计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330554
61(4): 809−823, 2024

从 BERT 到 ChatGPT: 大模型训练中的存储系统挑战与技术发展

冯杨洋　汪　庆　谢旻晖　舒继武
（清华大学计算机科学与技术系　北京　100084）
（fyy21@mails.tsinghua.edu.cn）

From BERT to ChatGPT: Challenges and Technical Development of Storage Systems for Large Model Training

Feng Yangyang, Wang Qing, Xie Minhui, and Shu Jiwu
（*Department of Computer Science and Technology*, *Tsinghua University*, *Beijing* 100084）

Abstract　The large models represented by ChatGPT have attracted a lot of attention from industry and academia for their excellent performance on text generation and semantic understanding tasks. The number of large model parameters has increased tens of thousands of times in three years and is still growing, which brings new challenges to storage systems. First, we analyze the storage challenges of large model training, pointing out that large model training has unique computation patterns, storage access patterns, and data characteristics, which makes traditional storage techniques inefficient in handling large model training tasks. Then, we describe three types of storage acceleration techniques and two types of fault-tolerant techniques. The storage acceleration techniques for large model training include: 1) distributed storage technique based on large model computation patterns designs the partitioning, storage, and transferring strategies of model data in distributed clusters based on the partitioning of large model computation tasks and the dependencies between computation tasks; 2) heterogeneous storage access pattern-aware technique for large model training develops data prefetching and transferring strategies among heterogeneous devices with the predictability of storage access patterns in large model training; 3) large model data reduction technique reduces the data size in the model training process according to the characteristics of large model data. The storage fault-tolerant techniques for large model training include: 1) parameter checkpointing technique stores the large model parameters to persistent storage devices; 2) redundant computation technique computes the same version of parameters repeatedly in multiple GPUs. Finally, we give the summary and suggestions for future research.

Key words　ChatGPT; large model; storage system; fault-tolerant; large model training system

摘　要　以 ChatGPT 为代表的大模型在文字生成、语义理解等任务上表现卓越，引起了工业界和学术界的广泛关注. 大模型的参数量在 3 年内增长数万倍，且仍呈现增长的趋势. 首先分析了大模型训练的存储挑战，指出大模型训练的存储需求大，且具有独特的计算模式、访存模式、数据特征，这使得针对互联网、大数据等应用的传统存储技术在处理大模型训练任务时效率低下，且容错开销大. 然后分别阐述了针对大模型训练的 3 类存储加速技术与 2 类存储容错技术. 针对大模型训练的存储加速技术包括：1）基于大模型计算模式的分布式显存管理技术，依据大模型计算任务的划分模式和计算任务间的依赖关系，设计模型数据在分布式集群中的划分、存储和传输策略；2）大模型训练访存感知的异构存储技术，借助大模型训练中的访存模式可预测的特性，设计异构设备中的数据预取和传输策略；3）大模型数据缩减技术，针

收稿日期：2023-07-03；**修回日期**：2023-11-27
基金项目：国家自然科学基金项目（U22B2023）
This work was supported by the National Natural Science Foundation of China (U22B2023).
通信作者：舒继武（shujw@tsinghua.edu.cn）

对大模型数据的特征,对模型训练过程中的数据进行缩减.针对大模型训练的存储容错技术包括:1)参数检查点技术,将大模型参数存储至持久化存储介质;2)冗余计算技术,在多张GPU中重复计算相同版本的参数.最后给出了总结和展望.

关键词 ChatGPT;大模型;存储系统;容错;大模型训练系统

中图法分类号 TP391

OpenAI推出的人工智能聊天机器人ChatGPT在发布后的2个月内用户数量就突破了1亿,成为历史上用户增长最快速的应用软件.其对文本卓越的理解和生成能力引起了工业界和学术界的广泛关注.ChatGPT使用了基于Transformer[1]结构的大规模深度学习模型,即大模型.相较于传统的深度学习模型结构,Transformer结构可以将模型的参数量推向更高的数量级.如图1所示,2020年由谷歌提出的BERT[2]模型的参数量为亿级,而ChatGPT所基于的GPT-3.5的参数量已达到千亿级,其下一代的GPT-4[3]的参数量甚至达到万亿级.

随着大模型的参数量持续增长,其训练过程中的数据量不断增加,对存储的需求也在日益增加.但是,以往为互联网、大数据等应用设计的存储技术不适用于大模型训练.在存储性能方面,这些传统的存储技术并未针对大模型训练的计算模式、访存模式和数据特征设计,无法在训练过程中充分发挥硬件的性能,影响了训练效率.在存储容错方面,大模型训练过程中的参数量庞大且更新频繁,参数数据之间存在依赖关系,传统的存储容错技术会为大模型训练带来巨大的容错开销.

在存储性能方面,现有工作提出了针对大模型训练的存储加速技术.这些技术可以总结为3类:基于大模型计算模式的分布式显存管理技术、大模型训练访存感知的异构存储技术和大模型数据缩减技术.基于大模型计算模式的分布式显存管理技术依据训练中计算任务的划分和计算任务之间的依赖关系,设计模型数据在GPU集群中的分布和传输策略,减少数据在分布式集群中传输带来的通信开销.大模型训练访存感知的异构存储技术利用模型训练过程中的访存模式可预测的特性,设计了数据的预取和卸载策略,以充分利用计算掩盖数据在各种存储介质间移动带来的开销.大模型数据缩减技术依据模型的数据特征,通过修改模型训练中的流程或降低数据精度,减少了模型训练中对存储空间的需求.

在存储容错方面,现有工作针对大模型训练的存算模式提出了模型参数检查点技术和冗余计算技术.模型参数检查点技术定期将数据保存至持久化介质中以保证模型数据在设备故障后不丢失;冗余计算技术则在多张GPU中重复计算相同版本的参数,以对模型训练数据容错.

1 大模型训练的存储挑战

大模型训练的存储容量需求高.大模型不仅参数量庞大,在训练的每一步还需要保存前向传播过程中产生的激活量和用于参数更新的优化器.

1)模型参数(model parameters).近年来,大规模深度学习模型的参数量呈爆炸式增长,且这种趋势仍在持续,如谷歌于2020年提出了亿级参数量的BERT,OpenAI于2021年和2023年分别提出了千万亿级参数量的GPT-3和万亿级参数量的GPT-4.海量

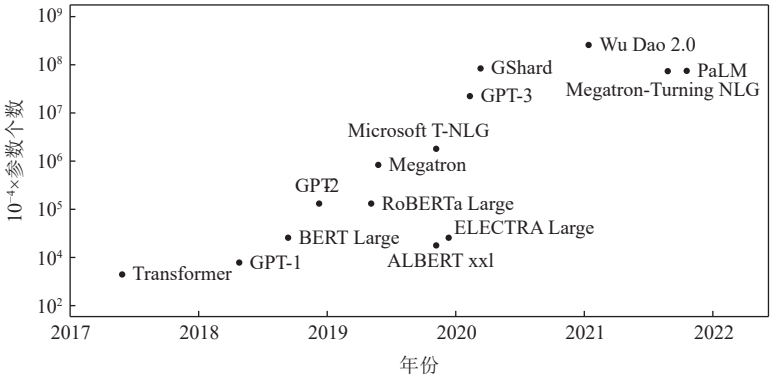


Fig. 1 Increase of model size

图1 模型规模增长

的模型参数需要大量的存储空间以支持大模型训练. 以训练 GPT-3 为例，其至少需要约 800 GB 的显存空间，约等于 10 张 H100 GPU 的显存总和.

2）激活量（activation）. 根据反向传播算法，前向传播阶段产生的激活量需要先被保存，然后在反向传播时用于计算梯度信息，最后在梯度计算结束后被释放. 在大模型训练过程中，激活量所占的存储空间庞大. 根据文献 [4]，激活量在训练过程中的存储开销占总存储开销的 70%. 这主要由 3 个方面导致：①随着模型层数的增多，所需要保存的激活量也随之增加；②在大模型中单层特征的维度较大，这导致激活量的特征维度增大；③在模型训练中，每一步会使用更多的样本以提高 GPU 的计算利用率，这会导致激活量的样本维度增加.

3）优化器（optimizer）. 计算得到的梯度被传输到优化器中以更新得到新版本的模型参数. 为让模型参数更新时能尽可能逼近最优值，许多不同的优化器被提出，如 SGD[5] 和 Adam[6]. 这些优化器需要保存额外的信息，如 Adam 需要保存与梯度规模一致的一阶动量和二阶动量，其大小是模型参数的 2 倍.

此外，大模型训练过程中的容错需求高. 大模型训练会使用大量 GPU，这提高了故障的可能性[7]. 例如，Meta 团队在训练 OPT-175B 的时候出现了百余次故障[8]；微软的集群中平均 45 min 就会出现 1 次 DNN 训练作业故障[9].

因此，大模型训练需要支持大规模数据的存储系统. 传统支持大规模数据存储系统所使用的存储技术可以分为 3 类[10-12]：基于同构存储介质的分布式存储技术、基于异构存储介质的异构存储技术和数据缩减技术. 传统的大规模数据存储系统还会采用日志、多副本以及纠删码等方式对数据进行容错. 但是，这些传统的大规模数据存储技术是针对互联网、大数据等应用而设计的. 这些技术一方面未充分利用大模型训练中的计算模式、访存模式和数据特征，另一方面不适用于大模型数据更新数据量大、更新频繁的特点，严重影响大模型训练的效率，其具体表现有 4 个方面：

1）传统的分布式存储技术不适用于大模型训练的计算模式. 大模型常使用多机多卡的配置进行分布式训练，因此其数据也需要被存储到多张 GPU 显存中. 但传统的分布式存储技术并不适用于大模型训练的计算模式. 一方面，大模型训练常使用的 GPU 具有计算资源和存储资源强耦合的特点. 系统在对其上的计算任务进行划分时，需要综合考虑计算任务与存储之间的依赖关系. 但传统的分布式存储技术并未针对大模型训练的这一特征进行设计，这可能导致数据不能及时传输至负责计算的 GPU 上，影响训练效率. 另一方面，传统的分布式存储技术未利用大模型训练中各个任务间的数据依赖关系进行优化，这可能导致相邻任务间的数据传输方案非最优，增加任务间通信的开销.

