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 language models with limited GPU memory: a survey*

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

National Key Laboratory of Parallel and Distributed Computing, College of Computer,
National University of Defense Technology, Changsha 410073, China

†E-mail: tangyu14@nudt.edu.cn; dsli@nudt.edu.cn

Received Oct. 17, 2023; Revision accepted Mar. 31, 2024; Crosschecked Feb. 10, 2025; Published online Mar. 17, 2025

Abstract: Large-scale models have gained significant attention in 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 graphics processing units (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:** TP389.1

1 Introduction

With the advent of deep learning (LeCun et al., 2015), neural networks have witnessed significant advancements across diverse domains, such as speech recognition (Povey et al., 2011; Ze et al., 2013; Cho et al., 2014), computer vision (Krizhevsky et al., 2012; Ji et al., 2013; Ren SQ 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 Z et al., 2021; Han K et al., 2023). Large-scale models, such as BERT (Devlin et al., 2019), ERNIE (Sun Y et al., 2019), T5 (Raffel et al., 2020), and GPT-3 (Brown et al., 2020; Zhou J et al., 2024), 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

‡ Corresponding author
* Project supported by the National Natural Science Foundation of China (Nos. 62025208 and 62421002)
ORCID: Yu TANG, https://orcid.org/0000-0002-8595-1547; Dongsheng LI, https://orcid.org/0000-0001-9743-2034

the need for memory optimization techniques during their training (Sun Y et al., 2021; Zeng et al., 2021).

The exponential growth of the number of model parameters places high demands for graphics processing unit (GPU) memory. Unfortunately, the linear increase of GPU memory capacity occurring recently cannot meet the requirements of the exponential growth of the number 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., 2024; 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 handle. Liang et al. (2022) presented a survey that introduced auto parallelism in detail. They gave a comprehensive understanding and analysis of parallel and distributed training of deep neural networks (DNNs). Gusak et al. (2022) introduced the techniques of large-scale model training, including memory optimization methods. They summarized the main strategies contributing to training large-scale models. However, their comparison is not enough. Different from them, we present 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 submodel 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 YP et al., 2019), PipeDream (Narayanan et al., 2019), and XPipe (Guan et al., 2019). Moreover, Mixed Parallel, consisting of more than two types of parallelism, could decrease the memory consumption of model parameters, such as HetPipe (Park et al., 2020), DAPPLE (Fan et al., 2021), and Chimera (Li SG and Hoefler, 2021). In terms of reducing memory consumption from model states, we introduce some optimizers that have achieved success in training large-scale models, such as the zero redundancy optimizer (ZeRO) (Rajbhandari et al., 2020), Patrick-Star (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 works include 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 (Ren J et al., 2021), 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 drawn plenty of interest in the industry. For example, Megatron-LM trains large-scale models with Data Parallel, MP, and Pipeline Parallel. Besides, for training PanGu-α with 20 billion parameters, rematerialization and zero optimizers are adopted beyond 3D Parallel. The detailed information of these works is summarized in Table 1. In the end, we provide our concerns and discussion of future research about training large-scale models with limited GPU memory.

2 GPU memory consumption analysis

Fig. 1 illustrates the growth trends of various DNN model sizes and GPU memory capacities since 2016. The red inverted triangles 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 two years, while GPU memory capacity has increased only from 12 GB in the NVIDIA P100 GPU released in 2016 to 141 GB 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 interconnection bandwidth has increased by only 30 times (Jia Z et al., 2018). Under such conditions, the scalability of distributed parallelism is greatly restricted. Therefore, the GPU memory wall problem significantly impedes the 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.

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 entire model in the GPU.

Another significant memory consumption stems from the model states during the training process. In the context of the stochastic gradient descent (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

