ZeRO eliminates this redundancy by partitioning and materializing activations checkpointing in each activation layer at a specific time. After performing the forward propagation within the model, the input activations are partitioned across all GPUs in the system or MP processes. ZeRO also allocates a sufficiently large buffer of constant size for temporary buffers

Algorithm 1 Adam Optimizer.

Input: model parameters θ, loss function f, moment estimates $\beta_1 \in [0,1)$ and $\beta_2 \in [0,1)$, learning rate η, regularization constant ϵ, training iteration T.
Output: θ.

Initialize $m_0 = 0$, $v_0 = 0$;
for $t = 1, \cdots, T$ **do**
  $g_t \leftarrow \nabla f(\theta_{t-1})$
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$
  $\hat{m}_t \leftarrow m_t/(1-\beta_1^t)$
  $\hat{v}_t \leftarrow v_t/(1-\beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \eta\hat{m}_t/(\sqrt{\hat{v}_t}+\epsilon)$
end for
return θ.

during the training process to minimize frequent memory access. Additionally, ZeRO tackles memory fragmentation by allocating contiguous memory chunks for activations checkpointing and temporary gradients during the back-propagation process and promptly managing them when produced. There are three stages in ZeRO, namely, ZeRO-1, ZeRO-2, and ZeRO-3. ZeRO-1 partitions the optimizer states, and ZeRO-2 partitions gradients. ZeRO-3 partitions model parameters to reduce memory redundancy. As for the communication overhead in ZeRo, ZeRO-1 and ZeRO-2 share the same communication volume as vanilla Data Parallel. However, ZeRO-3 costs $1.5\times$ communication overhead as those shared parameters are supposed to be gathered.

ZeRO has garnered significant interest and has been utilized in various works. For instance, Microsoft's DeepSpeed (Rasley et al., 2020), a distributed framework, enables researchers to train large-scale models of arbitrary sizes. DeepSpeed has been successfully employed in various projects, including the CPM model (Zhang et al., 2021). Besides, Li et al. introduced ColossalAI (Li et al., 2023), which incorporates the ZeRO optimizer into their framework. The PyTorch team has also implemented PyTorch Fully Shard Data Parallel (Zhao et al., 2023b), offering an alternative implementation of the ZeRO optimizer.

4.2 PatrickStar

Based on ZeRO, Fang et al. proposed a chunk-based memory management for training large-scale models. In their opinion, static sharding of model

Algorithm 2 Adafactor Optimizer.

Input: model parameters θ, loss function f, moment estimates $\beta \in [0, 1)$, learning rate η, regularization constant ϵ, training iteration T.

Output: θ.

Initialize $v_0 = 0$, $u_0 = 0$;
for $t = 1, \cdots, T$ **do**
  $g_t \leftarrow \nabla f(\theta_{t-1})$
  $\beta_t = 1 - 1/t$
  $v_t \leftarrow \beta_t \cdot v_{t-1} + (1 - \beta_t)(g_t^2 + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)\mathbf{1}_m$
  $u_t \leftarrow \beta_t \cdot u_{t-1} + (1 - \beta_t)\mathbf{1}_n^T(g_t^2 + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)$
  $\hat{w_t} \leftarrow (v_t u_t / \mathbf{1}_n^T)/v_t$
  $z_t = g_t/\sqrt{w_t}$
  $\eta_t = \max(\epsilon_2, RMS(\theta_{t-1}))\rho_t$
  $\theta_t \leftarrow \theta_{t-1} - \eta_t z_t$
end for
return θ.

states lacks the tolerance for training large-scale models without sufficient GPU or CPU memory. They proposed PatrickStar (Fang et al., 2023) which manages the fine-grained model states to make full use of heterogeneous memory. They put those fine-grained model states into chunks of the same size. During the training process, PatrickStar organizes the distribution in heterogeneous storage according to the tensor states dynamically. They also use *warm up iteration* to collect the information of model states in GPU and designed an efficient chunk-recovery strategy to reduce the communication between CPU and GPU. Experimental results show that PatrickStar achieves better efficiency compared to DeepSpeed and lower memory consumption.

4.3 Adafactor

In addition to ZeRO, there are other approaches aimed at reducing model states to gain memory optimization. Adafactor (Shazeer and Stern, 2018) is one such method. When training large-scale models using second-order estimators like Adam and AdamW, a significant amount of GPU memory is required. The algorithm for Adam can be found in Algorithm 1. To mitigate the memory footprint associated with these second-order estimators, Shazeer and Stern (2018) proposed constructing a low-rank approximation of the exponentially smoothed accumulator. By doing so, the memory requirement is reduced from $O(mn)$ to $O(m + n)$ for a matrix of size $M \times N$. The details of the Adafactor algorithm can be found in Algorithm 2. In Algorithm 2 $RMS(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x^2}$. Similar to optimizers like Adam, Adafactor uses a second-moment estimator (similar to the moving average of squared gradients) to adjust the learning rates adaptively. This helps in handling the geometry of the optimization landscape. In contrast to Adam, Adafactor employs adaptive learning rates for each parameter in the model, which means that the learning rate is adjusted individually for each parameter during training, allowing for more efficient and effective updates. Besides, Adafactor is designed to be memory-efficient, which is crucial for large-scale models and datasets. It uses a factored (or low-rank) approximation of the second-moment matrix, reducing the memory requirements compared to full-matrix methods. Adafactor does not require the first-order estimation of gradients to be maintained. It is important to note that Adafactor does not utilize momentum information (Cutkosky and Mehta, 2020; Sutskever et al., 2013), causing unstable convergence regarding training large-scale models. Like many adaptive learning rate algorithms, Adafactor's performance can be sensitive to the choice of hyperparameters. Suboptimal hyperparameter settings may lead to issues such as slow convergence or instability during training.

4.4 CAME

It is worth noting that the second-order estimator of the gradients puts on extra memory consumption in the training process, such as Adam (Kingma and Ba, 2015) and LAMB (You et al., 2020). On the other hand, Adafactor results in a decline in the model performance. Luo et al. proposed a confidence-guided strategy, CAME, which improves the robustness of Adafactor and gains an adaptive memory optimization (Luo et al., 2023). The algorithm of the CAME optimizer is displayed in Algorithm 3. In more detail, CAME utilizes a confidence measure to guide the sampling process during optimization. This confidence measure helps the algorithm focus on promising optimal solutions. Moreover, Luo et al. developed an adaptive memory mechanism with faster convergence. This memory-efficient approach helps avoid redundant evaluations of the objective function and speeds up the opti-

Algorithm 3 CAME Optimizer.

Input: model parameters θ, loss function f, moment estimates $\beta \in [0, 1)$, learning rate η, regularization constant ϵ, training iteration T.

Output: θ.

