

LLM 综述

xbZhong

2025-08-20

Contents

大模型训练过程	1
SFT (Supervised Finetune)	1
RL (Reinforcement Learning)	2
LLaMA	8
GPT	9
Deepseek	9
分布式训练	9
混合精度训练	22
注意力机制变体	23
注意力机制加速	29
GPU 工作原理	34

本页 PDF

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”。

Figure 1: image-20250817002125325

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

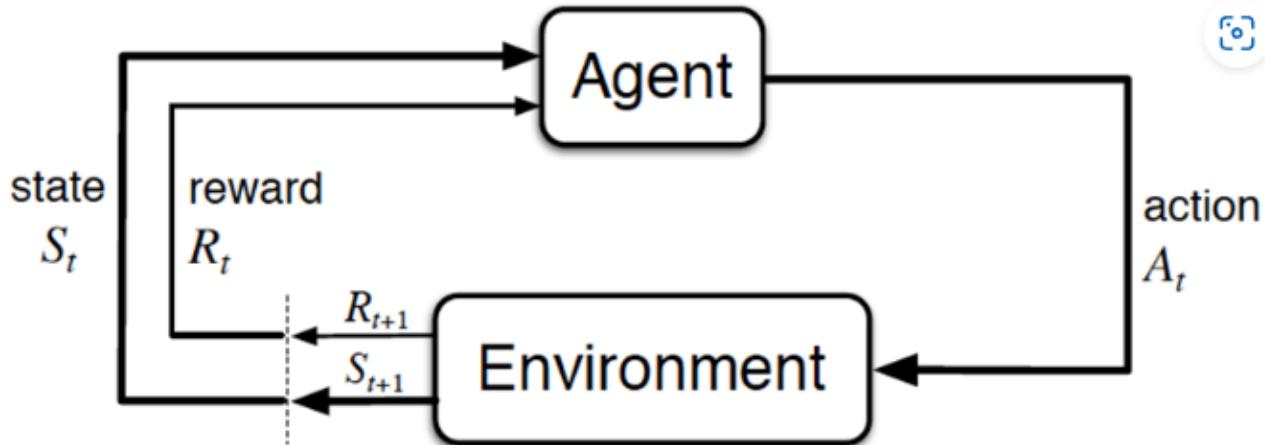


Figure 2: image-20250817115944207

- **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

$$\begin{array}{c}
 & & & \text{Today} \\
 & & & \begin{bmatrix} \text{sun} & \text{cloud} & \text{rain} \end{bmatrix} \\
 \text{Yesterday} & \begin{matrix} \text{sun} \\ \text{cloud} \\ \text{rain} \end{matrix} & \begin{bmatrix} 0.50 & 0.375 & 0.125 \\ 0.25 & 0.125 & 0.625 \\ 0.25 & 0.375 & 0.375 \end{bmatrix}
 \end{array}$$

Figure 3: image-20250817120703506

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

$$\begin{bmatrix} S_1 \rightarrow S_1 & S_1 \rightarrow S_2 & S_1 \rightarrow S_3 \\ S_2 \rightarrow S_1 & S_2 \rightarrow S_2 & S_2 \rightarrow S_3 \\ S_3 \rightarrow S_1 & S_3 \rightarrow S_2 & S_3 \rightarrow S_3 \end{bmatrix}$$

- 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

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- 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

$$V(s) = E[G_t|S_t = s] = R_{t+1} + \underbrace{\gamma R_{t+2} + \gamma^2 R_{t+3} + \dots}_{\gamma V(S_{t+1})} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

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

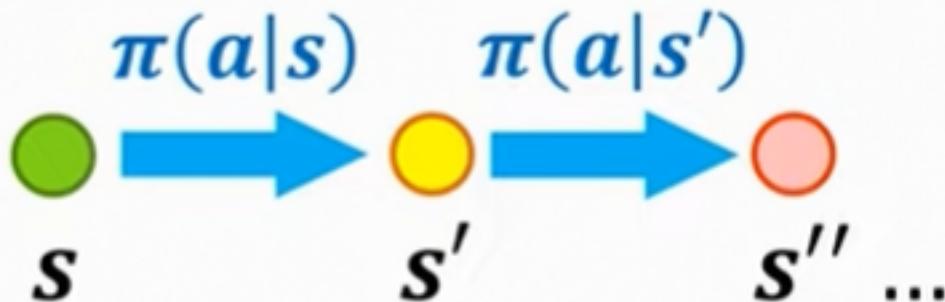
- 也就是通过加权求和把动作的维度消除, 减少运算量

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

- 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

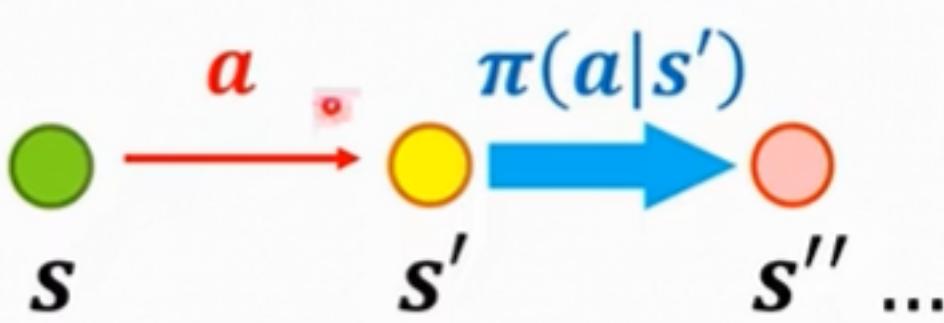
$$G_t = \underbrace{R_{t+1}}_{\text{立即奖励}} + \underbrace{\gamma R_{t+2} + \gamma^2 R_{t+3} + \dots}_{\text{后继状态的折扣价值}} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

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, 遵循策略 π 的期望回报



$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a]$$

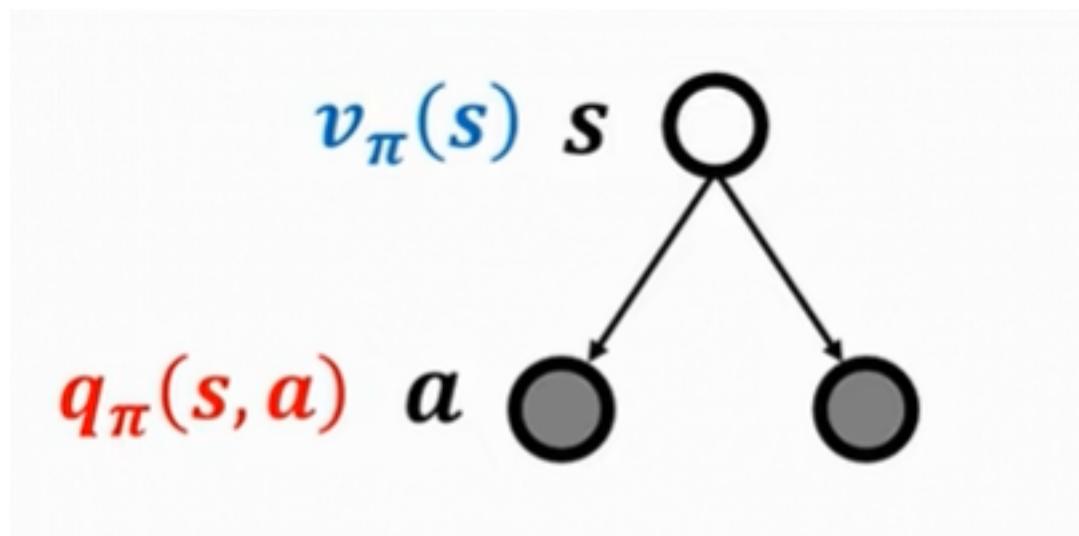
同上推导，可以得到**动作值函数的贝尔曼期望方程**

$$q_{\pi}(s, a) = E_{\pi}[R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

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

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

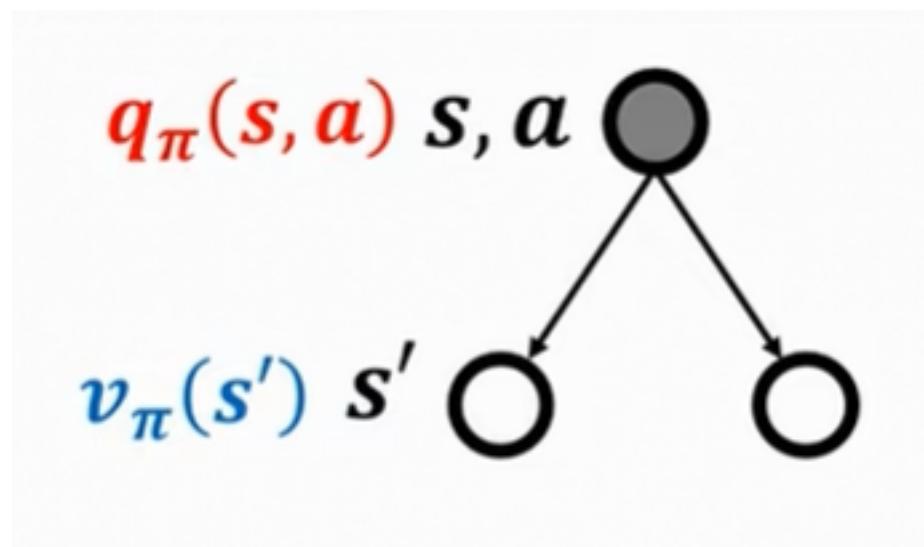


Figure 4: image-20250817181417147

贝尔曼最优方程

最优状态值函数

$$V_*(s) = \max_{\pi} V_{\pi}(s)$$

最优动作值函数

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

最优策略

- 存在一个最优策略，使 $\pi_* \geq any\pi$
- 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**

$$\pi_* \geq any\pi$$

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：

$$V_{\pi}(s) = \sum_{a \in A} \pi(a|s) (R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{\pi}(s'))$$

