

大模型训练过程

1. 预训练

- 目标：学会语言本身，语言的通用表示，进行语言建模
- 方法：自监督学习

2. SFT（有监督微调）：何使用标注数据对预训练模型进行监督训练的过程

- 目标：让模型遵循指令，能够理解人类提出的问题
- 方法：监督学习
- 通常有全参微调，LoRA微调，Adapter微调等，指令微调也是SFT的一部分

3. RLHF（人类反馈强化学习）

- 目标：让模型更符合人类的偏好，贴近人类意图
- 方法：强化学习

4. 增强与扩展

- 蒸馏，微调，迁移学习等

SFT (Supervised Finetune)

主要任务是数据收集与标注，标注数据的质量和数量对微调效果至关重要

数据生产工作不完全是dirty work，数据质量直接决定模型微调后性能的好坏

数据方面有以下几点要阐述的

- Few-Shot Prompting 是一种通过提供少量示例（通常1-5个）来引导模型生成符合任务要求的输出的技术
- Seed Prompt 是为特定任务类型（`task_type`）预先设计的指令模板，用于明确任务目标、输入输出格式及上下文约束
- 数据多样性
 - 数据质量和多样性比数据数量更为重要
 - answer是尽量不要出错，需要大量人工筛查
 - `task_type` 的划分就是sft数据最重要的基建工作，没有之一
- 数据生产
 - 生产prompt
 - 给每个`task_type`（任务类型分类）准备一些seed prompt，然后随机采样seed，喂给一个很强的pretrain模型，让他基于这些seed再续写出一些问题或者prompt
 - 生产answer
 - 不在乎成本，用GPT4/Claude3
 - 在乎成本，Qwen_72B/deepseek_MoE

引人深思的一段话

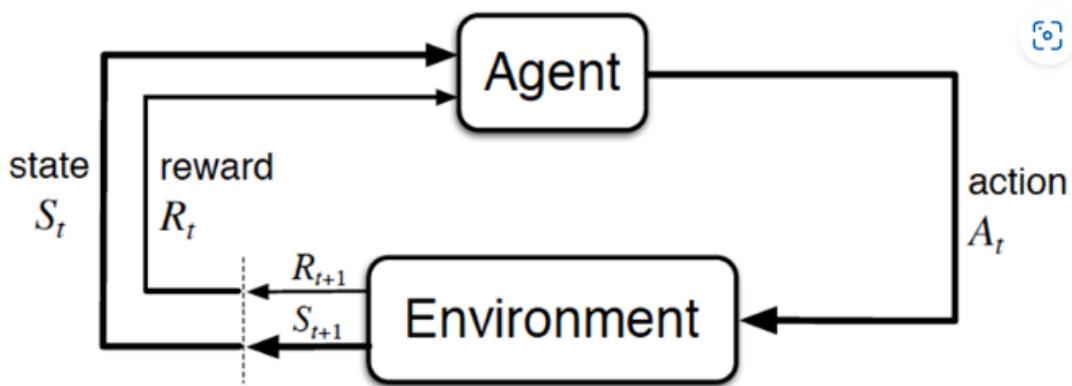
模型的上线并不代表着sft工作的结束，它反倒代表着sft真正工作的开始。只有到了这一刻，我们才开始接触“最真实的用户prompt”。

RL (Reinforcement Learning)

定义：基于智能体在复杂、不确定的环境中最大化他能获得的奖励，从而获得自主决策

核心目标：给定一个马尔可夫决策过程，寻找最优策略

经典的强化学习模型



- **Agent:** 智能体，就是我们要训练的模型
- **action:** 行为
- **Environment:** 环境，是提供reward的某个对象
- **reward:** 奖赏，可以类比为在明确目标的情况下，接近目标意味着奖，远离目标意味着做的不好则惩，最终达到奖励最大化
- **State:** 环境的状态

马尔可夫决策过程 (MDP)

状态转移矩阵

假设有一类不确定的现象，假设今天是晴天，但无法百分百确定明天一定是晴天还是雨天、阴天

对于这种假设具有M个状态的模型

1. 共有 M^2 的状态转移，因为任何一个状态都有可能是所有状态的下一个转移状态
2. 每一个状态转移都有一个概率值，相当于从一个状态转移到另一个状态的概率
3. 所有 M^2 个概率可以用一个状态转移矩阵表示

		Today		
		sun	cloud	rain
Yesterday	sun	0.50	0.375	0.125
	cloud	0.25	0.125	0.625
	rain	0.25	0.375	0.375

概率论的研究对象是静态的随机现象，而随机过程的研究对象是随时间演变的随机现象

- 随机现象在某时刻t的取值是一个向量随机变量，用 S_t 表示 比如上述天气转移矩阵便如下所示

$$[S_1 \rightarrow S_1 \quad S_1 \rightarrow S_2 \quad S_1 \rightarrow S_3 \quad S_2 \rightarrow S_1 \quad S_2 \rightarrow S_2 \quad S_2 \rightarrow S_3 \quad S_3 \rightarrow S_1 \quad S_3 \rightarrow S_2 \quad S_3 \rightarrow S_3]$$

- 在某个时刻t的状态 S_t 通常取决于t时刻之前的状态，将已知历史信息 (S_1, \dots, S_t) 时下一个时刻的状态 S_{t+1} 的概率表示成 $p(S_{t+1}|S_1, \dots, S_t)$
- 当且仅当某时刻的状态只取决于上一时刻的状态时，一个随机过程被称为具有**马尔可夫性质**

$$p(S_{\{t+1\}}|S_{\{t\}}) = p(S_{\{t+1\}}|S_1, \dots, S_t)$$

- 具有马尔可夫性质的随机过程称为**马尔可夫过程**

- 是一个二元组 $\langle S, P \rangle$, S 是有限状态集, P 是状态转移矩阵

$$P_{\{ss'\}} = p(S_{\{t+1\}}=s'|S_{\{t\}}=s)$$

马尔可夫奖励过程 (MRP)

是一个四元组 $\langle S, P, R, \gamma \rangle$

- S : 有限状态集
- P : 状态转移概率矩阵
- R : 奖励函数 $R_S = E[R_{t+1}|S_t = s]$
 - 也就是**状态转移概率加权和**
 - 表示从状态 s 到状态 s' 的收益
 - 取均值是因为状态 s' 并非固定，从 s 到下一个状态有多种可能，要取这多种可能的均值
- γ : 折扣因子/衰减系数 $\gamma \in [0, 1]$
- 回报 (Return) : G_t 是从时间t开始的总折扣奖励
 - 表示所有**奖励在当前的价值**，量化一个策略的长期表现
 - 设置衰减系数的原因：保证回报的收敛
- 值函数： $V(s)$ 表示一个状态 s 的长期价值
 - 是对回报的期望
 - 状态 s 后面的状态都是未知的，也就是有多个状态路径可以选择，**值函数就是对这多个状态路径的回报取了均值**

$$V(s) = E[G_t|S_t=s]$$

MRPs的贝尔曼方程

由下述式子可以推出 $V(s) = E[G_t|S_t=s]$ $G_t = R_{\{t+1\}} + \underbrace{\gamma R_{\{t+2\}} + \gamma^2 R_{\{t+3\}} + \dots}_{\gamma V(S_{\{t+1\}})} = \sum_{k=0}^{\infty} \gamma^k R_{\{t+k+1\}}$ 也就是 $V(s) = E[R_{\{t+1\}} + \gamma V(S_{\{t+1\}})|S_t=s]$

马尔可夫决策过程 (MDP)

马尔可夫决策过程是一个五元组 $\langle S, A, P, R, \gamma \rangle$

- S : 有限状态集
- A : 动作集
- P : 状态转移概率矩阵 $P_{ss'}^a = p[S_{t+1} = s' | S_t = s, A_t = a]$
- R : 奖励函数 $R_S^a = E[R_{t+1} | S_t = s, A_t = a]$
 - 也就是状态转移概率加权和
 - 表示从状态 s 执行动作 a 到状态 s' 的收益
 - 取均值是因为状态 s' 并非固定，从 s 到下一个状态有多种可能，要取这多种可能的均值
- γ : 折扣因子/衰减系数 $\gamma \in [0, 1]$

策略

π 是给定状态的动作分布， $\pi(a|s) = P[A_t = a | S_t = s]$

- 是一个随机变量，表示在状态 s 前提下作出动作 a 的可能性
- 完全决定智能体行为
- MDP 策略依赖于当前状态（无关历史），且无关时间

给定一个马尔可夫决策过程 $M = \langle S, A, P, R, \gamma \rangle$ 和策略 π ，其可转化为马尔可夫过程和马尔可夫奖励过程

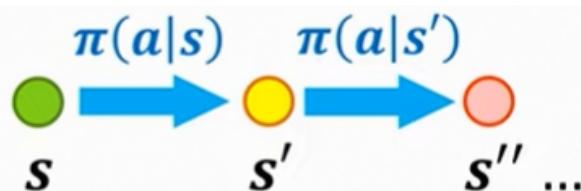
- 也就是通过加权求和把动作的维度消除，减少运算量

$$P_{\{s,s'\}}^{\{\pi\}} = \sum_{a \in A} \pi(a|s) P_{\{s,s'\}}^a \quad R_s^{\{\pi\}} = \sum_{a \in A} \pi(a|s) R_s^a$$

- 第一个可以理解成从状态 s 可以使用多个动作到状态 s' ，这里对其使用了加权求和
- 第二个也同理，加权求和，消除动作的维度

状态值函数

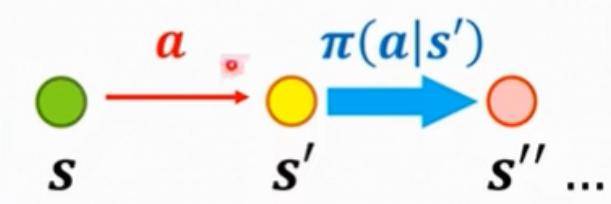
表示在状态 s 下，遵循策略 π 的期望回报



$V_{\{\pi\}}(s) = E_{\{\pi\}}[G_t | S_t = s]$ 回报的表达式 $G_t = \underbrace{R_{\{t+1\}}}_{\text{立即奖励}} + \underbrace{\gamma R_{\{t+2\}} + \gamma^2 R_{\{t+3\}} + \dots}_{\text{后继状态的折扣价值}} = \sum_{k=0}^{\infty} \gamma^k R_{\{t+k+1\}}$ 可以得到状态值函数的贝尔曼期望方程 $V_{\{\pi\}}(s) = E_{\{\pi\}}[R_{\{t+1\}} + \gamma V_{\{\pi\}}(S_{\{t+1\}}) | S_t = s]$

动作值函数

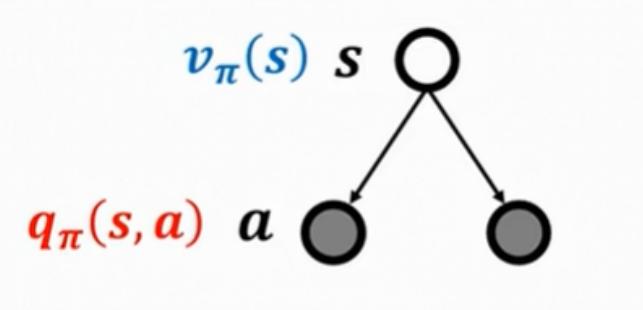
表示在状态 s 下执行动作 a 后，遵照策略 π 的期望回报



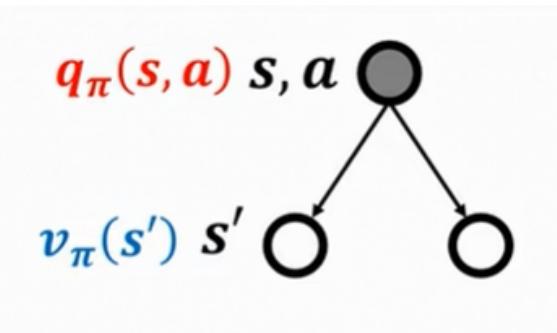
$q_{\{\pi\}}(s, a) = E_{\{\pi\}}[G_t | S_t = s, A_t = a]$ 同上推导，可以得到动作值函数的贝尔曼期望方程 $q_{\{\pi\}}(s, a) = E_{\{\pi\}}[R_{t+1} + \gamma q_{\{\pi\}}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$