1: Initialize $v_0 = 0$, $u_0 = 0$;
2: **for** $t = 1, \cdots, T$ **do**
3: $g_t \leftarrow \nabla f(\theta_{t-1})$
4: $\beta_t = 1 - 1/t$
5: $v_t \leftarrow \beta_t \cdot v_{t-1} + (1 - \beta_t)(g_t^2 + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)\mathbf{1}_m$
6: $u_t \leftarrow \beta_t \cdot u_{t-1} + (1 - \beta_t)\mathbf{1}_n^T(g_t^2 + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)$
7: $\hat{w}_t \leftarrow (v_t u_t / \mathbf{1}_n^T)/v_t$
8: $z_t = g_t / \sqrt{w_t}$
9: $\eta_t = \max(\epsilon_2, RMS(\theta_{t-1}))\rho_t$
10: $m_t = \beta_1 m_{t-1} + (1 - \beta_1)\eta_t$
11: $U_t = (\eta_t - m_t)^2$
12: $R_t = \beta_3 R_{t-1} + (1 - \beta_3)(U_t + \epsilon_2 \mathbf{1}_n \mathbf{1}_m^T)$
13: $C_t = \beta_3 C_{t-1} + (1 - \beta_3)\mathbf{1}_n^T(U_t + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)$
14: $S_t = R_t C_t / \mathbf{1}_n^T R_t$
15: $\theta_t \leftarrow \theta_{t-1} - \eta / \sqrt{S_t} m_t$
16: **end for**
17: **return** θ.

mization process. Line 11 to line 14 are the pseudocodes of the memory-efficient mechanism computation. The CAME optimizer corrects the update amount based on the updated confidence of the model and performs non-negative matrix decomposition on the introduced confidence matrix. Therefore, it obtains a comparable performance with Adam. The confidence-guided sampling and adaptive memory mechanisms also contribute to its ability to converge faster.

4.5 DeepZero

First-order or second-order optimizers rely on gradients computed during the training process. When obtaining first-order gradient information is difficult, zero-order optimization becomes a commonly used method in such scenarios (Liu et al., 2018). Compared to first-order optimization, zero-order optimization has the advantage of not requiring explicit gradient calculation; instead, it utilizes finite differences in function values to estimate gradients. DeepZero integrates zero-order optimization, model pruning, and parallel computing techniques(Chen et al., 2024). However, zero-order optimization is not yet widely applied in large-scale models, and further research is still needed to explore its potential.

Those optimizers reduce memory consumption resulting from model states. They focus on the gradients and their variance. However, how to reduce memory consumption and keep the performance and convergence at the same time remains a problem, especially for theoretical analysis.

5 Reducing memory of model activations

Model activations also play a crucial role in computing gradients during the backpropagation process. However, they consume a significant amount of GPU memory if staying in GPU, which poses a great challenge for efficient memory usage. Model activations also increase as the batch size or the sequence length increases. To address this issue, several methodologies have been proposed to reduce the memory consumption of training activations. In this section, we introduce two prominent approaches: rematerialization and swapping. In this section, fine-grained memory-optimization techniques refer to tensor-wise operations, whereas coarse-grained memory-optimization techniques pertain to operations on the model layer.

Table 3 Actions in reducing model activations.

Action	Meaning
Allocation	Allocate GPU memory for tensors
Eviction	Evict layers/tensors from GPU
Rematerialization	Recompute layers/tensors back to GPU
Offload	Move layers/tensors from GPU to CPU
Reload	Move layers/tensors from CPU to GPU
Swap	Including Offload and Reload
Release	Including Rematerialization and Swap

5.1 Rematerialization

Rematerialization methods involve evicting layers or evicting tensors during the forward pass of training DNNs and recomputing them during the backward pass, which is also named checkpointing (Chen et al., 2016). The detailed rematerialization process can be found in Fig. 10.

In the pioneering work of Chen et al. (2016), they introduced checkpointing every $\sqrt{n}$ layers during the forward pass to reduce the memory consumption of activations, where n is the number of model layers (Chen et al., 2016), as shown in Fig. 10.

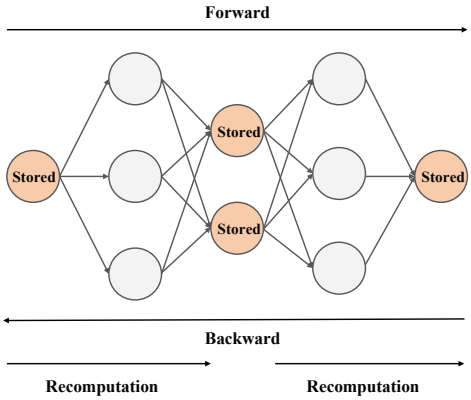

Fig. 10 The rematerialization process. The gray circles represent evicted activations and the rest are stored in GPU in the forward process. In the backward process, those evicted activations are rematerialized from the stored activations. It saves memory consumption at the cost of rematerialization time.

This approach resulted in sublinear memory cost, enabling the training of larger and deeper models. However, it had limitations as it required the manual selection of checkpointed layers and was not applicable to non-linear models like ResNet (He et al., 2016).

To address the limitations of Chen et al.'s work, Kirisame et al. designed DTR (Kirisame et al., 2021) based on PyTorch (Paszke et al., 2019). DTR is a fine-grained method that uses heuristic functions to determine the optimal tensors to evict. For each tensor t, the staleness time is $s(t)$ and is computed based on the time it has not been used. The memory consumption of tensor t is $m(t)$. The heuristic function in DTR considers the recomputation cost of a tensor, the recomputation cost of its parent tensors, and the tensor's memory consumption and staleness time. In DTR, the only action for reducing the activation redundancy is eviction. The recomputation cost for tensor t is $c_r(t)$. Besides, the recomputation cost of the parent tensors, t_p, of tensor t should also be taken into consideration. Therefore, the heuristic function of DTR is

$$h_{DTR} = \frac{c_r(t) + \sum_{t_p \in e(t)} c_r(t_p)}{m(t)s(t)}, \tag{2}$$

where $e(t)$ is the set of the parent tensors of tensor t. There are also other heuristic functions except Eq. 2, such as *least recently used* (LRU)(Lee et al., 1999), and *Greedy*. The tensor with a small value of the heuristic function is chosen for eviction.