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** $q_{\pi}(s, a)$

$$\pi_*(a | s) = \begin{cases} 1 & \text{if } a = \arg \max_a q_*(s, a), \\ 0 & \text{otherwise.} \end{cases}$$

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

$$q_*(s, a) = R_s^a + \gamma \sum_{s'} P_{s,s'}^a V_*(s')$$

最后得出状态值函数的贝尔曼最优方程

$$V_*(s) = \max_a (R_s^a + \gamma \sum_{s'} P_{s,s'}^a V_*(s'))$$

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

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

代入 $V_*(s')$ ，得到动作值函数的贝尔曼最优方程

$$q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a')$$

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

$$RMSNorm(x) = \frac{x}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon}} \cdot \gamma$$

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

$$Swish(x) = x \cdot \sigma(x)$$

- σ 为 Sigmoid 函数

GLU 激活函数

$$GLU(x) = \sigma(W_1x + b_1) \odot (W_2x + b_2)$$

- \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

$$SwiGLU(x) = Swish(Linear_1(x)) \odot Linear_2(x)$$

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

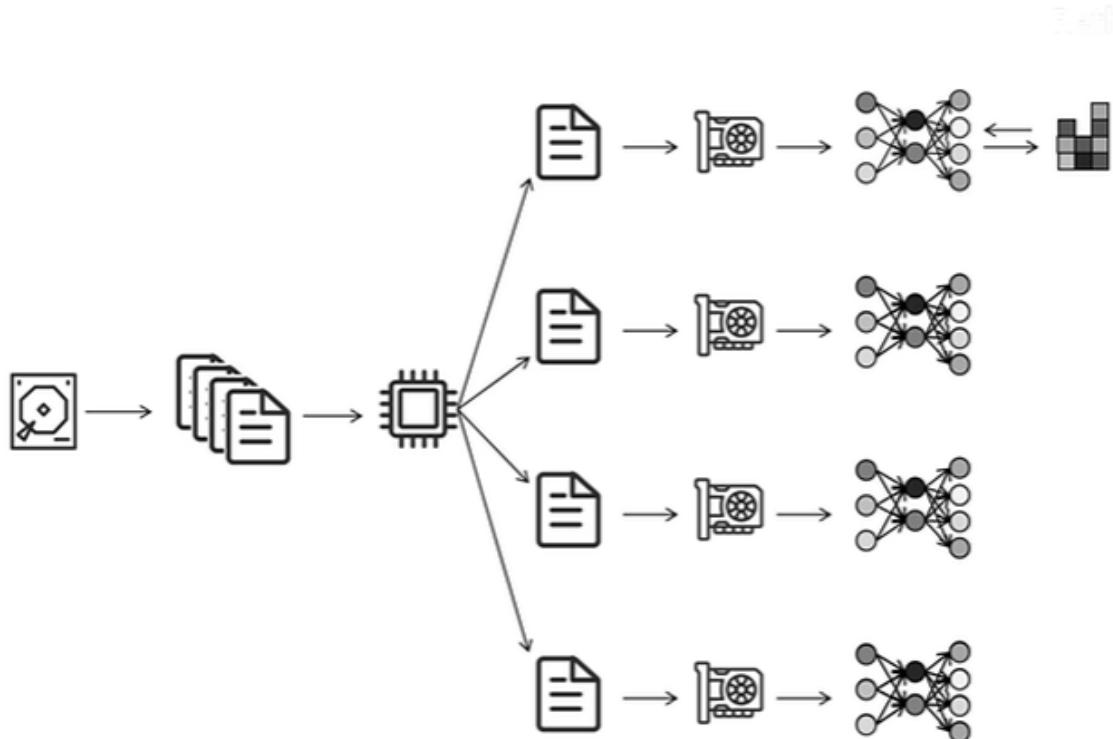


Figure 5: image-20250814165126760

工作流程

- 用 CPU 将数据分成多份，给每个 GPU 一份
- 每个 GPU 独自进行训练，将自己计算的梯度传递到 GPU0 上
- GPU0 用全局平均梯度更新自己的网络参数
- 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 $(j-i)\%n$ 块数据发送给 GPU_{j+1} ，并接收 GPU_{j-1} 的第 $(j-i-1)\%n$ 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 $(i + 1)\%n$ 块数据传递给其它 GPU
- 在第 i 次传递时， GPU_j 把自己的第 $(j - i - 1)\%n$ 块数据发送给自己的右邻居，同时接受左邻居的第 $(j - i - 2)\%n$ 块数据

