11 大模型推理

徐辉，xuh@fudan.edu.cn

本章学习目标：

- ⋆⋆⋆ 理解模型参数量化的基本原理
- ⋆⋆⋆ 理解 KV Cache 在自回归推理中的加速机制
- ⋆⋆⋆ 理解 FlashAttention 的计算与内存优化思想
- ⋆⋆ 了解 vLLM 和 SGLang 工具和关键优化

由于大模型参数规模庞大且中间计算状态占用显存较高，其推理过程往往受到内存带宽与计算资源的双重限制，导致延迟增加与吞吐下降。因此，提升推理效率成为大模型系统工程中的核心问题之一。本节主要介绍三种主流的推理优化技术：参数量化、KV Cache 以及 FlashAttention。

11.1 参数量化

参数量化（Quantization）是通过降低模型参数与计算中间变量的数值精度，从而减少显存占用并提升推理效率的重要技术。其核心思想是在尽可能保持模型性能的前提下，将高精度浮点表示（如 FP32）映射为低比特浮点数甚至整数表示。

11.1.1 整数与浮点数

在深度学习系统中，数值类型不仅决定了模型参数的存储开销，也直接影响硬件计算复杂度、数值精度以及推理效率。常见的数据格式主要包括浮点数与整数两类。

整数 整数类型采用定点（Fixed-Point）表示方式，即每一个比特位具有固定的数值含义。对于有符号整数（Signed Integer），通常采用二进制补码（Two's Complement）表示，其最高位为符号位（0 表示正数，1 表示负数）；其余位用于表示整数数值。

以 INT32 的整数 7 为例，其二进制表示为：

00000000 00000000 00000000 00000111

负数通常采用二进制补码表示。对于-7，首先写出正数 7 的二进制，随后按位取反

11111111 11111111 11111111 11111000

再加 1 即可得到：

11111111 11111111 11111111 11111001

因此，对于一个 n bit 的有符号整数，其可表示范围为：$[-2^{n-1}, 2^{n-1}-1]$

浮点数 浮点数（Floating-Point）可以支持小数或更大的数值范围，其基本结构通常为：

$$\text{Value} = (-1)^s \times m \times 2^e$$

其中：

- s：符号位（Sign Bit）
- e：指数（Exponent）
- m：尾数（Mantissa 或 Fraction）

FP32（单精度浮点数）是传统深度学习训练的标准格式，其结构如下：

$$1 \text{ bit Sign} + 8 \text{ bit Exponent} + 23 \text{ bit Mantissa}$$

以十进制小数 0.1 为例，将其转换为 FP32 需要首先将其转换为二进制小数：

$$0.1_{10} = 0.00011001100110011\cdots_2$$

可以看到，0.1 在二进制中是无限循环小数，因此无法被 FP32 精确表示。将其规格化：

$$0.0001100110011\cdots = 1.1001100110011\cdots \times 2^{-4}$$

因此得到，0.1 转化为 FP32 表示后的指数为-4，加上 IEE754 标注中规定的偏置 127 后实际指数为 123 （01111011_2），尾数部分是 $10011001100110011001100_2$。因此，FP32 中 0.1 的完整表示约为：

$$0 \mid 01111011 \mid 10011001100110011001100$$

由于尾数位数有限，FP32 实际保存的并不是精确的 0.1，而是：

$$0.10000000149011612$$

因此，浮点数计算通常会存在误差。这也是为什么在深度学习训练中，经常需要关注数值稳定性问题。表 11.1 总结了常见的一些数据类型。

表 11.1: 常见 FP 与 INT 数据格式结构与数值范围对比

数据类型	总位宽	符号位	指数位	尾数位	数值范围
FP32	32 bit	1	8	23	$\approx \pm 3.4 \times 10^{38}$
FP16	16 bit	1	5	10	$\approx \pm 6.55 \times 10^{4}$
FP8 (E4M3FN)	8 bit	1	4	3	$\approx [-240, 240]$
FP8 (E5M2)	8 bit	1	5	2	$\approx [-57344, 57344]$
INT32	32 bit	1	–	–	$[-2^{31},\ 2^{31}-1]$
INT16	16 bit	1	–	–	$[-2^{15},\ 2^{15}-1]$
INT8	8 bit	1	–	–	$[-2^{7},\ 2^{7}-1]$
INT4	4 bit	1	–	–	$[-2^{3},\ 2^{3}-1]$