2）传统的异构存储技术对大模型训练中的访存模式不感知. 在传统异构存储系统中，由于数据链路带宽低、存储介质带宽差异大，数据在不同存储介质间的移动开销高. 但是，大模型训练中的访存模式可预测. 系统可以利用数据的访存信息对模型数据进行预取，以减少数据在不同介质间的移动开销对计算产生的影响. 传统的异构存储系统并未利用这些访存模式设计数据的预取和传输策略，因而无法达到训练的最佳性能.

3）传统的存储缩减技术不适用于大模型训练中的数据特征. 在大模型训练过程中，参数和中间变量常采用全精度浮点数或半精度浮点数表示，以保证训练的精度，单位数据的存储占用高. 而且这些数据的稠密度高，难以通过传统的压缩方法缓解存储压力.

4）传统的存储容错技术在大模型训练场景下容错开销大. 大模型训练中的数据量庞大，并且数据更新频繁. 每一轮训练都会更新整个模型的参数以及部分优化器的参数. 若系统使用传统的数据容错技术对此类数据进行容错，会导致大量并且频繁的读写操作，降低了模型训练的效率.

近些年来，大量面向大模型训练的存储技术被提出. 其中一类工作着眼于大模型训练中的存储性能. 这类工作通过统一管理分布式显存、利用异构存储介质以及压缩训练数据的方式满足大规模模型训练中对存储容量的需求. 另一类工作着眼于大模型训练中的存储容错. 这类工作通过持久化存储模型训练的参数、在多 GPU 上冗余训练模型参数以实现对模型训练数据的容错.

2 面向大模型的存储加速技术

2.1 基于大模型计算模式的分布式显存管理技术

大模型训练中的计算模式可以分为层间并行和层内并行. 这 2 种计算模式对应不同的任务划分方式和任务依赖关系. 其中，层间并行以张量为粒度将模型划分到多张 GPU 上，单个计算任务仅依赖于负责计算相邻张量的任务；层内并行则是以张量的某一

维度为粒度,将单个张量的计算拆分到多张 GPU 上,单个计算任务依赖于多个负责计算相邻张量分片的任务.

若仅简单地将模型数据划分到多张 GPU 上,而不考虑模型训练任务划分和任务间的依赖关系,会从两方面增加训练过程中的通信开销:一方面,数据存储的位置可能和计算任务所在的位置不一致,导致计算前需要先从远端 GPU 传输所需的数据;另一方面,相邻任务间的数据传输所采用的通信链路或者通信路径非最优.

因此,现有工作依据大模型训练的 2 种计算模式的任务划分方式和任务间的依赖关系,对模型数据进行划分.其中:模型层间分片根据层间并行的计算模式,以张量为粒度对模型数据进行拆分;模型层内分片根据层内并行的计算模式,以张量中的某一维度为粒度对模型数据进行拆分.

2.1.1 基于模型层间并行的分布式显存管理技术

模型层间分片方式是以张量为粒度,将模型数据划分成多个小片,并分布式地存储到多张 GPU 中.此类工作在训练过程中传输的数据类型可以分为 2 类:模型数据和激活量.在传输模型数据的工作中,每张 GPU 负责所有模型分片的训练;在训练前,所需的模型数据被传输到负责计算的 GPU 显存中;在训练后,GPU 将释放不由此 GPU 存储的模型分片所占的显存空间.在传输激活量的工作中,每张 GPU 仅负责训练显存中存储的模型分片,并传输相邻分片的激活量.

ZeRO-DP[13] 是微软公司提出的大模型训练系统,它采用数据并行的方式对模型进行训练.为满足模型训练过程中对存储空间的需求,ZeRO-DP 采用层间分片的方式存储模型,并训练过程中采用在 GPU 间传输模型数据的方案.

如图 2 所示,ZeRO-DP 将模型数据分为优化器($\boldsymbol{P}_{os}$)、梯度($\boldsymbol{P}_{g}$)和模型参数($\boldsymbol{P}_{p}$).对于优化器,ZeRO-DP 将它按层均等地划分成多份.每张 GPU 仅存储某一份优化器分片,并仅负责更新其存储的优化器分片对应的模型参数.在每张 GPU 完成模型参数更新之后,ZeRO-DP 将更新得到的模型参数传输到其他 GPU 中,覆盖旧版本模型参数.对于梯度,ZeRO-DP 将每一份梯度分片聚合到存储有对应参数分片的 GPU 中,从而减少全局的梯度数据存储.对于模型参数,ZeRO-DP 将每一份参数分片存放在带有对应的优化器分片的 GPU 上.在训练过程中,若所需的模型分片不在负责计算的 GPU 上,该 GPU 则需从其他

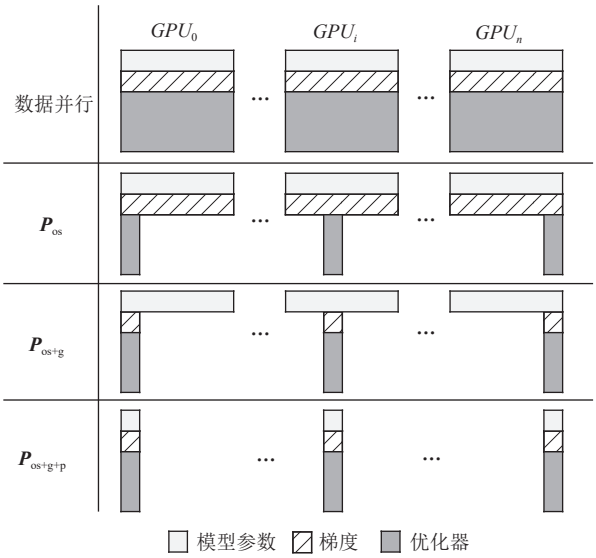


Fig. 2 Illustration of model partition in ZeRO-DP [13]

图 2 ZeRO-DP 模型划分示意图[13]

GPU 拉取模型分片,并在计算结束之后释放对应的显存空间.

用户可以选择是否对模型参数进行划分.若不对模型参数进行划分,仅对优化器和梯度进行划分($\boldsymbol{P}_{os+g}$),则训练过程中的通信操作仅有归约(All-Reduce)梯度和广播(Broadcast)更新后的参数,但系统可训练的模型规模受限于单张 GPU 的显存大小.如需要扩大训练的模型规模,则要对模型参数进行划分($\boldsymbol{P}_{os+g+p}$),但每次前向传播和反向传播前均需对所有参数进行广播,相比于 $\boldsymbol{P}_{os+g}$ 增加了 1 次广播通信.用户可以根据 GPU 通信和显存的大小选择所需的划分组合.

流水线并行训练模式使用了层间分片的方式存储模型.在训练过程中,流水线中的 GPU 间仅传输相邻分片的激活量.在流水线并行训练中,模型首先被划分成多个子模型,每张 GPU 负责计算某一个子模型.在训练过程中,一批样本(minibatch)被拆分成多份子样本(microbatch),系统以流水线的方式同时使用多份子样本进行训练,每张 GPU 仅向存储相邻子模型的 GPU 传输连接 2 个子模型的激活量.

GPipe[14] 是谷歌提出的基于流水线并行的模型训练框架.如图 3 所示,它首先使用一个样本中的所有子样本进行前向传播,接着对所有子样本进行反向传播,最后根据此样本训练得到的梯度进行参数更新.由于前向传播和反向传播的顺序不同,并且需要在使用一批样本完成训练后暂停模型训练以对参数进行更新,GPipe 产生了较多的计算空闲时间(即气泡).

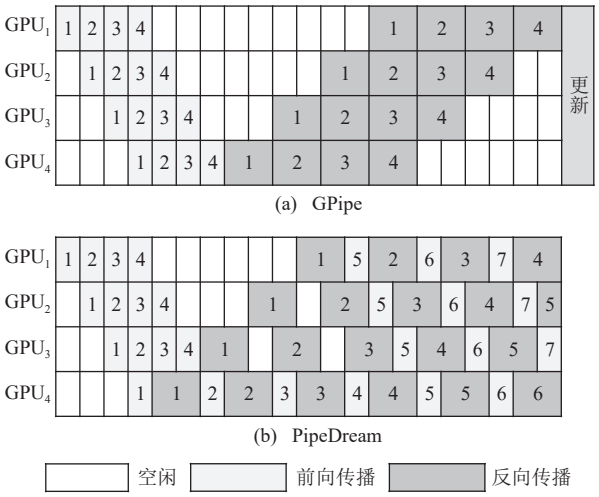


Fig. 3 Illustration of GPipe[14] and PipeDream[15]

图 3 GPipe[14] 和 PipeDream[15] 示意图

PipeDream[15] 是英伟达公司提出的基于流水线并行的模型训练模式，它在流水线中将前向传播和反向传播交替进行（1 forward 1 backward, 1F1B），解决了 GPipe 中的气泡问题. 在 PipeDream 中，某一子样本完成前向传播传播后会立即开始此子样本的反向传播. 当 PipeDream 启动并稳定后，流水线中将不存在气泡. 但是，PipeDream 中需要存放多个版本的参数，增加了额外的存储开销. 在 1F1B 中，某一子样本反向传输结束之后，子模型的参数立即被更新. 此时，流水线中仍存在着使用原版本参数进行前向传播但未完成反向传播的子样本. 为保证单个样本前向传播和反向传播所使用的子模型参数版本一致，需要在 GPU 中保存多个版本的模型数据，这增大了训练过程中的存储开销.

此外，PipeDream 的多版本参数还会影响收敛性. 不同于 GPipe 中所有子模型在完成一轮前向反向传播后同步更新参数，PipeDream 中的子模型在完成一个子样本的反向传播后立即更新参数. 这会导致训练中单个子样本在不同子模型使用的参数版本不一致，从而影响模型收敛.