将动作值函数和状态值函数联系起来

某一个状态的价值可以用该状态下的所有动作的价值表述 $V_{\{\pi\}}(s) = \sum_{a \in A} \pi(a|s) q_{\{\pi\}}(s, a)$



某一个动作的价值可以用该状态后继状态的价值表述 $q_{\{\pi\}}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\{\pi\}}(s')$



贝尔曼最优方程

最优状态值函数 $V_{\{\pi\}}(s) = \max_{\{\pi\}} V_{\{\pi\}}(s)$ 最优动作值函数 $q_{\{\pi\}}(s, a) = \max_{\{\pi\}} q_{\{\pi\}}(s, a)$ 最优策略

- 存在一个最优策略，使 $\pi_* \geq \text{any } \pi$
- 所有最优策略都能取得最优状态值函数
- 所有最优策略都能取得最优动作值函数
- 最优策略本质上是一个具体的动作链，而不是分布

π_* 表示任何策略

注：若 $V_{\pi'}(s) \geq V_{\pi}(s)$ ，则 $\pi' > \pi$

由上述定义可以写出贝尔曼最优方程

状态值函数的贝尔曼最优方程

首先根据 $V_{\{\pi\}}(s) = \sum_{a \in A} \pi(a|s) q_{\{\pi\}}(s, a)$ 和 $q_{\{\pi\}}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\{\pi\}}(s')$ 写出： $V_{\{\pi\}}(s) = \sum_{a \in A} \pi(a|s) (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\{\pi\}}(s'))$

(s')) 下面来推导贝尔曼最优方程 (动态规划方法)

从状态值函数的贝尔曼方程出发 $V_{\pi}(s) = \sum_{a \in A} \pi(a|s) q_{\pi}(s, a)$

最优策略 π_* 必须在每个状态 s 选择最大化 $q_*(s, a)$

- 最优策略 π_* 一定满足 $V_{\pi_*}(s) \geq V_{\pi}(s)$
- 也满足 $q_{\pi_*}(s, a) \geq q_{\pi}(s, a)$
- 最大化 $V_{\pi}(s)$ 等价于最大化 $q_{\pi}(s, a)$

$\pi(a | s) = \begin{cases} 1 & \text{if } a = \arg \max_a q_{\pi}(s, a), \\ 0 & \text{otherwise.} \end{cases}$ 因此，最优状态值需要满足： $V_*(s) = \max_a q_*(s, a)$ 代入动作值函数 q_*

$(s, a) = R_s^a + \gamma \sum_{s'} P_{ss'}^a V_*(s')$ 最后得出状态值函数的贝尔曼最优方程 $V_*(s) = \max_a (R_s^a + \gamma \sum_{s'} P_{ss'}^a V_*(s'))$

动作值函数的贝尔曼最优方程

同上，写出： $q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi}(s')$

要最大化动作值函数的贝尔曼方程，就是要最大化 V_{π} ，即 $q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi}(s')$ 代入 $V_*(s')$ ，得到动作值函数的贝尔曼最优方程 $q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_a q_{\pi}(s', a)$

LLaMA

与GPT类似，LLaMA也只使用了Transformer的解码器，即Decoder-only架构，且目前主流的语言大模型都采用了这个架构

- 只训练了单一模块，效率高
- 采用掩码注意力实现双向编码和单向生成的兼容
- 自回归任务天生适应

RMSNorm

通过计算输入的张量的均方根实现归一化

公式如下 $\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon}} \cdot \gamma$ 其中：

- x 是输入向量
- d 是输入向量的维度
- ϵ 是一个小常数，避免除零错误
- γ 是一个可学习的缩放参数

SwiGLU

是一种用于神经网络的激活函数，结合了Swish激活函数和门控机制的特点

Swish激活函数 $\text{Swish}(x) = x \cdot \sigma(x)$

- σ 为 Sigmoid 函数

GLU激活函数 $\text{GLU}(x) = \sigma(W_1 x + b_1) \odot (W_2 x + b_2)$

- \odot 为逐元素相乘
- W_1, W_2 为权重矩阵, b_1, b_2 为偏置项

SwiGLU激活函数 $\text{SwiGLU}(x) = \text{Swish}(\text{Linear_1}(x)) \odot \text{Linear_2}(x)$ 符号寓意同上

优势:

- 平滑非线性: Swish函数的平滑性可以缓解梯度消失
- 门控特性: GLU的门控机制允许模型动态调整信息流

RoPE

提过, 不说了, 在[其它笔记](#)里有

GQA

在LLaMA2、3中使用

具体理论在下面

GPT

Deepseek

MLA (多头潜在注意力)

下面说了, 不再赘述

分布式训练

为何要进行分布式训练 ?

- 模型太过庞大, 一个GPU放不下
- 用多张GPU加速模型训练

有哪些分布式训练框架 ?

- DP (Data Parallel)
- DDP (Distributed Data Parallel)
- FSDP (Fully Sharded Data Parallel)

All-Reduce (全局制约)

- 目标: 所有进程 (GPU) 都得到相同结果, 该结果是所有进程输入数据的聚合
- 操作流程
 - 输入: 每个GPU有一个本地数据
 - 数据交换: 对所有GPU的数据执行某种操作 (如 SUM、MAX)
 - 输出: 所有GPU得到完全相同的聚合结果

All-Gather (全局收集)

- 目标：所有进程（GPU）收集其它所有进程的数据，最终每个GPU拥有完整的数据拼接（不聚合）

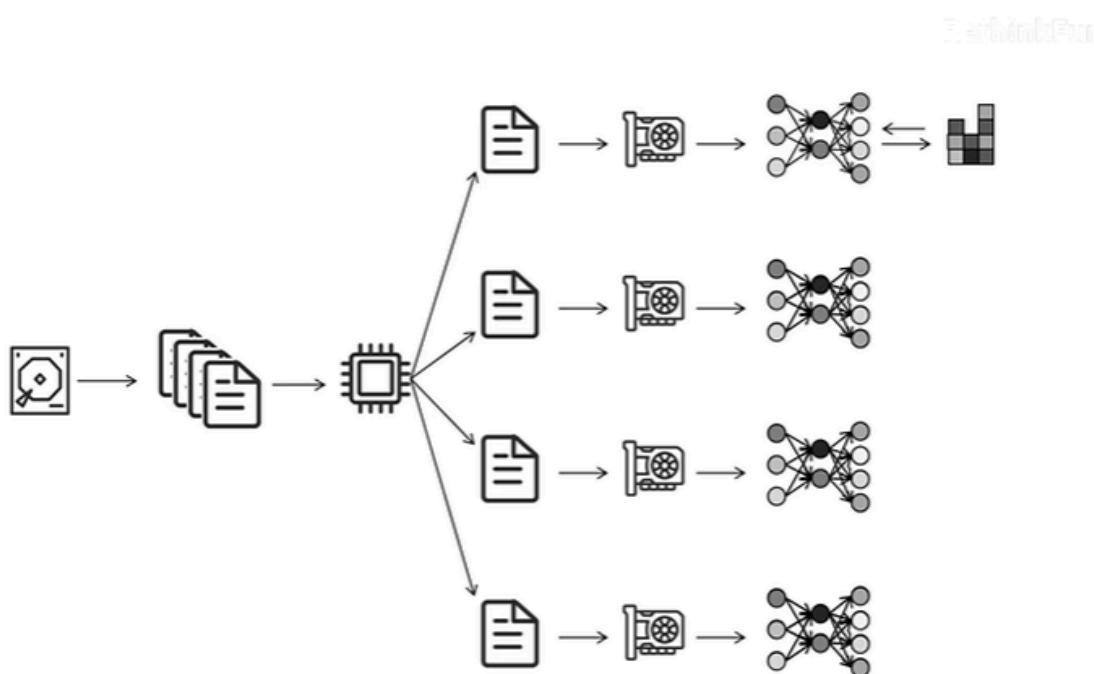
- 操作流程

- 输入：每个GPU有一个本地数据块
- 数据交换：所有GPU互相广播自己的数据块
- 输出：所有GPU得到所有数据块的完整拼接

数据并行

DP (Data Parallel)

单进程多线程，Python GIL（全局解释器锁）只能利用一个CPU核



工作流程

1. 用CPU将数据分成多份，给每个GPU一份
2. 每个GPU独自进行训练，将自己计算的梯度传递到GPU0上
3. GPU0用全局平均梯度更新自己的网络参数
4. 将更新后的参数广播到其它GPU上

通信分析

假设参数量位 ψ ，节点数为 N

- 对于GPU0，传入梯度为 $(N - 1)\psi$ ，传出参数为 $(N - 1)\psi$
- 对于其它GPU，传出梯度为 ψ ，传入参数为 ψ

问题

- 单进程，多线程，Python GIL只能利用一个CPU核

- GPU0负责收集梯度，更新参数，广播参数，计算压力大

DDP (Distributed Data Parallel)

生产环境常用，多进程多线程

- 执行多个相同的py脚本，但用rank进行不同脚本的区分，也就实现了多进程
- 注意的是通讯成本比计算成本大

工作步骤

- GPU0加载模型，并把模型同步到其它GPU
- 按照神经网络参数定义反序，把参数进行排列，即输出层在最前面，输入层在最后面
- 对每个参数注册一个监听器，将这些监听器按顺序放到一个个桶里
- GPU在进行计算的同时进行传输，先计算出来的梯度先进行同步
- 当多个GPU的同一个桶的梯度都计算完成后，就进行Ring-AllReduce的梯度同步
- 所有的桶都计算完毕后，每个GPU调用他们各自的优化器进行参数更新

通讯分析

假设参数量位 ψ ，进程数为 N

对于每个GPU进程

- Scatter-Reduce阶段传入/传出： $(N - 1)\frac{\psi}{N} \approx \psi$
- All-Gather阶段传入/传出： $(N - 1)\frac{\psi}{N} \approx \psi$

总传入/传出： 2ψ ，与集群大小无关

集群通信方式：Ring-AllReduce

- 将多个节点连成环进行通讯
- 具体就是每个GPU即在发送，也在接受数据
- 参数量除以总GPU个数就是一个GPU负责的块（也就是要同步的块）

工作流程：

Scatter-Reduce

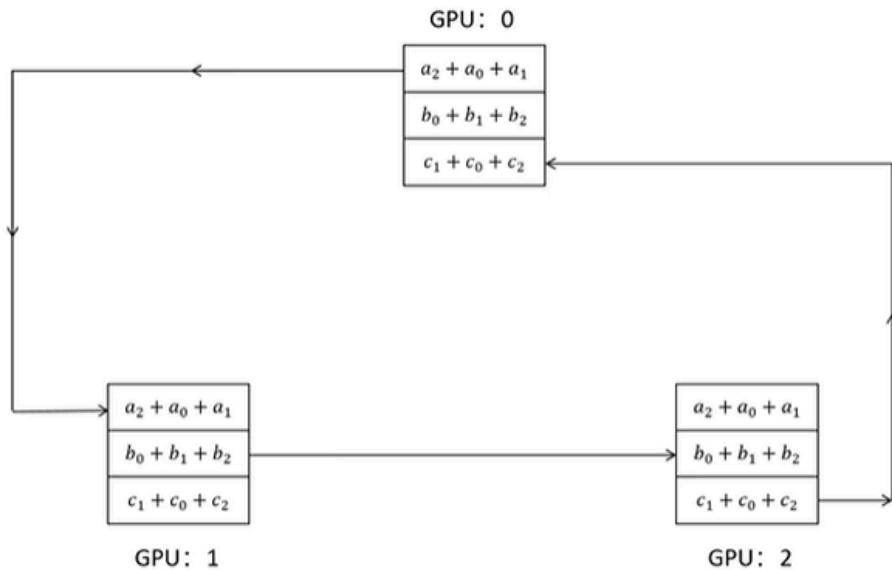
- 假设有n块GPU，那么将数据划分为n块，然后开始执行n-1次操作
- 第*i*次操作， GPU_j 会将自己的第 $(j - i)$ 块数据发送给 GPU_{j+1} ，并接收 GPU_{j-1} 的第 $(j - i - 1)$ 块数据