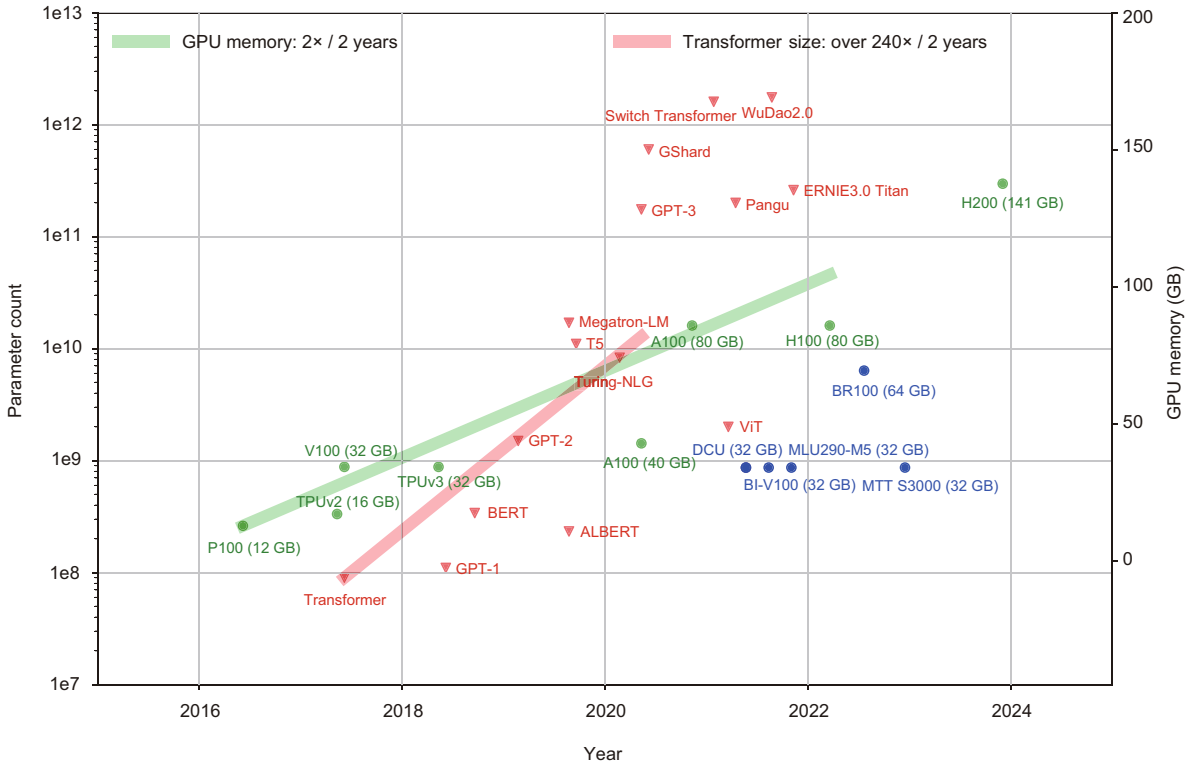

Fig. 1 Development of large-scale models' sizes and GPU memory capacity in recent years. It is obvious that the sizes of these models are increasing more and more rapidly and far beyond the capacity of GPU. References to color refer to the online version of this figure

Table 1 A summary of training methodologies regarding model parameters, model states, and model activations

Category	Method	Information		
		Method	Reference	Feature
Reducing memory of model parameters	Multi-GPU training	DistBelief	Dean et al., 2012	Model Parallel
		Megatron-LM	Shoeybi et al., 2020	
		Expert Parallel	Fedus et al., 2022	
		GPipe	Huang YP 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 SG and Hoefler, 2021	
		PipeTransformer	He CY et al., 2021	
		Hanayo	Liu ZM et al., 2023	
		Sequence Parallel	Li SG et al., 2022	Sequence Parallel
		MQSP	Zhong et al., 2023	
	Mixed-precision training	DoReFa	Zhou SC et al., 2018	–
		APEX	Micikevicius et al., 2018	–
		ActNN	Chen JF 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 Li SG et al., 2023 Zhao YL et al., 2023	DeepSpeed Colossal-AI PyTorch FSDP
		PatrickStar	Fang et al., 2023	–
		Adafactor	Shazeer and Stern, 2018	–
		CAME	Luo et al., 2023	–
		DeepZero	Chen AC et al., 2024	–
Reducing memory of model activations	Rematerialization	Checkpoint	Chen TQ et al., 2016	Static
		DTR	Kirisame et al., 2021	Dynamic
		Checkmate	Jain et al., 2020	Static
		Optimal Checkpoint	Beaumont et al., 2021	Static
		Rockmate	Zhao XY et al., 2023	Static
		Coop	Zhang JH et al., 2023	Dynamic
		MOCCASIN	Bartan et al., 2023	Static
	Swapping	vDNN	Rhu et al., 2016	Static
		SwapAdvisor	Huang CC et al., 2020	Static
		AutoTM	Hildebrand et al., 2020	Static
		FlashNeuron	Bae et al., 2021	Static
		ZeRO-Offload	Ren J et al., 2021	Static
		ZeRO-Infinity	Rajbhandari et al., 2021	Dynamic
	Combination	SuperNeurons	Wang LN 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

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. Assume that b is the input batch size and s is the sequence length. For a Transformer model of l layers, assuming that its hidden size is h and the vocabulary size is V, its parameter count is about $l(12h^2+13h)+Vh$. When h is large enough, the number of parameters is about $12lh^2$. For these activations, M_a is counted by $(34bsh+5bs^2a)l$, where a is the number of heads in the model (Korthikanti et al., 2023). M_a for GPT-4 (OpenAI et al., 2024) is represented as "–" because the setting 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 using 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 175B will produce about 366.00 GB activations. However, GPT-3 is trained with a long sequence of 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 SG et al. (2022) tried to train large-scale models with a sequence length of more than 2000, 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.

Table 2 Typical large-scale models and their parameter count together with the memory consumption of model parameters, model states, and model activations

Model	Number of parameters ($\times10^9$)	M_p (GB)	M_s (GB)	M_a (GB)
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	17.46	650.44	1951.32	366.00
GPT-4	≥1000	≥4000	≥12 000	–

M_p represents the memory consumption of model parameters; M_s represents the memory consumption of model states; M_a represents the memory consumption of model activations

Ideally, M_s accounts for the primary overhead, followed by M_p, and then 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. M_a is the most challenging to optimize. 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. 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), separately. For example, researchers usually adopt multi-GPU training and mixed-precision training to reduce the memory consumption of model parameters. Besides, they 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. Specifically, 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. In this section, we introduce mainly training large-scale models by decreasing the memory consumption using 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 tries to assign the model parameters to different GPUs and has proven pivotal in mitigating parameter consumption. In this subsection, we introduce mainly 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 M 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 a server and a worker simultaneously, and communication is limited to neighboring workers. PyTorch Distributed Data Parallel (PyTorch DDP) launches multi-progress as workers. Each worker stores a replica of the whole model (Li S et al., 2020). AllReduce is used 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 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 the number of model parameters within one single GPU, the parallel algorithms, such as MP and Pipeline Parallel, assign these model parameters to different GPUs, which reduces the memory consumption of model parameters within one GPU and has been widely adopted to alleviate the burden on model parameter memory consumption.

3.1.1 Model Parallel

MP is a widely used strategy in deep learning; it partitions the model into submodels and places each submodel on the GPU, such as DistBelief (Dean et al., 2012). It offers an alternative approach for mitigating memory redundancy within one GPU compared to Data Parallel. Fig. 2 shows the overview of MP. The model parameters are distributed among different workers. Each worker is responsible for sending and receiving activations instead of gradients, focusing solely on updating its assigned parameters.