表 1 所示总结了 GPipe, PipeDream 以及其他流水线在显存开销、气泡和收敛性上的差异. PipeDream-2BW[17] 沿用 1F1B 的计算模式，同时通过梯度累积减少了参数更新频率，进而降低了所需存储的模型参数版本数量. DAPPLE[18] 则在 PipeDream 中加入了类似于 GPipe 的梯度同步更新，综合了 GPipe 和 PipeDream 两者的优势. Chimera[19] 则提出了双头流水线的训练模式，通过增加一定量的存储开销减少启动流水线时带来的气泡.

Table 1 Comparison of Large Model Training Schemes Based on Different Pipeline Modes

表 1 基于不同流水线模式的大模型训练方案对比

系统	显存开销	气泡	收敛性
GEMS[16]	中	差	好
GPipe[14]	低	差	好
PipeDream[15]	高	好	差
PipeDream2BW[17]	中	好	差
DAPPLE[18]	低	差	好
Chimera[19]	中	中	好

2.1.2 基于模型层内并行的分布式显存管理技术

张量并行采用模型层内分片的策略，该策略以张量的某一维度为粒度对模型进行切分，并将切分得到的分片存储在多张 GPU 中. 相比于模型的层间分片以层为粒度存储数据，模型的层内分片将某一层的数据以更细的粒度拆分. 因此，模型的层内分片可将单层的数据存储到更多 GPU 中，从而支持单层数据量更大的模型[20].

如图 4 所示，张量可以从行和列 2 个维度被切分. 以列对张量 $\boldsymbol{A}$ 切分时，每一个张量的分片的计算需要完整的输入 $\boldsymbol{X}$，最后需要将所有分片计算得到的结果拼接以得到完整的结果. 以行为粒度对张量 $\boldsymbol{A}$ 切分时，输入 $\boldsymbol{X}$ 也需对应按列进行切分，最后需要将所有分片计算得到的结果相加以得到完整的计算结果.

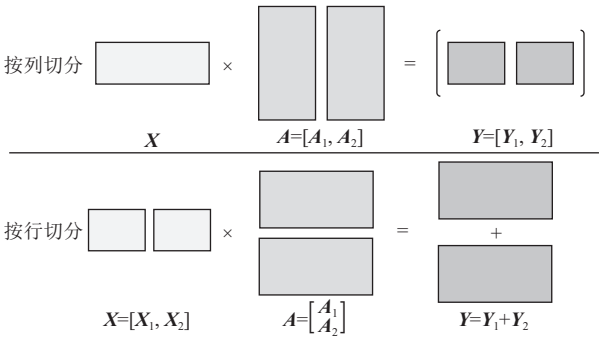


Fig. 4 Tensor splitting[20]

图 4 张量切分[20]

根据是否切分输入的张量和张量存放的方式，张量并行分为 4 种不同的模式. 其中每种张量并行的存储开销和通信开销的对比如表 2 所示.

1）1D 张量并行

1D 张量并行仅按行和列两者中的某一维度对张量进行切分. 以式（1）的矩阵乘法（GEMM）为例，假设有 P 张 GPU，且 P=2，其中的参数 $\boldsymbol{A}$ 若按列被切分，如式（2）所示，每一参数切片与输入相乘得到的子结果如式（3）所示.

Table 2 Comparison of Four Types of Tensor Parallelism

表 2 4 种张量并行的对比

张量并行维度	参数张量的存储开销	输入张量的存储开销	传输参数张量的通信开销	传输输入张量的通信开销
1D	低	高	无	中
2D	低	低	中	中
2.5D	中	低	中	低
3D	低	低	低	低

$$\boldsymbol{Y}=\boldsymbol{XA}, \tag{1}$$

$$\boldsymbol{A}=[\boldsymbol{A}_1,\boldsymbol{A}_2], \tag{2}$$

$$\boldsymbol{Y}=[\boldsymbol{XA}_1,\boldsymbol{AX}_2]=[\boldsymbol{Y}_1,\boldsymbol{Y}_2]. \tag{3}$$

下一个与之相邻的线性 GEMM 层参数 $\boldsymbol{B}$ 则按列进行切分. 如式(4)所示, 每一个参数切片与上一层的子结果相乘, 得到的结果通过 All-Reduce 以得到最终的计算结果.

$$\boldsymbol{B}=\begin{bmatrix}\boldsymbol{B}_1\\ \boldsymbol{B}_2\end{bmatrix}, \tag{4}$$

$$\boldsymbol{Z}=[\boldsymbol{Y}_1,\boldsymbol{Y}_2]\begin{bmatrix}\boldsymbol{B}_1\\ \boldsymbol{B}_2\end{bmatrix}=\boldsymbol{Y}_1\boldsymbol{B}_1+\boldsymbol{Y}_2\boldsymbol{B}_2. \tag{5}$$

在 1D 张量切分方式中, 单张 GPU 存储 $1/P$ 份参数张量和 1 份完整的输入张量. 在计算过程中需要传输 $P-1$ 份的输入张量.

2)2D 张量并行

1D 张量并行虽然将参数分布式存储在每一张 GPU 上, 但每一张 GPU 需要冗余保存一份完整的输入. 2D 张量并行[21]将参数和输入同时按照行和列进行切分, 并分布式保存在多张 GPU 上. 以 GEMM 为例, 假设有 $P=q\times q$张 GPU, 其中 $q=2$, GEMM 中的参数 $\boldsymbol{A}$ 同时按列和行进行切分, 与之对应的输入 $\boldsymbol{X}$ 也按照行和列进行切分, 切分结果如式(6)(7)所示:

$$\boldsymbol{A}=\begin{bmatrix}\boldsymbol{A}_{00} & \boldsymbol{A}_{01}\\ \boldsymbol{A}_{10} & \boldsymbol{A}_{11}\end{bmatrix}, \tag{6}$$

$$\boldsymbol{X}=\begin{bmatrix}\boldsymbol{X}_{00} & \boldsymbol{X}_{01}\\ \boldsymbol{X}_{10} & \boldsymbol{X}_{11}\end{bmatrix}. \tag{7}$$

计算分为 q 步. 以某一步为例, 首先 $\boldsymbol{X}_{i0}$ 在存储带有相同行的输入的 GPU 间被广播; $\boldsymbol{A}_{0j}$ 在存储有相同列的模型参数的 GPU 间被广播. 然后每张 GPU 上的模型和参数相乘. 其他步骤的计算与之类似. 最后每一步计算得到的子结果相加得到最终结果, 如式(8)所示.

$$\boldsymbol{Y}=\boldsymbol{XA}=\begin{bmatrix}\boldsymbol{X}_{00}\boldsymbol{A}_{00}+\boldsymbol{X}_{01}\boldsymbol{A}_{10} & \boldsymbol{X}_{00}\boldsymbol{A}_{01}+\boldsymbol{X}_{01}\boldsymbol{A}_{11}\\ \boldsymbol{X}_{10}\boldsymbol{A}_{00}+\boldsymbol{X}_{11}\boldsymbol{A}_{10} & \boldsymbol{X}_{10}\boldsymbol{A}_{01}+\boldsymbol{X}_{11}\boldsymbol{A}_{11}\end{bmatrix}. \tag{8}$$

在 2D 张量并行中, 单个处理器内存储 $1/P$ 份参数张量和$1/P$份输入张量. 但在计算过程中需要传输 $q-1$ 份参数张量和输入张量.

3)2.5D 张量并行

相比于 1D 张量并行, 2D 张量并行的通信开销大, 因此文献 [22] 提出了 2.5D 张量并行. 2.5D 张量并行将输入张量按行列切分后, 再将切分后的分片按行分组. 以 GEMM 为例, 在 $P=q\times q\times q$张 GPU 中, 其中 $q=2$, 输入张量首先被划分为$q\times q\times q$份, 然后得到的张量分片再被划分成 q 组, 划分结果如式(9)所示. 参数则仍按照式(7)划分, 并冗余 q 份存储在与输入 q 组对应的 GPU 中.

$$\boldsymbol{X}=\begin{bmatrix}\boldsymbol{X}_{00} & \boldsymbol{X}_{01}\\ \boldsymbol{X}_{10} & \boldsymbol{X}_{11}\\ \boldsymbol{X}_{20} & \boldsymbol{X}_{21}\\ \boldsymbol{X}_{30} & \boldsymbol{X}_{31}\end{bmatrix}=\begin{bmatrix}\begin{bmatrix}\boldsymbol{X}_{00} & \boldsymbol{X}_{01}\\ \boldsymbol{X}_{10} & \boldsymbol{X}_{11}\end{bmatrix}\\ \begin{bmatrix}\boldsymbol{X}_{20} & \boldsymbol{X}_{21}\\ \boldsymbol{X}_{30} & \boldsymbol{X}_{31}\end{bmatrix}\end{bmatrix}. \tag{9}$$

在计算过程中, 每一组中按照 2D 张量并行的方式进行分布计算, 由此得到的最后的结果如式(10)所示.

$$\boldsymbol{Y}=\begin{bmatrix}\boldsymbol{X}_{00}\boldsymbol{A}_{00}+\boldsymbol{X}_{01}\boldsymbol{A}_{10} & \boldsymbol{X}_{00}\boldsymbol{A}_{01}+\boldsymbol{X}_{01}\boldsymbol{A}_{11}\\ \boldsymbol{X}_{10}\boldsymbol{A}_{00}+\boldsymbol{X}_{11}\boldsymbol{A}_{10} & \boldsymbol{X}_{10}\boldsymbol{A}_{01}+\boldsymbol{X}_{11}\boldsymbol{A}_{11}\\ \boldsymbol{X}_{20}\boldsymbol{A}_{00}+\boldsymbol{X}_{21}\boldsymbol{A}_{10} & \boldsymbol{X}_{20}\boldsymbol{A}_{01}+\boldsymbol{X}_{21}\boldsymbol{A}_{11}\\ \boldsymbol{X}_{30}\boldsymbol{A}_{00}+\boldsymbol{X}_{31}\boldsymbol{A}_{10} & \boldsymbol{X}_{31}\boldsymbol{A}_{01}+\boldsymbol{X}_{31}\boldsymbol{A}_{11}\end{bmatrix}. \tag{10}$$

2.5D 张量并行牺牲了少量存储空间以减少了传输输入张量带来的通信开销. 在 2.5D 张量并行中, 单张 GPU 需要存储$1/q^2$份参数张量和$1/P$份输入张量; 在计算过程中, 2.5D 张量并行需要传输$(q-1)\times q$份的参数张量和 $q-1$份的输入张量.