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

Ring-AllReduce

来自TensorFlow bili

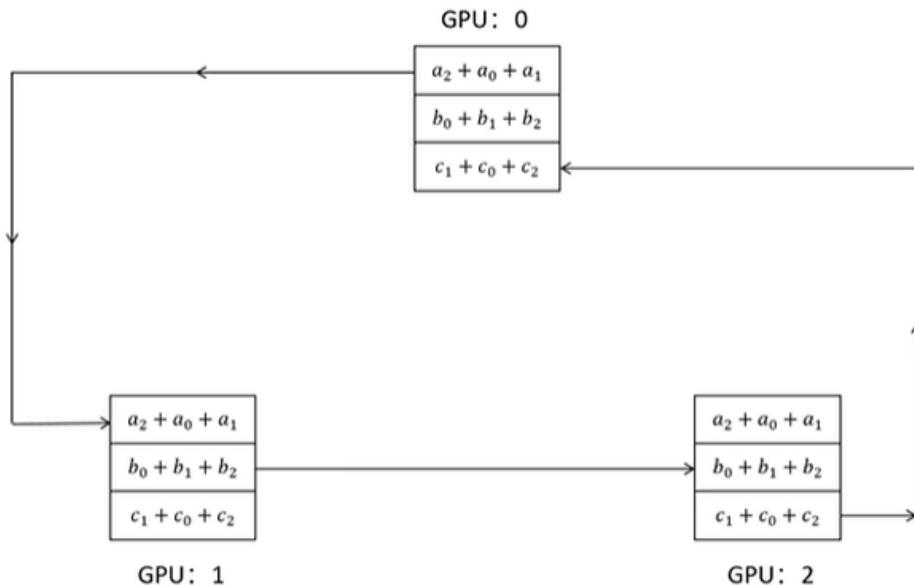


Figure 6: image-20250813101824156

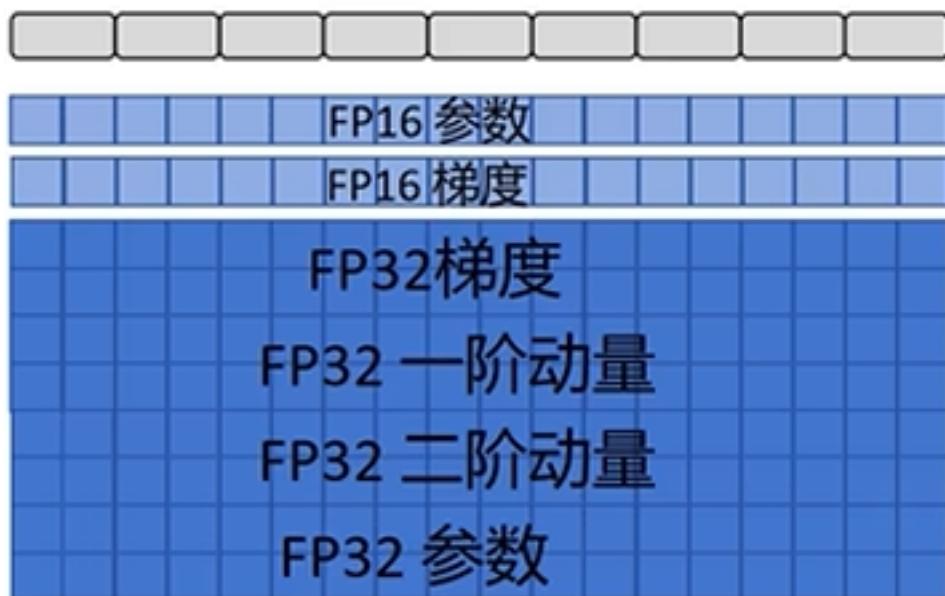


Figure 7: image-20250814170149855

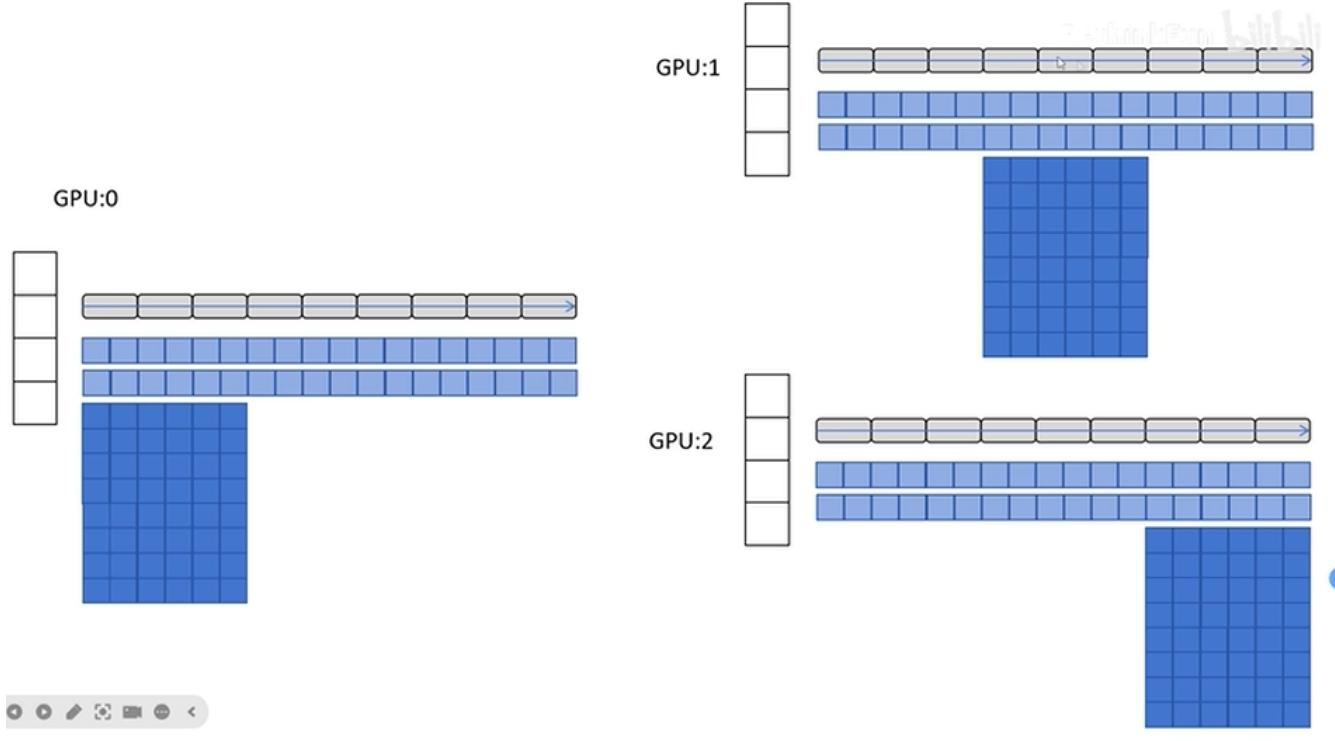


Figure 8: image-20250814170444276

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

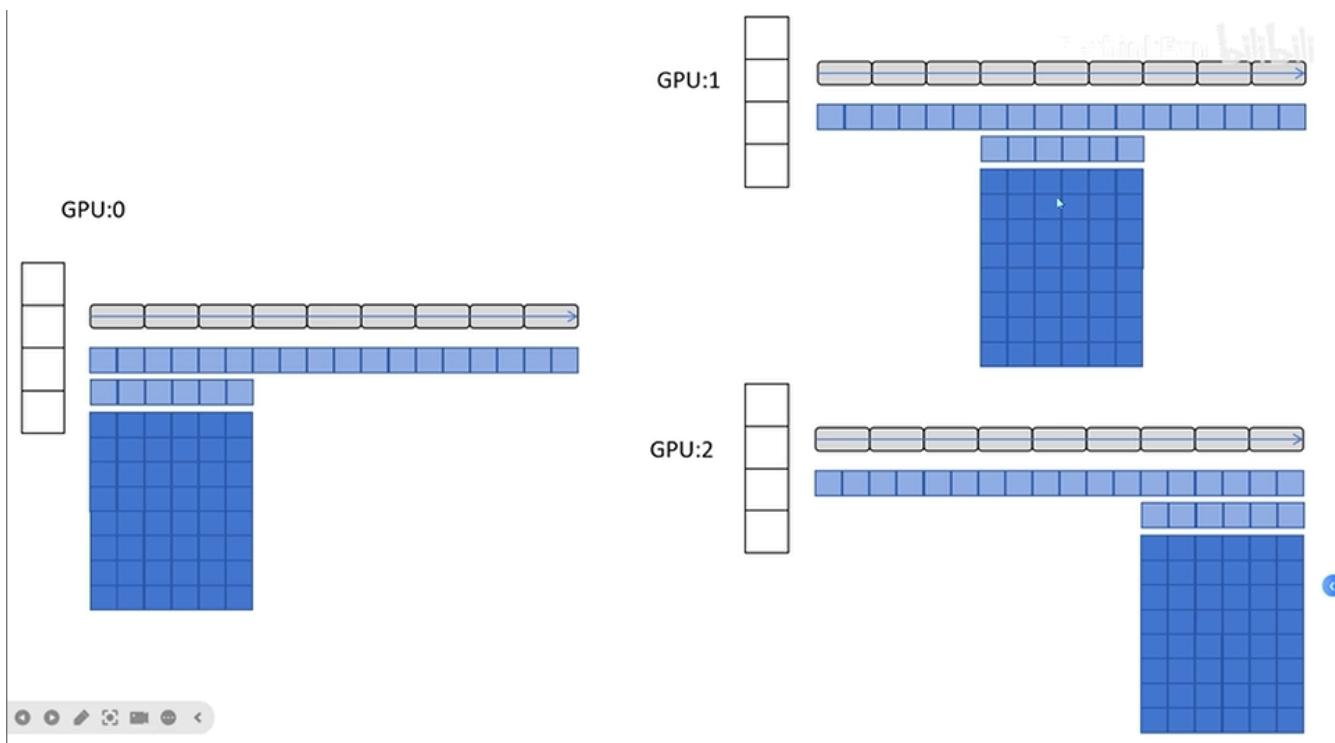


Figure 9: image-20250814171201197

- 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：

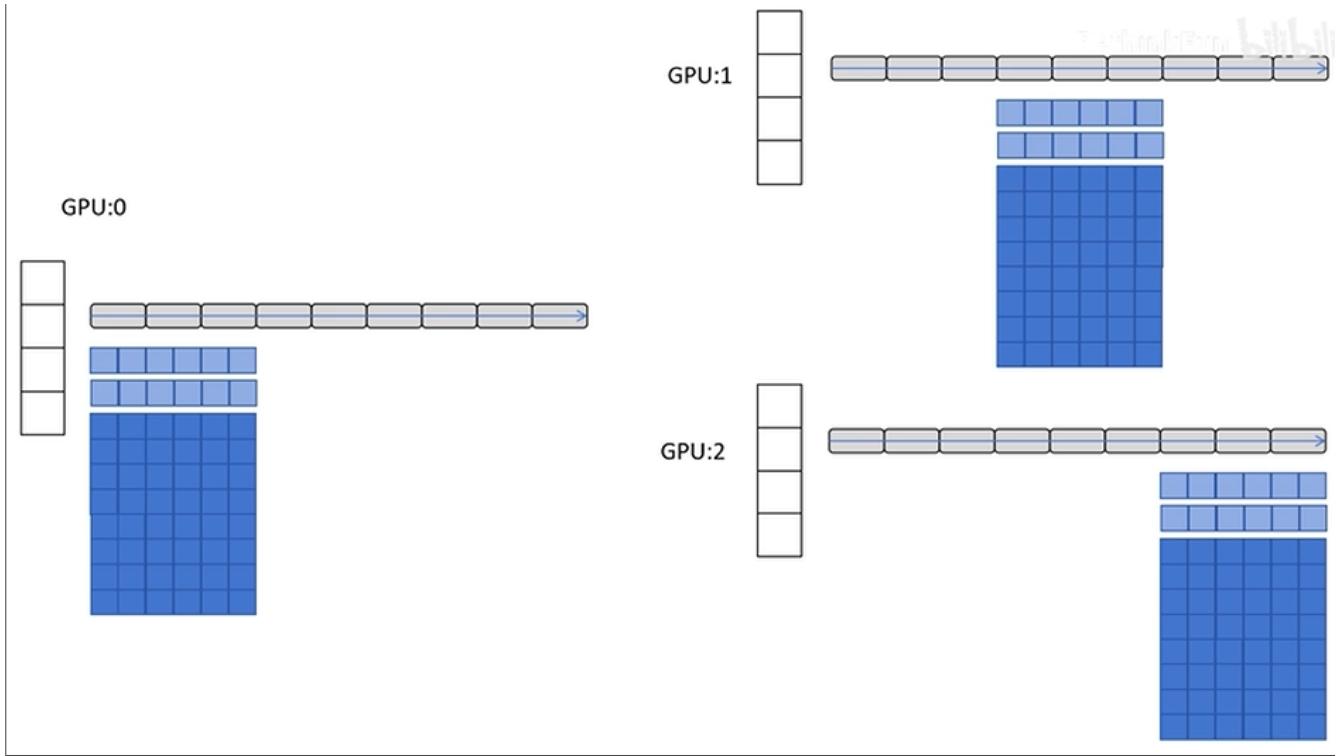


Figure 10: image-20250814172727460

- 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

$$\begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}$$

然后按列进行分割, 分配到两块 GPU 上进行计算

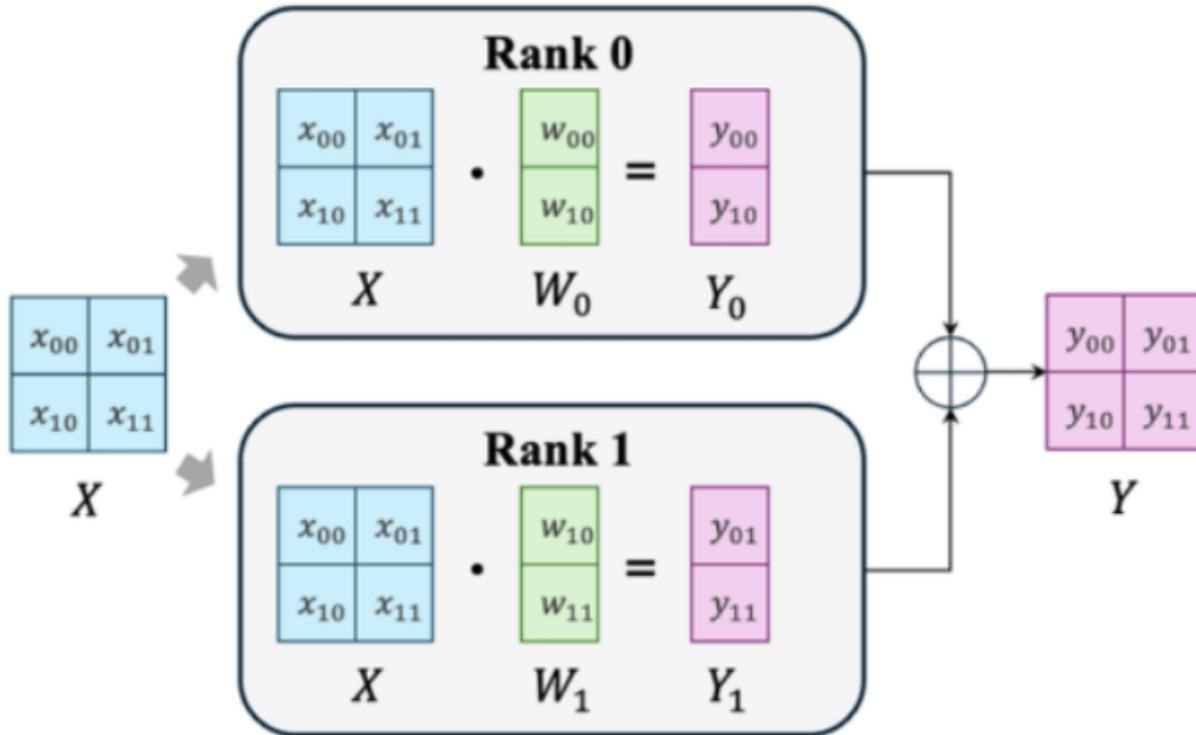
$$W = [W_0 \quad W_1]$$

接着