All-gather

- 将各个参数梯度求和值同步到其它GPU
- 第*i*块GPU的第 $(i + 1)$ 块数据传递给其它GPU
- 在第*i*次传递时， GPU_j 把自己的第 $(j - i - 1)$ 块数据发送给自己的右邻居，同时接受左邻居的第 $(j - i - 2)$ 块数据

Ring-AllReduce

LeviathanFan bili



FSDP (Fully Sharded Data Parallel)

核心原理：

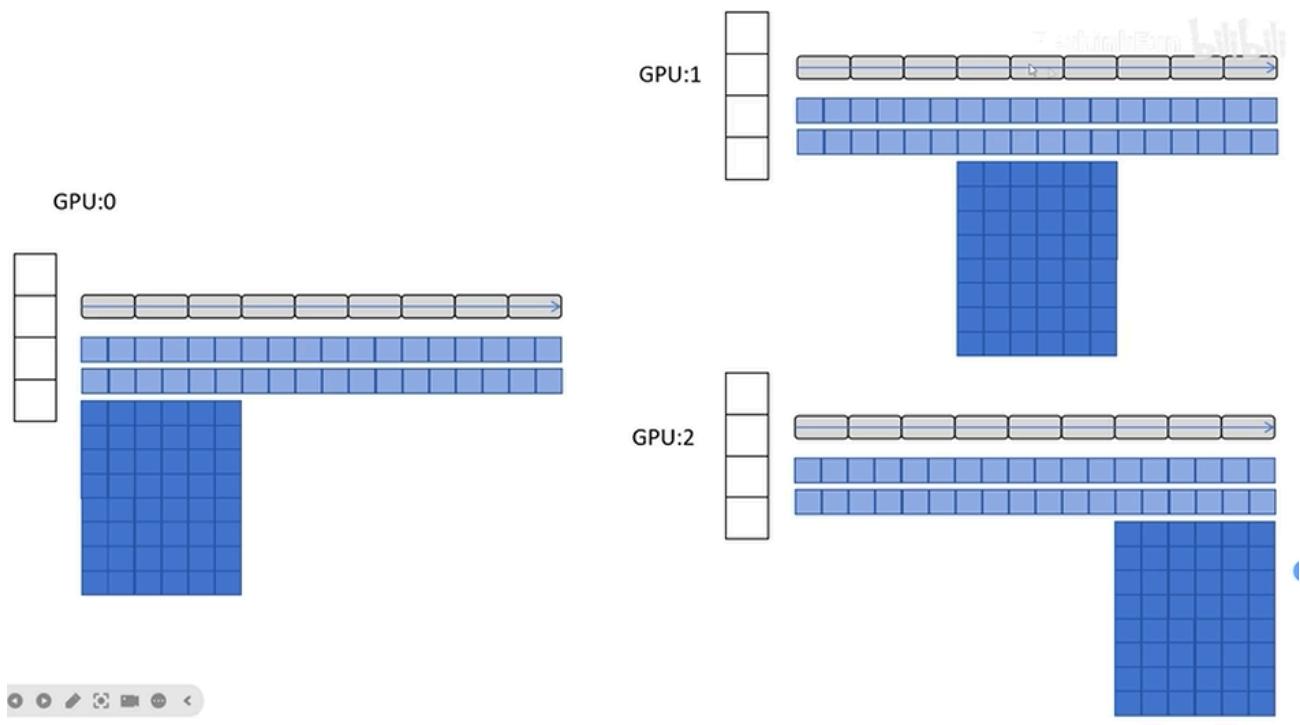
- 参数分片：对梯度、参数、优化器状态均匀拆分到各个GPU
- 按需通信：前向传播或者后向传播的时候依靠通信获取所需数据，使用完后立即释放
- 显存卸载：会将没用到的优化器状态或者参数从GPU显存卸载到CPU内存

DeepSpeed ZeRO-1

假设有3块GPU，使用混合精度训练，每个GPU都要存储以下东西



在GPU中，优化器状态占用大部分显存，ZeRO-1的出发点就是每块GPU只存储一部分优化器状态，存储了那部分优化器的状态就负责那几层网络的参数更新



工作流程

- 进行前向传播
- 进行后向传播的同时，GPU0和GPU1把计算出来的梯度传给GPU2，让GPU2去更新网络参数，其它GPU同理
- 反向传播完毕后，每个GPU更新各自优化器的梯度、一阶动量、二阶动量、FP32参数和FP16网络参数、梯度
- 最后把FP16参数广播到每个GPU完成一次训练

通信量分析

假设参数量位 ψ ，节点数为 N

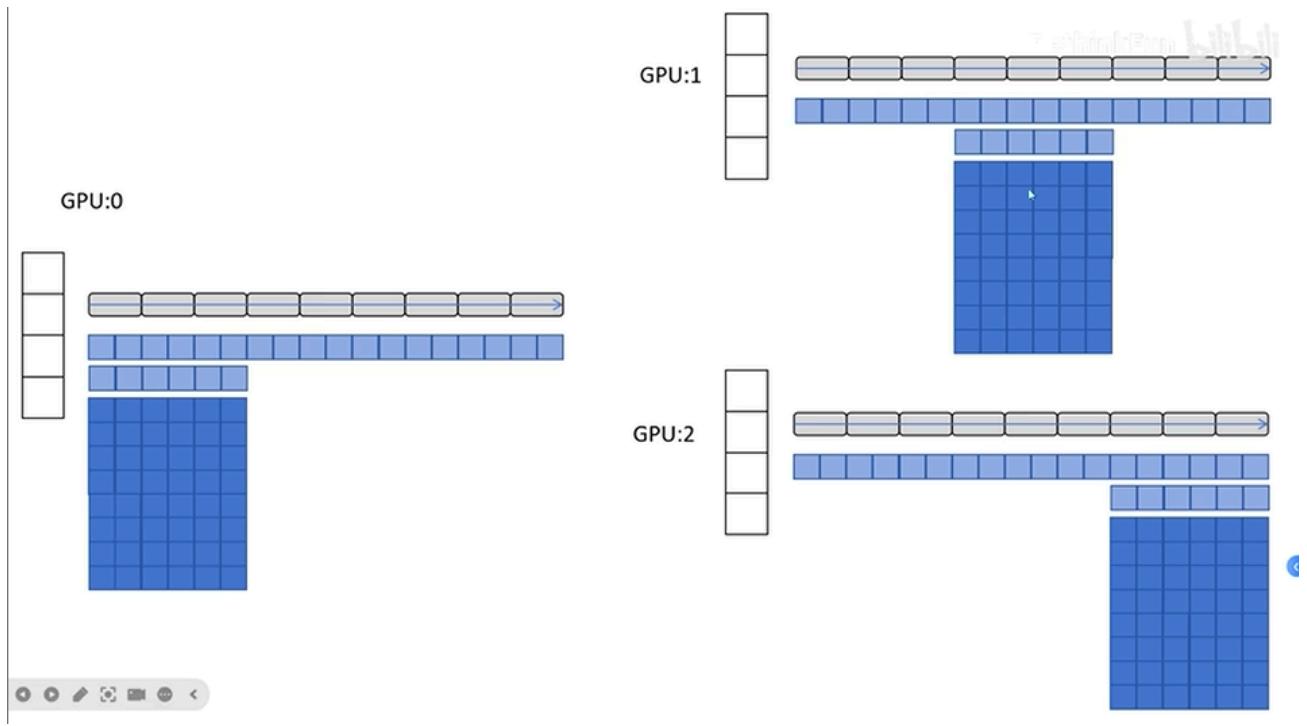
对于每个GPU进程：

- 梯度收集阶段传入/传出： $(N - 1)\frac{\psi}{N} \approx \psi$
- 参数广播阶段传入/传出： $(N - 1)\frac{\psi}{N} \approx \psi$

总传入/传出： 2ψ

DeepSpeed ZeRO-2

ZeRO-2中，把FP16的梯度也进行了划分，即GPU不再保存自己用不到的梯度



工作过程

- 进行前向传播
- 进行后向传播时，GPU0和GPU1计算出来梯度后，立即传递给GPU2，然后进行释放，其它GPU也同理
- 反向传播完毕后，每个GPU更新各自优化器的梯度、一阶动量、二阶动量、FP32参数和FP16网络参数、梯度
- 最后把FP16参数广播到每个GPU完成一次训练

通信量分析

假设参数量位 ψ ，节点数为 N

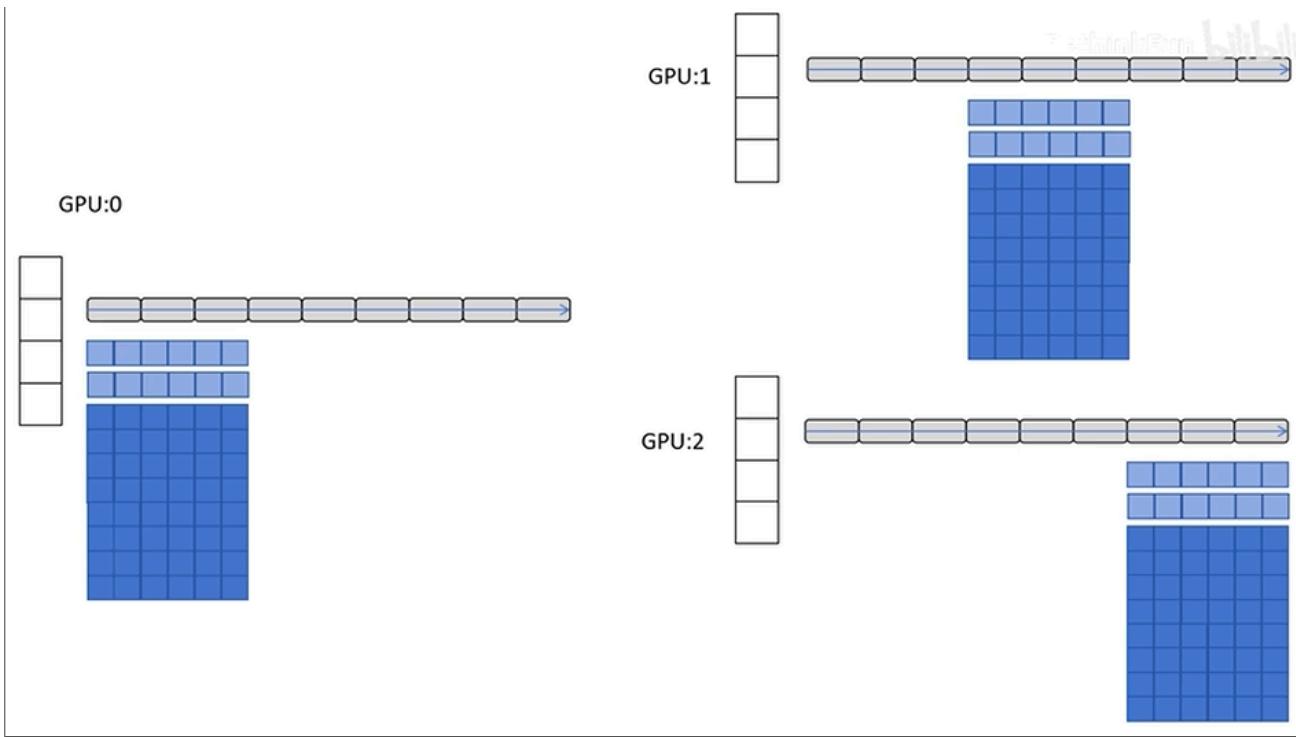
对于每个GPU进程：

- 梯度收集阶段传入/传出： $(N - 1) \frac{\psi}{N} \approx \psi$
- 参数广播阶段传入/传出： $(N - 1) \frac{\psi}{N} \approx \psi$