4)3D 张量并行

3D 张量[23]并行在 2.5D 张量并行的基础上, 将参数的分片按列进行分组, 并与输入的分组对应. 以 GEMM 为例, P 张 GPU 按照 $q\times q\times q$分组, 其中 $q=2$, 则输入的分组如式(11)所示, 参数的分组如式(12)所示. 其中$\boldsymbol{X}_{ijk}$和$\boldsymbol{A}_{kji}$被存储在第(i,j,k)张 GPU 上.

$$\boldsymbol{X}=\begin{bmatrix}\boldsymbol{X}_{000} & \boldsymbol{X}_{001}\\ \boldsymbol{X}_{010} & \boldsymbol{X}_{011}\\ \boldsymbol{X}_{100} & \boldsymbol{X}_{101}\\ \boldsymbol{X}_{110} & \boldsymbol{X}_{111}\end{bmatrix}, \tag{11}$$

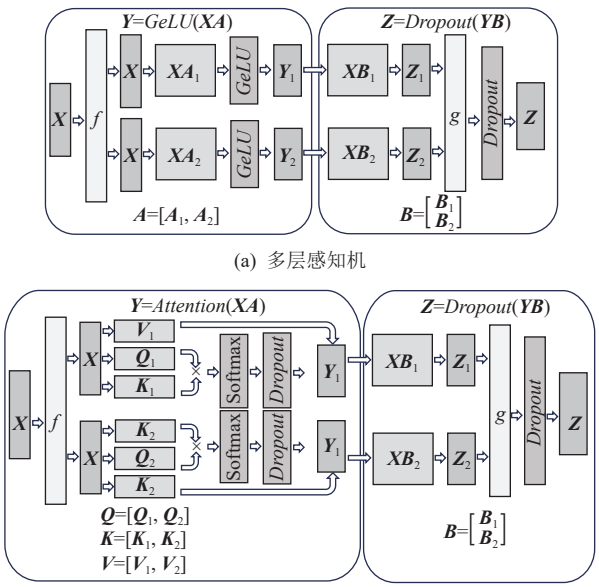
$$\boldsymbol{A}=\begin{bmatrix}\boldsymbol{A}_{000} & \boldsymbol{A}_{001} & \boldsymbol{A}_{010} & \boldsymbol{A}_{001}\\ \boldsymbol{A}_{100} & \boldsymbol{A}_{101} & \boldsymbol{A}_{110} & \boldsymbol{A}_{111}\end{bmatrix}. \tag{12}$$

在计算过程中, $\boldsymbol{X}_{ijk}$在 j 所在的 GPU 组中被广播, $\boldsymbol{A}_{kji}$在 i 所在的 GPU 组中被广播. 因此第(i,j,k)张 GPU 中有$\boldsymbol{X}_{ik},\boldsymbol{A}_{kj}$, 并将这两者相乘. 最后的子结果被 Reduce-Scatter 到所有 GPU 中.

$$\boldsymbol{Y}=\begin{bmatrix}\boldsymbol{Y}_{000} & \boldsymbol{Y}_{001}\\ \boldsymbol{Y}_{010} & \boldsymbol{Y}_{011}\\ \boldsymbol{Y}_{100} & \boldsymbol{Y}_{101}\\ \boldsymbol{Y}_{110} & \boldsymbol{Y}_{111}\end{bmatrix}. \tag{13}$$

在 3D 张量并行中，单个处理器内存储 1/P 份参数和 1/P 份输入. 3D 张量并行需要传输 q−1 份的输入张量和参数张量，以及 q−1 份的中间结果.

Megatron[24] 是由英伟达公司提出的一种基于 Transformer 结构的大模型训练框架，它使用了 1D 张量并行策略. 图 5 展示了 Megatron 的模型切分方案.



(a) 多层感知机

(b) 注意力层

Fig. 5 Illustration of 1D tensor parallelism in Megaton[24]

图 5 Megatron 的 1D 张量并行示意图[24]

单个 Transformer 块中包含一个多层感知机（MLP）和一个注意力层. 其中多层感知机分为 2 个线性的 GEMM 层和 1 个非线性的 *GeLU* 函数. 由于 *GeLU* 需要按照列为粒度进行计算，系统若对第 1 个 GEMM 层进行行切分，则需要将多个分片计算的结果同步相加之后才能将结果输入 *GeLU* 层进行计算，增加了通信的时间. 因此 Megatron 使用列切分的方式对第 1 个 GEMM 层进行切分. Megatron 使用行切分的方式对第 2 个 GEMM 进行切分，可以直接使用本地 *GeLU* 计算的结果作为输入，减少了等待所有 *GeLU* 分片结果并对其进行拼接的开销.

此外，Transformer 中的注意力层中包含了 3 个分别表示 $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ 的 GEMM，此外还包含 Softmax、*Dropout* 和输出前的 GEMM. 表示 $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ 的 GEMM 中包含了 1 个或者多个头（head）. Megatron 以头为粒度对 $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ 对应的 GEMM 按列进行切分，以保证模型计算的正确性. 对输出前的 GEMM 切分，Megatron 采用行切分的方式，以消除计算前输入的通信同步.

2.2 大模型训练访存感知的异构存储技术

若仅使用 GPU 显存存储模型数据，则需要更大规模的 GPU 集群以满足更大规模模型训练时的存储需求.

除 GPU 显存外，训练服务器中包含各种异构的存储资源，如 DRAM 和 SSD. 模型参数、优化器数据和训练产生的中间结果可以被存储到这些异构存储资源中，以提高模型训练的规模.

但是，异构存储系统的数据移动开销高. 虽然 GPU 内的存储资源的读写带宽高，以匹配其计算资源的性能，但与外部存储资源之间的互联带宽低. 这导致数据在 GPU 与外部设备间的搬移开销大. 如英伟达 GPU 配备有带宽高达数百 GB/s 的 DDR6 或 HBM. 但与外部的互联往往使用 PCIe，带宽仅有数十个 GBps，虽然有高速链路（如 NVLink）被提出，但仅支持高端的 GPU，硬件成本高昂，且扩展性有限. 如果在训练过程中仅简单地使用 GPU 外的存储空间，其带来的数据搬移开销大，限制了 GPU 内的存算资源利用率，影响模型训练的效率.

现有工作利用大模型训练中访存模式可预测的特性，以减少使用异构存储介质带来的数据移动开销. 在指定样本批大小等其他模型训练的超参数后，大模型训练中的每一个操作的计算开销和存储开销的变化相对固定. 并且大模型的模型结构固定，这使得每一个计算操作的顺序也随之固定. 因此，根据训练第一个样本批获得的每个操作的访存时刻和访存大小信息，可以预测后续模型训练过程中的访存模式[4, 25]. 表 3 对此类工作进行了总结. 本节将按照异构存储技术中使用的存储介质类型介绍此类工作.

2.2.1 基于 DRAM 的异构存储技术

相比于 GPU 显存，DRAM 的容量更大，因此，可以将模型数据或者训练的中间结果卸载到 DRAM 中，并在计算前按需将卸载数据上传到 GPU 显存之中，以支持更大规模的模型训练.

vDNN[26] 使用 DRAM 扩充显存容量以支持更大规模模型的训练. 在模型训练过程中，前向传播会产生大量的临时变量，这些临时变量需要被保存以用于反向传播过程中的梯度计算，其带来的显存开销的大小占整个训练过程中显存开销的 40%～80%[26]. 为解决此问题，vDNN 将前向传播计算得到的临时变量卸载到 DRAM 中，并在反向传播开始前将这些临时变量预取到显存中. vDNN 提出了异步的显存管理 API 以降低临时变量在显存中释放和分配的开销；使用 CUDA 的异步内存拷贝，以利用计算掩盖通信开销；提出了显存卸载策略以权衡显存开销和训练的性能.

Table 3 Comparison of Different Large Model Training System Using Heterogeneous Storage Resources

表 3 不同使用异构存储资源的大模型训练系统对比

系统	存储介质	并行策略	卸载模型参数	卸载优化器	卸载中间变量
vDNN[26]	显存+DRAM		否	否	是
SuperNeurons[27]	显存+DRAM		否	否	是（仅卸载卷积层）
SwapAdvisor[28]	显存+DRAM		是	否	是
Capuchin[29]	显存+DRAM		否	否	是
ZeRO-Offload[30]	显存+DRAM	ZeRO-DP	否	是	是
Harmony[31]	显存+DRAM	流水线	是	是	是
Mobius[32]	显存+DRAM	流水线	是	是	是
FlashNeuron[4]	显存+SSD		是	否	是
ZeRO-Infinity[33]	显存+DRAM+SSD	ZeRO-DP	是	是	是

1)异步显存管理

临时变量被卸载到 DRAM 后，显存空间需要被释放，以用于模型后续的计算；临时变量被上传到显存之前，系统需要预先分配一段显存空间. 而 CUDA 只提供同步的内存释放(cudaFree)与分配接口(cudaMalloc)，这会在模型训练过程中增加额外的同步开销. vDNN 则提出了异步的显存管理接口. vDNN 在系统启动后会预分配一段显存空间当作显存池. 临时变量需要分配显存空间时，vDNN 会直接从显存池中分配预先准备好的显存空间. 被释放的显存空间会被重新放回显存池中供之后的变量使用. 这避免了使用 CUDA 的同步显存分配和释放 API.

