

LLM Inference

基础问题

- 基础问题：

1. 大模型训练和大模型推理的区别

- 大模型训练：需要调整内部参数，训练需要较高的计算性能，海量的数据，计算资源密集
- 大模型推理：无需调整内部参数，但是需要高计算性能（高吞吐率），低功耗

总结对比表

特征	大模型训练	大模型推理
核心目标	学习知识，更新模型参数	应用知识，用固定参数生成结果
计算过程	前向传播 + 反向传播 + 参数更新	只有前向传播
资源需求	极高，大规模GPU集群	相对较低，硬件选择更灵活
时间周期	一次性，漫长（数周/月）	持续性，重复且频繁（毫秒/秒级）
关键指标	损失（Loss）、准确率、收敛性	延迟、吞吐量、成本/每次推理
主要成本	研发成本（一次性投入巨大）	运营成本（长期持续发生）
优化重点	并行计算、稳定性、数据质量	延迟、吞吐、量化、剪枝、KV缓存
类比	上学读书	参加工作/考试

2. Compute Bound & Memory Bound

[知乎的一篇科普](#)

- Compute Bound(计算边界/计算瓶颈)
- Memory Bound (内存边界/内存瓶颈)

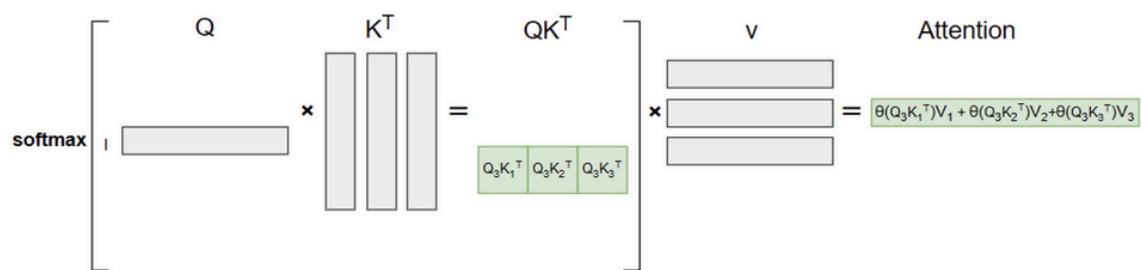
特性	Compute Bound (计算瓶颈)	Memory Bound (内存瓶颈)
瓶颈所在	CPU的计算能力	内存的带宽和延迟
CPU状态	持续高强度运算	经常空闲, 等待数据
提升性能关键	更强大/更多的CPU核心	更快、延迟更低的内存, 更大的CPU缓存
优化方法	优化算法, 并行计算	优化数据访问模式, 提高缓存命中率
典型例子	视频压缩、科学计算	大数据集遍历、数据库操作

3. Prefill & Decode & KV Cache

- KV Cache: 在LLM (transformer架构) 中, 将过去所有token的key和value向量缓存起来, 避免重复计算

KV Cache 机制的核心思想就是: 将过去所有Token的Key和Value向量缓存起来, 避免重复计算

具体来说, 在生成第 i 个Token时, 我们只需要计算当前Token的 Q , K , V 向量。然后, 将新生成的 K 和 V 向量与之前缓存的 **KV Cache** 进行拼接, 形成一个包含了到当前位置为止所有Token的 K 和 V 集合。这样, 在进行注意力计算时, 我们只需要使用当前Token的 Q 向量和更新后的完整 **KV Cache** 即可:



通过 **KV Cache**, 自注意力机制的计算复杂度从 $O(n^2)$ 降低到了 $O(n)$, 其中 n 是序列长度, 极大地提升了推理效率。

引入KV Cache后, LLM生成过程自然被分成以下两个阶段

- Prefill (预填充): 主要是Compute Bound

2.1 Prefill（预填充）阶段

Prefill阶段发生在处理用户输入的提示（Prompt）时。在这个阶段，模型会并行地处理输入Prompt中的所有Token，并为它们计算出相应的K和V向量，然后将这些向量填充到 KV Cache 中。

核心特点：

- **并行计算**：由于Prompt是已知的，模型可以一次性地、并行地对所有Token进行计算。这使得该阶段的计算非常密集，可以充分利用GPU的并行计算能力。
- **计算密集型（Compute-Bound）**：Prefill阶段涉及大量的矩阵乘法运算，其瓶颈主要在于GPU的计算能力。
- **生成第一个Token**：此阶段的最终输出是生成回复的第一个Token，同时填充好初始的 KV Cache。

举个例子，当用户输入“中国的首都是哪里？”时，模型会将这句话分词（Tokenize），然后并行处理“中国”、“的”、“首都”、“是”、“哪里”、“？”这几个Token，计算出它们各自的K和V向量，并将这些向量存入 KV Cache。这个过程的耗时，直接影响了用户感知的“首字延迟”（Time to First Token, TTFT）。

- Decode（解码）：主要是Memory Bound

2.2 Decode（解码）阶段

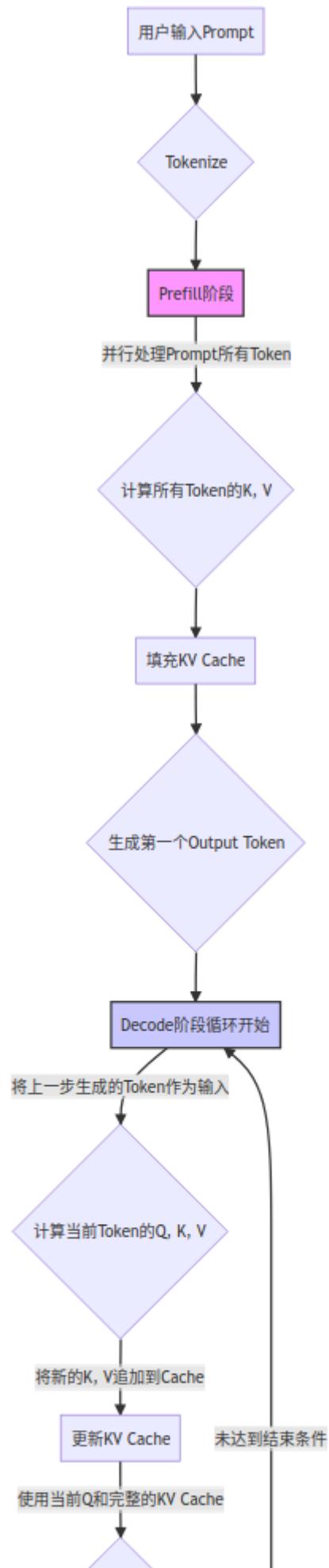
Decode阶段发生在逐个生成输出Token时。当 Prefill 阶段完成后，模型进入自回归（Autoregressive）的生成循环。在每一步中，模型只处理上一步新生成的那个Token。

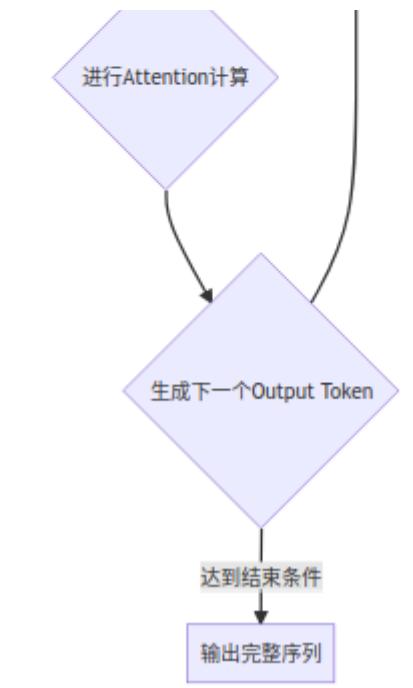
核心特点：

- **串行计算**：每次只处理一个Token，其计算依赖于上一步的结果，因此是串行的、自回归的。
- **内存带宽密集型（Memory-Bandwidth-Bound）**：在 Decode 的每一步，模型都需要从GPU内存中读取庞大的模型权重和完整的 KV Cache。计算量本身不大（主要是一个向量和矩阵的乘法），但数据传输量巨大。因此，其瓶颈在于GPU显存的带宽。
- **利用并更新KV Cache**：每生成一个新Token，都会计算出其K和V向量，并将其追加到 KV Cache 中，供下一个Token的生成使用。这个过程的耗时，影响了“字间延迟”（Time Between Tokens），即生成后续内容的速度。

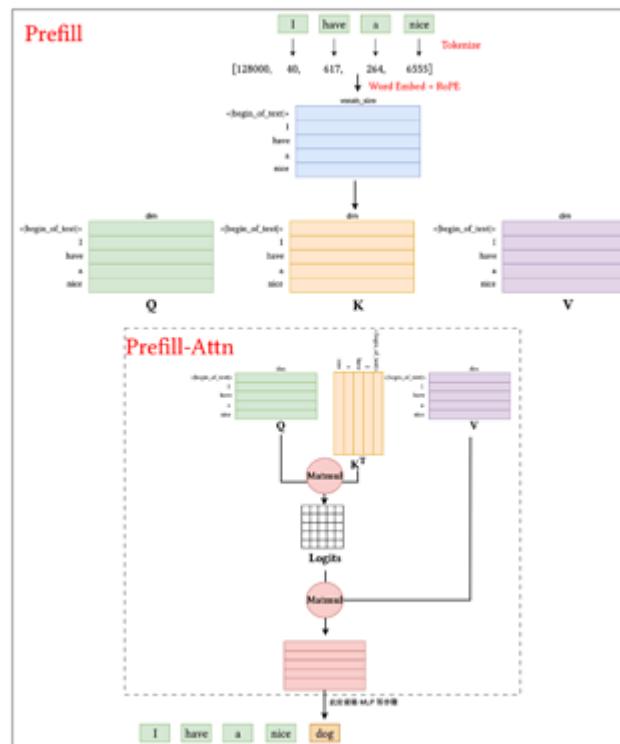
继续上面的例子，在生成第一个Token“北”之后，Decode 阶段开始。模型会计算“北”的K和V，并将其加入到之前由Prompt生成的 KV Cache 中。然后利用“北”的Q和更新后的 KV Cache 来生成下一个Token“京”。如此循环，直到生成完整的答案“北京。”。

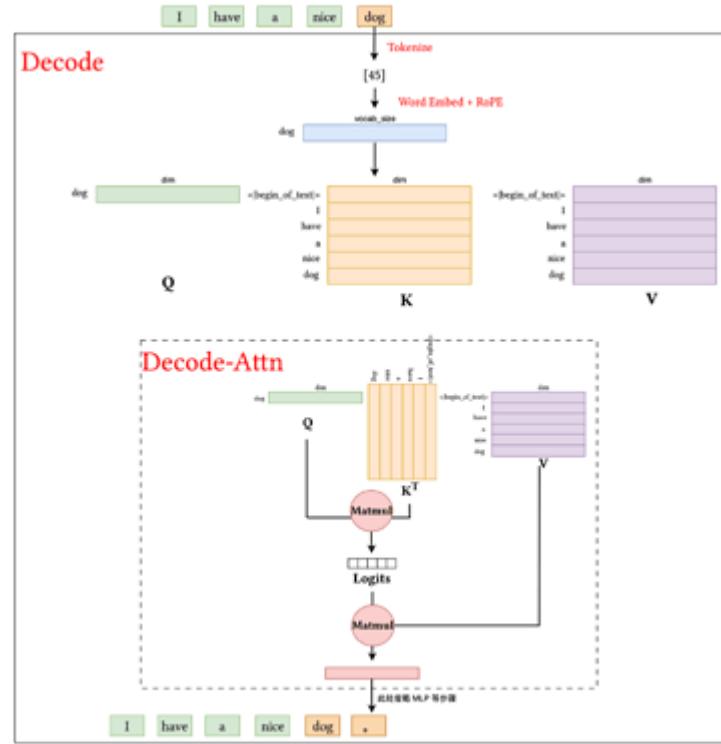
- 协同工作流程





■ 图示流程：





优化KV Cache

- 优化KV Cache

原因：KV Cache引入带来了省略计算的同时，但也增加了内存不足的问题，过大的KV Cache会占用过多的内存。所以如何降低内存占用的同时能够保证高并发请求下稳定运行，成了目前的一个研究方向。

- 过多的KV Cache除了会导致内存/显存的开销外还有什么缺点

- 带宽压力增大：大量KV数据每次在Decode的时候都需要从显存中加载到Cache中，造成带宽利用率很高，变成memory-bound，SM核心空闲等待数据
- 计算效率反而下降：前一步的情况，会造成memory-bound，进而影响计算效率；而且并行度也会降低（虽然计算矩阵这部分计算可以并行，但是管理和调度如此大规模的数据本身会引入开销，可能无法充分利用并行计算资源）
- 限制最大上下文长度：由于显存是有限的，KV Cache的大小直接限制了模型能处理的最大序列长度
- Cache污染：每使用一次KV Cache，都会计算生成新的KV Cache覆盖掉原来的，如果cache采用写回+写分配的原则，KV Cache将一直将cache标记为Dirty，导致其他数据无法正常使用（cache line被标记为dirty），并且增加了内存到cache的开销（因为需要一直写入内存），并且过长的KV会在cache中挤掉其他的数据，进一步造成了cache miss
- 能源效率降低：搬运数据比计算更加耗电，会增加更高的成本

2.3 键值缓存管理中的挑战

如2.2.2节所分析的，重用缓存的键值对使大语言模型能够避免重新计算过去的token，从而在推理过程中显著提高速度。然而，随着序列长度的增加，键值缓存的大小也会成比例增加，给内存带来巨大压力。因此，如何有效地管理这个缓存以加速大语言模型计算，同时又不过度占用空间，成为了一个挑战。

1. 缓存淘汰策略：当缓存达到容量时，确定淘汰哪些项目是一个复杂的问题。像最近最少使用 (LRU) 或最不经常使用 (LFU) 这样流行的策略，并不符合大语言模型的模式，会导致性能不佳。
 2. 内存管理：键值缓存所需的内存随着序列长度和层数线性增长，这可能很快就会超过硬件内存限制，特别是对于长序列。因此，管理不同类型存储硬件（如GPU、CPU或外部内存）之间的协作，成为了一个重大挑战。
 3. 延迟瓶颈：在每个解码步骤访问和更新缓存会引入延迟，特别是对于内存带宽有限的硬件。
 4. 压缩权衡：压缩键值缓存可以减少内存使用，但如果丢失关键信息，可能会降低模型性能。
 5. 动态工作负载：处理动态且不可预测的工作负载（其中访问模式和数据需求频繁变化），需要能够实时响应的自适应缓存策略。
 6. 分布式协调：在分布式键值缓存中，跨多个节点维护协调，以确保一致性、容错性和高效的资源使用，这增加了显著的复杂性。
2. 当前针对KV Cache的稀疏化方法

[A Survey on Large Language Model Acceleration based on KV Cache Management](#)

- Abstract摘要&Introduction介绍&Taxonomy分类法：

LLMs的计算和内存需求，特别是在推理期间，为其实际应用带来了挑战

自回归 (auto-regressive) 弊端：随着输入序列的变长，这种方法的计算和内存需求会随序列长度成二次增长（这就是引入KV Cache的原因）

KV Cache：减少冗余计算，提高内存利用率

本综述将全面概述于**KV Cache**的**token级**，**模型级**，**和系统级优化**

- token级：

KV cache selection键值缓存选择：对最相关的token进行优先级排序和存储

budget allocation预算分配：根据token动态分配内存资源，以确保在有限内存下高效利用缓存

merging合并：组合相似或重叠的键值对来减少冗余

quantization量化：降低缓存键值对的精度来最小化内存占用

low-rank decomposition低秩分解：使用低秩分解技术来减少缓存大小

- 模型级：

Attention Grouping and Sharing注意力组和分享：研究键值的冗余功能，并在Transformer层内或跨层对键值缓存进行分组和共享

Architecture Alteration：设计新的注意力机制或构建外部模块以进行键值优化

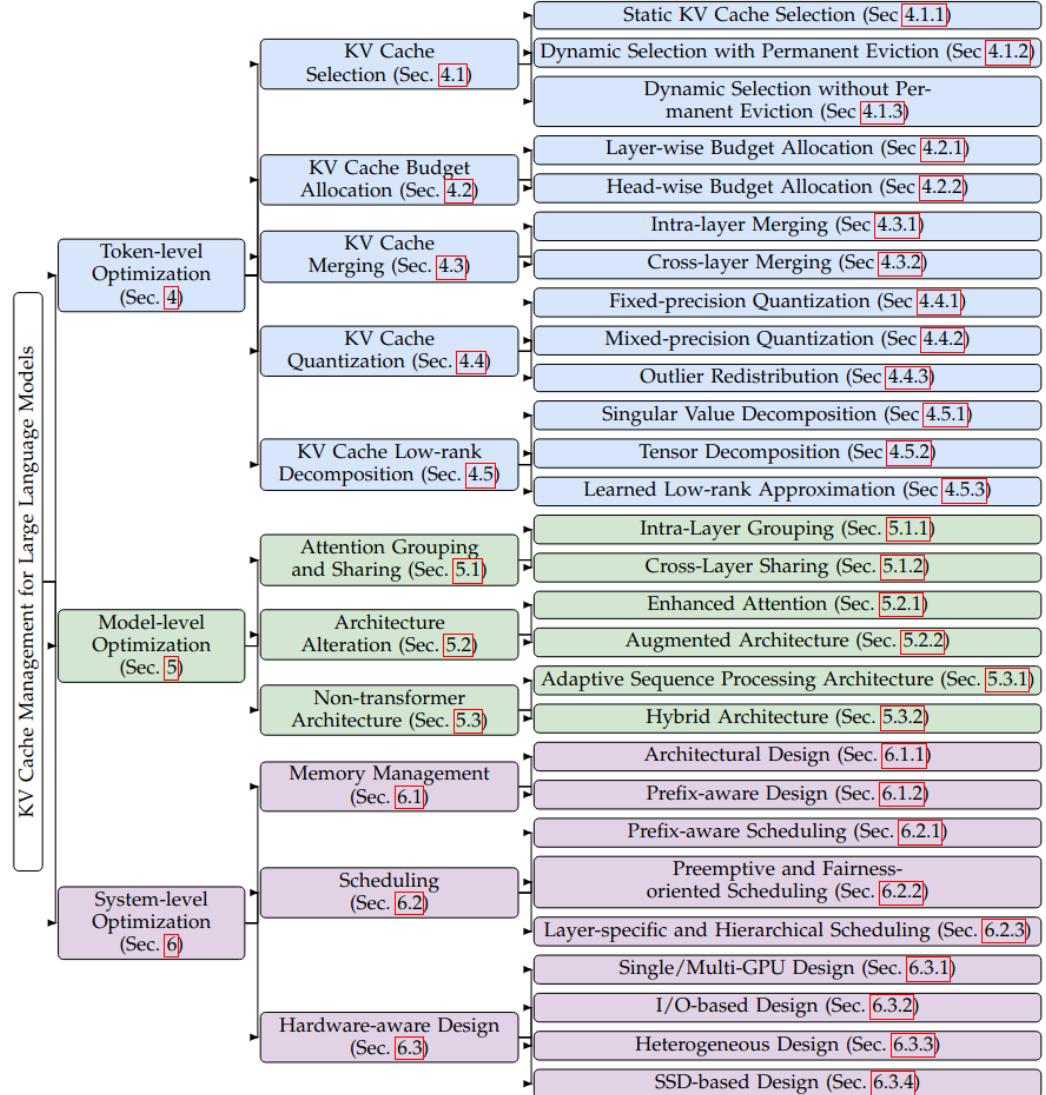
Non-transformer Architecture：用其他内存高效设计（如循环神经网络）来优化传统Transformer中的键值缓存。

- 系统级：

memory management(内存管理): 侧重于架构创新, 如虚拟内存适配, 智能前缀共享, 层感知资源分配

scheduling(调度): 前缀感知方法 (用于最大化缓存重用), 抢占式技术 (用于公平的上下文切换), 特定层机制 (用于细粒度的缓存控制) 来实现多样化的优化目标

hardware-aware designs(硬件感知设计): 单GPU/多GPU设计, 基于I/O的解决方案, 异构计算和基于SSD的解决方案



■ Preliminary(预备知识):

■ Transformer Decoder:

Tokenizer分词->Positional Encoding($X = X + PE(X)$)->Attention(compute Q, K, V)

$$\mathbf{Q}_i = \mathbf{X} \mathbf{W}_{Q_i}, \quad \mathbf{K}_i = \mathbf{X} \mathbf{W}_{K_i}, \quad \mathbf{V}_i = \mathbf{X} \mathbf{W}_{V_i}, \quad (1)$$

$$\mathbf{Z}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \text{Softmax} \left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}} \right) \mathbf{V}_i, \quad (2)$$

通过一次变换连接起来

$$\mathbf{Z} = \text{Concat}(\mathbf{Z}_1, \mathbf{Z}_2, \dots, \mathbf{Z}_h) \mathbf{W}_O, \quad (3)$$

然后再通过线性变换+激活函数+线性变换的FNN前馈层

$$\text{FFN}(\mathbf{Z}) = \sigma(\mathbf{Z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \quad (4)$$

- Auto-regressive Generation自回归生成:

每个Decode都会生成这么一个条件概率

$$P(x_{t+1}|x_1, x_2, \dots, x_t) = \text{Softmax}(\mathbf{h}_t \mathbf{W}_{\text{out}} + \mathbf{b}_{\text{out}}), \quad (5)$$

然后选取一个概率最高的生成下一个token

$$x_{t+1} \sim P(x_{t+1}|x_1, x_2, \dots, x_t). \quad (6)$$

生成的token被附加到token序列之中，这个过程一直持续，直到遇到结束标志符EOS/达到预定义的最大长度

- **Key-Value Cache in Transformer Models:**

计算挑战：自回归机制的每一步都需要重新计算Key&Value，对长序列生成带来了挑战

而KV Cache通过存储先前计算的Key&Value，并在后续token生成中重用它们，从而减少了冗余计算，提高了推理效率

- Auto-regressive Generation with KV Cache

在解码步骤t，新的token经过Positional Encoding->Attention后生成本token的q, k, v

$$\mathbf{q}_i^t = \mathbf{x}_t \mathbf{W}_{Q_i}, \quad \mathbf{k}_i^t = \mathbf{x}_t \mathbf{W}_{K_i}, \quad \mathbf{v}_i^t = \mathbf{x}_t \mathbf{W}_{V_i}, \quad (7)$$

然后新计算的k,v与存储在Cache的Key&Value进行Concat (主要是这一步不同)

$$\mathbf{K}_i^t = \text{Concat}(\hat{\mathbf{K}}_i^{t-1}, \mathbf{k}_i^t), \quad \mathbf{V}_i^t = \text{Concat}(\hat{\mathbf{V}}_i^{t-1}, \mathbf{v}_i^t), \quad (8)$$

然后进行子注意力机制计算

$$\mathbf{z}_i^t = \text{Softmax} \left(\frac{\mathbf{q}_i^t \mathbf{K}_i^{t \top}}{\sqrt{d_k}} \right) \mathbf{V}_i^t, \quad (9)$$

- Time and Space Complexity Analysis时间空间复杂度分析

这里就是将时间复杂度降到O(n)了吧

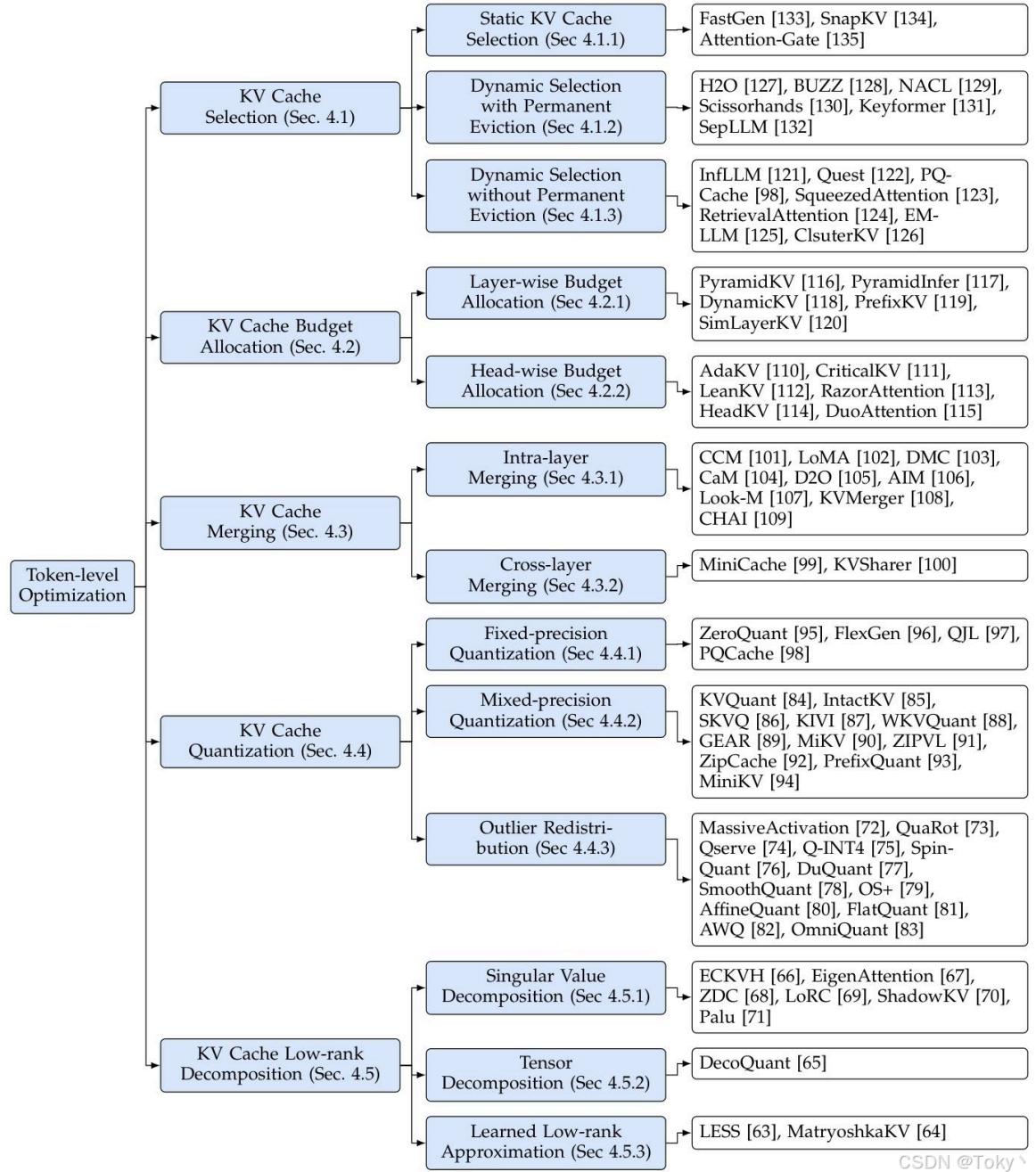
- Challenges in KV Cache Management

1. 缓存淘汰策略：当缓存达到容量时，确定淘汰哪些项目是一个复杂的问题。像最近最少使用 (LRU) 或最不经常使用 (LFU) 这样流行的策略，并不符合大语言模型的模式，会导致性能不佳。
2. 内存管理：键值缓存所需的内存随着序列长度和层数线性增长，这可能很快就会超过硬件内存限制，特别是对于长序列。因此，管理不同类型存储硬件（如GPU、CPU或外部内存）之间的协作，成为了一个重大挑战。
3. 延迟瓶颈：在每个解码步骤访问和更新缓存会引入延迟，特别是对于内存带宽有限的硬件。
4. 压缩权衡：压缩键值缓存可以减少内存使用，但如果丢失关键信息，可能会降低模型性能。
5. 动态工作负载：处理动态且不可预测的工作负载（其中访问模式和数据需求频繁变化），需要能够实时响应的自适应缓存策略。
6. 分布式协调：在分布式键值缓存中，跨多个节点维护协调，以确保一致性、容错性和高效的资源使用，这增加了显著的复杂性。

正片开始!!!