Different from DTR, Checkmate (Jain et al., 2020) was constructed on TensorFlow (Abadi et al., 2016) and regards the rematerialization problem as a constrained optimization problem in which the trade-off between training time and memory consumption needs to be optimized. First, Checkmate converts the input model into a static DAG, $G(V, E)$, where V is the set of the vertexes and E is the set of the edges in G. The vertex stands for the operations between tensors and the edge is the dependency between operators. Similar to DTR, for operator v, its memory consumption is m_v and computation cost is c_v. For each operator, there are only two states in this design, namely *Recomputation* and *Stay-in-GPU*. They use a binary indicator to represent the state of each tensor. To be more specific, $S_{t,i} \in \{0, 1\}$ indicates that the result of operation i should be kept *Stay-in-GPU* at stage $t-1$ until stage t. $R_{t,i} \in 0, 1$ indicates whether the operation i is recomputed at time step t. Under this formulation,

$$\begin{aligned} &\arg\min_{R,S} \sum_{t=1}^{n} \sum_{i=1}^{t} C_i R_{t,i} \\ &\textit{subject to } R_{t,j} \leq R_{t,i} + S_{t,i} \quad \forall t \forall (v_i, v_j) \in E, \\ &\qquad S_{t,i} \leq R_{t-1,i} + S_{t-1,i} \quad \forall t \geq \forall i, \\ &\qquad \sum_i S_{1,i} = 0, \\ &\qquad \sum_t R_{t,n} \geq 1, \\ &\qquad R_{t,i}, S_{t,i} \in \{0, 1\} \quad \forall t \forall i \end{aligned} \tag{3}$$

These constraints are formulations of the relationship of the graph G. Then, they use Integer Linear Programming(ILP) (Margot, 2010) and the Gurobi tool to solve Eq. (3) and get the states for each operator. The training process proceeds based on the rebuilt graph with rematerialization, leading to increased input batch size with the saved memory.

There are also some other works concerning rematerialization. Beaumont et al. (2020) employ dynamic programming to find the best rematerialization strategies for DNN models. To formalize the optimization problem, they use $\mathcal{G} = (V, E)$ to represent the model graph, where V is the set of vertex and E is the set of edges between vertex, which is similar as the formulation of Checkmate. However, these works require further evaluation of real-world datasets and large-scale language models. Besides, they also developed an open-source framework regarding rema-

terialization with Pytorch, Rotor[1]. Then, Zhao et al. combined Checkmate (Jain et al., 2020) and Rotor (Herrmann et al., 2019) that focuses on the heterogeneous chains and proposed Rockmate (Zhao et al., 2023a). They considered each model as a sequence of blocks and applied Checkmate to each block in the sequence. Then Rotor is applied to the entire sequence. They gained a speedup compared to Checkmate in rematerialization.

Tensor rematerialization results in severe memory fragments in the training process. To address this problem, Zhang et al. proposed Coop (Zhang et al., 2023). They tried to evict tensors within a sliding window algorithm. They use a list to store all the tensors' states and sort these tensors by their addresses and the tensors in the sliding window are supposed to be evicted. Besides, they define a heuristic function as

$$h(t) = c(t)/s(t), \tag{4}$$

where $c(t)$ is the sum computation cost in the window and $s(t)$ is the staleness time. This window slides within the list and compares the summed heuristics of continuous tensors in the window, of which the smallest heuristics are evicted.

In 2023, Bartan et al. proposed a new optimization model formulation named Moccasin. Moccasin defines each variables *retention intervals* through which each node in the computation graph indicates the retention of its output in local memory. The formulation only includes $O(n)$ integer variables, where n is the node in the graph, which is much smaller than the $O(n^2)$ variables in Checkmate.

5.2 Swapping

Swapping methods involve transporting data between the GPU to other physical devices, such as CPU or SSD, to reduce the GPU memory usage. Swapping also indicates offloading in some papers (Ren et al., 2021; Rajbhandari et al., 2021).

Rhu et al. proposed vDNN, which swaps out training activations to the CPU during the forward process and swaps them back to the GPU during the backward process for gradient computation(Rhu et al., 2016). To minimize the transportation time, vDNN performs computation and data transmission concurrently to hide the transmission time. However, vDNN introduces time delays due to synchronization of computation and data transmission after each layer. Furthermore, vDNN is applicable only to convolutional neural networks.

SwapAdvisor (Huang et al., 2020) is a method proposed based on MXNet (Chen et al., 2015). It takes the model graph as input and uses Genetic Algorithm (Holland, 1992) to find the optimal operator schedule in the constructed search space, considering memory allocation and operator schedules. SwapAdvisor firstly decides which tensors to swap out, determines the optimal timing for swap-in and swap-out operations to optimize the overlapping between computation and communication and outputs an augmented graph. Then the augmented graph is simulated to measure the corresponding execution time. This process iterates until the original graph is fully optimized. SwapAdvisor employs Belady's strategy for selecting tensors that will not be used in the future to be swapped out and adopts prefetching techniques to maximize the overlapping between computation and communication.

Hildebrand et al. proposed AutoTM which utilized ILP to find the optimal tensor assignment and movement between DRAM or NVDIMMs. AutoTM takes a DNN model as input and tries to minimize the execution time. Besides, AutoTM is the first to employ DRAM and nonvolatile memory to reduce the memory of model activations. However, they cannot deal with large-scale models because the search space of large-scale models is extremely large.

Bae et al. proposed FlashNeuron which uses an NVMe SSD as a backing store (Bae et al., 2021). They introduced an offloading scheduler to find the optimal offloading schedule with an input batch size and a DNN model. Firstly, FlashNeuron conducts a profiling iteration. Within this iteration, all the buffered tensors during the forward process are offloaded to SSD to avoid the out-of-memory error. All the tensors' size and time are collected within this profiling iteration. This information is used to determine which tensors are supposed to be offloaded. This process is executed by the scheduler. The scheduler checks whether the total data-offloading time computed by accumulating the offloading time of each tensor is less than the total execution time of all layers during forward propagation. If so, the offloading scheduler adopts this schedule to achieve fully overlap between the computation and offload-

[1]https://gitlab.inria.fr/hiepacs/rotor

ing processes.

ZeRO-Offload (Ren et al., 2021) built upon ZeRO, addresses the training of large-scale models by offloading all fp32 model states and fp16 gradients from GPU to CPU memory. The parameter updates are performed on the CPU, while the fp16 parameters remain on the GPU, and both the forward and backward processes are executed on the GPU. ZeRO-Offload utilizes the CPU-Adam optimizer to accelerate the CPU computation process. It is integrated into the DeepSpeed framework (Rasley et al., 2020).

On the other hand, ZeRO-Infinity (Rajbhandari et al., 2021) aims to reduce the memory footprint of large-scale model training using CPU, GPU, and NVMe (Xu et al., 2015). It introduces the "Infinity Offload Engine" which offloads all model states to CPU or NVMe memory. The engine comprises two main components: DeepNVMe, a C++ NVMe library for asynchronous management of read and write requests to overlap computation and communication, and the pinned memory management layer, a memory buffer that offloads model states to CPU or NVMe memory to prevent memory fragmentation in CPU and GPU memory. ZeRO-Infinity also includes CPU offloading of activations similar to ZeRO-Offload. The "Overlap Engine" in ZeRO-Infinity overlaps GPU-GPU communication with GPU computation, as well as NVMe to CPU and CPU to GPU communication. It consists of a dynamic Prefetcher that reconstructs model parameters before their use in the forward and backward processes and a data movement management component that executes the required data movement for gradients during backward computation.

Swapping methods are constrained by the bandwidth of communication links, such as PCIe (Neugebauer et al., 2018). Swapping incurs higher costs compared to rematerialization due to these hardware limitations. Achieving maximal overlapping between computation and communication remains a challenge in these methods.