2)异步数据传输

为降低数据传输带来的额外开销，在显存卸载时，vDNN 首先会分配一段固定的 DRAM 空间，避免了数据传输经过回弹缓冲区(bounce buffer)，以最大化数据传输的带宽. 并且，vDNN 使用异步显存拷贝接口 cudaMemcpyAsync 将显存数据传输到 DRAM 上，以利用计算掩盖一部分数据通信所带来的开销. 在卸载数据重新上传回显存时，除利用异步显存拷贝接口外，vDNN 选取最近将被计算的层对应的临时变量进行预取，以最小化预取带来的显存开销的同时利用计算掩盖通信带来的开销.

3)显存卸载策略

为最大化降低显存开销，一种卸载策略是使用内存友好的卷积算法并且将所有临时变量卸载到 DRAM 中($vDNN_{all}$). 但内存友好的卷积算法可能会影响模型的收敛；并且卸载所有的临时变量会增加卸载和预取带来的通信开销，影响训练的性能. 为平衡显存开销和训练的性能，vDNN 提出了静态和动态的卸载策略.

在静态卸载策略中，vDNN 仅卸载计算开销最大的卷积层($vDNN_{conv}$)，以尽可能利用计算掩盖通信开销. 在动态卸载策略中，vDNN 首先测试显存开销最小的方案是否可以满足训练过程中的显存限制，即使用 $vDNN_{all}$ 和内存友好的卷积算法. 若显存开销最小的方案不满足训练过程中的显存限制，则说明此模型不能在给定的硬件条件下进行训练；若显存开销最小的方案满足显存限制，vDNN 再测试显存开销最大但训练性能最优的方案是否满足训练过程中的显存限制，即不对显存进行卸载并使用性能最优的卷积算法进行训练. 如果性能最优的方案满足显存需求，则使用此方案；否则，vDNN 测试 $vDNN_{conv}$ 在使用算法性能友好而非内存友好的卷积算法时是否满足显存限制. 若不满足，vDNN 使用贪心算法，将部分卷积层所使用的算法改为内存友好的算法. 当所有 $vDNN_{conv}$ 中的卷积层使用显存友好的算法后，训练过程仍不满足显存限制时，vDNN 开始使用 $vDNN_{all}$，并再次使用贪心算法，将部分卷积层所使用的算法改为内存友好的算法，以搜索符合显存限制的卸载策略.

SwapAdvisor[28] 认为 vDNN 这类工作基于人工经验选择卸载的数据，如 vDNN 仅卸载了卷积层. 这类工作一方面不具有通用性，另一方面不能达到最优的性能. SwapAdvisor 从张量的角度对 DNN 模型中的计算和存储进行建模，以支持模型训练中各种不同类型的模型数据的卸载；并采用遗传算法搜索卸载和预取的近似最优方案.

ZeRO-Offload[30] 是利用异构存储方案的分布式训练框架. ZeRO-Offload 采用与 ZeRO-DP 类似的训练流程，但是将模型数据和参数更新的过程卸载到 DRAM 和 CPU 中. 如图 6 所示，ZeRO-Offload 在 DRAM

中保存精度为 FP32 的参数与优化器，在显存中以 ZeRO-DP 的方式存储精度为 FP16 的参数. 在完成一次训练之后，精度为 FP16 的梯度信息会被传输到 DRAM 中，并在 CPU 中用作参数更新. 更新得到的参数将被转化成 FP16，并传输到显存中以更新旧版本的参数. 若 CPU 的参数更新的速度慢于 GPU 计算 1 步（step）的速度，ZeRO-Offload 将多步更新 1 次，以降低 CPU 参数更新带来的开销.

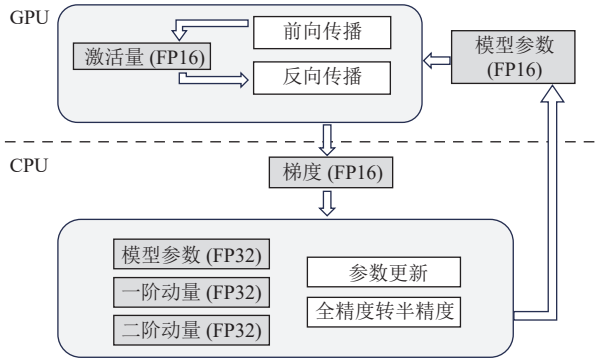


Fig. 6 Illustration of ZeRO-Offload[30]

图 6 ZeRO-Offload 示意图[30]

Mobius[32] 是清华大学于 2023 年提出的基于消费级 GPU 服务器的大模型训练系统. 如图 7 所示，不同于数据中心 GPU 服务器，消费级 GPU 服务器的通信资源有限. 一方面，消费级 GPU 服务器中的 GPU 不支持高带宽的 NVLink 链路，GPU 间的通信带宽受限于 PCIe 的链路带宽；另一方面，消费级 GPU 之间的数据通信不支持 GPU P2P，这导致 GPU 间的通信需要首先传输到 DRAM，再被传输到目标 GPU 中. 若共享 PCIe 链路的 GPU 同时通信，会出现通信竞争. 因此，在消费级 GPU 中使用类似于 ZeRO-Offload 此类 GPU 间通信频繁的系统，会带来巨大的通信开销，严重影响训练的性能.

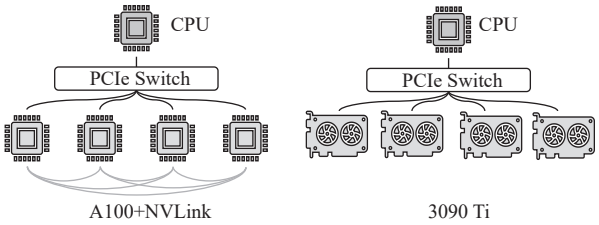


Fig. 7 Data center GPU server and commodity GPU server[32]

图 7 数据中心 GPU 服务器和消费级 GPU 服务器[32]

Mobius[32] 提出了基于流水线并行的异构存储训练策略. 具体地，该策略将模型分成多份子模型存储在 DRAM 中，按照模型训练的顺序依次将子模型上传到负责训练的 GPU 中，采用流水线并行的方式进行训练. 上传后的子模型在完成所有子样本的前向传播后将从显存中释放. 反向传播的过程与前向传播类似，得到的梯度数据会被传输到 DRAM 中用于参数更新. 为了降低数据传输带来的通信开销，Mobius 对上述训练过程进行建模，并使用混合线性规划的方式找出最优的子模型划分和预取方案. 为避免通信竞争，Mobius 尽可能地将预取时间近似的子模型映射到不共享 PCIe 链路的 GPU 中.

2.2.2 基于 SSD 的异构存储技术

随着模型规模的增长，模型数据量和中间结果到达了 TB 量级. 现有工作尝试将模型数据和中间变量卸载到 SSD 以支持更大规模的模型训练. 相较于 DRAM，SSD 的容量更大、价格与能耗更低，可以支持更大规模的模型训练，但 SSD 的读写带宽低，给模型训练带来了新的挑战.

FlashNeuron[4] 将模型参数卸载到 SSD 中. 为最小化卸载带来的开销，FlashNeuron 设计了离线的卸载选择策略，并提出了 GPU 直访 SSD 的数据传输方案.

在制定卸载策略时，FlashNeuron 首先将每个张量卸载到 SSD 中，以采样每一个张量在 GPU 中的显存开销、张量的计算开销、卸载到 SSD 的时间开销和对张量进行压缩的时间开销. 根据上述采样的结果，FlashNeuron 使用 2 阶段算法选择需要被卸载的张量. 在第 1 阶段中，FlashNeuron 将张量依次卸载到 SSD 中，直到显存空间满足训练需求为止，如果卸载的时间小于训练需要的时间，则使用此卸载方案. 否则，FlashNeuron 进入第 2 阶段，使用压缩张量比率最高的张量依次替换第 1 阶段得到的策略中不可被压缩的张量，以降低卸载过程中数据传输的开销. 为减少卸载与预取过程中 CPU 的开销，FlashNeuron 利用 GDRCopy 和 SPDK 实现了 GPU 对 SSD 的直接访问.

图 8 中颜色越深的张量压缩率越高，白色为不可压缩的张量，箭头长度表示张量卸载的时间开销，斜花纹的方块表示被卸载的张量. 若对图 8 所示模型训练，在不卸载任何张量时，模型训练超出 GPU 显存 16 MB. 在第 1 阶段中，FlashNeuron 按照模型顺序依次选择需要被卸载的张量，直到第 D 个张量被选择卸载之后，显存可满足训练所需的存储空间需求. 但此卸载策略下，模型训练计算的时间不能掩盖卸载带来的通信开销，因此进入第 2 阶段. FlashNeuron 使用压缩率更高的第 F 个张量和第 E 个张量替换了第 1 阶段中压缩不友好的第 B 个和第 D 个张量，此时卸载时间小于训练时间，因此此方案为最终卸载所使

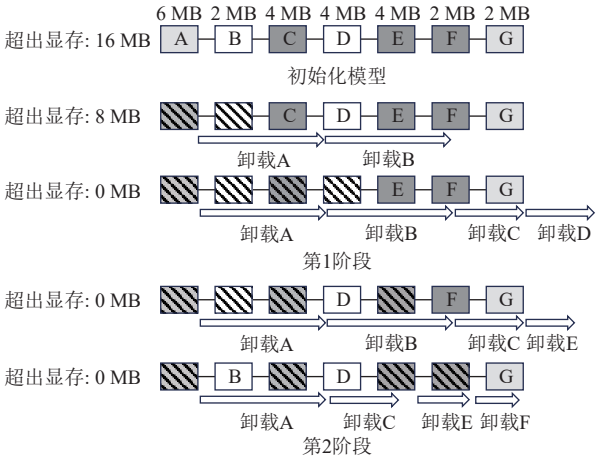


Fig. 8 Illustration of offloading strategy in FlashNeuron[4]

图 8 FlashNeuron 卸载策略示意图[4]

用的方案.