■ Token-Level Optimization

完全基于token键值对的特征和模式，主要受LLM和序列输入的观察结果指导



CSDN @Tokyo

1. KV Cache Selection:

1. static KV cache selection: 仅在预填充阶段进行**token**过滤, 然后进行一次性压缩, 选定的 token 在后续解码步骤中保持固定
2. dynamic KV cache selection: 解码阶段持续更新键值缓存, 实现自适应缓存管理

在动态键值缓存选择方法中, 未被选择的键值缓存 token 可能会被永久淘汰, 或卸载到如 CPU 内存这样的分层缓存设备中, 实现多层次存储策略。鉴于解码过程中的实时键值缓存选择可能会带来巨大的计算开销, 一些研究专注于开发优化的检索算法, 以提高这一过程的效率。这些优化包括以**块级检索代替 token 级粒度**以降低搜索复杂度、采用**异步查询机制**来隐藏延迟, 以及使用并行检索管道来加速选择过程。这些优化工作旨在减轻计算负担, 同时保持 token 选择的有效性。键值缓存选择的总结见表 2。

1. dynamic selection with permanent eviction(永久淘汰)

解码阶段频繁更新 KV cache, 从内存中永久删除未被选择的 KV cache

2. dynamic selection without permanent eviction (无永久淘汰)

无永久淘汰通常采用多层缓存系统（比如CPU-GPU分层缓存），并利用先进数据结构和系统级增强来优化检索效率，从而减少**GPU**缓存占用的情况下实现高效推理

2. Budget Allocation:

LLM的层级架构是异质的（heterogeneous）。不同层、不同注意力头所处理的信息和功能不同，因此其对KV缓存的需求和重要性也完全不同

通过根据每个组件对预测准确性的重要性，智能地分配内存资源，从而在最小化准确性下降的同时优化内存利用率

1. 层级预算分配：在不同Transformer层之间分配内存

2. 头级预算分配：在每一层内的各个注意力头之间进行精确的内存分配

3. Merging:

识别并利用KV缓存中的冗余信息，将多个Token的键值对压缩或整合成一个更具代表性的向量，从而直接减少需要存储的向量数量

1. 层内合并：单个transformer层内进行操作

2. 跨层合并：不同transformer层间KV Cache存在相似性

4. Quantization:

降低每个K/V向量数值的表示精度，从而减少每个token占用的内存空间

1. 固定精度量化：对所有K/V采用统一的相同比特宽度的量化

2. 混合精度量化：为关键token和部分K/V分配更高或者全精度，对不太关键的部分使用较低精度

3. 异常值重新分布：将异常值重新分布到新添加的虚拟token中，或者利用等效变换函数来平滑K/V，以提高量化精度

5. Low-rank Decomposition:

KV矩阵具有低秩性，这意味着虽然矩阵很大，但其包含的信息实际上可以用一个维度低很多的矩阵表示

1. 奇异值分解：利用SVD将其分解为三个更简单的矩阵，只保留几个最大的奇异值和其对应的向量

2. 张量分解：利用张量分解技术将矩阵分解成一系列小型局部张量的乘积

3. 学习低秩近似：训练一个小的神经网络来学会如何高效生成低秩KV缓存近似

■ Model-Level Optimization

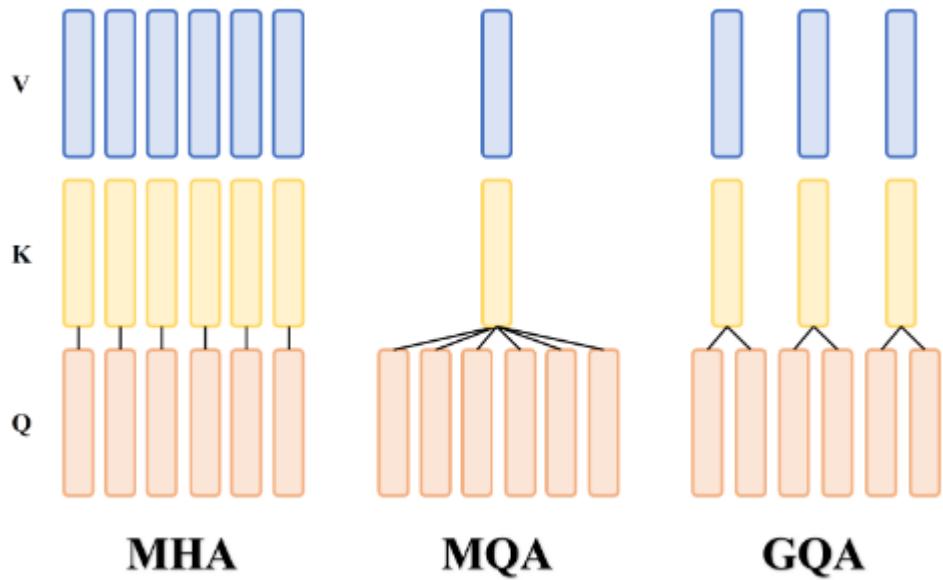
1. 注意力分组和共享

■ 层内分组：专注于单个层内对Q, K, V进行分组，以减少冗余提高效率

MQA (Multi-Query Attention): 所有注意力头共享单个Key和Value--极大加速Decode过程，但会导致质量下降，还会导致训练不稳定

GQA(Grouped Query Attention): 将查询头分成多个组，每个组共享自己的Key和Value--性能上接近MHA，同时提供与MQA相当的推理时间

向上训练：通过与每个组相关连的Key和Value头进行平均池化，将MHA转换为GQA



- 跨层共享: 跨层共享K, V或注意力组件, 以提高信息重用并减少键值缓存需求

这三个是针对层和头来说的

CLA(Cross Layer Attention):在相邻层之间共享Key和Value头, 进一步减少了KV Cache中的冗余

LCKV: 仅为一小部分层 (甚至仅顶层) 计算和缓存Key和Value, 然后让底层的Query和保存的Key和Value配对进行推理

MLKV (Multi-Layer Key-Value): 层内和MQA一样, 多个层之间也这么共享这个键值头
这两个是针对层和权重来说的

SA (Shared Attention): 跨多个层重用计算得到的注意力权重, 而不是每一层重新计算

LISA (Lightweight Substisute for Attention) :引入微小的前馈网络来对齐层间的注意力头, 并使用低秩矩阵来近似层间注意力权重的变化 (而不是像上面一样直接共享)

WU等人的研究成果: 将KVCache减少2倍可以在不显著损失准确性的基础上提高吞吐量, 进一步减少需要采用替代设计并增加训练成本

CLLA (Cross-Layer Latent Attention) : 综合框架, 并整合了注意力头大小, 维度缩减, 跨层缓存共享, 量化等手段

DHA (Decoupled Head Attention): 通过分离不同层次的Key Heads 和 Value Heads, 根据实际需要自适应配置各层头的数量, 减少了冗余

2. 架构更改

1. 改进注意力机制--增强注意力

比如用低秩, 压缩, 近似的表示来替代完整的KV

2. 引入结构变化--增强架构

通过引入新的组件/层级来管理上下文信息, 从而避免生成庞大的KV Cache

核心还是注意力机制, 与下面引入新的核心计算单元相区分

3. 非transformer架构

这一部分的思想就是摆脱在transformer架构上的优化，转而使用一种新架构

1. 自适应序列处理架构--使用全新的完全不同的架构

比如RWKV，结合RNN和Transformer优点的架构

2. 混合架构--在transformer上面替换某个模块，融入其他非transformer的模块

- System-Level Optimization

1. 内存管理

1. 架构设计：采用经典操作系统原理来创建灵活的动态内存分配系统

将KV Cache从一大块连续内存分页为多个固定大小的块，并维护一个逻辑块到物理块的映射表。

2. 前缀感知设计：通过组织数据结构实现高效的缓存去重和共享公共前缀

识别不同请求中的相同前缀（如相同的系统提示词），在内存中只存储一份KV Cache，并通过前缀树（Trie）等数据结构让多个请求共享它。

2. 调度策略--调度器方面

1. 前缀感知调度：故意将有相同前缀的请求安排在一起处理，这样共享统一份KV Cache

2. 抢占式和公平导向调度：允许中断当前的回答并保存目前KV Cache的状态，转而去处理一个短回答

3. 层特定和分层调度：不同层的KV Cache的价值不同因而可以差异化管理

3. 硬件感知设计

1. 单/多GPU设计：比如将prefill和decode两个阶段分离到不同的GPU上执行

2. 基于I/O的设计：当GPU放不下所有KV Cache的时候，将部分cache换出到CPU内存/重新计算

3. 异构设计：让CPU和GPU协同工作

4. 基于固态硬盘SSD的设计：将固态硬盘纳入内存层次结构，作为KV Cache的存储空间

3. 什么是“attention sink”现象，论文中是怎么处理的？--这里属于上面KV Cache优化方法中token级优化的
KV Cache Selection---dynamic selection（永久淘汰）

Efficient Streaming Language Models with Attention Sinks

- 背景：

1. KV Cache的存储开销很大--下面滚动KV（只保留最近的解决了这个问题）

2. 大多数LLM只在训练的时候接触有限长度的序列，导致在遇到更长序列的时候泛化能力很差，造成性能下降--（下面设定初始sink token解决了这个问题）

■ Attention Sink：初始tokens获得了异常高的注意力分数，即使他们在语义上并不重要，这导致了初始tokens很大程度上影响了window attention的性能表现

■ 为什么会有attention sink这种现象？

因为在后续token生成中，每一个token都能注意到初始token并利用他，这导致初始token的比重很大，也就是造成了注意力分数很大，而这个又会影响softmax的计算

$$\text{SoftMax}(x)_i = \frac{e^{x_i}}{e^{x_1} + \sum_{j=2}^N e^{x_j}}, \quad x_1 \gg x_j, j \in 2, \dots, N$$

如果去掉初始token，也就会使计算的分母大大减小，使概率的分布发生显著变化，导致推理偏离预期

- Streaming LLM：一个高效的框架，它使得使用有限长度注意力窗口训练的LLM能够泛化到无限序列长度，而无需任何微调

1. 引入可学习的占位符token：在预训练阶段添加一个专用的可学习token，作为attention sink，进一步提升流式部署性能

- 引入的目的：无需多个初始token来卸载过多的注意力分数（为什么模型关注多个初始标记：SoftMax函数的性质使得所有经过attention结构的激活张量不能全部为零，虽然有些位置其实不需要给啥注意力。因此，模型倾向于将不必要的注意力值转嫁给特定的token，作者发现就是**initial tokens**。）而引入了这样一个token（原文其实是引入四个），就可以作为不必要的注意力分数存储库

- 另一种方法（如果不引入这样一个token的话）：换softmax
这种方法不需要所有上下文token的得分总和为1
这种方法等价于把一个Key和Value全为0的token加到前面
虽然这种方法有一定的缓解，但这种方法仍然依赖其他token作为**attention sinks**

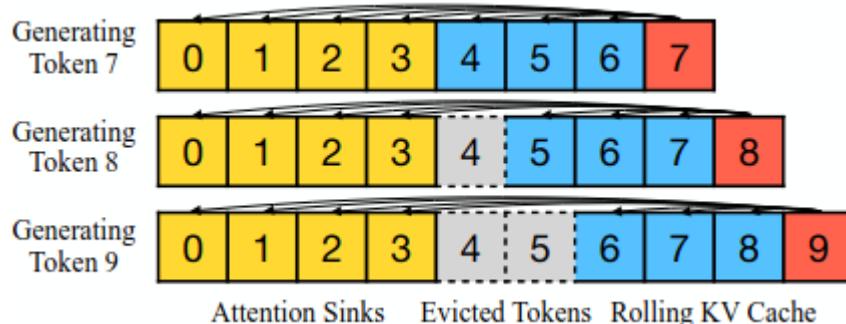
$$\text{SoftMax}_1(x)_i = \frac{e^{x_i}}{1 + \sum_{j=1}^N e^{x_j}}, \quad (2)$$

2. 将**attention sink**（只需4个初始token）与滑动窗口的键值对（这就是实现方法）一起保存，即可锚定注意力计算并稳定模型性能

- Rolling KV Cache with Attention Sinks--对应上面的第三条

KVCache包括两部分：attention sink（4个）&滑动窗口保留的KV Cache（即最近的token）

关于位置：Streaming LLM关注的是缓存中的位置，而不是原文中的位置，如下图为原文中的位置，而分配的位置为[0,1,2,3,4,5,6,7]（即：按照缓存中的相对位置进行位置分配）



比如应用RoPE，是在每轮滑动窗口更新完之后，在缓存中确定了相对位置之后，再对其进行位置变换（位置编码）

4. DeepSeek --NSA--稀疏化Attention：只计算部分关键token的注意力（核心1），从而显著降低计算和显存开销--模型级优化

算法+代码

[Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention--硬件友好且支持原生训练的稀疏化Attention](#)

[中文版--知乎](#)

- 背景：长文本建模成为下一代LLM的关键能力，但是原始注意力机制的高计算复杂度随着序列长度的增加而显著提升，成为显著瓶颈--稀疏注意力机制可以有效解决这个问题

- 目前稀疏注意力机制的局限性：

大多数方法仅在推理阶段应用稀疏性，同时保留预训练的全注意力骨干网络，这种方法可能引入架构偏差，限制了充分发挥稀疏注意力优势的潜力

尽管注意力计算中实现了稀疏性，但许多未能降低推理延迟，这是因为

- 阶段不均衡的稀疏性：这种现象就是上面说的没有全阶段使用稀疏性，至少有一个阶段使用的还是全注意力，这种降低了在不同场景下的加速效果（因为不同场景所要求的阶段不同）--

DeepSeek解决方法（核心2）：Natively Trainable--训练过程中就可以引入

这里解释一下为什么在训练过程就可以引入：因为它这种稀疏不是建立在一个密集计算上的技巧（比如掩码），而是一个基础算子，框架知道如何对它进行前向和反向计算

也就是说因为它的算子是固定且稳定的，模型训练的时候可以直接推导出稳定的反向传播过程，也就实现了稳定的训练过程

- 与先进注意力架构的兼容性问题：部分稀疏注意力方法难以适配现代高效解码架构（比如MQA, GQA），因为虽降低了计算量，但是其分散的内存访问模式与先进架构的高效内存访问设计相互制约

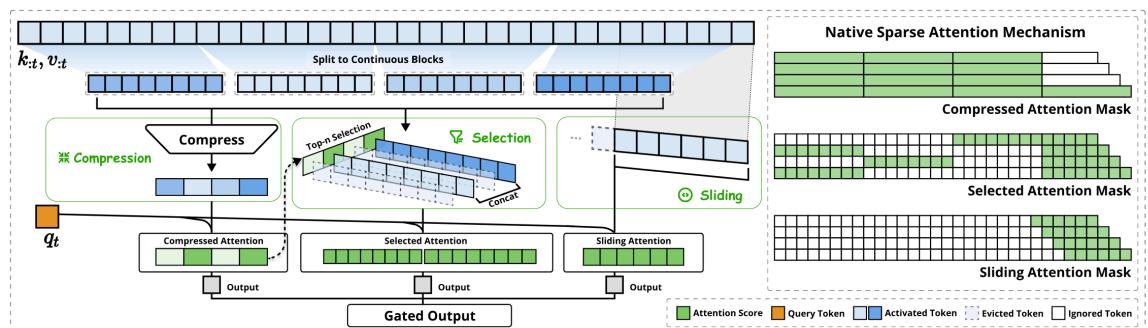
所以这个要求我们开发新的算法，将先进架构设计与硬件高效结合，充分发挥稀疏性在提升模型效率方面的潜力--Deepseek解决方法（核心3）：Hardware-Aligned（即在底层算子上进行优化）

- 有效部署稀疏注意力机制需要面临的挑战：

1. **硬件友好的推理加速**：在预填充和解码阶段采用硬件优化的算法设计，优化内存访问模式和硬件调度机制，将理论计算优化转化为实际性能提升；
2. **训练感知的算法设计**：引入可训练算子支持端到端计算，在降低训练成本的同时维持模型性能。这些要求对于实现实用化的快速长文本推理和训练至关重要。从这两个维度考量，现有方法仍存在显著优化空间。

- 本文提出NSA(Native Sparse Attention)--一种集成分层token建模的原生可训练系数注意力架构

分层token：NSA通过将键值对组织为时间块，并通过三条注意力路径处理以减少查询计算量：压缩的粗粒度token、选择性保留的细粒度token以及用于局部上下文信息的滑动窗口。同时开发了专用计算内核以最大化实际效率



- 创新点：

1. 硬件对齐系统：优化分块稀疏注意力以提升Tensor Core利用率和内存访问效率，实现算术强度的均衡

2. 训练感知设计：通过高效算法和反向传播算子实现稳定的端到端训练，使NSA能够同时支持高效的部署和端到端训练
- 可训练稀疏性的迷思和原生稀疏架构的必要性

正片开始！！！

- 背景（下面这两部分其实上面都有涉及）：

- 注意力机制

注意力机制 在语言建模中，注意力机制被广泛应用。每个查询token q_t 需要计算与所有历史键 $k_{:t}$ 的相关性分数，以生成值 $v_{:t}$ 的加权和。对于长度为 t 的输入序列，注意力操作形式化定义为：

$$\mathbf{o}_t = \text{Attn}(\mathbf{q}_t, \mathbf{k}_{:t}, \mathbf{v}_{:t}) \quad (1)$$

其中 Attn 表示注意力函数：

$$\text{Attn}(\mathbf{q}_t, \mathbf{k}_{:t}, \mathbf{v}_{:t}) = \sum_{i=1}^t \frac{\alpha_{t,i} \mathbf{v}_i}{\sum_{j=1}^t \alpha_{t,j}}, \quad \alpha_{t,i} = e^{\frac{\mathbf{q}_t^\top \mathbf{k}_i}{\sqrt{d_k}}}. \quad (2)$$

这里， $\alpha_{t,i}$ 表示 q_t 和 k_i 之间的注意力权重， d_k 表示键的特征维度。

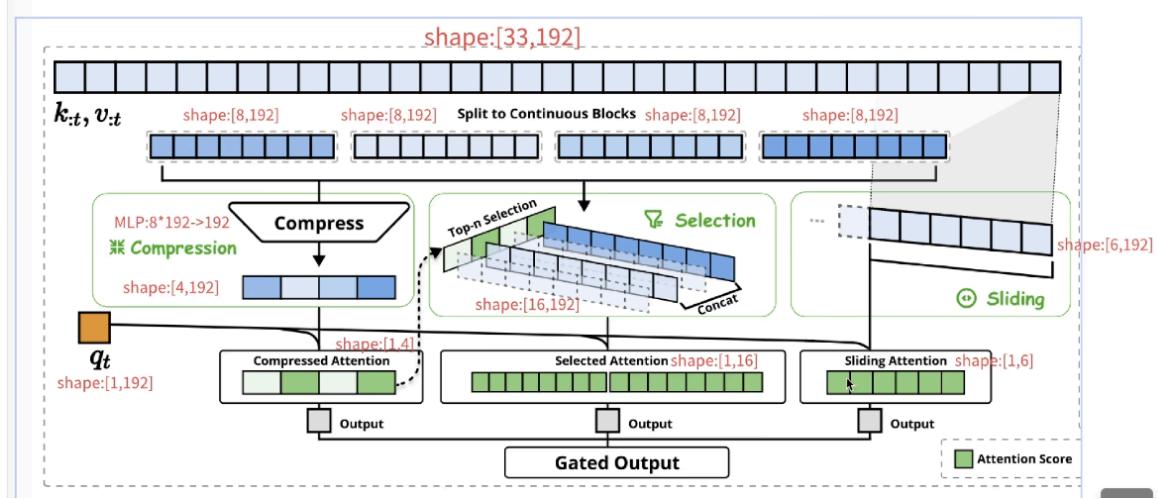
随着序列长度增加，注意力计算在整体计算开销中的占比不断提升，这对长文本处理带来了重大挑战。

- memory bound&compute bound

算术密度 算术密度指计算操作与内存访问的比值，是硬件算法优化的关键指标。每个GPU都有其特定的临界算术密度，由峰值计算能力和内存带宽的比值决定。对于计算任务而言，当算术密度高于此临界阈值时会受GPU计算能力限制（计算受限），低于此阈值则会受内存带宽限制（内存受限）。

在因果自注意力机制中，训练和预填充阶段的批量矩阵乘法和注意力计算具有高算术密度，在现代加速器上表现为计算受限。相比之下，自回归解码阶段则受内存带宽限制，因为每次前向传播仅生成一个token但需要加载完整的键值缓存，导致算术密度较低。这种特性要求在不同阶段采用不同的优化策略：训练和预填充阶段重点降低计算开销，解码阶段则着重减少内存访问。

- 整体框架：



整体框架

为充分发挥注意力机制中固有的稀疏特性，研究提出用更紧凑且信息密集的表示键值对 k_t , v_t 替代公式(1)中的原始键值对 $k_{:t}$, $v_{:t}$ 。对于每个查询 q_t ，优化后的注意力输出定义如下：

$$\tilde{K}_t = f_K(\mathbf{q}_t, \mathbf{k}_{:t}, \mathbf{v}_{:t}), \quad \tilde{V}_t = f_V(\mathbf{q}_t, \mathbf{k}_{:t}, \mathbf{v}_{:t}) \quad (3)$$

$$\mathbf{o}_t^* = \text{Attn}(\mathbf{q}_t, \tilde{K}_t, \tilde{V}_t) \quad (4)$$

其中 \tilde{K}_t , \tilde{V}_t 基于当前查询 q_t 和上下文内存 $\mathbf{k}_{:t}$, $\mathbf{v}_{:t}$ 动态构建。通过设计不同的映射策略可以获得更多种类的 \tilde{K}_t^c , \tilde{V}_t^c , 并按如下方式组合：

$$\mathbf{o}_t^* = \sum_{c \in C} g_t^c \cdot \text{Attn}(\mathbf{q}_t, \tilde{K}_t^c, \tilde{V}_t^c). \quad (5)$$

如图2所示，NSA采用三种映射策略 $\mathcal{C} = \{\text{cmp}, \text{slc}, \text{win}\}$ ，分别对应键值对的压缩、选择和滑动窗口处理。 $g_t^c \in [0, 1]$ 表示策略 \mathcal{C} 对应的门控权重，通过MLP和sigmoid激活函数从输入特征计算得到。设 N_t 为重映射键/值的总数：

$$N_t = \sum_{c \in C} \text{size}[\tilde{K}_t^c]. \quad (6)$$

通过确保 $N_t \ll t$ 来实现高稀疏度。

上述公式及其框架图解读：拿到K, V->分块(三部分，token压缩，token选择，滑动窗口)（这一步是经过映射的）（注意即使是分块了，所有的K, V都需要进行这三个部分，而不是只有K, V中的一部分进行三个阶段中的某一个阶段）->分别进行注意力机制（上面K, V头上的那个C就是代表你是三个部分中的哪个部分）->然后三部分结果进行加权求和（也就是上文说的MLP&sigmoid）

- 算法设计：本节介绍上面的映射策略fK和fV的设计，包括**token**压缩，**token**选择和滑动窗口三个组件
 - token压缩：通过连续的键值序列聚合为块级表示，可获得包含完整块信息的压缩键值对--这里主要是为了保留更粗粒度的高层语义信息，同时因为压缩个数减少了，也产生了减低一会注意力机制计算量的效果

压缩K的形式化表示:

$$\tilde{K}_t^{\text{cmp}} = f_K^{\text{cmp}}(\mathbf{k}_{\cdot t}) = \left\{ \varphi(\mathbf{k}_{id+1:id+l}) \mid 1 \leq i \leq \left\lfloor \frac{t-l}{d} \right\rfloor \right\} \quad (7)$$

其中 l 表示块长度, d 表示相邻块间的滑动步长, φ 是具有块内位置编码的可学习MLP, 用于将块内键映射为单个压缩键。 $\tilde{K}_t^{\text{cmp}} \in \mathbb{R}^{d_k \times \lfloor \frac{t-l}{d} \rfloor}$ 表示压缩键构成的张量。为减少信息碎片化, 通常采用 $d < l$ 的设置。压缩值表示 \tilde{V}_t^{cmp} 采用类似的构造方式。这种压缩表示能够捕获更粗粒度的高层语义信息, 同时降低注意力机制的计算复杂度。

窗口长度 l , 步长 d , 第 i 次移动, 窗口中的 K 的索引为 $\text{keys}[id, id+l]$ (0 为第一个索引, 这里不同于上面的公式, 因为上面的公式是 1 为第一个索引), 然后再将每个窗口中的 keys 经过 φ (也就是MLP 进行压缩)

t 为 keys 的整体长度, 所以我们可以得到公式 $t = i \times d + l$, 所以我们就得到了公式 i 的范围, $i = (t-l) / d$, 也就是块数

注意这里要求 $d < l$, 也就是每次移动的步长小于块长, 也就是说每次压缩前的 keys 都有重叠, 这样做主要是让语义连续, 防止语义的跳跃

- token选择: 选择性保留关键的单个键值对--保留重要的细粒度信息

选用分块选择策略的原因:

1. 硬件效率: 现代CPU架构处理连续块访问的时候相比随机索引读取具有更显著的吞吐量, 并且分块计算能够实现Tensor Core的最优利用
2. 注意力分数的内在分布特征: 注意力分数通常呈现空间连续性, 即相邻键往往相似的重要程度

实现: 首先将键值序列 **划分为选择块**, 然后需要为每个块分配 **重要性分数** 以识别最关键的块

- 重要性分数计算: 这里需要利用刚才压缩 **token->注意力** 计算产生的注意力分数, 然后用这个注意力分数, 有效推导选择块中的重要性分数

$$\mathbf{p}_t^{\text{cmp}} = \text{Softmax} \left(\mathbf{q}_t^T \tilde{K}_t^{\text{cmp}} \right), \quad (8)$$

其中 $\mathbf{p}_t^{\text{cmp}} \in \mathbb{R}^{\lfloor \frac{t-l}{d} \rfloor}$ 表示 \mathbf{q}_t 与压缩键 \tilde{K}_t^{cmp} 之间的注意力分数。设 l' 为选择块大小, 当压缩块和选择块采用相同的分块方案时 (即 $l' = l = d$), 可直接取 $\mathbf{p}_t^{\text{slc}} = \mathbf{p}_t^{\text{cmp}}$ 作为选择块重要性分数。对于不同分块方案的情况, 根据空间关系导出选择块的重要性分数。给定 $d \mid l$ 和 $d \mid l'$, 有:

$$\mathbf{p}_t^{\text{slc}}[j] = \sum_{m=0}^{l'-1} \sum_{n=0}^{\lfloor \frac{l}{d} \rfloor - 1} \mathbf{p}_t^{\text{cmp}} \left[\frac{l'}{d} j + m + n \right], \quad (9)$$

其中 $[\cdot]$ 表示向量元素的索引操作。对于采用GQA或MQA的模型, 由于键值缓存在查询头间共享, 需要确保所有头之间的一致块选择, 以最小化解码阶段的KV缓存加载。组内头间共享的重要性分数定义为:

$$\mathbf{p}_t^{\text{slc}'} = \sum_{h=1}^H \mathbf{p}_t^{\text{slc},(h)}, \quad (10)$$

其中 (h) 表示头索引, H 表示每组中的查询头数量。这种聚合机制确保了同组内各头之间的块选择一致性。

上面第二个公式的理解和计算

理解：

1. 先从二次求和变成一次求和

1) 从双重求和到单次求和 (把索引合并)

原式 (先把论文符号换成更便于推导的符号) :

$$\mathbf{p}^{\text{slc}}[j] = \sum_{m=0}^{A-1} \sum_{n=0}^{B-1} \mathbf{p}^{\text{cmp}}[Aj + m + n],$$

其中我令 $A = \frac{l'}{d}$, $B = \frac{l}{d}$ 。 (若 Aj 本身为整数, 则不需要 floor; 若不是, floor 会把起始索引向下取整——工程上通常使 A 和 j 使其为整数。)

现在把双和里的 m, n 的和记作 $k = m + n$ 。那么外面的索引变为:

- k 的可能取值范围: 最小为 0, 最大为 $(A - 1) + (B - 1) = A + B - 2$ 。
- 对固定的 k , 满足 $m + n = k$ 的 (m, n) 对的个数不是恒定的; 我们把这个个数记为 c_k 。

于是可以把双重求和改写为单层关于 k 的求和:

$$\mathbf{p}^{\text{slc}}[j] = \sum_{k=0}^{A+B-2} c_k \mathbf{p}^{\text{cmp}}[Aj + k].$$

这就是“把双重和合并成一次和, 并把重复计数用系数 c_k 表示”。

2. C_k 的具体表达: 为什么呈三角形

k 固定, $k = m + n$, m 和 n 可以有不同的表示形式

比如 k 等于 3, 有 4 种组合情况, $k=2$, 有 3 种, $k=1$, 有两种, $k=0$, 有一种

所以 k 从 0 到 3, 对应 $j=0$ 的时候 \mathbf{p} 有 1 个 $\mathbf{p}[0]$, 2 个 $\mathbf{p}[1]$, 3 个 $\mathbf{p}[2]$, 4 个 $\mathbf{p}[3]$ 组成, 就成了一个三角形

可以看作是一维卷积+下采样

因为中间元素出现次数多, 所以就起到了突出中间信息, 抑制边缘噪音的效果 (中间的信息最能代表该窗口整体的语义)

第三个公式理解: 如果是 GQA 或者 MQA 的话

那么因为多个 Q 依赖同一个 K , V , 这就要求每个 Q 得到的 \mathbf{p} 是一样的, 那么就在刚才第二个公式的基础上求和 (为什么求和: 因为该大的地方求和之后还是大, 所以求和基本上不会掩盖原来的重要性分布的信息)

■ Top-n 块选择: 在获得选择块重要性分数后, 保留重要性分数排名前 n 的稀疏块中的 token

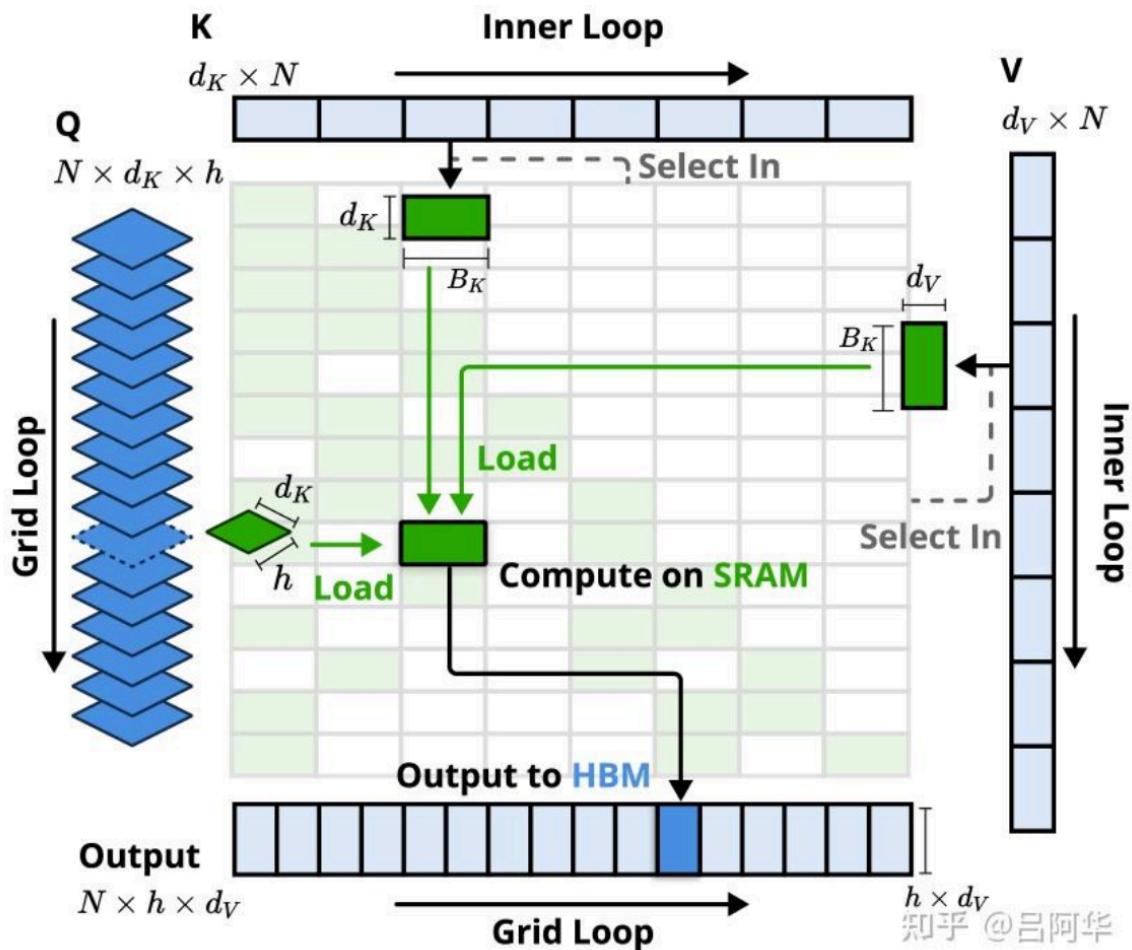
Top-n 块选择 在获得选择块重要性分数后, 保留重要性分数排名前 n 的稀疏块中的 token, 形式化表示为:

$$\mathcal{I}_t = \{i \mid \text{rank}(\mathbf{p}_t^{\text{slc}'}[i]) \leq n\} \quad (11)$$
$$\tilde{K}_t^{\text{slc}} = \text{Cat} [\{\mathbf{k}_{il'+1:(i+1)l'} \mid i \in \mathcal{I}_t\}], \quad (12)$$