5.3 Combination of rematerialization and swapping

Methods that combine rematerialization and swapping techniques have been explored to optimize GPU memory usage in training deep neural networks.

SuperNeurons (Wang et al., 2018a) proposed a method that analyzes the liveness of each layer, reducing the maximum memory consumption during the forward process. For each layer, l, the memory usage in the forward process is denoted as m_l^f while the memory usage in the backward is represented as m_l^b. Therefore, the total memory consumption is $\sum_{i=1}^{N} m_{l_i}^f + \sum_{i=1}^{N} m_{l_i}^b$, where N is the number of model layers. Among these layers, we have $m_{peak} = max(l_i)$, where $i \in [1, N]$. SuperNeurons includes a module named *Liveness Analysis* analyzing the liveness of each layer. The memory consumption in the forward process is $\sum_{i=1}^{k} m_{l_i}^f$ at layer k, where $k \leq N$. *Liveness Analysis* reduces the maximum memory consumption from $\sum_{i=1}^{N} m_{l_i}^f + \sum_{i=1}^{N} m_{l_i}^b$ to $\sum_{i=1}^{N} m_{l_i}^f + m_{l_N}^b$. Assuming the memory cost of each layer is identical, during the backward process, the memory consumption at layer k is $\sum_{i=1}^{k} m_{l_i}^f + m_{l_k}^b$. Therefore, the peak memory is $\sum_{i=1}^{N} m_{l_i}^f + m_{l_N}^b$. In this way, SuperNeurons reduces the memory consumption to $\sum_{i=1}^{N} m_{l_i}^f + m_{l_N}^b$, where l_i^f is not the checkpointing layer. It is obvious that SuperNeurons only swaps the {CONV} operator, limiting its applicability.

Capuchin (Peng et al., 2020) made significant progress by combining rematerialization and swapping in a fine-grained manner. They made decisions on rematerialization or swapping for tensors based on information from the first iteration, with swapping preferred to maximize computation and communication overlap. They introduced the Memory Saving Per Second (MSPS) (Eq.5) metric to measure the benefit of evicting or swapping tensors.

$$MSPS = \frac{Memory\ saving}{Rematerialization\ Time} \tag{5}$$

Beaumont et al. proposed dynamic programming to find the optimal sequence combination of rematerialization and swapping and proposed an algorithm named `pofo`(Beaumont et al., 2021) for layers. Moreover, they can only be used for linear networks, which is a huge limitation for language models nowadays.

DELTA (Tang et al., 2022) aimed to achieve automatic and fine-grained memory optimization for training large-scale models. The optimization process was decoupled into three modules: Filter, Director, and Prefetcher. Filter presents a heuristic function to select the optimal tensor to get *Release*.

They define the heuristic function F_{Filter} taking the staleness time and the memory consumption into consideration, as Eq. (6).

$$F_{Filter}(t) = \frac{1}{m(t)s(t)} \tag{6}$$

The filtering function could also be alternated with LRU and Greedy. Director is supposed to choose a better action from {*Eviction*, *Offload*} for the selected tensor from Filter. In Tang et al. (2022), $c_r(t)$ is defined as the rematerialization cost while $c_s(t)$ stands for the swapping cost of tensor t. They compare $c_r(t)$ and $c_s(t)$ to get a better action for tensor t. If $c_r(t) > c_s(t)$, the rematerialization cost is larger than the swapping cost, Director returns *Offload* for the tensor. Otherwise, the Director returns *Eviction*. As far as Prefetcher is concerned, it works in the backward process. The Prefetcher is responsible for prefetching those offloaded tensors back to the GPU. Delta employed overlapping strategies to reduce the time overhead caused by eviction and offload operations. Different from Capuchin (Peng et al., 2020), DELTA is a dynamic manager for training large-scale models. However, DELTA dynamically managed memory for large-scale models, but memory fragmentation remained an unresolved issue.

RecShard (Sethi et al., 2022) focused on fine-grained embedding table sharding for deep learning recommendation models (Ko et al., 2022; Ali et al., 2020; Acun et al., 2021). They use embedding hashing (Kang et al., 2021; Weinberger et al., 2009) to map feature values to output values, constrained by a specified hash size.

Nie et al. split the tensors into some micro tensors(Nie et al., 2022). Each of these micro tensors is a fine-grained unit for a memory operation, as shown in Table 3. Firstly, they built the execution operation schedule based on the computation graph. They finish the eviction and offloading based on the micro tensors.

Although these strategies have been developed, there is still a lack of their efficient usage in paralleled training. For example, how to train large-scale models with 4D Parallel efficiently and dynamically, combining 3D Parallel and memory optimization remains a problem for researchers.

6 Discussion

Looking ahead, future research should prioritize the development of memory-efficient techniques that also ensure high-training speeds. Finding a balance between memory optimization and training efficiency will be crucial for further advancements in large-scale language models. Therefore, we suggest a few trends that may be research hit in the future:

A) Supporting new parallel methods

3D Parallel has attracted plenty of attention nowadays. Researchers are also seeking new parallel dimensions to reduce memory consumption beyond 3D Parallel. We expect new parallel methods or a new combination of some parallelism to train large-scale models while achieving memory reduction in the future, such as the combination of Data Parallel and Sequence Parallel. Besides, with the emergence of new technologies, the performance of existing parallel methods can be further improved, such as FlashAttention (Dao et al., 2022) and FlashAttention2 (Dao, 2023). They could be combined with Sequence Parallel and speed up the vanilla Sequence Parallel.

On the other hand, as mentioned in Section 3.1.4, auto parallelism aims to find the optimal solution for training large-scale models given the model and the device topology. However, the sequence parallel and expert parallel have not been researched in auto parallelism. We expect that new parallel methods could be combined with 3D Parallel and achieve training large-scale models in an automated fashion.

B) Ensuring the convergence of optimizers

Optimizers need to ensure the convergence of training large-scale models. This is because optimizers such as Adafactor are sensitive to hyper-parameter settings. Different hyper-parameter settings could result in non-convergence training results. There still lacks a new optimizer with stable convergence and less memory consumption. How to ensure the convergence with less memory consumption of model states remains an open domain in the future.

C) Improving rematerialization and swapping

Both rematerialization and swapping gain memory reduction at a cost of low throughput. Moreover, the bandwidth between CPU and GPU limits

the speed of swapping data between CPU and GPU, causing severe time delays in swapping. Another concern is how to accelerate rematerialization and swapping when training large-scale models in memory optimization. In general, the overlap between communication and computation remains to be settled perfectly.

On the other hand, for fine-grained rematerialization and swapping methods, memory fragmentation is a vital problem nowadays, especially for dynamic methods, such as DTR and DELTA. How to improve the memory fragmentation in the dynamic methods remains to be settled.

