XTuner + vLLM 强化学习架构设计与技术实现报告

作者：谭邵杰等

一、 项目背景

顺应多模态 RL 技术演进趋势，为支撑浦江大模型RL（例如，InternS1 241B ）的高效迭代，项目组针对 **GPU 侧 XTuner+LMDeploy 架构在 NPU 生态适配难** 的问题，依托 **NPU 成熟的 vLLM 技术底座**，成功研发并落地了适配国产算力的 **XTuner+vLLM 全流程方案**，实现了架构的国产化平滑演进。

二、 软件架构设计

首先，从软件架构设计层面来展开介绍XTuner+vLLM涉及的设计修改点。

#部署图，多少硬件承载，怎么部署的，32+2

GPU和NPU服务器架构均采用32+2，其中32机参与真实的训练和推理，2机只运行judger服务。

推理阶段：单机部署服务，单次下发128prompt*16个请求，即单机每次处理64个请求，4次下发共计处理512*16个请求；

judger：基于每个prompt的16个response，判断该prompt是否保留用于训练阶段输入；

训练阶段：FSDP框架下，单卡单次处理单条数据。以GPU服务器集群部署为例，共32*8=256张H200参与训练，单步训练消耗256条样本（prompt+response），故单步RL训练需要2048/256=8步

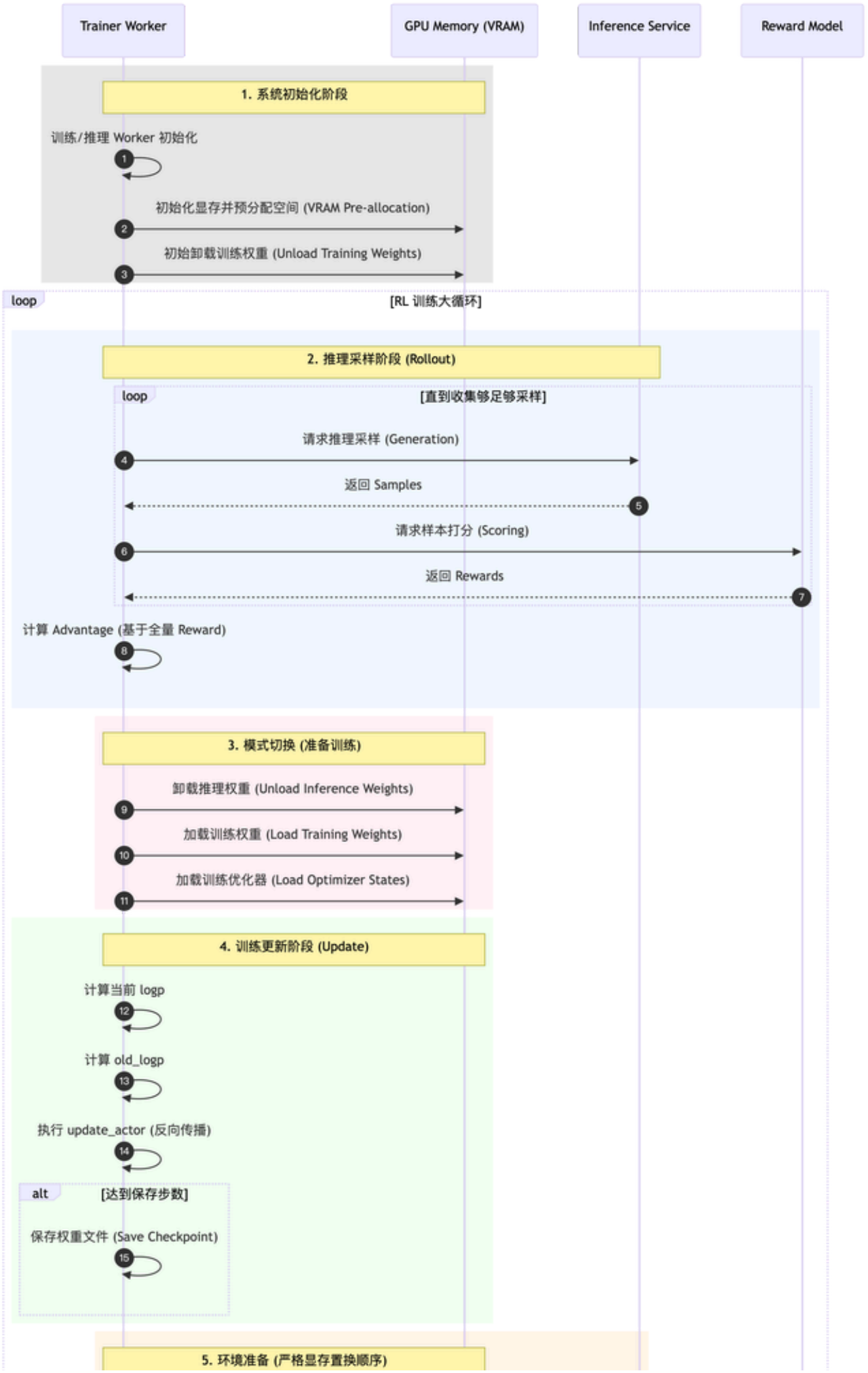
GPU和NPU集群拓扑架构对比分析

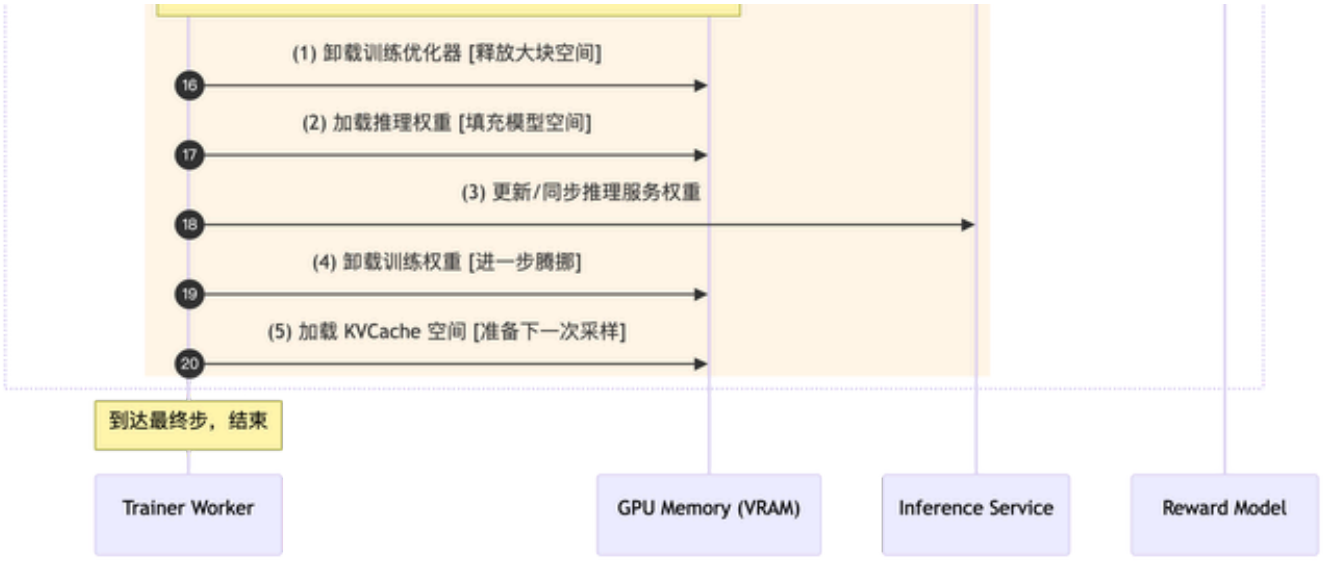
GPU：H200*256卡，FSDP训练框架下

NPU：A3 * 512die

RL整体流程

每个模块的部署，控制范围和边界


Trainer Worker
GPU Memory (VRAM)
Inference Service
Reward Model
1. 系统初始化阶段
训练/推理 Worker 初始化
1
初始化显存并预分配空间 (VRAM Pre-allocation)
2
初始卸载训练权重 (Unload Training Weights)
3
loop
[RL 训练大循环]
2. 推理采样阶段 (Rollout)
loop
[直到收集够足够采样]
请求推理采样 (Generation)
4
返回 Samples
5
请求样本打分 (Scoring)
6
返回 Rewards
7
计算 Advantage (基于全量 Reward)
8
3. 模式切换 (准备训练)
卸载推理权重 (Unload Inference Weights)
9
加载训练权重 (Load Training Weights)
10
加载训练优化器 (Load Optimizer States)
11
4. 训练更新阶段 (Update)
计算当前 logp
12
计算 old_logp
13
执行 update_actor (反向传播)
14
alt
[达到保存步数]
保存权重文件 (Save Checkpoint)
15
5. 环境准备 (严格显存置换顺序)



具体逻辑如下：

1. 系统初始化与预分配

在系统启动初期，架构会执行**静态显存分析**。

- **预分配空间**：通过预先划定显存池（Memory Pool），避免频繁调用 `cudaMalloc` 造成的内存碎片化。
- **初始卸载**：由于 RL 第一步是采样，系统会立即执行“卸载训练权重”操作，确保显存处于“推理 Ready”状态，将冗余的训练算子移出计算单元。

2. Rollout 阶段

在采样阶段，系统关注的是**生成吞吐量**。

- **推理采样与打分**：样本生成与 Reward Model 打分同步进行。
- **Advantage 后置计算**：[架构设计关键点] 区别于传统的即时计算，我们选择在**全量 Reward 收集完毕后统一计算 Advantage**。这种“批处理”模式减少了推理与计算之间的上下文切换，为后续的模式切换腾出了清晰的边界。

3. 模式切换

为了后续的正常训练

- 卸载推理权重 (Unload Inference Weights)：通过vLLM的sleep mode实现，sleep_mode=2可以把kvcache一并卸载。
- 加载训练权重 (Load Training Weights)
- 加载训练优化器 (Load Optimizer States)

4. 更新阶段

进入 Training 模式后：

- 计算当前 Logp 与 Old-Logp 的偏移，通过 `update_actor` 执行反向传播。

- **检查点存盘（Checkpointing）**：在权重更新后，根据配置步数执行异步存盘，确保模型在长周期 RL 训练中异常中断后能正常续训。

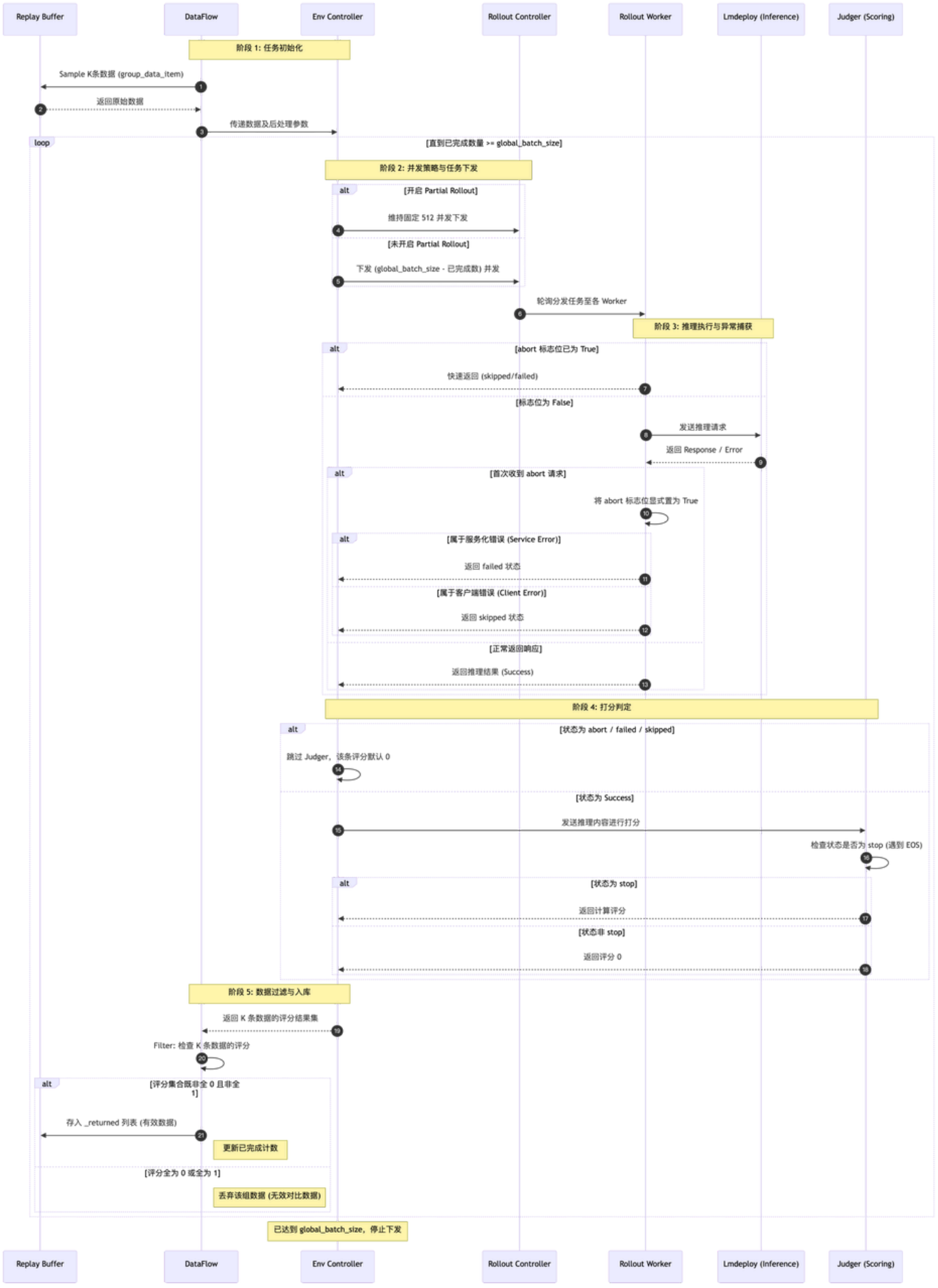
5. 显存切换逻辑

为了在有限的显存中塞进训练所需的巨大状态量，框架采用了**五步法严格时序卸载策略**：

1. **卸载训练优化器**：首先释放显存中占比不大的（通常是参数量 4 倍以上但被分布式优化器切分）优化器状态。
2. **加载推理权重**：在优化器腾出的空位中，精准置入推理所需的最新权重参数。
3. **服务权重同步**：触发推理引擎（Inference Service）的权重热更新，确保策略的一致性。
4. **卸载训练权重**：此时训练 Backbone 已完成其阶段性使命，将其移出显存以释放连续地址空间。
5. **加载 KVCache 空间**：最后一步，利用清理出的最大连续显存块分配给 KVCache。由于生成任务的长度具有不确定性，KVCache 的加载必须置于最后，以确保其拥有最大的动态伸缩空间。