Tensor Model Parallel, a specific form of MP, encompasses Column Parallel and Row Parallel techniques. A notable example of Tensor Model Parallel is showcased in the work of Megatron-LM (Shoeybi et al., 2020), where the multilayer perceptron (MLP) and self-attention modules within the Transformer layers are divided using Column Parallel and Row

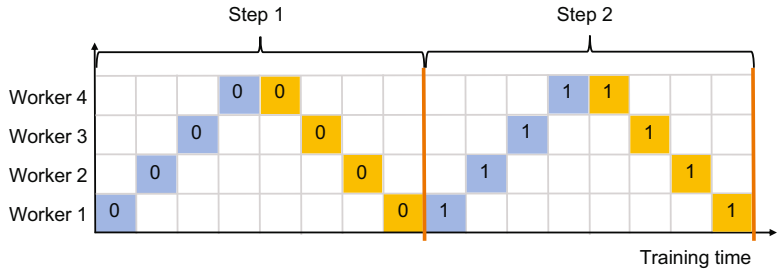

Fig. 2 Overview of Model Parallel. The entire model is split onto different GPUs and the training data batch flows from worker 1 to worker 4, which causes severe bubbles in the training process

Parallel strategies. Synchronization of data within the matrices is achieved through the utilization of AllReduce to ensure convergence.

Expert Parallel has been developed in Mixture of Expert (MoE) models (Fedus et al., 2022; Rajbhandari et al., 2022). 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 used in conjunction with Data Parallel and MP. Furthermore, Fedus et al. (2022) adopted a combination of parallel methods, including Data Parallel, MP, and Expert Parallel, to tackle models with over a trillion parameters. In this setting, Data Parallel and MP operate conventionally, while each expert is assigned a computation core in Expert Parallel, handling the corresponding allocated data efficiently.

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

3.1.2 Pipeline Parallel

GPipe (Huang YP et al., 2019), brought out by Google, primarily focuses on Pipeline Parallel (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 used 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 one forward one backward (1F1B) 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 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 be updated until all the micro-batches' weights are updated before old micro-batches, which incurs misconvergence. To solve this problem, PipeDream uses weight stashing to avoid mismatches between different versions of weights. To be more specific, weight stashing keeps multiple versions of weights for every mini-batch in GPU. PipeDream also supports checkpointing (Chen TQ et al., 2016) on each stage for fault tolerance. PipeDream gains comparable accuracy and a better speedup compared to Data Parallel.

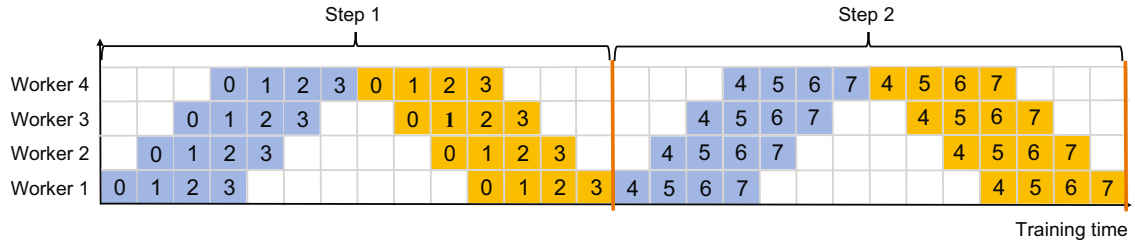

Fig. 3 Overview of GPipe, in which DNNs are trained in a pipeline. However, the bubbles result in low efficiency of GPU

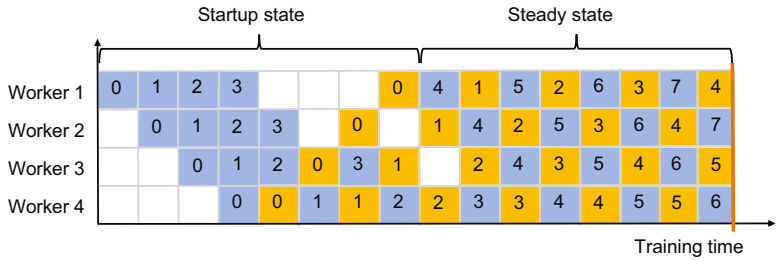

Fig. 4 Overview of PipeDream, which adopts the 1F1B strategy and reduces the bubbles in GPipe. The blue blocks represent the forward process in each worker and the orange ones the corresponding backward process. References to color refer to the online version of this figure

Guan et al. (2019) proposed XPipe as an approach for enhancing the efficiency of GPipe and tackling the challenges posed by synchronous training and asynchronous model training. PipeDream, on the other hand, uses 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 introduced the concept of "weight difference" as a measure of the number of weight updates.

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 complexities of the self-attention mechanism increase 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 GPU, and each GPU retains only a subsequence of the complete sequence (Li SG 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. (2023) proposed MQSP to solve the difficulty of Transformer-based models in scaling up to a long sequence. They split the input sequence into several subsequences. They also projected the local queries, keys, and values in self-attention. Different from vanilla Sequence Parallel, they designed a distributed self-attention by synchronizing the queries across all the devices. A distributed softmax was used, incurring negligible communication. They designed Micro-Q, which means the subqueries are split from the local query, ensuring a 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), y axis (MP), and z axis (Data Parallel), are represented (Fig. 5). This subsection primarily focuses on methods that leverage mixed parallelism.

HetPipe (Park et al., 2020) is a training framework that combines pipeline parallelism and

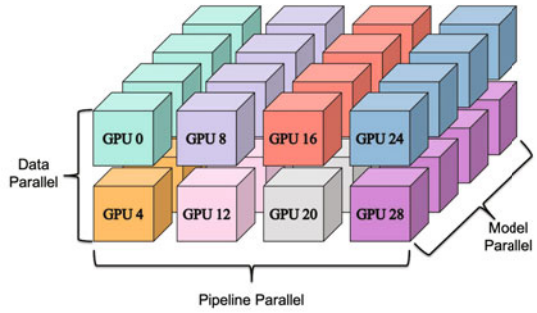

Fig. 5 Overview of 3D Parallel, which consists of Data Parallel, MP, and Pipeline Parallel. These parallel dimensions shard GPUs as their degrees in the training system

data parallelism for heterogeneous GPU clusters (Fig. 6). In the training system, data parallelism is facilitated through the concept of virtual worker (VW), 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 (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 (Fig. 7). The Profiler component takes the DNN model as input and profiles information such as the execution time, activations, and parameters for each layer. The Planner component aims to minimize pipeline latency between pipeline stages, leveraging dynamic programming to find optimal solutions, including device assignment. The Runtime component constructs forward and backward graphs for each pipeline stage, inserting split and concatenation operations for activation communications between adjacent stages. The Runtime component also builds a subgraph for 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 SG and Hoefler, 2021) is a synchronous method that combines data parallelism and pipeline parallelism (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