其中 $\text{rank}(\cdot)$ 表示降序排序位置 ($\text{rank}=1$ 对应最高分数), \mathcal{I}_t 为选中块的索引集合, Cat 表示连接操作。 $\tilde{K}_t^{\text{slc}} \in \mathbb{R}^{d_k \times nl'}$ 表示由压缩键组成的张量。细粒度值 \tilde{V}_t^{slc} 采用类似的方式。最后, 按照公式(5)所定义的方式, 将选中的键值对参与与 \mathbf{q}_t 的注意力计算。

- 滑动窗口：用来处理局部上下文，使刚才两个分支能够专注于各自特征的学习
这里滑动窗口内部就是维护w长的最近token序列
- 计算核心设计--基于Triton实现了硬件友好的稀疏注意力计算核心，重点关注具有KV共享缓存架构的GQA&MQA（第一篇论文模型级优化提到了这两个词）
压缩和滑动窗口计算可以适配现有FlashAttention-2计算核心
但稀疏选择注意力需要专门的核心设计（改进就是将QMA组内有共同稀疏KV块的查询头一并加载至SRAM）
此图是在硬件层面进行稀疏化Attention的流程图
注意力需要专门的核心设计。如果采用FlashAttention的策略将时序连续的查询块加载至SRAM，由于块内查询可能需要访问不连续的KV块，将导致内存访问效率低下。

为此，研究提出了新的查询分组策略作为关键优化：对查询序列的每个位置，将GQA组内所有共享相同稀疏KV块的查询头一并加载至SRAM。



知乎 @吕阿华

图3展示了前向传播的具体实现。该计算核心架构具有以下关键特征：

1. **基于组的数据加载**: 在每个内循环中, 加载位置 t 处组内所有查询头 $Q \in \mathbb{R}^{[h, d_k]}$ 及其共享的稀疏键/值块索引 \mathcal{I}_t 。
2. **共享式KV获取**: 内循环中, 顺序加载由 \mathcal{I}_t 索引的连续键/值块至SRAM ($K \in \mathbb{R}^{[B_k, d_k]}$, $V \in \mathbb{R}^{[B_k, d_v]}$), 以最小化内存加载开销。其中 B_k 表示满足 $B_k | l'$ 的核心块大小。
3. **网格化外循环**: 由于内循环长度 (与选定块数 n 成正比) 对不同查询块基本一致, 因此将查询/输出循环置于Triton的网格调度器中, 以简化和优化核心性能。

该设计通过以下方式实现了接近最优的算术密度：

1. 利用组内共享机制消除冗余的KV数据传输;
2. 在GPU流式多处理器间实现计算负载的均衡分配。

代码实现

[NSA代码实现--知乎](#)

量化Quant

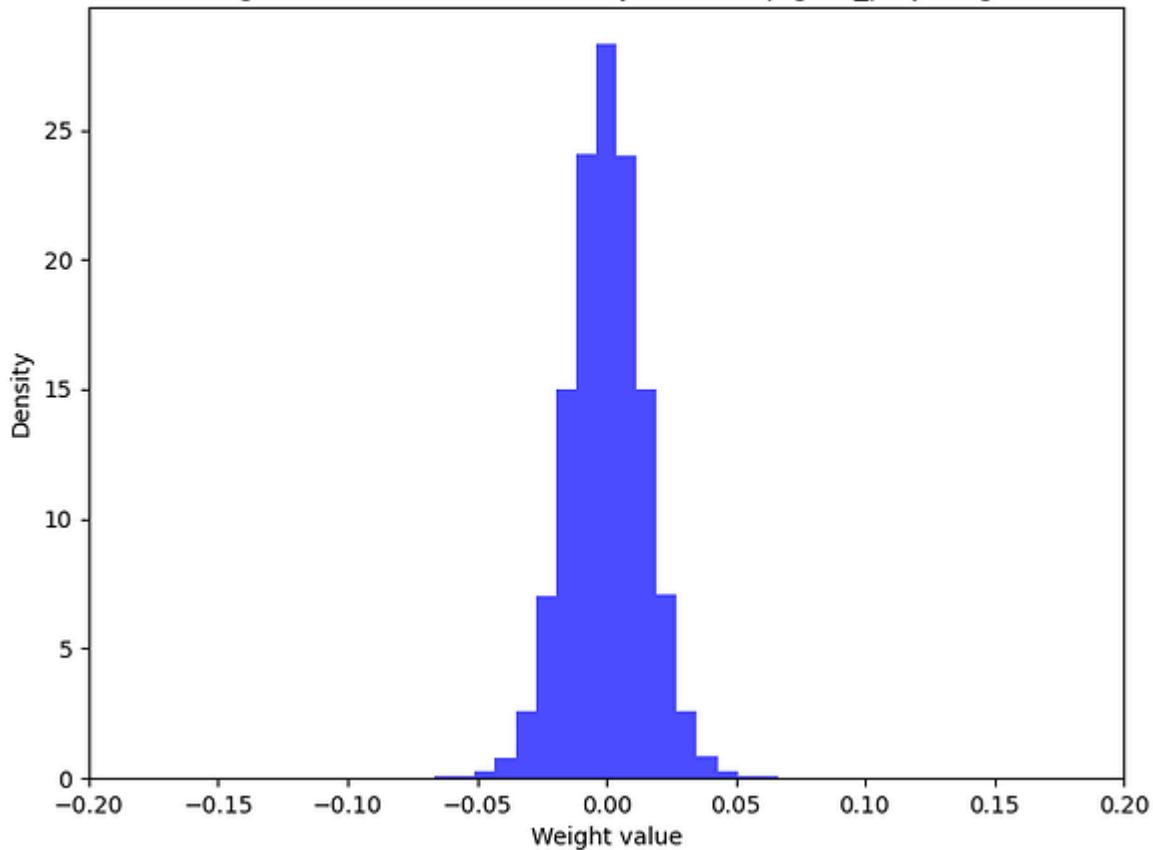
量化Quant: 将模型参数与中间计算由高精度的浮点表示 (FP32,FP16) 压缩到更低精度的表示 (INT8/INT4/3bit/1bit), 以此减少显存占用和内存带宽压力, 同时加快计算速度

为什么能用量化这种方法进行优化:

如下图, 大模型的权重的数值分布非常密集, 而我们用**FP32**和**FP16**的原因就是他们有更高的精度, 也就是有更大的数据表示范围, 但是对于下图这种情况来说, 高精度的优势是用不上的

所以我们使用更低精度的表示, 这样既减少了内存开销, 也不会损失模型精度, 更加速了计算速度

Weight Distribution of model.layers.26.mlp.gate_proj.weight



1. 目前流行的量化算法有哪些

[A Survey of Quantization Methods for Efficient Neural Network Inference](#)

- Basic Concepts of Quantization

- 1. 损失函数:

- • 假设NN有L层具有可学习参数, 记为{W1, W2, ..., WL}, 其中θ表示所有这些参数的组合。在不损失一般性的情况下, 我们专注于监督学习问题, 其名义目标是优化以下经验风险最小化函数:

$$L(\theta) \frac{1}{N} \sum_{i=1}^N l(x_i, y_i; \theta)$$

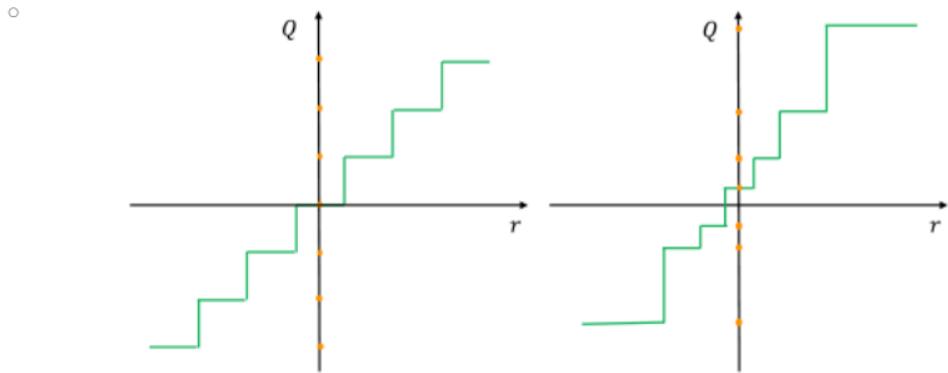
- 式中(x, y)为输入数据及对应的标号, l(x, y; θ)为损失函数(如均方误差或交叉熵损失), N为数据点总数。我们也将第i层的输入隐藏激活表示为hi, 对应的输出隐藏激活表示为ai。我们假设我们有训练好的模型参数θ, 以浮点精度存储。在量化中, 目标是将参数(θ)以及中间激活映射(即hi, ai)的精度降低到低精度, 同时对模型的泛化能力/精度影响最小。为此, 我们需要定义一个量化运算符, 将浮点值映射到量化的浮点值, 这将在下面描述。

2. 均匀量化/对称量化&非对称量化

这里可以看出来, 量化是一个从均匀值到离散值的一个映射过程

均匀量化就是映射后的离散值是等间距的

我们首先需要定义一个函数，它可以将神经网络的权重和激活量化到一个有限的值集。该函数接受浮点实值，并将它们映射到较低的精度范围，如下图所示。



- 均匀量化(左)与非均匀量化(右)的比较。连续域r中的实值映射到量子化域Q中离散的、精度较低的值，用橙色子弹标记。请注意，在均匀量化中，量子化值(量化水平)之间的距离是相同的，而在非均匀量化中，它们可以变化。

■ 量化函数：

量化函数的常用选择如下: $Q(r) = \text{Int}(r/S) - Z$ 。其中Q为量化算子，r为实值输入(激活或权重)，S为实值缩放因子，Z为整数零点。此外，Int函数通过舍入操作(例如，舍入到最近和截断)将实值映射为整数值。本质上，这个函数是从实数r到某个整数值的映射。这种量化方法也被称为均匀量化，因为得到的量化值(又称量化级别)是均匀间隔的(上图，左)。还有一些非均匀量化方法，其量化值不一定是均匀间隔的(图1，右)，这些方法将在第III-F节中进行更详细的讨论。通过一种通常称为去量化的操作，可以从量子化的Q®中恢复实值r:

$$\bar{r} = S(Q(r) + Z)$$

- 请注意，由于舍入操作，恢复的实值 \bar{r} 将不完全匹配r。

S: 实值缩放因子 (步长)

比如将 $[-1,1]$ 的数据映射到7位比特上面，就是将这个均匀的离散数据平均分为 $2^b - 1$ 段，每一段代表7位比特中的一个数值(因为-1所以少一个数值)，这就完成了由连续值到离散值的一个映射

- 均匀量化的一个重要因素是上公式中比例因子S的选择。这个比例因子本质上是将给定的实数r范围划分为多个分区：

$$S = \frac{\beta - \alpha}{2^b - 1}$$

- 式中 $[\alpha, \beta]$ 表示裁剪范围，即对实值进行裁剪的有界范围， b 为量化位宽度。因此，为了定义比例因子，首先要确定裁剪范围 $[\alpha, \beta]$ 。选择裁剪范围的过程通常称为校准。

上面 r/S 的意思就是看 r 在哪一段里面，并带有舍入

这里的Z就是映射后的零点(这里的零点不一定为0，如果是对称量化的话零点就是0，如果不是就不为0)

- 一个简单的选择是使用信号的最小/最大值作为裁剪范围，即 $\alpha = r_{\min}$, $\beta = r_{\max}$ 。这种方法是一种非对称量化方案，因为裁剪范围不一定是相对于原点对称的，即 $-\alpha \neq \beta$ ，如下图(右)所示。通过选择 $\alpha = -\beta$ 的对称裁剪范围，也可以使用对称量化方案。一个流行的选择是根据信号的最小/最大值来选择这些: $-\alpha = \beta = \max(|r_{\max}|, |r_{\min}|)$ 。与对称量化相比，非对称量化通常导致更窄的裁剪范围。当目标权重或激活不平衡时，这一点尤其重要，例如，ReLU之后的激活总是具有非负值。然而，使用对称量化，通过将零点替换为 $Z = 0$ ，简化了 Eq. 2 中的量化函数:

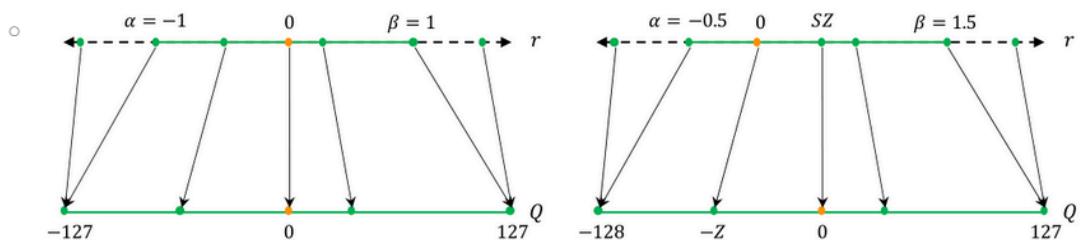
-

$$Q(r) = \text{Int}\left(\frac{r}{S}\right)$$

- 这里，比例因子有两种选择。在“全量程”对称量化中， S 被选择为 $\frac{2\max(|r|)}{2^n - 1}$ (带舍入模式)，以使用 $[-128, 127]$ 的全INT8范围。然而，在“限制范围”中， S 被选择为 $\frac{\max(|r|)}{2^{n-1} - 1}$ ，它只使用 $[-127, 127]$ 的范围。正如预期的那样，全范围方法更加准确。对称量化在量化权重的实践中被广泛采用，因为将零点归零可以减少推理过程中的计算成本，并且使实现更加简单。然而，请注意，对于激活，由于非对称激活中的偏移而占用的交叉项是一个静态的数据独立项，可以在偏置中吸收(或用于初始化累加器)。
- 利用信号的最小/最大值进行对称和非对称量化是一种常用的方法。然而，这种方法容易受到激活中的异常数据的影响。这些可能会不必要地增加范围，从而降低量化的分辨率。解决这个问题的一种方法是使用百分位数而不是信号的最小/最大值。也就是说，不是使用最大/最小值，而是使用第 i 个最大/最小值作为 β/α 。另一种方法是选择 α 和 β ，以最小化实值和量化值之间的KL散度(即信息损失)。
- 摘要(对称与非对称量化)。对称量化使用对称范围对裁剪进行分区。这样做的好处是更容易实现，因为它会导致 Eq. 2 中的 $Z = 0$ 。然而，对于范围可能偏斜且不对称的情况，它是次优的。对于这种情况，**不对称量化是首选**。

比例因子 (如果是带舍入的):

$$2\max(|r|)/2^n - 1$$



- 对称量化和非对称量化的说明。限制范围的对称量化将实值映射到 $[-127, 127]$ ，对于 8 位量化，全范围映射到 $[-128, 127]$ 。

3. 静态量化和动态量化--何时确定裁剪范围

因为训练的时候权重已经确定了，推理的时候权重不变，所以权重使用量化在推理运行前进行

然后对于激活值 (**Activations**) (推理时产生的输出数据) 在每次推理的时候是不一样的

所以对于激活值的量化可以分为动态量化和静态量化

- 动态量化：在推理运行时为每个激活映射动态计算

开销大 (实时统计)，会导致更高的精度，因为每个输入信息范围是精确计算的

- 静态量化：裁剪范围预先计算，推理期间是静态的

开销小，但是精度也小

4. 量化粒度--如何为权重计算裁剪范围[]的粒度

- 分层量化：考虑一层中所有的权重来确定裁剪范围

方法容易实现，但对于权重范围变化很大的层可能会导致精度不是最优

比如同一层中都是较小的权重但是出现了一两个较大的权重，那么量化就会出现范围拉得比较大的情况，从而导致权重较小的失去了它的分辨率

- 分组量化：一层中分组多个不同的通道来计算裁剪范围

与下面的区别就是下面是每一个，上面是组

- 信道量化：对于一个权重张量我们为每一个输出通道设计一个专属的裁剪范围/缩放因子S

目前采用的标准量化方法

- 子通道量化：将每个输出通道划分为多个子通道并设计专属的裁剪范围/缩放因子S

粒度最优

5. 非均匀量化--量化后的离散值不是均匀分布的

- 文献中的一些工作也探索了非均匀量化，其中量化步骤和量化水平可以是非均匀间隔的。

◦

$$Q(r) = X_i, \text{ if } r \in [\Delta i, \Delta i + 1).$$

- 具体来说，当实数r的值落在量化步长 Δi 和 $\Delta i + 1$ 之间时，量化器Q将其投影到相应的量化级别 X_i 。请注意， X_i 和 Δi 的间距都不是均匀的。

基于对数分布的非均匀量化，基于二进制码的量化，可学习量化器

可以实现更高的精度，因为可以通过关注重要值区域或找到适当的动态范围来更好的捕获分布，但是难以在通用计算硬件上有效部署

目前还是均匀量化做的多

6. 微调方法--量化后需要调整神经网络中的参数

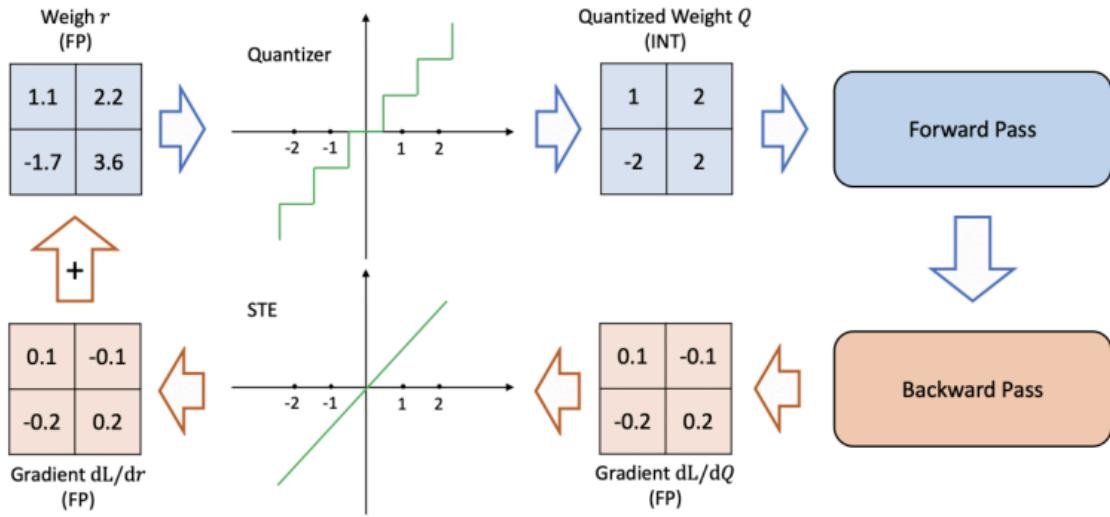
量化感知训练/训练时量化QAT：在训练时就引入量化

过程：在前向和反向传播完成后，即权重更新完成后，立即量化一次，然后放回模型中进行下一次前向传播

问题解决：量化操作是不可微分的，在反向传播的过程中，量化操作的梯度几乎处处为0（因为舍入操作的存在）导致梯度无法传播，训练无法进行

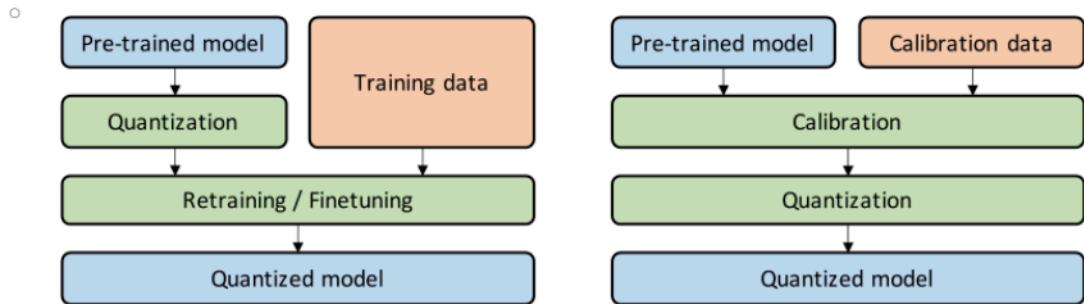
STE直通估计器：在反向传播时完全忽略量化操作（用恒等函数1来近似它的梯度）

缺点：重新训练模型花的成本和时间都太高（但是效果也确实好）



后训练量化PTQ:不重新训练的情况下完成

优点: 需任何微调--开销很低, 可以应用于数据有限资源有限的情况 (不过精度较低)



量化感知训练 (QAT, 左) 和训练后量化 (PTQ, 右) 之间的比较。在 QAT 中, 对预训练模型进行量化, 然后使用训练数据进行微调, 以调整参数并恢复准确性下降。在 PTQ 中, 使用校准数据 (例如, 训练数据的一个小子集) 来校准预训练模型, 以计算裁剪范围和缩放因子。然后, **根据校准结果对模型进行量化**。请注意, 校准过程通常与 QAT 的微调过程并行进行。

零成本量化: 在极端场景下使用--完全无法获取原始训练数据

- ZSQ+PTQ: 无数据, 无微调, 最快最简单
- ZSQ+QAT: 无数据, 但需要微调, 精度更高

7. 随机量化

与确定性量化相比, 随机量化可以让神经网络探索更多, 或者说随机舍入可能会为神经网络提供逃脱的机会, 从而更新其参数 (因为小的权重更新可能不会导致任何权重的变化, 因为舍入操作可能会返回相同的权重)

正片开始! ! !

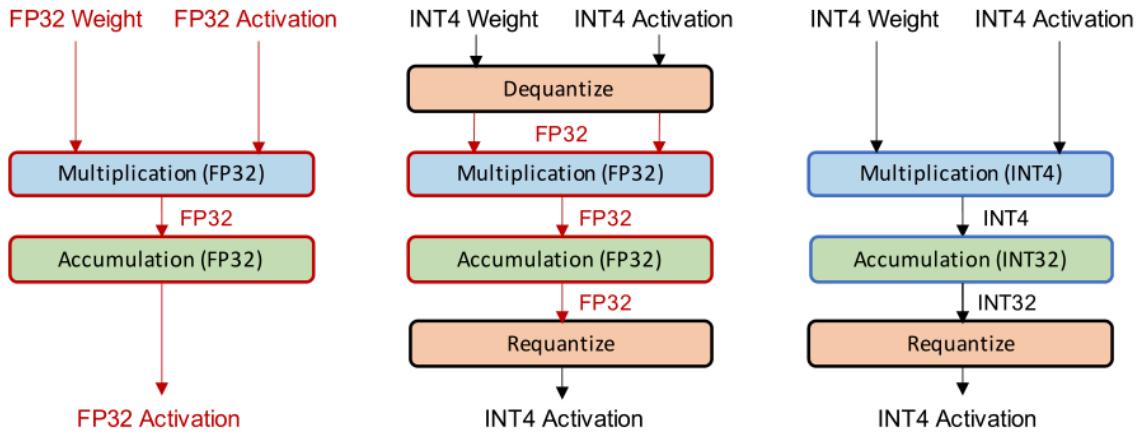
- Advanced Concepts : Quantization below 8 Bits (8比特一下的量化)

1. Simulated and Integer-only Quantization--模拟量化和纯整数量化

模拟量化: 量化后的模型参数以低精度存储, 但运算用浮点运算进行, 因此在浮点运算之前需要对量化参数进行反量化

模拟量化人们无法受益于低精度所有的算术执行

纯整数量化：所有运算都是使用低精度整数算术执行，所以允许使用高效的整数算术来执行整个推理

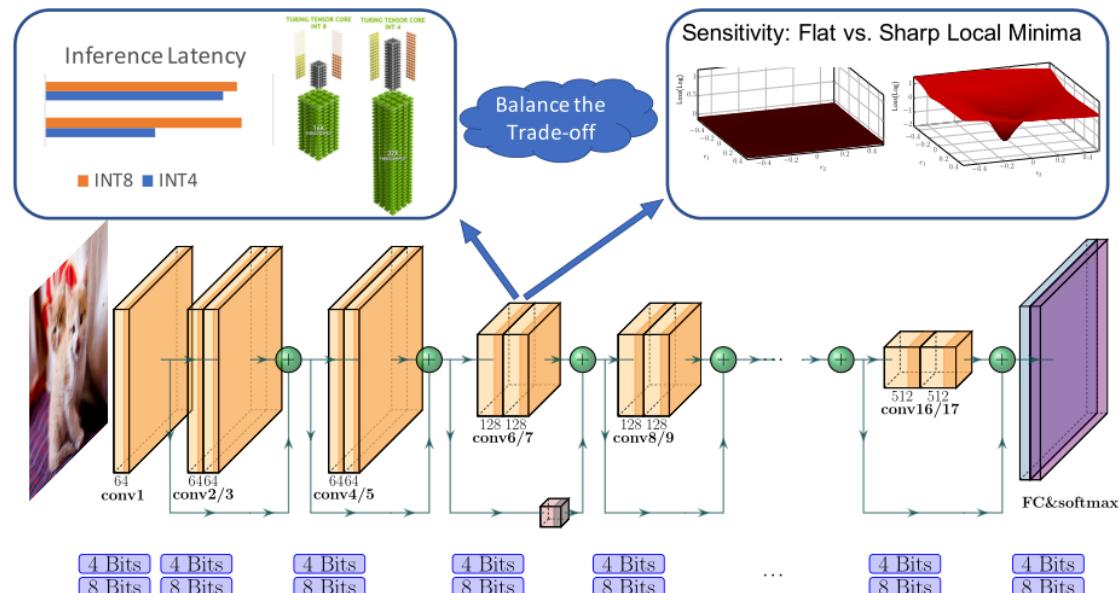


全精度推理（左），模拟量化推理（中），纯整数量化推理（右）

2. Mixed-Precision Quantization--混合精度量化

每一层都以不同的位精度进行量化，敏感高效的层保持在较高的精度，并且对不敏感和低效的层应用低精度量化

这种方法在硬件性能和精度上面达到了一种平衡，并且仍然可以减少内存占用



如何为每一层选择这种混合精度：基于搜索方法/基于正则化

3. Hardware Aware Quantization--硬件感知量化

由于量化的效果也取决于硬件，片上内存，带宽和缓存层次结构

在量化训练过程中，严格模拟目标硬件在推理时的计算行为

4. Distillation-Assisted Quantization--蒸馏辅助量化

结合模型蒸馏来提高量化速度

5. Extreme Quantization--极端量化

将权重和激活值量化为极低的位宽（1bit二值化/2bit三值化）

追求极致的推理速度和能耗降低，但是会带来巨大的信息损失

6. Vector Quantization--向量量化

分组替换，不是直接缩小每个数字的体积而是将大量相似的数据点分组，然后用一个代表来替代整个组，最后只存储分组索引和代表值

压缩比极高

- Quantization and Hardware Processors

量化减小模型大小，实现更快的速度和更低的功耗，特别是对于具有低精度的硬件，因此，**量化对于物联网和移动应用的边缘部署尤其重要**

2. 量化分为训练时量化 (Quantization-Aware Training, QAT) 以及后训练量化 (Post-Training Quantization, PTQ)，这里我们主要关注 PTQ 方法。阅读下面的论文，总结两种方法的异同：

量化方法：

- 训练时量化：在量化到再训练/微调过程中对模型进行量化，并使用某种近似微分机制进行舍入运算
- 训练后量化：使用适度的资源（通常几千个数据样本和几个小时的计算时间）来量化预训练模型

对于大规模模型尤其有用，因为对于大规模模型来说，完整模型训练甚至微调都可能成本高昂

[GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers](#)

GPTQ：生成式预训练Transformer的精确后训练量化（PTQ）

类型：3-4bit量化

保持高准确性的同时，实现对**大型GPT模型**的高效量化

- 基础知识：

- 层级量化--Layer-Wise Quantization（量化粒度为层级）：

目标：保证量化前后的输入误差最小

在量化过程中，通常对每一层分别进行处理，解决一个对应的重建问题。对于给定的线性层 l ，其权重矩阵为 W_l ，输入数据为 X_l 。目标是找到量化后的权重矩阵 \widetilde{W}_l ，使得量化前后的输出误差最小：

$$\min_{\widetilde{W}_l} \|W_l X_l - \widetilde{W}_l X_l\|_F^2$$

这里， $\|\cdot\|_F$ 表示Frobenius范数。

Frobenius范数：

将范数里面矩阵的每个元素平方，然后将所有平方后的元素加起来，然后对求得的和进行平方根运算

数学公式

对于一个 $m \times n$ 的实数矩阵 A ，其元素记为 a_{ij} ，它的Frobenius范数正式定义为：

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} = \sqrt{\text{tr}(A^T A)}$$