参数量化 参数量化（Quantization）是指将浮点数转换为整数表示，或采用更低比特宽度的数据类型来表示数值的一种模型压缩与加速技术。其主要目标包括两个方面：一是提升模型推理阶段的运算性能，二是降低模型的存储开销与显存占用。

相比浮点运算，整数运算无需处理指数对齐、尾数规格化以及浮点舍入等复杂操作，因此硬件实现更加简单，能够在相同芯片资源下实现更高的计算并行度与吞吐率。因此，将同等位宽的浮点运算转换为整数运算，通常可以显著提升推理效率。此外，无论是浮点数还是整数，采用更低位宽的数据类型都能够进一步降低计算开销。例如，相比 FP32，FP16、FP8、INT8 以及 INT4 的数据位宽更小，因此在矩阵乘法过程中能够减少显存带宽压力，并提高缓存利用率与计算吞吐率。另一方面，低比特表示还能显著减少模型参数存储空间。例如，相比 FP16，INT8 的存储开销降低约 50%，而 INT4 则仅需约 25% 的存储空间。这对于大模型部署尤为重要，因为模型权重通常占据主要显存开销。因此，现代大模型系统通常会结合低比特整数计算与混合精度技术，在尽可能保持模型精度的前提下，实现更高的推理速度与更低的部署成本。

当前主流的大模型量化方法多采用训练后量化策略，即在模型训练完成后直接对权重进行量化处理，而无需重新训练模型或对模型结构进行修改。参数量化通常通过缩放与离散映射的方式实现。

- **浮点 → 低精度浮点（FP32 → FP16 / FP8）**：该类量化过程不引入缩放操作，仅通过截断尾数位与指数位实现精度压缩，同时保留浮点结构。因此其数值误差主要来源于舍入，而非量化映射误差。
- **浮点 → 整数（FP → INT8 / INT4）**：由于同比特浮点数表示的数值范围远大于整数，该类量化需要缩放操作，将浮点空间映射到离散整数空间。数学形式为：

$$q = \text{round}\left(\frac{x}{s}\right)$$

其中：

  - x：原始浮点数
  - q：量化后的整数表示
  - s：缩放因子

从可量化性的角度来看，神经网络主要由线性变换（如矩阵乘法）构成，这类运算在数值上对均匀缩放具有较好的等价性，使得参数在不同精度表示下仍能保持相对稳定的计算行为。同时，归一化（Normalization）与 ReLU 等非线性操作通过对激活分布进行重标定或截断，在一定程度上改善了数值分布的稳定性与动态范围，从而降低了低比特量化对模型精度的影响。因此，在上述结构特性共同作用下，模型在低精度表示下通常仍能够保持较好的数值稳定性与表达能力。

11.2 KV Cache

在大语言模型的自回归生成过程中，模型逐个词元生成输出。若每次生成新词元都重新计算所有历史词元的 Key 和 Value，将导致大量重复的线性投影计算。KV Cache（Key-Value Cache）技术通过缓存历史 Key/Value，使得在生成第 t 个词元时无需重复计算 $x_{1:t-1}$ 的 Key 和 Value，仅需计算当前词元的表示，从而显著降低推理过程中的重复计算开销，是 LLM 推理加速的核心机制。

问题：自回归生成的重复计算 考虑生成第 t 个词元时，输入序列为 $x_{1:t} = [x_1, x_2, \ldots, x_t]$。标准注意力计算为：

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_t K_{1:t}^\top}{\sqrt{d}}\right) V_{1:t}$$

其中，$Q_t = x_t W_Q$ 表示当前词元的 query；$K_{1:t} = [x_1; \ldots; x_t] W_K$ 表示历史词元的 key；$V_{1:t} = [x_1; \ldots; x_t] W_V$ 表示历史词元的 value。

在自回归生成过程中，当生成第 $t+1$ 个词元时，$K_{1:t}$ 与 $V_{1:t}$ 与上一轮计算结果完全相同，但标准实现仍会重复计算 $K_{1:t+1}$ 和 $V_{1:t+1}$，导致计算冗余，并使整体推理复杂度随序列长度呈平方级增长（主要来自 attention score 计算）。