$$Y_0 = XW_0 = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} w_1 \\ w_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_3 \end{bmatrix} Y_1 = XW_1 = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} w_2 \\ w_4 \end{bmatrix} = \begin{bmatrix} y_2 \\ y_4 \end{bmatrix}$$

然后进行拼接

$$Y = [Y_0 \quad Y_1]$$



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

(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

$$\begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} w_1 & w_2 \\ w_3 & w_4 \end{bmatrix}$$

然后按列进行分割，分配到两块 GPU 上进行计算

$$W = \begin{bmatrix} W_0 \\ W_1 \end{bmatrix}$$

此时 X 也要进行切割

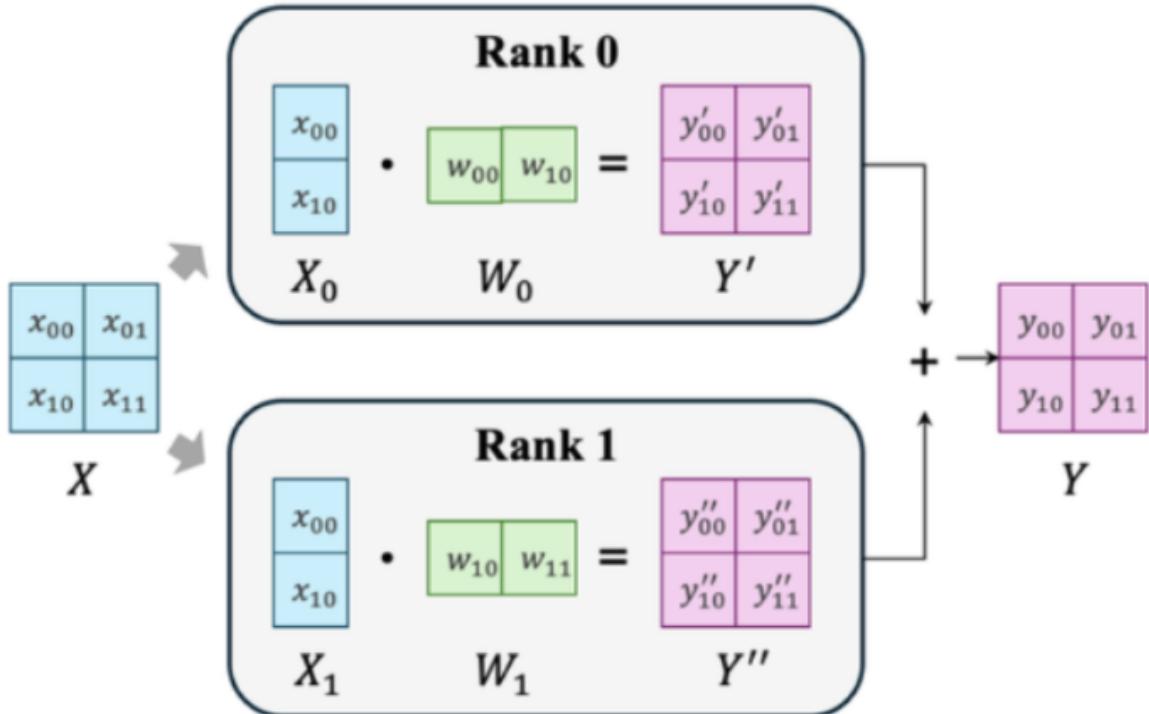
$$X = [X_1 \quad X_2]$$

接着进行计算

$$Y_1 = \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} [w_1 \quad w_2] = \begin{bmatrix} y_{11} & y_{12} \\ y_{31} & y_{32} \end{bmatrix} Y_2 = \begin{bmatrix} x_2 \\ x_4 \end{bmatrix} [w_3 \quad w_4] = \begin{bmatrix} y_{23} & y_{24} \\ y_{43} & y_{44} \end{bmatrix}$$