其中：

- $\text{tr}(A^T A)$ 是矩阵 $A^T A$ 的迹 (Trace)，即矩阵主对角线元素之和。
- A^T 是 A 的转置。

简单计算步骤：

1. **平方**：将矩阵中的每个元素进行平方。
2. **求和**：将所有平方后的元素加起来。
3. **开根**：对求得的和进行平方根运算。

直观理解与例子

假设我们有一个矩阵：

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

计算它的Frobenius范数：

1. **平方每个元素**： $1^2, 2^2, 3^2, 4^2 \rightarrow 1, 4, 9, 16$
2. **求和**： $1 + 4 + 9 + 16 = 30$
3. **开根**： $\|A\|_F = \sqrt{30} \approx 5.477$

这个值 (5.477) 就代表了矩阵 A 的整体“能量”或“幅度”。

- Optimal Brain Quantization (OBQ) 方法--GPTQ方法就是在这个方法上面进行改进的：

类型：极低bit，2/3/4bit

1. 一种逐个量化权重的方法，目的是在量化过程中最小化整体误差
2. 逐个量化权重，按照权重对量化误差的影响，从大到小选择权重进行量化

3. 误差补偿：每次量化一个权重后，会调整其他尚未量化的权重，以补偿因量化引入的误差
4. 通过Hessian矩阵（损失函数对权重的二阶导数）来精确计算如何最优的调整剩余权重

具体过程：

1. 量化权重 w_q ：首先量化为最接近的离散值

$$\hat{w}_q = \text{quant}(w_q)$$

2. 计算调整量 δ_F

1. 目标：最小化误差的增加

神经网络训练的目标是最小化损失函数 L 。量化可以看作是对最优权重 \mathbf{w}^* 引入了一个扰动 δ ，导致损失函数增加 ΔL 。

OBQ的目标就是让这个 ΔL 尽可能小。对于一个特定的权重扰动 δ ，其引起的损失变化可以用泰勒展开式近似（围绕原始最优权重 \mathbf{w}^* ）：

$$\Delta L \approx \underbrace{\delta^T \mathbf{g}}_{\text{一阶项}} + \frac{1}{2} \underbrace{\delta^T \mathbf{H} \delta}_{\text{二阶项}}$$

其中：

- \mathbf{g} 是梯度（Gradient）。由于我们是在训练好的最优权重附近操作，梯度 $\mathbf{g} \approx 0$ （saddle point 或局部最优点），所以一阶项可以忽略。
- \mathbf{H} 是Hessian矩阵，即损失函数对权重的二阶导数矩阵 $(H_{ij}) = \frac{\partial^2 L}{\partial w_i \partial w_j}$ 。它编码了权重之间相互作用的“曲率”信息。

因此，目标简化为最小化二阶项：

$$\Delta L \approx \frac{1}{2} \delta^T \mathbf{H} \delta$$

2. 我们的扰动 δ 是什么？

在我们的场景中，扰动 δ 由两部分组成：

1. 对权重 w_q 的强制量化扰动： $\delta_q = \text{quant}(w_q) - w_q$ 。这是一个固定值，我们无法改变。
2. 对其他权重 F 的补偿性调整： δ_F 。这是我们希望通过计算得到的、可以自由选择的变量。

所以，完整的扰动向量 δ 可以写为：

$$\delta = \begin{bmatrix} \delta_F \\ \delta_q \end{bmatrix}$$

我们的目标是：找到一个最优的 δ_F ，使得在给定固定量化误差 δ_q 的前提下，总的损失增加 ΔL 最小。

3. 将扰动代入目标函数

将 $\delta = [\delta_F, \delta_q]^T$ 代入二阶误差公式：

$$\Delta L \approx \frac{1}{2} \begin{bmatrix} \delta_F^T & \delta_q^T \end{bmatrix} \begin{bmatrix} H_{F,F} & H_{F,q} \\ H_{q,F} & H_{q,q} \end{bmatrix} \begin{bmatrix} \delta_F \\ \delta_q \end{bmatrix}$$

展开这个二次型：

$$\Delta L \approx \frac{1}{2} [\delta_F^T H_{F,F} \delta_F + \delta_F^T H_{F,q} \delta_q + \delta_q^T H_{q,F} \delta_F + \delta_q^T H_{q,q} \delta_q]$$

由于 H 是对称矩阵 ($H_{q,F} = H_{F,q}^T$)，上式可以写为：

$$\Delta L \approx \frac{1}{2} [\delta_F^T H_{F,F} \delta_F + 2\delta_F^T H_{F,q} \delta_q + \delta_q^T H_{q,q} \delta_q]$$

4. 求解最优的 δ_F

现在，我们的任务是最小化 ΔL 。注意到上式右边：

- 第三项 $\delta_q^T H_{q,q} \delta_q$ 是常数（因为 δ_q 是固定的）。
- 第一项和第二项是关于 δ_F 的二次函数。

因此，最小化 ΔL 等价于最小化：

$$J(\delta_F) = \frac{1}{2} \delta_F^T H_{F,F} \delta_F + \delta_F^T (H_{F,q} \delta_q)$$

这是一个经典的无约束二次优化问题。其最优解可以通过对 $J(\delta_F)$ 求导并令导数为零得到：

$$\frac{\partial J}{\partial \delta_F} = H_{F,F} \delta_F + H_{F,q} \delta_q = 0$$

解这个方程：

$$H_{F,F} \delta_F = -H_{F,q} \delta_q$$

$$\delta_F = -H_{F,F}^{-1} H_{F,q} \delta_q$$

回想一下， $\delta_q = \text{quant}(w_q) - w_q$ ，但我们有 $(w_q - \text{quant}(w_q)) = -\delta_q$ 。所以代入上式：

$$\begin{aligned} \delta_F &= -H_{F,F}^{-1} H_{F,q} (-\delta_q) = -H_{F,F}^{-1} H_{F,q} \cdot (- (w_q - \text{quant}(w_q))) \\ \delta_F &= H_{F,F}^{-1} H_{F,q} (w_q - \text{quant}(w_q)) \end{aligned}$$

注意：原PPT中的公式是：

$$\delta_F = -H_{F,F}^{-1} H_{F,q} (w_q - \text{quant}(w_q))$$

这里的负号 (-) 与上面的推导差了一个负号。这可能是由于对 δ_q 的定义不同（是量化误差还是量化扰动）。在实际的OBQ论文中，通常使用负号的形式来确保调整方向是补偿而非放大误差。我们可以这样理解：量化操作 ($\text{quant}(w_q)$) 使得权重变小了（假设 $w_q > \text{quant}(w_q)$ ），那么为了补偿这个“损失”，我们需要让其他权重相应地增大一点。负号确保了 δ_F 的符号与误差相反，从而实现补偿。

3. 更新未量化权重F

$$w_F \leftarrow w_F + \delta_F$$

这种做法的缺点和优点：精确，但是计算量及其大

- GPTQ

1. 舍弃贪心顺序--解决计算量大的问题：观察发现按照任意固定顺序量化权重结果和贪心策略相近--这样做的好处是舍弃了贪婪顺序的计算成本，并且也保证了最终的计算结果

- 这么做的原因：在大型网络中，由于可调整补偿的权重 (F) 非常多，即使早期量化了一些误差较大的权重，后期也有足够多的其他权重可以对其进行补偿。因此，量化顺序的重要性下降了
- 改进的优势：

比如在原方法中第一行处理第二个元素B，第二行处理第一个元素A，这样就造成了每一行都有自己的 $H_{F,F}$ (需要每行行内再去搜索，这样造成的计算复杂度就是 $O(\text{行} \times \text{列})$) 但是如果我每一行都处理第一个元素，那么每行内就不用重新搜索，这样就将计算复杂度降为 $O(\text{列})$

2. 延迟批量更新--解决内存问题

尽管上述方法已经优化了计算，但还是存在着memory bound，因为 $H_{F,F}$ 这个矩阵实在是太大了，这样实现它就得等着将整个矩阵加载，造成了计算资源需要等待内存访问资源

- 观察：在量化第i列的权重的时候，所需的计算只以来当前已更新的 $H_{F,F}$ 中与第i列相关的部分
- 改进：利用cache减少访问内存的次数（利用的思想就是根据cache将矩阵进行分块计算，减少访问内存的次数）。

每次不是处理一整列，而是每次处理一个Block (128列)，每次只需要用到这个B列的块和 $H_{F,F}$ 中对应的 $B \times B$ 大小的子块，这样在B列的处理之中就可以只将这 $B \times B$ 大小的子块存入cache，等计算完这B列再将这一批的更新更新回W矩阵和 $H_{F,F}$ 矩阵上

3. Cholesky分解重构

- 问题：在量化超大模型的时候，由于计算机浮点数精度的限制，数值计算误差会累积，成为一个严重问题--这导致对 $H_{F,F}$ 的计算可能会变得不准确
- 观察，在量化某个权重q的时候，算法真正需要从 $H_{F,q}$ 中获取的信息，其实仅仅是第q行的数据，因此不要冒险去计算和更新整个 $H_{F,F}$ 矩阵，预先计算并存储需要的矩阵分解结果
- 改进：使用乔列斯基分解：将一个正定矩阵 A 分解为 $A = L L^T$ ，利用预先计算并存储的矩阵分解结果，在量化过程中避免重复的矩阵求逆操作，提高数值的稳定性

AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration

- 前置知识：

1. 量化的两种分类

2. LLM量化：第一种是W8A8，激活和权重均量化为INT8，第二种仅权重量化，即进权重量化为低位（本文关注后面一种）（大幅降低存储和内存带宽，加速推理）

- AWQ:激活感知权重量化--本文中介绍的量化为训练后量化

- 通过保留1%的显著权重来就可以极大减少量化误差

- 这么做的原因：LLM的权重并非同等重要，一小部分的显著权重对性能起影响作用，跳过这一部分的量化有助于弥补量化造成的性能下降
- 如何保留这些显著权重：根据激活值的幅度来选择保留的权重，那些会产生较大激活值的输入特征对应的权重被认为更重要

- 局限性：这种虽然开销很小，并且弥补了性能，但是这种INT3和FP16夹杂的数据类型会给实际系统实现和计算带来困难
所以我们需要一种其他的方法来保留这些显著的权重
- 通过激活感知缩放来保留显著权重
 - 仅权重的量化带来的误差

假设一组权重 \mathbf{w} ，线性操作是 $y = \mathbf{w}\mathbf{x}$ ，权重量化后为 $y = Q(\mathbf{w})\mathbf{x}$ ，展开如下：

$$y = Q(\mathbf{w}) \cdot x = \Delta \cdot \text{Round}\left(\frac{\mathbf{w}}{\Delta}\right) \cdot x, \quad \Delta = \frac{\max(|\mathbf{w}|)}{2^{N-1}}$$

N 是量化 bits 数， Δ 是量化缩放系数。假设某个权重值 $w \in \mathbf{w}$ ，我们额外乘以 $s > 1$ ，具体公式如下：

$$y = Q(w \cdot s) \cdot \frac{x}{s} = \Delta' \cdot \text{Round}\left(\frac{ws}{\Delta'}\right) \cdot x \cdot \frac{1}{s}$$

Δ' 是新的量化缩放系数，论文发现如下几点：

1. $\text{Round}\left(\frac{w}{\Delta}\right)$ 误差 RoundErr 大致均匀分布在 0-0.5，平均误差为 0.25
2. 放大某个单一权重 w 并不会影响该组权重的极大值，因此可以认为 $\Delta' \approx \Delta$
3. 新 $\text{Round}\left(\frac{ws}{\Delta'}\right)$ 误差 $\text{RoundErr}' = \frac{\Delta'}{\Delta} \cdot \text{RoundErr} \cdot \frac{1}{s}$
4. $\frac{\text{RoundErr}'}{\text{RoundErr}} = \frac{\Delta'}{\Delta} \cdot \frac{1}{s}$ ，由于 $\Delta' \approx \Delta$ 且 $s > 1$ ，新 $\text{RoundErr}'$ 相比原始 RoundErr 会进一步降低

第一个公式就是普通的缩放因子，上上篇学习量化基础知识的时候就已经知道这个公式了

Round就是将其缩放到整数区间，然后四舍五入到最近的整数，然后再乘上 δ 将其缩放回原始的数值范围，这样得到的 $Q(\mathbf{w})$ 就是可以被低精度表示的权重值

这样做的问题就是只能保证 $Q(\mathbf{w})$ 近可能最小化权重误差，但是忽略了输入激活 \mathbf{x} 的分布

第二个公式：AWQ采用的做法

这样做在数学上是等价的和上面那个公式，但是表达的含义不一样，这个公式将 w 放大了 s 倍，将 x 缩小了 s 倍

论文的四点发现在上面显示出来了

这里做一下通俗解释：

1. 这样做缩放系数又 $\max(|w|)$ 决定，当我们放大某一个权重几乎不会影响 $\max(|w|)$ ，除非这个权重本身就是最大值，既然 $\max(|w|)$ 没变，那么权重系数也可以认为几乎不变（因为我们保护的是激活值大的那些权重，而这些权重与 $\max(|w|)$ 的重合概率其实是很低的）
2. 新的 RoundError 误差：因为多 $\times (1/s)$ 所以误差进一步降低
3. 新 $\text{Round}\left(\frac{ws}{\Delta'}\right)$ 误差 $\text{RoundErr}' = \frac{\Delta'}{\Delta} \cdot \text{RoundErr} \cdot \frac{1}{s}$
4. $\frac{\text{RoundErr}'}{\text{RoundErr}} = \frac{\Delta'}{\Delta} \cdot \frac{1}{s}$ ，由于 $\Delta' \approx \Delta$ 且 $s > 1$ ，新 $\text{RoundErr}'$ 相比原始 RoundErr 会进一步降低

如何对上上面的思想：即保留那些激活值大的权重的精度，答案就是设置不同的缩放因子，对那些激活值大的权重的 s （这里先告诉你是这么个逻辑，下面一部分我们讲怎么找出这么一个 s ）就设置的大一点

根据上面那个RoundError的误差公式， s 越大，误差就越小，精度就越大，这样就做到了保留激活值较大的那些权重的效果

- 寻找 $scale$ --即寻找每一个输出通道的最优的缩放因子 s

寻找最优 scaling

为了同时考虑显著权重和非显著权重，论文通过最小化某层量化前后的差值来寻找最优 scaling。

$$s^* = \arg \min_s \mathcal{L}(s), \quad \mathcal{L}(s) = \|Q(W \cdot s)(s^{-1} \cdot X) - WX\|$$

s 是 per-channel scaling 因子， $s^{-1} \cdot X$ 可以被融合至前面的算子

为了让量化过程更稳定，论文为最优 scaling 定义了一个搜索空间，由于显著权重由 activation 大小来决定，因此搜索空间定义如下：

$$s = s_{\mathbf{X}}^{\alpha}, \quad \alpha^* = \arg \min_{\alpha} \mathcal{L}(s)$$

s 只与 activation 大小 $s_{\mathbf{X}}$ 有关，论文通过一个超参 α 来平衡显著通道权重和非显著通道权重。另外使用 weight clipping 来减少 Δ' 从而减少量化误差，进而减少 MSE 误差。

OPT / PPL↓		1.3B	2.7B	6.7B	13B	30B
FP16	-	14.62	12.47	10.86	10.13	9.56
INT3 g128	RTN	119.47	298.00	23.54	46.04	18.80
	1% FP16	16.91	13.69	11.39	10.43	9.85
	$s = 2$	18.63	14.94	11.92	10.80	10.32
	AWQ	16.32	13.58	11.39	10.56	9.77

Table 3. AWQ protects salient weights and reduces quantization error by using a scaling-based method. It consistently outperforms Round-to-nearest quantization (RTN) and achieves comparable performance as mixed-precision (1% FP16) while being more hardware-friendly.

Table 3 可以发现，AWQ 方案效果要优于固定的 $scaling=2$ 情况，另外相比混合精度量化（1% 显著通道权重使用 FP16）效果相当，但是会硬件更友好。

上图第一个公式：定义一个损失函数来衡量量化前后的差异

目标就是找到使这个损失函数值最小的这么一个 s

然后我们需要定义一个搜索空间（因为 s 作为一个向量其搜索空间如果不做优化的话会非常大）

$$s = s_{\mathbf{X}}^{\alpha}, \quad \alpha^* = \arg \min_{\alpha} \mathcal{L}(s)$$

s_x :为基准缩放向量，如果一个通道的激活值越大，那么我们就将这个 s_x 的值设置的就越大

α :这里是一个全局的超参数，用来控制缩放的程度

通过基于每个通道的激活值统计设置 s_x 然后在 α 的预定范围内均匀的选几个值，通过对比损失函数的大小来找出最优的这么一个 s

AWQ和GPTQ的异同点：

相同点：

1. 都是PTQ后训练量化方法
2. 目的相同：都是使最后的损失函数的误差最小

不同点：

1. 核心思想不同：一个是针对于OBQ二阶Hessian矩阵的优化，一个是基于激活的权重重要性进行的优化

分布式推理

因为单块GPU难以满足推理所需的显存和算力，因而以来GPU集群来协同运行

1. 大模型并行推理方法

- o Pipeline Parallelism--流水线并行

以Gpipe为例

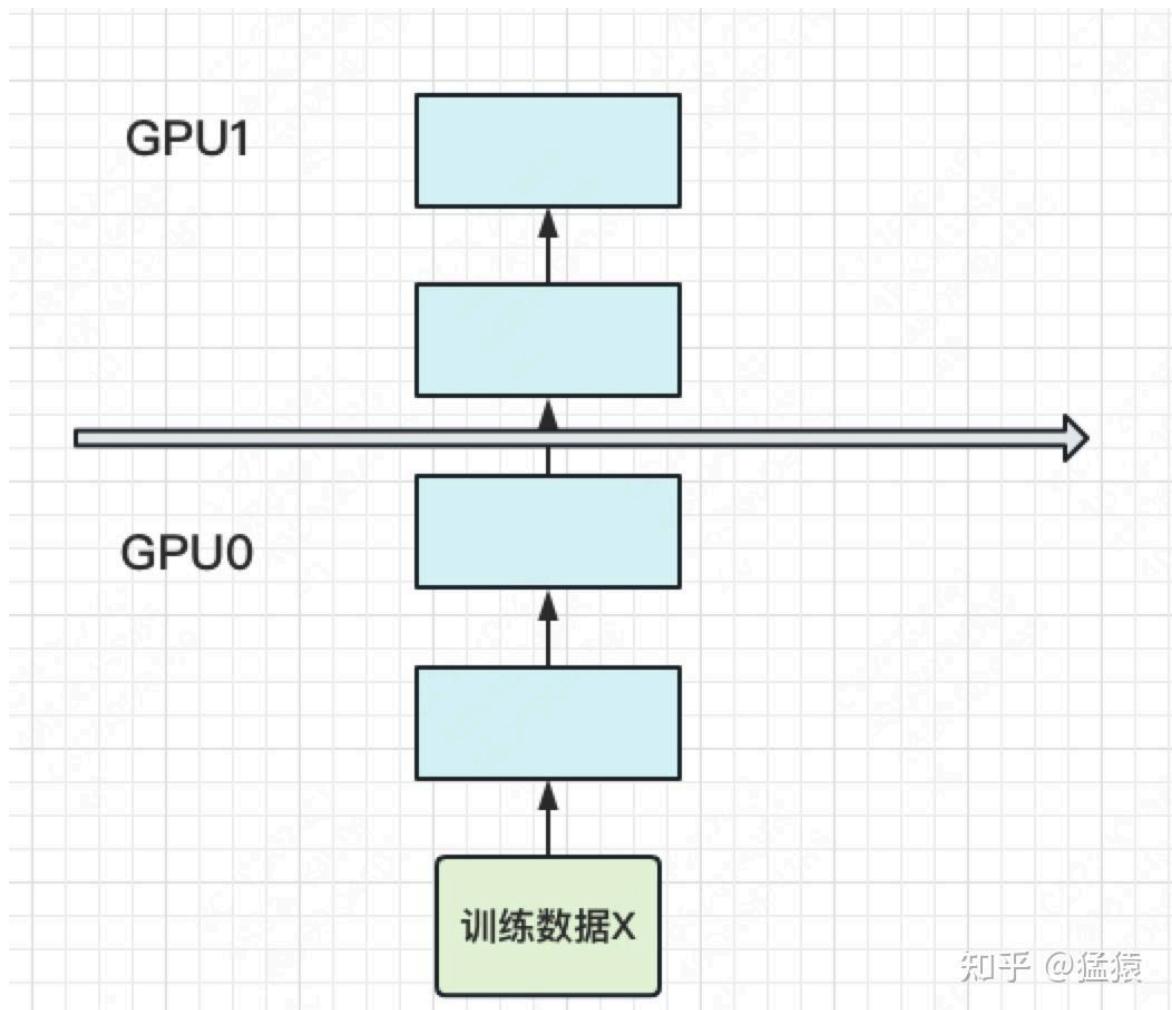
- 优化目标：

GPU上的内存限制：更大的模型意味着更多的训练数据，增加了每块GPU的内存压力

GPU间的带宽限制：数据在每块CPU之间传输的网络通讯开销

- 模型并行：

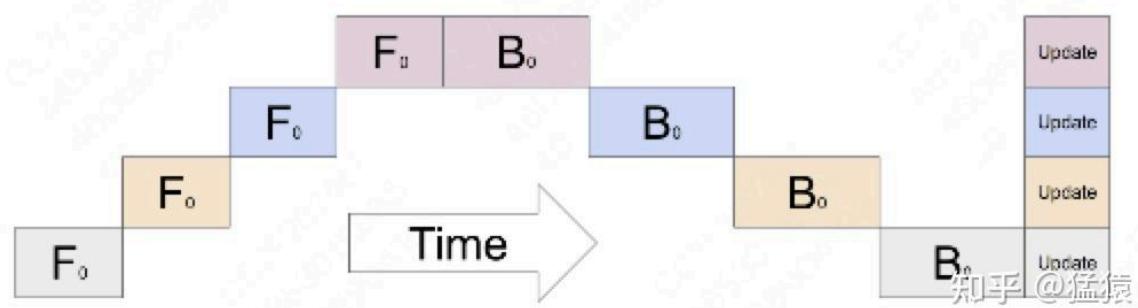
把模型的每个层，每一层放在一块GPU上



知乎 @猛猿

模型做一轮forward和backward的过程：

每一行 (y轴) 代表一个GPU, 从左到右为时间的流逝



问题：

1. GPU利用率不够：每一轮总有GPU在空转

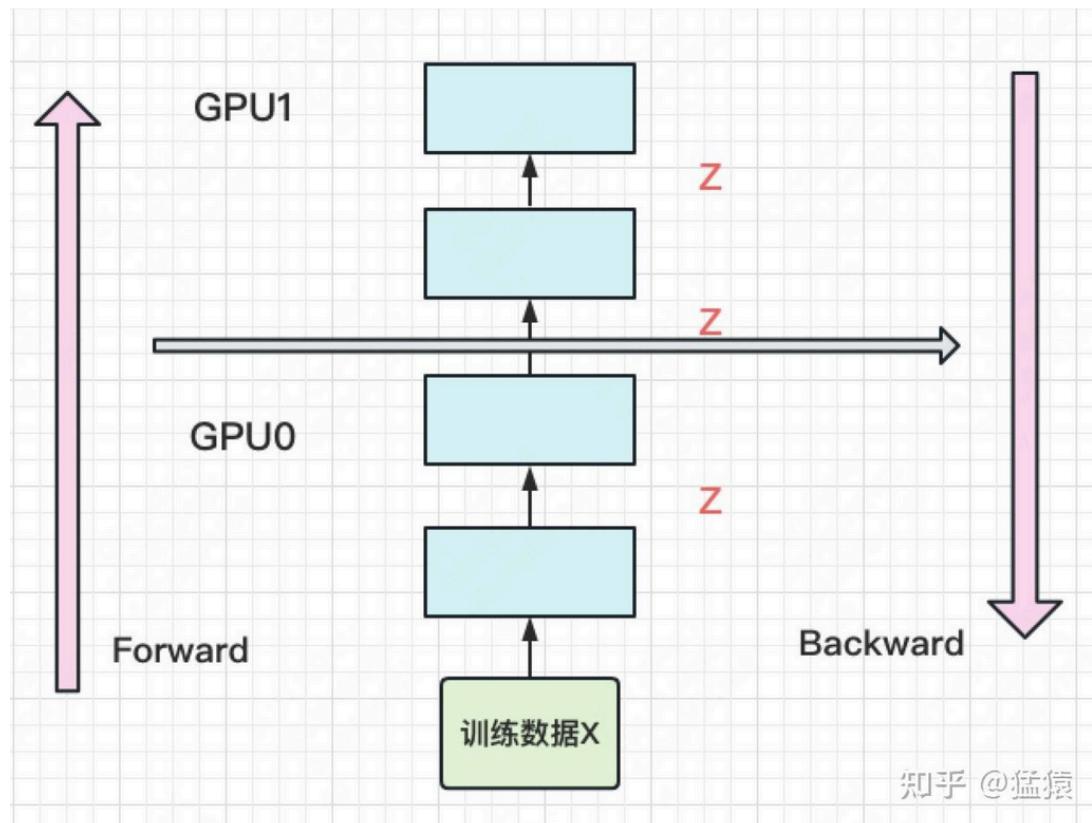
如图，阴影部分所表示的时间段里，总有GPU在空转。在Gpipe中，将阴影部分定义为bubble。我们来计算一下bubble。假设有 K 块GPU，而单块GPU上做一次forward和backward的时间为： $t_{fb} = (t_f + t_b)$ 。则：

- 图中灰色长方形的整体面积为： $K * Kt_{fb}$ （宽= K ，长= Kt_{fb} ）
- 图中实际在做forward和backward的面积为： Kt_{fb}
- 图中阴影部分的面积为： $K * Kt_{fb} - Kt_{fb} = (K - 1)Kt_{fb}$
- 图像阴影部分的占比为： $(K - 1)Kt_{fb} / K * Kt_{fb} = (K - 1) / K$

则我们定义出bubble部分的时间复杂度为： $O(\frac{K-1}{K})$ ，当 K 越大，即GPU的数量越多时，空置的比例接近1，即GPU的资源都被浪费掉了。因此这个问题肯定需要解决。

2. 中间结果占用大量内存

个人理解就是随着网络层数的增加，因为前一层要依靠后一层的中间结果，这样就造成在反向传播后期的层需要存储大量的中间结果



在做backward计算梯度的过程中，我们需要用到每一层的中间结果 z 。假设我们的模型有 L 层，每一层的宽度为 d ，则对于每块GPU，不考虑其参数本身的存储，额外的空间复杂度为 $O(N * \frac{L}{K} * d)$ 。从这个复杂度可以看出，随着模型的增大， N ， L ， d 三者的增加可能会平滑掉 K 增加带来的GPU内存收益。因此，这也是需要优化的地方。

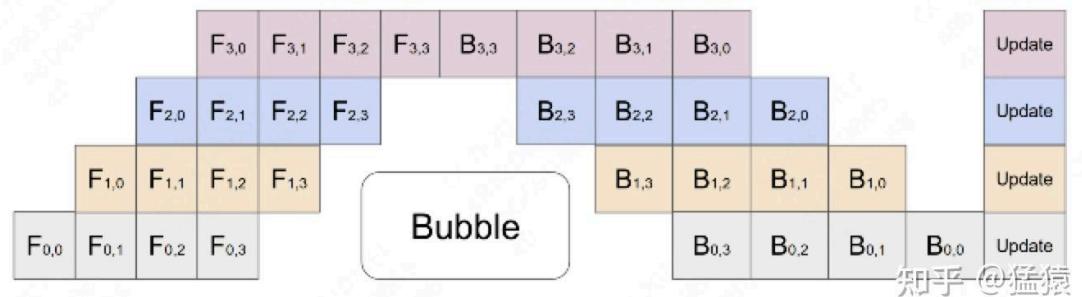
■ 流水线并行：

1. 切分micro-batch--解决上面所说的gpu利用率不高的问题

在模型并行的基础上进一步引入数据并行的办法，把原先的数据划分成若干个batch (micro-batch)，送入GPU进行训练

类比CPU的流水线：

第一个下标表示GPU编号，第二个下标表示micro-batch编号



整体面积: $K \times (K + M - 1) f_{tb}$ (每送入一个新的micro-batch, 流水线整体完成时间增加1个时间单位)