KV Cache 核心思想 将已计算的 Key 和 Value 缓存在显存中，后续步骤直接复用，仅计算当前 token 的 K_t, V_t。具体做法如下：

- 初始化空缓存：$\text{Cache}_K = []$，$\text{Cache}_V = []$
- 第 1 步：计算 K_1, V_1，存入缓存
- 第 2 步：计算 K_2, V_2，追加到缓存
- 第 t 步：仅计算 K_t, V_t 并追加到缓存

此时注意力计算变为：

$$\text{Attention}(Q_t, K_{1:t}^{cache}, V_{1:t}^{cache})$$

从计算复杂度来看，KV Cache 将每步 Key/Value 的重复投影计算从 $O(td^2)$ 降低为 $O(d^2)$（仅当前 token），从而显著降低自回归推理开销。然而，其代价是显存开销随序列长度线性增长。对于 L 层模型、隐藏维度 d、序列长度 T，KV Cache 的显存需求为：

$$\text{Memory}_{KV} = 2 \times L \times T \times d \times \text{bytes}$$

示例：以 LLaMA 3 8B（$L = 32$, $d = 4096$）为例，在 $T = 2048$、FP16 条件下，如果不考虑其它优化，KV Cache 显存可近似估算为：

$$2 \times 32 \times 2048 \times 4096 \times 2 \approx 1 \text{ GB}$$

11.3 FlashAttention

传统的缩放点积注意力可以写为：

$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V. \tag{11.1}$$

其中 $Q,K,V\in\mathbb{R}^{N\times d}$，注意力分数矩阵为：

$$S=\frac{QK^\top}{\sqrt{d}}\in\mathbb{R}^{N\times N}.$$

因此，标准注意力在计算过程中需要显式构造并存储一个 $N\times N$ 的中间矩阵。随着序列长度 N 增大，这种二次规模的显存开销会成为 Transformer 的主要瓶颈。FlashAttention 的核心目标是重新组织计算过程，使得该中间矩阵无需完整存储到显存中，而是以分块（block-wise）和流式（online）的方式进行计算与累积。

对于第 i 个 query，其输出可以展开为：

$$O_i=\frac{\sum_{j=1}^{N}\exp(S_{ij})V_j}{\sum_{j=1}^{N}\exp(S_{ij})}. \tag{11.2}$$

这里可以看到，注意力由两部分组成：上方是未归一化的加权和，下方是 softmax 的归一化分母。由于二者都是求和结构，因此可以按 block 进行分解：

$$\sum_{j=1}^{N}=\sum_{b=1}^{B}\sum_{j\in\mathcal{J}_b},\quad \{\mathcal{J}_b\}_{b=1}^{B}\text{ 为对索引的分块划分.}$$

也就是说，如果能够逐块处理 key/value，而不是一次性构造完整的 $N\times N$ 注意力矩阵，就可以避免中间显存的巨大开销。

关键难点 softmax 依赖全局归一化项，并不满足简单的线性可加性。直接计算 $\exp(S_{ij})$ 在数值上容易出现溢出或下溢问题，因此实际实现中利用 softmax 的平移不变性，将其改写为：

$$P_{ij}=\frac{\exp(S_{ij}-m_i)}{\sum_k\exp(S_{ik}-m_i)},\qquad m_i=\max_j S_{ij}. \tag{11.3}$$

于是 attention 输出可写为：

$$O_i=\frac{\tilde{O}_i}{l_i}, \tag{11.4}$$

其中未归一化加权和为：

$$\tilde{O}_i=\sum_j\exp(S_{ij}-m_i)V_j, \tag{11.5}$$

归一化因子为：

$$l_i=\sum_j\exp(S_{ij}-m_i). \tag{11.6}$$

FlashAttention 的关键在于：即使注意力被拆分为多个 block，也可以在线维护 softmax 的正确归一化结果。对每个 query 行，只需维护三个量：

$$(m_i,\,l_i,\,\tilde{O}_i),$$

其中：

- m_i：当前已经看到的 attention score 最大值；
- l_i：softmax 归一化因子；
- $\tilde{O}_i$：未归一化的 value 加权和。

当处理新的 block 时，首先计算该 block 内的局部最大值 m_i^{block}，并更新全局最大值：

$$m_i^{new} = \max(m_i^{old}, m_i^{block}). \tag{11.7}$$

FlashAttention 的在线更新依赖指数函数的平移性质：

$$\exp(x - m^{new}) = \exp(x - m^{old}) \cdot \exp(m^{old} - m^{new}). \tag{11.8}$$