最后把 Y_1 和 Y_2 进行相加得到最终输出

$$Y = Y_1 + Y_2$$



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

(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**

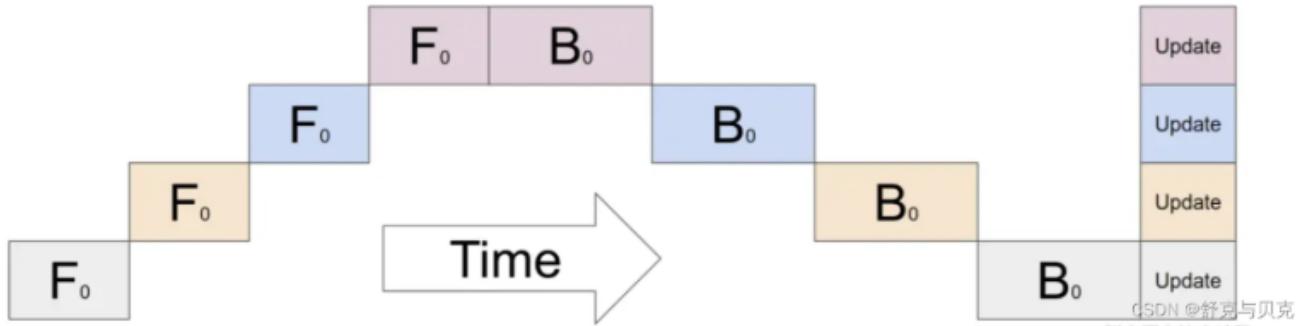


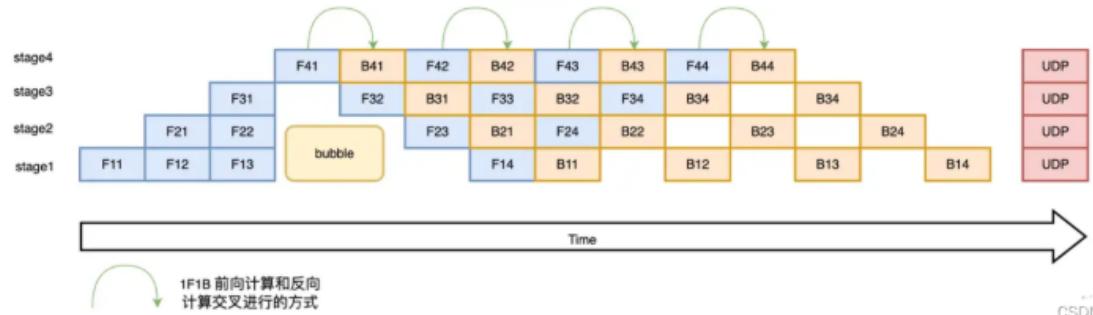
Figure 11: image-20250815223849142

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：

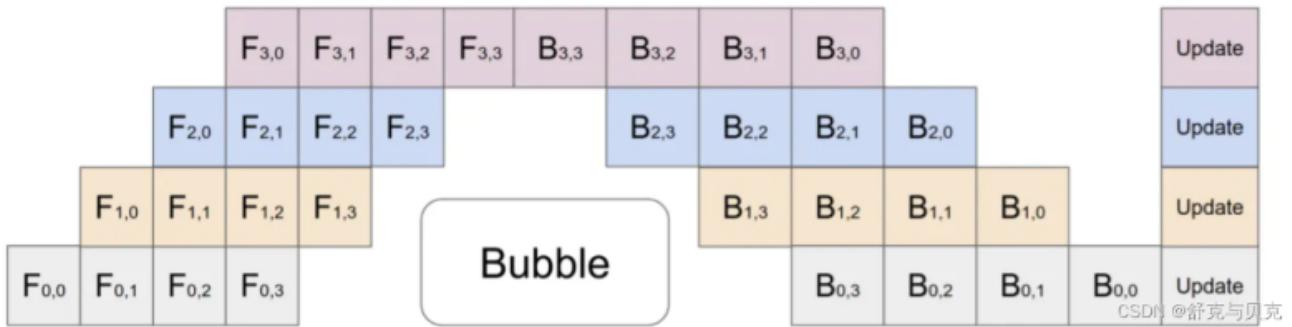


Figure 12: image-20250815224552849

- 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 micro-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

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 $W \boxtimes$ ）
 - 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 $W \boxtimes$ ，而不是最新的权重

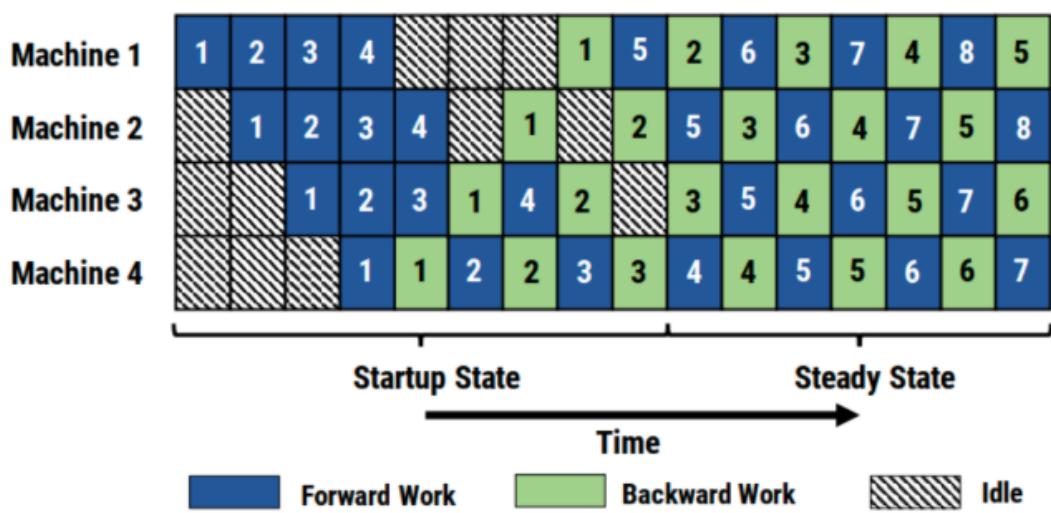


Figure 8: An example pipeline with 4 machines, showing startup and steady states.

CSDN @舒克与贝克

Figure 13: image-20250815235148856

- 3. **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_{\square} 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**:
 1. Micro-batch 进入流水线时, Machine 1 会记录当前最新版本的权重 (如 W_{\square}), 并 将该版本号传递给后续所有 Stage
 2. 所有 Stage 在处理该 Micro-batch 时, 都使用 W_{\square} , 而不是各自的最新权重
 3. 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_{\square} 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)

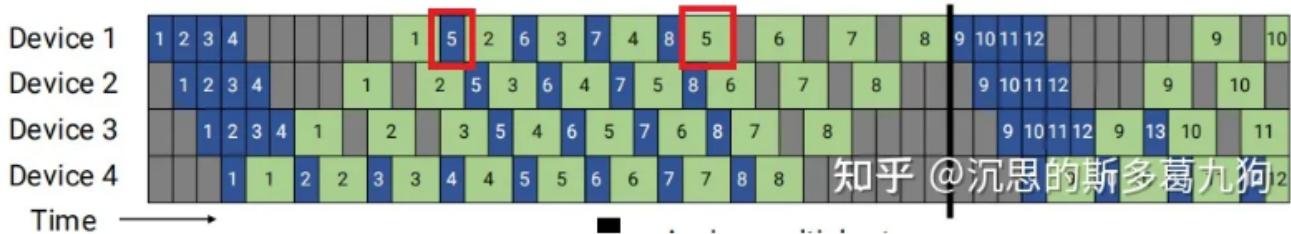


Figure 14: image-20250816005408042

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)