ZeRO-Infinity[33] 将模型数据存储在 SSD 中, 它沿用了 ZeRO 训练模式, 同时也使用了 ZeRO-Offload 中的 CPU 更新参数的策略. 在 ZeRO-Infinity 中, 为了解决 SSD 读写带宽较低的问题, 某一层的数据以图 9 中的方式被拆分成更小的分片存储到多个 SSD 中, 并由多张 GPU 同时负责传输, 以充分利用 SSD 的聚合带宽. 不同于传统的 ZeRO 中使用广播的方式, 由于单层的数据分散在多张 GPU 中, ZeRO-Infinity 需要使用全局聚合的方式让每一张 GPU 获得完整的用于训练的模型数据.

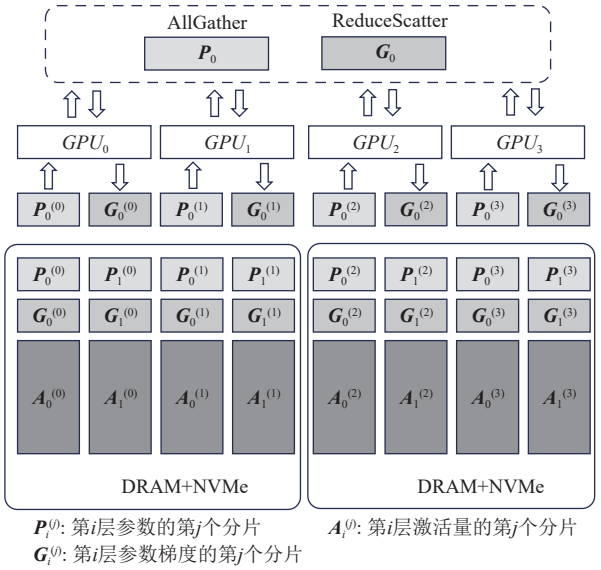


Fig. 9 Illustration of ZeRO-Infinity[33]

图 9 ZeRO-Infinity 示意图[33]

2.3 大模型数据缩减技术

传统的数据缩减技术往往通过压缩的方式减少存储开销[34-36]. 但是大模型的数据稠密[37] 难以被压缩. 现有工作根据大模型数据特征, 从 2 方面对大模型训练中的数据进行缩减: 增加计算量和牺牲模型精度. 具体地, 激活量检查点与重算(checkpoint and recomputation)算法通过增加计算量减少了所需存储的激活量数据; 混合精度与量化通过牺牲模型精度以降低模型数据和中间变量的大小. 本节将具体介绍这 2 种不同的大模型数据缩减技术.

2.3.1 激活量检查点与重算

在模型训练过程中, 前向传播产生的激活量会被保存在显存中, 并在反向传播过程中被用于计算梯度. 为减少保存激活量带来的存储开销, 有研究提出在前向传播时仅保存少量的激活量作为检查点, 并丢弃其他的激活量. 在需要使用被丢弃的激活量进行反向传播时, 系统利用激活量检查点重新进行前向传播, 以恢复之前所丢弃的激活量.

Chen 等人[38] 在 2016 年提出了激活量检查点与重算的方法, 并且提出了 3 种选择激活量检查点的算法. 第 1 种算法是直接选择计算开销小的操作, 如批归一化和池化操作, 这样减少显存开销的同时带来的额外计算开销较小. 第 2 种算法是使用贪心算法找到平衡额外计算开销和存储开销的最佳方案. 第 3 种算法是采用递归的方式, 通过不断扩大子模型的范围以找到最优的激活量检查点选择方案, 但这种算法的搜索空间大、成本高.

Checkmate[39] 是伯克利大学提出的复杂神经网络结构的激活量检查点选择算法. 随着残差神经网络的提出, 基于线性网络的激活量检查点选择方案不再适用. Checkmate 将模型中的数据依赖、存储开销、计算开销与重算开销进行建模, 使用线性规划的方式找出最优的激活量检查点的选择与重算的时刻. 由于整数线性规划是 NP 难问题, Checkmate 提出了一种 2 阶段的近似求解方案.

2.3.2 混合精度训练与量化

模型训练过程可以使用半精度(FP16)以降低显存开销. 但是 FP16 的有效范围和精度均比全精度小, 如果直接使用 FP16 会导致数据溢出和误差的问题. 一方面, 随着模型在训练过程中不断收敛, 梯度会随之变小. 此时, 梯度和学习率的乘积可能过小导致数据下溢. 另一方面, 由于 FP16 的间隔较小, 若梯度较小, 参数更新过程中会直接将结果舍入, 进而影响模型的收敛. 百度公司和英伟达公司在 2017 年提出了混合精度的训练方法[40], 其流程如图 10 所示. 该方法通过权重备份、损失量(loss)扩展和算数精度提高等方法解决了上述问题.

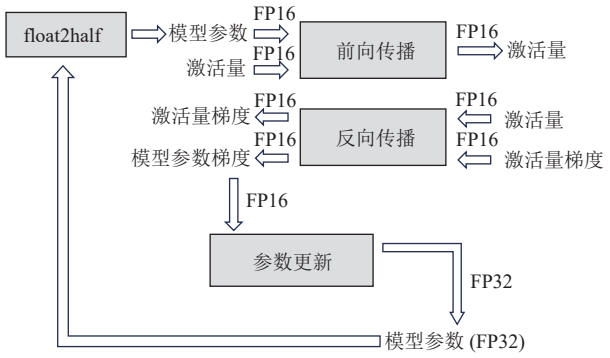


Fig. 10 Illustration of mixed precision training[40]

图 10 混合精度训练示意图[40]

1)权重备份

在存储模型数据时，模型参数、激活量和梯度均以半精度的格式存储. 除此之外，系统中还保存一份全精度格式的模型参数. 模型训练过程使用半精度的模型参数进行计算，并生成半精度的梯度数据. 在参数更新时，半精度的梯度数据与全精度版本的参数相加得到全精度的新模型参数. 新的全精度模型参数在下一轮训练前会先转化成半精度版本.

2)损失量扩展

为解决 FP16 向下溢出的问题，反向传播所产生的损失量会等比平移扩展到 FP16 的有效范围中，扩展的损失量会由反向传播作用于梯度之中. 在参数更新时，扩展的梯度先被反向平移到全精度范围中，再进行参数更新.

3)算数精度提高

某些网络结构可以通过使用全精度矩阵累加半精度矩阵乘法的结果，避免由 2 个半精度矩阵相加导致的舍入误差.

ActNN[41] 将前向传播过程中产生的激活量进行压缩以减少现存开销. ActNN 使用 2 比特压缩量化算法，这是因为 2 比特压缩量化算法在 GPU 上相较于其他压缩算法开销较小. ActNN 根据梯度信息，利用不同样本、不同维度和不同层之间的异构特性，对更重要的激活值分配更多比特，并且保证所有激活量中所有的浮点数平均大小为 2 b，以此在降低存储开销的同时减少压缩带来的误差. ActNN 提供 5 个优化等级，等级越高显存开销越小，但用于压缩的计算开销越大，运行速度越慢.

2.4 总结与比较

如表 4 所示的各类存储技术在单步训练中所需的显存开销、通信开销以及引入的额外计算开销各有差异.

Table 4 Comparison of Different Large Model Training Storage Technologies

表 4 各种不同大模型训练存储技术的比较

存储技术	显存开销	通信开销	额外计算开销
基于模型层间并行的分布式显存管理技术（以 ZeRO-DP 中 $\boldsymbol{P}_{os+g+p}$ 为例）	$P+O+A+T$	$3P(N-1)$	
基于模型层间并行的分布式显存管理技术（以 GPipe 为例）	$P+O+A+T$	$2a(N-1)$	
基于模型层内并行的分布式显存管理技术（以 1D 张量并行为例）	$P+O+A+T$	$2A(N-1)$	
基于 DRAM 的异构存储技术	$(1-\alpha)P+(1-\beta)O+(1-\gamma)A+T$	$3\alpha P+\beta O+2\gamma A$	
基于 SSD 的异构存储技术	$(1-\alpha)P+(1-\beta)O+(1-\gamma)A+T$	$3\alpha P+\beta O+2\gamma A$	
激活量检查点与重算技术	$P+O+\theta A+T$		重算开销
混合精度技术	$pP+qO+kA+T$		精度转化开销

注：N表示 GPU 数量；P，O，A，a，T分别表示模型参数、优化器参数、所有激活量、单层的激活量和其他临时变量；α，β，γ分别表示模型参数、优化器参数和所有激活量数据卸载至显存外的存储介质比例；θ表示激活量检查点的比例；p，q，k 分别表示模型参数、优化器参数和激活量使用混合精度后数据的压缩比例. 在训练过程中前向传播的激活量和反向传播的梯度激活量大小一致；反向传播后产生的梯度大小和模型参数大小一致.

基于大模型计算模式的分布式显存管理技术仅适用于 GPU 聚合显存容量，可满足大模型训练存储需求的场景. 其中基于模型层间并行的分布式显存管理技术中的 ZeRO-DP 训练模式需要在 GPU 中广播和聚合大量的模型梯度、优化器和参数；而流水线并行训练模式仅需要在 2 个 GPU 间点对点地传输子模型间的少量激活量. 因此，ZeRO-DP 训练模式更适用于高带宽、全互联的 GPU 集群，而流水线并行训练模式更适合于非全互联且卡间通信带宽低的 GPU 集群. 与 ZeRO-DP 类似，基于模型层内并行训练分布式显存管理技术需在 GPU 集群中使用集合通信聚合和分散激活量和参数，但对数据存储的粒度比 ZeRO-DP 更小，存储的可扩展性更高. 因此，它更适用于高带宽、全互联的 GPU 集群且模型中单层数据量庞大的场景.