推理模块流程

#每个模块的部署，控制范围和边界



原始xtuner+lmdeploy推理流程拆解：

1. **ReplayBuffer**

- 从数据集中采样 `K` 条组数据（`group_data_item`），返回原始数据。
- 接收 DataFlow 有效响应数据，将符合过滤规则的数据存入_returned缓冲区。

2. **DataFlow**
   - 接收 ReplayBuffer 采样的原始数据及后处理参数，传递至 EnvController。
   - 管理任务下发并发度：开启 `partial_rollout` 时维持 **512 固定并发任务**；关闭时按 `global_batch_size - 已完成数` 动态下发任务。
   - 接收 EnvController 回流的 `K` 条数据评分结果集，执行过滤逻辑：
     - 若 `K` 条数据评分全为 0 或全为 1，标记为无效对比数据并丢弃；
     - 否则将有效数据存入 ReplayBuffer。
   - 更新已完成任务计数，当计数达到 `global_batch_size` 时，停止下发新任务并触发 `abort` 信号。
3. **EnvController**
   - 接收 DataFlow 传递的单任务 `K` 条数据，分发至 RolloutController。
   - 聚合 RolloutController 回流的 `K` 条推理结果状态：
     - 若存在 `abort/skipped/failed` 状态，跳过打分，默认评分为 0；
     - 若全部为 `success` 状态，将数据推送至 Judger 进行打分。
   - 接收 Judger 评分结果后，将 `K` 条数据的评分结果集回流至 DataFlow。
4. **RolloutController**
   - 接收 EnvController 下发的单条数据任务，以轮询方式将任务调度至可用 RolloutWorker。
5. **RolloutWorker**
   - 承载单实例 lmdeploy 服务（单机 16 卡对应 1 个 Worker），将任务格式化为 lmdeploy 标准输入格式，例如token进token出，需要传入prompt的token ids，再将请求发给lmdeploy的generate接口。
   - 解析 lmdeploy 返回结果，提取 `response/token_ids/logprobs/finish_reason` 等关键字段。
   - 若检测到 `abort` 标志位为 True，终止后续未发送的请求，直接返回对应状态；否则等待 lmdeploy 响应并处理结果。
6. **Lmdeploy (Inference)**
   - 接收 RolloutWorker 的推理请求，执行模型推理。
   - 返回响应结果或错误状态：`success`（正常响应）、`Service Error`（服务化错误）、`Client Error`（客户端错误）。
   - 收到 `abort` 请求后，将标志位置为 True，终止当前正在处理的推理任务。

7. **Judger (Scoring)**
   - 接收 EnvController 推送的 `success` 状态响应数据，执行评分逻辑：
     - 若响应状态为 `stop`（EOS 触发结束）：打分正确得 1 分，错误得 0 分；
     - 若响应状态非 `stop`（如长度限制终止），默认得 0 分。
   - 将评分结果返回至 EnvController。

设计修改点（基础功能打通）

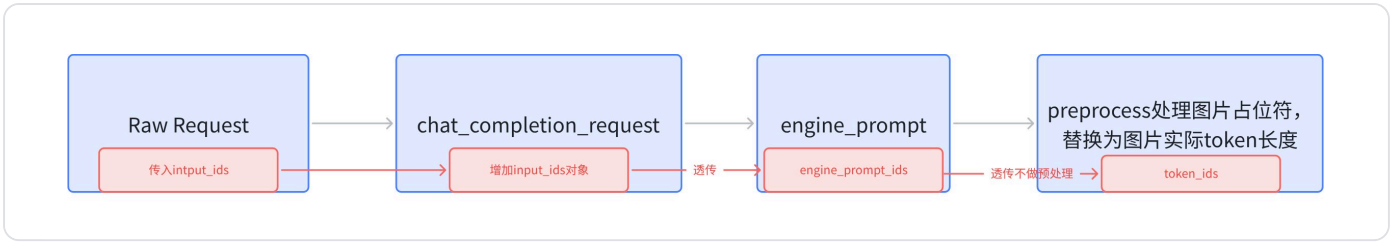
#每个功能点的原因和背景

修改背景	层级	LMDeploy（旧）	vLLM（新）	
相较于lmdeploy，vllm在npu上的开发加成熟。	**推理引擎**	Lmdeploy	vLLM Engine	后端整体替换
测量训推一致性，保证token对齐	**输入输出数据格式**	支持token进token出	不支持token进token出	vLLM内部新增token进t
加速rollout流程，提升rl整体性能	**模型并行**	TP16	DP4TP4	ray设备资源映射、权重
减少内存占用，从而增加kv_cache容量	**ViT 切分**	TP并行 切分	原始vllm无TP切分	新增ViT TP切分
为了vllm支持rl中partial rollout功能	**Abort请求功能**	支持abort服务化内所有请求	仅支持abort服务化内指定请求	增加abort服务化内所有

1. **推理框架替换与服务化初始化**：

将原有推理链路中的 lmdeploy 框架替换为 VLLM 框架；在启动推理服务前，先完成 VLLM 服务化所需的配置项加载、环境变量设置和*Ray设备的映射关系传入（VLLM已内置该功能直接设置VLLM_RAY_BUNDLE_INDICES环境变量即可）*；随后使用 VLLM 的 `run_server` 组件替代原 lmdeploy server，完成推理服务的启动。

2. **请求格式适配与参数转换**：
   a. 推理输入统一适配为 VLLM 的 `/v1/chat/completions` 接口规范，将 prompt 文本与图片按照 OpenAI 标准格式组织后传入；同时将后处理参数从原 lmdeploy 变量名转换为 VLLM 所需的变量名。
   b. 针对 VLLM 接口的返回格式，调整并实现对应的 Response 提取逻辑。
3. **VLLM token进token出**：
   a. token进实现：



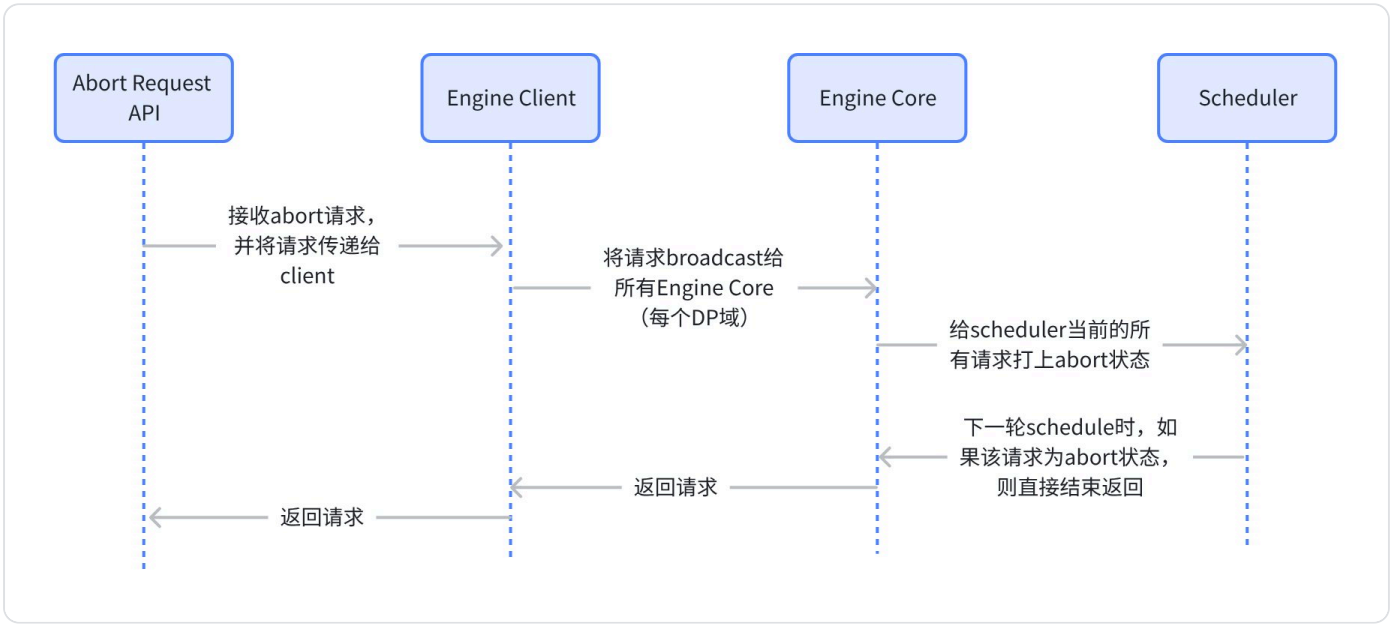
原始request传入input_ids，接着input_ids会透传到engine_prompt，engine_prompt后会将图片占位符替换为图片实际token长度的占位符，由于我们传入的input_ids已经包含了图片token长度的占位符，因此跳过多模态的preprocess，直接透传。

b. token出实现：

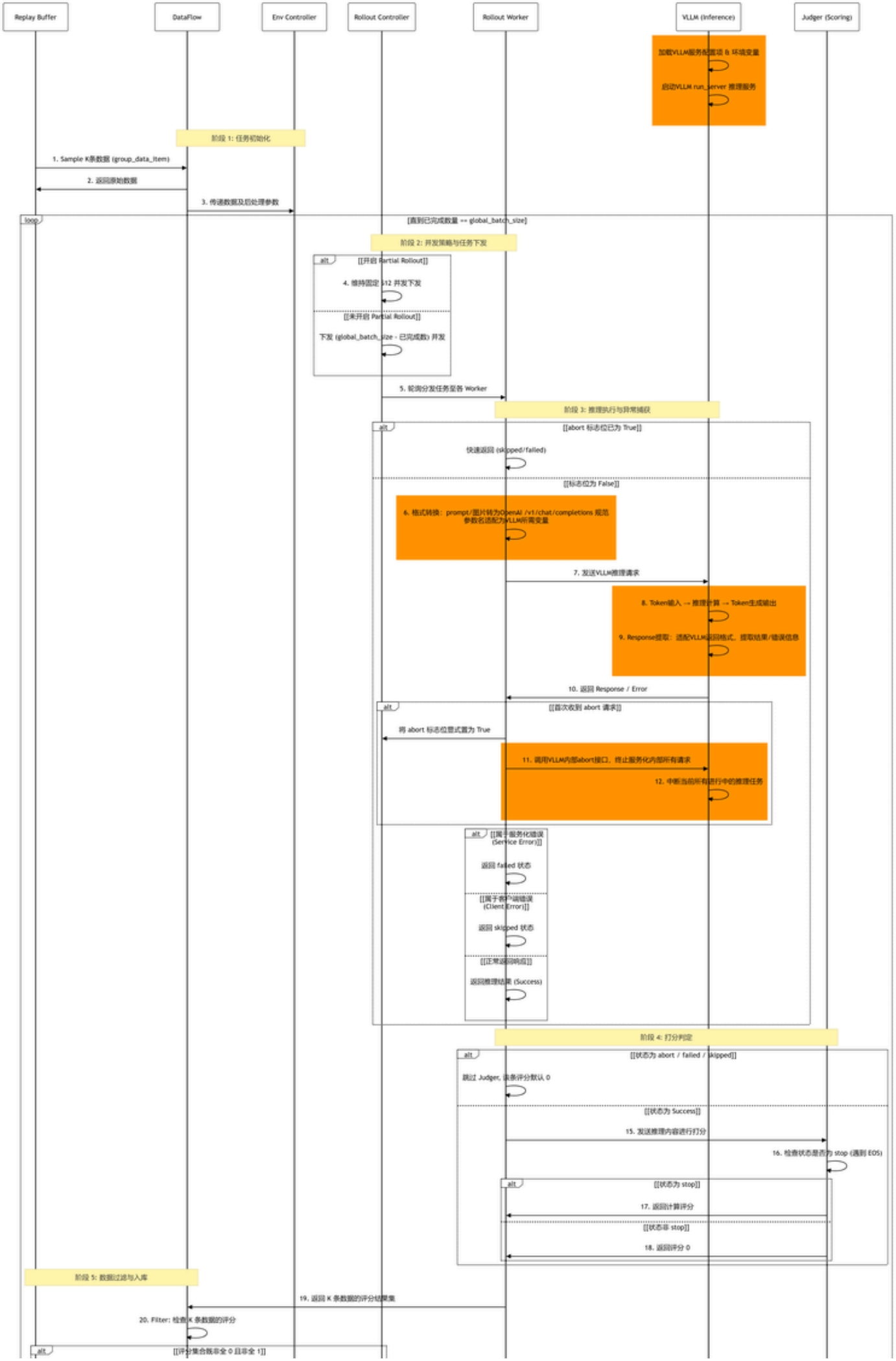
使用openai的/v1/chat/completions接口，直接传入return_token_ids即可返回token_id