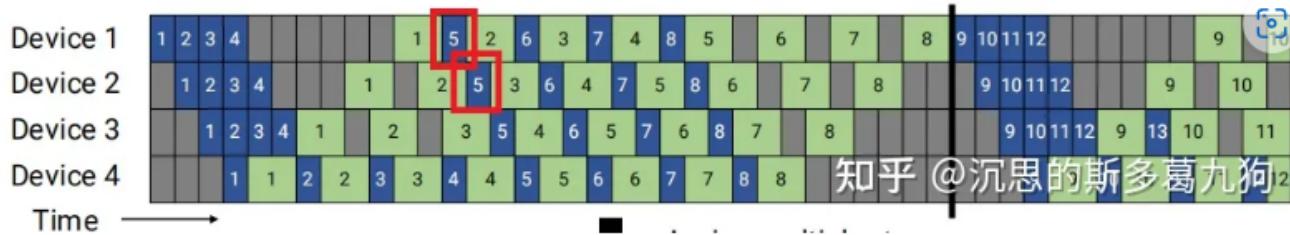


Figure 15: image-20250816005535158

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

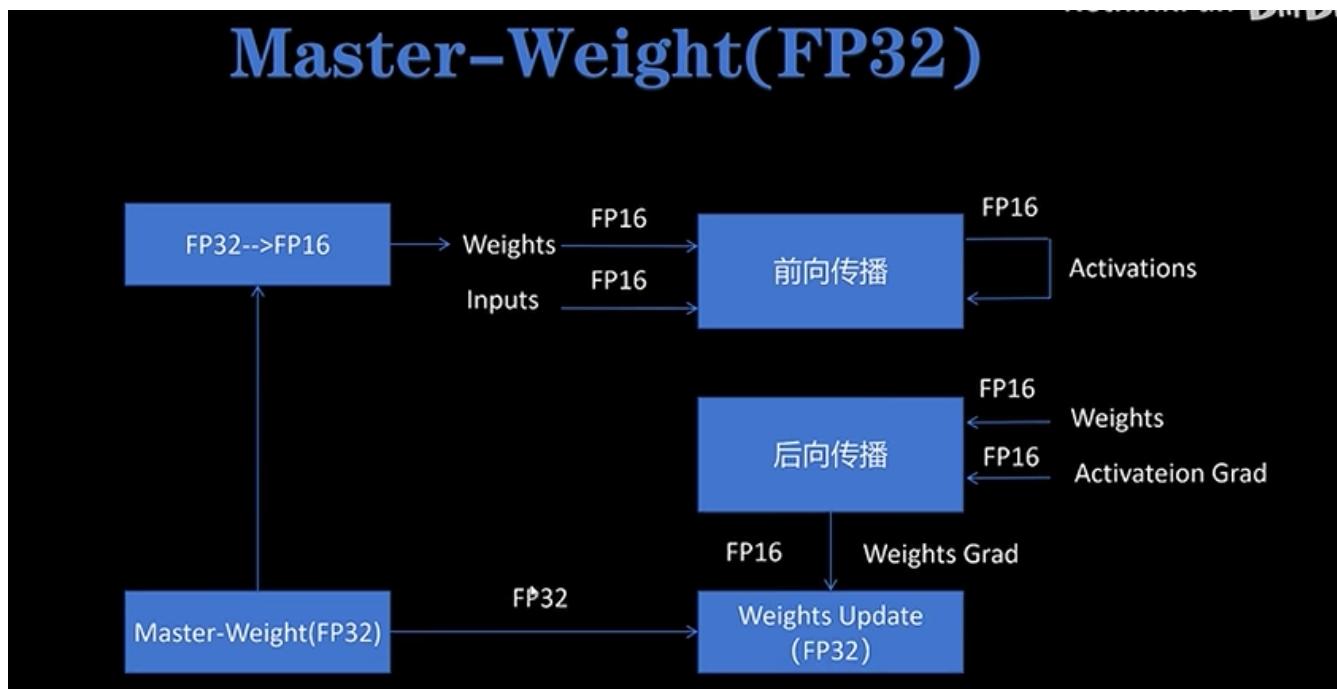


Figure 16: image-20250813120951515

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

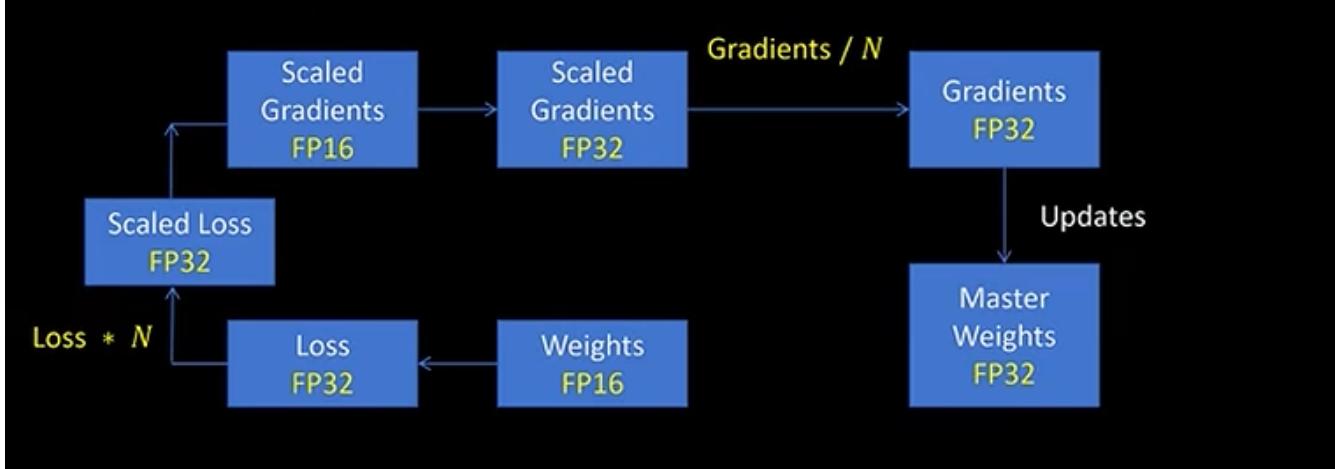


Figure 17: image-20250813121114207

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



Figure 18: image-20250813115950009

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

注意力机制变体

MHA、MQA 和 GQA

不同浮点类型

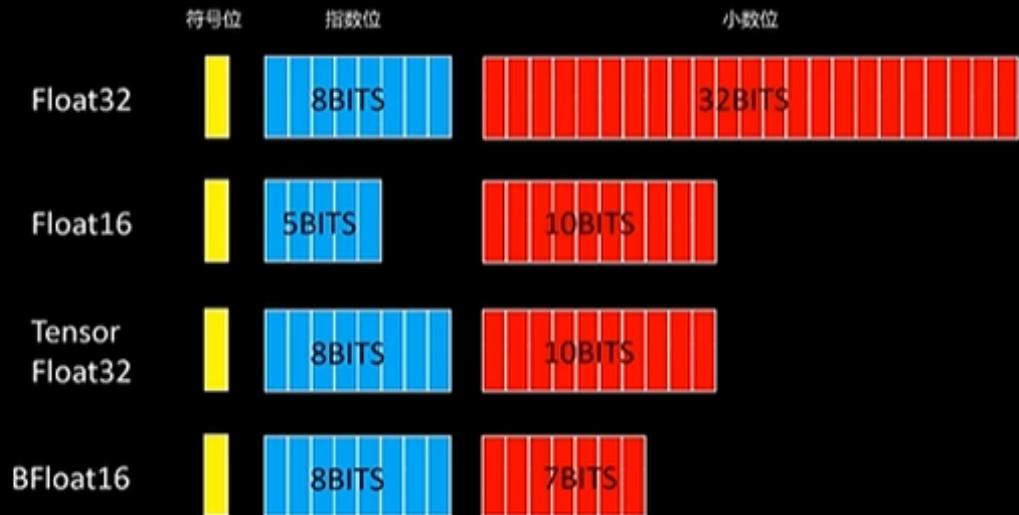


Figure 19: image-20250813115500851

低精度带来的问题

一. 表示范围问题

二. 大数吃小数问题

$$2048 + 0.5 = 2048$$

$$2048 \quad 1.0 * 2^{11}$$

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

Figure 20: image-20250813115927153

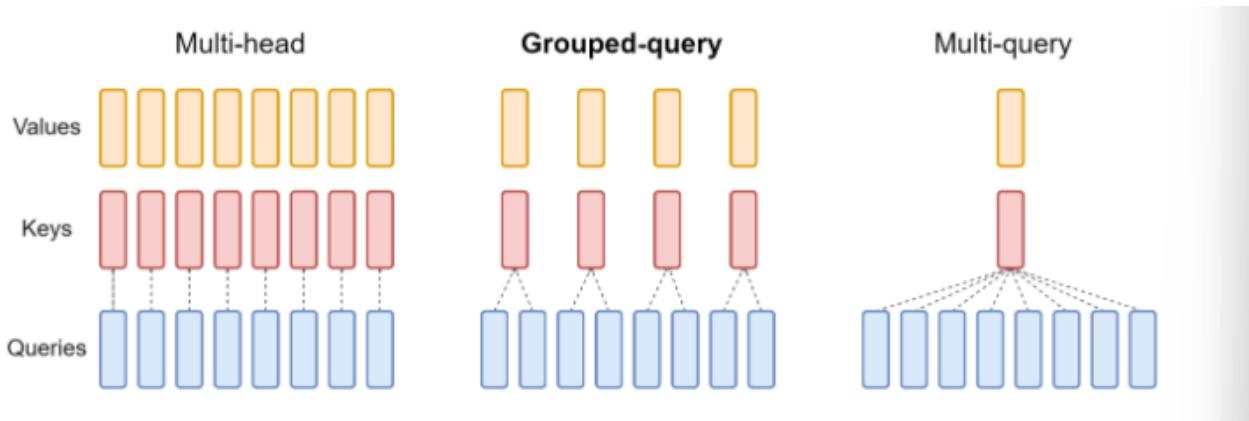


Figure 21: image-20250816120438735

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

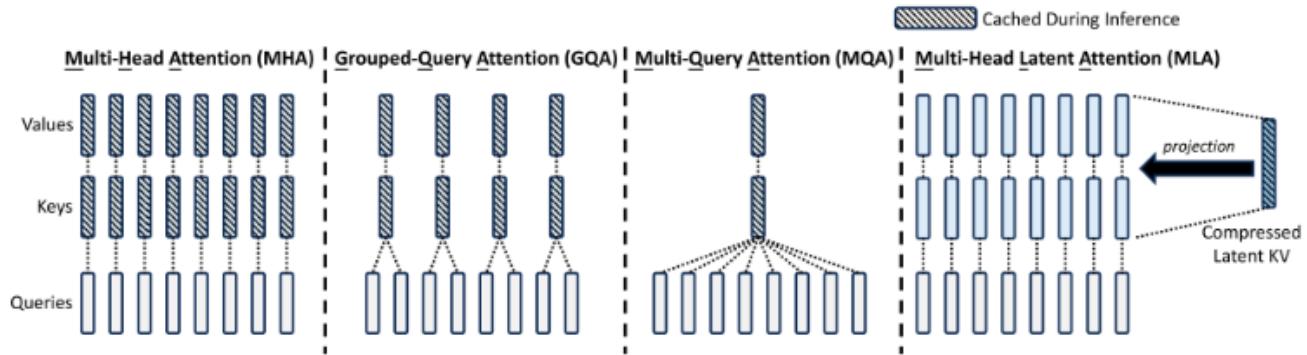


Figure 22: image-20250816120905940

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

$$c_t^{KV} = W^{DKV} h_t k_t^C = W^{UK} c_t^{KV} v_t^C = W^{UV} c_t^{KV}$$

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

Multi-Head Latent Attention (MLA)

