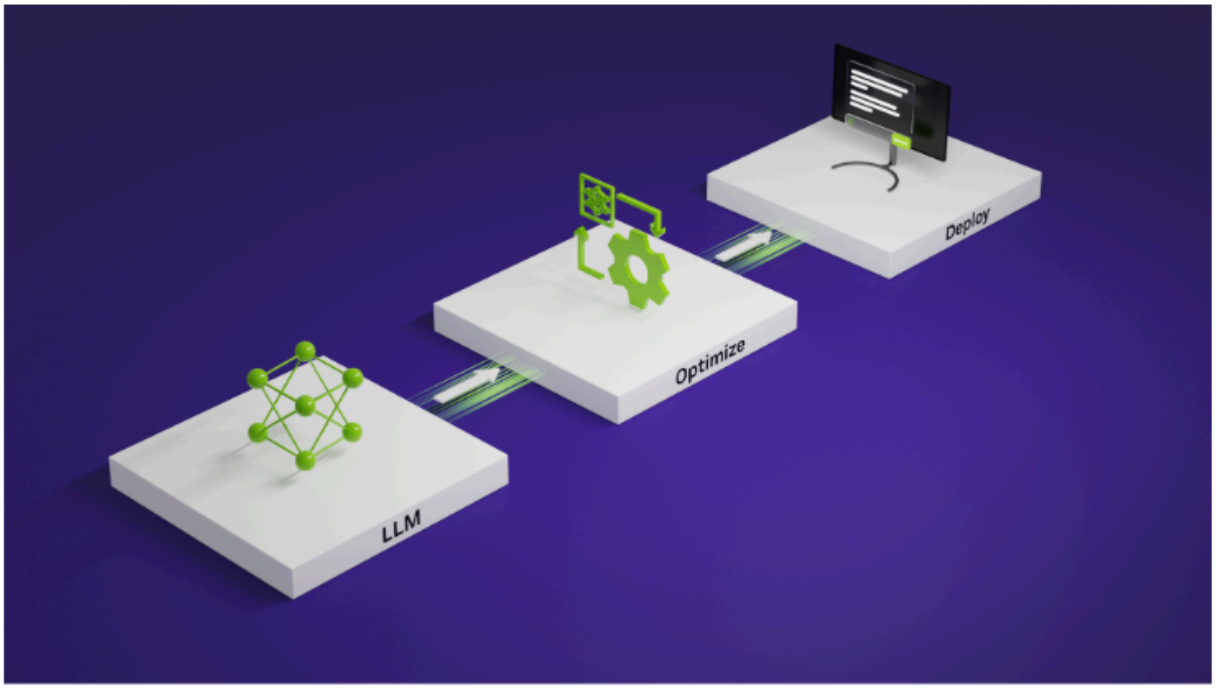
llm推理优化技术

> 原文链接：Mastering LLM Techniques: Inference Optimization | NVIDIA Technical Blog



堆叠Transformer层以创建大型模型可以获得更好的准确性、few-shot学习能力，甚至在各种语言任务中具有接近人类的涌现能力。这些基础模型的训练成本很高，而且在推理过程中可能需要大量的内存和计算（经常性成本）。当今最流行的大型语言模型（LLM）的大小可以达到数百亿到数千亿个参数，并且根据用例的不同，可能需要摄入长输入（或上下文），这也会增加开销。

这篇文章讨论了LLM推理中最紧迫的挑战，以及一些实用的解决方案。读者应该对transformer架构和注意力机制有一个基本的了解。

1.理解LLM推理

大多数流行的only-decode LLM（例如 GPT-3）都是针对因果建模目标进行预训练的，本质上是作为下一个词预测器。这些 LLM 将一系列tokens作为输入，并自回归生成后续tokens，直到满足停止条件（例如，生成tokens数量的限制或遇到停止词）或直到生成特殊的 `<end>` 标记生成结束的tokens。该过程涉及两个阶段：预填充阶段和解码阶段。

请注意，tokens是模型处理的语言的原子部分。一个tokens大约是四个英文字符。所有自然语言在输入模型之前都会转换为tokens。

1.1 预填充阶段或处理输入

在预填充阶段，LLM处理输入token以计算中间状态（keys和value），用于生成“第一个”token。每个新的token都依赖于所有先前的token，但由于输入的全部已知，因此在运算上，都是高度并行化矩阵运算，可以有效地使用GPU。

only-decode LLM包含两个阶段：

1. **Prefill（预填充）阶段**：处理整个输入 prompt（如 `"Hello, how are"`）
2. **Decoding（解码）阶段**：逐个生成新 token（如 `"you?"`）

> **Note**
>
> prefill具体流程如下：
>
> **步骤 1：Prefill（并行处理整个 prompt）**
>
> - 输入：prompt tokens = $[x_0, x_1, \ldots, x_{n-1}]$（长度为 n）
> - 模型一次性前向传播整个序列：

    - 计算每一层的 **Key（K）和 Value（V）缓存（KV Cache）**
    - 计算最后一个位置（位置 $n-1$）的 **hidden state**
- 由于所有输入已知，**Q/K/V 的计算可以完全并行化**（使用标准的 self-attention，causal mask 仅作用于未来位置）

步骤 2：生成第一个输出 token

- 用 prefill 阶段得到的 **最后一个 hidden state** 经过 LM head → 得到 logits
- 采样或选 top-k → 得到 **第一个生成的 token**（即第 n 个 token）

“第一个生成的 token” 是 **prefill 阶段的直接输出结果**，而不是在 decoding 循环中“凭空”产生的。

举个具体例子:

```
Prompt: "The capital of France is"

Tokens: [x₀="The", x₁="capital", x₂="of", x₃="France", x₄="is"]
```

- Prefill 阶段计算:
    - `logits₀` → 应该预测 "capital"（但我们不在乎）
    - `logits₁` → 应该预测 "of"（也不在乎）
    - ...
    - `logits₄` → **预测下一个词，比如 "Paris"**
- 所以:
    - **输入**：`["The", "capital", "of", "France", "is"]`
    - **第一个生成的 token**：`"Paris"` ← 来自 `logits₄`

> **Tip**
>
> Decoder 阶段的详细操作流程:
>
> 假设:
>
> - Prompt 长度 = n（tokens: $x_0 \dots x_{n-1}$）
> - Prefill 阶段已完成，KV Cache 中已存入 $K_0 \dots K_{n-1}, V_0 \dots V_{n-1}$
> - 当前要生成第 t 个新 token（$t = n, n+1, n+2, \dots$）
>
> **步骤 1：输入当前 token（初始时是 prompt 的最后一个 hidden state 的输出）**
>
> - **第一次 decoding（生成第 n 个 token）**：
>   - 实际输入是 prefill 阶段最后一个位置的 hidden state（不需要再输入 token id，因为 logits 已出）
>   - 但为了统一接口，有些框架会将采样出的 x_n 作为下一步的输入
> - **后续 decoding（生成 $x_{n+1}, x_{n+2}, \dots$）**：
>   - 输入：上一步生成的 token id（如 x_n）
>
> **步骤 2：Embedding + Position Encoding**
>
> - 将 token id → embedding vector
> - 加上位置编码（position = t）
>
> **步骤 3：逐层 Transformer Decoder Block 计算**
>
> 对每一层:
>
> Self-Attention:
>
> - 计算当前 token 的 Query（Q_t）
> - Key/Value 不重新计算！而是:

    - 从 KV Cache 中读取 $K_0 \dots K_{t-1}, V_0 \dots V_{t-1}$
    - 只计算当前 token 的 K_t, V_t，并写入 KV Cache（供下一步使用）
- 计算 attention：$Q_t @ [K_0 \dots K_t]^\top \rightarrow$ 加权求和 $V_0 \dots V_t$

FFN（前馈网络）：

- 标准计算

关键优化：因为 K/V 大部分来自缓存，每步只计算一个位置的 Q/K/V，计算量小，但受限于内存带宽（memory-bound）。

步骤 4：输出 logits → 采样新 token

- 最后一层 hidden state → LM Head → logits
- 应用 sampling 策略（greedy, top-k, top-p, temperature 等）
- 得到下一个 token x_{t+1}

步骤 5：循环继续

- 将 x_{t+1} 作为下一轮输入
- KV Cache 扩展一位（追加 K_{t+1}, V_{t+1}）
- 重复上述过程，直到生成 `<eos>` 或达到最大长度

