vLLM大语言模型推理系统实验报告

周楷竣，柯竟俞，刘子彬

0. vLLM简介

vLLM是一个高效的大语言模型(LLM)推理和服务框架。它的主要特点包括:

1. PagedAttention机制

- 创新性地引入了虚拟内存管理的思想来管理注意力缓存(KV Cache)
- 将连续的内存空间分成固定大小的页(blocks),动态分配和回收
- 有效解决了KV Cache碎片化问题,提高内存利用率

2. Continuous Batching

- 支持动态批处理,新请求可以立即加入正在处理的批次
- 不需要等待整个批次完成才能处理新请求
- 显著提高了系统吞吐量

3. 推理过程
   vLLM的推理过程主要分为两个阶段:

a) Prefill阶段

- 处理输入prompt的所有token
- 生成并缓存所有token的KV Cache
- 计算第一个输出token的概率分布

b) Decode阶段

- 基于概率分布采样生成新token
- 只需处理新生成的token
- 复用之前缓存的KV Cache

1. 调度器实现原理

vLLM调度器是一个专为优化生成式模型而设计的调度器。与普通的任务调度器相比，它的职责更为复杂。除了需要管理多个请求的排队和处理外，还需要动态分配GPU和CPU内存、控制数据的加载与交换，确保系统在不同负载下均能高效运行。

1.1 调度器设计目标

vLLM调度器的设计旨在优化多任务环境中的GPU和CPU资源利用率。它通过以下机制来实现目标：

1. 优先级调度：确保关键任务或延迟敏感任务优先处理
2. 资源预算控制：定义每轮调度的最大内存消耗和任务数量限制

3. 抢占策略：在资源不足时，执行被动的抢占，可暂停低优先级任务来处理高优先级请求

1.2 核心组件：状态队列

vLLM中使用`SequenceGroup`来存储一个推理请求的相关信息，并使用三个队列来记录所有请求。队列的属性如下：

- **`WAITING`**：一个请求处在waiting队列中，即表示其没有做过prefill。
- **`RUNNING`**：一个请求处在running队列中，即已经开始做推理。
- **`SWAPPED`**：一个请求正在swapped队列中，表示此时gpu资源不足，相关的seq_group被抢占，导致其暂停推理，相关的KV block被置换到cpu上（swap out），等待gpu资源充足时再置换回来重新计算（swap in）。

```
class Scheduler:
    def __init__(self):
        self.waiting: Deque[SequenceGroup] = deque()  # 等待队列
        self.running: Deque[SequenceGroup] = deque()  # 运行队列
        self.swapped: Deque[SequenceGroup] = deque()  # 换出队列
```

1.3 调度流程

vLLM调度器的工作流程可以分为以下几个主要阶段：

请求入队

新请求首先被封装为SequenceGroup对象，包含了请求的优先级、到达时间等元数据。调度器将其放入waiting队列，等待后续处理。这个阶段主要是请求的初始化和排队。

预填充处理

调度器从waiting队列中选择合适的请求进行预填充。由于预填充需要处理整个输入序列，这个阶段需要较大的计算资源。调度器会根据SchedulingBudget检查当前是否有足够的资源来处理该请求。如果资源充足，则分配GPU内存并开始计算；否则请求继续等待。

解码处理

预填充完成后的请求会被移至running队列。在解码阶段，请求会逐步生成新的token。这个过程是增量的，每次只需要处理新生成的token。调度器需要在多个正在解码的请求之间合理分配计算资源，确保系统的整体效率。

资源不足的抢占处理

在若干个推理阶段后，若某一时刻GPU上的资源不够了，某个seq_group不幸被调度器抢占（preemption），它相关的KV block也被swap out到cpu上。此时所有seq的状态变为swapped。当一个seq_group被抢占时，对它的处理有两种方式：

- **Swap**：如果该seq_group下的seq数量 > 1，此时会采取swap策略，即把seq_group下所有seq的KV block从gpu上卸载到cpu上。

- **Recomputation**：如果该seq_group下的seq数量 = 1，此时会采取recomputation策略，即把该seq_group相关的物理块都释放掉，然后将它重新放回waiting队列中。等下次它被选中推理时，从prefill阶段开始重新推理。

1. 实验环境

1.1 硬件环境

- GPU: NVIDIA GeForce RTX 4090
- CPU: AMD Ryzen 9 5950X 16-Core Processor
- 内存: 128GB