总传入/传出： 2ψ

DeepSpeed ZeRO-3

ZeRO-3对FP16参数也进行了划分



工作过程

- 前向传播，GPU遇到自己没有的参数，靠其它GPU来进行广播，计算完后立即释放，不占用显存
- 反向传播同样要用到参数，这时候需要利用广播获取参数，使用完参数后立即释放
- 每个GPU利用优化器进行参数更新，每个GPU仅更新自己分区的参数
- 最后把参数进行广播，完成一次训练

通信量分析

假设参数量位 ψ ，节点数为 N

对于每个GPU进程：

- 梯度收集阶段传入/传出： $(N - 1)\frac{\psi}{N} \approx \psi$
- 参数广播阶段传入/传出： $2(N - 1)\frac{\psi}{N} \approx 2\psi$

总传入/传出： 3ψ

模型并行

张量并行 (TP)

核心原理：将模型中的张量进行拆分然后分配到不同的GPU上

列并行

将权重矩阵按列进行切分

假设有 $Y = XW$ 其中：

- $X \in R^{2 \times 2}$, $W \in R^{2 \times 2}$

可以看成

$$[y_1 \quad y_2 \setminus \textcolor{red}{y}_3 \quad y_4]$$

=

$$[x_1 \quad x_2 \setminus \textcolor{red}{x}_3 \quad x_4]$$

$$[w_1 \quad w_2 \setminus \textcolor{red}{w}_3 \quad w_4]$$

然后按列进行分割，分配到两块GPU上进行计算 $W =$

$$[W_0 \quad W_1]$$

接着 $Y_0 = XW_0 =$

$$[x_1 \quad x_2 \setminus \textcolor{red}{x}_3 \quad x_4]$$

$$[w_1 \quad w_3]$$

=

$$[y_1 \setminus \textcolor{red}{y}_3]$$

$\setminus Y_1 = XW_1 =$

$$[x_1 \quad x_2 \setminus \textcolor{red}{x}_3 \quad x_4]$$

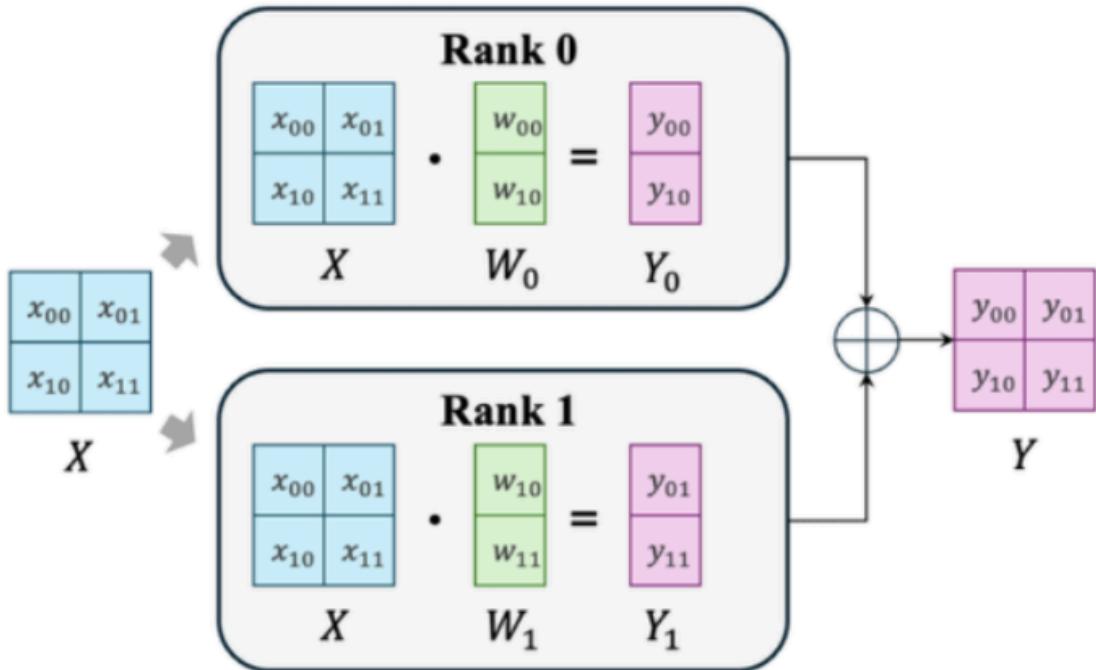
$$[w_2 \quad w_4]$$

=

$$[y_2 \setminus \textcolor{red}{y}_4]$$

然后进行拼接 $Y =$

$$[Y_0 \quad Y_1]$$



$$W = [W_0 \ W_1], Y = [Y_0 \ Y_1]$$

(b) 列并行

行并行

将权重矩阵按行进行切分

假设有 $Y = XW$ 其中：

- $X \in R^{2 \times 2}, W \in R^{2 \times 2}$

可以看成

$$[y_1 \ y_2 \ y_3 \ y_4]$$

=

$$[x_1 \ x_2 \ x_3 \ x_4]$$

$$[w_1 \ w_2 \ w_3 \ w_4]$$

然后按列进行分割，分配到两块GPU上进行计算 $W =$

$$[W_0 \ W_1]$$

此时 X 也要进行切割 $X =$

$$[X_1 \ X_2]$$

接着进行计算 $Y_1 =$

$$[x_1 \ x_3]$$

$$[w_1 \quad w_2]$$

=

$$[y_{11} \quad y_{12} \quad \textcolor{red}{y}_{31} \quad y_{32}]$$

\ Y_2 =

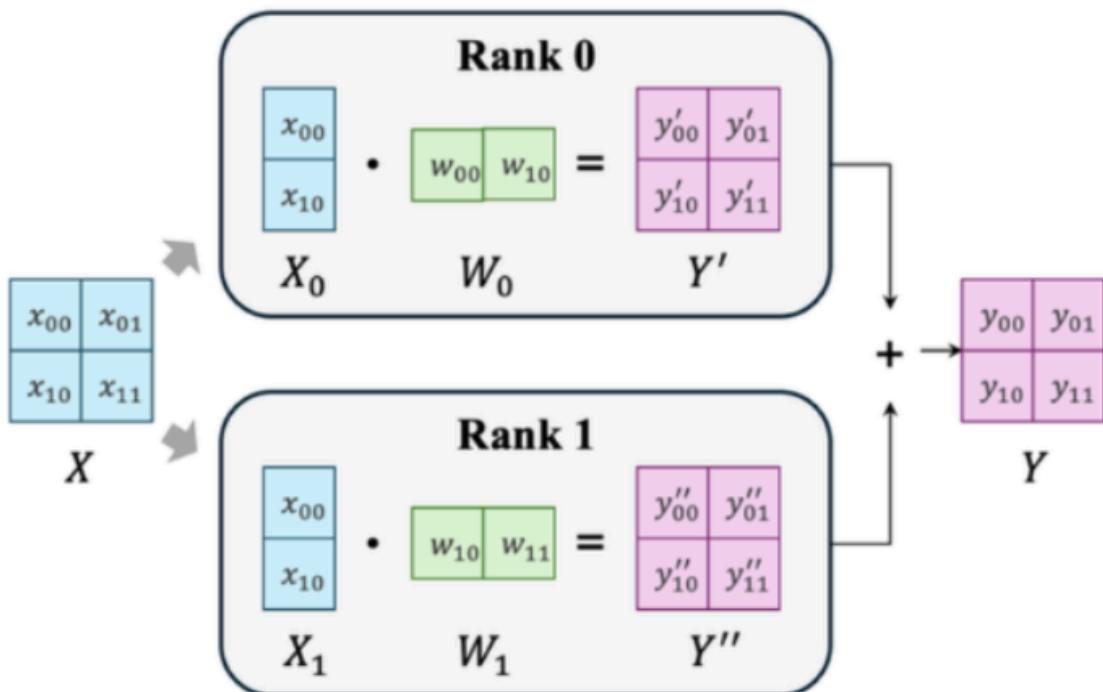
$$[x_2 \quad x_4]$$

$$[w_3 \quad w_4]$$

=

$$[y_{23} \quad y_{24} \quad \textcolor{red}{y}_{43} \quad y_{44}]$$

最后把 Y_1 和 Y_2 进行相加得到最终输出 $Y = Y_1 + Y_2$



$$X = [X_0 \quad X_1], W = [W_0 \quad W_1]^T, Y = Y' + Y''$$

(c) 行并行

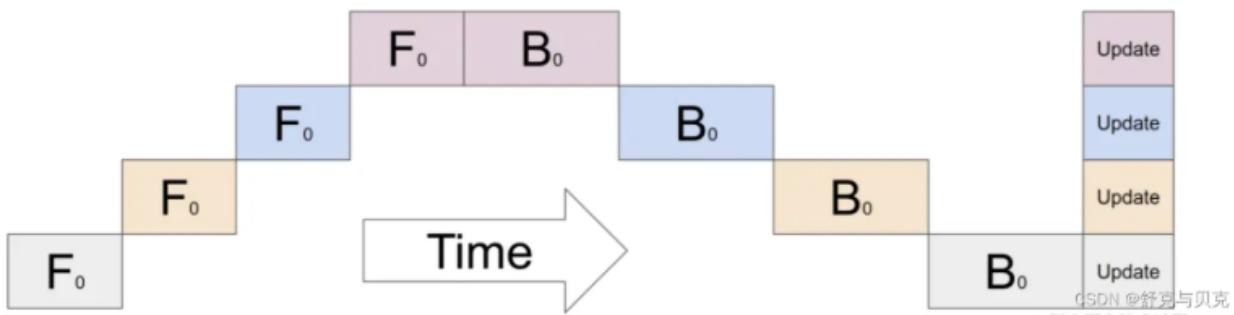
CSDN ©

流水线并行 (PP)

本质上是层间并行，将模型的不同层分发到不同的GPU上

朴素流水线并行

- 将模型按照层间切分成多个部分 (Stage)，并将每个部分 (Stage) 分配给一个 GPU
- 对小批量数据进行常规的训练，在模型切分成多个部分的边界处进行通信



工作流程

- 模型进行前向传播，到边界处把结果张量传递给GPU2
- GPU2计算得到模型的输出
- 进行反向传播，反向传播至边界处将梯度发送给GPU1，GPU1继续进行反向传播