因此，当 softmax 的归一化基准从 m^{old} 变化到 m^{new} 时，旧 block 的所有指数项都只需乘同一个缩放因子即可完成坐标系转换。定义：

$$\alpha = \exp(m_i^{old} - m_i^{new}). \tag{11.9}$$

因此归一化项可以递推更新为：

$$l_i^{new} = l_i^{old}\alpha + \sum_{j \in block} \exp(S_{ij} - m_i^{new}). \tag{11.10}$$

类似地，未归一化输出也满足相同的缩放关系：

$$\tilde{O}_i^{new} = \tilde{O}_i^{old}\alpha + \sum_{j \in block} \exp(S_{ij} - m_i^{new})V_j. \tag{11.11}$$

当所有 block 处理完成后，只需进行一次最终归一化即可得到 attention 输出：

$$O_i = \frac{\tilde{O}_i}{l_i}. \tag{11.12}$$

整个过程中，算法无需显式存储完整的 $N \times N$ 注意力矩阵，而是分块完成计算与累积，从而显著降低 HBM 显存访问开销，复杂度从 $O(N^2)$ 降低为 $O(N)$。。

算法总结 FlashAttention 的在线计算过程总结如下：

- 将 K, V 按 block 划分；
- 对每个 block 计算局部 attention score；
- 更新全局最大值 m_i；
- 对旧结果进行指数缩放修正；
- 累积归一化项与输出值；
- 最终进行一次归一化得到输出。

11.4 大模型部署

实际大模型部署通常依赖专用推理框架，以提升吞吐率、降低延迟并优化显存使用。典型工具包括 vLLM[1] 与 SGLang[2]。这些系统的核心目标不是改变模型结构，而是优化推理过程中的内存管理与执行调度。

11.4.1 vLLM

vLLM 是一个开源的大语言模型推理系统，由加州大学伯克利分校提出并开源实现，已成为工业界最广泛使用的 LLM serving 框架之一。

Paged Attention vLLM 的核心创新是 PagedAttention，其本质是对 KV Cache 的分页式管理。传统 Transformer 推理中，KV Cache 需要为每个请求分配连续显存：$K, V \in \mathbb{R}^{T \times d}$。这种方式的问题在于必须按最大上下文长度预分配显存。由于不同请求长度不同，导致显存浪费。尤其在多请求并发时容易产生显存碎片。

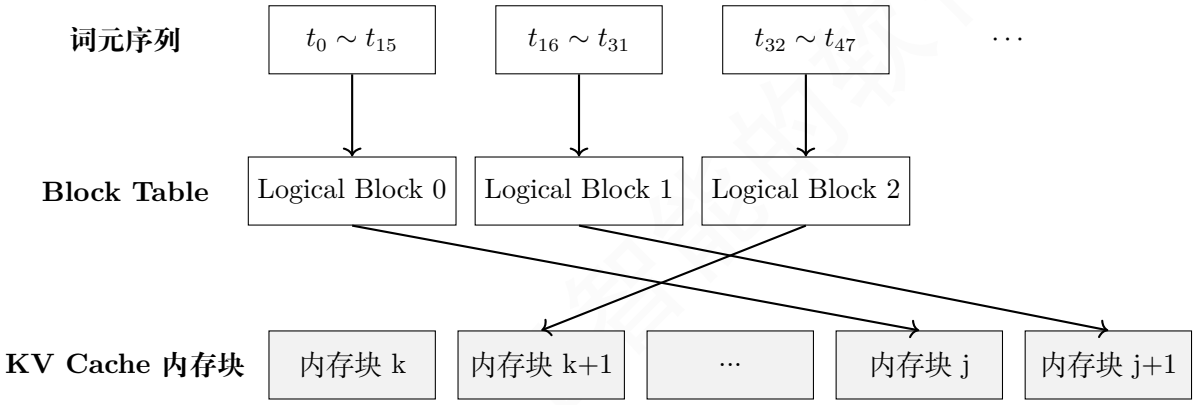

图 11.1: PagedAttention 中的分页式 KV Cache 管理

如图 11.1所示，PagedAttention 的关键思想是：

- 将 KV Cache 切分为固定大小的 block（例如 16 或 32 个词元）
- 每个请求的 KV 不再要求连续存储
- 使用 block table 维护“词元 → 内存块”的映射关系