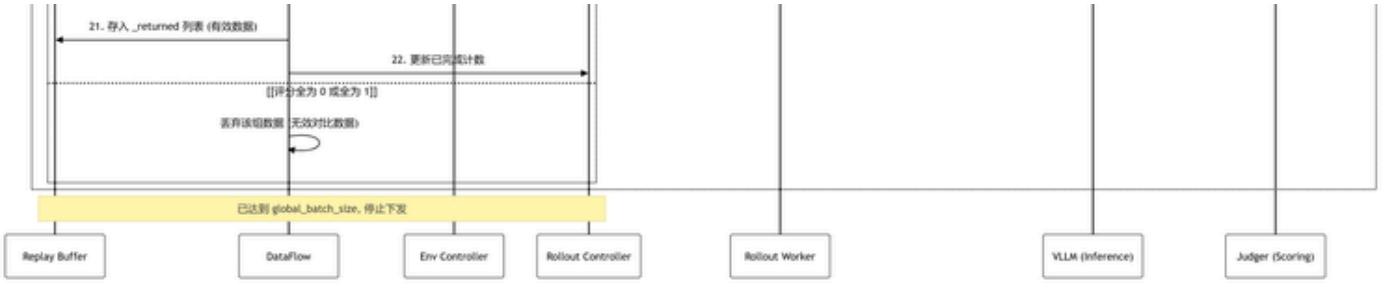
4. **Abort服务化内所有请求接口：**

给vllm服务化添加abort请求的API，设计流程图如下



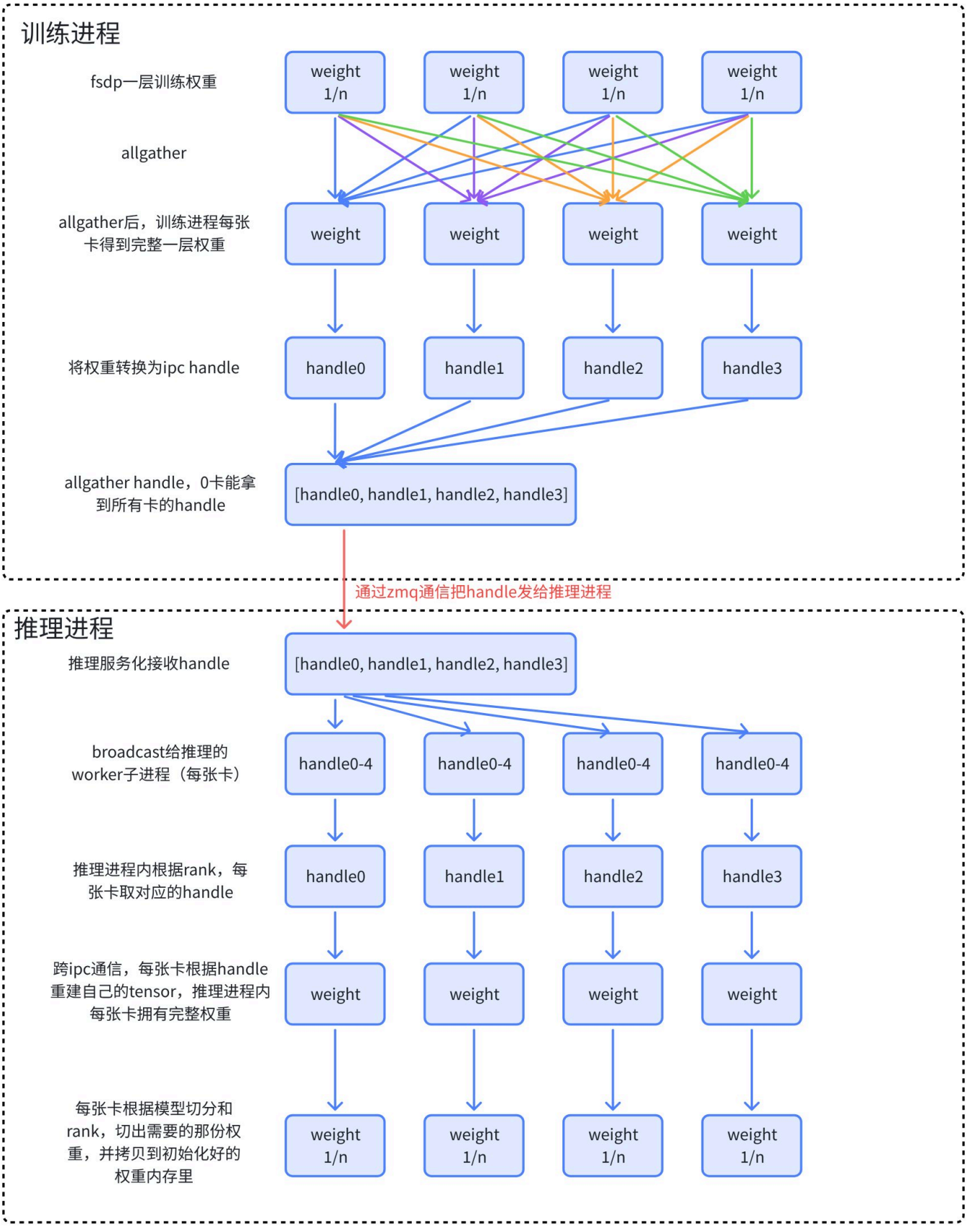
以上所有功能修改点在rollout流程图中高亮，如下：


Replay Buffer
DataFlow
Env Controller
Rollout Controller
Rollout Worker
VLLM (Inference)
Judger (Scoring)
加载VLLM服务配置项 & 环境变量
启动VLLM run_server 推理服务
阶段 1: 任务初始化
1. Sample K条数据 (group_data_item)
2. 返回原始数据
3. 传递数据及后处理参数
loop
[直到已完成数量 == global_batch_size]
阶段 2: 并发策略与任务下发
alt
[[开启 Partial Rollout]]
4. 维持固定 512 并发下发
[[未开启 Partial Rollout]]
下发 (global_batch_size - 已完成数) 并发
5. 轮询分发任务至各 Worker
阶段 3: 推理执行与异常捕获
alt
[[abort 标志位已为 True]]
快速返回 (skipped/failed)
[[标志位为 False]]
6. 格式转换：prompt/图片转为OpenAI /v1/chat/completions 规范
参数名适配为VLLM所需变量
7. 发送VLLM推理请求
8. Token输入 → 推理计算 → Token生成输出
9. Response提取：适配VLLM返回格式，提取结果/错误信息
10. 返回 Response / Error
alt
[[首次收到 abort 请求]]
将 abort 标志位置式置为 True
11. 调用VLLM内部abort接口，终止服务化内部所有请求
12. 中断当前所有进行中的推理任务
alt
[[属于服务化错误 (Service Error)]]
返回 failed 状态
[[属于客户端错误 (Client Error)]]
返回 skipped 状态
[[正常返回响应]]
返回推理结果 (Success)
阶段 4: 打分判定
alt
[[状态为 abort / failed / skipped]]
跳过 Judger，该条评分默认 0
[[状态为 Success]]
15. 发送推理内容进行打分
16. 检查状态是否为 stop (遇到 EOS)
alt
[[状态为 stop]]
17. 返回计算评分
[[状态非 stop]]
18. 返回评分 0
阶段 5: 数据过滤与入库
19. 返回 K 条数据的评分结果集
20. Filter: 检查K 条数据的评分
alt
[[评分集合既非全 0 且非全 1]]



5. 权重更新

权重更新流程如下，训练将每层权重allgather后，把权重转化成ipc_handle，再将每张卡的ipc handle传递给推理进程，推理进程根据rank再重建每张卡权重，这样就省去了传完整tensor的通信时间



vllm针对服务化的worker（每张卡上的进程），可以通过worker_extension_cls，给worker添加自定义的函数，因此我们在worker上添加了update_weight_npu_ipc的函数，用于接收ipc handle，重建tensor，并通过模型的load_weights接口加载权重, 代码如下。

```python
def update_weight_npu_ipc(self, data):
    import base64
    from multiprocessing.reduction import ForkingPickler
    import torch
    def _construct(item):
        func, args = item
        args = list(args)
        args[6] = torch.npu.current_device()
        return func(*args)

    serialized_data = data["serialized_named_tensors"]
    if isinstance(serialized_data, list):
        serialized_data = serialized_data[self.local_rank]
    weights = ForkingPickler.loads(base64.b64decode(serialized_data))
    weights = [(k, _construct(v)) for k, v in weights]
    torch.npu.synchronize()
    self.model_runner.model.load_weights(weights=weights)
    del weights
    torch.npu.synchronize()
```

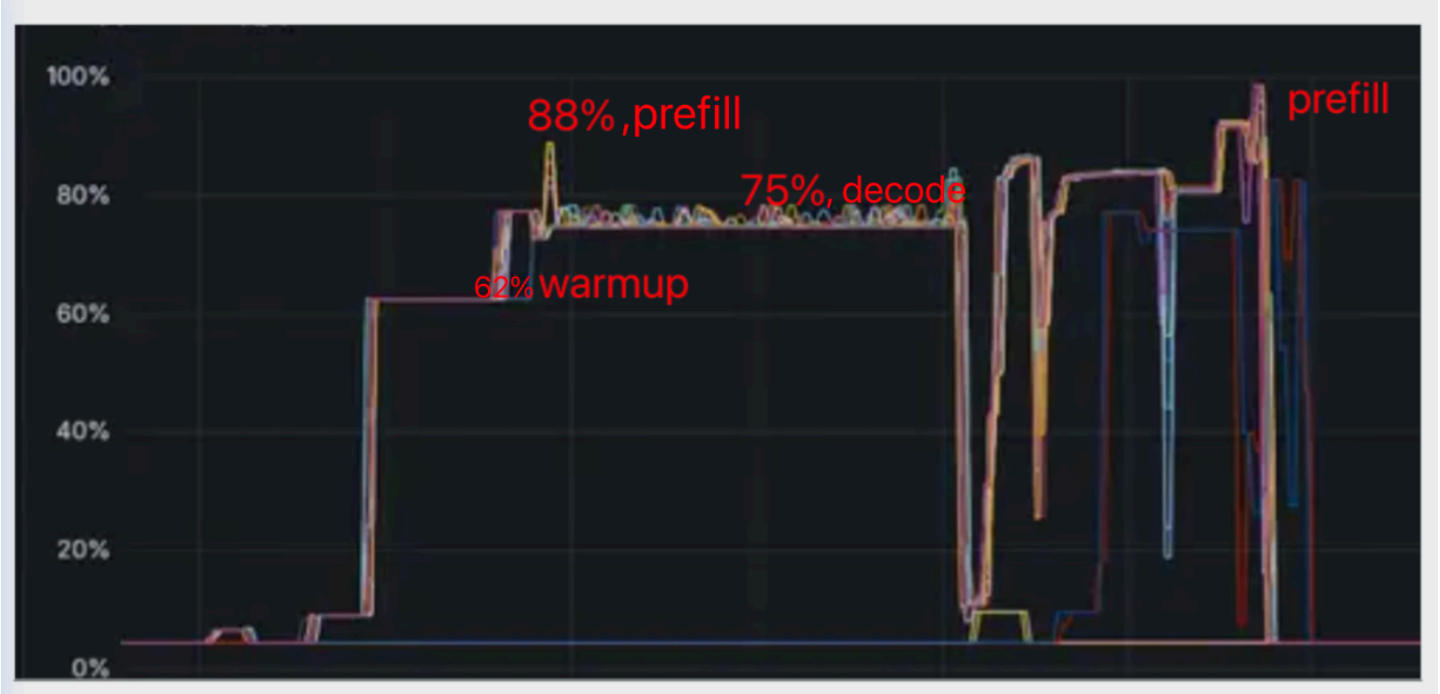
三、 内存拓扑拆解与性能分析

推理显存模型

#warmup显存拆解写明白，后续VIT的prefill尖刺也拆明白。

#total*gpu_utilization(0.8) - activation - weights = kv_cache, 怎么推出0.8，怎么回退

在基于 vLLM 等现代推理引擎的运行周期中，显存（VRAM）的变化通常经历三个核心阶段：预热（Warmup）、预填充（Prefill）和解码（Decode）。各阶段的显存峰值（M_{peak}）可由以下专业公式表达：



1. 预热阶段 (Warmup Phase)

引擎在启动时会模拟最大负载以探测峰值显存，以此作为后续内存分配的基准。

$$M_{warmup}^{peak} = M_{weights} + Act_{simulated}(N_{batched\_tokens}^{max})$$

- $M_{weights}$: 模型静态权重占据的显存。
- $Act_{simulated}$: 模拟运行时的临时激活值显存，由参数 `max_num_batched_tokens` ($N_{batched\_tokens}^{max}$) 强控制。