存在的问题

- GPU利用率低，在任意时刻只有一个GPU工作
- 计算和通信没有重叠

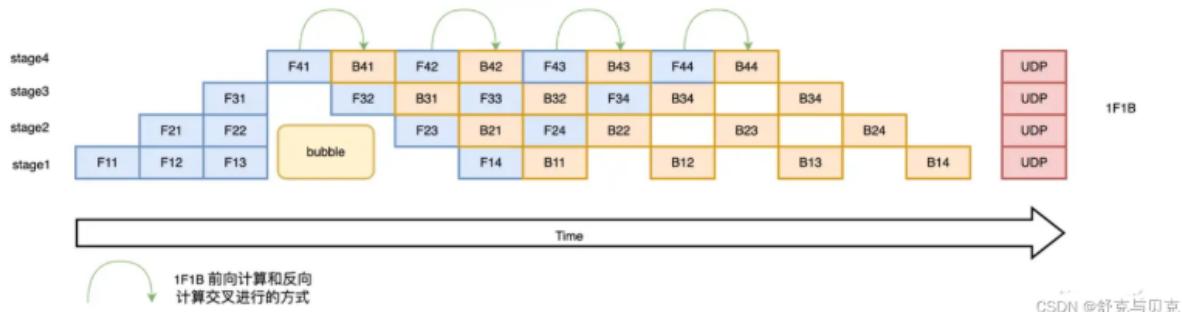
流水线并行策略

F-then-B策略

- 先进行前向计算，再进行反向计算
- 具体来说就是前向计算完一个mini-batch，再反向传播这个mini-batch

1F1B策略

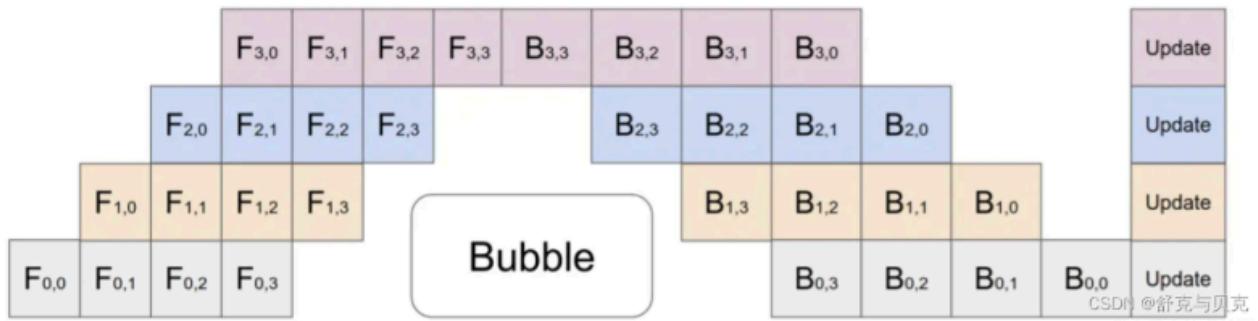
- 前向计算和反向计算交叉进行，可以释放不必要的中间变量
- 示例如下图所示，以 stage4 的 F42 (stage4 的第 2 个 micro-batch 的前向计算) 为例，F42 在计算前，F41 的反向 B41 (stage4 的第 1 个 micro-batch 的反向计算) 已经计算结束，即可释放 F41 的中间变量，从而 F42 可以复用 F41 中间变量的显存



- 显存占比明显下降

Gpipe流水线并行

谷歌提出的流水线并行，使用的是F-then-B策略



核心：它把一个Mini-batch，拆解成更小的Micro-batches，比如上图的把一个Mini-batch，拆成4个Micro-batches

好处：当前向计算的第一个Micro-batch1被GPU0计算完毕，它就会传递到模型的下一层GPU1，然后GPU0可以继续计算Micro-batch2

坏处：对于那些需要统计量的层（如：Batch Normalization），就会导致计算变得麻烦，需要重新实现。在Gpipe中的方法是，在训练时计算和运用的是micro-batch里的均值和方差，同时持续追踪全部mini-batch的移动平均和方差，以便在测试阶段进行使用

PipeDream --- DeepSpeed

微软提出的流水线并行策略，非交错式1F1B

Gpipe的流水线有以下问题：

- 将mini-batch切分成m份micro-batch之后，会带来更频繁的流水线刷新（当GPU流水线并行训练时出错进行检查点重载）
 - Pipeline Flush（流水线刷新）是流水线并行系统中的一种重置操作，其本质是：
 - 丢弃所有未完成的计算（半成品微批次）
 - 释放被占用的计算/通信资源
 - 回滚到最近的安全状态（如检查点）
- 将mini-batch切分成m份micro-batch之后，需要缓存m份激活值，会导致内存增加

PipeDream具体方案如下：

- 一个阶段做完一次micro-batch的前向传播之后，就立刻进行mirco-batch的后向传播，然后释放内存
- 如下图所示，machine1先执行micro-batch的forward，然后把激活值传给machine2，以此类推
- 从machine4进行反向传播，逐步传递梯度至machine1，然后逐渐进入稳定状态（1F1B）
- 斜线的方块就是Bubble，也就是GPU的空闲时间

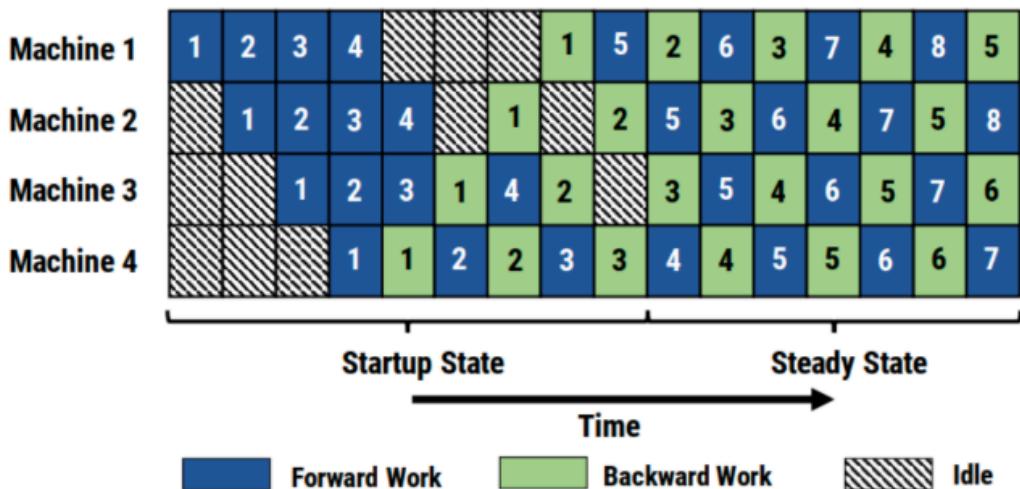


Figure 8: An example pipeline with 4 machines, showing startup and steady states.

© 2023 CSDN @舒克与贝克

问题：

- 当前向传播的5号Micro-batch在Machine1上就开始传递的时候，实际上它使用的权重是Micro-batch 1做完了反向传播之后更新的权重，此时FW2-4并没有进行更新
 - 如果 Micro-batch 5 的前向计算使用了 最新版本的权重（即 Micro-batch 1 反向更新后的权重），而 Micro-batch 5 的反向计算又使用了 更晚更新的权重（如 Micro-batch 4 反向更新后的权重），就会导致梯度计算不一致，影响训练稳定性
- 同理，Machine2上FW5又是在batch1-2做完反向传播后进行更新的，而Machine1是在做完了batch1的反向传播后进行更新的，Machine1和Machine2的权重又起了冲突

解决方法：

在1F1B的基础上，PipeDream引入了Weight stashing和Vertical Sync两种技术来矫正权重的冲突和同步

- Weight stashing (权重暂存)：为权重维护多个版本，每个 Micro-batch 的前向计算所使用的权重会被暂存，并在其反向计算时复用
 - 具体流程：
 - 前向计算时：Machine 1 使用 当前最新版本的权重 计算 Micro-batch 5 的前向结果，并 保存该版本的权重（记为 W_s ）
 - 反向计算时：Machine 1 在计算 Micro-batch 5 的反向传播时，使用之前暂存的 W_s ，而不是最新的权重
 - 参数更新：反向传播完成后，梯度更新仍然作用于最新版本的权重（即 W_s 的梯度会更新到当前最新的权重上）
- Vertical Sync (垂直同步)：每个 Micro-batch 进入流水线时，会记录当前最新版本的权重，并在整个生命周期（所有 Stage）中使用该版本
 - 具体流程：
 - Micro-batch 进入流水线时，Machine 1 会记录当前最新版本的权重（如 W_1 ），并 将该版本号传递给后续所有 Stage
 - 所有 Stage 在处理该 Micro-batch 时，都使用 W_1 ，而不是各自的最新权重

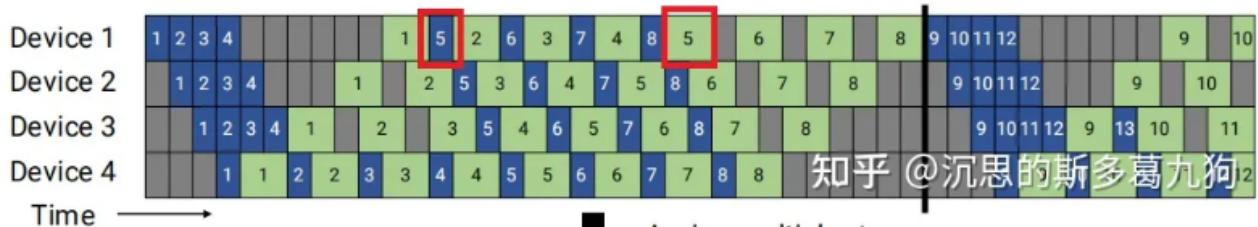
- c. 梯度更新：反向传播完成后，梯度仍然更新到 **最新版本的权重**（即 W_1 的梯度会更新到当前最新的权重上）

问题解析

问题1：同一个微批次数据，相同的device（相同stage），在前向计算和反向计算，采用不同版本的模型参数

示例：

- Device 1 的微批次5数据，在前向传播使用了第1个版本模型（微批次1反向传播完成），
- Device 1 的微批次5数据，在反向传播使用了第4个版本模型（微批次1、2、3、4反向传播完成）



解决办法：Weight Stashing 方法

每个device多备份几个不同版本的权重，确保同一个微批次数据，在前向计算和后向计算采用同一个版本的模型权重。计算前向传播之后，会将这份前向传播使用的权重保存下来，用于同一个 minibatch 的后向计算

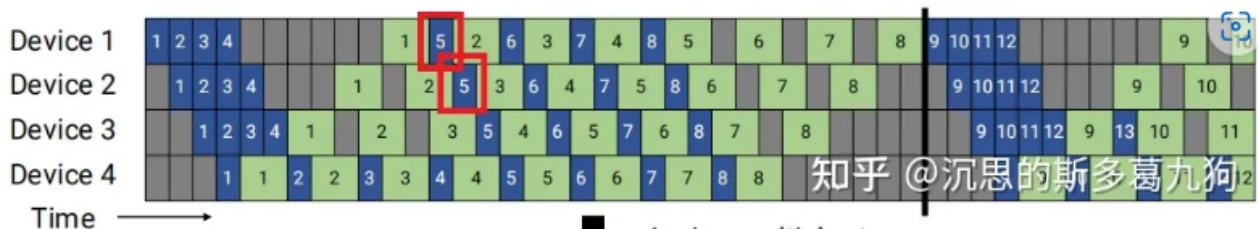
示例：

- Device 1 的微批次5数据，在前向传播使用了第1个版本模型（微批次1反向传播完成）
- Device 1 的微批次5数据，在反向传播使用了第1个版本模型（微批次1反向传播完成）

问题2：同一个微批次数据，相同的操作（都是前向或者都是反向），在不同的device上（不同stage），采用不同版本的模型参数

示例：

- Device 1 的微批次5数据，在前向传播使用了第1个版本模型（微批次1反向传播完成）
- Device 2 的微批次5数据，在前向传播使用了第2个版本模型（微批次1、2反向传播完成）



解决方法：Vertical Sync 方法

每个批次数据进入pipeline时都使用当前device（阶段）最新版本的参数，并且参数版本号会伴随该批次数据整个生命周期，从而实现了device（阶段）间的参数一致性