大模型训练访存感知的异构存储技术使用成本更低的 DRAM 和 SSD 扩充高成本的 GPU 显存，以满足大模型训练过程中的存储需求. 相比于 SSD，DRAM 的读写带宽更高，单位容量的成本也更高. 因此基于

DRAM 的异构存储技术适用于对训练性能要求较高的场景, 而基于 SSD 的异构存储技术更适用于对成本控制更高的场景. 大模型数据缩减技术中的激活量检查点与重算技术通过增加计算开销以降低存储开销; 混合精度技术则通过牺牲模型训练的精度以减少模型数据大小. 此类技术可以与大模型训练访存感知的异构存储技术综合使用以降低数据在 GPU 间和异构存储介质间搬移的通信开销.

图 11 展示了根据算法性能需求和所用硬件条件选择最合适的存储技术的流程. 在实际训练过程中, 可能会混合采用多种不同的存储技术. 例如, 在 GPU 集群中, 通过高带宽的 NVLink 互联的 GPU 组使用基于层内并行的分布式显存管理技术; 通过带宽较低的 PCIe 互联的 GPU 组使用流水线并行的方式.

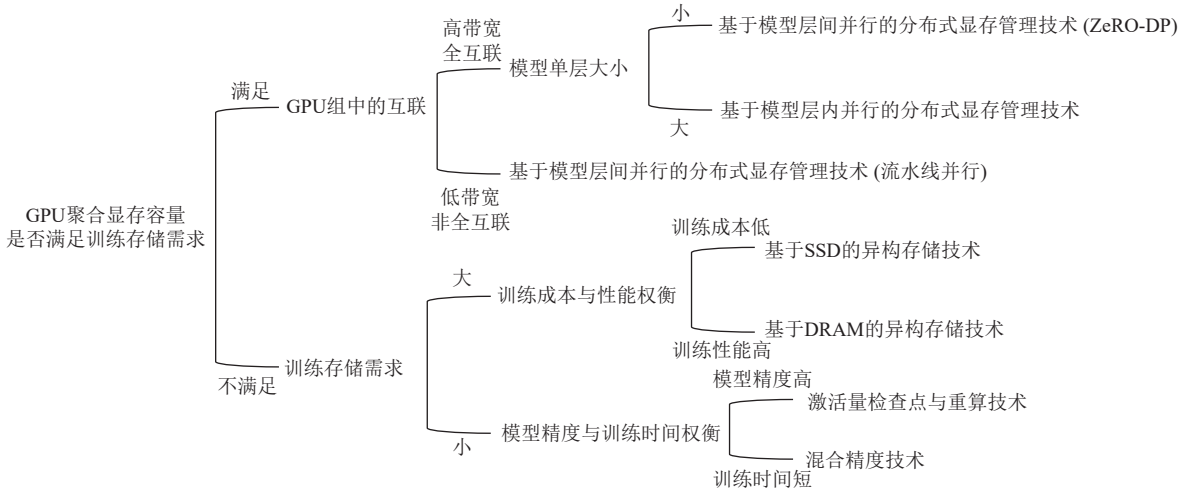


Fig. 11 Usage scenarios of various large model training storage technologies

图 11 各类大模型训练存储技术使用场景

3 面向大模型的存储容错技术

GPU 故障数量随着 GPU 集群规模的增大而提高. GPU 的频繁故障一方面会导致训练得到的参数丢失; 另一方面由于大模型训练中各 GPU 间的数据存在依赖关系, 单 GPU 的故障会扩散到整个 GPU 集群中. 有 2 类主要的工作解决大模型训练故障的问题: 参数检查点和冗余计算. 本节将具体介绍这 2 类不同的大模型训练容错技术.

3.1 参数检查点

参数检查点技术以设定的频率将训练得到的参数信息存储到持久化的存储介质中, 以对 GPU 故障进行容错. 在 GPU 故障后, 参数检查点技术利用最新且完整的参数进行恢复.

参数检查点技术存在 2 方面的挑战: 1)参数持久化的开销大, 会阻塞计算任务. 在参数完全写入到持久化介质前, 模型参数需要停止更新, 以避免参数丢失. 但是大模型参数量庞大, 且持久化存储介质的读写带宽远低于 GPU 显存的读写带宽与 GPU 的计算带宽, 这导致持久化模型参数开销大, 计算任务被长时间阻塞. 例如, 使用 64 台 EC2 的抢占式实例训练 GPT-2 模型, 77% 的训练时间将被用于参数的持久化操作[7]. 2)参数检查点的频率会影响训练与恢复的效率. 若参数检查点频率较高, 计算任务被参数持久化操作阻塞的频率也会随之提高, 影响模型训练的效率; 若参数检查点的频率较低, 故障时模型只能使用较早的参数版本进行恢复, 需要重新对大量数据集进行训练.

CheckFreq[42] 针对上述 2 方面挑战分别提出了 2 阶段参数检查点技术和基于采样的参数检查点策略.

2 阶段参数检查点技术将检查点的过程拆分为快照(snapshot)和持久化(persist)2 个阶段. 具体地, 在快照阶段, 若 GPU 显存中有空余空间, 系统将更新后的参数保存一份副本在显存中; 反之, 系统则将更新后的参数传输到 DRAM 中. 由于下一次参数更新后模型参数才会被修改, 快照可与下一次更新前的前向传播和反向传播并行进行. 在持久化阶段, 系统将快照的模型参数写入持久化介质中. 由于快照的模型参数数据与训练过程的模型参数数据是相互独立的, 持久化操作可以在后台进行, 不影响后续的计算任务. 如图 12 所示, 第 1 轮的参数更新后, 数据先快照到显存或者 DRAM 中. 快照过程可以与第 2 轮的前向传播与反向传播并行进行. 快照结束之后,

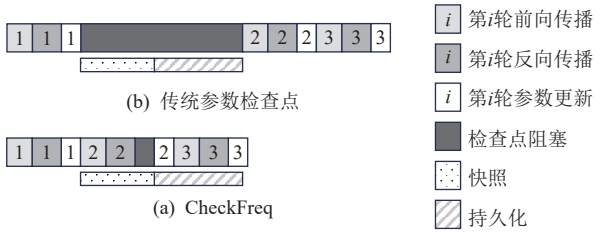


Fig. 12 Illustration of CheckFreq[42]

图 12 CheckFreq 示意图[42]

CheckFreq 再将快照的数据存储至持久化介质中.

ChekckFreq 依据训练过程中的系统采样数据，动态指定检查点的频率. CheckFreq 指出不同的模型规模和硬件性能会导致不同的检查点开销与恢复时间. 例如，模型规模越大，快照和持久化的数据量也会越多，这会增加参数检查点的开销. 因此，参数检查点的频率需要依据模型和硬件情况设定. CheckFreq 首先动态采样模型的训练时间、在显存与内存上快照的开销、参数持久化的大小以及持久化介质的带宽等信息，然后根据所采样的信息通过算法得出最优的检查点频率.

3.2 冗余计算

参数检查点技术需要大容量的持久化存储设备以保存检查点信息. 并且在恢复阶段，参数检查点需要从持久化介质中读取之前版本的参数，这导致恢复开销高. 有工作利用冗余计算的方式，在多张 GPU 中重复计算相同版本的参数，以对模型训练数据容错.

Bamboo[7] 在流水线并行训练模式中的每张 GPU 上冗余计算 2 个相邻的子模型，以达到容错的目的. 为了减少冗余计算带来的额外开销，Bamboo 利用流水线并行中天然存在的气泡执行前向传播任务的冗余计算，并采用了惰性执行反向传播的冗余计算的策略. 在模型训练中，反向传播的计算开销要远高于前向传播的计算开销，并且在 PipeDream 中越后执行前向传播的 GPU 计算开销越大，这会导致流水线中存在大量的气泡. 如图 13，假设 GPU_1 中前向传播的时间是 1 个时间单位，反向传播的时间是 2 个时间单位；GPU_2 中前向传播的时间是 1.2 个时间单位，反向传播的时间是 2.4 个时间单位. 在这 2 个 GPU 中使用流水线并行会导致 0.6 个时间单位的气泡. Bamboo

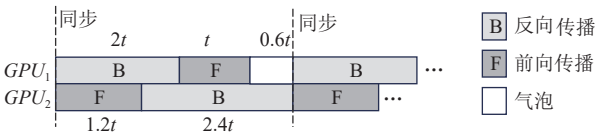


Fig. 13 Bubbles in pipeline parallelism[7]

图 13 流水线并行的气泡[7]

利用这些气泡对相邻的子模型进行冗余的前向传播. 对于计算开销较大的反向传播，Bamboo 仅在出错后利用冗余前向传播得到的结果进行反向传播，以减少冗余计算对训练带来的影响.

在某 GPU 故障后，其对应的计算任务会被路由到负责冗余计算的 GPU 上. 若连续多个 GPU 故障，导致某子模型数据从 GPU 显存中丢失，则需要从参数检查点中重新加载之前版本的模型参数.

4 总结与展望

本文详细阐述了 ChatGPT 时代下大模型训练过程中存在的存储挑战，并以此出发介绍了现有的大模型训练中所使用的存储技术. 如今有大量大模型训练系统支持了本文所介绍的存储技术[43-46]，并提供了简洁高效的调用接口，降低了用户进行大模型训练的门槛. 但随着模型规模的急剧增加和硬件设备的发展，大模型训练中仍存在着一些存储问题亟需解决：

1）大模型训练的存储成本. 一方面，为充分利用硬件的计算资源，训练需要使用读写带宽更高的存储介质，如 HBM. 而这些存储介质的单位比特的价格高昂. 另一方面，随着模型规模的不断攀升，训练所需的存储空间也随之增长，因此需要更大容量的存储介质，这增加了训练过程中用于存储设备的成本.

2）大模型训练的绿色存储. 大模型训练能耗大，导致大量碳排放，从而影响全球环境. 从存储角度讲，大模型训练需要使用数量庞大的高带宽存储介质满足大模型训练中的存储需求，这些存储介质相较于读写带宽较低的 SSD 和 HDD 能耗高. 并且大模型训练中数据读写频繁，进一步提高了存储能耗.