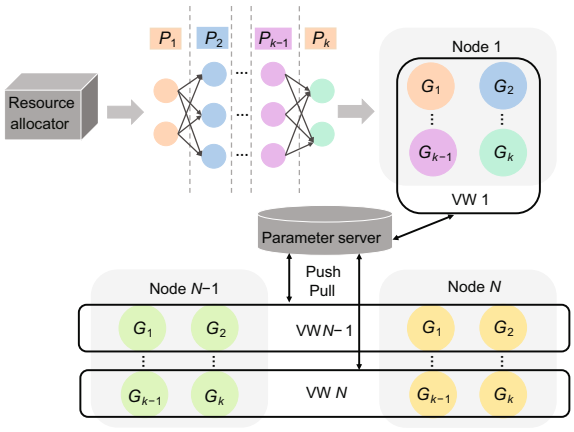

Fig. 6 HetPipe architecture. Data parallelism is executed among each node, also known as the virtual worker. Pipeline parallelism is executed from node to node

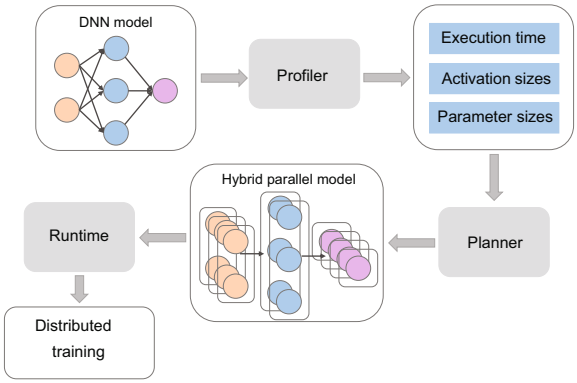

Fig. 7 DAPPLE architecture. The Profiler component takes the DNN model as input and profiles the execution time, activation sizes, and parameter sizes for the Planner. Then the Planner component finds the hybrid parallelism for this model through dynamic programming. The Runtime component launches distributed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

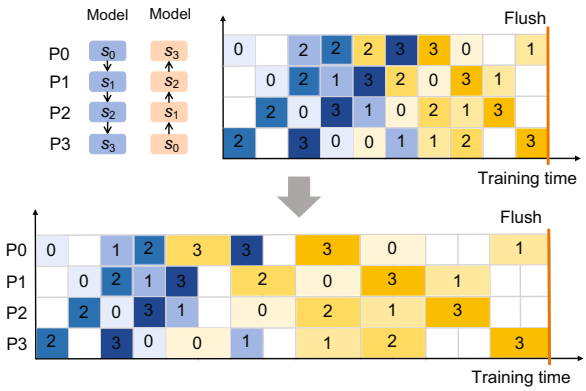

Fig. 8 Architecture of Chimera, which incorporates two pipeline directions in opposite order

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 CY 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 ZM et al. (2023) 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. 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 is transformed into two single-directional pipelines in a wave-like fashion. In this pattern, Hanayo employs the same model replication and obtains a performance at least as good as, if not better than, 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 (Jia ZH et al., 2019; Unger et al., 2022; Yuan et al., 2022; Zheng et al., 2022; Chen JF et al., 2023; Lin ZQ et al., 2023). However, these topics are beyond the scope of this paper, and readers can refer to the survey by Liang et al. (2022) for details. Furthermore, addressing the dilemma of overlapping communication and computation to accelerate the training of large-scale models remains a challenge.

3.2 Mixed-precision training

Mixed-precision training, such as those introduced by NVIDIA APEX (Micikevicius et al., 2018), has been developed to train DNNs using half-precision floating point numbers. By using lower precision methods, the memory and computing requirements of the training process can be significantly reduced. Additionally, researchers have presented several key ideas to enable 8-bit floating point (FP8) training, including (1) novel hybrid FP8 formats to represent weights, activations, and gradients, (2) chunk-based hierarchical accumulations to minimize low-precision accumulation errors, and (3) selective precision rules (for first, last, and depthwise convolutional layers) (Wang NG et al., 2018; Sun X et al., 2019). Banner et al. (2018) 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 X et al. (2020) explored a novel adaptive gradient scaling technique (GradScale) that addresses the challenges of insufficient range and resolution in quantized gradients and trains DNNs using 4-bit floating point. Xi et al. (2023) proposed a training method for Transformers with all matrix multiplications implemented with the INT4 arithmetic, and the numerical formats were supported by contemporary hardware. Furthermore, Ma et al. (2022) proposed 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.

Quantized training (Zhang DQ et al., 2018; Zhou SC et al., 2018; Chen JF et al., 2020) 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 (Fu et al., 2020; Chen JF et al., 2021) achieve memory savings by compressing activations to lower numerical precision through quantization. Moreover, weight compression techniques (Han S et al., 2016) and gradient compression techniques (Lin YJ 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 subsection, we present some notable works related to this field.

Pudipeddi et al. (2020) 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 stores only three layers.

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

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

where 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 the number of model parameters. Comparisons among ReZero, the vanilla Transformer, and the Pre-LN Transformer (Xiong et al., 2020) validate that ReZero enables training with deeper Transformer models and reduced GPU memory. The experimental results verify that ReZero makes it possible to train a deeper Transformer model.