1.2 解码阶段或生成输出

在解码阶段，LLM一次自回归生成一个输出token，直到满足停止条件。每个输出tokens都需要直到之前迭代的所有输出状态（keys和values）。这与预填充输入处理相比，就像矩阵向量运算未充分利用GPU计算能力。数据（weights, keys, values, activations） 从内存传输到GPU的速度决定了延迟，而不是计算实际时间消耗。即，这是一个内存限制操作。

本文中的许多推理挑战和相应的解决方案都涉及此解码阶段的优化：高效的注意力模块、有效管理键和值等。

不同的LLMs可能使用不同的tokenizers，因此比较它们之间的输出tokens可能并不简单。在比较推理吞吐量时，即使两个 LLMs每秒输出的tokens相似，如果它们使用不同的tokenizers，也可能不相等。这是因为相应的tokens可能代表不同数量的字符。

1.3 批处理（Batching）

提高 GPU 利用率和有效吞吐量的最简单方法是通过**批处理**。由于多个请求使用相同的模型，因此权重的内存成本被分散。大批量数据传输到 GPU 一次处理，将提高GPU资源的利用率。

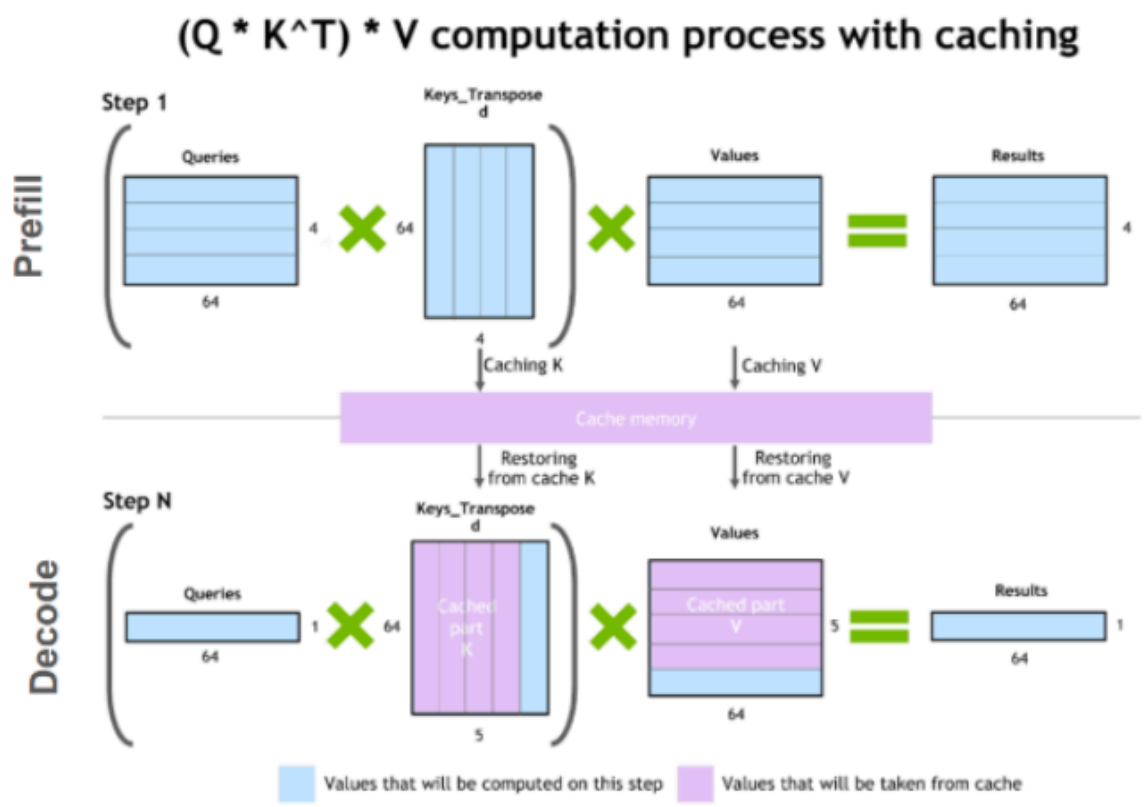
然而，批量大小只能增加到一定限制，此时可能会导致内存溢出。为了防止这种情况发生，需要查看键值 (KV) 缓存和 LLM 内存要求。

传统批处理（也称为静态批处理， static batching）不是最佳的。这是因为**对于批次中的每个请求，LLM 可能会生成不同数量的tokens，并且不同tokens有不同的执行时间**。因此，批次中的所有请求都必须等待最长token的处理完成，而生成长度的巨大差异可能会加剧这种情况。有一些方法可以缓解这种情况，例如稍动态批处理。

1.4 KV缓存

解码阶段的一种常见优化是 KV 缓存。解码阶段在每个时间步生成单个token，但每个token依赖于之前token的键和值张量（包括预填充时计算的输入tokens的 KV 张量，以及当前时间步之前计算的任何新 KV 张量）。

为了避免在每个时间步重新计算所有tokens的这些张量，**可以将它们缓存在 GPU 内存中**。每次迭代，当需要计算新token时，它们都会被添加到正在运行的缓存中，以便在下一次迭代中使用。在一些实现中，模型的每一层都有一个KV缓存。



1.5 LLM内存需求

实际上，LLM 对 GPU 显存的需求主要是**模型权重**和**KV 缓存**：

- 模型权重:模型参数占用内存。例如，具有 70 亿个参数的模型（例如 Llama2-7B），以 16 位精度（FP16 或 BF16）加载，将占用大约$7\,\mathrm{B} \times \mathrm{sizeof}(\mathrm{FP16}) \approx 14\,\mathrm{GB}$的内存。
- KV 缓存：自注意力张量的缓存占用内存，避免冗余计算。使用批处理时，批处理中每个请求的 KV 缓存仍然必须单独分配，并且可能会占用大量内存。

下面的公式描述了 KV 缓存的大小，适用于当今最常见的 LLM 架构：

$$\text{每个 token 的 KV 缓存大小 (字节)} = 2 \times (\text{num\_layers}) \times (\text{num\_heads} \times \text{dim\_head}) \times \text{precision\_in\_bytes}$$

- 第一个因子 **2** 代表 **K** 和 **V** 矩阵。
- 通常，$(\text{num\_heads} \times \text{dim\_head})$ 的值与 Transformer 的 `hidden_size`（或模型的维度，`d_model`）相同。
- 这些模型属性通常可以在配置文件中找到。

输入批次中输入序列中的每个 token 都需要此内存大小。假设半精度（FP16），KV 缓存的总大小由以下公式给出：

$$\text{总 KV 缓存大小 (字节)} = (\text{batch\_size}) \times (\text{sequence\_length}) \times 2 \times (\text{num\_layers}) \times (\text{hidden\_size}) \times \mathrm{sizeof}(\mathrm{FP16})$$

例如，对于 16 位精度的 Llama 2 7B 模型，批量大小为 1，KV 缓存的大小将为：

$$1 \times 4096 \times 2 \times 32 \times 4096 \times 2 \text{ 字节} \approx 2\,\mathrm{GB}$$

> **注**：Llama2-7B 的典型配置为 `hidden_size = 4096`，`num_layers = 32`，`sizeof(FP16) = 2` 字节。

高效的管理 KV 缓存是一项具有挑战性的工作。内存需求随着批量大小和序列长度**线性增长**，可以快速扩展。因此，它限制了可服务的吞吐量，并对长上下文输入提出了挑战。这就是本文中介绍的多项优化背后的动机。

2.模型并行化扩展LLM

减少模型权重在每设备的显存占用的一种方法是**将模型分布在多个 GPU 上**。分散内存和计算可以运行更大的模型或更大批量的输入。模型并行化是训练或推理模型所必需的，模型并行化需要比单个设备更多的内存，用来训练和推理（延迟或吞吐量）。根据模型权重的划分方式，有多种方法可以并行化模型。