示例：

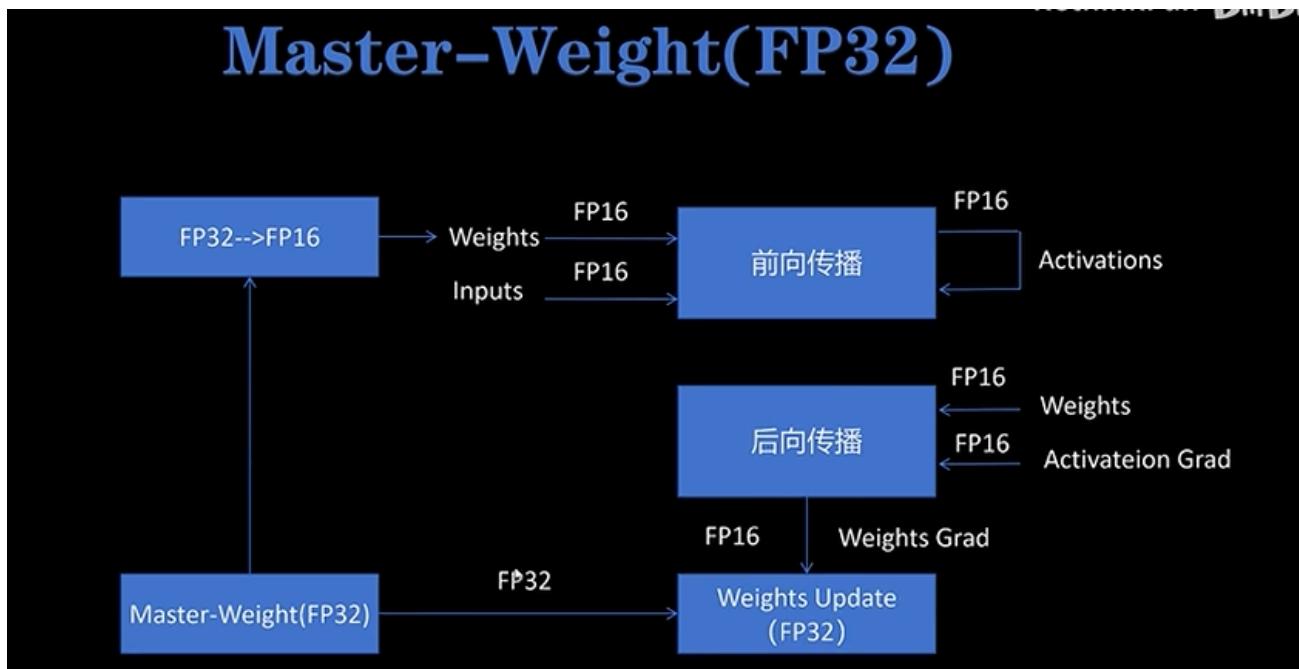
- Device 1 的微批次5数据，在前向传播使用了第1个版本模型（微批次1反向传播完成）
- Device 2 的微批次5数据，在前向传播使用了第1个版本模型（微批次1反向传播完成）

混合精度训练

大部分情况下，计算都是在float16下进行，但是优化器会保存一份float32的精度值（Master-Weight）

训练过程

1. 将Master-Weight转换为fp16
2. 和inputs一起进行前向传播、反向传播，存储的激活值也是FP16
3. 梯度传入优化器，被转换为fp32用于参数更新

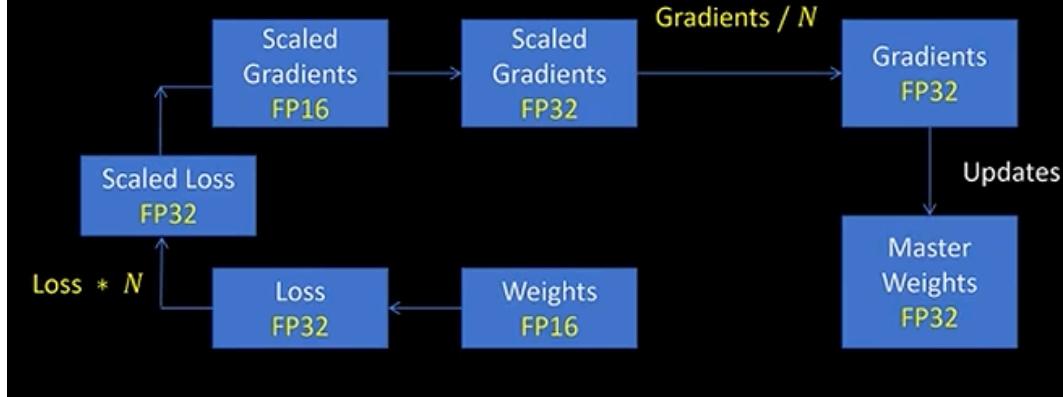


Loss Scaling

1. 用fp16精度的权重进行前向传播计算损失，损失为fp32
2. 然后进行损失缩放，得到fp32（缩放后）的损失
3. 再把损失转换为fp16的梯度，进行反向传播梯度计算，接着转回fp32的梯度
4. 再次进行缩放，得到fp32位的梯度，进行参数更新

将梯度进行放大，可以避开fp16的下溢区间（因为梯度一般来说都很小，会在FP16的下溢区间）

Loss Scaling



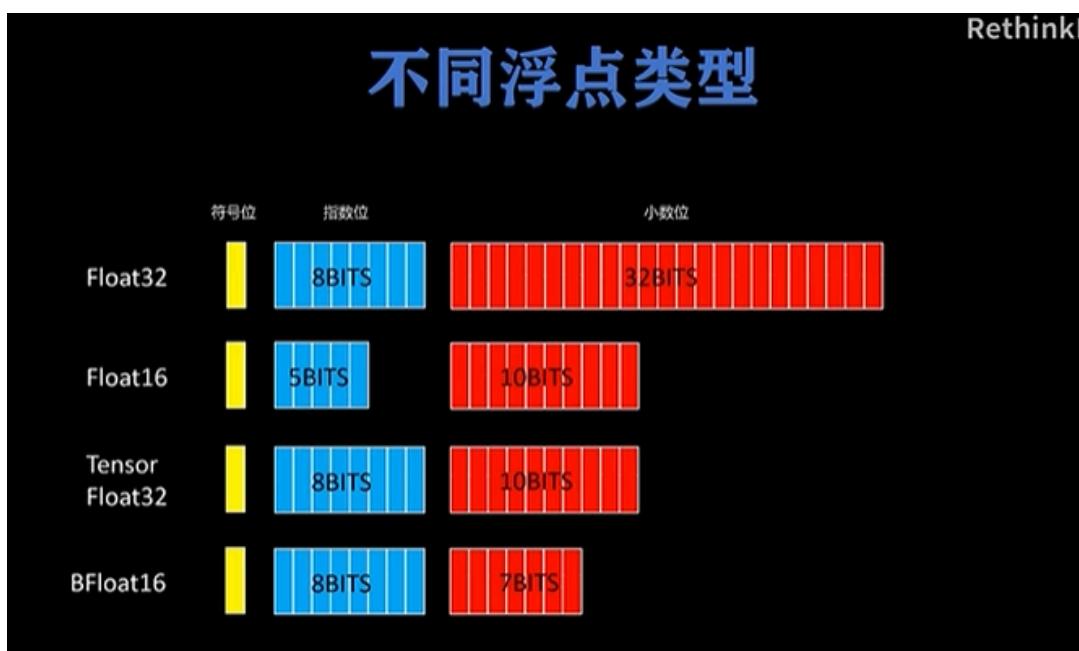
一般在线性层、卷积、RNN使用fp16，在累加过程中用fp32

浮点型的表示

将浮点数转换为二进制，再转换为科学计数法



各种精度



低精度带来的问题

1. 表示范围有限

2. 大数吃小数问题，大的数和小的数相加，小的数要转换为和大的数一样的指数表示，转换后的小数位太小，低精度无法表示，因此溢出

低精度带来的问题

RethinkFun bilibili

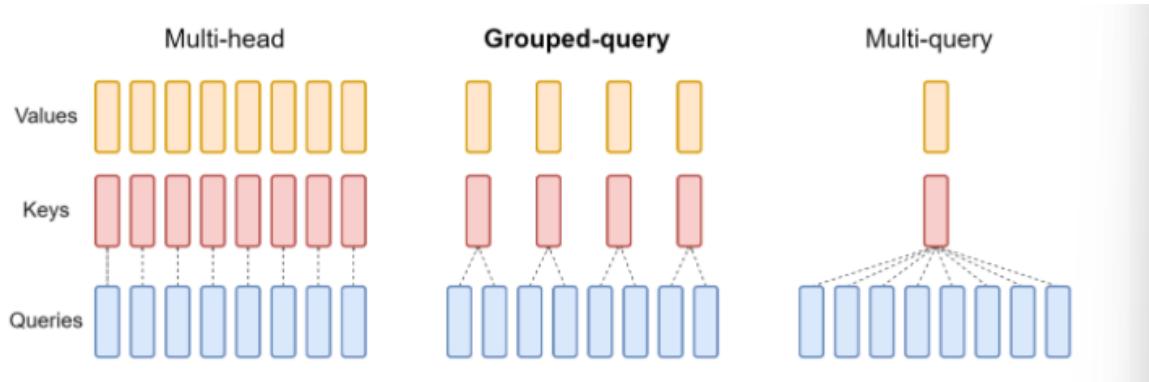
- 一. 表示范围问题
- 二. 大数吃小数问题

$2048 + 0.5 = 2048$

$2048 \quad 1.0 * 2^{11}$
 $0.5 \quad 1.0 * 2^{-1} \quad 0.00000000001 * 2^{11}$

注意力机制变体

MHA、MQA和GQA



MHA

之前说过，不赘述

MQA (Multi-Query Attention)

多头查询注意力

原理：

- 将原生Transformer每一层多头注意力的key矩阵、value矩阵改为该层下所有头共享，即K、V矩阵每层只有一个，Q矩阵不受影响。
- 在同一层的注意力机制中，多个头共享相同的K、V矩阵

- 层内KV共享，而不是跨层共享

好处与坏处：

- 大幅度减少了参数量，推理得到了加速
- 但会造成模型性能的损失，且训练的时候模型不稳定

GQA (Grouped-Query Attention)

分组查询注意力

原理：

- 将Query进行分组，每个组内共享一组Key、Value
- 令组的个数为N，N=1时为MQA，N等于Query的数量则退化为MHA

好处：

- 通过分组减少模型性能损失，使推理性能接近MHA，同时也减少了参数量

MLA (Multi-Head Latent Attention)

多头潜在注意力，在Deepseek-V3中提出

为何提出？

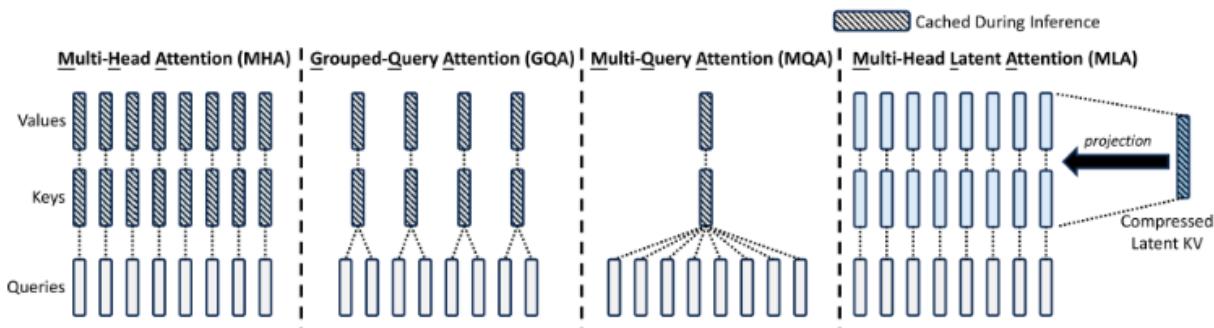
- KV Cache虽然加速了模型推理，但是占据了大量显存空间
- 上文提到的MQA和GQA虽然降低了KV Cache计算量（），但是性能损失太多

目的是什么？

- 降低推理过程中的KV Cache资源开销
- 缓解MQA，GQA对性能的损耗

怎么做的？

- 对Key和Value矩阵进行了一个低秩联合压缩（通过低秩转换为一个压缩的KV，使得存储的KV的维度显著减小）
- 如下图所示，阴影表示的是KV缓存，在MLA中的KV缓存是最少的