实际在做的面积: $M \times K \times f_{tb}$

阴影部分的面积 (上面俩作差): $K \times f_{tb} \times (K-1)$

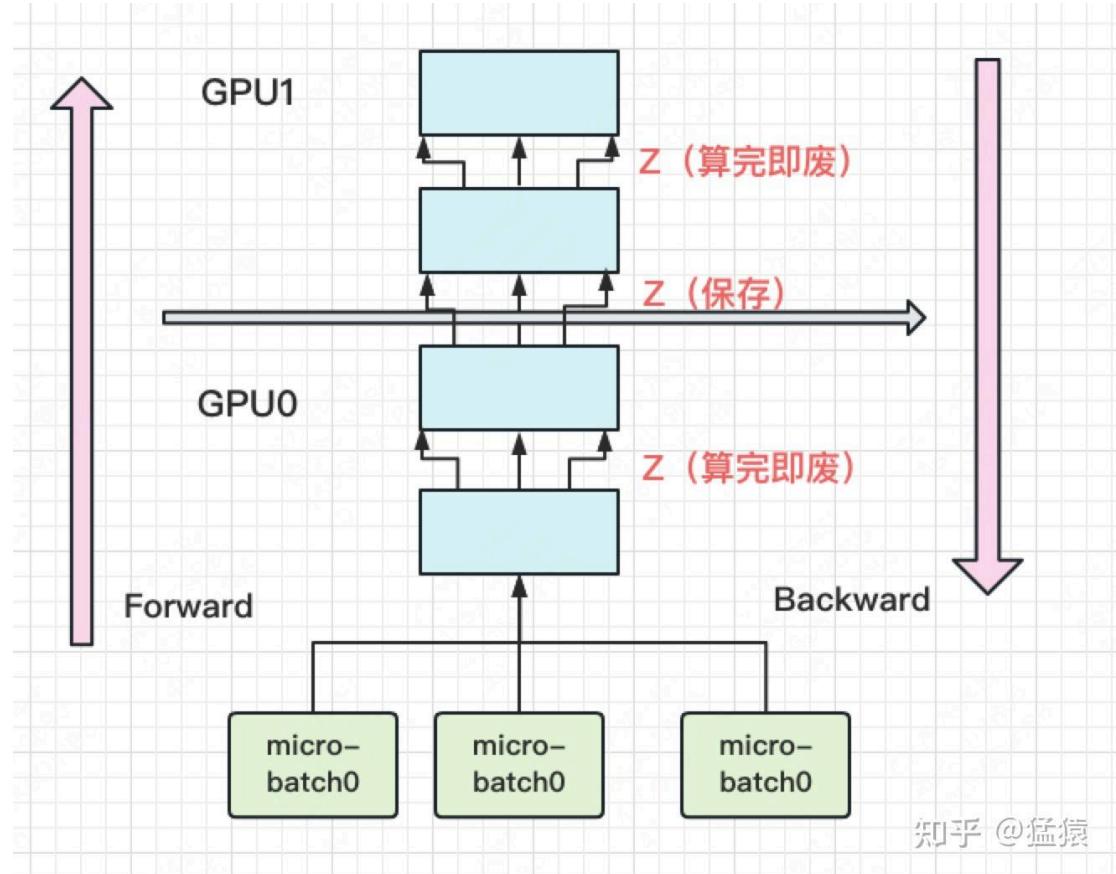
阴影部分的占比: $K-1/K+M-1$

其中, 第一个下标表示GPU编号, 第二个下标表示micro-batch编号。假设我们将mini-batch划分为M个, 则流水线并行下, bubble的时间复杂度为: $O(\frac{K-1}{K+M-1})$ (推导过程略, 可参照第二部分的bubble推导流程)。Gpipe通过实验证明, 当 $M \geq 4K$ 时, bubble产生的空转时间占比对最终训练时长影响是微小的, 可以忽略不计。

将batch切好, 并逐一送入GPU的过程, 就像一个流水生产线一样 (类似于CPU里的流水线), 因此也被称为Pipeline Parallelism。

2. re-materialization/active checkpoint--解决上面说的中间结果占用大量内存的问题

用时间换空间, 几乎不存中间结果, 等到backward的时候再重新算一遍forward



知乎 @猛猿

空间计算：

因为每一块GPU上面同时只能存在一个micro-batch，所以需要容纳的空间就是

每块（一共K块GPU）上一共多少层： L/K ，每一层宽为 d （这里的 d 就是每一层产生的激活量的个数，就这么理解）

一共有 N 这么多的起始数据， M 个micro-batch，所以每个micro-batch有 N/M 这么多的数据（每个GPU一个micro-batch处理这么多的数据）

所以在最后一个micro-batch进行的时候（空间利用最多的情况下），每个GPU存储 $N+N/M \times L/K \times d$ 这么多的空间

每块GPU上，我们只保存来自上一块的最后一层输入 z ，其余的中间结果我们算完就废。等到backward的时候再由保存下来的 z 重新进行forward来算出。

现在我们来计算每块GPU峰值时刻的内存：

每块GPU峰值时刻存储大小 = 每块GPU上的输入数据大小 + 每块GPU在forward过程中的中间结果大小

每块GPU上固定需要保存它的起始输入，我们记起始输入为 N （即mini-batch的大小）。

每个micro-batch是流水线形式进来的，算完一个micro-batch才算下一个。在计算一个micro-batch的过程中，我们会产生中间变量，它的大小为 $\frac{N}{M} \times \frac{L}{K} \times d$ （其中 M 为micro-batch个数）。

因此，每块GPU峰值时刻的空间复杂度为 $O(N + \frac{N}{M} \times \frac{L}{K} \times d)$

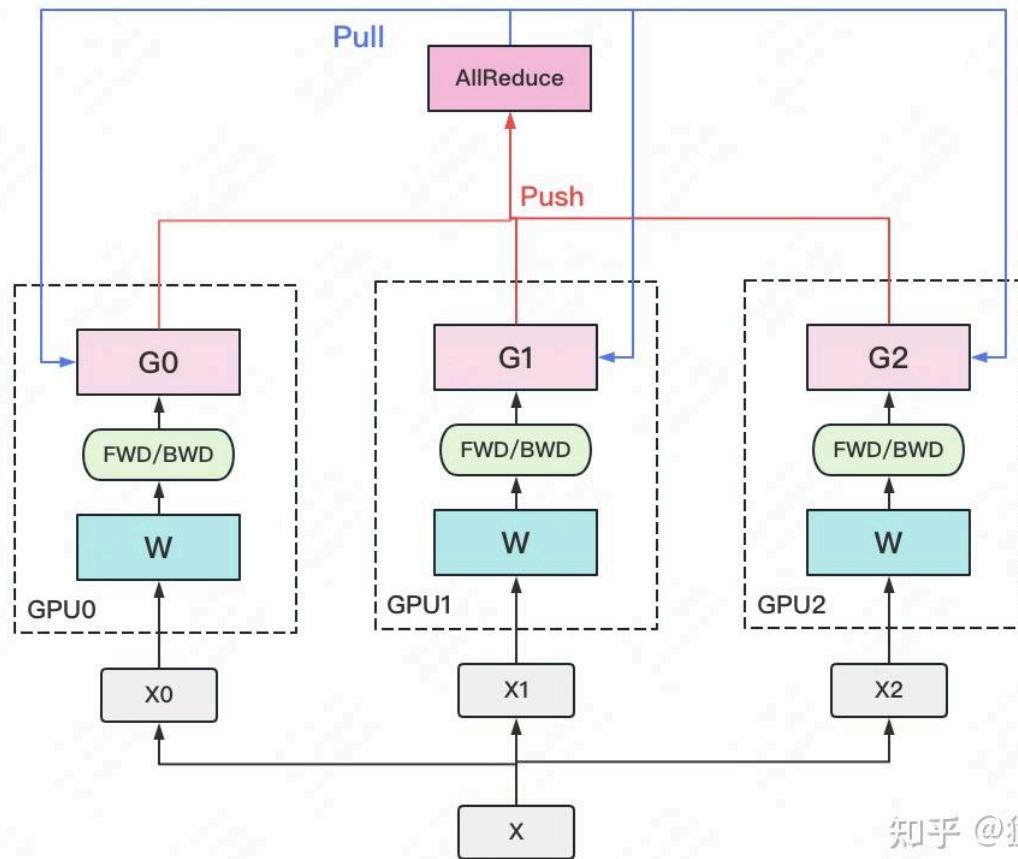
将其与朴素模型并行中的GPU空间复杂度 $O(N \times \frac{L}{K} \times d)$ 比较，可以发现，由于采用了micro-batch的方法，当 L 变大时，流水线并行相比于朴素模型并行，对GPU内存的压力显著减小。

- Data Parallelism--数据并行

在各个GPU上都拷贝一份完整模型，各自吃一份数据，算一份梯度，最后对梯度进行累加来更新整体模型

- DP (Data Parallelism): 一般用于单机多卡的模型

整体结构：



知乎 @猛猿

数据并行流程：

一个经典数据并行的过程如下：

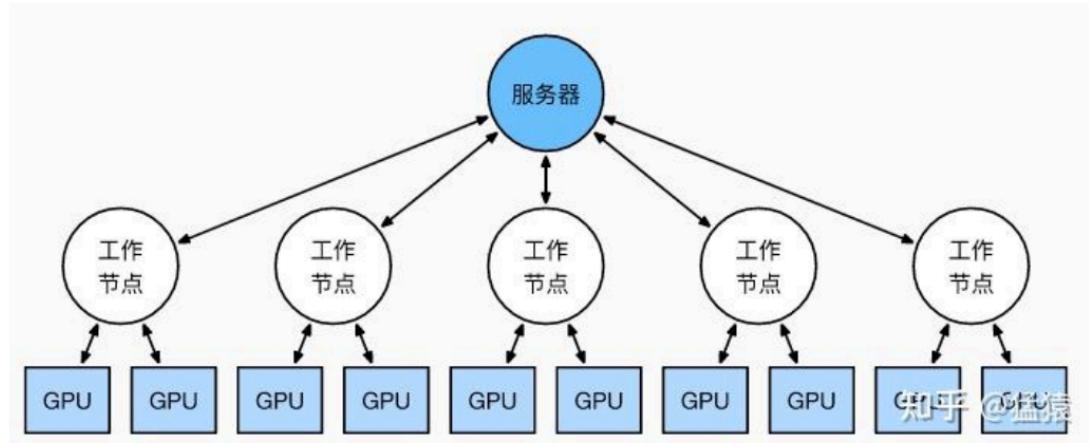
- 若干块计算GPU，如图中GPU0~GPU2；1块梯度收集GPU，如图中AllReduce操作所在GPU。
- 在每块计算GPU上都拷贝一份完整的模型参数。
- 把一份数据X（例如一个batch）均匀分给不同的计算GPU。
- 每块计算GPU做一轮FWD和BWD后，算得一份梯度G。
- 每块计算GPU将自己的梯度push给梯度收集GPU，做聚合操作。这里的聚合操作一般指**梯度累加**。当然也支持用户自定义。
- 梯度收集GPU聚合完毕后，计算GPU从它那pull下完整的梯度结果，用于更新模型参数W。更新完毕后，计算GPU上的模型参数依然保持一致。
- 聚合再下发梯度的操作，称为**AllReduce**。

一个DP的经典框架：参数服务器

前文说过，实现DP的一种经典编程框架叫“参数服务器”，在这个框架里，**计算GPU称为Worker，梯度聚合GPU称为Server**。在实际应用中，为了尽量减少通讯量，一般可选择一个Worker同时作为Server。比如可把梯度全发到GPU0上做聚合。需要再额外说明几点：

- 1个Worker或者Server下可以不止1块GPU。
- Server可以只做梯度聚合，也可以梯度聚合+全量参数更新一起做

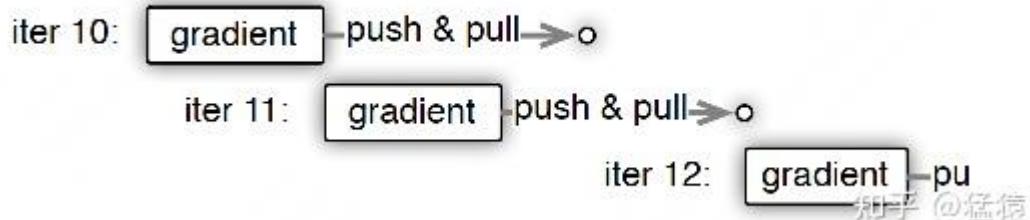
在参数服务器的语言体系下，DP的过程又可以被描述下图：



图片来源：https://zh.d2l.ai/chapter_computational-performance/parameterserver.html

缺点：存储开销大（每块GPU上都存了一份完整的模型），通讯开销大（Server需要和每一个Worker进行梯度传输，当Server和Worker不在一台机器上时，Server带宽会成为整个系统的计算效率瓶颈）
(注意下面的方法主要是为了优化这两个缺点的)

梯度异步更新--为了解决上面**Server干活的时候Worker不干活**的问题：

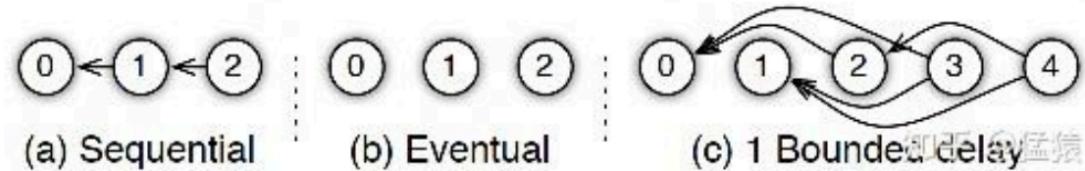


上图是某个Worker的计算顺序：即在第10轮计算的时候，计算完Worker将梯度push给Server，但是Worker不会等着把聚合梯度拿回来，而是继续拿着旧的梯度吃新的数据继续11轮的计算，这样就**保证了在Server计算聚合梯度的时候，底下的Worker不会闲着**

上图是**延迟为1**的异步更新，保证第12轮的时候，该Worker可以不拿回11轮的聚合梯度，但是需要拿回第10轮的聚合梯度，但是这时可能第11轮的聚合梯度也已经计算完成了这时就可以连着11轮一并拿回来如果11轮没计算好，那么就拿第10轮的就行

延迟情况：

参数服务器的框架下，延迟的步数也可以由用户自己决定，下图分别刻画了几种延迟情况：



图片来源：<https://web.eecs.umich.edu/~mosharaf/Readings/Parameter-Server.pdf>

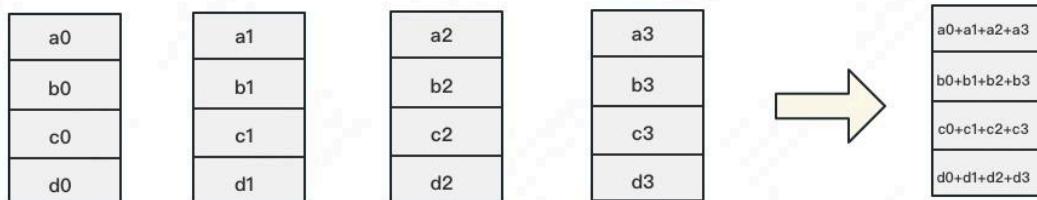
- (a) 无延迟
- (b) 延迟但不指定延迟步数。也即在迭代2时，用的可能是老权重，也可能是新权重，听天由命。
- (c) 延迟且指定延迟步数为1。例如做迭代3时，可以不拿回迭代2的梯度，但必须保证迭代0、1的梯度都已拿回且用于参数更新。

梯度异步更新的优点就是提高了硬件利用率，但是引入了梯度延迟，可能会影响模型的收敛速度和最终精度

- DDP (Distributed Data Parallelism): 更加通用，既能多机也能单机

Ring-AllReduce--解决刚才DP中遇到的通信负载不均的问题

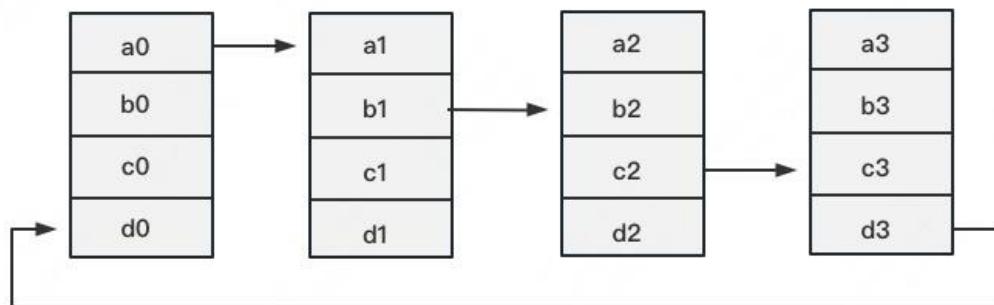
目标：



知乎 @猛猿

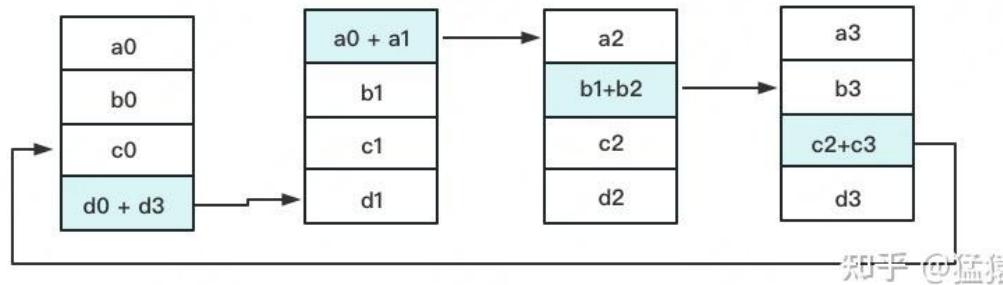
1. Reduce-Scatter

定义网络拓扑关系，每个GPU只和其相邻的两块GPU通讯，每次发送对应位置的数据进行累加
每次累加形成一个拓扑环，如下图

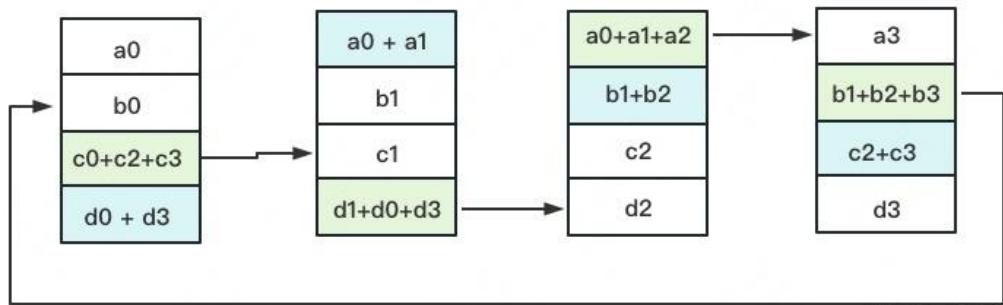


知乎 @猛猿

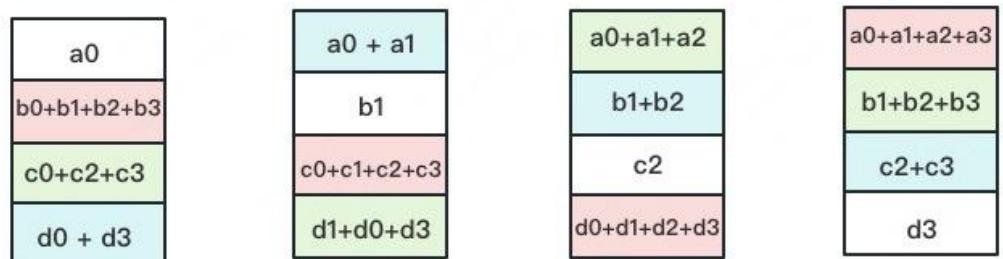
一次累加完成后，被更新的数据块将成为下一次更新的起点



知乎 @猛猿



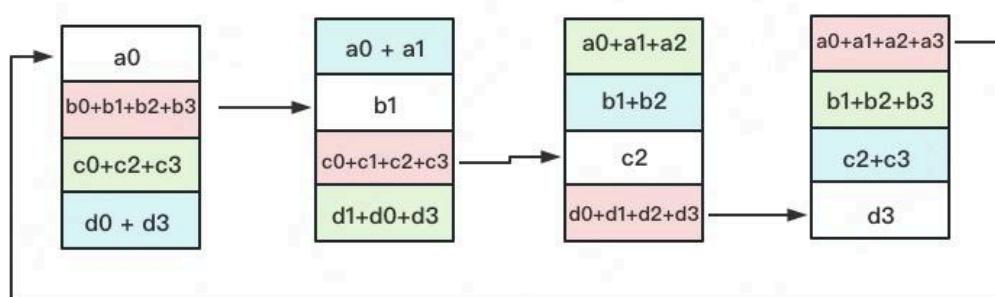
知乎 @猛猿



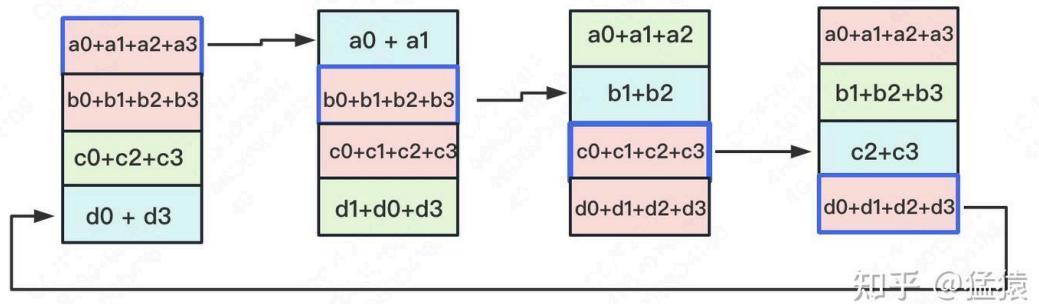
知乎 @猛猿

3轮更新之后，出现全部目标数据（红色块）

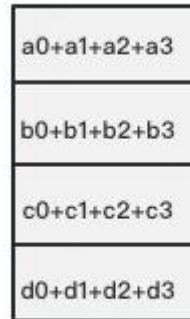
2. All-Gather--将刚才红色块数据聚合在一起



知乎 @猛猿



还是这样，三轮后每一块上面都是完整的数据



通信量分析

假设模型参数W的大小为 Φ ，GPU个数为 N 。则梯度大小也为 Φ ，每个梯度块的大小为 $\frac{\Phi}{N}$ 对单卡GPU来说（只算其send通讯量）：

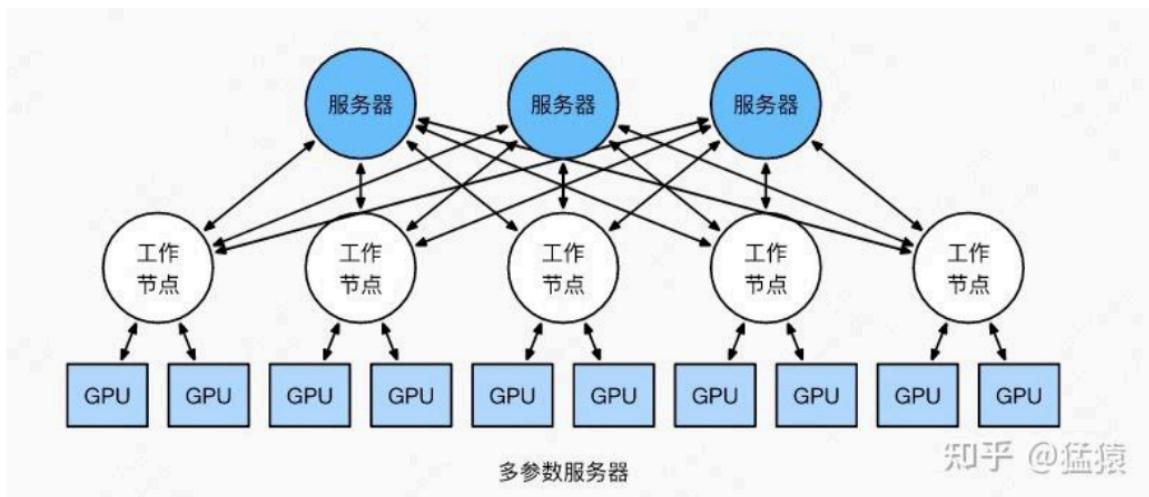
- Reduce-Scatter阶段，通讯量为 $(N - 1) \frac{\Phi}{N}$
- All-Gather阶段，通讯量为 $(N - 1) \frac{\Phi}{N}$

单卡总通讯量为 $2(N - 1) \frac{\Phi}{N}$ ，随着N的增大，可以近似为 2Φ 。全卡总通讯量为 $2N\Phi$

而对前文的DP来说，它的Server承载的通讯量是 $N\Phi$ ，Workers为 $N\Phi$ ，全卡总通讯量依然为 $2N\Phi$ 。虽然通讯量相同，但搬运相同数据量的时间却不一定相同。DDP把通讯量均衡负载到了每一时刻的每个Worker上，而DP仅让Server做勤劳的搬运工。当越来越多的GPU分布在距离较远的机器上时，DP的通讯时间是会增加的。

多Server下的参数服务器

但这并不说明参数服务器不能打（有很多文章将参数服务器当作old dinosaur来看）。事实上，参数服务器也提供了多Server方法，如下图：



在多Server的模式下，进一步，每个Server可以只负责维护和更新某一块梯度（也可以某块梯度+参数一起维护），此时虽然每个Server仍然需要和所有Worker通讯，但它的带宽压力会小非常多。经过调整设计后，依然可以用来做DDP。虽然这篇文章是用递进式的方式来介绍两者，但不代表两者间一定要决出优劣。我想表达的观点是，方法是多样性的。对参数服务器有兴趣的朋友，可以阅读参考的第1个链接。

- ZeRO:

为了解决上面DP遇到的显存开销的问题，思想就是用通讯换显存

存储消耗：

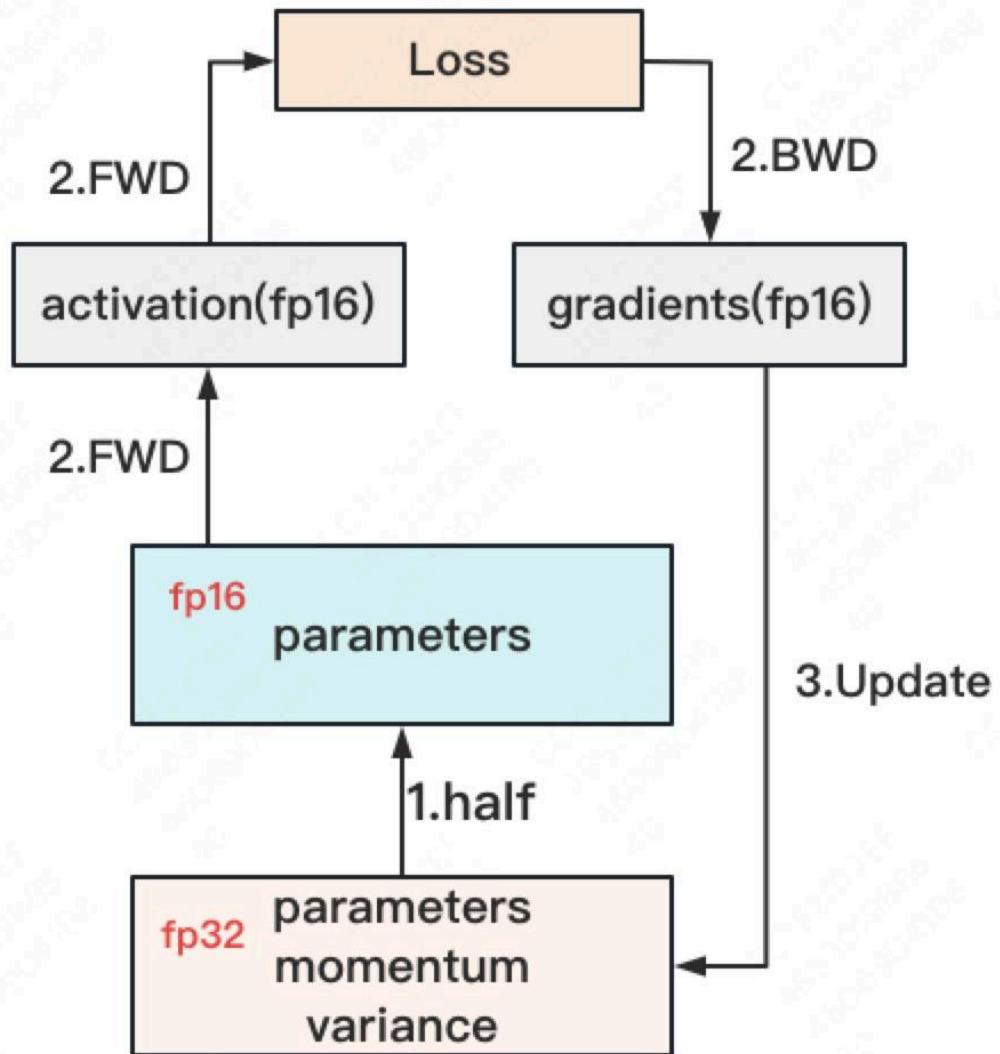
1. Model States: 必须存储的内容

1. optimizer states--优化器状态
2. gradients:模型梯度
3. parameters:模型参数

2. Residual States: 训练过程中额外产生的内容并非模型所必须的

1. activation:激活值
2. temporary buffers: 临时存储（例如把梯度发送到某块GPU上时做总聚合时产生的存储）
3. unusable fragment memory: 碎片化的存储空间（会造成即使存储空间是够的但是相关请求fail的问题）

精度混合训练：



知乎 @猛猿

- 存储一份fp32的parameter, momentum和variance (统称model states)
- 在forward开始之前, 额外开辟一块存储空间, 将fp32 parameter减半到fp16 parameter。
- 正常做forward和backward, 在此之间产生的activation和gradients, 都用fp16进行存储。
- 用fp16 gradients去更新fp32下的model states。
- 当模型收敛后, fp32的parameter就是最终的参数输出。

存储大小

现在，我们可以来计算模型在训练时需要的存储大小了，假设模型的参数W大小是 Φ ，以 byte为单位，存储如下：

	分类	大小	大小总计
必存	parameter (fp32)	4 Φ	12 Φ
	momentum(fp32)	4 Φ	
	variance (fp32)	4 Φ	
中间值	parameter (fp16)	2 Φ	4 Φ
	gradients (fp15)	2 Φ	
总计			16 Φ

知乎 @猛猿

因为采用了Adam优化，所以才会出现momentum和variance，当然你也可以选择别的优化办法。因此这里为了更通用些，记模型必存的数据大小为 $K\Phi$ 。因此最终内存开销为：

$$2\Phi + 2\Phi + K\Phi$$

另外，这里暂不将activation纳入统计范围，原因是：

- activation不仅与模型参数相关，还与batch size相关
- activation的存储不是必须的。存储activation只是为了在用链式法则做backward的过程中，计算梯度更快一些。但你永远可以通过只保留最初的输入X，重新做forward来得到每一层的activation（虽然实际中并不会这么极端）。
- 因为activation的这种灵活性，纳入它后不方便衡量系统性能随模型增大的真实变动情况。因此在这里不考虑它，在后面会单开一块说明对activation的优化。