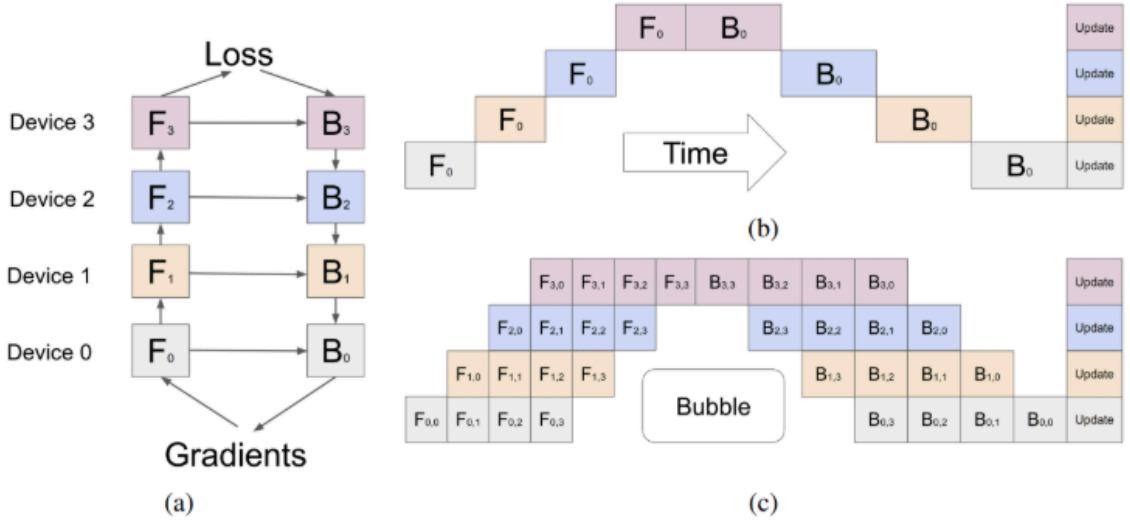
请注意，数据并行性也是一种经常在与下面列出的其他技术相同的的技术。在这种情况下，模型的权重被复制到多个设备上，并且输入的（全局）批量大小在每个设备上被分成微批次。它通过处理较大的批次来减少总体执行时间。然而，这是一种训练时间优化，在推理过程中不太相关。

2.1 Pipeline并行

Pipeline并行化将模型（垂直）分片为块，其中每个块包含在单独设备上执行的层的子集。图 2a 说明了四路Pipeline，其中模型按顺序分区，并且所有层的四分之一子集在每个设备上执行。一个设备上的一组操作的输出被传递到下一个设备，后者继续执行后续块。 F_n 和 B_n 分别表示设备 n 上的前向传播和后向传播。每个设备上存储模型权重的内存需求被分成四份。

该方法的缺点是，**由于处理的顺序性质，某些设备或层在等待前一层的输出（激活、梯度）时可能保持空闲状态。这会导致前向和后向传递效率低下或出现“Pipeline bubbles”**。在图 2b 中，白色空白区域是Pipeline并行性产生的Pipeline bubbles，其中设备闲置且未得到充分利用。

微批处理可以在一定程度上缓解这种情况，如图 2c 所示。输入的全局批次大小被分成子批次，这些子批次被一一处理，最后累积梯度。请注意，$F_{n,m}$ 和 $B_{n,m}$ 分别表示设备n上m批次的前向和后向传递。这种方法缩小了管道气泡的尺寸，但并没有完全消除它们。



2.2 Tensor并行

Tensor并行化**将模型的各个层（水平）分片为更小的、独立的计算块，这些计算块可以在不同的设备上执行。**Transformer的主要组成部分，注意力块和多层感知器（MLP）层是可以利用Tensor并行化的。在多头注意力块中，每个头或一组头可以分配给不同的设备，以便它们可以独立且并行地计算。

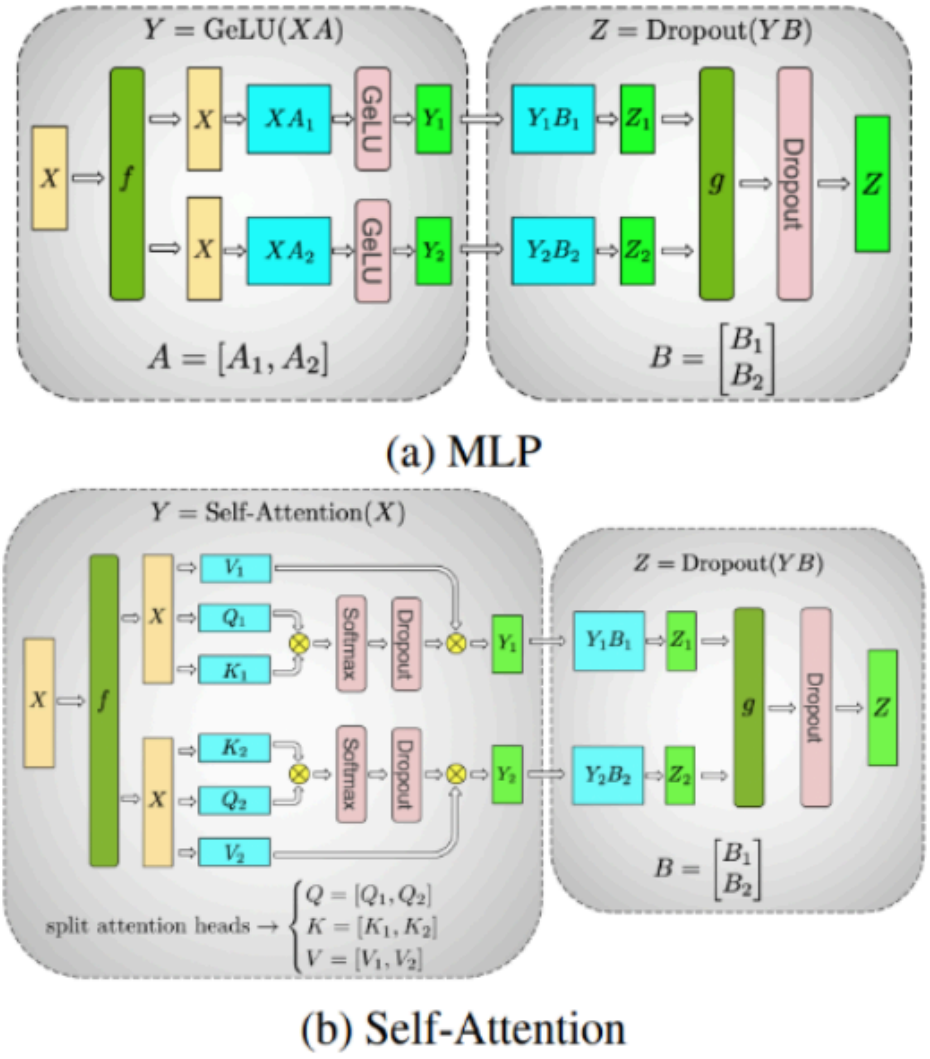


图 3a 显示了两层 MLP Tensor并行的示例，每一层都由一个圆角框表示。在第一层中，权重矩阵 A 分为 A_1 和 A_2。对于输入 X，可以在同一批次不同设备上计算 XA_1 和 XA_2，其中，f 是 identity 操作。这将每个设备上存储权重的内存需求减半。归约操作 g 组合了第二层的输出。

> **Note**
>
> idenity操作指代输入 X 不做任何变换，直接传给每个设备上的子矩阵 A_1 和 A_2，用于并行计算 XA_1 和 XA
>
> **为什么需要这个“f”?**
>
> 这是为了统一张量并行的计算模式：在 Megatron / FasterTransformer 的设计中，所有张量并行层都遵循一个通用模板：
>
> ```
> 输入 X
> ↓
> f(X)  ← 可能是切分、复制、或 identity
> ↓
> 与局部权重相乘 → 得到局部输出
> ↓
> g(局部输出) ← 归约操作（如 All-Reduce 或 All-Gather）
> ↓
> 得到完整输出
> ```
>
> - 在 MLP 第一层中，因为权重 A 是按列切分的（$A = [A_1, A_2]$），所以输入 X 需要完整地传给每个设备 → 所以 $f = \text{identity}$。
> - 而在第二层（B 矩阵按行切分），输出 $Y = [Y_1, Y_2]$ 需要拼接 → 所以 $g = \text{concat}$，或者更高效地用 **All-Reduce**（因为最终是求和）。

图 3b 是自注意力层中 Tensor 并行的示例。多个注意力头本质上是并行的，并且可以跨设备分割。

2.3 Sequence并行

Tensor并行化是有局限性，它需要将层划分为独立的、可管理的块，不适用于 `LayerNorm` 和 `Dropout` 等操作，而是在tensor并行中复制。虽然 `LayerNorm` 和 `Dropout` 的计算成本较低，但它们确实需要大量内存来存储（冗余）激活。

为什么 Tensor Parallelism（TP）对 LayerNorm / Dropout 无效?