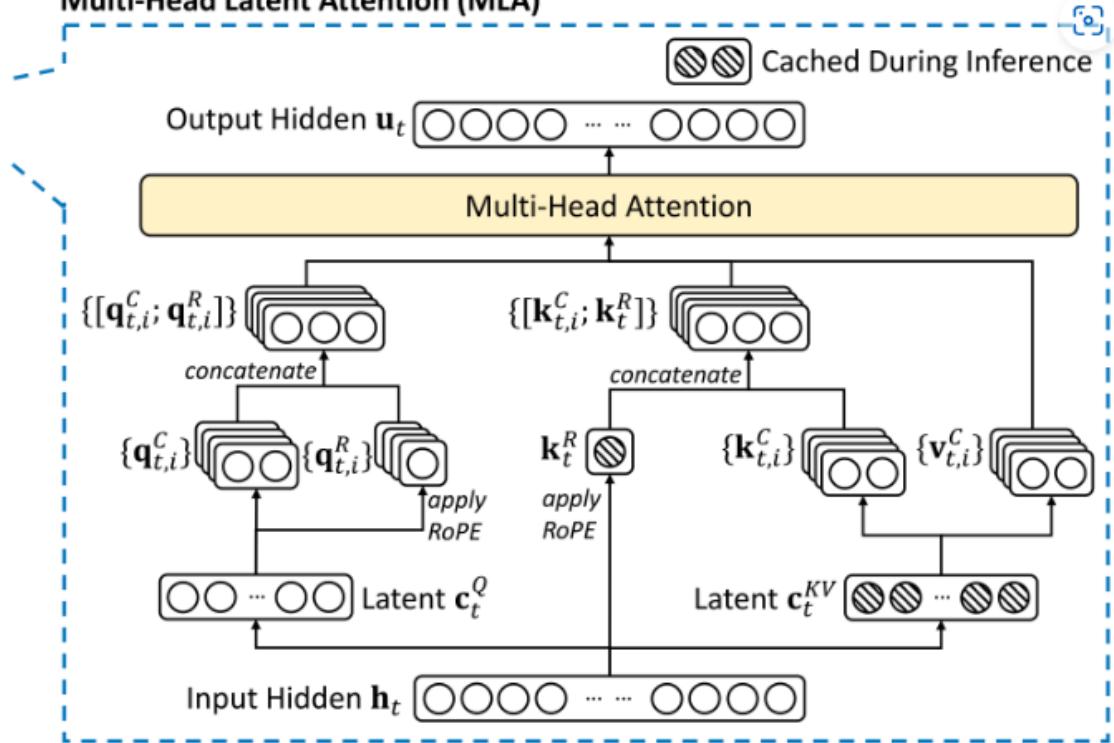


Figure 23: image-20250816131927886

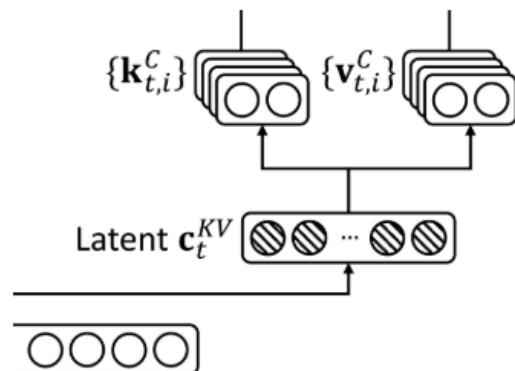


Figure 24: image-20250816133715163

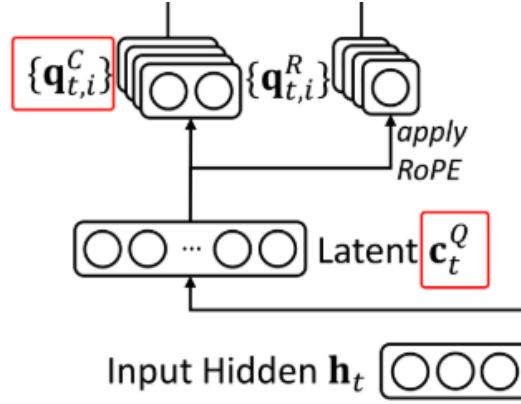


Figure 25: image-20250816135113159

公式如下

$$c_t^Q = W^{DQ} h_t q_t^C = W^{UQ} c_t^Q$$

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

$$S = (W^{UQ})^T (c_t^Q)^T R_m^T R_n c_t^{KV} W^{UK}$$

它**不等于**

$$S = (c_t^Q)^T R_m^T (W^{UQ})^T W^{UK} R_n c_t^{KV} = (c_t^Q)^T R_m^T W_{merge} R_n c_t^{KV}$$

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：显存被多个并发生成序列分割占用，剩余的空闲块尺寸小于新请求所需的最小单位

具体是怎么做的

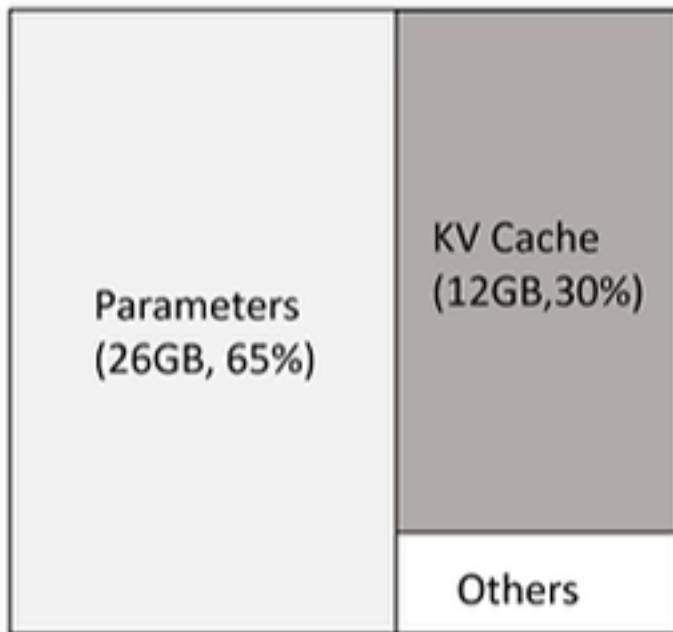
- 分页存储：vLLM 启动时会预先分配一批固定大小的显存页，形成一个空闲页池
- 动态分配显存：每次生成都会预留一页 token 的显存，当生成新 token 的时候检查当前页是否用完，用完了就从空闲页池申请新页
- 逻辑表映射：分配显存的时候，每一页物理上是不连续的，但是 Paged Attention 通过维护一个逻辑页表将物理不连续的显存页映射为逻辑上连续的 KV Cache
- 显存不足：会将不活跃的页换出到 CPU，需要的时候再换回，或者尝试释放已使用完的页
- KV Cache 共享：当 Prompt 相同的时候，Paged Attention 会将相同的 KV Cache 进行共享
 - 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？