因此，KV Cache 的访问从连续数组变为：词元逻辑索引 → block table → KV 块内存地址。这种设计使得显存分配更接近操作系统中的“分页内存管理”，从而减少碎片并提升利用率。

安装和使用 安装：

```
pip install vllm
```

启动服务（kaggle T4 环境下测试通过）：

```
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen2.5-1.5B-Instruct \
```

[1] https://github.com/vllm-project/vllm

[2] https://github.com/sgl-project/sglang

```
--host 0.0.0.0 \
--port 8000 \
--max-num-seqs 2 \
--max-model-len 1024 \
--attention-backend TRITON_ATTN \
--gpu-memory-utilization 0.85 \
--tensor-parallel-size 1
```

参数说明 vLLM 的关键运行参数包括：

- `--model`：指定加载的模型名称或本地路径，例如 `Qwen/Qwen2.5-1.5B-Instruct`
- `--host`：服务绑定地址。`0.0.0.0` 表示允许局域网或外部机器访问
- `--port`：HTTP API 服务监听端口，例如 `8000`
- `--max-num-seqs`：单次调度中允许并发处理的最大请求数，会影响吞吐量与显存占用
- `--max-model-len`：模型支持的最大上下文长度（token 数），该值越大，KV Cache 显存占用越高
- `--attention-backend`：指定 Attention 计算后端，例如 `TRITON_ATTN` 使用 Triton 实现的高性能 Attention Kernel
- `--gpu-memory-utilization`：GPU 显存使用比例上限，例如 `0.85` 表示最多使用 85% 显存
- `--tensor-parallel-size`：Tensor Parallelism 使用的 GPU 数量。`1` 表示单 GPU 运行

启动以后，vLLM 默认会在本地 `8000` 端口启动一个兼容 OpenAI API 的 HTTP 服务：`http://localhost:8000/v1`

11.4.2 SGLang

SGLang 是由 LMSYS 提出的开源大语言模型推理与编排框架，其设计目标是提升复杂 LLM 应用（如多轮对话和工具调用）的执行效率，并在推理层面进一步优化 KV cache 复用与前缀共享能力。

Radix Attention SGLang 的核心优化机制是 Radix Attention，其本质是一种基于前缀树的 KV cache 复用机制，用于在多请求场景下共享重复的 prompt 前缀计算。

在传统 Transformer 推理中，KV cache 被用于在单个序列内部复用历史词元的 Key-Value 表示，从而避免在自回归生成过程中对历史上下文进行重复计算。具体而言，在同一请求的生成过程中，模型仅需对新增词元计算 attention，而历史词元的 KV 状态可以直接复用，从而显著降低计算复杂度。然而，在实际服务化部署中，多轮对话通常并不以“单一连续序列”的形式存在，而是采用 request-based 的执行范式：每一轮用户输入都会被重新组织为包含 system prompt、历史对话与当前输入的完整 prompt，并作为独立请求提交给推理引擎。因此，即使在同一会话内部，不同轮次之间仍然会产生重复的前缀计算。更进一步，在多用户或高并发场景中，这种前缀重复会跨请求广泛存在，例如 system prompt、模板化指令或 few-shot 示例在不同请求之间高度共享。由于标准 KV cache 的复用范围仅限于单个请求生命周期，因此无法跨请求共享这些重复前缀的 attention 计算。