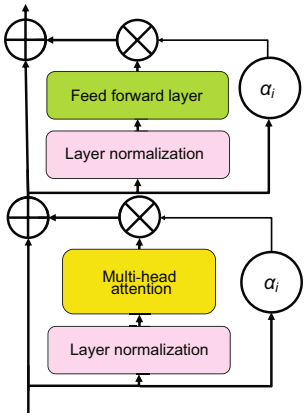

Fig. 9 Architecture of the ReZero Transformer. The parameter α determines whether the layer is calculated in the process

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 (Rajbhandari et al., 2020). In this paper, the model states include only the optimizer states and the gradients because we introduce these methods to reduce the memory consumption of model states. Therefore, in this section, we introduce mainly 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. For instance, training GPT-3 with 175 billion parameters requires approximately 2.1 TB of GPU memory for the model states when employing mixed-precision training together with Adam.

Rajbhandari et al. (2020) 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× 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 includes activations, commonly observed in model parallelism, temporary buffers, and memory fragmentation. For training activations, ZeRO employs a partitioning approach. In model parallelism, the redundancy arises from repeated activations. Hence, ZeRO eliminates this redundancy by partitioning and materializing activations in each activation layer at a specific time. After performing forward propagation within the model, the input activations are partitioned across all GPUs in the system or MP processes. ZeRO allocates a sufficiently large buffer of constant size for temporary buffers during the training process to minimize frequent memory access. Additionally, ZeRO tackles memory fragmentation by allocating contiguous memory chunks for activations 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, ZeRO-2 partitions gradients, and 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× communication overhead as those shared parameters are supposed to be gathered.

ZeRO has garnered significant interest and has been used 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 ZY et al., 2021). Besides, Li SG et al. (2023) introduced Colossal-AI, which incorporates the ZeRO optimizer into their framework. The PyTorch team has also implemented PyTorch Fully Shard Data Parallel (Zhao YL et al., 2023), offering an alternative implementation of the ZeRO optimizer.

4.2 PatrickStar

Based on ZeRO, Fang et al. (2023) proposed a chunk-based memory management for training large-scale models. In their opinion, static sharding of model states lacks the tolerance for training large-scale models without sufficient GPU or central processing unit (CPU) memory. They proposed PatrickStar, 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 dynamically according to the tensor states. They also used 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 showed that PatrickStar achieved better efficiency compared to DeepSpeed and lower memory consumption.

4.3 Adafactor

In addition to ZeRO, there are other approaches aimed at reducing the number of 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 to construct 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 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, and number of training iterations T.

Output: θ.

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

Algorithm 2, $\mathrm{RMS}(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_n^2}$. 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 an adaptive learning rate 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. Note that Adafactor does not use momentum information (Sutskever et al., 2013; Cutkosky and Mehta, 2020), 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.

Algorithm 2 Adafactor optimizer

Input: model parameters θ, loss function f, moment estimates $\beta \in [0,1)$, learning rate η, regularization constants ϵ_1 and ϵ_2, and number of training iterations T.

Output: θ.

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

4.4 CAME

Note that the second-order estimator of the gradients adds 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. (2023) proposed a confidence-guided strategy, CAME, which improves the robustness of Adafactor and gains an adaptive memory optimization. The algorithm of the CAME optimizer is displayed in Algorithm 3. In more detail, CAME uses 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. (2023) developed an adaptive memory mechanism with faster

Algorithm 3 CAME optimizer

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

Output: θ.

Initialize $v_0 = 0$, $u_0 = 0$
for $t = 1, 2, \cdots, T$ **do**
  $g_t \leftarrow \nabla f(\theta_{t-1})$
  $\beta_t = 1 - 1/t$
  $v_t \leftarrow \beta_t v_{t-1} + (1-\beta_t)(g_t^2 + \epsilon_1 1_n 1_m^{\mathrm{T}}) 1_m$
  $u_t \leftarrow \beta_t u_{t-1} + (1-\beta_t) 1_n^{\mathrm{T}} (g_t^2 + \epsilon_1 1_n 1_m^{\mathrm{T}})$
  $\hat{w}_t \leftarrow (v_t u_t / 1_n^{\mathrm{T}}) / v_t$
  $z_t = g_t / \sqrt{w_t}$
  $\eta_t = \max(\epsilon_2, \mathrm{RMS}(\theta_{t-1}))\rho_t$
  $m_t = \beta_1 m_{t-1} + (1-\beta_1)\eta_t$
  $U_t = (\eta_t - m_t)^2$
  $R_t = \beta_3 R_{t-1} + (1-\beta_3)(U_t + \epsilon_2 1_n 1_m^{\mathrm{T}})$
  $C_t = \beta_3 C_{t-1} + (1-\beta_3) 1_n^{\mathrm{T}} (U_t + \epsilon_1 1_n 1_m^{\mathrm{T}})$
  $S_t = R_t C_t / (1_n^{\mathrm{T}} R_t)$
  $\theta_t \leftarrow \theta_{t-1} - \eta m_t / \sqrt{S_t}$
end for
return θ

convergence. This memory-efficient approach helps avoid redundant evaluations of the objective function and speeds up the optimization process. Lines 11 to 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 comparable performance to Adam. The confidence-guided sampling and adaptive memory mechanisms also contribute to its ability to converge faster.

4.5 DeepZero

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

These 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 back-propagation process. However, they consume a significant amount of GPU memory if staying in GPU, which poses a great challenge for efficient memory usage. The number of model activations increases 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. Here, fine-grained memory optimization techniques refer to tensor-wise operations, whereas coarse-grained memory optimization techniques pertain to operations on the model layer.

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 TQ et al., 2016). The detailed rematerialization process can be found in Fig. 10.