D) Combining memory-optimization methods

Now, these memory-optimization methods pay attention to one single part concerning model parameters, model states, and model activations. Therefore, we expect these memory-optimization methods to be used together when training large-scale models (Sun et al., 2021; Zeng et al., 2021). However, there is still a lack of such a work combining these methods. In reality, large-scale models have been trained with more than 3D Parallel methods (Yao et al., 2023; Wang et al., 2023). Wang et al. integrate memory optimization (rematerialization and swapping) and parallelism techniques (data parallelism, model parallelism, and pipeline parallelism). In this paper, they assigned each technique as a kind of switch. It is well known that there are forward passes and backward passes for each layer in the training model. Then, for all the layers in the model, they obtain the decision for the switches of all the layers. However, fine-grained combinations of these methods have not been explored. For example, DELTA could be integrated with mixed parallel methods to obtain better memory reduction. However, it is hard for DELTA to fit the tensor MP situations, for the tensor is split on each GPU. How to evict or offload these split tensors remains a problem for DELTA to be settled.

Moreover, memory-optimization methods are not considered in recent auto-paralleled methods. When faced with memory optimization, it is much more complex for the runtime optimization applied to the profiled model. Therefore, we expect memory optimization could be considered in auto parallelism and we could achieve training large-scale models with limited memory in an automated fashion.

E) Reducing the communication overhead

In distributed parallel training processes, performance degradation occurs due to communication across GPUs, especially when the number of GPUs increases. This communication involves all-reduce communication for both data and tensor parallelism, as well as point-to-point communication in pipeline parallelism. Reducing the communication overhead or achieving overlap between communication and computation is currently a research hotspot.

7 Conclusions

In recent years, large-scale language models have gained significant attention, with their scale increasing exponentially. However, the *GPU memory wall* problem poses a major obstacle to further advancements in large-scale models, as training them with limited GPU memory remains a challenge. In this paper, we first analyze the main factors contributing to massive memory consumption in training large-scale models. Then we reviewed various methods of training large-scale models with limited GPU memory. Through the analysis of the training process of large-scale models, we have identified model parameters, model states, and model activations as the main contributors to GPU memory consumption. We discussed the approaches and techniques employed to train large-scale models in these areas. Finally, we present our discussion about the future of training large-scale models with limited memory.

Contributors

Yu TANG proposed this idea and wrote most sections of this paper. Linbo QIAO and Dongsheng LI are responsible for overseeing the entire survey paper and ensuring that it provides a comprehensive overview of the field. Lujia YIN has written a section of the paper that discusses the methods that reduce the memory consumption of model states. Peng LIANG wrote a section of the paper that discusses the different approaches regarding distributed and parallel algorithms. Ao SHEN wrote a section of the paper that discusses the different techniques for mixed precision training. Zhilin YANG and Lizhi ZHANG helped draw pictures and helped discussed this topic.

Conflict of interest

All the authors declare that they have no conflict of

interest.

Data availability

Due to the nature of this research, participants of this study did not agree for their data to be shared publicly, so supporting data is not available.

References

Abadi M, Barham P, Chen J, et al., 2016. Tensorflow: A system for large-scale machine learning. Proc 12^{th} USENIX Conf on Operating Systems Design and Implementation, p.265-283, https://doi.org/10.5555/3026877.3026899

Acun B, Murphy M, Wang X, et al., 2021. Understanding training efficiency of deep learning recommendation models at scale. Proc IEEE Int Symp on High-Performance Computer Architecture, p.802-814, https://doi.org/10.1109/HPCA51647.2021.00072

Ali Z, Kefalas P, Muhammad K, et al., 2020. Deep learning in citation recommendation models survey. *Exp Syst Appl*, 162:113790. https://doi.org/10.1016/j.eswa.2020.113790

Amari Si, 1993. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185-196.

Bachlechner T, Majumder BP, Mao H, et al., 2021. Rezero is all you need: Fast convergence at large depth. Proc Thirty-Seventh Conf on Uncertainty in Artificial Intelligence, p.1352-1361.

Bae J, Lee J, Jin Y, et al., 2021. Flashneuron:ssd-enabled large-batch training of very deep neural networks. Proc 19^{th} USENIX Conf on File and Storage Technologies, p.387-401.

Banner R, Hubara I, Hoffer E, et al., 2018. Scalable methods for 8-bit training of neural networks. Proc 32^{nd} Int Conf on Neural Information Processing Systems, p.5151-5159.

Bartan B, Li H, Teague H, et al., 2023. Moccasin: efficient tensor rematerialization for neural networks. International Conference on Machine Learning, p.1826-1837.

Beaumont O, Herrmann J, Pallez G, et al., 2020. Optimal memory-aware backpropagation of deep join networks. *Philos Trans Roy Soc A*, 378(2166):20190049.

Beaumont O, Eyraud-Dubois L, Shilova A, 2021. Efficient combination of rematerialization and offloading for training dnns. Proc 35^{th} Conf on Neural Information Processing Systems, p.23844-23857.

Brown TB, Mann B, Ryder N, et al., 2020. Language models are few-shot learners. Proc 34^{th} Conf on Neural Information Processing Systems, p.1877-1901.

Chen AC, Zhang YM, Jia JH, et al., 2024. Deepzero: Scaling up zeroth-order optimization for deep model training, . https://doi.org/10.48550/arXiv.2310.02025.

Chen JF, Gai Y, Yao ZW, et al., 2020. A statistical framework for low-bitwidth training of deep neural networks. Proc 34^{th} Int Conf on Neural Information Processing Systems, article 75, p.883-894, https://doi.org/10.5555/3495724.3495799

Chen JF, Zheng LM, Yao ZW, et al., 2021. Actnn: Reducing training memory footprint via 2-bit activation compressed training. Proc 38^{th} Int Conf on Machine Learning, p.1803-1813.

Chen JF, Li SG, Gun R, et al., 2023. Autoddl: Automatic distributed deep learning with asymptotically optimal communication, . https://doi.org/10.48550/arXiv.2301.06813

Chen TQ, Li M, Li YT, et al., 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, . https://doi.org/10.48550/arXiv.1512.01274

Chen TQ, Xu B, Zhang CY, et al., 2016. Training deep nets with sublinear memory cost, . https://doi.org/10.48550/arXiv.1604.06174

Cho K, van Merriënboer B, Bahdanau D, et al., 2014. On the properties of neural machine translation: Encoder–decoder approaches. Proc SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, p.103-111, https://doi.org/10.3115/v1/W14-4012

Choquette J, Gandhi W, Giroux O, et al., 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29-35. https://doi.org/10.1109/MM.2021.3061394

Chowdhury GG, 2003. Natural language processing. *Annu Rev Inf Sci Technol*, 37(1):51-89. https://doi.org/10.1002/aris.1440370103

Cutkosky A, Mehta H, 2020. Momentum improves normalized sgd. Proc 37^{th} Int Conf on Machine Learning, p.2260-2268.

Dao T, Fu D, Ermon S, et al., 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. Proc 36^{th} Conf on Neural Information Processing Systems, p.16344-16359.