KV Cache 容量分配法则

在预热完成探测后，系统会将剩余可用显存一次性分配给 KV Cache 池：

$$M_{kv\_capacity} = (U_{gpu} \times M_{total}) - M_{warmup}^{peak}$$

- U_{gpu} : GPU 显存利用率上限参数（ `gpu_memory_utilization` ，如 0.90）。
- M_{total} : 物理 GPU 的总显存（ 64GB）。
- **Warmup 的峰值直接决定了可用 KV Cache 的天花板。**

---

2. 预填充阶段 (Prefill Phase)

处理输入 Prompt 并生成初态 KV Cache 的计算图阶段。考虑到您提到的包含强化学习（RL）残留环境的特殊场景：

基础 Prefill（第一步） 显存峰值：

$$M_{prefill}^{base} = M_{weights} + M_{kv\_used} + Act_{prefill}(N_{batched\_tokens}^{max})$$

由于RL涉及到推理和训练的切换，推理和训练的引擎往往不同（比如推理上vllm，sglang，训练是fsdp或者megatron），这导致切换后碎片或者未完全卸载的显存，会出现第二步之后的RL，显存占用比第一步更高。

带 RL 残留的全局 Prefill （第二步）显存峰值：

$$M_{prefill}^{total} = M_{prefill}^{base} + M_{RL\_residual}$$

- M_{kv_used} : 当前批次已生成的实时 KV Cache 大小。
- $Act_{prefill}$: Prefill 阶段运算产生的真实激活值，同样受限于 `max_num_batched_tokens` 。
- $M_{RL\_residual}$: 强化学习训练或特定生成框架中上一轮次/环境未释放的残留空间（RL Residual Space）。

---

3. 解码阶段 (Decode Phase)

自回归逐字生成阶段。此阶段的计算特征表现为 Memory-bound（访存密集）。

$$M_{decode}^{peak} = M_{weights} + M_{kv\_used} + Act_{decode}(N_{seqs}^{max})$$

- M_{kv_used} : 随生成长度不断累加的 KV Cache 大小（趋近于分配上限）。
- Act_{decode} : Decode 阶段产生的极小激活值，由系统允许的最大并发请求/序列数 `max_num_seqs` (N_{seqs}^{max}) 控制。

RL显存峰值计算

RL的E2E的显存峰值，往往就在第二步的推理prefill，通过上面的公式带入计算：

$$M_{prefill}^{total} = M_{weights} + (U_{gpu} \times M_{total}) - (M_{weights} + Act_{simulated}(N_{batched\_tokens}^{max})) + Act_{prefill}(N_{batched\_tokens}^{max}) + M_{RL\_residual} = U_{gpu} \times M_{total} - Act_{simulated}(N_{batched\_tokens}^{max}) + Act_{prefill}(N_{batched\_tokens}^{max}) + M_{RL\_residual}$$

可见只与 `gpu_memory_utilization` 和 `max_num_batched_tokens` 有关，后续显存优化会参考该公式。

四、 核心技术攻关

精度对齐攻关

Rollout 阶段的精度诊断

背景：

H200的精度基线只有512die训练的rewards、entropy和训推一致性的kl散度等指标，由于rollout本身具有很强的随机性，这几个指标对精度的对齐不太敏感，在没有512die GPU时无法确认rollout部分的精度是否正确，因此需要设计一套维测指标对精度进行细致分析。

维测指标设计

#原始客户的kl、entropy，A3和GPU对不上，然后再深入发现prob不对齐；每个指标的值域范围讨论

3.1 Entropy

公式：

$$H = -\sum_i p_i \log p_i$$

指标结果：lmdeploy中通过对句子级平均entropy的打开，我们发现了部分句子有极低的entropy，再通过打开token的prob，我们发现有的句子能连续几千个token采样极低概率token，这证明lmdeploy在RL中石油精度问题的。

句子级entropy指标的表格或图【待补充】

连续低概率的图【待补充】

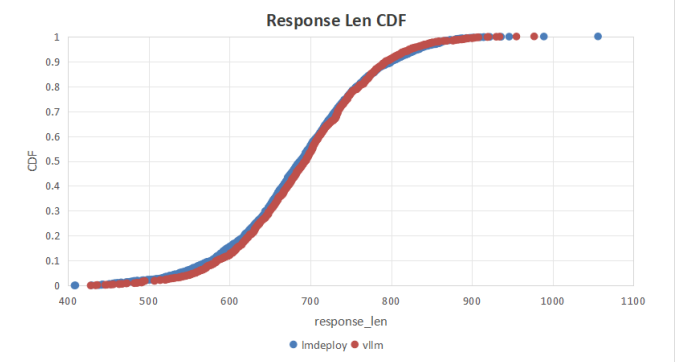
临时规避：topk=20 + 过滤 avg prob < 0.1

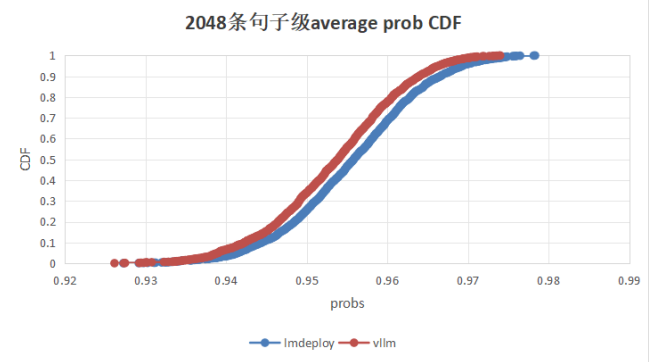
总结：由此可见该精度指标可以指导我们发现可能的后处理采样问题或其它精度问题。

3.2 句子级Prob CDF

指标解释：通过保存一条句子所有被选中token的prob，求平均值可以得到这条句子的平均值prob。

指标结果：为了确保vllm和lmdeploy换框架后精度是对齐的，我们在RL全流程中，使用固定的1条句子rollout 2048遍，看这条数据的response_len的分布和probs的分布是否一致，我们发现，response_len基本重合，但是lmdeploy的平均probs会比vllm高些，因此我们需要再打开模型去看两边的差一点究竟由什么导致。





进一步验证：通过输入单条请求对模型的打点我们发现，在开启完全确定性计算的情况下，ViT 的 输出tensor的l1_norm偏差两个框架在0.001%左右， 但是到了 首 Token Logits后该误差被放大到了放大至 0.3%；继续定位发现 ViT 中 RMSNorm、Projector 的切分细节未对齐，lmdeploy都进行了TP切分，而VLLM中没有，在对齐后，logits的误差变为0。

总结：由此可见，该精度维测指标可以用来判断两个框架的精度对齐情况，同时指导我们从正向去排查导致精度的差异的模型配置或代码，而非一开始就进入细节从而节省时间。

内存极致优化

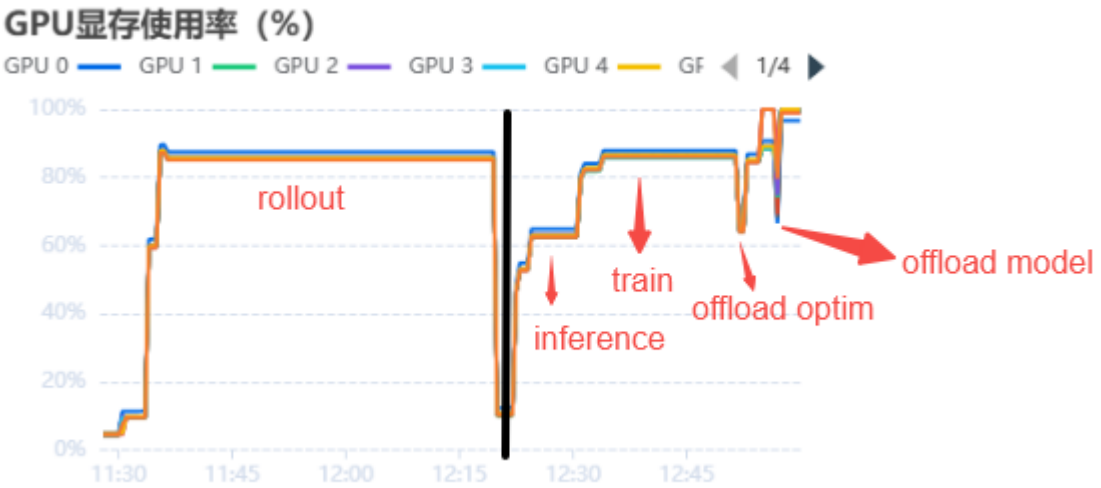
NPU单die显存64G，FSDP框架下不开启序列并行SP，跑36k序列会因激活值太大而OOM。xtuner框架官方没有提供rl SP功能适配。

以客户配置（2048条数据，512die，NPU）为例，不开启SP，测试不同序列长度的多轮grpo训练功能，结果如下：

（1）5k序列（4k prompt + 1k response），5-10轮训练中不定时出现OOM导致中断

（2）36k序列（4k prompt + 32k reponse），第一轮训练的rollout完成后立刻OOM报错并中断

首先分析5k短序列中的显存问题，采集某次实验中OOM报错的显存占用动态变化曲线如下。可以简单将一轮序列分为rollout、inference、train和offload四个阶段，5k序列的OOM总是发生在offload阶段。



我们参考xtuner的rl代码实现，对offload阶段的显存行为具体分析。训练结束后offload_optimizer释放显存占用（从80%到60%+），随后rollout推理onload_weights（显存占用从60%+到90%），updata_weight促使显存快速增加，随后offload_model基本没对显存占用产生效果（微小下降），最后onload_kvcache直接导致显存触顶并OOM报错中断训练。

```
# 5. Saving and sync weights
with timer("saving and sync_weight", step_timer_dict):
    ray.get(self._train_controller.offload.remote(target="optimizer",print_memory=True))
    self._maybe_save_hf()
    bind_train_rollout(
        train_controller=self._train_controller, env_controller=self._rollout_env_controll
    )
    ray.get(self._rollout_env_controller.onload_weights.remote())
    ray.get(self._train_controller.update_weights.remote())
    self.logger.info("Model weights synchronized successfully.")
    ray.get(self._train_controller.offload.remote(target="model",print_memory=True))
    ray.get(self._rollout_env_controller.onload_kvcache.remote())
```

基于以上分析，5k序列长度训练的显存缺口其实并不大，我们快速迁移以往sft训练中曾用的显存优化方案，即解决了短序列场景的OOM问题：

（1）逐层卸载前向激活值（activation offload）

（2）将优化器状态移至CPU侧，仅在optimizer.step时短暂swap至NPU侧（swap optimizer）

（3）开启虚拟内存

不幸的是，36k序列第一轮rl训练都无法完成，在rollout之后的241B模型（6B ViT + 235B LLM）前向即OOM，可见长序列下的激活值增加带来了沉重的显存负担。依据以往的sft训练调优经验，235B的纯LLM MoE模型在32k序列长度下，显存就已达到极限；而添加了额外6B的ViT后，32k序列长度下必须至少开启SP2才能使能NPU训练。

序列并行

4. 训练阶段的SP逻辑

在训练前向中，SP切分序列后到输出logits之前其实都不做all_gather通信；

在计算loss过程中，其实是以token维度进行（即，每个token算出对应的softmax loss就可以），最后进行all_reduce在所有卡间（对各token loss）求和

但是存在两个问题（1）某些token是被padding的，其loss需要被屏蔽；（2）需要计算一个权重 $\frac{1}{\Sigma_i^G|o_i|}$

这两个问题都可以归结为为每个token的loss分配对应的权重（即policy_weight），并且（1）和（2）都需要样本维度的token信息：

（1）单条样本中哪些token的loss需要保留（**sp_group的all_gather**）；

（2）单条样本中**未被padding的token**个数（**sp_group的all_gather**）$|O_i|$ --->进而计算单个mini_batch中所有样本**未被padding的token**个数总和（**所有卡的all_reduce**）$\Sigma_i^G|o_i|$

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\left[\frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{[\pi_\theta(o_{i,t} \mid q, o_{i,<t})]_{\text{no grad}}}\hat{A}_{i,t} - \beta\mathbb{D}_{\text{KL}}\left[\pi_\theta \| \pi_{\text{ref}}\right]\right]$$

5. RL Training实现GAP

一旦开启SP，第一步就会切分seq_ctx和loss_ctx_input了

- 第2、4和5步都需要完整的一条样本，其计算涉及到loss_ctx_input的三个属性，因此需要添加**loss_ctx_input.sp_gather**
- 第3和7步，涉及到模型的前反向计算，因36k序列长度会OOM、必须开启SP，所以
  - 2-3步之间涉及**loss_ctx_input.sp_split**
  - 5-7步之间涉及**loss_ctx_input.sp_split**