因为：

1.LayerNorm：对每个 token 独立做归一化（沿 hidden dimension D）：

$$\text{LayerNorm}(x_s) = \gamma \cdot \frac{x_s - \mu_s}{\sigma_s} + \beta$$

其中 $x_s \in \mathbb{R}^D$ 是第 s 个 token 的向量。

2.Dropout：对每个 token 的每个元素独立随机置零。

> **Note**
>
> **关键点**：这些操作**不跨 token 交互**，只依赖当前 token 的数据。

但在标准 **Tensor Parallelism**（按 hidden dimension 切分）中：

- 每个 GPU 只持有 x_s 的一部分（例如前 2048 维，若 $D = 4096$ 且 TP = 2）。
- 要计算 LayerNorm，需要**完整的** x_s（因为必须计算该 token 在整个 D 维上的均值 μ_s 和标准差 σ_s）。
- 所以必须通过 **All-Gather** 拼出完整x_s → **通信开销大**
- 或者干脆**在每个 GPU 上复制完整输入** → **内存冗余**

> **Note**
>
> **什么是"序列维度"（sequence dimension）？**
>
> 在 Transformer 中，一个 batch 的输入通常表示为：
>
> $$X \in \mathbb{R}^{B \times S \times D}$$
>
> - **第 0 维（B）**：batch 维度 → 有多少个句子（比如 2 个）
> - **第 1 维（S）**：sequence 维度 → 每个句子有多少个词（比如 4 个）
> - **第 2 维（D）**：hidden/feature 维度 → 每个词用多少维向量表示（比如 3 维）
>
> **"序列维度" = 第二个维度（S）**，表示 token 的位置顺序。

如Reducing Activation Recomputation in Large Transformer Models所示，这些操作在输入序列中是独立的，并且这些操作**可以沿着"序列维度"进行分区**，从而提高内存效率。这称为序列并行性。

核心思想：不再按 hidden dimension（D）切分，而是按 sequence dimension（S）切分！

即：

- 把 **整个序列切成若干段**，每段分配给一个 GPU。
- 每个 GPU 持有**完整 hidden dimension**，但只处理**部分 token。**

这样切分后，LayerNorm / Dropout 怎么算?

因为每个 GPU 拥有**完整的 hidden vector**（例如 4096 维）：

- **GPU0** 对 tokens 0~3 各自做 LayerNorm → 无需通信!
- **GPU1** 对 tokens 4~7 各自做 LayerNorm → 无需通信!

同样，**Dropout** 也可以独立应用，因为它是 element-wise 且 token-wise 独立的。

激活内存从每个 GPU 存 `B × S × D` → 降到 `B × (S/N) × D`（`N` 是 GPU 数）。

> **Tip**

> **为什么内存能降到 `B×(S/N)×D`？**
>
> 因为序列并行把序列切开，每个 GPU 只处理部分 token，且每个 token 的向量是完整的，所以 LayerNorm 等操作无需通信，也无需存储其他 token 的数据。

但 Attention 和 MLP 怎么办？它们需要全局信息！

这是序列并行的挑战所在，但有巧妙解法：

(1) Self-Attention：

- Attention 需要**所有 token 的 K/V** 来计算每个 Q（因为注意力是全局的）。
- 所以在 attention 计算前，需要 **All-Gather** 所有 GPU 的 K 和 V，拼出完整序列的 `(B × S × D)` K/V。
- 计算完 attention 输出后，再通过 **Reduce-Scatter** 把输出按序列维度切分回各 GPU（每个 GPU 只保留自己负责的 token 子集）。

虽然引入通信，但**只在 attention 前后通信一次**，而 LayerNorm / Dropout / FFN 的大部分计算全程无通信。

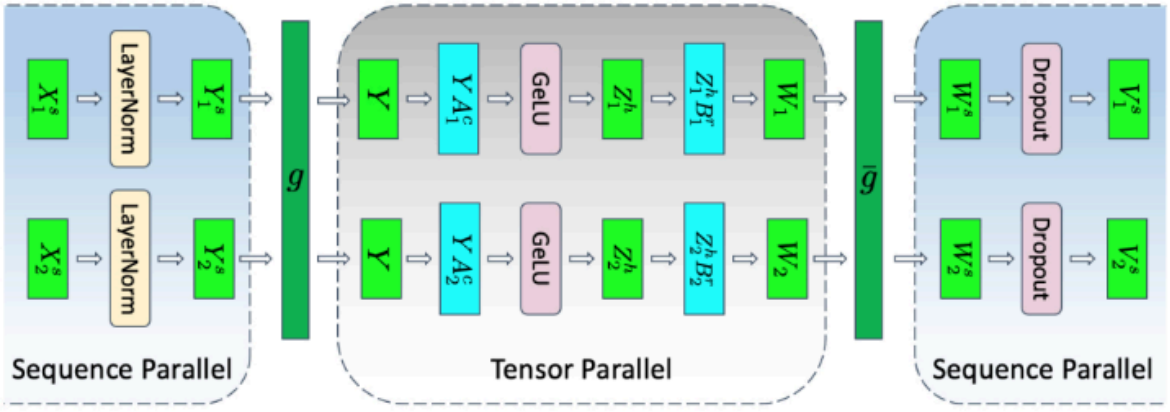
(2) MLP：

- MLP 是**逐 token 独立**的：

$$Y_s = \mathrm{GeLU}(X_s W_1) W_2$$

- 因此，每个 GPU 可以**独立计算自己那部分 token 的 MLP**，无需与其他 GPU 通信！

> 总结：序列并行通过"**切分序列、本地计算非全局操作、仅在必要时通信**"的策略，在保持模型语义不变的前提下，显著降低了激活内存占用，尤其适合长上下文场景。



> 图4，transformer层的tensor并行化和sequence并行化

模型并行技术不是唯一的，可以结合使用。它们可以帮助扩展和减少 LLM 的每 GPU 内存占用量，但也有专门针对注意力模块的优化技术。

3.注意力机制优化

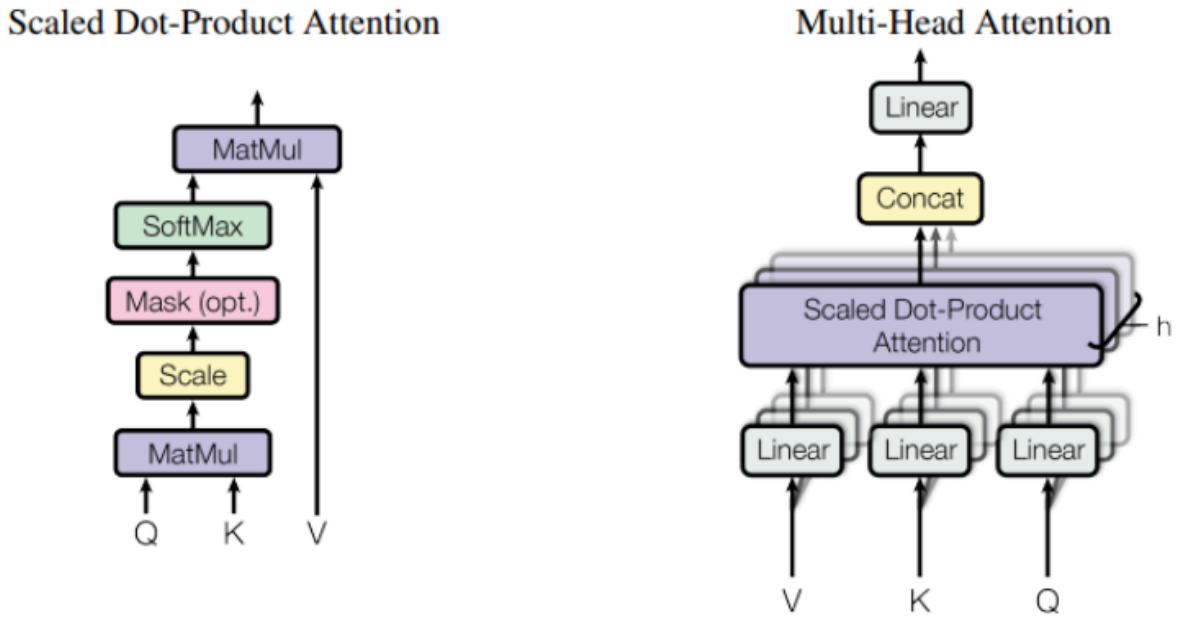
缩放点积注意力 (SDPA， scaled dot-product attention) 操作将 `query` 和 `key` 对映射到输出，如论文Attention Is All You Need所述。