ZeRO-DP--对model states的显存优化

知道了什么东西会占存储，以及它们占了多大的存储之后，我们就可以来谈如何优化存储了。

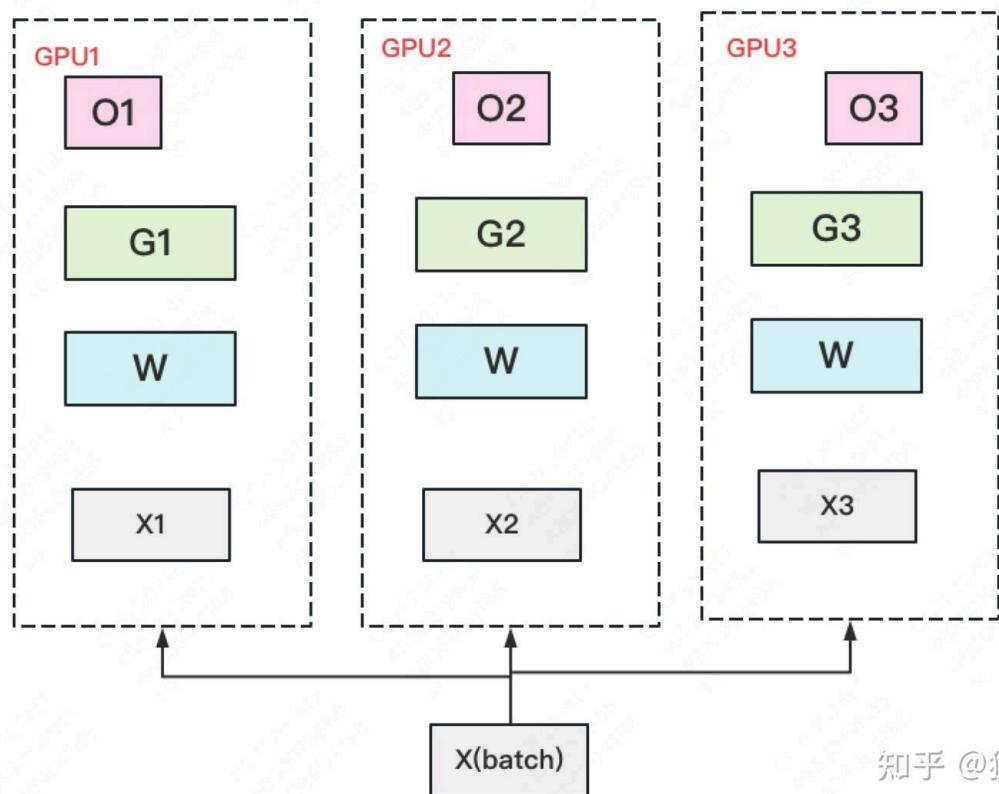
注意到，在整个训练中，有很多states并不会每时每刻都用到，举例来说；

- Adam优化下的optimizer states只在最终做update时才用到
- 数据并行中，gradients只在最后做AllReduce和updates时才用到
- 参数W只在做forward和backward的那一刻才用到
- 诸如此类

所以，ZeRO想了一个简单粗暴的办法：**如果数据算完即废，等需要的时候，我再想办法从个什么地方拿回来，那不就省了一笔存储空间吗？**

1. P_{os} :优化状态分割

从optimizer state开始优化：将optimizer state分成若干份，每块GPU各自维护一份，这样就减少了一部分开销



复习一下这里用的是数据并行的方法：

1. DP整体的结构：每块GPU上都复制一份完整的参数，然后将整体数据分为3份，每块GPU上吃一份，做完forward和backward后（这里采用的是上面说的混合梯度训练）各得一份梯度

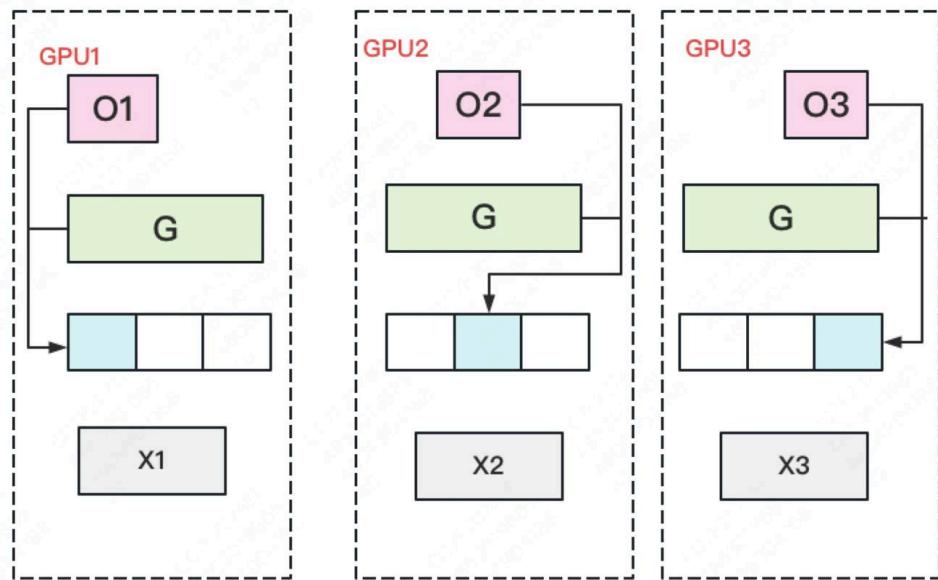
2. DDP更新梯度的方法：reduce-scatter+all-gather

（这里只需要做一次reduce-scatter就可以了，产生的通信量为 phi ）

这里是让各GPU之间通信从而得到完整的梯度

3. 得到完整的梯度就可以用优化器对权重进行更新了

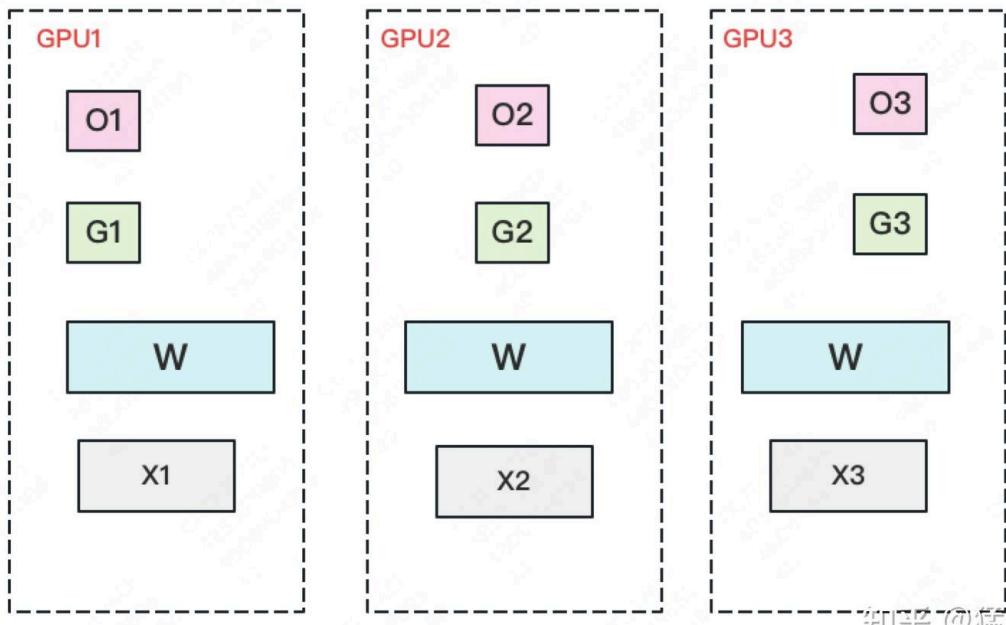
因为每块GPU之间只保留了一部分优化器，所以只能将每块GPU上的对应的权重进行更新（下图中对应的蓝色的部分）



知乎 @猛猿

4. 然后对其使用一次All-gather将完整的权重更新到每块GPU上面
2. $P_{os} + P_g$; 优化状态与梯度分割

现在将梯度也拆开，每个GPU上面各维护一块梯度



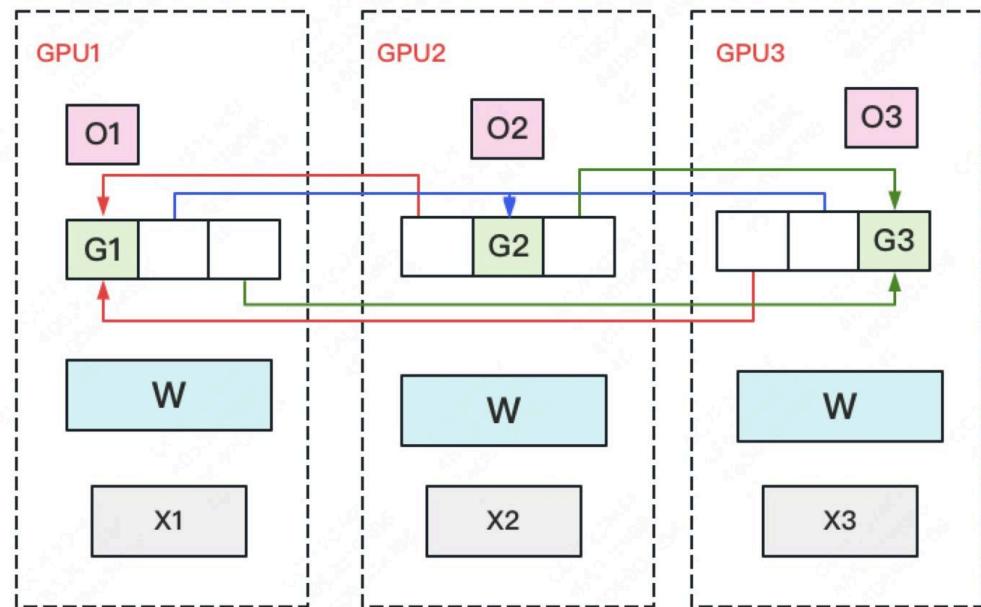
知乎 @猛猿

整体流程：

1. DP整体的结构：每块GPU上都复制一份完整的参数，然后将整体数据分为3份，每块GPU上吃一份，做完forward和backward后（这里采用的是上面说的混合梯度训练）各得一份梯度
2. DDP更新梯度的方法：reduce-scatter+all-gather

这里有所不同，因为每一块得到的是一份梯度，上面方法是把各自得到的所有梯度传输到其他的GPU上面，每块GPU上聚合的梯度都是完整梯度

而这里我们把梯度重新划分了，比如对于GPU1, 它只需要得到其他两块GPU对应G1位置上面的梯度，从而在**GPU1**上聚合的完整梯度只是**G1**（即上面完整梯度的一部分）



做完对应梯度加总后，白色梯度可以移除，绿色梯度为最终汇总梯度

知乎 @猛猿

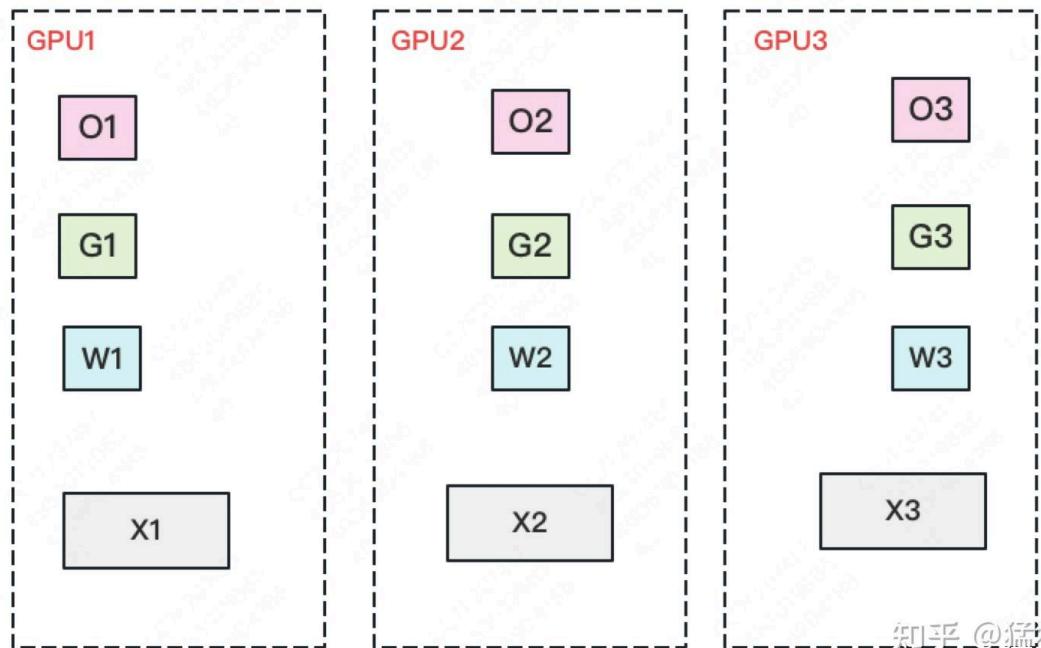
3. 然后权重更新

每块GPU用自己那块优化器和梯度去更新得到一块更新完毕的权重，然后再经过all-gather将别的GPU算好的权重汇聚，这样每块GPU最终得到的就是完整的权重了

3. $P_{os} + P_g + P_p$: 优化状态，梯度与参数分割

现在我们把参数W也切了

所以现在每块GPU只维护，各自对应的那部分优化器，梯度，权重



知乎 @猛猿

1. 每个部分只保存部分的参数W（不同之处），然后将整体数据分为3份，每块GPU上面吃一份
2. 做forward前，对参数W做一次All-gather，取回分布在别的GPU上面的W，得到一份完整的W，**forward**做完立即把不是自己维护的W抛弃
3. 做backward前，对参数W做一次All-gather，取回完整的W，**backward**做完立即把不是自己维护的W抛弃

这里做forward和backward用的都是上面说的混合精度训练法

4. 梯度更新：对上面梯度进行一次reduce-scatter从别的GPU上聚合自己维护的那部分梯度，聚合操作完成后立即把不是自己维护的那部分梯度抛弃

5. 更新W：用自己维护的优化器和梯度更新自己维护的那部分W

显存和通信量：

最终我们用1.5倍的通信量换回120倍的显存

	显存	显存 (单位GB) $K = 12, \Phi = 7.5B, N_d = 64$	单卡通讯量
朴素DP	$(2 + 2 + K)\Phi$	120GB	2Φ
P_{os}	$(2 + 2 + \frac{K}{N_d})\Phi$	31.4GB	3Φ
$P_{os} + P_g$	$(2 + \frac{2 + K}{N_d})\Phi$	16.6GB	2Φ
$P_{os} + P_g + P_p$	$(\frac{2 + 2 + K}{N_d})\Phi$	1.9GB	3Φ

知乎 @猛猿

ZeRO-R--对Residual States的优化

1. 激活值

前面说过，对activation的存储是灵活的。不像optimizer states, gradients和parameters对模型更新是必须的，activation只是起到加速梯度计算的作用。因此，在哪几层保存activation，保存哪些activation都是可以灵活设置的。同样，我们也可以仿照以上切割方式，每块GPU上只维护部分的activation，需要时再从别的地方聚合过来就行。需要注意的是，activation对显存的占用一般会远高于模型本身，通讯量也是巨大的，所以这块要灵活、有效地实验设计。

2. 临时存储

使用固定大小的临时存储

3. 碎片化的存储空间

设置机制，对碎片化的存储空间重新整合，整理出连续的存储空间

ZeRO-Offload与ZeRO-Infinity

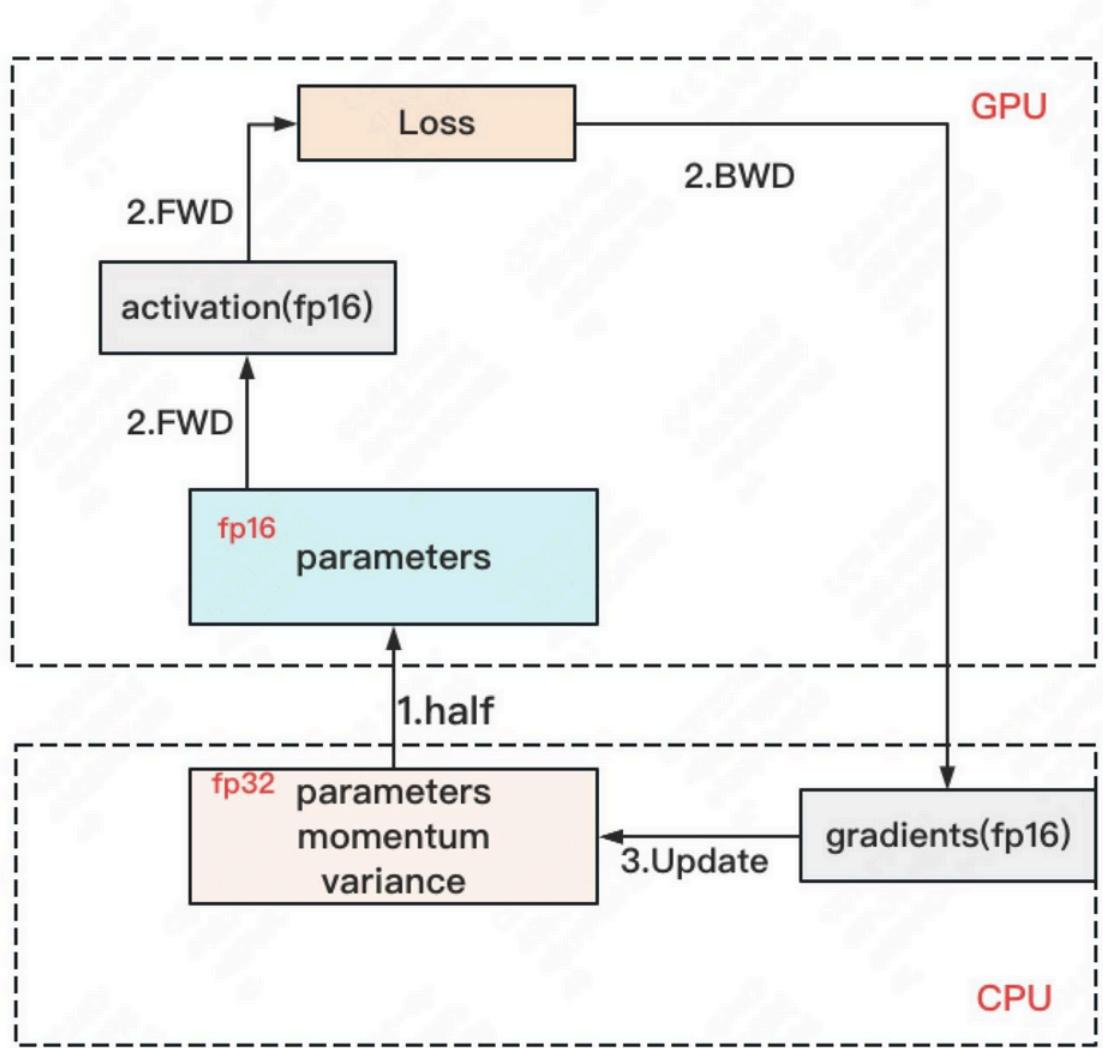
ZeRO-Offload：显存不够，内存来凑

将存储的大头卸载到CPU上面，计算部分还是在GPU上面

具体做法：

- **forward**和**backward**计算量高，因此和它们相关的部分，例如参数W (fp16) , activation, 就全放入GPU。
- **update**的部分计算量低，因此和它相关的部分，全部放入CPU中。例如W(fp32), optimizer states (fp32) 和gradients(fp16)等。

具体切分如下图：



ZeRO-infinity: 同理

- Tensor Parallelism--张量并行

以NVIDIA的张量模型TP为例：把模型的参数纵向切开，放到不同的反而GPU上面进行独立计算，然后再做聚合

全文结构如下：

- 一、切分权重的两种方法
- 二、MLP层
- 三、self-attention层
- 四、Embedding层
- 五、Cross-entropy层
- 六、经典并行：TP + DP (Megatron + ZeRO)
- 七、实验效果与GPU利用率
- 八、参考

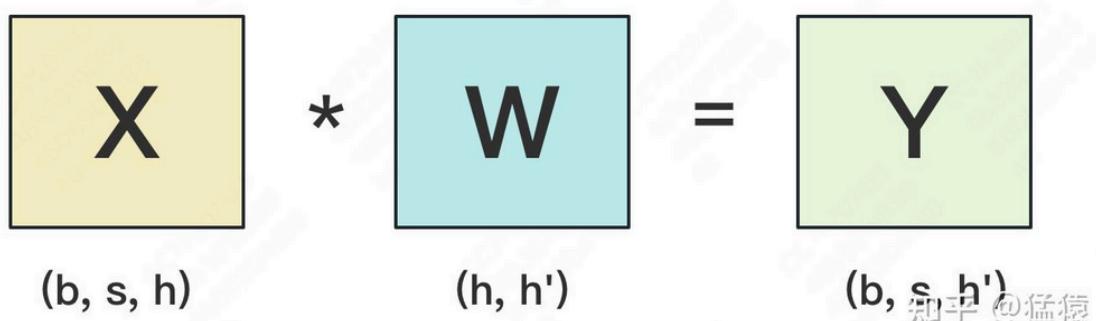
1. 切分权重

三 目录

设输入数据为 X ，参数为 W 。 X 的维度 = (b, s, h) ， W 的维度 = (h, h') 。其中：

- b : batch_size，表示批量大小
- s : sequence_length，表示输入序列的长度
- h : hidden_size，表示每个token向量的维度。
- h' : 参数 W 的hidden_size。

则每次forward的过程如下：



为画图方便，图中所绘是 $b=1$ 时的情况。

理解： b 可以理解有 b 个这么大的 X 矩阵， s 可以理解成 X 矩阵包括多少个向量，理解成矩阵的横， h 是向量的维度，可以理解成纵，那么 W 矩阵的作用就是把 X 矩阵从 h 维映射成 h' 维

如果 W 矩阵太大的话单卡装不下，那么我们就需要对 W 矩阵进行切分

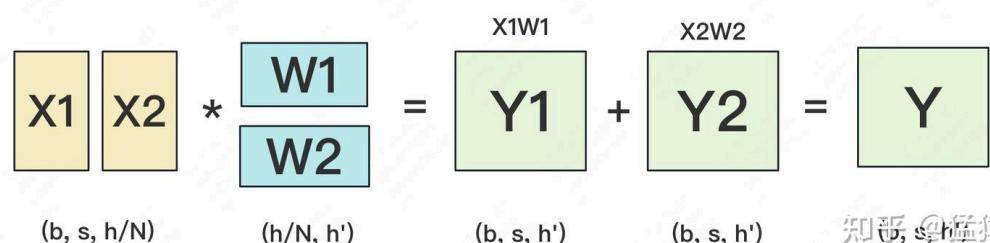
1. 按行切分

1. forward

N代表GPU的数量，下面介绍的是N=2的情况

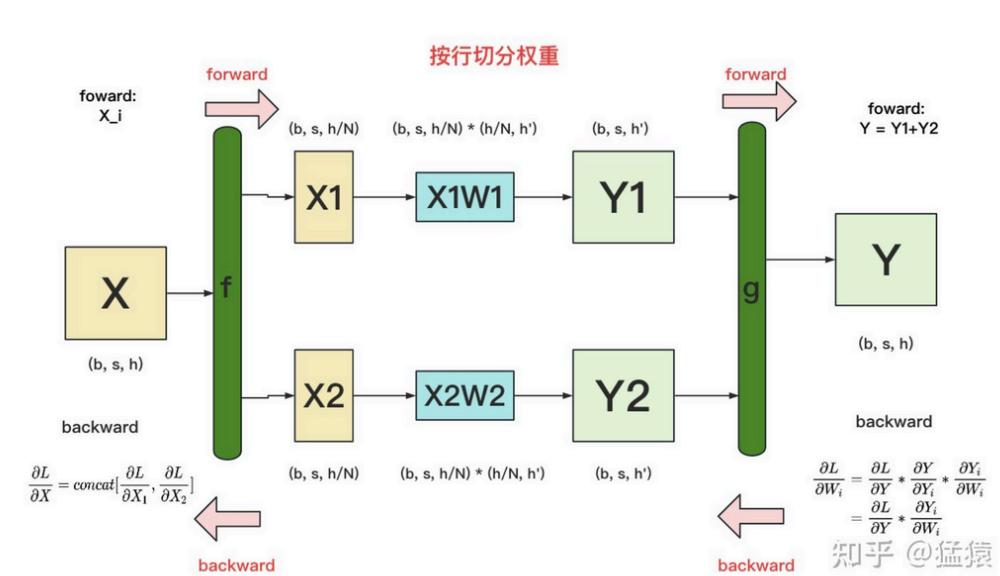


知乎 @猛猿



知乎 @猛猿

2. backward



- **f** 和 **g**：分别表示两个算子，每个算子都包含一组forward + backward操作。forward操作已讲过，不再赘述。
- 图中的每一行，表示单独在一块GPU上计算的过程
- **g 的backward**：假定现在我们要对 W_i 求梯度，则可推出 $\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial Y} * \frac{\partial Y}{\partial Y_i} * \frac{\partial Y_i}{\partial W_i} = \frac{\partial L}{\partial Y} * \frac{\partial Y_i}{\partial W_i}$ ，也就是说，只要把 $\frac{\partial L}{\partial Y}$ 同时广播到两块GPU上，两块GPU就可以独立计算各自权重的梯度了。
- **f 的backward**：在上图中，我们只画了模型其中一层的计算过程。当模型存在多层时，梯度要从上一层向下一层传播。比如图中，梯度要先传播到 X ，然后才能往下一层继续传递。这就是 f 的backward的作用。这里也易推出， $\frac{\partial L}{\partial X} = concat[\frac{\partial L}{\partial X_1}, \frac{\partial L}{\partial X_2}]$

2. 按列切分

forward:

$$X * W1 \quad W2 = Y1 \quad Y2 = Y$$

(b, s, h)

(h, h'/N)

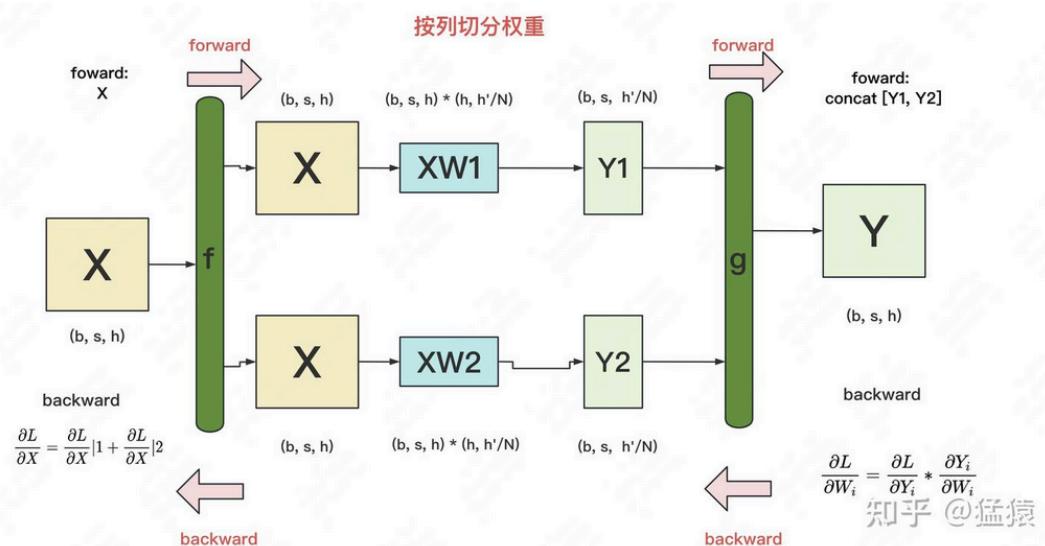
(b, s, h'/N)

Y

(b, s, h')

知乎 @猛猿

backward:



- g 的backward: 易推出 $\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial Y_i} * \frac{\partial Y_i}{\partial W_i}$
- f 的backward: 因为对于损失 L , X 既参与了 $XW1$ 的计算, 也参与了 $XW2$ 的计算。因此有 $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial X} |1 + \frac{\partial L}{\partial X}|2$ 。其中 $\frac{\partial L}{\partial X} |i$ 表示第 i 块 GPU 上计算到 X 时的梯度。

现在, 我们已分别介绍完了“按行”和“按列”切分权重的方法。在Megatron-LM中, 权重的切分操作就是由这两个基础算子组合而成的。接下来, 针对Transformer模型, 我们依次来看在不同的部分里, Megatron-LM是怎做切分的。

2. MLP层

MLP计算过程 (其中A, B为线性层):

$$GELU(X * A) * B = Y$$

(b, s, h)

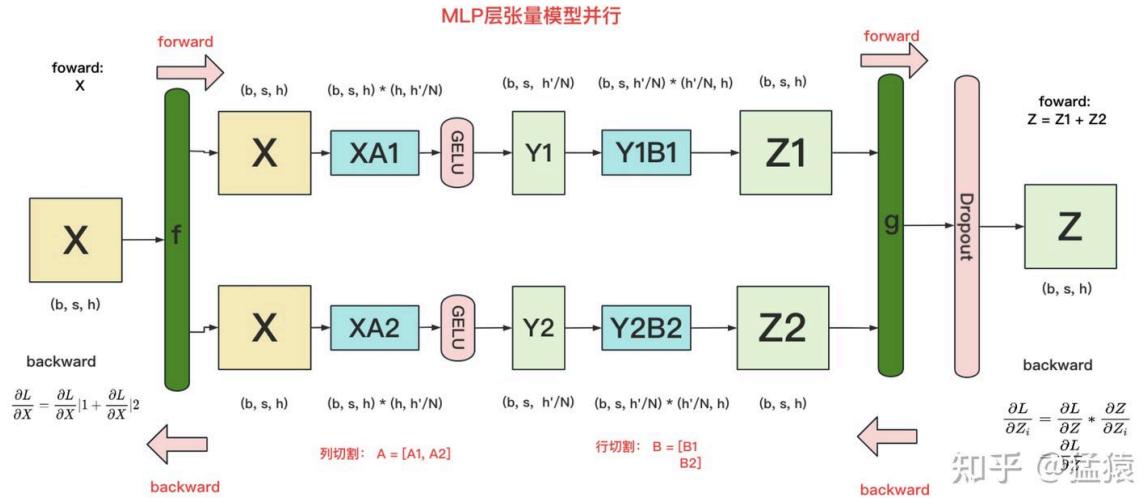
(h, h')