13B LLM on A100-40GB

Figure 26: image-20250819121745375

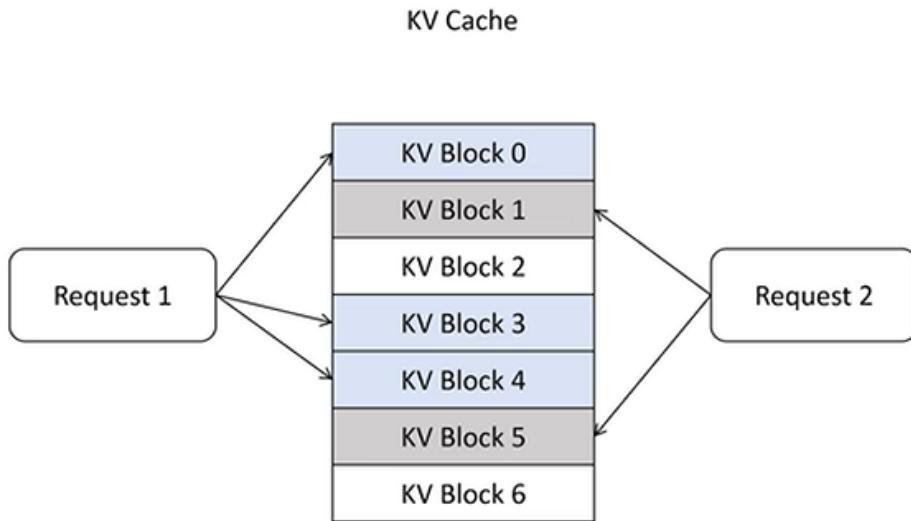


Figure 27: image-20250819121817645

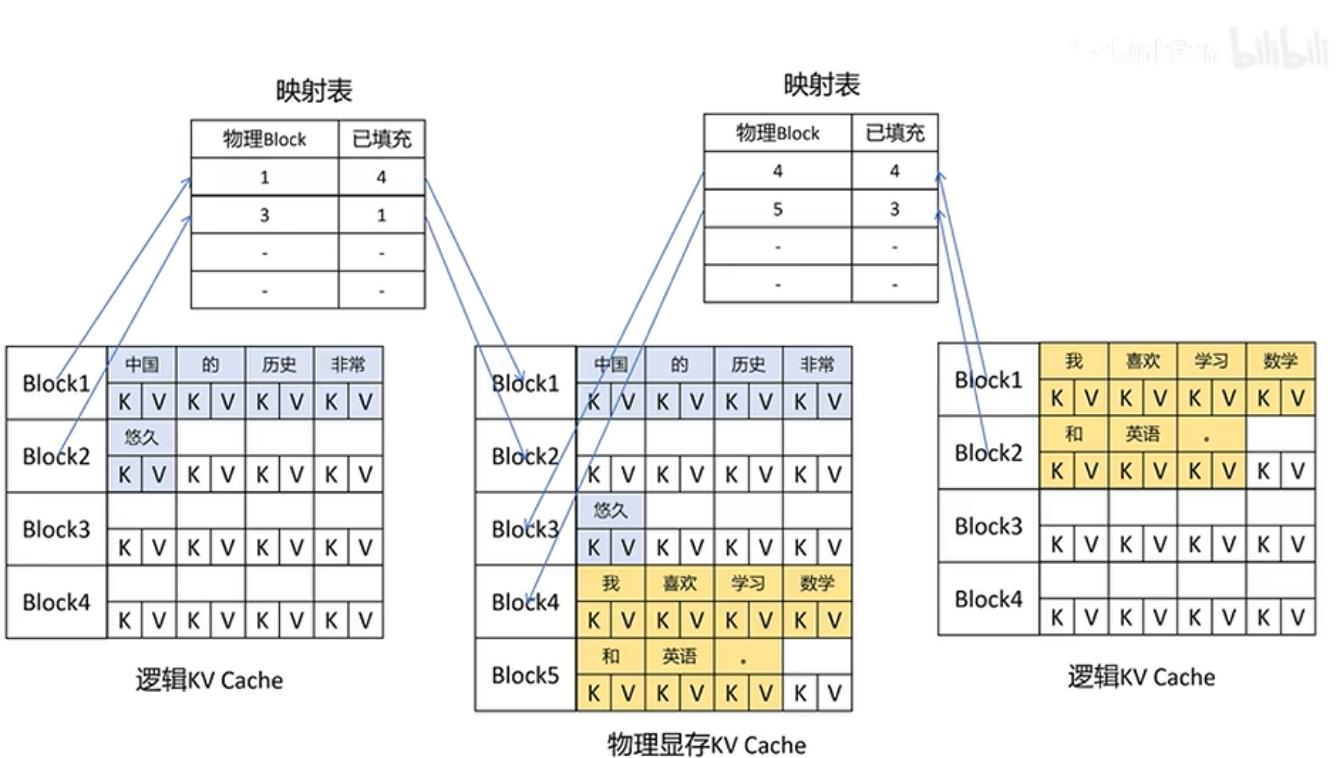


Figure 28: image-20250819121841203

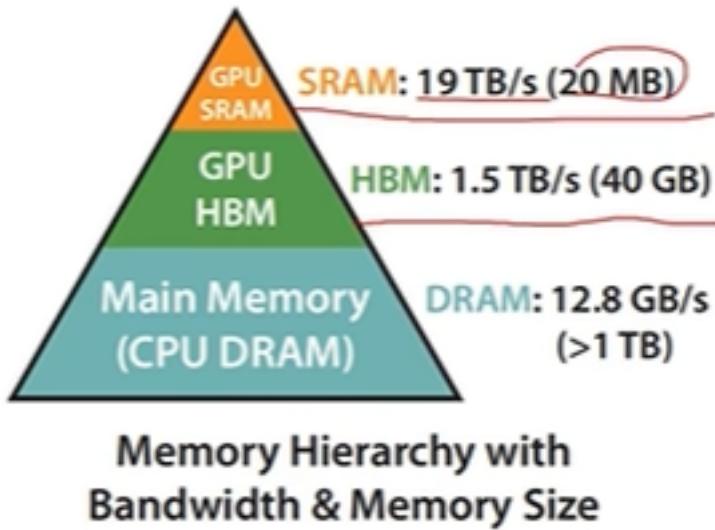


Figure 29: image-20250820134630626

- 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 Kernal 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 $l(x)$

$$f(x) = [e^{m(x_1)-m(x)} f(x_1), e^{m(x_2)-m(x)} f(x_2)] f(x) = [e^{-2}(e^{-1}, e^0), e^0(e^{-1}, e^0)] l(x) = e^{m(x_1)-m(x)} [e^{-1}, e^0] l(x) = e^{m(x_1)-m(x)} [e^{-1}, e^0]$$

- 计算 $O = PV$
- 从 SRAM 中取出 O 回到 HBM

GPU 工作原理

使用 SIMD 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

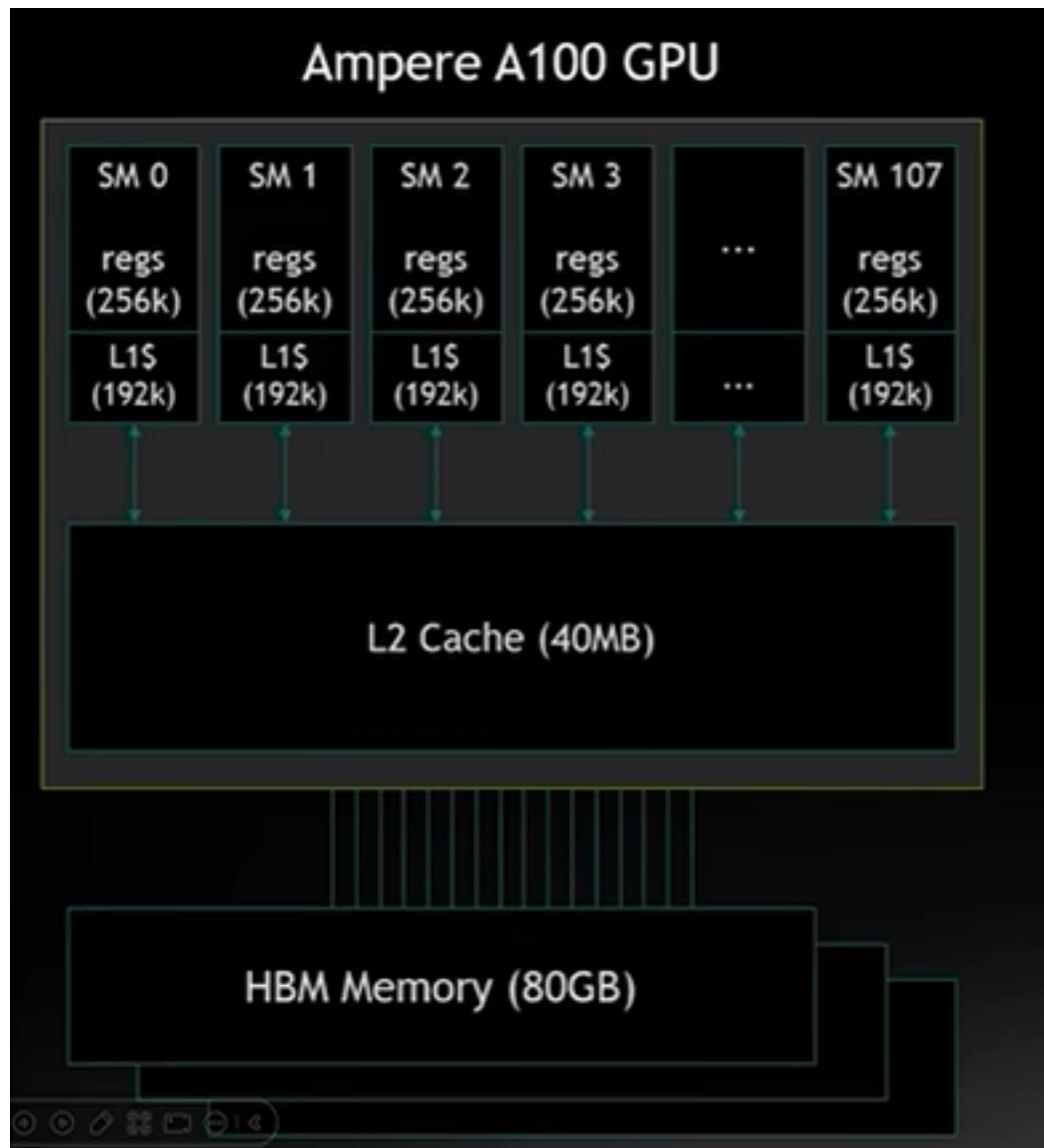


Figure 30: image-20250819141437101

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

Figure 31: image-20250819132629103

- 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