下面来看看怎么压缩的

对Query和Key进行拆分，拆分为 $[q_t^R \quad q_t^C]$, $[k_t^R \quad k_t^C]$ ，其中一部分做压缩(q_t^C, k_t^C)，另一部分做RoPE编码(q_t^R, k_t^R)

1. 输入隐藏层状态 h_t

2. 通过低秩分解生成 c_t^{KV} 、 c_t^Q

◦ c_t^{KV} 为 KV 潜在向量

◦ c_t^Q 为 Q 潜在向量

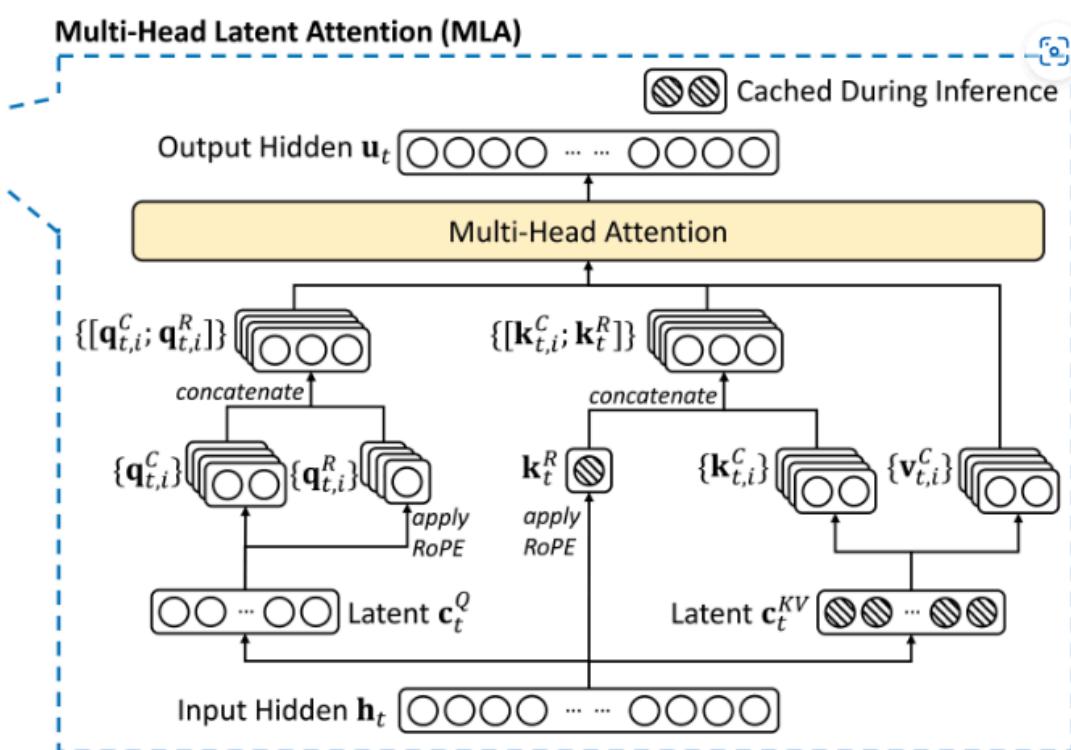
3. 将 Key 和 Value 分成两部分

◦ 静态缓存部分：推理时缓存的低维 Key 和 Value。即 $k_{t,i}^C, v_{t,i}^C$

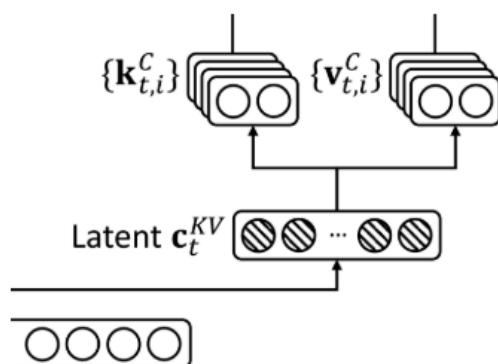
◦ 动态残差部分：补充压缩可能丢失的细节信息。即 $k_{t,i}^R$ (直接通过输入的隐藏层获取)

4. 同时对 Query 也分成两部分，一部分做压缩 $q_{t,i}^R$ ，一部分应用旋转编码 $q_{t,i}^C$ (这里是先压缩再分成两部分的)

5. 对 Q、K、V 的不同部分进行拼接，进行多头注意力计算



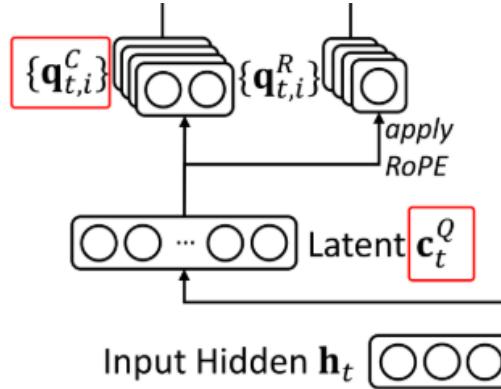
对 KV 进行联合压缩



三个变量 c_t^{KV} 、 k_t^C 、 v_t^C 分别通过如下三个公式得来 $c_t^{KV} = W^{\{KV\}}h_t \setminus k_t^C = W^{\{UK\}}c_t^{\{KV\}} \setminus v_t^C = W^{\{UV\}}c_t^{\{KV\}}$ 就是针对 KV 先一块降维，然后再分别升维

在推理的过程中，只需要缓存每一步的 c_t^{KV} ，然后再计算还原回原始的 K、V 即可

对 Q 压缩降维、再升维



公式如下 $c_t^Q = W_Q \cdot h_t + q_t^C = W_Q \cdot c_t^Q + q_t^R$ 这并不能降低 KV Cache，但可以减少训练过程中的激活内存

MLA 对 Q 和 K 的 RoPE 编码

MLA 需要对 k_t^R 和 q_t^R 应用 RoPE 编码，但是 RoPE 与 低秩压缩是不兼容的

- RoPE 对 Q 和 K 的编码是动态的，因此在推理的时候，RoPE 必须实时计算
- 如果对压缩后的 c_t^{KV} 应用 RoPE，这就意味着旋转矩阵 R_n 与低秩矩阵 W_{QK} 耦合，并且无法缓存 K_{rot} ，因为 RoPE 需要进行实时计算，KV Cache 的优势丧失

$$K_{\{rot\}} = R_n(W_{\{QK\}} \cdot c_t^{KV})$$

- 且矩阵乘法不满足交换律，无法将 W_{QK} 吸收到其它权重中

原始注意力分数 $S = (W^{\{UQ\}})^T (c_t^Q)^T R_m T_{nc} c_t^{\{KV\}} W^{\{UK\}}$ 它不等于 $S = (c_t^Q)^T R_m T_{nc} (W^{\{UQ\}})^T (W^{\{UK\}}) R_{nc} c_t^{\{KV\}}$ 因为矩阵不满足交换律，也就无法将 W_{QK} 吸收到其它权重中

MLA 通过对 RoPE 和 c_t^{KV} 进行解耦！！！来解决这个问题

1. 静态缓存部分 (k_t^C)：
 - 低秩压缩后的 K (c_t^{KV}) 不应用 RoPE，直接缓存
 - 推理时只需存储低维 c_t^{KV} ，显存占用极低
2. 动态 RoPE 部分 (k_t^R)：
 - 从输入 h_t 动态生成，单独应用 RoPE
 - 补充位置信息，避免与低秩矩阵的耦合
3. 合并计算
 - 最终 $K = [k_t^C; k_t^R]$ ，既保留了位置信息，又利用了 KV Cache

MFA (Multi-matrix Factorization Attention)

多矩阵分解注意力

没博客和视频教啊！！！后面看论文吧

注意力机制加速

Paged Attention

内存优化

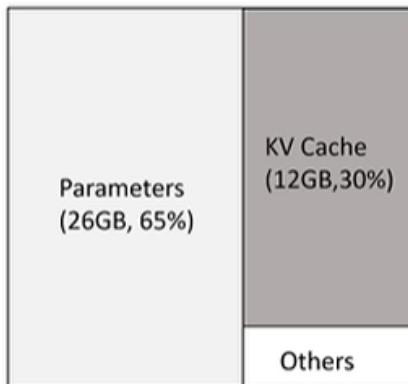
核心思想：将操作系统的内存分页思想引入显存管理，大幅度降低显存占用

首先要明确：

- GPU、CPU对连续内存的访问效率远高于非连续内存
- 频繁申请/释放GPU显存会让其显存变得零散，降低利用率

传统注意力计算的显存有三种类型的浪费

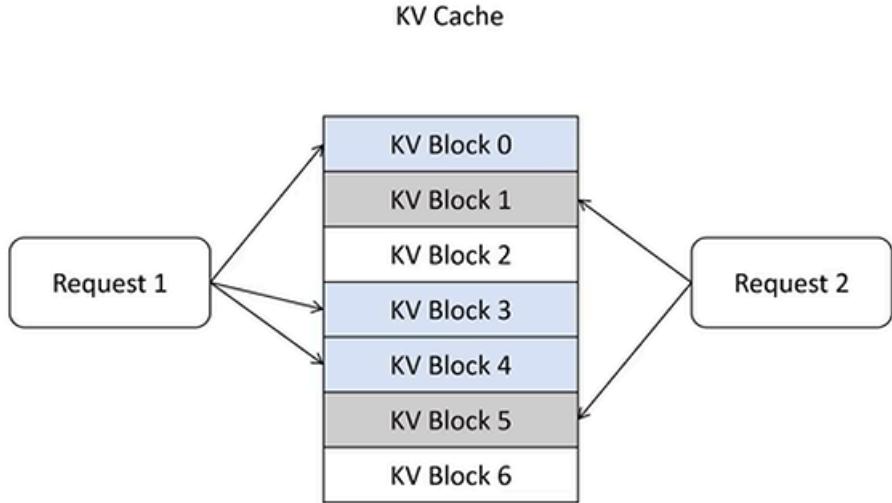
- 推理的时候是按照**最大序列长度预分配显存的**，如果输出的序列**没这么长**，会导致显存浪费
 - 比如输出的序列是“我爱中国”，模型输出最大序列长度是1024，那么预分配的显存就是1024个token，但输出序列只占了4个token，造成显存大量浪费
- 大模型的推理是**自回归任务**，下一个token即使还未生成，显存已经为其预留好了，无法和**其它生成任务并行**
 - 比如我这个任务要生成**1000个token**，现在生成到第5个token，此时有个新的任务**只需要生成10个token**，但后面995个token的显存已经被占用了，**新任务无法分配到显存**
- 显存利用碎片化，显存分配是随机的
 - **已释放的显存空间不连续**：显存占用情况：[空闲500MB] [已占1GB] [空闲300MB] [已占2GB]。假设有一个任务占用500MB显存，现在任务完成将他释放，但是新任务需要600MB显存，**没法为其分配连续的显存，报错**
 - **序列间的显存块太小**：显存被多个**并发生序列分割占用**，剩余的空闲块尺寸小于**新请求所需的最小单位**