6. Loss计算实现GAP

原始的RLLoss计算实现代码不考虑SP的情况，只有

（1）一个all_reduce计算所有卡间**未被padding的token个数**之和

这段逻辑对应GRPOLossContex内部重写的build_batches_loss_kwargs方法

```python
class GRPOLossContext(BaseLossContext[RLLossContextInputItem]):
    def build_batches_loss_kwargs(
        cls,
        data_batches: list[RLLossContextInputItem],
        loss_cfg: GRPOLossConfig,
        cu_seq_lens_list: list[torch.Tensor] | None = None,
        sp_mesh: DeviceMesh | None = None,
    ) -> list[GRPOLossKwargs]:
        shifted_labels_list = [item.shifted_labels for item in data_batches]

        # Compute the denominator used in the global calibration of the loss
        rank_grad_tokens = sum((labels != loss_cfg.ignore_idx).sum() for labels in shifted_labels_list)
        rank_grad_tokens = cast(torch.Tensor, rank_grad_tokens)
        global_grad_tokens = rank_grad_tokens
        if dist.is_initialized():
            dist.all_reduce(global_grad_tokens, op=dist.ReduceOp.SUM)
```

开启SP之后，需要先对SP组做all_gather拼接所有的shifted_labels；否则，直接进行全部卡all_reduce会出现超时报错退出的未预期行为

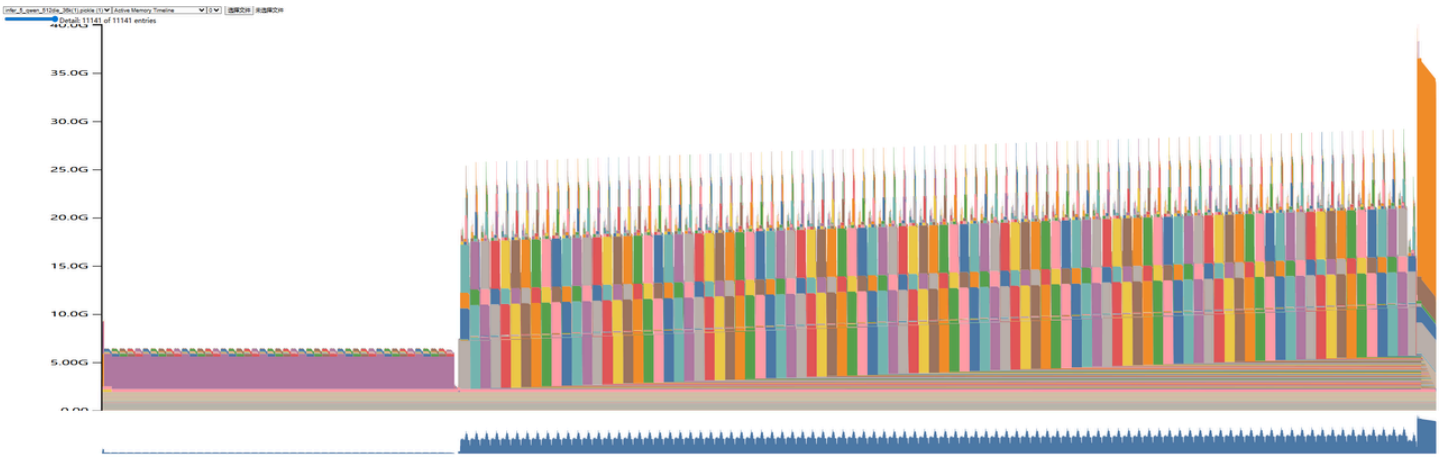
（2）一个all_reduce计算各token的loss之和

这个操作没毛病，该部分参考 xtuner的chunk_loss实现分析 中对于xtuner的loss计算类BaseLossContext的forward函数中的all_reduce，GRPOLossContext直接继承了forward方法，没有改写

Chunk loss

在当前rl训练配置下，虽然6B ViT的权重更新被冻结，但负担36k序列长度，故在inference阶段输出模型前向logits后、计算loss（actor_logprob）时出现了显存缺口。这一计算涉及hidden_size到vocab_size维度的映射，词表维度vocab_size通常是个非常大的值，因此这一阶段通常也是LLM 预训练和sft训练过程中的显存瓶颈。

缩减序列长度，采集inference阶段的内存快照，计算loss处有明显尖刺。



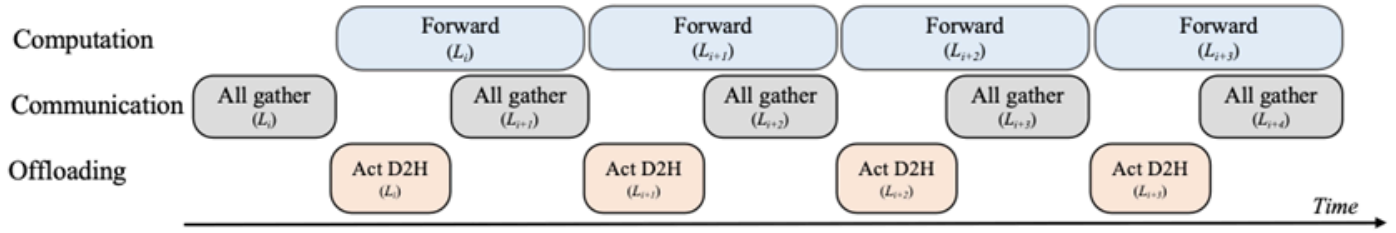
xtuner框架本身为训练适配了chunk_loss特性，将logits按照seq_len维度进行切分，循环多次计算loss，避免shape为(batch_size, seq_len, vocab_size)的超大内存占用。当前模型配置下，这个shape是(1,36*1024,)，bf16格式下显存占用为

Asynchronize Activation Offload (SwapActivation)

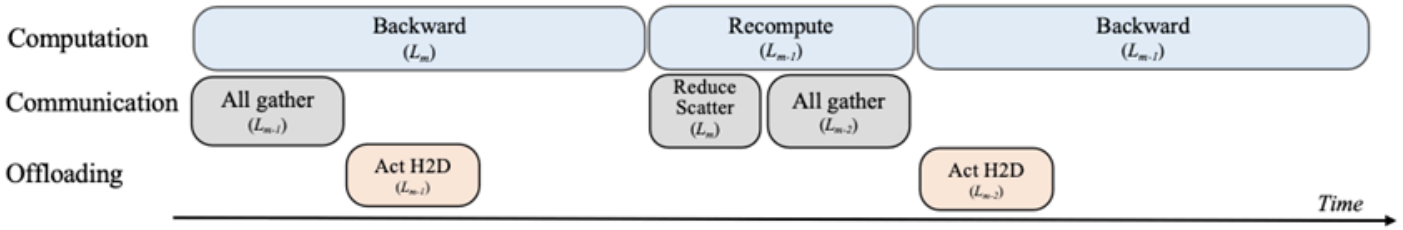
显存资源紧张是制约大模型训练规模与效率的关键瓶颈，尤其激活值堆积、优化器状态占用、大词表损失计算引发的显存峰值，易导致训练中断。**SwapActivation（激活值卸载与预取机制）**：在强化学习等显存极其受限的场景下，该特性通过精细的多流并行与软硬件协同调度，极大地缓解了显存压力并保障了训练吞吐。其核心逻辑在于**利用计算与通信的间隙，完全隐藏GPU与Host主机内存之间的数据迁移（PCIe）开销**。

- **在前向传播（Forward）阶段**：如图(b)所示，当第 LiL_iLi 层的计算完成后，系统会立即启动异步的 Device-to-Host (D2H) 传输，将高显存占用的激活值卸载到CPU内存。这一卸载过程被巧妙地与第 Li+1L_{i+1}Li+1 层的前向计算以及 All-gather 通信完全重叠（Overlap），实现了“边算边卸载”。
- **在反向传播（Backward）阶段**：如图(c)所示，系统采用前瞻性预取策略，通过异步的 Host-to-Device (H2D) 传输，将下一层重计算（Recompute）所需的激活值提前从CPU拉回GPU。这一加载过程与当前层（如 LmL_mLm）的反向计算无缝并行。

通过这种“前向异步卸载、反向提前预取”的设计，SwapActivation不仅打破了物理显存对RL大模型训练规模的限制，还确保了NPU计算流的连续性，使得模型能够在不牺牲计算效率的前提下，以更大的Batch Size或序列长度进行高效训练。



(b) Activation Checkpointing Swap Overlap Forward



(c) Activation Checkpointing Swap Overlap Backward

ViT TP

在 vLLM（以及许多其他推理框架如 DeepSpeed, HuggingFace Accelerate）中，**ViT（Vision Transformer）部分默认不进行 Tensor Parallelism (TP) 切分**，而只对 LLM 部分进行 TP 切分。但是在RL场景下，我们打破了这一惯例，选择**对 6B ViT 同样启用张量并行切分**。这一架构设计的动机与收益分析如下：

对 ViT 引入 TP 的主要顾虑在于增加网络通信（如 All-Reduce）开销，从而拖慢模型前向传播的速度。但通过对 VLM 推理全生命周期的计算特征剖析，我们发现这是一个具有高度非对称性的权衡（Trade-off）：

- **通信开销在 Prefill 阶段被摊销：** 视觉编码器（ViT）仅在输入预处理（Prefill）阶段执行一次。在 6B 参数规模下，算子内部的计算密度已具有一定规模，切分后引入的 All-Reduce 延迟在整个单次图文预处理生命周期中的绝对时间占比极小（通常在几十毫秒级别）。
- **规避 Decode 阶段的重计算灾难：** 在 RL 的经验采样（Rollout）环节，由于需要超大的全局 Batch Size 以保证强化学习梯度的无偏性和稳定性，系统对 KV Cache 的容量需求呈指数级上升。若不切分 ViT，单卡被挤占的 12GB 显存会导致 KV Cache 槽位严重不足；一旦触发显存峰值，系统将被迫驱逐（Evict）部分 KV Cache 并引发重计算（Recomputation）。

对于高达 235B 参数规模的 LLM 而言，由于缺失 KV Cache 导致的 Autoregressive Decode 阶段重计算开销是极其高昂的（秒级甚至更高），其代价成百上千倍地远超 ViT 切分带来的几十毫秒通信延迟。

基于上述推演，我们实现了对 6B ViT 的张量并行切分。通过消除 ViT 的数据冗余，将单卡释放出的约 10GB 静态显存还回KV Cache，将 235B 大语言模型的重计算触发率进一步降低。

DP4TP4

由于Intern S1模型是GQA，kvhead只有4，TP16切分会导致kv head出现冗余导致KV Cache的浪费。同时我们观察到rollout后半段会出现kv cache不足，开始重计算的情况，重计算会影响rollout的推理性能。因此为了提升KV Cache容量，减少重计算，需要rollout推理服务走DP4TP4切分

1. Ray设备资源映射

   vllm可以通过传入*VLLM_RAY_BUNDLE_INDICES*环境变量，告诉vllm的worker用ray设备资源中的哪一个npu，但在DP4下，每个Engine Core下的worker仅能看到自己DP域的4张卡，因此需要在分配ray资源时，按顺序将*VLLM_RAY_BUNDLE_INDICES*均分给每个dp域，例如dp0，取bundle_indeces的0-3，dp1取4-7，依次类推

2. 权重更新

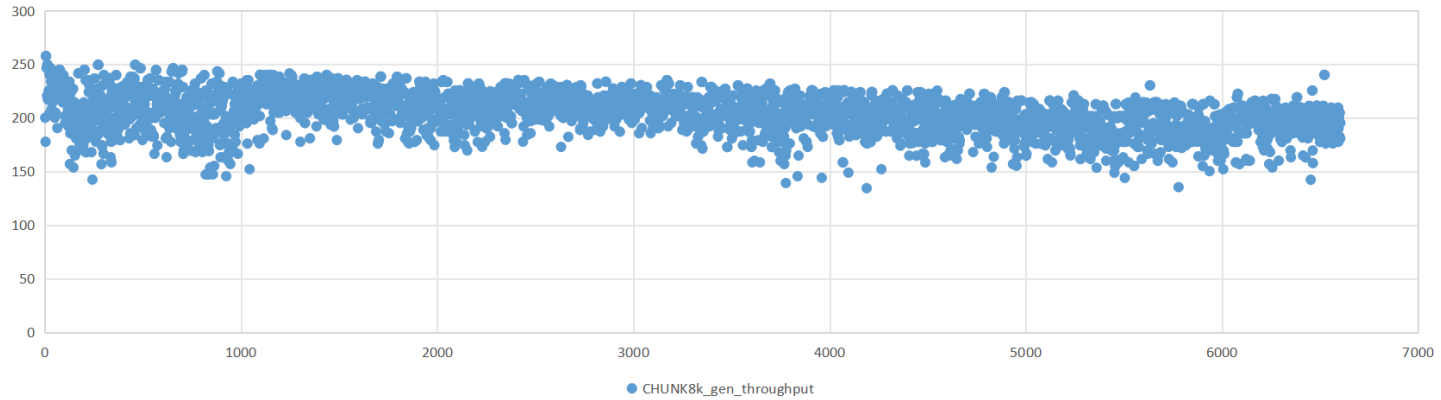
和之前提到的一样，每个worker仅能看见自己DP域的4张卡，而训练进程传递ipc handle时，传递了A3的16张卡的handle。因此为了保证DP域下，worker取到正确的ipc handle，我们需要给worker添加全局的rank的属性，让每个worker知道自己在A3的16张卡的全局位置。