(h', h)

(b, s, h)

知乎 @猛猿

拆分方法: 对A进行列切割, 对B进行行切割



- f 的forward计算：把输入X拷贝到两块GPU上，每块GPU即可独立做forward计算。
- g 的forward计算：每块GPU上的forward的计算完毕，取得Z1和Z2后，GPU间做一次**AllReduce**，相加结果产生Z。
- g 的backward计算：只需要把 $\frac{\partial L}{\partial Z}$ 拷贝到两块GPU上，两块GPU就能各自独立做梯度计算。
- f 的backward计算：当当前层的梯度计算完毕，需要传递到下一层继续做梯度计算时，我们需要求得 $\frac{\partial L}{\partial X}$ 。则此时两块GPU做一次**AllReduce**，把各自的梯度 $\frac{\partial L}{\partial X}|1$ 和 $\frac{\partial L}{\partial X}|2$ 相加即可。

为什么这么切割：

为什么我们对A采用列切割，对B采用行切割呢？这样设计的原因是，我们尽量保证各GPU上的计算相互独立，减少通讯量。对A来说，需要做一次GELU的计算，而GELU函数是非线形的，它的性质如下：

$$\text{GELU}(\boxed{Y}) = \text{GELU}(\boxed{Y_1} + \boxed{Y_2}) \neq \text{GELU}(\boxed{Y_1}) + \text{GELU}(\boxed{Y_2})$$

$$\text{GELU}(\boxed{Y}) = [\text{GELU}(\boxed{Y_1}), \text{GELU}(\boxed{Y_2})]$$

知乎 @猛猿

也就意味着，如果对A采用行切割，我们必须在做GELU前，做一次AllReduce，这样就会产生额外通讯量。但是如果对A采用列切割，那每块GPU就可以继续独立计算了。一旦确认好A做列切割，那么也就相应定好B需要做行切割了。

MLP层的通信量分析：

记住一次AllReduce的通信量是2phi

由2.1的分析可知，**MLP层做forward时产生一次AllReduce，做backward时产生一次AllReduce**。在之前的文章里我们讲过，AllReduce的过程分为两个阶段，Reduce-Scatter和All-Gather，每个阶段的通讯量都相等。现在我们设每个阶段的通讯量为 Φ ，则一次**AllReduce产生的通讯量为 2Φ 。MLP层的总通讯量为 4Φ** 。

根据上面的计算图，我们也易知， $\Phi = b * s * h$

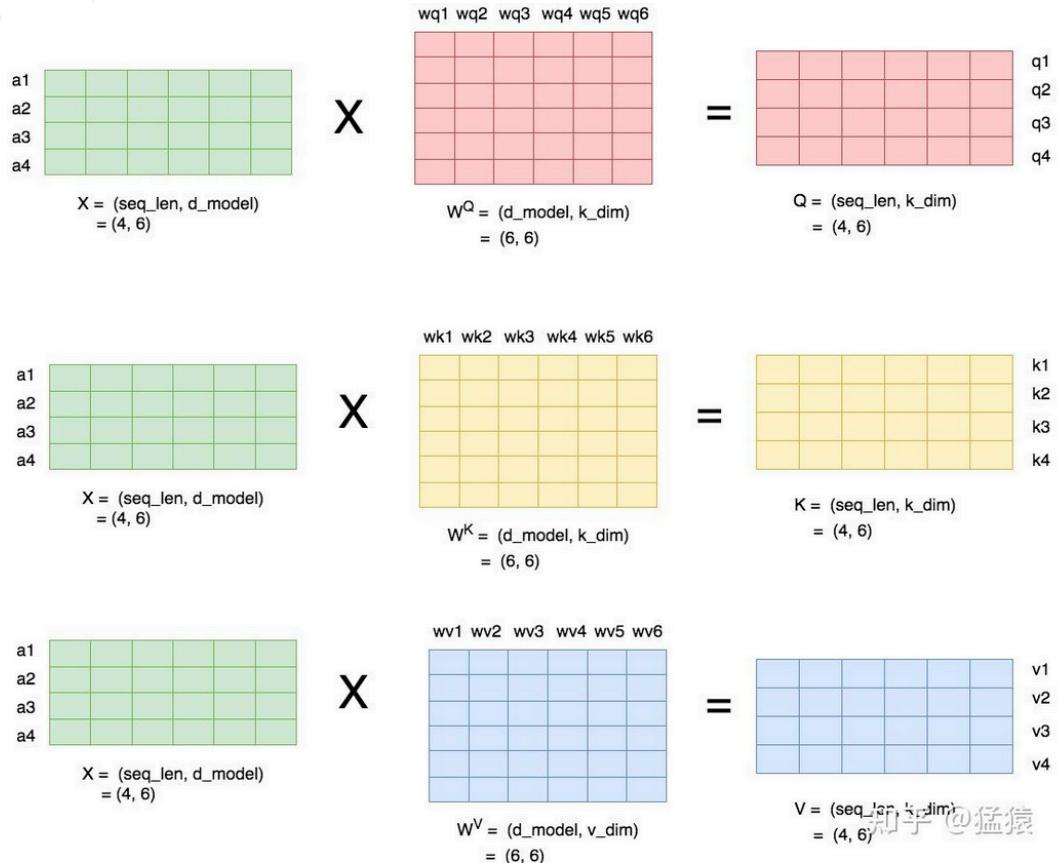
3. self-attention层

1. Multi-head Attention计算方法

head = 1时

当head数量为1时，self-attention层的计算方法如下：

≡ 目录

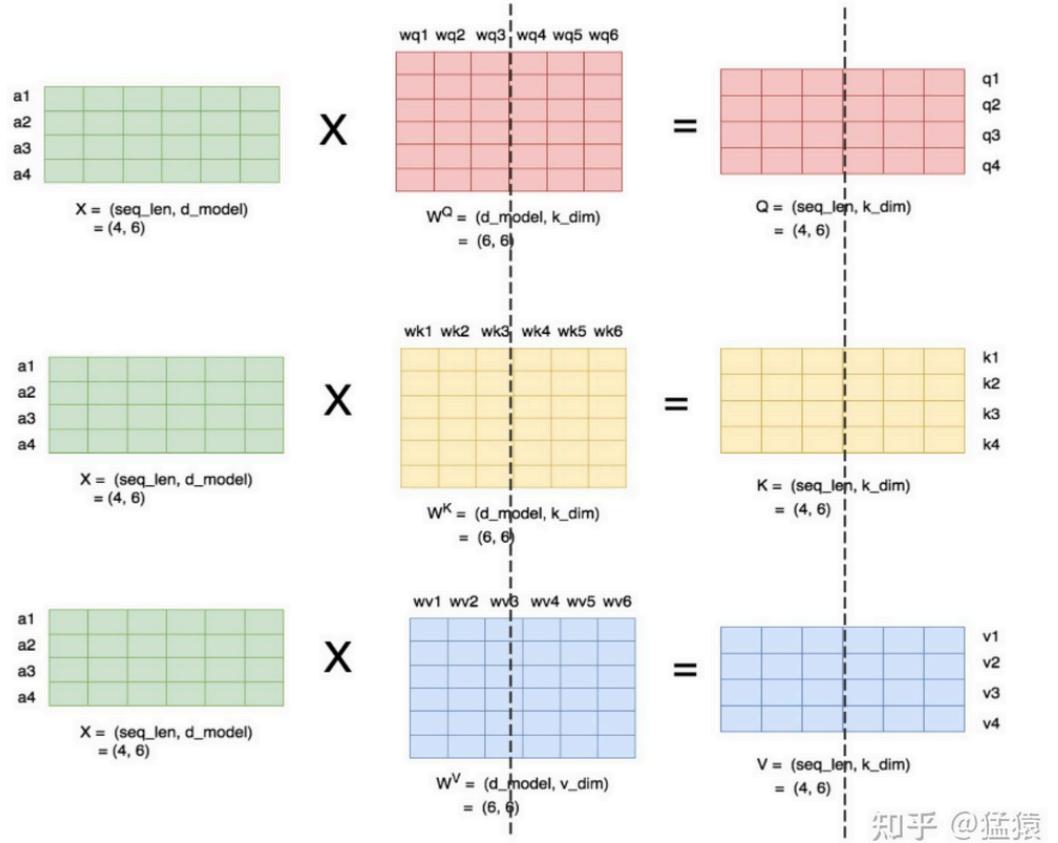


- seq_len, d_model分别为本文维度说明中的s和h，也即序列长度和每个token的向量维度
- W^Q, W^K, W^V 即attention层需要做训练的三块权重。
- k_dim, v_dim满足：

$$k_dim = v_dim = d_model // num_heads = h // num_heads$$

head = 2时

理清了单头，我们来看多头的情况，下图展示了当 $\text{num_heads} = 2$ 时 attention 层的计算方法。即对每一块权重，我们都沿着列方向 (k_{dim}) 维度切割一刀。此时每个 head 上的 W^Q, W^K, W^V 的维度都变成 $(d_{\text{model}}, k_{\text{dim}}/2)$ 。每个 head 上单独做矩阵计算，最后将计算结果 concat 起来即可。整个流程如下：

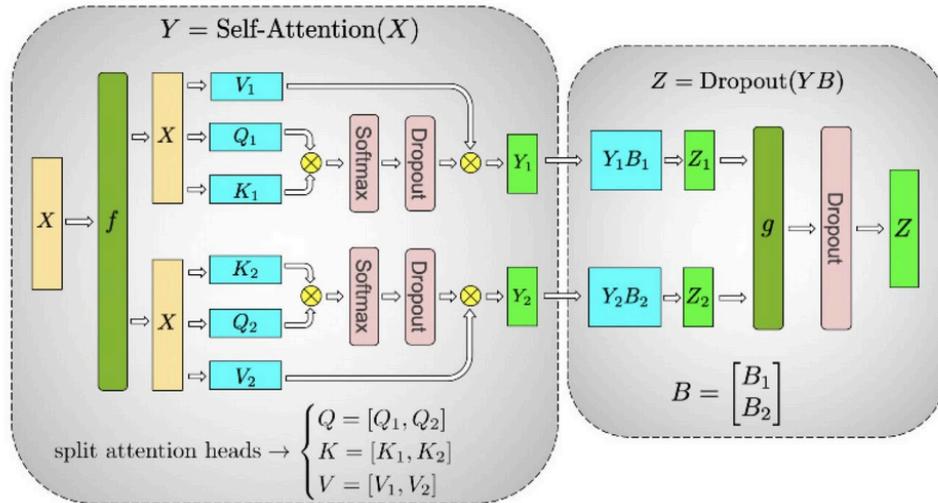


知乎 @猛猿

2. 拆分方法：其实和Multi-head多头的思想不谋而合，就是列切割嘛

可以发现，attention的多头计算简直是为张量模型并行量身定做的，因为每个头上都可以独立计算，最后再将结果concat起来。也就是说，**可以把每个头的参数放到一块GPU上**。则整个过程可以画成：

目录



(b) Self-Attention

知乎 @猛猿

对三个参数矩阵Q, K, V，按照“**列切割**”，每个头放到一块GPU上，做并行计算。对线性层B，按照“**行切割**”。切割的方式和MLP层基本一致，其forward与backward原理也一致，这里不再赘述。

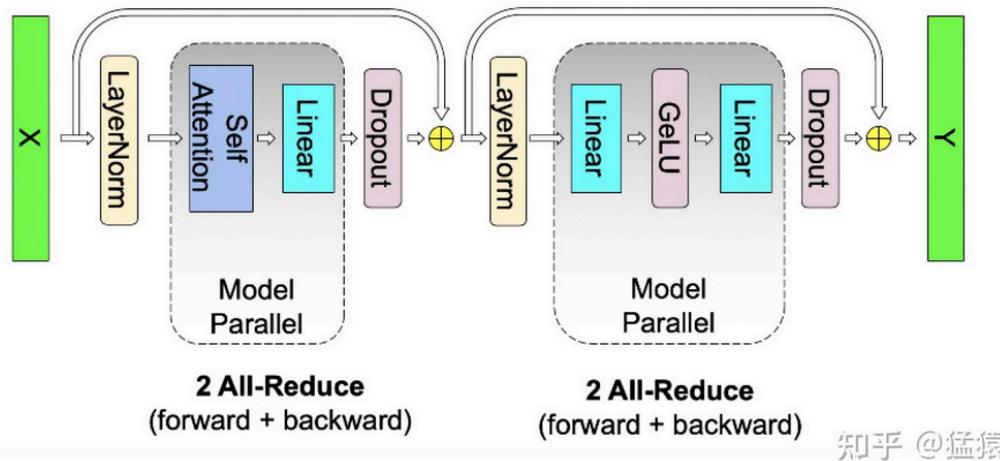
最后，在实际应用中，**并不一定按照一个head占用一块GPU来切割权重**，我们也可以一个多个head占用一块GPU，这依然不会改变单块GPU上独立计算的目的。所以实际设计时，我们**尽量保证head总数能被GPU个数整除**。

3. 通信量分析

首先我们应该明白，通信量出现在什么地方，参考**MLP**

类比于MLP层，self-attention层在forward中做一次AllReduce，在backward中做一次AllReduce。总通讯量也是 4Φ ，其中 $\Phi = b * s * h$

写到这里，我们可以把self-attention层拼接起来看整体的计算逻辑和通讯量：



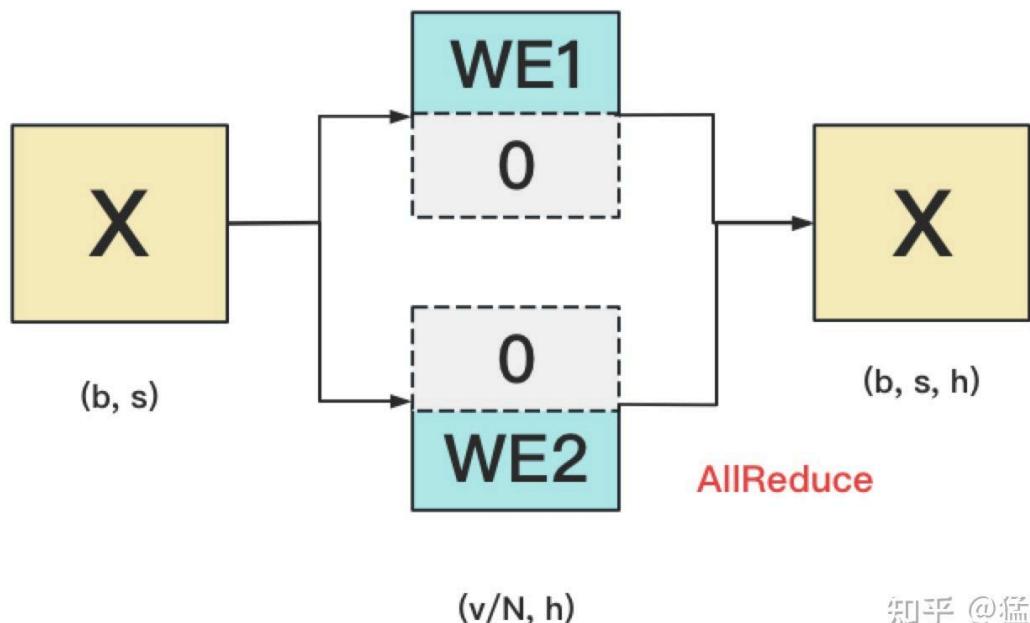
4. Embedding层

1. 输入层Embedding

对于word embedding

就是分块，下面是GPU为两块的情况，上面一块GPU维护上半部分的word embedding，下面一块维护下半部分的word embedding

然后在AllReduce就可以得到完整的word embedding了



对于positional embedding（维度 (max_s, h) ，其中 max_s 表示模型允许的最大序列长度）， max_s 本身不会太长，因此每个GPU上都拷贝一份，对显存的压力也不会太大

2. 输出层Embedding

输出层中，同样有一个word embedding，把输入再映射回词表里，得到每一个位置的词。一般来说，输入层和输出层共用一个word embedding。其计算过程如下：

需要注意的是，我们必须时刻保证输入层和输出层共用一套**word embedding**。而在backward的过程中，我们在输出层时会对**word embedding**计算一次梯度，在输入层中还会对**word embedding**计算一次梯度。在用梯度做**word embedding**权重更新时，我们必须保证用两次梯度的总和进行更新。

当模型的输入层到输入层都在一块GPU上时（即流水线并行深度=1），我们不必担心这点（实践中大部分用Megatron做并行的项目也是这么做的）。但若模型输入层和输出层在不同的GPU上时，我们就要保证在权重更新前，两块GPU上的word embedding梯度做了一次**AllReduce**。现在看得有些迷糊没关系～在本系列下一篇的源码解读里，我们会来分析这一步。

5. Cross-entropy层--计算损失函数

All-Gather

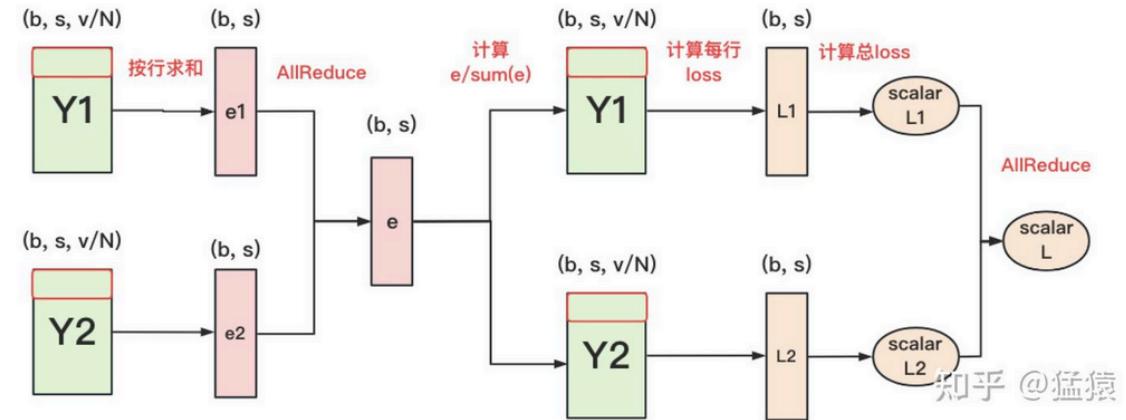
Y1 Y2 = Y

(b, s, v/N) (b, s, v)

求softmax

正常来说，我们需要对Y1和Y2做一次**All-Gather**，把它们concat起来形成Y，然后对Y的每一行做softmax，就可得到对于当前位置来说，每个词出现的概率。接着，再用此概率和真值组做cross-entropy即可。

但是All-Gather会产生额外的通讯量 $b * s * v$ 。当词表v很大时，这个通讯开销也不容忽视。针对这种情况，可以做如下优化：

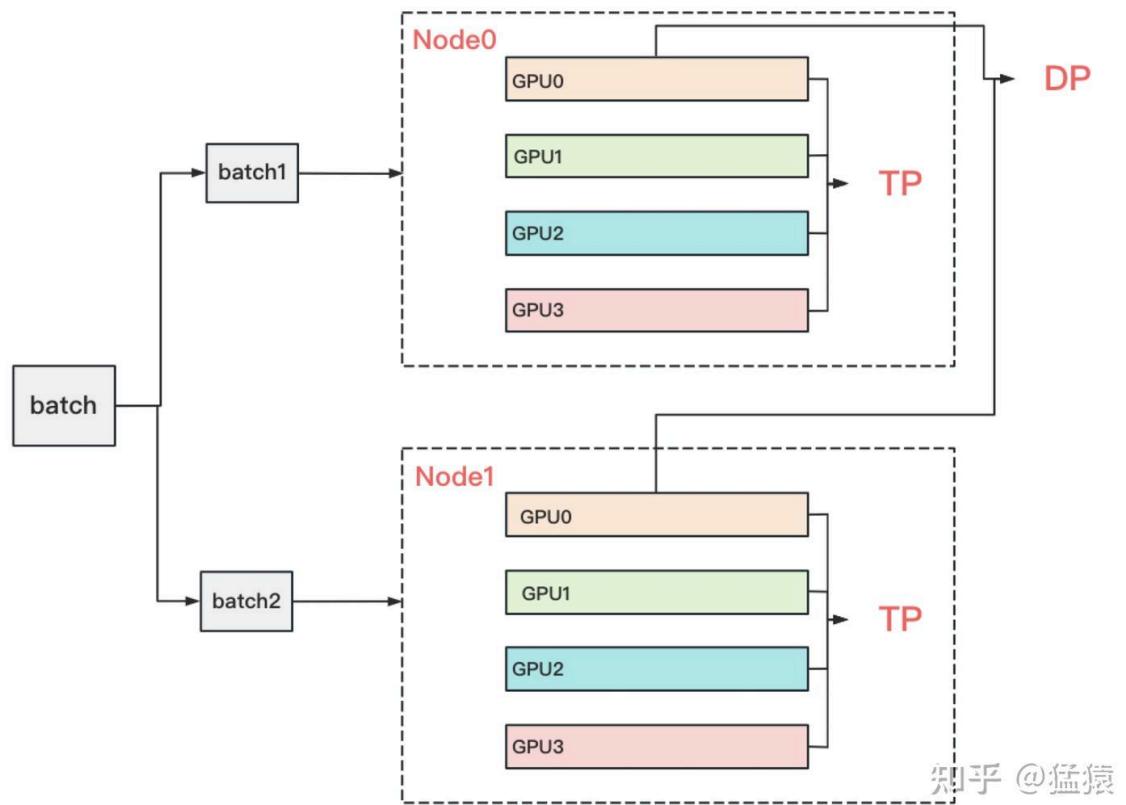


- 每块GPU上，我们可以先按行求和，得到各自GPU上的GPU_sum(e)
- 将每块GPU上结果做AllReduce，得到每行最终的sum(e)，也就softmax中的分母。此时的通讯量为 $b * s$
- 在每块GPU上，即可计算各自维护部分的 $e/sum(e)$ ，将其与真值做cross-entropy，得到每行的loss，按行加总起来以后得到GPU上scalar Loss。
- 将GPU上的scalar Loss做AllReduce，得到总Loss。此时通讯量为N。

这样，我们把原先的通讯量从 $b * s * v$ 大大降至 $b * s + N$ 。

6. 经典并行：TP+DP--张量模型并行+数据并行

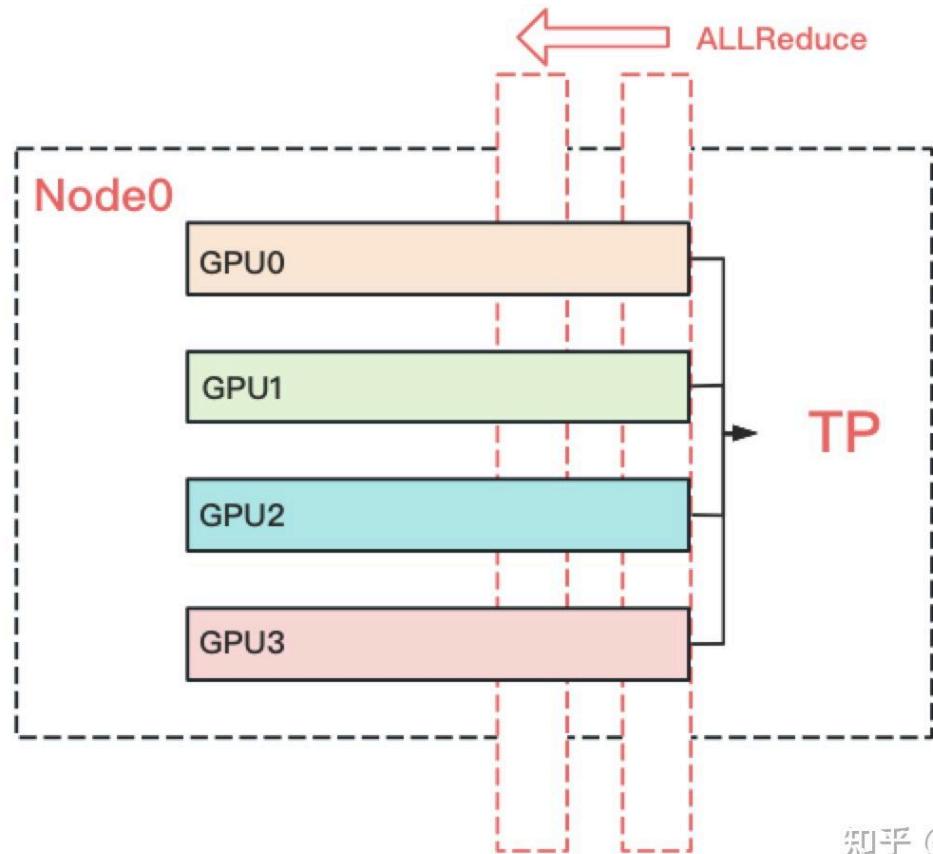
node表示一台机器，我们通常在同一台机器的GPU间做张量并行，在不同机器之间做数据并行，颜色相同的部分为一个数据并行组



1. TP和DP通信量

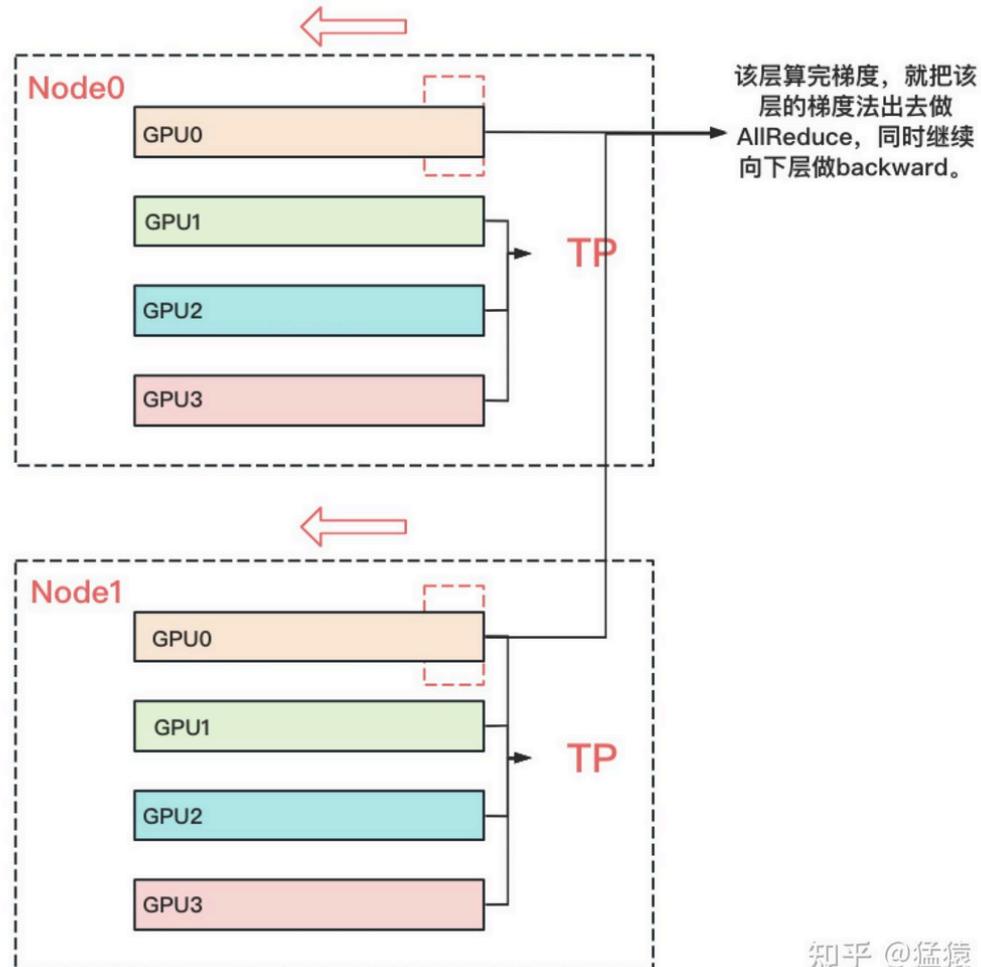
2. TP和DP计算backward的方式

TP在从上一层到下一层做backward的时候，所有GPU之间需要做一次AllReduce



知乎 @猛猿

而对DP来说，本层算完梯度以后，就正常把本层的梯度发出去，和属于一个DP组的GPU做AllReduce，同时继续往下一层做backward。下一层也是同理。**也就是在DP组中，下一层不依赖上一层的梯度聚合结果**。因此在DP组中对带宽的要求就没那么高了。所以可以放到机器间做DP。例如下图：