3.1 多头注意力（MHA）

作为 SDPA 的增强，三个变换张量对 Q、K、V 分别进行线性变换。这些变换不会改变原有张量的尺寸，使模型能够共同关注来自不同位置的不同表示子空间的信息。这些子空间是独立学习的，使模型能够更丰富地理解输入中的不同位置。

如图 5 所示，多个并行注意力操作的输出被拼接后，再通过一个线性投影进行组合。每个并行注意力层称为一个"头"，这种方法称为**多头注意力**（Multi-Head Attention, MHA）。

当使用八个并行注意力头时，每个注意力头的维度都会减少（例如 $d_{\text{model}}/8$）。这使得多头注意力的总计算成本与单头注意力相近，同时显著提升了模型的表达能力。**Q、K、V 每一个都会被“分成”8 个部分**，每个部分对应一个“头”（head），每个头在自己的子空间中独立计算注意力。



缩放点积注意力（左）和多头注意力（右）的图示，并行的多个 SDPA 头

3.2 多查询注意力（MQA）

MHA 的推理优化之一称为多查询注意力 (MQA)，如 Fast Transformer Decoding 中提出的，**在多个注意力头之间共享键和值**。与以前一样，查询向量仍然被投影多次。

虽然 MQA 中完成的计算量与 MHA 相同，但从内存读取的数据量（键、值）只是以前的一小部分。当受内存带宽限制时，这可以实现更好的计算利用率。它还减少了内存中 KV 缓存的大小，为更大的批量大小留出了空间。

key头的减少会带来潜在的准确性下降。此外，需要在推理时利用这种优化的模型需要在启用 MQA 的情况下进行训练（或至少使用大约 5% 的训练量进行微调）。(**训练和推理的结构必须一致**)

方法	QUERY(Q)	KEY(K)	VALUE(V)
MHA（标准多头）	每个头独立：$Q_1, Q_2, ..., Q_h$	每个头独立：$K_1, K_2, ..., K_h$	每个头独立：$V_1, V_2, ..., V_h$
MQA（多查询注意力）	每个头独立：$Q_1, Q_2, ..., Q_h$	**所有头共享同一个 K**	**所有头共享同一个 V**

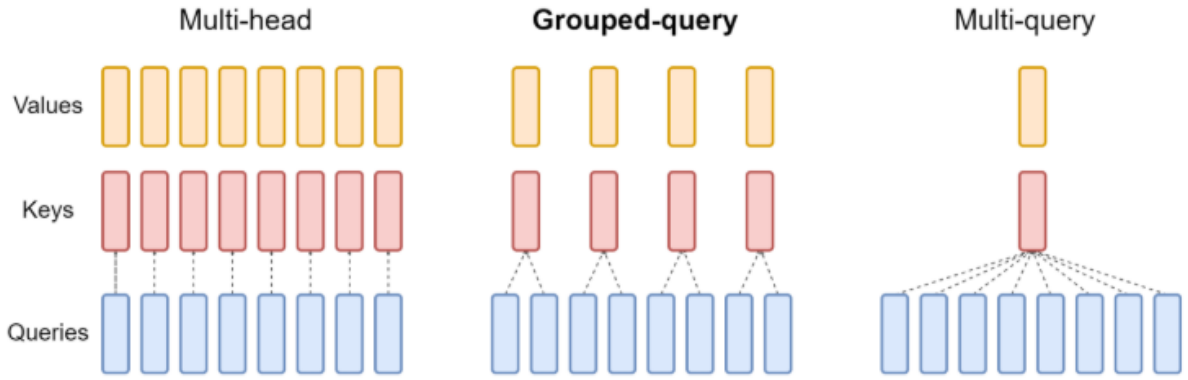
> **Tip**
>
> 为什么“key头的减少会带来潜在的准确性下降”?
>
> 核心原因：**表达能力下降**
>
> 在标准 MHA 中：
>
> - 每个头可以独立地关注输入的不同方面
>   - 头1：关注语法结构
>   - 头2：关注语义角色
>   - 头3：关注指代关系
>   - ...
> - 因为每个头有自己的 K/V，它们可以学习不同的“检索视角”：
>
>   > “当我（Q_1）想找主语时，我用 K_1 去匹配；当我（Q_2）想找宾语时，我用 K_2 去匹配。”
>
> 但在 MQA 中：

- 所有头**共用同一个 K/V**
- 相当于所有头只能用**同一个“检索视角”**去匹配内容
- 虽然 Q 不同（每个头有自己的“问题”），但“记忆库”（K/V）是同一个

3.3 分组注意力（GQA）

分组查询注意力 (GQA) 通过将键和值投影到几组查询头，在 MHA 和 MQA 之间取得平衡（图 6）。在每个组中，它的行为类似于多查询注意力。

图 6 显示多头注意力有多个键值头（左）。分组查询注意力（中心）的键值头多于一个，但少于查询头的数量，这是内存需求和模型质量之间的平衡。多查询注意力（右）具有单个键值头，有助于节省内存。



最初使用 MHA 训练的模型可以使用原始训练计算的一小部分通过 GQA 进行“升级训练”。它们获得接近 MHA 的质量，同时保持接近 MQA 的计算效率。 Llama 2 70B 是利用 GQA 的模型示例。

MQA 和 GQA 等优化通过减少存储的key头和value头的数量来帮助减少 KV 缓存所需的内存。 KV 缓存的管理方式可能仍然效率低下。与优化注意力模块本身不同，下一节将介绍一种更高效的 KV 缓存管理技术。

3.4 Flash attention

优化注意力机制的另一种方法是**修改某些计算的顺序，以更好地利用 GPU 的内存层次结构。**神经网络通常用层来描述，大多数实现也以这种方式布局，每次按顺序对输入数据进行一种计算。这并不总是能带来最佳性能，因为对已经进入内存层次结构的更高、性能更高级别的值进行更多计算可能是有益的。

在实际计算过程中将多个层融合在一起可以最大限度地减少 GPU 需要读取和写入内存的次数，并将需要相同数据的计算分组在一起，即使它们是神经网络中不同层的一部分。

一种非常流行的融合是 FlashAttention，这是一种 I/O 感知精确注意算法，详细信息请参阅 FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness。精确注意力意味着它在数学上与标准多头注意力相同（具有可用于多查询和分组查询注意力的变体），因此可以无需修改即可交换到现有的模型架构，甚至是已经训练的模型。

I/O 感知意味着在将操作融合在一起时，它会考虑前面讨论的一些内存移动成本。特别是，FlashAttention 使用“平铺”一次性完全计算并写出最终矩阵的一小部分，而不是分步对整个矩阵进行部分计算，写出中间的中间值。

Tip

1. 问题根源：标准 Attention 的 I/O 效率极低

标准多头注意力（MHA）的计算流程（以一个头为例）：

```
QK = Q @ K.T                    # (S×d) @ (d×S) → (S×S)
A = softmax(QK / sqrt(d))       # (S×S)
O = A @ V                       # (S×S) @ (S×d) → (S×d)
```

- 中间结果 `QK` 和 `A` 的大小都是 **S×S**（序列长度的平方！）
- 当 S = 2048 时，`A` 就是 **16 MB（FP16）**；S = 8192 时，**256 MB!**
- 这些中间矩阵必须：
  - 从 **HBM**（高带宽显存，慢）读入

- 写回 HBM 供下一步使用

大量时间花在"搬运数据"，而不是"计算" → GPU 计算单元经常空闲！

2. FlashAttention 的核心：分块计算 + SRAM 重用

FlashAttention 引入了 **tiling（平铺/分块）** 技术：

- 把 Q、K、V 沿序列维度切成小块（tiles）
- 对每一块组合，在 GPU 的 on-chip SRAM（快但小）
  - 计算局部 QK
  - 计算局部 softmax
  - 计算局部 O
- 累加所有块的结果，最后**只写一次最终输出 O 到 HBM**