Dao T, 2023. Flashattention-2:faster attention with better parallelism and work partitioning, https://doi.org/10.48550/arXiv.2307.08691

Dean J, Corrado GS, Monga R, et al., 2012. Large scale distributed deep networks. Proc 25^{th} Int Conf on Neural Information Processing Systems, p.1223-1231. https://doi.org/10.5555/2999134.2999271

Devlin J, Chang MW, Lee K, et al., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. Proc Conf of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, Minnesota, p.4171-4186. https://doi.org/10.18653/v1/N19-1423

Dong L, Yang N, Wang WH, et al., 2019. Unified language model pre-training for natural language understanding and generation. Proc 33^{rd} Int Conf on Neural Information Processing Systems, article 1170. https://doi.org/10.5555/3454287.3455457

Fan SQ, Rong Y, Meng C, et al., 2021. Dapple: A pipelined data parallel approach for training large models. Proc 26^{th} ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.431-445. https://doi.org/10.1145/3437801.3441593

Fang JR, Zhu ZL, Li SG, et al., 2023. Parallel training of pre-trained models via chunk-based dynamic memory management. *IEEE Trans Parallel Distrib Syst*, 34(1):304-315. https://doi.org/10.1109/TPDS.2022.3219819

Fedus W, Zoph B, Shazeer N, 2021. Switch transformers: Scaling to trillion parameter models with simple and

efficient sparsity. *J Mach Learn Res*, 23(1):120. https://doi.org/10.5555/3586589.3586709

Fu FC, Hu YZ, He YH, et al., 2020. Don't waste your bits! squeeze activations and gradients for deep neural networks via tinyscript. Proc 37^{th} Int Conf on Machine Learning, article 309, p.3304-3314. https://doi.org/10.5555/3524938.3525247

Gholami A, Yao ZW, Kim S, et al., 2021. Ai and memory wall. *IEEE Micro*, . https://doi.org/10.1109/MM.2024.3373763

Guan L, Yin WT, Li DS, et al., 2019. Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training, . https://doi.org/10.48550/arXiv.1911.04610

Gusak J, Cherniuk D, Shilova A, et al., 2022. Survey on large scale neural network training, . https://doi.org/10.48550/arXiv.2202.10435

Gustafson JL, 1988. Reevaluating amdahl's law. *Commun ACM*, 31(5):532-533. https://doi.org/10.1145/42411.42415

Han K, Wang YH, Chen HT, et al., 2023. A survey on vision transformer. *IEEE Trans Pattern Anal Mach Intell*, 45(1):87-110. https://doi.org/10.1109/TPAMI.2022.3152247

Han S, Mao HZ, Dally WJ, 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, https://doi.org/10.48550/arXiv.1510.00149

He CY, Li S, Soltanolkotabi M, et al., 2021. Pipetransformer: automated elastic pipelining for distributed training of large-scale models. Proc 38^{th} Int Conf on Machine Learning, p.4150-4159.

He KM, Zhang XY, Ren SQ, et al., 2016. Deep residual learning for image recognition. Proc IEEE Conf on Computer Vision and Pattern Recognition, p.770-778. https://doi.org/10.1109/CVPR.2016.90

Herrmann J, Beaumont O, Eyraud-Dubois L, et al., 2019. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory, https://doi.org/10.48550/arXiv.1911.13214

Hildebrand M, Khan J, Trika S, et al., 2020. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. Proc Twenty-Fifth Int Conf on Architectural Support for Programming Languages and Operating Systems, p.875-890. https://doi.org/10.1145/3373376.3378465

Holland JH, 1992. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT Press, Cambridge, USA.

Huang CC, Jin G, Li JY, 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. Proc Twenty-Fifth Int Conf on Architectural Support for Programming Languages and Operating Systems, p.1341-1355. https://doi.org/10.1145/3373376.3378530

Huang YP, Cheng YL, Bapna A, et al., 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Proc 33^{rd} Conf on Neural Information Processing Systems, p. 32.

Jain P, Jain A, Nrusimha A, et al., 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization. Proc 3^{rd} MLSys Conf, p.497-511.

Ji SW, Xu W, Yang M, et al., 2013. 3d convolutional neural networks for human action recognition. *IEEE Trans Pattern Anal Mach Intell*, 35(1):221-231. https://doi.org/10.1109/TPAMI.2012.59

Jia Z, Maggioni M, Staiger B, et al., 2018. Dissecting the nvidia volta gpu architecture via microbenchmarking. https://doi.org/10.48550/arXiv.1804.06826

Jia ZH, Zaharia M, Aiken A, 2019. Beyond data and model parallelism for deep neural networks. p.1-13.

Kang WC, Cheng DZ, Yao TS, et al., 2021. Learning to embed categorical features without embedding tables for recommendation. Proc 27^{th} ACM SIGKDD Conf on Knowledge Discovery & Data Mining,, p.840-850. https://doi.org/10.1145/3447548.3467304

Kim C, Lee H, Jeong M, et al., 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models, . https://doi.org/10.48550/arXiv.2004.09910

Kingma DP, Ba J, 2015. Adam: A method for stochastic optimization. Proc 3^{rd} Int Conf on Learning Representations.

Kirisame M, Lyubomirsky S, Haan A, et al., 2021. Dynamic tensor rematerialization. Proc 9^{th} Int Conf on Learning Representations.

Kitaev N, Kaiser Ł, Levskaya A, 2020. Reformer: The efficient transformer. Proc 8^{th} Int Conf on Learning Representations.

Ko H, Lee S, Park Y, et al., 2022. A survey of recommendation systems: recommendation models, techniques, and application fields. *Electronics*, 11(1):141. https://doi.org/10.3390/electronics11010141

Korthikanti VA, Casper J, Lym S, et al., 2023. Reducing activation recomputation in large transformer models. Proc 6^{th} MLSys Conf, p. 5.

Krizhevsky A, Sutskever I, Hinton GE, 2012. Imagenet classification with deep convolutional neural networks. Proc 25th Int Conf on Neural Information Processing Systems, p.1097-1105. https://doi.org/10.5555/2999134.2999257

Lai ZQ, Li SW, Tang XD, et al., 2023. Merak: An efficient distributed dnn training framework with automated 3d parallelism for giant foundation models. *IEEE Trans Parallel Distrib Syst*, 34(5):1466-1478. https://doi.org/https://doi.org/10.1109/TPDS.2023.3247001

LeCun Y, Bengio Y, Hinton G, 2015. Deep learning. *Nature*, 521(7553):436-444. https://doi.org/10.1038/nature14539

Lee D, Choi J, Kim JH, et al., 1999. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. Proc ACM SIGMETRICS Int Conf on Measurement and Modeling of Computer Systems, p.134-143. https://doi.org/10.1145/301453.301487

Li M, Andersen DG, Park JW, et al., 2014. Scaling distributed machine learning with the parameter server. Proc 11^{th} USENIX Conf on Operating Systems Design and Implementation, p.583-598. https://doi.org/10.5555/2685048.2685095

Li S, Zhao YL, Varma R, et al., 2020. Pytorch distributed: experiences on accelerating data parallel training. *Proc VLDB Endow*, 13(12):3005-3018. https://doi.org/10.14778/3415478.3415530

Li SG, Xue FZ, Baranwal C, et al., 2022. Sequence parallelism: long sequence training from system perspective, https://doi.org/10.48550/arXiv.2105.13120

Li SG, Liu HX, Bian ZD, et al., 2023. Colossal-ai: A unified deep learning system for large-scale parallel training. Proc 52^{nd} Int Conf on Parallel Processing, p.766-775. https://doi.org/10.1145/3605573.3605613

Li SG, Hoefler T, 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, p.1-14. https://doi.org/10.1145/3458817.3476145

Liang P, Tang Y, Zhang XD, et al., 2022. A Survey on Auto-Parallelism of Neural Networks Training. https://doi.org/10.36227/techrxiv.19522414.v1

Lin YJ, Han S, Mao HZ, et al., 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. Proc 6^{th} Int Conf on Learning Representations.