- 通信量排序：通信量=一次推理的通信次数×每次通信的传输次数

从上面也可以看出

张量并行>流水线并行>数据并行

2. 理解NVIDIA--NCCL-- AllReduce 算子和 AllGather 算子--阅读源码

两个算子存储在src/device/all_reduce.h | all_gather.h文件中

由于all_gather只是all_reduce的一部分，所以我们只解析all_reduce的函数不解析all_gather了

All_reduce

runRing函数：

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

电子科技大学

GPU 相关: ringIdx CPU 驱动
nrank GPU 总量

1956

偏移量 offset = Count \times Index

张量

环 channel | 块 chunk

grid offset: 当前环需要处理的
在整个张量的
偏移量

elemOffset: 当前环的偏移量

chunkOffset: 块的偏移量

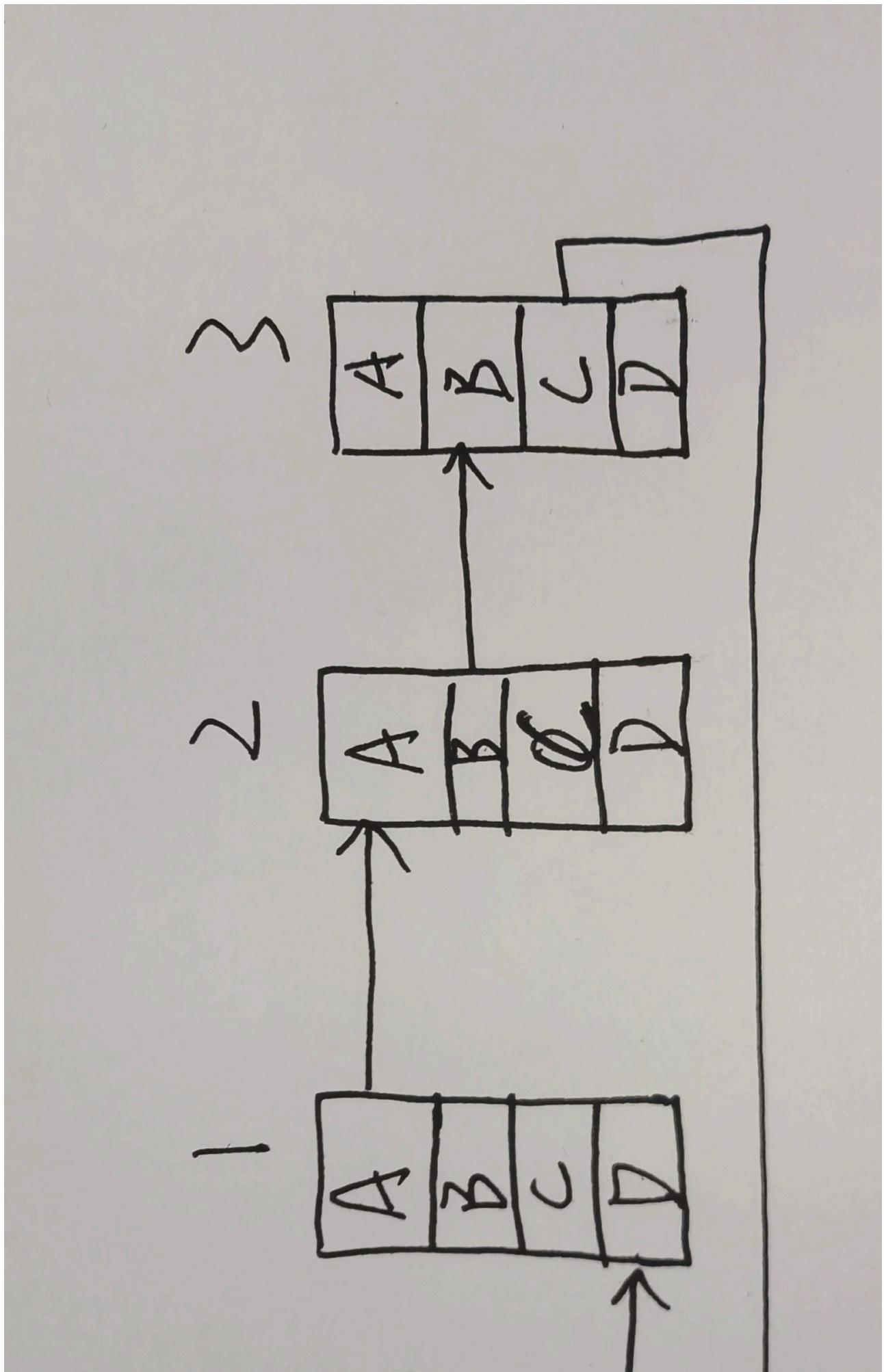
channelCount: 环需要处理的
元素总数

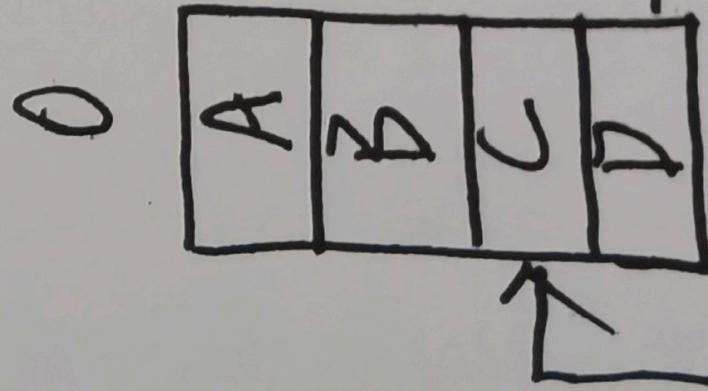
chunkCount: 块需要处理的
元素总数

loopCount = nrank * chunkCount
一次环需要处理的元素总数
(channelCount \geq loopCount)

remCount = channelCount % loopCount
剩余量

chunk: 当前块的索引 |





step 0:

参数介绍：

从大到小：张量->channel(环)->chunk(块)

```

//获取当前通道的环信息--这里应该是用一个数据结构存储的
//感觉这里的环像一个环形双向链表
ncclRing *ring = &ncclShmem.channel.ring;

//当前GPU在这个环中的逻辑排名 (Index)

```

```
int ringIx = ring->index;
//环中GPU总量
const int nRanks = ncclShmem.comm.nRanks;

//当前环需要处理的数据在整个张量的起始偏移量
ssize_t gridOffset;
//当前环需要处理的元素总数
ssize_t channelCount;

//块大小, 每次操作的基本数据单元包含的元素数量
ssize_t chunkCount;

//大循环: 也就是需要多少个环
//一次大循环处理nRanks个块, 每个块包括chunkCount这么多的元素
//即loopCount为一个环需要处理的元素总量
//这里需要区分的是channelCount >= loopCount
const ssize_t loopCount = nRanks * chunkCount;
```

大循环:

```
//每次大循环就是每个环
for (ssize_t elemOffset = 0; elemOffset < channelCount; elemOffset += loopCount) {

    ssize_t remCount = channelCount - elemOffset;
    ssize_t chunkOffset;

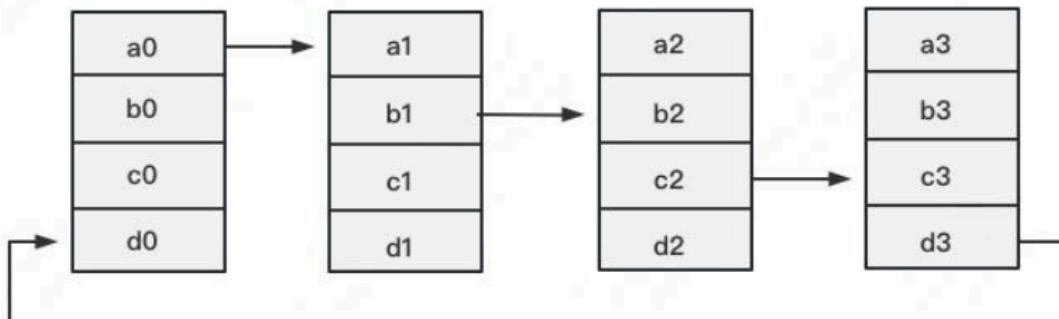
    if (remCount < loopCount) chunkCount = alignUp(divUp(remCount, nranks),
16/sizeof(T));
    //定义取模的那个函数
    auto modRanks = [&]__device__(int r)->int {
        return r - (r >= nranks ? nranks : 0);
}
```

环算法具体解析:

1. Reduce-Scatter

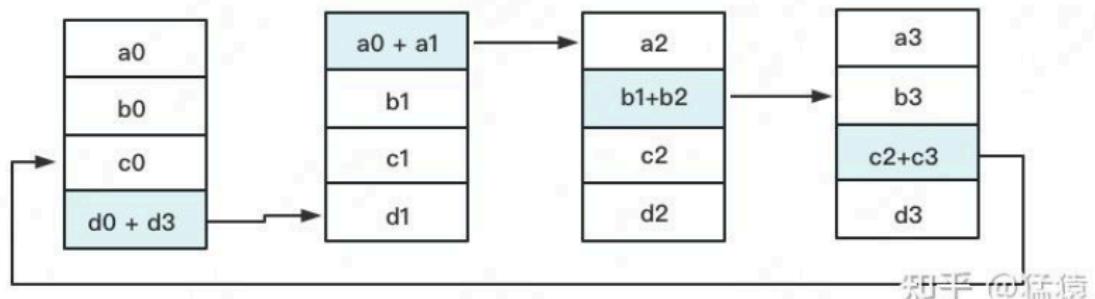
定义网络拓扑关系，每个GPU只和其相邻的两块GPU通讯，每次发送对应位置的数据进行累加

每次累加形成一个拓扑环，如下图

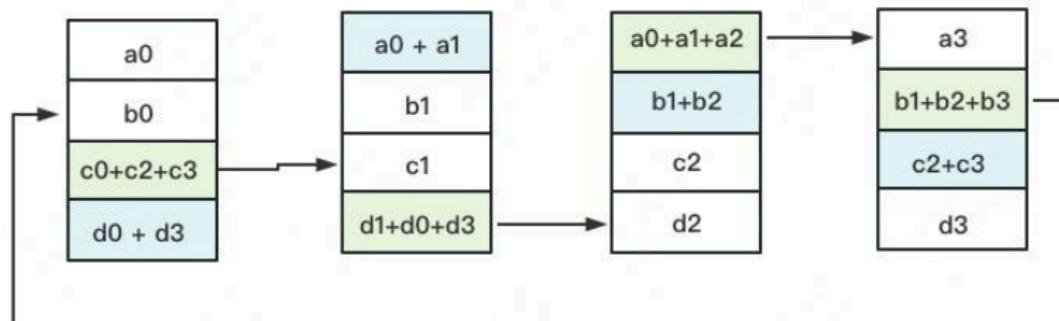


知乎 @猛猿

一次累加完成后，被更新的数据块将成为下一次更新的起点



知乎 @猛猿



知乎 @猛猿

Debug: 以4个GPU为例，假设每个GPU上面都有一个数据块（chunk）A, B, C, D，目标是所有GPU都得到A+B+C+D

步骤0：目的就是发送初始值到下一块GPU

```
// step 0: push data to next GPU
chunk = modRanks(ringIx + nRanks - 1);
chunkOffset = chunk * chunkCount;
```

```

    offset = gridOffset + elemOffset + chunkOffset;
    nelem = (int)min(chunkCount, remCount - chunkOffset);
    prims.directSend(offset, offset, nelem);
/*chunk: 每一块发送的块的ID
比如: 第0个GPU, 发送 (0+4-1) %4 = 3(D块), 所以1->A (0) ,2->B (1) ,3->C (2)

chunkOffset:块的偏移量, 即该GPU需要处理的块的起始位置

offset: 当前GPU需要处理的块的偏移量, 起始位置
nelem: 当前GPU需要处理的块的元素总量
directSend: 将起始位置为offset的nelem个元素发送给下一块GPU
*/

```

步骤nranks - 2: 和第0步大致一样, 只不过最后的函数变了

```

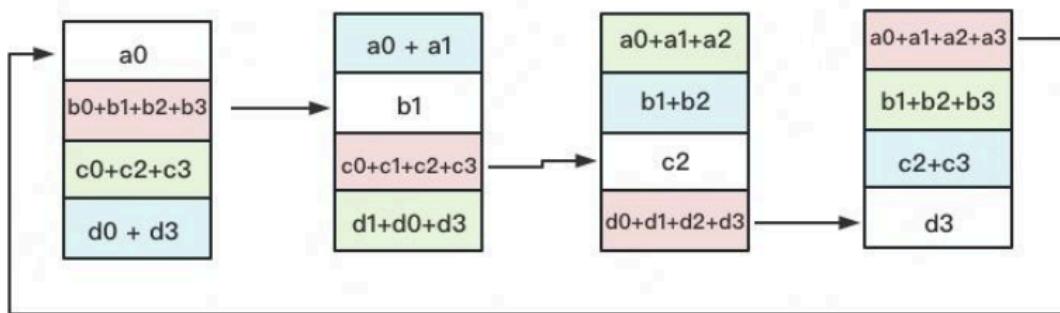
// k-2 steps: reduce and copy to next GPU
for (int j = 2; j < nranks; ++j) {
    chunk = modRanks(ringIx + nranks - j);
    chunkOffset = chunk * chunkCount;
    offset = gridOffset + elemOffset + chunkOffset;
    nelem = (int)min(chunkCount, remCount - chunkOffset);
    prims.directRecvReduceDirectSend(offset, offset, nelem);
}
/*还有不同点就是chunk的索引这里, 从-1, 变成-j, (-2, -3, -4, -5, .....), 这里就对应了Reduce-Scatter里面
的那一条, “被更新的数据块成为下一次更新的起点”, 即起点顺延
directRecvReduceDirectSend:这个函数包括三个操作
接收上一个GPU的数据块
将接收到的数据与本地相同偏移量处的数据进行归约操作
将归约后的结果发送给下一个GPU
*/

```

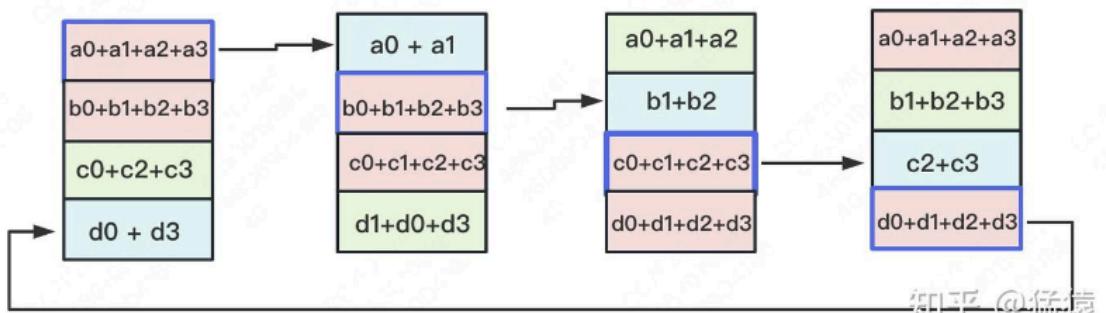
上述一共nranks-1步, 对应Reduce-Scatter

下面是All-Gather的过程

2. All-Gather--将刚才红色块数据聚合在一起



知乎 @猛猿



知乎 @猛猿

还是这样，三轮后每一块上面都是完整的数据

注意下面提到的本地输出缓冲区就是所说的覆盖（即上图所说的）

```

chunk = ringIx + 0;
chunkOffset = chunk * chunkCount;
offset = gridOffset + elemOffset + chunkOffset;
nelem = (int)min(chunkCount, remCount - chunkOffset);
prims.directRecvReduceCopyDirectSend(offset, offset, nelem, /*postOp=*/true);

/*
不同: All-Gather中
chunk为本块的索引, 即每块GPU负责它自己的那一块chunk
directRecvReduceCopyDirectSend:
从上一块接收数据
与本块上的数据进行归约 (为什么这里还需要归约, 是因为上面的最后一步只是发了还没有接收)
将最终结果写回本地输出缓冲区
将最终结果发送给下一块
*/

```

```

for (int j = 1; j < nranks - 1; ++j) {
    chunk = modRanks(ringIx + nranks - j);
    chunkOffset = chunk * chunkCount;
    offset = gridOffset + elemOffset + chunkOffset;
    nelem = (int)min(chunkCount, remCount - chunkOffset);
    prims.directRecvCopyDirectSend(offset, offset, nelem);
}

```

```

    }

/*
chunk: -j是因为顺延起点
directRecvCopyDirectSend:
从上一块接收数据
将最终结果写回本地输出缓冲区
将最终结果发送给下一块
*/

```

```

chunk = modRanks(ringIx + 1);
chunkOffset = chunk * chunkCount;
offset = gridOffset + elemOffset + chunkOffset;
nelem = (int)min(chunkCount, remCount - chunkOffset);

prims.directRecv(offset, nelem);
/*每一个块接受它下一个块modRanks(ringIx + 1)
directRecv: 从前一块接收数据, 将其写入本地输出缓冲区
*/

```

runTreeUpDown函数: 基于树算法的All-Reduce操作

分级: 张量->树->chunk块

参数说明: 参数和刚才的差不多

```

//获取当前通道的树形拓扑信息, 这棵树定义了当前GPU的父结点和子结点
//这里就是将GPU的信息用树这个数据结构进行存储了起来
ncclTree *tree = &ncclShmem.channel.tree;
size_t gridOffset;
size_t channelCount;
size_t chunkCount;
//这个函数不用管, 你就知道它把上面的变量赋值了即可
ncclCollCbdPart(work, ncclShmem.channelId, Proto::Id, sizeof(T), (size_t*)nullptr,
&gridOffset, &channelCount, &chunkCount);
size_t offset;
int nelem;

```

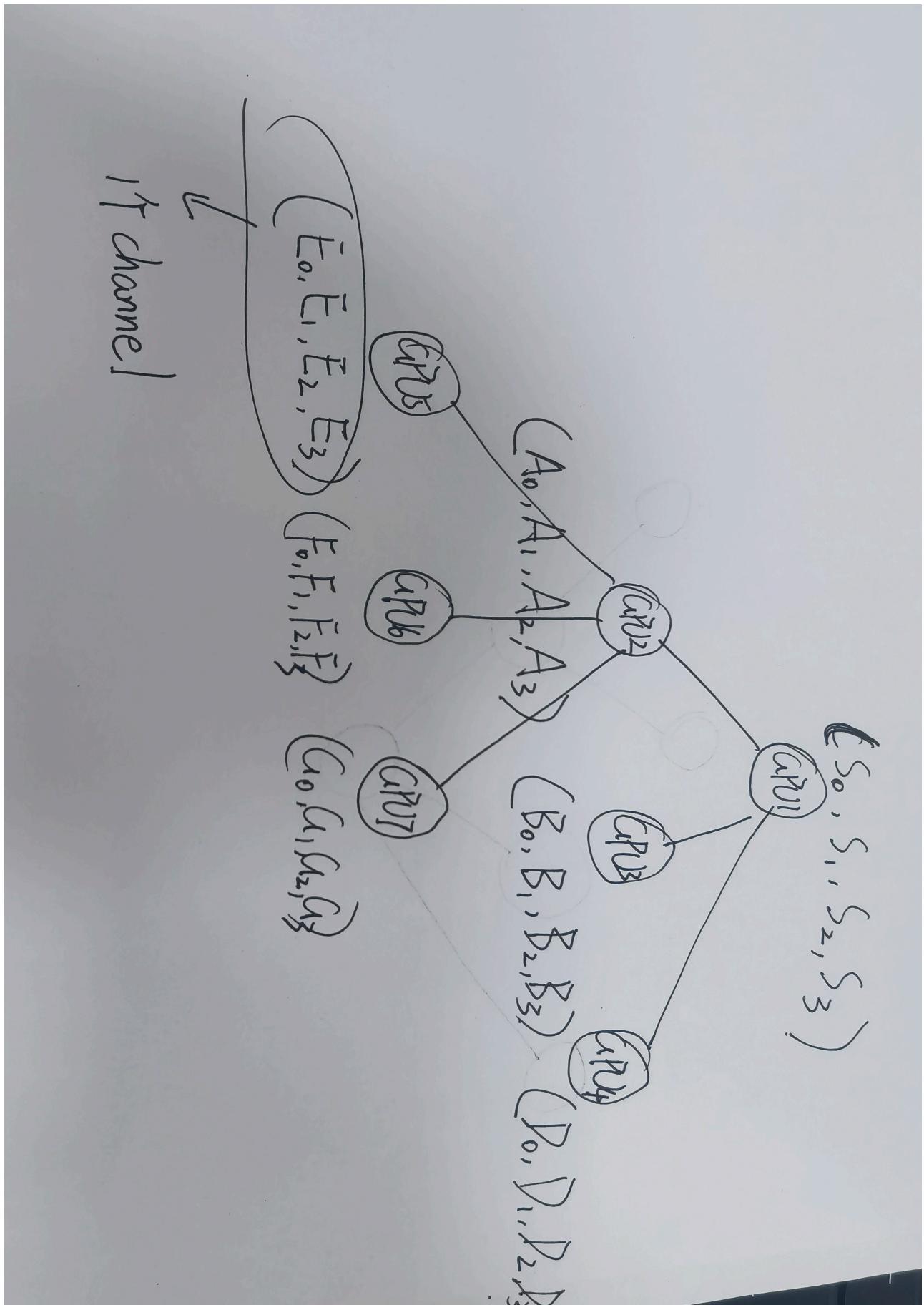
第一阶段: Reduce: 从叶子节点向上归约到根节点

与上面不同的是, 上面每个elemOffset的步长是chunkCount×nranks (每个GPU实际上处理的是一个chunk), 而这里elemOffset的步长是chunkCount, 但其实每个GPU维护的都是一整个channel的东西

因为在ring里面nranks个chunk组成一个channel, 每块GPU维护其中一个chunk

而在tree里面, 一个channel就是一个chunk, 每块GPU维护其中一个chunk, 但chunk在channel里面是并列排放的

比如像下面这个东西, 归约就是每个GPU维护的channel将其传输到根节点, 最终 S0+A0+B0+C0+D0+E0+F0+G0, 然后根节点再把这些向下传播到每个GPU上, 使每个GPU上面都是最终结果



```

{ // Reduce : max number of recv is 3, max number of send is 1 (binary tree + local)
// 在归约阶段, 每个节点最多可以接收三个子结点的数据, 但最多只向一个父结点发送数据
// 这里不用管, 这里是归约阶段初始化原语对象

```

```

    Primitives<T, RedOp, FanAsymmetric<NCCL_MAX_TREE_ARITY, 1>, /*Direct=*/1, Proto,
0> prims
    (tid, nthreads, tree->down, &tree->up, work->sendbuff, work->recvbuff, work-
>redOpArg, 0, 0, 0, work);

    //第一种情况: 当前节点是根节点
    if (tree->up == -1) {
        for (size_t elemOffset = 0; elemOffset < channelCount; elemOffset += chunkCount)
    {
        offset = gridOffset + elemOffset;
        nelem = min(chunkCount, channelCount - elemOffset);
        prims.directRecvReduceCopy(offset, offset, nelem, /*postOp=*/true);
        /*directRecvReduceCopy: 从所有子节点接收他们归约好的数据, 并与本地数据进行最终归约, 将
最终结果写入本地缓冲区*/
    }
    }

    //第二种情况: 当前节点是叶子节点 (没有down子节点)
    else if (tree->down[0] == -1) {
        for (size_t elemOffset = 0; elemOffset < channelCount; elemOffset += chunkCount)
    {
        offset = gridOffset + elemOffset;
        nelem = min(chunkCount, channelCount - elemOffset);
        prims.directSend(offset, offset, nelem);
        /*directSend: 将自己本地输入数据发送给父结点*/
    }
    }

    //第三种情况: 当前结点是中间节点
    else {
        for (size_t elemOffset = 0; elemOffset < channelCount; elemOffset += chunkCount)
    {
        offset = gridOffset + elemOffset;
        nelem = min(chunkCount, channelCount - elemOffset);
        prims.directRecvReduceDirectSend(offset, offset, nelem);
        /*directRecvReduceDirectSend: 从所有子节点接收数据, 将接收到的数据与本地数据进行归
约, 将这个中间结果发送给父结点*/
    }
    }
}

```

第二阶段: Broadcast

```

{ // Broadcast : max number of recv is 1, max number of send is 3 (binary tree + local)
    Primitives<T, RedOp, FanAsymmetric<1, NCCL_MAX_TREE_ARITY>, /*Direct=*/1, Proto,
0> prims
    (tid, nthreads, &tree->up, tree->down, work->sendbuff, work->recvbuff, work-
>redOpArg, 0, 0, 0, work);

    //case1: 根节点
    if (tree->up == -1) {
        for (size_t elemOffset = 0; elemOffset < channelCount; elemOffset += chunkCount)
    {

```

```

        offset = gridOffset + elemOffset;
        nelem = min(chunkCount, channelCount - elemOffset);
        prims.directSendFromOutput(offset, nelem);
        /*directSendFromOutput: 从输出缓冲区读取数据，并将其发送给所有子结点*/
    }
}

//case2: 叶子节点
else if (tree->down[0] == -1) {
    for (size_t elemOffset = 0; elemOffset < channelCount; elemOffset += chunkCount)
{
    offset = gridOffset + elemOffset;
    nelem = min(chunkCount, channelCount - elemOffset);
    prims.directRecv(offset, nelem);
    /*directRecv: 从父结点接受最终结果，并将其写入本地输出缓冲区，*/
}
}

//case3: 中间节点
else {
    for (size_t elemOffset = 0; elemOffset < channelCount; elemOffset += chunkCount)
{
    offset = gridOffset + elemOffset;
    nelem = min(chunkCount, channelCount - elemOffset);
    prims.directRecvCopyDirectSend(offset, offset, nelem);
    /*directRecvCopyDirectSend: 从父结点接收数据，将接收到的数据写入本地输出缓冲区，将数据
转发给所有子结点*/
}
}
}

```

runTreeSplit函数分析：相比上面的tree函数就是需要根据proto通信确定线程分割策略

和runTreeUpDown的区别：

1. 该函数利用了线程分割策略，将线程分为了两部分，一部分reduce，一部分broadcast，runTreeUpDown这个是所有线程先执行reduce，再执行broadcast

```

// 根据协议类型确定线程分割策略
if (Proto::Id == NCCL_PROTO_SIMPLE) {
    // SIMPLE协议：一半线程用于reduce，一半用于broadcast
    nthreadsSplit = nthreads/2;
    if (nthreadsSplit >= 256) nthreadsSplit += 64; // 优化调整
} else { // LL & LL128协议
    // LL协议：70%线程用于reduce，30%用于broadcast（因为reduce更计算密集）
    nthreadsSplit = (nthreads*7/(10*WARP_SIZE))*WARP_SIZE; // 对齐到WARP_SIZE
}

```

2. 所用协议不一样

```

//根节点的情况
if (tree->up == -1) {

```

```

// 初始化prims, 使用对称通信模式 (FanSymmetric)
Primitives<T, RedOp, FanSymmetric<NCCL_MAX_TREE_ARITY>, /*Direct=*/1, Proto,
0>
    prims(tid, nthreads, tree->down, tree->down, work->sendbuff, work->recvbuff,
work->redOpArg, 0, 0, 0, work);
    .....略
    /*directRecvReduceCopyDirectSend: 接收->归约->复制->发送 (允许异步提交) */
}
}

//中间结点+叶子结点: 属于Reduce组 (前nthreadsSplit线程)
else if (tid < nthreadsSplit) {
    //初始化prims, 使用非对称通信模式 (FanAsymmetric)
    Primitives<T, RedOp, FanAsymmetric<NCCL_MAX_TREE_ARITY, 1>, /*Direct=*/1, Proto,
0>
    prims(tid, nthreadsSplit, tree->down, &tree->up, work->sendbuff, work->recvbuff,
work->redOpArg, 0*Proto::MaxGroupWidth, 0, 0, work);
    //如果是叶子节点, 那么直接发送数据给父结点
    if (tree->down[0] == -1) {
        .....略
    }
}
//如果不是, 就需要接收子结点数据->归约->再发送给父结点
else {
    .....略
}
}

//中间结点+叶子结点: 属于Broadcast组 (剩余的线程)
else {
    //初始化prims: 使用非对称通信模式
    Primitives<T, RedOp, FanAsymmetric<1, NCCL_MAX_TREE_ARITY>, /*Direct=*/1, Proto,
0>
    prims(tid-nthreadsSplit, nthreads-nthreadsSplit, &tree->up, tree->down, work-
>sendbuff, work->recvbuff,
        work->redOpArg, 1*Proto::MaxGroupWidth, 0, 0, work);
    //叶子节点: 同样只发送数据
    if (tree->down[0] == -1) {
        .....略
    }
}
//中间节点: 接受+归约+发送
else {
    .....略
}
}
}

```

模板特化：为不同的算法+协议提供特化的实现

NCCL_ALGO_RING, NCCL_PROTO_SIMPLE : 环状算法, 简单协议->runRing

NCCL_ALGO_TREE, NCCL_PROTO_SIMPLE :树形算法, 简单协议->runTreeUpDown(特定CUDA版本及其架构使用)/runTreeSplit (标准的树算法实现)

NCCL_ALGO_RING, NCCL_PROTO_LL : 环状算法, 低延迟协议->runRing

NCCL_ALGO_TREE, NCCL_PROTO_LL :树形算法, 低延迟协议->runTreeSplit

NCCL_ALGO_RING, NCCL_PROTO_LL128 : 环状算法, LL128->runRing

NCCL_ALGO_TREE, NCCL_PROTO_LL128 : 树形算法, LL128->runTreeSplit

Simple,LL,LL128三者协议的区别:

- Simple高带宽, 数据量很大, 也带来了延迟高的缺点
- LL最小化延迟, 数据量小, 但是带宽利用率低
- LL128: 在延迟和带宽之间做了平衡, 缺点就是中庸

这四个专为硬件优化的算法, 懒得看了, 等水平高了之后再回来看源码

NCCL_ALGO_COLNET_DIRECT, NCCL_PROTO_SIMPLE : CollNet Direct算法, 简单协议

专为CollNet硬件进行优化的算法

NCCL_ALGO_COLNET_CHAIN, NCCL_PROTO_SIMPLE : CollNet 链式算法, 简单协议

NCCL_ALGO_NVLS, NCCL_PROTO_SIMPLE : 针对NVLink和SHARP技术的GPU优化的算法 (Hopper架构的NVIDIA GPU), 简单协议

NCCL_ALGO_NVLS_TREE, NCCL_PROTO_SIMPLE : 针对NVLink和SHARP技术的Tree算法, 简单协议

All-Gather

这个All-Gather是All-Reduce的一部分, 感觉都是一样的逻辑, 所以这里也不展开来说了