关键效果：

- **中间矩阵（QK, A）不再写入 HBM**
- 所有临时数据只在 SRAM 内流转
- **HBM 访问次数大幅减少**（论文证明 IO 复杂度更低）

> 类比：
> 标准 Attention：每次算一点，就拿去仓库（HBM）存一下，再取回来算下一步 → 来回跑腿
> FlashAttention：把一小批原料搬进车间（SRAM），全部加工完再把成品送回仓库 → 跑腿次数少得多！

3. I/O-Aware（I/O 感知）：算法设计考虑硬件层次

FlashAttention 不是单纯"融合算子"，而是**从算法层面重新设计计算顺序**，使其匹配 GPU 内存层次：

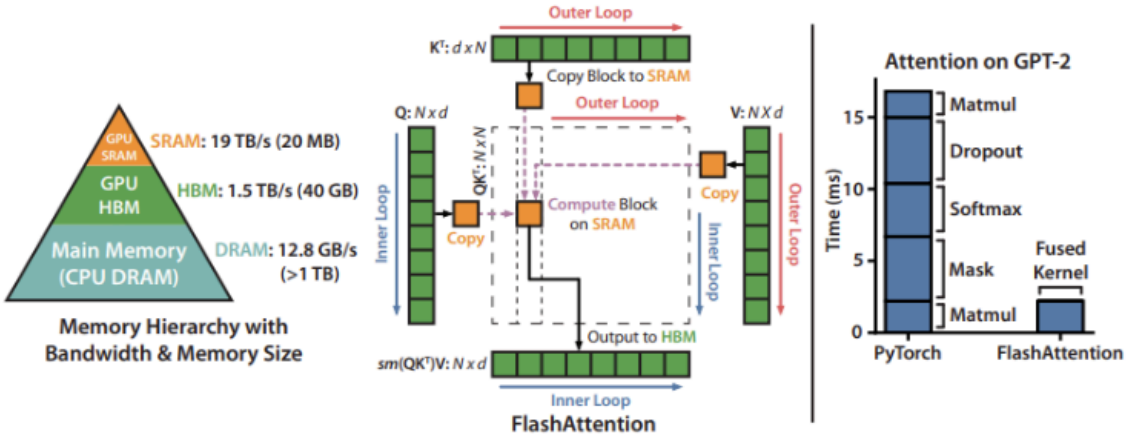
内存层级	速度	容量	FLASHATTENTION的策略
SRAM（on-chip）	极快	很小（~200KB/SM）	在这里完成 QK → softmax → O 的完整局部计算
HBM（显存）	慢（~2TB/s vs SRAM ~100TB/s）	大（40GB+）	只读输入 Q/K/V，只写最终输出 O

通过数学上的**重计算**（recomputation）技巧（比如 softmax 的数值稳定 trick），它能在不存储中间 A 的情况下，正确累加最终结果。

总结：FlashAttention 的三大核心思想

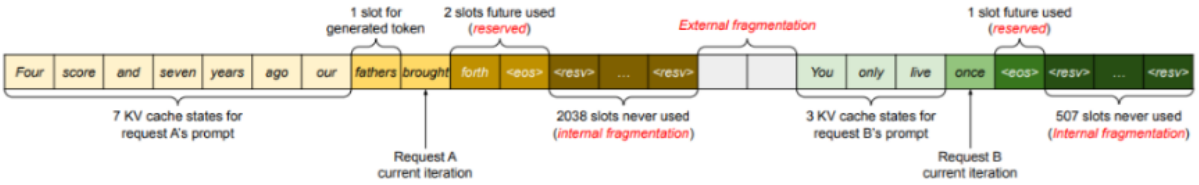
1. **分块**（Tiling）
2. **SRAM 内完成完整局部计算，避免中间结果进出 HBM**
3. **I/O 感知的算法设计：让计算顺序匹配硬件内存层次**

图 7 显示了 40 GB GPU 上的平铺 FlashAttention 计算模式和内存层次结构。右图显示了对注意力机制的不同组件进行融合和重新排序所带来的相对加速。



4.KV缓存的分页高效管理

有时，KV 缓存会静态地“过度配置”(over-provisioned)，以考虑最大可能的输入（支持的序列长度），因为输入的大小是不可预测的。例如，如果模型支持的最大序列长度为 2,048，则**无论请求中输入和生成的输出的大小如何，都将在内存中保留大小为 2,048 的数据。该空间可以是连续分配的，并且通常其中大部分未被使用，从而导致内存浪费或碎片。**该保留空间在请求的生命周期内被占用。



> 图8 由于过度配置和低效的 KV 缓存管理而导致的内存浪费和碎片

> **Tip**
>
> ## 核心概念回顾
>
> - **KV Cache**：在自回归生成中，为了加速计算，会缓存每个 token 的 Key 和 Value 向量。
> - **Slot（槽位）**：系统为 KV Cache 分配的最小存储单元。每个 token 需要占用一个或多个 slot。
> - **Request（请求）**：一个用户请求，比如“生成一段话”。
> - **Fragmentation（碎片化）**：
>   - **External Fragmentation（外部碎片）**：指系统中有足够的总空闲空间，但这些空间是分散的，无法满足一个连续的大请求。
>   - **Internal Fragmentation（内部碎片）**：指分配给某个请求的空间比它实际需要的多，造成浪费。

受操作系统分页的启发，PagedAttention 算法能够**将连续的键和值存储在内存中的不连续空间中。**它将每个请求的 KV 缓存划分为代表固定数量token的块，这些块可以不连续存储。

在注意力计算期间，使用根据记录索引获取这些块。当新的token产生时，就会进行新的区块分配。这些块的大小是固定的，消除了因不同请求需要不同分配等挑战而产生的低效率。这极大地限制了内存浪费，从而实现了更大的批量大小（从而提高了吞吐量）。

5.模型优化技术

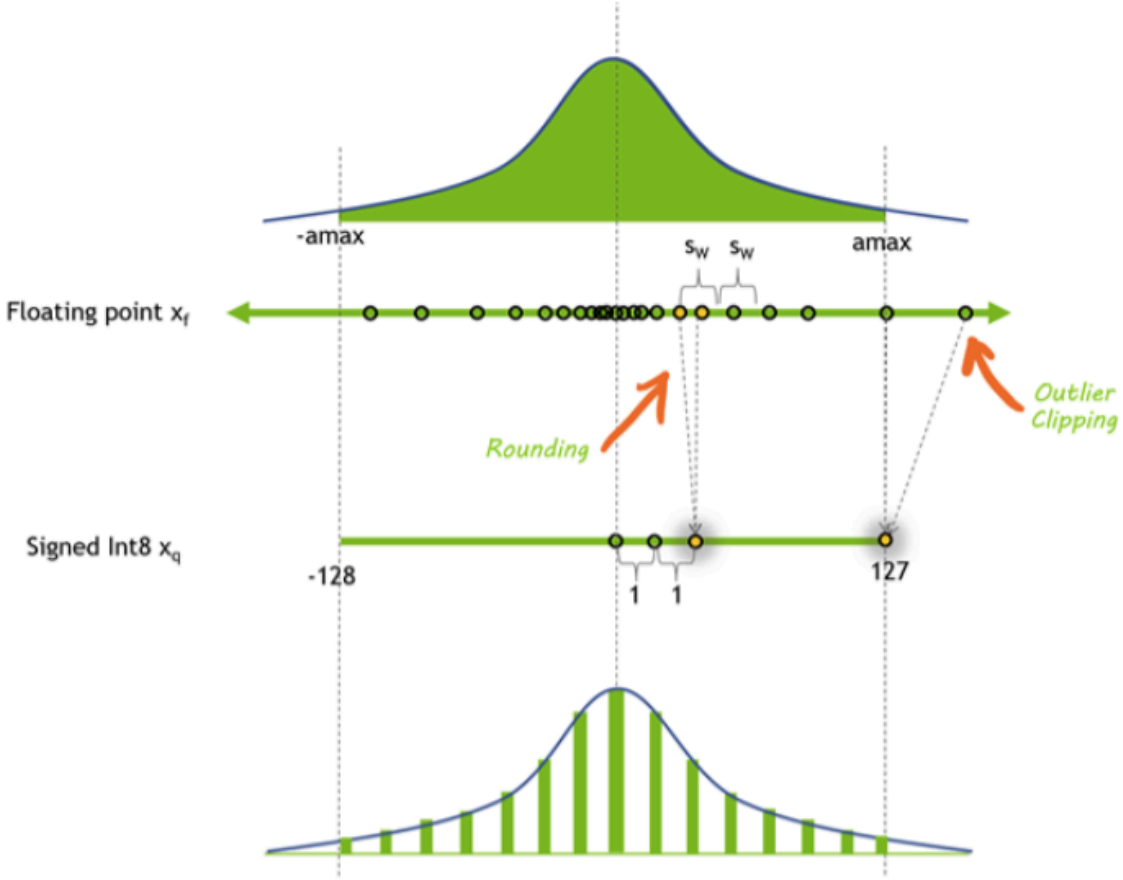
到目前为止，我们已经讨论了 LLM 消耗内存的不同方式、跨多个不同 GPU 分配内存的一些方式，以及优化注意力机制和 KV 缓存。还有多种模型优化技术可以通过修改模型权重本身来减少每个 GPU 上的内存使用。 GPU 还具有专用硬件来加速这些修改值的运算，从而为模型提供更多加速。