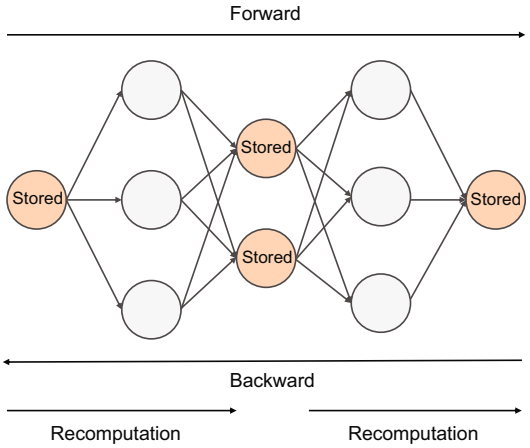

Fig. 10 Rematerialization process. The gray circles represent evicted activations and the rest are stored in GPU in the forward process. In the backward process, the evicted activations are rematerialized from the stored activations. Memory consumption is reduced at the cost of more rematerialization time

In the pioneering work of Chen TQ 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 (Fig. 10). This approach results in sublinear memory cost, enabling the training of larger and deeper models. However, it has limitations as it requires manual selection of checkpointed layers and is not applicable to nonlinear models like ResNet (He KM et al., 2016).

To address the limitations of Chen TQ et al. (2016), Kirisame et al. (2021) designed DTR 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 $s(t)$, computed from the time at which 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 $c_{\mathrm{r}}(t)$. Besides, the recomputation cost of the parent tensors of tensor t, t_{p}, should be taken into consideration. Therefore, the heuristic function of DTR is

$$h_{\mathrm{DTR}}=\frac{c_{\mathrm{r}}(t)+\sum_{t_{\mathrm{p}}\in e(t)}c_{\mathrm{r}}(t_{\mathrm{p}})}{m(t)s(t)}, \tag{2}$$

where $e(t)$ is the set of parent tensors of tensor t. There are other heuristic functions except for the one given in 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) is 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 directed acyclic graph (DAG), $G(V,E)$, where V is the set of vertices and E is the set of 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 whether 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, we have

$$\begin{aligned}&\arg\min_{R,S}\sum_{t=1}^{n}\sum_{i=1}^{t}C_iR_{t,i}\\ &\text{s.t.}\quad R_{t,j}\le R_{t,i}+S_{t,i}\ \ \forall t,\ \forall(v_i,v_j)\in E,\\ &\qquad S_{t,i}\le R_{t-1,i}+S_{t-1,i}\ \ \forall t\ge 2,\ \forall i,\\ &\qquad \sum_i S_{1,i}=0,\ \sum_t R_{t,n}\ge 1,\\ &\qquad R_{t,i},S_{t,i}\in\{0,1\}\quad\forall t,\ \forall i,\end{aligned} \tag{3}$$

where C_i is the time cost from the first operation to the i^{th} operation. These constraints are formulations of the relationship of graph G. Then, integer linear programming (ILP) (Margot, 2010) and the Gurobi tool are used to solve Eq. (3) and obtain the states for each operator. The training process proceeds based on the graph rebuilt with rematerialization, leading to increased input batch size with reduced memory.

There are some other works concerning rematerialization. Beaumont et al. (2020) employed dynamic programming to find the best rematerialization strategies for DNN models. To formalize the optimization problem, they used $\mathcal{G}=(V,E)$ to represent the model graph, where V is the set of vertices and E is the set of edges between vertices, which is similar to the formulation of Checkmate. However, they required further evaluation of real-world datasets and large-scale language models. Besides, they developed an open-source framework regarding rematerialization with PyTorch, Rotor (https://gitlab.inria.fr/hiepacs/rotor). Then, Zhao XY et al. (2023) combined Checkmate (Jain et al., 2020) and Rotor (Herrmann et al., 2019) that focuses on the heterogeneous chains and proposed Rockmate. They considered each model as a sequence of blocks and applied Checkmate to each block in the sequence. Then Rotor was 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 JH et al. (2023) proposed Coop. They tried to evict tensors within a sliding window algorithm. They used a list to store all the tensors' states and sorted these tensors by their addresses, and the tensors in the sliding window were supposed to be evicted. Besides, they defined 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.

5.2 Swapping

Swapping methods involve transporting data between the GPU to other physical devices, such as

CPU or solid state drive (SSD), to reduce the GPU memory usage. Swapping also indicates offloading in some papers (Rajbhandari et al., 2021; Ren J et al., 2021).

Rhu et al. (2016) 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. 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 (CNNs).

SwapAdvisor (Huang CC et al., 2020) is a method proposed based on MXNet (Chen TQ et al., 2015). It takes the model graph as input and uses the genetic algorithm (Holland, 1992) to find the optimal operator schedule in the constructed search space, considering memory allocation and operator schedules. SwapAdvisor first 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 is iterated until the original graph is fully optimized. SwapAdvisor employs Belady's strategy to select 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. (2020) proposed AutoTM, which uses ILP to find the optimal tensor assignment and movement between dynamic random access memory (DRAM) or non-volatile dual in-line memory modules (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. (2021) proposed FlashNeuron, which uses an NVMe SSD as a backing store. They introduced an offloading scheduler to find the optimal offloading schedule with an input batch size and a DNN model. First, 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 full overlap between the computation and offloading processes.

ZeRO-Offload (Ren J 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 uses 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 achieve overlapping between 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 achieves overlapping between computation and communication, 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 maximum 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 DNNs.