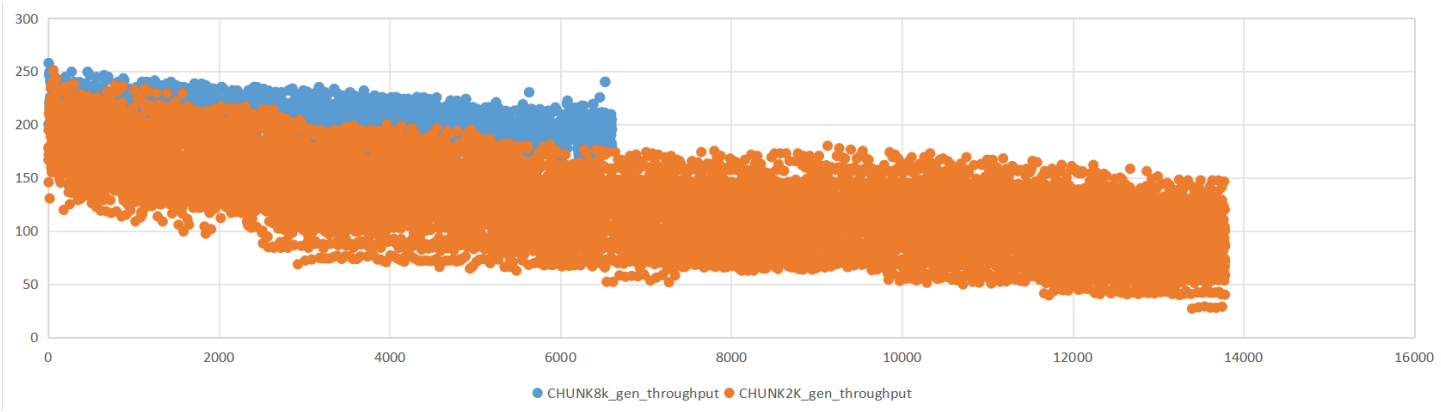
优化项1: chunk prefill显存

切换到DP4TP4后，第二步就出现了OOM问题，通过第三节显存逻辑的梳理后，检查设置发现TP16->DP4TP4切换前后，`max_num_batched_tokens` 在 TP16 时的默认值是 8K，切换成 TP4DP4 之后，我们没有变更，这导致激活值由于TP的减小而增大四倍。为此我们需要将 `max_num_batched_tokens` 减少 4倍来维持激活值的恒定, 避免第二步的OOM问题。

优化项2: PD混部性能

`max_num_batched_tokens` 由8K降低到2K，显存问题解决了，但是导致性能劣化一倍。8k时平均吞吐时200，但是2k时只有120左右。





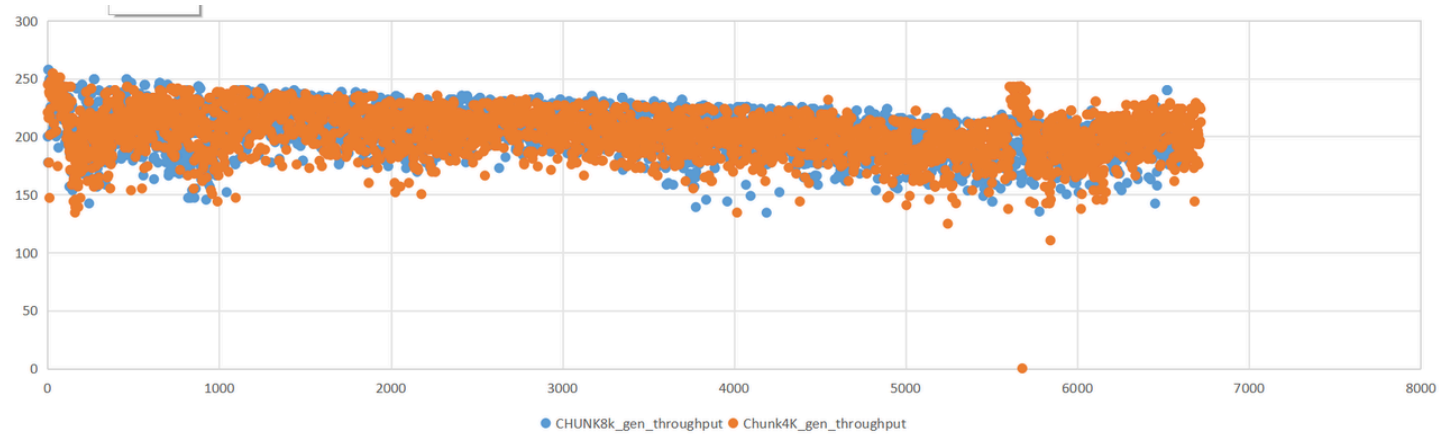
原因是PD混部：Chunk prefill产生的更多P，在 **Continuous Batching（连续批处理）下**会打乱D的批量处理导致性能劣化。

- **Prefill 是计算密集型（GEMM）：** 大矩阵乘法，疯狂榨干算力单元（CUBE）。
- **Decode 是访存密集型（GEMV）：** 矩阵向量乘法，极其依赖显存带宽（HBM），对算力要求极低。
- 当这两者被强行塞在一起时，Prefill 巨大的算力需求和长尾时间，会**严重拖慢同 Batch 里 Decode 阶段的生成速度**。

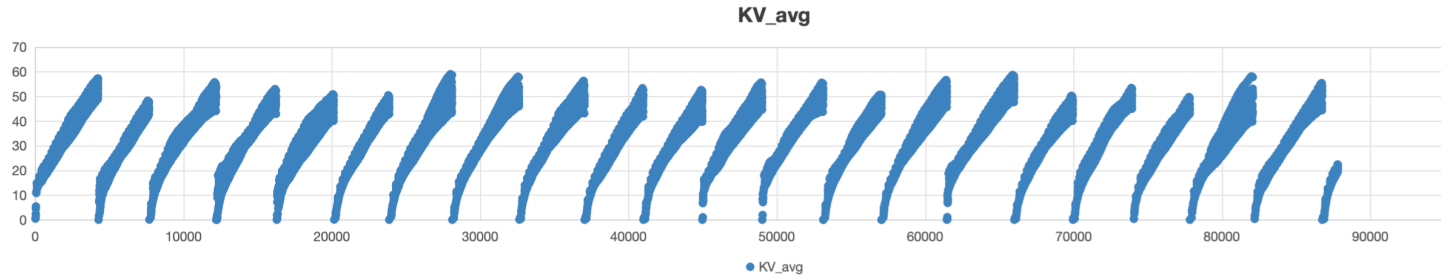
vLLM这样设计是为了最小首token延迟，但是在RL场景下，只关心推理采样总吞吐，这完全是负收益设计。

咨询了vllm_ascend专家，vllm社区是如此设计，有段时间vllm_ascend支持关闭该功能。但是后续为了社区演进，PD混部是不能避免的。

说明Chunk prefill 切的很碎，导致decode被混入。通过将 `max_num_batched_tokens` 从2K调整到4K避免性能劣化



根据前面的RL流程中推理最大的显存峰值公式，将 `max_num_batched_tokens` 从2K调整到4K会导致显存峰值上升，为此只能减小 `gpu_memory_utilization` 。值得庆幸的是，当前kv最高的利用率不过70%，因此将 `gpu_memory_utilization` 从0.83降低到0.78不会引来性能问题。



优化项3: 碎片显存清理

跑到第10步时，出现权重保存后多出15GB显存，导致的OOM问题。通过手动触发碎片回收解决。

代码块

```
import torch
import gc

gc.collect()
torch.npu.empty_cache()
```

但是TP16保存权重并没有出现该问题，推测是切分的不同带来显存分布的不同，在TP16场景下显存缓慢增加达到GC回收阈值，触发了碎片自动回收；但是DP4场景时一次申请大显存，直接OOM了。为此推荐在每个大阶段结束时主动触发碎片回收。

#虚拟显存无法开启的解决方案

显存优化方案总结

针对 6B ViT + 235B LLM 在 512 卡（单机16卡，64G NPU）上的强化学习训练，我们从**计算张量切分（SP/Chunk Loss/TP/DP）**、**软硬件存储调度（Swap/Offload）**以及**推理引擎底层机制（Prefill/Decode混部与显存碎片化管理）**三个维度进行了深度改造。不仅解决了 36k 序列的 OOM 绝对瓶颈，还通过架构微调（ViT TP与DP4TP4）与 PD(Prefill-Decode) 混部超参（`max_num_batched_tokens`、`gpu_memory_utilization`）的权衡，将单机释放出的数

十 GB 显存还反哺给 KV Cache，从而在受限硬件条件下保障了 RL 训练的稳定吞吐与高效 Rollout 性能。

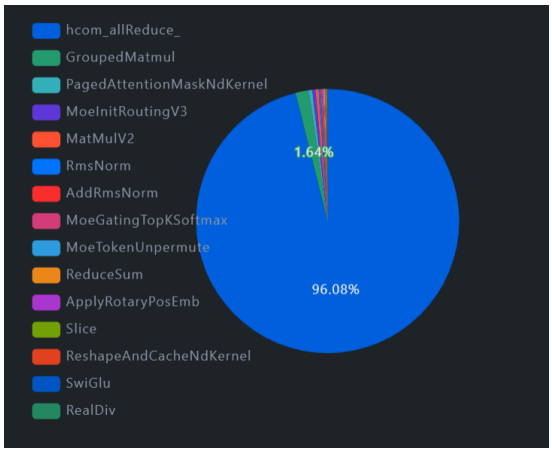
优化点 / 特性	作用模块 / 阶段	核心机制与做法	解决的痛点与优化效果收益
基础显存优化 *(SFT迁移方案)*	Train阶段	1. 逐层卸载前向激活值（Activation Offload） 2. 优化器状态移至CPU侧（Swap Optimizer） 3. 开启虚拟内存	**消除短序列OOM：** 彻底解决了 5k 短 offload 阶段触顶导致的 OOM 中断问
序列并行 (SP) 适配	Train阶段 *(前向与Loss)*	打通RL链路的SP逻辑，在前后向计算间插入 sp_split 与 sp_gather，并在计算 Token Loss 权重时按样本正确切分与归约处理。	**突破长序列显存瓶颈：** 将单条超长序卡，成功解决了 36k 序列在 241B 模立刻引发的 OOM 问题。
Chunk Loss	Train阶段 *(Actor Loss计算)*	计算 actor_logprob 时，将超大的 logits 张量按 seq_len 维度切分，通过循环多次计算 loss 替代整体计算。	**削峰填谷：** 避免了生成 (batch_size, vocab_size) 这一超大中间张量，消 算 Loss 时的显著显存尖刺。
异步激活值卸载 *(SwapActivation)*	Train阶段 *(前后向)*	**前向边算边卸载：** 算完即异步D2H迁移至CPU。 **反向提前预取：** 重计算前异步H2D拉回NPU。 通过多流并行将 PCIe 传输开销完全隐藏。	**零吞吐损耗的极致显存压缩：** 打破了限，进一步释放显存空间支持更大的 超长序列，且不牺牲计算效率。
ViT 张量并行 *(ViT TP)*	Rollout阶段 *(Prefill)*	打破常规的仅对 LLM 进行 TP 切分，对 6B ViT 也启用张量并行。用极短的 Prefill 通信开销换取显存。	**消灭重计算灾难：** 消除单卡冗余存储 **10GB** 静态显存还给 KV Cache 槽位， 235B LLM 在 Decode 阶段的重计算
DP4TP4 基础拓扑 *(替代原TP16)*	Rollout阶段 *(整体架构)*	1.**Ray资源映射：** 按序均分 BUNDLE_INDICES 给各个DP域。 2. **权重更新适配：** 为Worker注入全局Rank，确保正确获取16卡IPC handle。	**提升KV容量：** 针对 GQA 模型（KV H 除了 TP16 下严重的 KV Cache 冗余 Rollout 后半段的重计算频次。
Chunk Prefill 显存控制	Rollout阶段 *(Prefill)*	针对 TP 减小导致激活值膨胀4倍的问题，同步将 max_num_batched_tokens 从 8K 动态下调（初期调至2K）。	**规避引擎OOM：** 避免了 TP16 切换至 因单卡处理的 Token 量激增而在推理 OOM 的问题。
PD混部性能调优 *(Prefill-Decode)*	Rollout阶段 *(连续批处理)*	解决 Chunk 切分过细导致计算密集(Prefill)与访存密集(Decode)强行混部互相拖慢的问题。将 max_num_batched_tokens 回调至 4K，同时降低 gpu_memory_utilization (0.83->0.78)。	**兼顾吞吐与显存：** 在不影响实际 KV （峰值<70%）的前提下，避免了 PD Decode 节奏，**使平均吞吐从 120 恢**
碎片显存手动清理	全局切换阶段 *(Weight Save)*	DP4TP4 架构中一次性申请大显存且未达到自动 GC 回收阈值导致的内存碎片累积（第10步多出 15GB）。通过在各大阶段交替主动调用 gc.collect() 与 empty_cache() 解决。	**消除因碎片化导致的隐藏OOM：** 彻 重保存及大规模中间态流转过程中， 失败导致的未预期训练中断。