5.1 量化（Quantization）

量化是降低模型权重和激活精度的过程。大多数模型都以 32 或 16 位精度进行训练，其中每个参数和激活元素占用 32 或 16 位内存（单精度浮点）。然而，大多数深度学习模型可以用每个值八个甚至更少的位来有效表示。

图 9 显示了一种可能的量化方法之前和之后的值分布。在这种情况下，舍入会丢失一些精度，并且剪裁会丢失一些动态范围，从而允许以更小的格式表示值。

- 这一层展示了如何将浮点数 `xᵢ` 映射到 Signed Int8 范围 `[-128, 127]`。
- 步骤一：缩放（Scaling）
  - 将原始范围 `[-amax, amax]` 线性映射到 `[-128, 127]`。
  - 缩放因子 `s_w = amax / 127`（因为 127 是 Int8 的最大正值）。
  - 公式：`x_q = round(x_f / s_w)`
- 步骤二：舍入（Rounding）
  - 把缩放后的浮点数四舍五入到最接近的整数。
  - 图中用橙色箭头指向一个例子：某个浮点数被映射到 `1`。
- 步骤三：异常值裁剪（Outlier Clipping）
  - 对于那些大于 `amax` 的极端值（outliers），直接“裁剪”到 `127`。
  - 图中用红色箭头指向一个超出 `amax` 的点，它被强制映射到 `127`。



> 图9 一种可能的量化方法之前和之后的值分布

降低模型的精度可以带来多种好处。如果模型占用的内存空间较少，则可以在相同数量的硬件上安运行更大的模型。量化还意味着可以在相同的带宽上传输更多参数，这有助于加速带宽有限的模型。

LLM 有许多不同的量化技术，涉及**降低激活、权重或两者的精度**。量化权重要简单得多，因为它们在训练后是固定的。然而，这可能会留下一些性能问题，因为激活仍然保持在更高的精度。 GPU 没有用于乘以 INT8 和 FP16 数字的专用硬件，因此必须将权重转换回更高精度以进行实际运算。

还可以量化激活、Transformer块和网络层的输入，但这也有其自身的挑战。激活向量通常包含异常值，有效地增加了它们的动态范围，并使以比权重更低的精度表示这些值变得更具挑战性。

一种选择是通过模型传递代表性数据集并选择以比其他激活更高的精度表示某些激活来找出这些异常值可能出现的位置（`LLM.int8()`）。另一种选择是借用易于量化的权重的动态范围，并在激活中重用该范围。

> **Note**
>
> 量化类型：降低**权重**、**激活**，还是**两者**?
>
> 量化本质上是将高精度数值（如 FP16 / FP32）转换为低精度整数（如 INT8 / INT4），以节省内存和加速计算。根据**量化对象不同**，可分为三类：

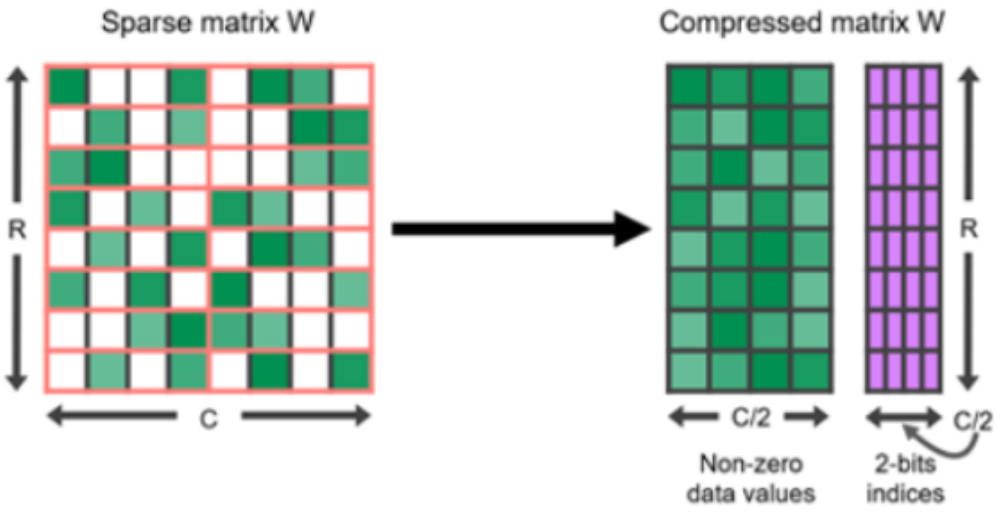
类型	量化对象	特点	难度	典型方法
Weight-only Quantization（仅权重量化）	只量化模型权重（W）	权重训练后固定，易于离线量化 显存大幅减少（如 INT4 权重 → 1/8 原始大小） 激活仍为 FP16，计算时需反量化	⭐ 简单	GPTQ, AWQ, SqueezeLLM
Activation Quantization（激活量化）	只量化激活（A）	激活动态范围大、含异常值 需在推理时动态确定量化参数	⭐⭐⭐ 困难	SmoothQuant, OmniQuant
Weight + Activation Quantization（全量化）	同时量化 W 和 A	可完全使用 INT8 计算（如 Tensor Core） 端到端加速显著 对异常值极其敏感，精度易崩	⭐⭐⭐⭐ 极难	LLM.int8(), FP8, QLoRA（部分）

🔍 关键细节补充：

- **为什么"仅量化权重"更简单?**
  权重是静态的，可以在训练后用校准数据集一次性确定量化参数（如缩放因子）。而激活是动态的，每层每 token 都不同。
- **GPU 硬件限制**:
  大多数 GPU（如 A100/H100）的 Tensor Core **只支持 INT8×INT8 → INT32 或 FP8×FP8。**
  如果权重是 INT8，但激活是 FP16，**无法直接用 INT8 硬件单元计算**，必须先把权重反量化为 FP16 → **失去计算加速优势**!

5.2 稀疏（Sparsity）

与量化类似，事实证明，许多深度学习模型对于修剪或用 `0` 本身替换某些接近 `0` 的值具有鲁棒性。稀疏矩阵是许多元素为 0 的矩阵。这些矩阵可以用压缩形式表示，比完整的稠密矩阵占用的空间更少。



> 图10，以压缩格式表示的稀疏矩阵，由非零数据值及其相应的两位索引组成

GPU 尤其具有针对某种结构化稀疏性的硬件加速，其中每四个值中有两个由零表示。稀疏表示还可以与量化相结合，以实现更大的执行速度。寻找以稀疏格式表示大型语言模型的最佳方法仍然是一个活跃的研究领域，并为未来提高推理速度提供了一个有希望的方向。