SuperNeurons (Wang LN et al., 2018) uses 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 process is represented as m_l^{b}. Therefore, the total memory consumption is $\sum_{i=1}^{N} m_{l_i}^{\mathrm{f}} + \sum_{i=1}^{N} m_{l_i}^{\mathrm{b}}$, where N is the number of model layers. Among these layers, we have $m_{\mathrm{peak}} = \max(l_i)$, where $i = 1, 2, \ldots, N$. SuperNeurons includes a module named liveness analysis, analyzing the liveness of each layer. The memory consumption in the forward process is $\sum_{i=1}^{k} m_{l_i}^{\mathrm{f}}$ at layer k, where $k \leq N$. Liveness analysis reduces the maximum memory consumption from $\sum_{i=1}^{N} m_{l_i}^{\mathrm{f}} + \sum_{i=1}^{N} m_{l_i}^{\mathrm{b}}$ to $\sum_{i=1}^{N} m_{l_i}^{\mathrm{f}} + m_{l_N}^{\mathrm{b}}$. Assuming that the memory cost of each layer is identical, during the backward process, the memory consumption at layer k is $\sum_{i=1}^{k} m_{l_i}^{\mathrm{f}} + m_{l_k}^{\mathrm{b}}$. Therefore, the peak memory is $\sum_{i=1}^{N} m_{l_i}^{\mathrm{f}} + m_{l_N}^{\mathrm{b}}$. In this way, SuperNeurons reduces the memory consumption to $\sum_{i=1}^{N} m_{l_i}^{\mathrm{f}} + m_{l_N}^{\mathrm{b}}$, 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) makes 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) metric to measure the benefit of evicting or swapping tensors:

$$\mathrm{MSPS} = \frac{\text{Memory saving}}{\text{Rematerialization time}}. \tag{5}$$

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

DELTA (Tang et al., 2022) aims to achieve automatic and fine-grained memory optimization for training large-scale models. The optimization process is decoupled into three modules: Filter, Director, and Prefetcher. Filter presents a heuristic function to select the optimal tensor to obtain Release. The heuristic function F_{Filter} is defined by taking staleness time and memory consumption into consideration:

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

The filtering function could 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), $c_{\mathrm{r}}(t)$ was defined as the rematerialization cost while $c_{\mathrm{s}}(t)$ the swapping cost of tensor t. They compared $c_{\mathrm{r}}(t)$ and $c_{\mathrm{s}}(t)$ to obtain a better action for tensor t. If $c_{\mathrm{r}}(t) > c_{\mathrm{s}}(t)$, the rematerialization cost is larger than the swapping cost, and Director returns Offload for the tensor. Otherwise, Director returns Eviction. As far as Prefetcher is concerned, it works in the backward process. Prefetcher is responsible for prefetching those offloaded tensors back to the GPU. Delta employs overlapping strategies to reduce the time overhead caused by eviction and offloading operations. Different from Capuchin (Peng et al., 2020), DELTA is a dynamic manager for training large-scale models. However, DELTA dynamically manages memory for large-scale models, but memory fragmentation remains an unresolved issue.

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

Nie et al. (2022) split the tensors into some micro-tensors. Each of these micro-tensors is a fine-grained unit for a memory operation (Table 3). They built the execution operation schedule based on the computation graph, and finished 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 parallel

Table 3 Actions in reducing model activations

Action	Meaning
Allocation	Allocating GPU memory for tensors
Eviction	Evicting layers/tensors from GPU
Rematerialization	Recomputing layers/tensors back to GPU
Offloading	Moving layers/tensors from GPU to CPU
Reloading	Moving layers/tensors from CPU to GPU
Swapping	Including offloading and reloading
Releasing	Including rematerialization and swapping

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 improvement will be crucial for further advancements in large-scale language models. Therefore, we suggest a few plausible trends in future research.

1. Supporting new parallel methods

3D Parallel has attracted much attention nowadays. Researchers are 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, such as the combination of Data Parallel and Sequence Parallel. With the emergence of new technologies, the performance of existing parallel methods can be further improved, such as FlashAttention (Dao et al., 2022) and FlashAttention-2 (Dao, 2023). They could be combined with Sequence Parallel and speed up 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, 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 the ability to train large-scale models in an automated fashion.

2. 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 hyperparameter settings. Different hyperparameter settings could result in non-convergent 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.

3. 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 memory fragmentation in dynamic methods remains to be settled.

4. Combining memory optimization methods

Currently, memory optimization methods pay attention to one single part concerning model parameters, model states, and model activations. We expect these memory optimization methods to be used together when training large-scale models (Sun Y et al., 2021; Zeng et al., 2021). However, there is a lack of work combining these methods. In reality, large-scale models have been trained with more than 3D Parallel methods (Wang YZ et al., 2023; Yao et al., 2023). Wang YZ et al. (2023) integrated memory optimization (rematerialization and swapping) and parallelism techniques (data parallelism, model parallelism, and pipeline parallelism). 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, as 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 runtime optimization applied to the profiled model. Therefore, we expect memory optimization could be considered in auto parallelism and large-scale models can be trained with limited memory in an automated fashion.

5. Reducing 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 AllReduce 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 analyzed 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 discussed the future of training large-scale models with limited memory.

Contributors

Yu TANG proposed this idea and wrote Sections 1, 3.3, 5, and 7. Linbo QIAO and Dongsheng LI are responsible for overseeing the entire survey paper and ensuring it is comprehensive. Linbo QIAO wrote Section 4. Lujia YIN wrote Section 2. Peng LIANG wrote Sections 3.1 and 6. Ao SHEN wrote Section 3.2. Zhilin YANG and Lizhi ZHANG drew figures and contributed to discussions. Yu TANG revised and finalized the paper.

Conflict of interest

Dongsheng LI is a corresponding expert of *Frontiers of Information Technology & Electronic Engineering*, and he was not involved with the peer review process of this paper. All the authors declare that they have no conflict of interest.

References

Abadi M, Barham P, Chen JM, et al., 2016. TensorFlow: a system for large-scale machine learning. Proc 12th USENIX Conf on Operating Systems Design and Implementation, p.265-283.

Acun B, Murphy M, Wang XD, 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. https://doi.org/10.1016/0925-2312(93)90006-O

Bachlechner T, Majumder BP, Mao H, et al., 2021. ReZero is all you need: fast convergence at large depth. Proc 37th 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 19th 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 32nd 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. Int Conf on Machine Learning, p.1826-1837.

Beaumont O, Herrmann J, Pallez G, et al., 2020. Optimal memory-aware backpropagation of deep join networks. *Phil 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 35th 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 34th 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 34th Int Conf on Neural Information Processing Systems, Article 75.