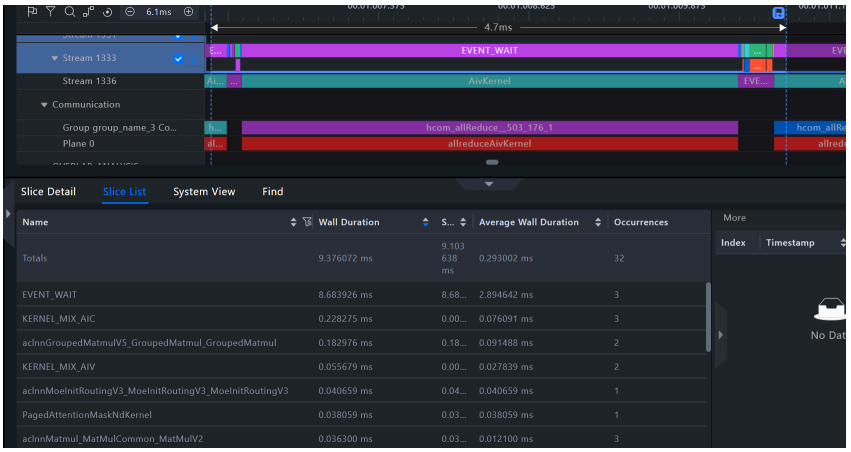
端到端性能优化

Profiling分析

#GMM在什么shape下不亲和需要转置

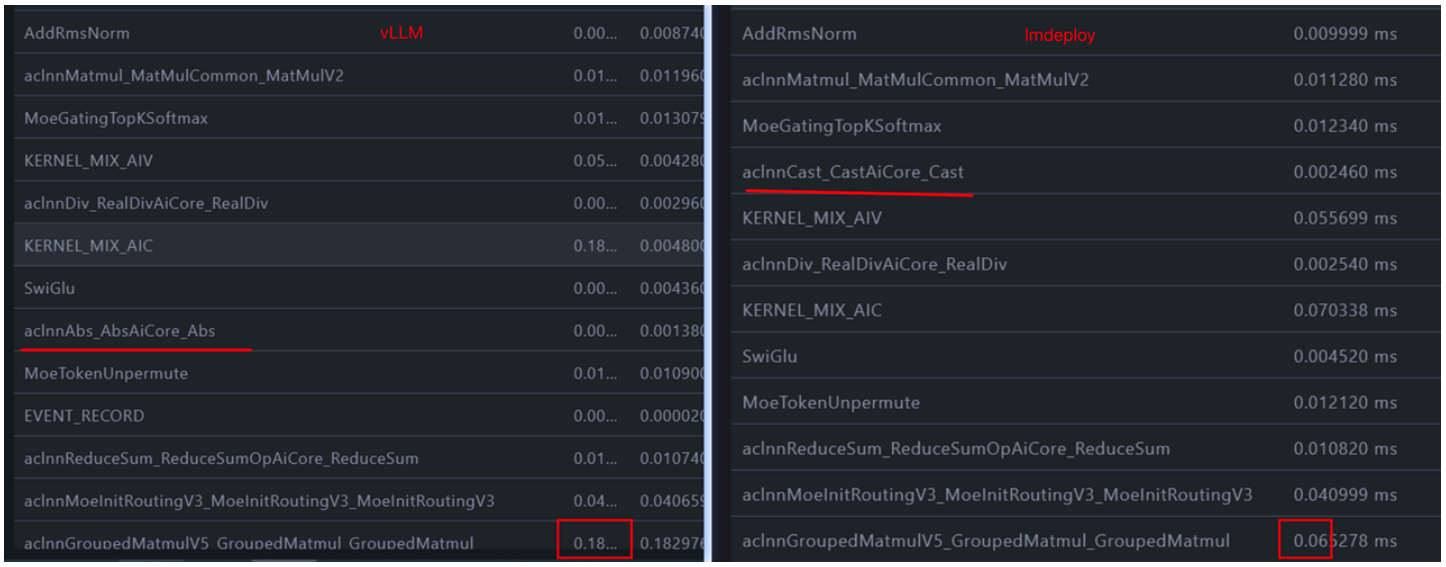
开发功能时尽管已经把lmdeploy和vllm的算子实现对齐过，但是性能vllm还是比lmdeploy差20%～30%。通过在代码的decode执行的最小核心前向加入profiling代码，并筛选BS=32的profiling结果如下：





由于客户现场版本的图模式采集没有shape、datatype等信息，先采集eager版本profiling。eager模式下大部分是TP16通信时间(快慢卡等待)，但是这些开销在图模式下时会基本消失。

重点对比计算算子：打开MindStudio，选中NPU时间轴，算子将自动按照执行顺序排序。由于之前基本对齐两边的算子实现。所以算子排列基本相同：



仔细对比后发现：

1. 区别点不少，比如多个cast算子，少一个abs算子，但是这些差异不大。
2. 其中GMM性能差异大，差了俩三倍，0.06劣化到了0.18。

GMM的差异和单次decode前向的时间差异也一致：

Vllm 计算时间 0.43ms = GMM 0.18 + MOE routing 0.04 + Pageattention 0.038 + 其他

Lmdeploy 计算 0.33ms = GMM 0.06 + MOE routing 0.04 + Pageattention 0.038 + 其他

GMM亲和计算

针对aclnnGroupedMatmulV5_GroupedMatmul_GroupedMatmul算子的差异查看

lmdeploy 算子细节如下：

Wall Duration	34us939ns
Input Shapes	"256,4096;128;192,4096;;;;;128"
Input Data Types	DT_BF16;DT_BF16;FLOAT;UINT64;FLOAT;FLOAT16;FLOAT16;INT64
Input Formats	ND;NCL;ND;ND;ND;ND;ND;ND
Output Shapes	"256,192"
Output Data Types	DT_BF16
Output Formats	ND

vllm 算子细节如下

Wall Duration	134us117ns
Input Shapes	"256,4096;128;4096,192;;;;;128"
Input Data Types	DT_BF16;DT_BF16;FLOAT;UINT64;FLOAT;FLOAT16;FLOAT16;INT64
Input Formats	ND;NCL;ND;ND;ND;ND;ND;ND
Output Shapes	"256,192"
Output Data Types	DT_BF16
Output Formats	ND

差异就在shape的第三、四维度顺序不同。

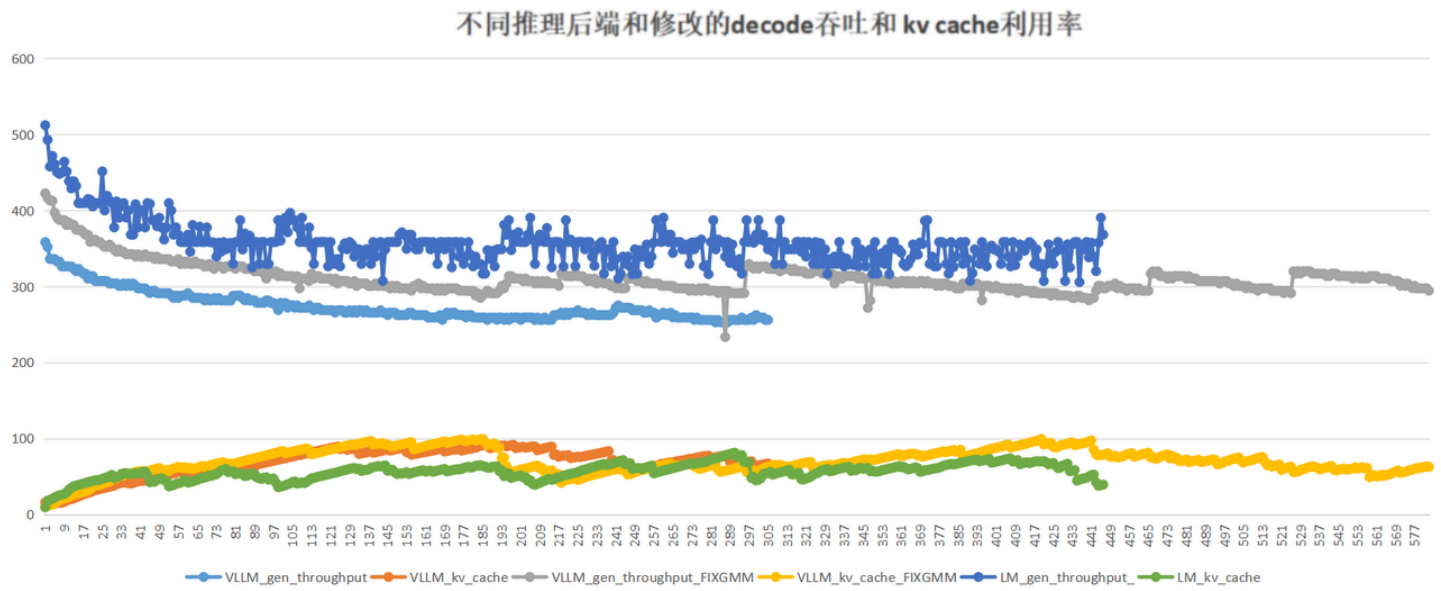
咨询了算子同事，这是典型的matmal最后一维度要是256和512的倍数才是NPU计算亲和的。

第三、四维度是输入的权重位置。为此通过在代码初始化时，将对应权重转置、前向时对GMM输入权重执行w1=w1.transpose(0,1) 即可。（transpose只会改变tensor的view，在没执行contiguous时并不会实际转置）

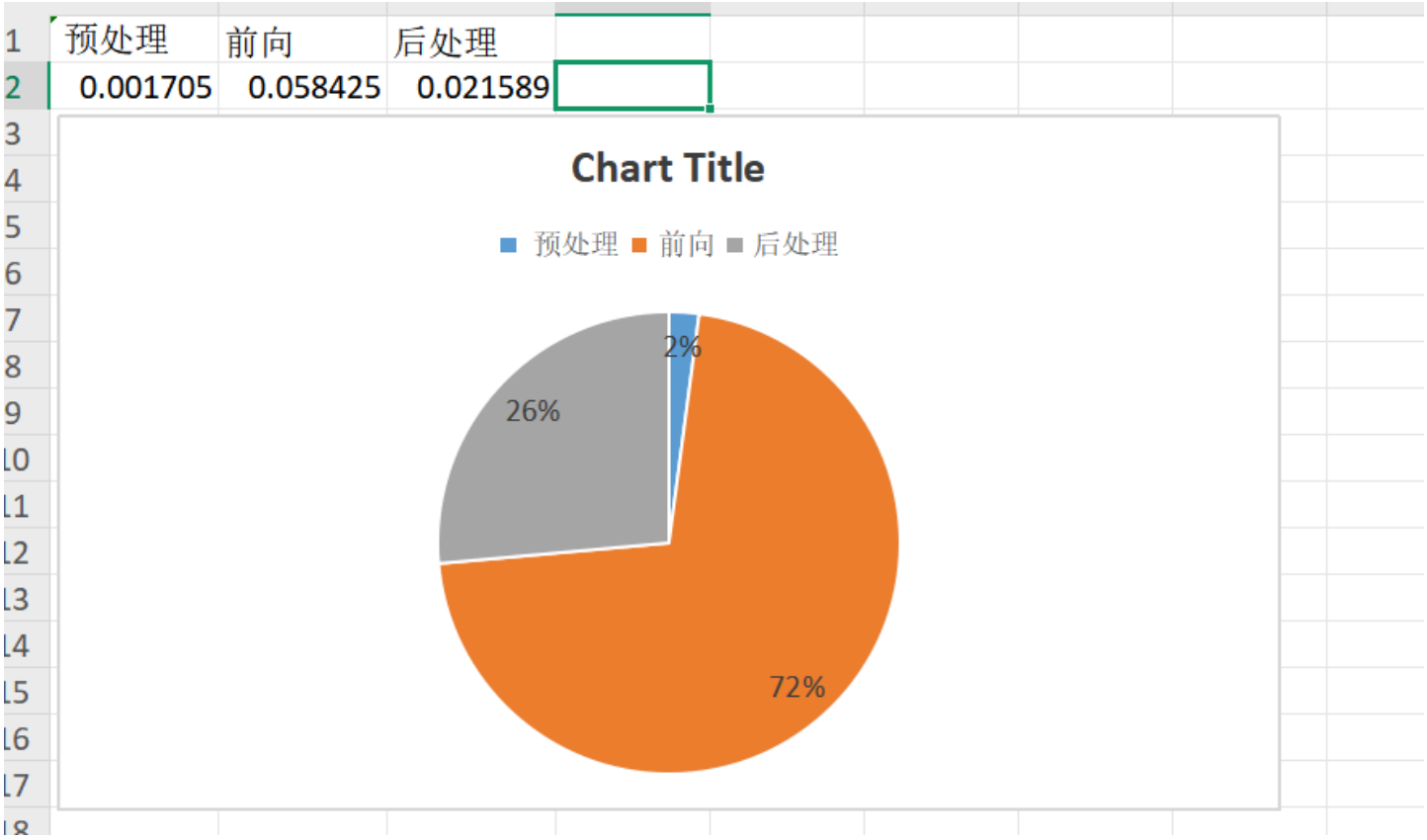
后处理优化

#后处理性能对比+标配写法

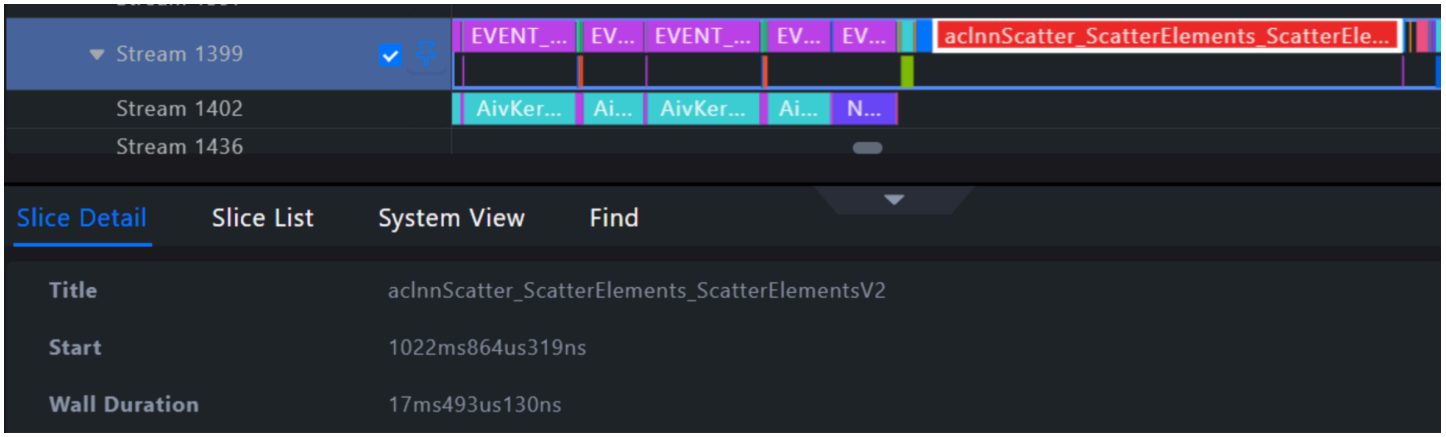
完成第一个优化点后，vllm性能提升了20%，但是和lmdeploy还是差10%左右。