5.3 蒸馏（Distillation）

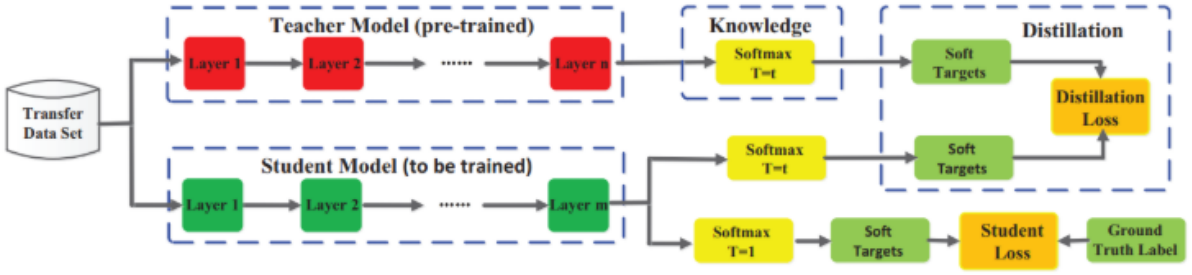
缩小模型大小的另一种方法是通过称为蒸馏的过程**将其知识转移到较小的模型**。此过程涉及训练较小的模型（称为学生）来模仿较大模型（教师）的行为。

蒸馏模型的成功例子包括 DistilBERT，它将 BERT 模型压缩了 40%，同时保留了 97% 的语言理解能力，速度提高了 60%。

虽然LLMs中的蒸馏是一个活跃的研究领域，但神经网络的一般方法首次在Distilling the Knowledge in a Neural Network中提出：

- 学生网络经过训练，可以反映较大教师网络的性能，使用损失函数来测量其输出之间的差异。该目标还可能包括将学生的输出与真实标签进行匹配的原始损失函数。
- 匹配的教师输出可以是最后一层（称为 `logits`）或中间层激活。

图 11 显示了知识蒸馏的总体框架。教师的 `logits` 是学生使用蒸馏损失进行优化的软目标。其他蒸馏方法可能会使用其他损失措施来从老师那里“蒸馏”知识。



蒸馏的另一种方法是使用教师合成的数据对LLMs学生进行监督培训，这在人工注释稀缺或不可用时特别有用。一步一步蒸馏！更进一步，除了作为基本事实的标签之外，还从LLMs教师那里提取基本原理。这些基本原理作为中间推理步骤，以数据有效的方式培训规模较小的LLMs。

值得注意的是，当今许多最先进的LLMs都拥有限制性许可证，禁止使用他们的成果来训练其他LLMs，这使得找到合适的教师模型具有挑战性。

> **Tip**
>
> **核心思想：让学生模型模仿教师模型的“软标签”（soft targets），而不是只学“硬标签”（ground truth），从而学到教师模型内部更丰富的知识。**

步骤	操作	目的
1. 输入数据	给教师和学生模型相同的输入	确保对比公平
2. 教师输出	用 T>1 的 softmax 得到软标签	提供更丰富的知识
3. 学生输出	用相同 T 得到自己的软标签	学习教师的分布
4. 蒸馏损失	KL 散度比较软标签	让学生模仿教师
5. 真实标签损失	交叉熵比较硬标签	确保准确性
6. 总损失	加权求和	平衡模仿与真实

6.模型服务技术

模型执行通常受内存带宽限制，特别是权重中的带宽限制。即使在应用了前面描述的所有模型优化之后，它仍然很可能受到内存限制。因此，在加载模型权重时尽可能多地处理它们。换句话说，尝试并行。可以采取两种方法：

- 动态批处理(**In-flight batching**)：同时执行多个不同的请求。
- 预测推理(**Speculative inference**)：并行执行序列的多个不同步骤以尝试节省时间。

6.1 动态批处理（In-flight batching)

LLMs 具有一些独特的执行特征，这些特征可能导致在实践中难以有效地处理批量请求。一个模型可以同时用于多种不同的任务。从聊天机器人中的简单问答响应到文档摘要或代码块的生成，工作负载是高度动态的，输出大小变化几个数量级。

这种多功能性使得批处理请求并有效地并行执行它们变得具有挑战性，这是服务神经网络的常见优化。这可能会导致某些请求比其他请求更早完成。

为了管理这些动态负载，许多LLMs 服务解决方案包括一种称**为连续或动态批处理的优化调度技术**。这利用了这样一个事实：**LLMs的整个文本生成过程可以分解为模型上的多次执行迭代。**

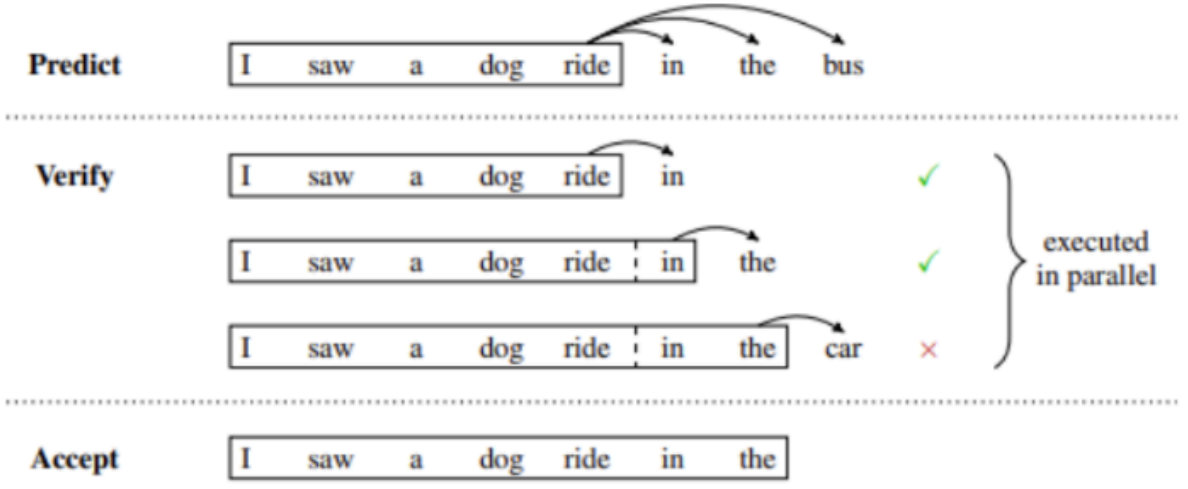
通过动态批处理，服务器运行时会**立即从批处理中剔除已完成的序列，而不是等待整个批处理完成后再继续处理下一组请求。**然后，它开始执行新请求，而其他请求仍在进行中。因此，动态批处理可以极大地提高实际用例中 GPU 的整体利用率。

6.2 预测推理（Speculative inference)

预测推理也称为推测采样、辅助生成或分块并行解码，是并行执行 LLM 的另一种方式。通常，GPT 风格的大语言模型是自回归模型，逐个生成文本标记。

生成的每个标记都依赖于它之前的所有标记来提供上下文。这意味着在常规执行中，**不可能从同一个序列并行生成多个token，必须等待第 n 个token生成后才能生成 n+1 个token。**

图 12 显示了预测推理的示例，其中临时模型临时预测并行验证或拒绝的多个未来步骤。在这种情况下，临时模型中的前两个预测token被接受，而最后一个在继续生成之前被拒绝并删除。



预测性抽样提供了一种解决方法。这种方法的基本思想是使用一些“更便宜”的过程来生成几个token长的临时序列。然后，并行执行多个步骤的主要“验证”模型，使用廉价临时序列作为需要的执行步骤的“预测”上下文。

如果验证模型生成与临时序列相同的token，那么就知道接受这些token作为输出。否则，可以丢弃第一个不匹配标记之后的所有内容，并使用新的临时序列重复该过程。

如何生成临时token有许多不同的选项，每个选项都有不同的权衡。可以训练多个模型，或在单个预训练模型上微调多个头，以预测未来多个步骤的标记。或者，可以使用小型模型作为临时模型，使用更大、功能更强大的模型作为验证器。

总结：**用大模型（验证器）并行验证小模型（草稿模型）预测的 token 是否“符合大模型自己的意愿**

概念	说明
草稿模型（Draft）	小、快，用于“猜”未来 token
目标模型（Target）	大、准，用于“验证”草稿是否可信
验证方式	并行计算 logits，逐个比对 argmax
不是和真实标签比	因为没有真实标签！是自回归生成
核心收益	减少大模型的串行调用次数，提升吞吐

7.结论

这篇文章概述了许多最流行的解决方案，以帮助高效地优化和服务LLMs，无论是在数据中心还是在 PC 边缘。其中许多技术都经过优化并通过 NVIDIA TensorRT-LLM 提供，这是一个开源库，由 TensorRT 深度学习编译器以及优化的内核、预处理和后处理步骤以及多 GPU/多节点通信原语组成，可在 NVIDIA 上实现突破性的性能GPU。