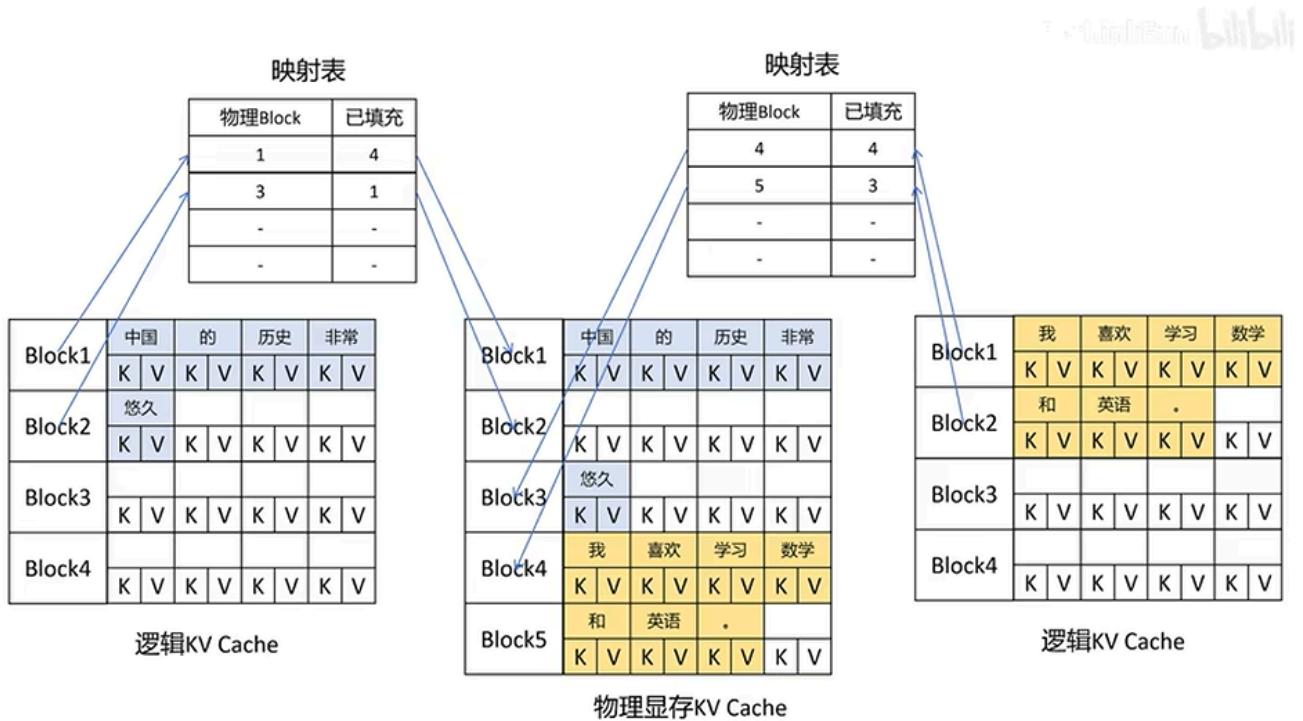
13B LLM on A100-40GB

具体是怎么做的

- **分页存储**：vLLM 启动时会预先分配一批固定大小的显存页，形成一个空闲页池



- 动态分配显存:** 每次生成都会预留一页token的显存，当生成新token的时候检查当前页是否用完，用完了就从空闲页池申请新页
- 逻辑表映射:** 分配显存的时候，每一页物理上是不连续的，但是Paged Attention通过维护一个逻辑页表将物理不连续的显存页映射为逻辑上连续的KV Cache

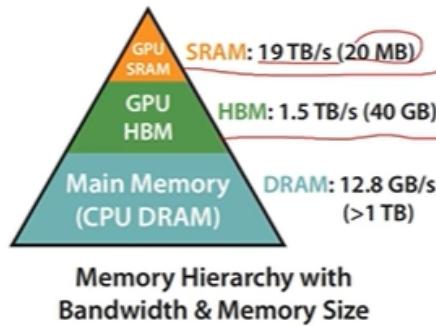


- 显存不足:** 会将不活跃的页换出到CPU，需要的时候再换回，或者尝试释放已使用完的页
- KV Cache共享:** 当Prompt相同的时候，Paged Attention会将相同的KV Cache进行共享
 - 页表映射共享:** 多个生成序列的逻辑页表可以指向同一个物理显存页，存储共享的KV数据
 - 写时复制(Copy-on-Write):** 当某个请求需要修改共享页的内容时，系统会先复制该物理页到一个新页（如从页3复制到页5），仅更新当前请求的页表，其他请求仍指向原页

Flash Attention

计算优化，着眼于减少IO访问量，通过芯片内缓存（SRAM中的）加快IO速度

看完下文GPU工作原理，你可以知道计算单元从SRAM读取数据的速度要比从HBM读取快得多，因此Flash attention的目标就是避免Attention计算的时候从HBM读写数据



这项技术为何而生？

- 传统注意力计算是从HBM读取数据进行计算，速度要慢得多
- 读取速度太慢，算力太强，导致GPU算力资源一直在等待数据读取
- 传统注意力计算不对矩阵进行分块处理，导致SRAM放不下这么大的数据，Pytorch虽然做了分块，但是需要多次读写中间结果，导致计算瓶颈

先来看看pytorch在attention上的实现

- 从HBM加载Q、K到SRAM
- 计算出 $S = QK^T$
- 将 S 写到HBM（这里是为了存储中间激活值，做反向传播）
- 从HBM读取 S 到HRAM
- 计算 $P = \text{softmax}(S)$
- 将 P 写到HBM（这里是为了存储中间激活值，做反向传播）
- 从HBM加载 P 到SRAM
- 计算 $O = PV$
- 把 O 写到HBM
- 返回 O

传统计算的性能瓶颈

- **Compute-Bound (计算瓶颈)：**
 - 来源：大的矩阵乘法，多通道的卷积
 - 程序性能受限于GPU的计算能力，大部分时间花在数值运算或逻辑处理上
- **Memory-Bound (内存瓶颈)：**
 - 来源：ReLU, Softmax, Sum, Dropout等
 - 程序性能受限于内存访问速度，GPU经常等待数据加载

先介绍下Safe-Softmax

传统softmax $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$ 若使用混合精度进行训练，当输入数值 x_i 非常大时，会造成数值溢出

Safe Softmax $\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_{j=1}^n e^{x_j - \max(x)}}$

- 通过减去输入的最大值 $\max(x)$ 来避免数值溢出
- 相当于分子分母同时除以 $e^{\max(x)}$ ，结果不变

核心工作原理是什么？

- **Tiling Algorithm (分块计算)**
 - 核心思想就是根据SRAM块的大小将大的矩阵拆分成小的矩阵送入SRAM进行计算
 - 传统注意力机制需要重复读写SRAM，而Flash Attention将矩阵分块后在SRAM中一次性算出最终结果
- **Recomputation (重计算)**
 - GPU计算时间小于HBM读写时间，不再存储中间结果，反向传播的时候重新计算即可
 - 对于 $S = QK^T$ ，Flash Attention不进行HBM的存储，而是继续在SRAM中计算
 - 对于 $P = \text{softmax}(S)$ ，也不进行HBM的存储，而是继续集散
- **Kernal Fusion (融合计算)**
 - 将多个操作融合成一个操作，以此减少HBM的读写
 - 分块计算可以用一个Kernel完成注意力计算的所有操作

接下来看看分块计算在SRAM中是如何一次性到位的

- 首先取Q、K、V的小块，送入SRAM
- 计算 $S = QK^T$
- 计算 $P = \text{softmax}(S)$
 - 这里并不是全局的softmax，而是局部的softmax
 - 取出局部最大值，然后使用Safe-Softmax
 - 下一次进入Safe-Softmax的时候，用上次的局部最大值和当前块的最大值算出新的最大值，然后进行Safe-Softmax的更新
 - 举例：分为2块计算， $x_1 = [1, 2] x_2 = [3, 4]$
 - 计算第一块内的最大值 $m(x_1) = 2 = m(x)$
 - 第一个块内进行指数计算， $f(x_1) = [e^{1-m(x_1)}, e^{2-m(x_1)}] = [e^{-1}, e^0]$
 - 第一个块计算归一化因子 $l(x_1) = e^{-1} + e^0$
 - 操作第二个块，更新最大值 $m(x) = \max(m(x_1), m(x_2)) = 4$
 - 同上， $f(x_2) = [e^{3-m(x_2)}, e^{4-m(x_2)}] = [e^{-1}, e^0]$
 - 第二个块计算归一化因子， $l(x_2) = e^{-1} + e^0$
 - 计算全局 $f(x)$ 和 $l(x)$
- $f(x) = [e^{m(x_1)-m(x)}f(x_1), e^{m(x_2)-m(x)}f(x_2)]$ $f(x) = [e^{-2}(e^{-1}, e^0), e^0(e^{-1}, e^0)]$
 $l(x) = e^{m(x_1)-m(x)} * l(x_1) + e^{m(x_2)-m(x)} * l(x_2)$ $l(x) = e^{-3} + e^{-2} + e^{-1} + e^0$
- 计算 $O = PV$
- 从SRAM中取出 O 回到HBM

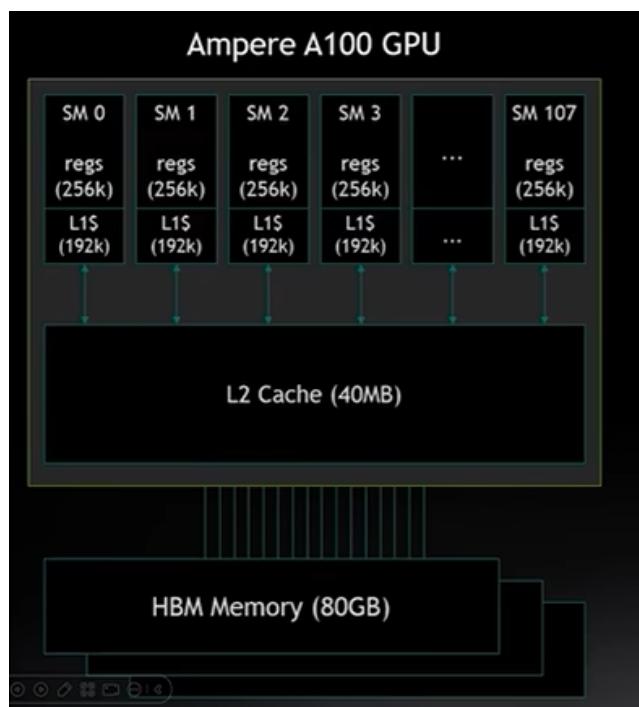
GPU工作原理

使用 SIMT 单一指令，多线程进行，比如矩阵乘法里结果里的每个元素可以分配一个线程

GPU任务调度的最小单元：Warp (32个线程一组)

SM (流式多处理器)：GPU核心计算单元

- CUDA Cores (CUDA 核心)：执行基本算术和逻辑运算
- Tensor Cores (张量核心)：专用于加速矩阵乘法，支持混合精度计算
- Warp Scheduler (Warp 调度器)：管理Warp的执行，调度线程用的
- Shared Memory：供同一线程块的线程共享数据，可显式控制
- L1 Cache：缓存频繁访问的数据，硬件自动管理，无法直接控制
- Load/Store Units：负责从全局内存 (HBM) 或者共享内存 (Shared Memory) 加载数据
- Register File (寄存器文件)：存储线程的私有变量
- SFU：执行超越函数或者复杂运算



GPU的存储分为芯片内和芯片外：

- SRAM (芯片内)：L1 Cache (仅对当前 SM有效)，L2 Cache (全局共享，所有SM共用)，Shared Memory (同一线程块共享)，用来存储数据，供计算单元快速读取
- HBM (芯片外)：我们常说的显存 (访问速度慢，空间大)，所有线程可访问

访问速度

Data Location	Bandwidth (GB/sec)
L1 Cache	19,400
L2 Cache	4,000
HBM	1,555
NVLink	300
PCIe	25

- NVLink : GPU间高速互联 (分布式训练)
- PCIe : CPU-GPU通信总线

因此要让计算模块尽可能多的从寄存器读取数据进行计算，因为寄存器带宽极高，访问速度极快

训练的时候数据传输流程：

1. 数据加载：通过PCIe把数据从CPU内存拷贝到GPU HBM
2. 计算准备：数据从HBM加载到L2 Cache然后到Shared Memory或者L1 Cache，最后传输到寄存器（线程私有）
 - 先从HBM到全局共享区，所有SM共用，然后加载到特定SM，线程块可用，然后加载到指定线程的寄存器，给核心计算
3. 核心计算：CUDA Core或者Tensor Core参与计算
4. 数据传回：算好的数据传回HBM