Lin ZQ, Miao YS, Liu GD, et al., 2023. Superscaler: Supporting flexible dnn parallelization via a unified abstraction, https://doi.org/10.48550/arXiv.2301.08984.

Liu SJ, Kailkhura B, Chen PY, et al., 2018. Zeroth-order stochastic variance reduction for nonconvex optimization. Proc 32^{nd} Int Conf on Neural Information Processing Systems, p.3731-3741. https://doi.org/10.5555/3327144.3327289

Liu Z, Lin YT, Cao Y, et al., 2021. Swin transformer: Hierarchical vision transformer using shifted windows. Proc IEEE/CVF Int Conf on Computer Vision, p.9992-10002. https://doi.org/10.1109/ICCV48922.2021.00986

Liu ZM, Cheng SG, Zhou HT, et al., 2023. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, article 56, p.1-13. https://doi.org/10.1145/3581784.3607073

Luo Y, Ren XZ, Zheng ZW, et al., 2023. Came: Confidence-guided adaptive memory efficient optimization. Proc 61^{st} Annual Meeting of the Association for Computational Linguistics, p.4442-4453. https://doi.org/10.18653/v1/2023.acl-long.243

Ma ZX, He JA, Qiu JZ, et al., 2022. Bagualu: targeting brain scale pretrained models with over 37 million cores. Proc 27^{th} ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.192-204. https://doi.org/10.1145/3503221.3508417

Margot F, 2010. Symmetry in integer linear programming. *In: JÃijnger M, Liebling TM, Naddef D, et al (Eds), 50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, :647-686. https://doi.org/https://doi.org/10.1007/978-3-540-68279-0_17

Micikevicius P, Narang S, Alben J, et al., 2018. Mixed precision training, https://doi.org/10.48550/arXiv.1710.03740

Narayanan D, Harlap A, Phanishayee A, et al., 2019. Pipedream: generalized pipeline parallelism for dnn training. Proc 27^{th} ACM Symp on Operating Systems Principles, p.1-15. https://doi.org/10.1145/3341301.3359646

Neugebauer R, Antichi G, Zazo JF, et al., 2018. Understanding pcie performance for end host networking. Proc Conf of the ACM Special Interest Group on Data Communication, p.327-341. https://doi.org/10.1145/3230543.3230560

Nie XN, Miao XP, Yang Z, et al., 2022. Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting. Proc IEEE 38^{th} Int Conf on Data Engineering, p.2615-2628. https://doi.org/10.1109/ICDE53745.2022.00241

OpenAI, Achiam J, Adler S, et al., 2024. Gpt-4 technical report, https://doi.org/10.48550/arXiv.2303.08774

Park JH, Yun G, Yi CM, et al., 2020. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. Proc USENIX Annual Technical Conf, p.307-321.

Paszke A, Gross S, Massa F, et al., 2019. Pytorch: An imperative style, high-performance deep learning library. Proc 33^{rd} Int Conf on Neural Information Processing Systems, article 721. https://doi.org/10.5555/3454287.3455008

Peng X, Shi XH, Dai HL, et al., 2020. Capuchin: Tensor-based gpu memory management for deep learning. Proc Twenty-Fifth Int Conf on Architectural Support for Programming Languages and Operating Systems, p.891-905. https://doi.org/10.1145/3373376.3378505

Povey D, Ghoshal A, Boulianne G, et al., 2011. The kaldi speech recognition toolkit. Proc IEEE 2011 Workshop on Automatic Speech Recognition and Understanding.

Pudipeddi B, Mesmakhosroshahi M, Xi JW, et al., 2020. Training large neural networks with constant memory using a new execution algorithm, https://doi.org/10.48550/arXiv.2002.05645

Qiu XP, Sun TX, Xu YG, et al., 2020. Pre-trained models for natural language processing: A survey. *Sci China Technol Sci*, 63(10):1872-1897. https://doi.org/10.1007/s11431-020-1647-3

Raffel C, Shazeer N, Roberts A, et al., 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J Mach Learn Res*, 21(1):140. https://doi.org/10.5555/3455716.3455856

Rajbhandari S, Rasley J, Ruwase O, et al., 2020. Zero: Memory optimizations toward training trillion parameter models. Proc SC20: Int Conf for High Performance Computing, Networking, Storage and Analysis, p.1-16. https://doi.org/10.1109/SC41405.2020.00024

Rajbhandari S, Ruwase O, Rasley J, et al., 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. Proc Int Conf for High Performance Computing, Networking, Storage and Analysis, article 595, p.1-14. https://doi.org/10.1145/3458817.3476205

Rajbhandari S, Li CL, Yao ZW, et al., 2022. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. Proc 39^{th} Int Conf on Machine Learning, p.18332-18346.

Rajpurkar P, Zhang J, Lopyrev K, et al., 2016. Squad: 100,000+ questions for machine comprehension of text.

Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, p.2383-2392, https://doi.org/10.18653/v1/D16-1264

Rasley J, Rajbhandari S, Ruwase O, et al., 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. Proc 26^{th} ACM SIGKDD Int Conf on Knowledge Discovery & Data Mining, p.3505-3506, https://doi.org/10.1145/3394486.3406703

Ren J, Rajbhandari S, Aminabadi RY, et al., 2021. Zero-offload: Democratizing billion-scale model training. Proc USENIX Annual Technical Conf, p.551-564.

Ren SQ, He KM, Girshick R, et al., 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. p.91-99. https://doi.org/10.5555/2969239.2969250

Rhu M, Gimelshein N, Clemons J, et al., 2016. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. Proc 49^{th} Annual IEEE/ACM Int Symp on Microarchitecture, p.1-13. https://doi.org/10.1109/MICRO.2016.7783721

Sergeev A, Del Balso M, 2018. Horovod: fast and easy distributed deep learning in tensorflow, https://doi.org/10.48550/arXiv.1802.05799

Sethi G, Acun B, Agarwal N, et al., 2022. Recshard: statistical feature-based memory optimization for industry-scale neural recommendation. Proc 27^{th} ACM Int Conf on Architectural Support for Programming Languages and Operating Systems, p.344-358. https://doi.org/10.1145/3503222.3507777

Shazeer N, Stern M, 2018. Adafactor: Adaptive learning rates with sublinear memory cost. Proc 35^{th} Int Conf on Machine Learning, p.4596-4604.