由于此时算子应该完全对齐了。我们将怀疑点放到decode前向外的更大范围，前后处理部分。通过代码CPU侧打点（记得加`torch.npu.synchronize()`，不然不准）发现后处理异常长。



profiling后训练流程发现一个异常冗长的scatter算子，耗时 17ms占了单次后训练21ms的大半。



阅读对应代码

vllm > vllm > v1 > sample > ops > topk_topp_sampler.py

```
178    ) -> torch.Tensor:
183
184        The logits tensor may be updated in-place.
185        """
186        if p is None:
187            if k is None:
188                return logits
189
190            # Avoid sorting vocab for top-k only case.
191            return apply_top_k_only(logits, k)
192
193        logits_sort, logits_idx = logits.sort(dim=-1, descending=False)
194
195        if k is not None:
196            # Apply top-k.
197            top_k_mask = logits_sort.size(1) - k.to(torch.long)  # shape: B
198            # Get all the top_k values.
199            top_k_mask = logits_sort.gather(1, top_k_mask.unsqueeze(dim=1))
200            top_k_mask = logits_sort < top_k_mask
201            logits_sort.masked_fill_(top_k_mask, -float("inf"))
202
203        if p is not None:
204            # Apply top-p.
205            probs_sort = logits_sort.softmax(dim=-1)
206            probs_sum = torch.cumsum(probs_sort, dim=-1, out=probs_sort)
207            top_p_mask = probs_sum <= 1 - p.unsqueeze(dim=1)
208            # at least one
209            top_p_mask[:, -1] = False
210            logits_sort.masked_fill_(top_p_mask, -float("inf"))
211
212        # Re-sort the probabilities.
213        logits = logits_sort.scatter(dim=-1, index=logits_idx, src=logits_sort)
214        return logits
215
```

发现其实现十分的不亲和。

- Scatter 操作本质上是一个**基于索引的非连续/离散内存写入**操作。硬件需要读取 `logits_idx` 中的每一个值，计算目标地址，然后写入数据。对于 15 万级别大小的词表，这会造成海量的 **Cache Miss（缓存未命中）**，导致极低的内存带宽利用率。
- 对于类似昇腾（Ascend，从 `aclnn` 前缀可看出）或 GPU 这样的 AI 芯片，它们极其擅长大规模连续内存计算，但对这种大规模离散内存寻址操作非常不友好，导致这一个算子就吃掉了约 17.5ms。

查看了vllm_ascend的代码，发现其有优化代码实现如下：

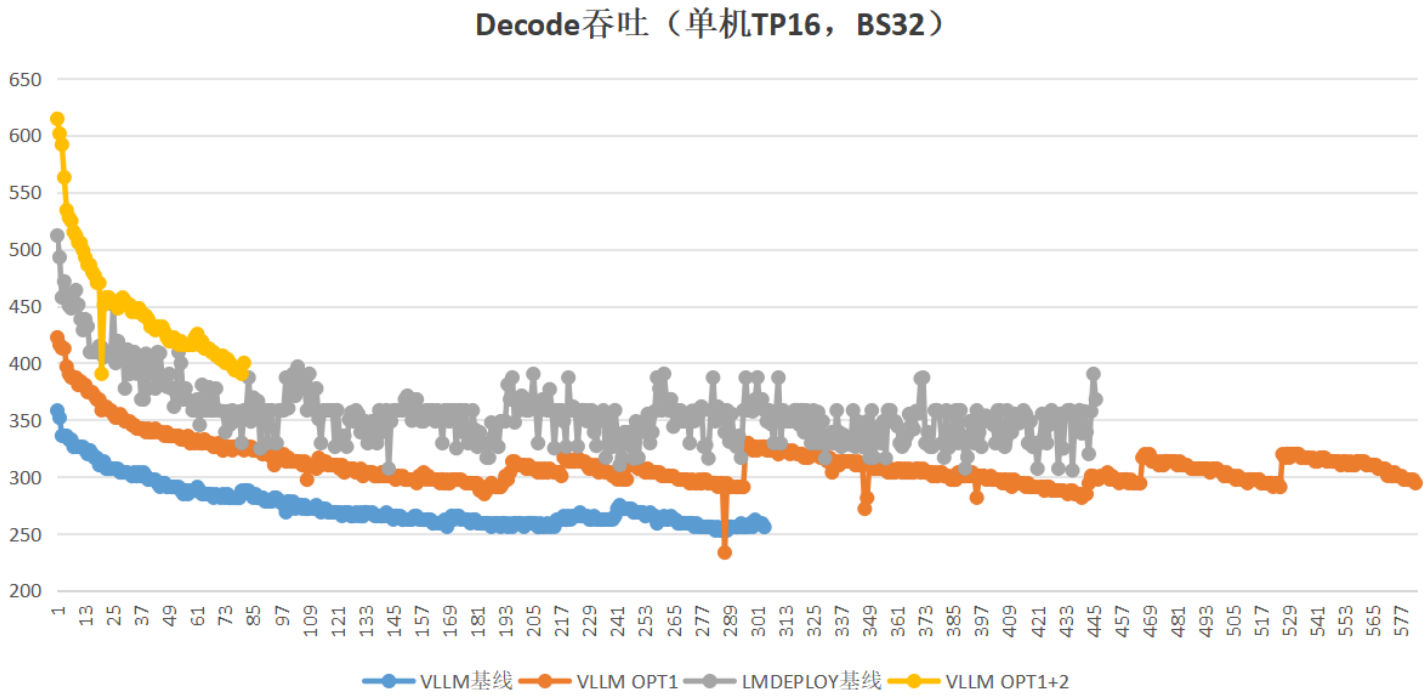
vllm-ascend > vllm_ascend > sample > sampler.py

```
25  class AscendTopKTopPSampler(TopKTopPSampler):
32      ) -> torch.Tensor:
38
39          if p is None and k is None:
40              return logits
41
42          probs = logits.softmax(dim=-1)
43          probs_sort, _ = probs.sort(dim=-1, descending=False)
44
45          if k is not None:
46              top_k_count = probs_sort.size(1) - k.to(
47                  torch.long)  # shape: (batch, )
48              top_k_count = top_k_count.unsqueeze(dim=1)
49              top_k_cutoff = probs_sort.gather(-1, top_k_count)
50
51              # Make sure the no top-k rows are no-op.
52              no_top_k_mask = (k == logits.shape[1]).unsqueeze(dim=1)
53              top_k_cutoff.masked_fill_(no_top_k_mask, -float("inf"))
54
55              elements_to_discard = probs < top_k_cutoff
56              logits.masked_fill_(elements_to_discard, -float("inf"))
57
58          if p is not None:
59              cumprob = torch.cumsum(probs_sort, dim=-1)
60              top_p_mask = cumprob <= 1 - p.unsqueeze(dim=1)
61              top_p_mask[:, -1] = False  # at least one
62
63              top_p_count = top_p_mask.sum(dim=-1).unsqueeze(1)
64              top_p_cutoff = probs_sort.gather(-1, top_p_count)
65              elements_to_discard = probs < top_p_cutoff
66              logits.masked_fill_(elements_to_discard, -float("inf"))
67
68          return logits
```

该版本巧妙地改变了寻找 Top-K/Top-P 的思路。vllm_ascend 通过排序和mask过滤，省掉了后处理的这个17ms的scatter。

经过测试，相对于原本vllm版本21ms的后处理，vllm_ascend版本的后处理只要2ms。

最终优化效果



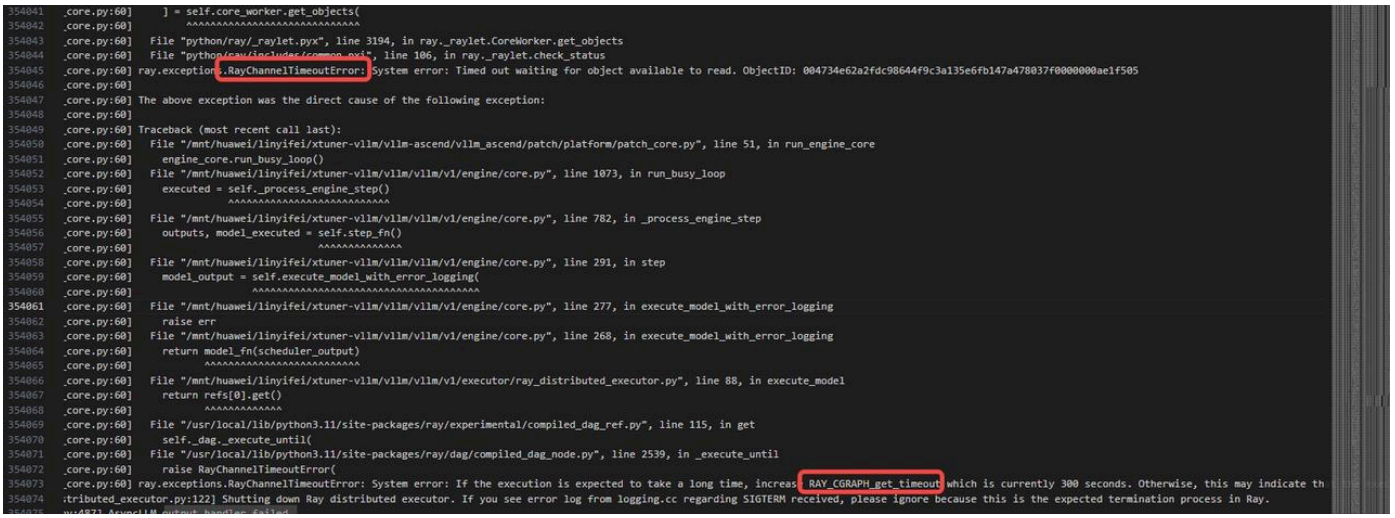
如上图，GMM亲和性计算，性能提升20%；后处理优化，性能提升30%，最终实现吞吐超过LMDeploy。

优化项	提升幅度	技术原理
SP4 → SP2	训练效率 +30%（单轮 -300s）	减少通信开销
GMM 亲和性计算	推理性能 +20%	计算密度提升
后处理链路优化	推理性能 +30%	减少冗余计算

RL/分布式稳定性治理

- **问题：** DP + 重计算场景下 Ray 超时

  在DP场景下，会偶现以下ray超时报错的问题，通过堆栈可以定位到是模型forward推理时的异常超长时延导致的，类似问题特战队之前在qwen3 235B时也遇到，在KV Cache满时，DP和重计算配合可能有问题，但是未定位到根因。



KV Cache的散点图【待补充】

可以看到，在rollout推理过程中，后半段KV Cache不足很容易进入重计算

- **方案：** 为了缓解DP+重计算的问题，当KV Cache达到某个阈值时（90%），我们阻塞prefill下发，从而避免更多的重计算。

五、 长远规划与演进路线

本次架构升级至 XTuner+vLLM，不仅是解决了当下的精度与适配瓶颈，更是为浦江大模型**选择了具备长期生命力的技术主航道**。vLLM 凭借 PagedAttention 等先进的内存管理设计，已确立了业界推理标准；而 **vLLM-Ascend** 依托国产算力底层的先进架构支持，实现了软硬件的深度协同。

这一技术选型使我们具备**双向复用**的独特优势：**对外**无缝接入 vLLM 开源社区的最新优化成果，**对内**直接获得 vllm-ascend 研发团队的底层支撑，确保架构具备持续演进能力。未来，我们将沿着“稳定-高效-智能”的路径稳步推进：

- **短期目标：夯实底座，支撑规模化训练**
  重点完善 **KV Cache 流控机制**，解决高并发下的显存管理难题，确保系统在 **DP4TP4** 超大规模分布式场景下的极致稳定性，为 RL 大规模迭代扫清障碍。
- **中期目标：深度优化，突破性能极限**
  打通**训推算子共享**链路以降低显存冗余，引入 **Speculative Decoding（投机采样）** 加速 Rollout 生成，并适配 **动态 MoE 扩缩容** 架构，在保证精度的前提下大幅提升系统吞吐。
- **远期愿景：构建智能弹性推理池**
  打造**自适应弹性推理池**，实现计算资源的动态调度；支持**多模型热切换**，让不同尺寸、不同架构的模型能够即插即用，最终构建灵活、高效的国产化 MaaS 基础设施。