Chen JF, Zheng LM, Yao ZW, et al., 2021. ActNN: reducing training memory footprint via 2-bit activation compressed training. Proc 38th 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 8th 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 Inform Sci Technol*, 37(1):51-89. https://doi.org/10.1002/aris.1440370103

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

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

Dao T, Fu D, Ermon S, et al., 2022. FlashAttention: fast and memory-efficient exact attention with IO-awareness. Proc 36th Conf on Neural Information Processing Systems, p.16344-16359.

Dean J, Corrado GS, Monga R, et al., 2012. Large scale distributed deep networks. Proc 25th Int Conf on Neural Information Processing Systems, p.1223-1231.

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, 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 33rd Int Conf on Neural Information Processing Systems, Article 1170.

Fan SQ, Rong Y, Meng C, et al., 2021. DAPPLE: a pipelined data parallel approach for training large models. Proc 26th 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 Parall Distrib Syst*, 34(1):304-315. https://doi.org/10.1109/TPDS.2022.3219819

Fedus W, Zoph B, Shazeer N, 2022. Switch Transformers: scaling to trillion parameter models with simple and efficient sparsity. *J Mach Learn Res*, 23(1):120.

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 37th Int Conf on Machine Learning, Article 309.

Gholami A, Yao ZW, Kim S, et al., 2024. AI and memory wall. *IEEE Micro*, 44(3):33-39. 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 Patt 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 38th 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 25th 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 25th 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 33rd Conf on Neural Information Processing Systems, Article 10.

Jain P, Jain A, Nrusimha A, et al., 2020. Checkmate: breaking the memory wall with optimal tensor rematerialization. Proc 3rd Conf on Machine Learning and Systems, p.497-511.

Ji SW, Xu W, Yang M, et al., 2013. 3D convolutional neural networks for human action recognition. *IEEE Trans Patt 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. Proc 2nd Conf on Machine Learning and Systems, p.1-13.

Kang WC, Cheng DZ, Yao TS, et al., 2021. Learning to embed categorical features without embedding tables for recommendation. Proc 27th 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 3rd Int Conf on Learning Representations.

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

Kitaev N, Kaiser Ł, Levskaya A, 2020. Reformer: the efficient Transformer. Proc 8th 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 V, Casper J, Lym S, et al., 2023. Reducing activation recomputation in large Transformer models. Proc 6th Conf on Machine Learning and Systems, 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.

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 11th USENIX Conf on Operating Systems Design and Implementation, p.583-598.

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, 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

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 52nd Int Conf on Parallel Processing, p.766-775. https://doi.org/10.1145/3605573.3605613

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 6th 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 32nd Int Conf on Neural Information Processing Systems, p.3731-3741.

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. https://doi.org/10.1145/3581784.3607073

Luo Y, Ren XZ, Zheng ZW, et al., 2023. CAME: confidence-guided adaptive memory efficient optimization. Proc 61st 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 27th 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: Jünger 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. Springer, Berlin, p.647-686. 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 27th 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 38th 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 33rd Int Conf on Neural Information Processing Systems, Article 721.

Peng X, Shi XH, Dai HL, et al., 2020. Capuchin: tensor-based GPU memory management for deep learning. Proc 25th 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 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.

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. 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 39th 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. Proc Conf 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 26th 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. Proc 28th Int Conf on Neural Information Processing Systems, p.91-99.

Rhu M, Gimelshein N, Clemons J, et al., 2016. vDNN: virtualized deep neural networks for scalable, memory-efficient neural network design. Proc 49th 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 27th 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 35th 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.

Sun X, Wang NG, Chen CY, et al., 2020. Ultra-low precision 4-bit training of deep neural networks. Proc 34th Int Conf on Neural Information Processing Systems, Article 152.

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 30th Int Conf on Machine Learning, p.1139-1147.

Sutskever I, Vinyals O, Le QV, 2014. Sequence to sequence learning with neural networks. Proc 27th Int Conf on Neural Information Processing Systems, p.3104-3112.

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 16th 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 31st Int Conf on Neural Information Processing Systems, p.6000-6010.

Wang LN, Ye JM, Zhao YY, et al., 2018. SuperNeurons: dynamic GPU memory management for training deep neural networks. Proc 23rd 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., 2018. Training deep neural networks with 8-bit floating point numbers. Proc 32nd Int Conf on Neural Information Processing Systems, p.7686-7695.

Wang YZ, Han X, Zhao WL, et al., 2023. H3T: efficient integration of memory optimization and parallelism for high-throughput Transformer training. Proc 37th Conf on Neural Information Processing Systems.

Weinberger K, Dasgupta A, Langford J, et al., 2009. Feature hashing for large scale multitask learning. Proc 26th 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 37th Conf on Neural Information Processing Systems.

Xiong RB, Yang YC, He D, et al., 2020. On layer normalization in the Transformer architecture. Proc 37th Int Conf on Machine Learning, p.10524-10533.

Xu QM, Siyamwala H, Ghosh M, et al., 2015. Performance analysis of NVMe SSDs and their implication on real world databases. Proc 8th 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 8th Int Conf on Learning Representations.

Yuan JH, Li XQ, Cheng C, et al., 2022. 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 15th European Conf on Computer Vision, p.373-390. 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 34th 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., 2023. Rockmate: an efficient, fast, automatic and generic tool for re-materialization in PyTorch. Proc 40th Int Conf on Machine Learning, p.42018-42045.

Zhao YL, Gu A, Varma R, et al., 2023. 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 16th 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. https://openreview.net/forum?id=gfr5yILQc7_ [Accessed on Sept. 17, 2023].

Zhou J, Ke P, Qiu XP, et al., 2024. ChatGPT: potential, prospects, and limitations. *Front Inform Technol Electron Eng* 25(1):6-11. https://doi.org/10.1631/FITEE.2300089

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. https://arxiv.org/abs/2202.00089