参 考 文 献

[1] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need [C/OL] //Proc of the 30th Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2017 [2023-05-30]. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[2] Devlin J, Chang M, Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding [J]. arXiv preprint, arXiv: 1810.04805, 2018

[3] OpenAI. GPT-4 technical report [J]. arXiv preprint, arXiv: 2303.08774, 2023

[4] Bae J, Lee J, Jin Y, et al. FlashNeuron: SSD-enabled large-batch training of very deep neural networks [C] //Proc of the 19th USENIX

Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2021: 387-401

[5] Ruder S. An overview of gradient descent optimization algorithms[J]. arXiv preprint, arXiv: 1609.04747, 2016

[6] Kingma D, Ba J. Adam: A method for stochastic optimization[J]. arXiv preprint, arXiv: 1412.6980, 2014

[7] Thorpe J, Zhao Pengzhan, Eyolfson J, et al. Bamboo: Making preemptible instances resilient for affordable training of large DNNs[C]//Proc of the 20th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2023: 497-513

[8] Zhang Susan, Roller S, Goyal N, et al. OPT: Open pre-trained transformer language models[J]. arXiv preprint, arXiv: 2205.01068, 2022

[9] Jeon M, Venkataraman S, Phanishayee A, et al. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads[C]// Proc of USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2019: 947-960

[10] Shvachko K, Kuang H, Radia S, et al. The Hadoop distributed file system[C/OL]//Proc of the 26th IEEE Symp on Mass Storage Systems and Technologies. Piscataway, NJ: IEEE, 2010[2023-05-21].https://www.computer.org/csdl/proceedings-article/msst/2010/05496972/12OmNwxlrhU

[11] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[C/OL]//Proc of the 2nd USENIX Workshop on Hot Topics in Cloud Computing. Berkeley, CA: USENIX Association, 2010[2023-05-21].https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf

[12] Weil S, Brandt S, Miller E, et al. Ceph: A scalable, high-performance distributed file system[C]//Proc of the 7th Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2006: 307−320

[13] Rajbhandari S, Rasley J, Ruwase O, et al. ZeRO: Memory optimizations toward training trillion parameter models[C/OL]//Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020[2023-05-21].https://dl.acm.org/doi/pdf/10.5555/3433701.3433727

[14] Huang Yanping, Cheng Youlong, Bapna A, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism[C/OL]// Proc of the 33rd Conf on Neural Information Processing Systems. Cambridge, MA: MIT, 2019[2023-05-30].https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf

[15] Narayanan D, Harlap A, Phanishayee A, et al. PipeDream: Generalized pipeline parallelism for DNN training[C/OL]//Proc of the 27th ACM Symp on Operating Systems Principles. New York: ACM, 2019[2023-05-20].https://dl.acm.org/doi/pdf/10.1145/3341301.3359646

[16] Jain A, Awan A, Aljuhani A, et al. GEMS: GPU-enabled memory-aware model-parallelism system for distributed DNN training [C/OL]// Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020[2023-05-21].https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9355254

[17] Narayanan D, Phanishayee A, Shi Kaiyu, et al. Memory-efficient pipeline-parallel DNN training[C]//Proc of the 38th Int Conf on Machine Learning Research. New York: PMLR, 2021: 7937−7947

[18] Fan Shiqing, Rong Yi, Meng Chen, et al. DAPPLE: A pipelined data parallel approach for training large models[C]//Proc of the 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2021: 431−445

[19] Li Shigang, Hoefler T. Chimera: Efficiently training large-scale neural networks with bidirectional pipelines[C/OL]// Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2020[2023-05-21]. https://dl.acm.org/doi/abs/10.1145/3458817.3476145

[20] Shoeybi M, Patwary M, Puri R, et al. Megatron-LM: Training multi-billion parameter language models using model parallelism[J]. arXiv preprint, arXiv: 1909.08053, 2019

[21] Xu Qifan, You Yang. An efficient 2D method for training super-large deep learning models[C]//Proc of Int Symp on Parallel and Distributed Processing. Piscataway, NJ: IEEE, 2021: 222−232

[22] Wang Boxiang, Xu Qifan, Bian Zhengda, et al. Tesseract: Parallelize the tensor parallelism efficiently[C/OL]//Proc of the 51st Int Conf on Parallel Processing. New York: ACM, 2022[2023-05-21]. https://dl.acm.org/doi/abs/10.1145/3545008.3545087

[23] Bian Zhengda, Xu Qifan, Wang Boxiang, et al. Maximizing parallelism in distributed training for huge neural networks[J]. arXiv preprint, arXiv: 2105.14450, 2021

[24] Narayanan D, Shoeybi M, Casper J, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM[C/OL]// Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2021[2023-05-21].https://dl.acm.org/doi/abs/10.1145/3458817.3476209

[25] Fang Jiarui, Zhu Zilin, Li Shenggui, et al. Parallel training of pre-trained models via chunk-based dynamic memory management[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 34(1): 304−315

[26] Rhu M, Gimelshein N, Clemons J, et al. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design[C/OL]//Proc of the 49th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2016[2023-05-21]. https://ieeexplore.ieee.org/abstract/document/7783721

[27] Wang Linna, Ye Jinmian, Zhao Yiyang, et al. SuperNeurons: Dynamic GPU memory management for training deep neural networks[C]// Proc of the 23rd ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2018: 41−53

[28] Huang C, Jin Gu, Li Jinyang. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping[C]//Proc of the 25th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 1341−1355

[29] Peng Xuan, Shi Xuanhua, Dai Hulin, et al. Capuchin: Tensor-based GPU memory management for deep learning[C]//Proc of the 25th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 891−905

[30] Ren Jie, Rajbhandari S, Aminabadi R Y, et al. ZeRO-Offload:

Democratizing billion-scale model training[C]//Proc of USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2021: 551−564

[31] Li Youjie, Phanishayee A, Murray D, et al. Harmony: Overcoming the hurdles of GPU memory capacity to train massive DNN models on commodity servers[J]. arXiv preprint, arXiv: 2202.01306, 2022

[32] Feng Yangyang, Xie Minhui, Tian Zijie, et al. Mobius: Fine tuning large-scale models on commodity GPU servers[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2023: 489−501

[33] Rajbhandari S, Ruwase O, Rasley J, et al. ZeRO-Infinity: Breaking the GPU memory wall for extreme scale deep learning[C/OL]// Proc of the Int Conf for High Performance Computing, Networking, Storage and Analysis. Piscataway, NJ: IEEE, 2021[2023-05-21].https://dl.acm.org/doi/abs/10.1145/3458817.3476205

[34] Buluç A, Fineman J T, Frigo M, et al. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks[C]//Proc of the 21st Annual Symp on Parallelism in Algorithms and Architectures. New York: ACM, 2009: 233−244

[35] Scipy. scipy. sparse. coo_matrix[EB/OL]. 2023 [2023-06-29].https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.coo_matrix.html

[36] Scipy. scipy. sparse. lil_matrix[EB/OL]. 2023 [2023-06-29].https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html

[37] Fedus W, Zoph B, Shazeer N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity[J]. The Journal of Machine Learning Research, 2022, 23(1): 5232−5270

[38] Chen Tianqi, Xu Bing, Zhang Chiyuan, et al. Training deep nets with sublinear memory cost[J]. arXiv preprint, arXiv: 1604.06174, 2016

[39] Jain P, Jain A, Nrusimha A, et al. Checkmate: Breaking the memory wall with optimal tensor rematerialization[C/OL]// Proc of the 3rd Machine Learning and Systems. 2020[2023-05-20].https://proceedings.mlsys.org/paper_files/paper/2020/file/0b816ae8f06f8dd3543dc3d9ef196cab-Paper.pdf

[40] Micikevicius P, Narang S, Alben J, et al. Mixed precision training[J]. arXiv preprint, arXiv: 1710.03740, 2017

[41] Chen Jianfei, Zheng Lianmin, Yao Zhewei, et al. ActNN: Reducing training memory footprint via 2-bit activation compressed training[C]//Proc of the 38th Int Conf on Machine Learning Research. New York: PMLR, 2021: 1803−1813

[42] Mohan J, Phanishayee A, Chidambaram V. CheckFreq: Frequent, fine-frained DNN checkpointing[C]//Proc of the 19th USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2021: 203−216

[43] NVIDIA. Megatron-LM [EB/OL]. 2023 [2023-06-01].https://github.com/NVIDIA/Megatron-LM

[44] Microsoft. DeepSpeed [EB/OL]. 2023 [2023-06-10].https://github.com/microsoft/DeepSpeed

[45] HPCAITech. Colossal-AI [EB/OL]. 2023 [2023-06-10].https://github.com/hpcaitech/ColossalAI

[46] OneFlowInc.OneFlow[EB/OL].2023[2023-06-10].https://github.com/Oneflow-Inc/oneflow



Feng Yangyang, born in 1998. PhD candidate. His main research interests include storage systems and machine learning systems.

冯杨洋, 1998 年生. 博士研究生. 主要研究方向为存储系统、机器学习系统.



Wang Qing, born in 1997. PhD. His main research interests include storage systems and memory systems.

汪　庆, 1997 年生. 博士. 主要研究方向为存储系统、内存系统.



Xie Minhui, born in 1997. PhD candidate. Student member of CCF. His main research interests include storage systems and machine learning systems.

谢旻晖, 1997 年生. 博士研究生. CCF 学生会员. 主要研究方向为存储系统、机器学习系统.



Shu Jiwu, born in 1968. PhD, professor, PhD supervisor. Fellow of CCF. His main research interests include intelligent storage systems, non-volatile memory storage systems and technologies, storage security and reliability, and parallel and distributed computing.

舒继武, 1968 年生. 博士, 教授, 博士生导师. CCF 会士. 主要研究方向为智能存储系统、非易失内存存储系统与技术、存储安全与可靠性、并行与分布式计算.