Shoeybi M, Patwary M, Puri R, et al., 2020. Megatron-lm: Training multi-billion parameter language models using model parallelism, https://doi.org/10.48550/arXiv.1909.08053

Sun X, Choi J, Chen CY, et al., 2019. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Proc 33rd Int Conf on Neural Information Processing Systems, article 441*, . https://doi.org/10.5555/3454287.3454728

Sun X, Wang NG, Chen CY, et al., 2020. Ultra-low precision 4-bit training of deep neural networks. Proc 34^{th} Int Conf on Neural Information Processing Systems, p.1796-1807. https://doi.org/10.5555/3495724.3495876

Sun Y, Wang SH, Li YK, et al., 2019. Ernie: Enhanced representation through knowledge integration, https://doi.org/10.48550/arXiv.1904.09223

Sun Y, Wang SH, Feng SK, et al., 2021. Ernie 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation, https://doi.org/10.48550/arXiv.2107.02137

Sutskever I, Martens J, Dahl G, et al., 2013. On the importance of initialization and momentum in deep learning. Proc 30^{th} Int Conf on Machine Learning, p.1139-1147. https://doi.org/10.5555/3042817.3043064

Sutskever I, Vinyals O, Le QV, 2014. Sequence to sequence learning with neural networks. Proc 27^{th} Int Conf on Neural Information Processing Systems, p.3104-3112. https://doi.org/10.5555/2969033.2969173

Tang Y, Wang CY, Zhang YF, et al., 2022. Delta: Dynamically optimizing gpu memory beyond tensor recomputation, https://doi.org/10.48550/arXiv.2203.15980

Unger C, Jia ZH, Wu W, et al., 2022. Unity: Accelerating dnn training through joint optimization of algebraic transformations and parallelization. Proc 16^{th} USENIX Symp on Operating Systems Design and Implementation, p.267-284.

Vaswani A, Shazeer N, Parmar N, et al., 2017. Attention is all you need. Proc 31^{st} Int Conf on Neural Information Processing Systems, p.6000-6010. https://doi.org/10.5555/3295222.3295349

Wang LN, Ye JM, Zhao YY, et al., 2018a. Superneurons: Dynamic gpu memory management for training deep neural networks. Proc 23^{rd} ACM SIGPLAN Symp on Principles and Practice of Parallel Programming, p.41-53. https://doi.org/10.1145/3178487.3178491

Wang NG, Choi J, Brand D, et al., 2018b. Training deep neural networks with 8-bit floating point numbers, :7686-7695. https://doi.org/10.5555/3327757.3327866

Wang YZ, Han X, Zhao WL, et al., 2023. H3t: Efficient integration of memory optimization and parallelism for high-throughput transformer training. Proc 37^{th} Conf on Neural Information Processing Systems.

Weinberger K, Dasgupta A, Langford J, et al., 2009. Feature hashing for large scale multitask learning. Proc 26^{th} Annual Int Conf on Machine Learning, p.1113-1120. https://doi.org/10.1145/1553374.1553516

Xi HC, Li CH, Chen JF, et al., 2023. Training transformers with 4-bit integers. Proc 37^{th} Conf on Neural Information Processing Systems.

Xiong RB, Yang YC, He D, et al., 2020. On layer normalization in the transformer architecture. Proc 37^{th} Int Conf on Machine Learning, p.10524-10533, https://doi.org/10.5555/3524938.3525913

Xu QM, Siyamwala H, Ghosh M, et al., 2015. Performance analysis of nvme ssds and their implication on real world databases. Proc 8^{th} ACM Int Systems and Storage Conf, article 6. https://doi.org/10.1145/2757667.2757684

Yao ZW, Aminabadi RY, Ruwase O, et al., 2023. DeepSpeed-Chat: Easy, Fast and Affordable RLHF Training of ChatGPT-like Models at All Scales, https://doi.org/10.48550/arXiv.2308.01320

You Y, Li J, Reddi SJ, et al., 2020. Large batch optimization for deep learning: Training bert in 76 minutes. Proc 8^{th} Int Conf on Learning Representations.

Yuan JH, Li XQ, Cheng C, et al., 2021. Oneflow: Redesign the distributed deep learning framework from scratch, https://doi.org/10.48550/arXiv.2110.15032

Ze HG, Senior A, Schuster M, 2013. Statistical parametric speech synthesis using deep neural networks. Proc IEEE Int Conf on Acoustics, Speech and Signal Processing, p.7962-7966. https://doi.org/10.1109/ICASSP.2013.6639215

Zellers R, Bisk Y, Schwartz R, et al., 2018. Swag: A large-scale adversarial dataset for grounded commonsense inference. Proc Conf on Empirical Methods in Natural Language Processing, p.93-104, https://doi.org/10.18653/v1/D18-1009

Zeng W, Ren XZ, Su T, et al., 2021. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation, https://doi.org/10.48550/arXiv.2104.12369

Zhang DQ, Yang JL, Ye DQ, et al., 2018. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. Proc 15^{th} European Conf on Computer Vision, p.365-382. https://doi.org/10.1007/978-3-030-01237-3_23

Zhang JH, Ma SH, Liu PH, et al., 2023. Coop: Memory is not a commodity. Proc 34^{th} Conf on Neural Information Processing Systems.

Zhang ZY, Han X, Zhou H, et al., 2021. Cpm: A large-scale generative chinese pre-trained language model. AI Open, 2:93-99, https://doi.org/10.1016/j.aiopen.2021.07.001

Zhao XY, Le Hellard T, Eyraud-Dubois L, et al., 2023a. Rockmate: an efficient, fast, automatic and generic tool for re-materialization in pytorch. Proc 40^{th} Int Conf on Machine Learning, p.42018-42045. https://doi.org/10.1145/3581784.3607073

Zhao YL, Gu A, Varma R, et al., 2023b. Pytorch fsdp: experiences on scaling fully sharded data parallel. *Proc VLDB Endow*, 16(12):3848-3860. https://doi.org/10.14778/3611540.3611569

Zheng LM, Li ZH, Zhang H, et al., 2022. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. Proc 16^{th} USENIX Symp on Operating Systems Design and Implementation, p.559-578.

Zhong Y, Zhu JJ, Yang PC, et al., 2023. Mqsp: Micro-query sequence parallelism for linearly scaling long sequence transformer, .

Zhou SC, Wu YX, Ni ZK, et al., 2018. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients, https://doi.org/10.48550/arXiv.1606.06160

Zhuang ZX, Liu MR, Cutkosky A, et al., 2022. Understanding adamw through proximal methods and scale-freeness. *Trans Mach Learn Res*, .