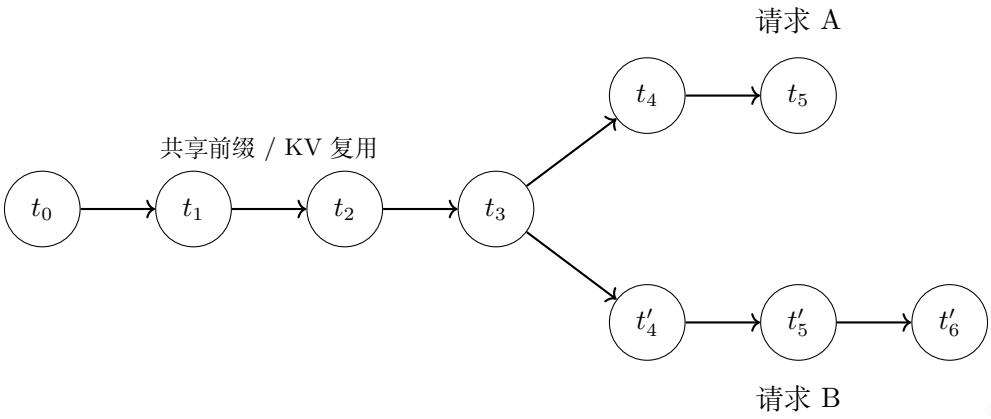

图 11.2: 基于 Radix Tree 的 KV Cache 复用

为了解决这一问题，Radix Attention 引入基于 radix tree 的前缀索引结构，对 token 序列进行结构化组织，从而实现跨请求的 KV cache 共享。如图 11.2所示，对于请求集合 $\{t^{(i)}\}$，当其词元序列在 radix tree 上存在可复用的公共前缀时，即不同请求共享相同的前缀路径，则这些词元对应的 KV 值只需计算一次，并由 radix tree 中的共享节点进行复用，从而减少重复的 attention 计算开销。

为了更好地发挥 RadixAttention 的性能，SGLang 在系统设计上另外引入了多项关键机制，主要包括结构化生成与程序化 Prompt 执行等。

安装与使用 安装：

```
pip install sglang
```

启动 OpenAI 兼容服务：

```
python -m sglang.launch_server \
    --model-path XXX \
    --port 8000
```

参数说明 SGLang 在启动服务时提供了一系列关键配置参数，用于控制模型加载、并行策略以及运行时性能优化，常见参数如下：

- `--model-path`：指定模型路径或 HuggingFace 模型名称（支持模型列表[3]）
- `--port`：服务监听端口，用于提供 OpenAI-compatible HTTP API
- `--host`：服务绑定地址，默认通常为 `0.0.0.0`
- `--tensor-parallel-size`：张量并行 GPU 数量，用于分布式推理加速
- `--dtype`：计算精度类型，如 fp16、bf16
- `--max-total-tokens`：系统允许的最大词元总数，用于控制 KV cache 上限
- `--context-length`：模型支持的最大上下文长度
- `--disable-radix-cache`：关闭 RadixAttention cache

[3] `https://sgl-project.github.io/supported_models/text_generation/generative_models.html`

调用示例 vLLM 和 SGLang 提供的服务均兼容 OpenAI API 接口，因此可以直接使用 OpenAI 官方 Python SDK 进行调用。代码 11.1展示了一个基于历史消息维护的多轮对话 Agent，实现与本地大模型服务的持续交互。

```python
from openai import OpenAI

# vLLM 服务地址
client = OpenAI(
    api_key="EMPTY",  # vLLM 默认不校验
    base_url="http://127.0.0.1:8000/v1",
)

# 对话历史
messages = [
    {
        "role": "system",
        "content": "你是一个有帮助的AI助手"
    }
]

def chat_with_qwen(user_input: str):
    # 加入用户消息
    messages.append({
        "role": "user",
        "content": user_input
    })
    # 调用 vLLM
    response = client.chat.completions.create(
        model="Qwen/Qwen2.5-1.5B-Instruct",
        messages=messages,
        temperature=0.7,
        max_tokens=512,
    )
    # 取回复
    reply = response.choices[0].message.content
    # 加入历史
    messages.append({
        "role": "assistant",
        "content": reply
    })
    return reply

if __name__ == "__main__":
    print("=== Local Qwen (vLLM) ===")
    while True:
        user_input = input("You: ")
        if user_input.strip().lower() in {"exit", "quit"}:
            break
        reply = chat_with_qwen(user_input)
        print("Qwen:", reply)
```

代码 11.1: 基于 OpenAI SDK 的本地 vLLM/SGLang 多轮对话 Agent 示例

练习

1) 完成一个端到端的大模型推理系统实验，包括模型部署、API 调用与性能分析三个步骤。

   (a) **模型部署**：使用 vLLM 或 SGLang 在本地部署一个大语言模型服务（Kaggle 推荐使用 Qwen2.5-1.5B-Instruct 模型）。在 Kaggle 环境中进行实验时，建议将服务进程放到后台运行，可先将启动命令写入 shell 脚本，再通过脚本启动服务。

   (b) **API 调用测试**：使用 Python OpenAI SDK 或 curl 调用本地服务，完成至少一次多轮对话请求，验证服务正常运行。

   (c) **性能分析**：调整相关优化配置，观察并分析模型的显存占用和响应延迟变化。