1.2 软件环境

- Python 3.9
- vLLM 0.6.4

1.3 部署模型

我们选用了通义千问最新发布的Qwen2.5-7B-Instruct模型作为实验对象，运行所占用显存指定为gpu_memory_utilization=0.9。该模型是一个轻量级但性能优秀的开源模型,适合用于推理性能测试。

2. 用户请求场景设计

2.1 请求生成方法

为了全面评估vLLM调度器在不同负载场景下的性能，我们实现了一个模拟多用户并发请求的测试框架。该框架通过配置不同的负载参数，可以模拟从轻负载到重负载的各种场景。具体实现如下：

```
def simulate_requests(mode, load_scenario):
    """模拟多用户在不同负载场景下的并发请求

    Args:
        mode: 调度策略模式
        load_scenario: 负载场景(low/medium/high)
    """
    scenario_configs = {
        "low": {
            "user_count": 8,
            "lambda_requests": 5,
            "lambda_frequency": 0.05,
            "weights": [0.5, 0.3, 0.15, 0.05],
            "length_lambdas": [20, 100, 2000, 15000]
        },
        "medium": {
            "user_count": 12,
            "lambda_requests": 10,
            "lambda_frequency": 0.2,
            "weights": [0.2, 0.3, 0.35, 0.15],
```

```
            "length_lambdas": [20, 100, 2000, 20000]
        },
        "high": {
            "user_count": 24,
            "lambda_requests": 20,
            "lambda_frequency": 0.5,
            "weights": [0.1, 0.2, 0.35, 0.35],
            "length_lambdas": [100, 1500, 5000, 30000]
        }
    }

    # 获取场景配置
    config = scenario_configs[load_scenario]
    user_count = config["user_count"]

    # 生成所有用户的请求事件
    all_events = []
    for user_id in range(1, user_count + 1):
        # 使用泊松分布生成每个用户的请求数量
        user_request_count = np.random.poisson(config["lambda_requests"])
        if user_request_count <= 0:
            continue

        # 生成请求长度和时间间隔
        lengths = generate_mixed_lengths(
            user_request_count,
            config["weights"],
            config["length_lambdas"]
        )
        intervals = np.random.exponential(
            scale=1.0 / config["lambda_frequency"],
            size=user_request_count
        )
        send_times = np.cumsum(intervals)

        # 添加到事件列表
        for i in range(user_request_count):
            all_events.append((send_times[i], user_id, lengths[i]))

    # 按时间排序并并发执行
    all_events.sort(key=lambda x: x[0])
    with ThreadPoolExecutor(max_workers=user_count) as executor:
        futures = []
        for idx, event in enumerate(all_events):
            futures.append(executor.submit(_process_event,
                                           idx,
                                           event,
                                           time.time(),
                                           mode,
                                           all_events))
```

在这个实现中，我们为每个负载场景定义了一组配置参数：

- user_count: 并发用户数
- lambda_requests: 每个用户的平均请求数(泊松分布)
- lambda_frequency: 请求频率(指数分布的λ参数)
- weights: 不同长度请求的权重分布
- length_lambdas: 各类请求的平均长度

请求生成过程使用了以下随机过程：

1. 用泊松分布生成每个用户的请求数量
2. 使用指数分布生成请求间的时间间隔
3. 根据配置的权重随机选择请求长度
4. 使用ThreadPoolExecutor实现并发请求的模拟

这种实现方式有以下优点：

1. 真实性：通过泊松分布和指数分布模拟了真实场景中的请求到达特性
2. 可配置性：支持通过参数调整来模拟不同的负载场景
3. 并发处理：使用线程池实现真实的并发请求模拟

在数据收集方面,我们基于真实的代码样本构建测试集。通过分析开源项目中的代码文件,我们收集了不同长度和复杂度的代码片段,并使用模型API统计其token数量。这确保了测试数据的真实性和多样性。

性能指标的采集主要关注两个关键指标:

1. TPOT(Total Processing Time per Token)
   该指标衡量系统生成每个token的平均时间，用于评估系统的整体处理效率：

```
def get_tpot(content: str):
    """测量生成所有token的总时间 (Total Processing Time per Token - TPOT)"""
    start_time = time.time()
    chat_response = client.chat.completions.create(
        model="Qwen/Qwen2.5-3B",
        messages=[
            {"role": "system", "content": "You are Qwen, created by Alibaba Cloud. You are
a helpful assistant."},
            {"role": "user", "content": content},
        ],
        temperature=0.7,
        top_p=0.8,
        max_tokens=2048,
    )

    total_time = time.time() - start_time
    complete_tokens = chat_response.usage.completion_tokens
    tpot = complete_tokens / total_time if total_time > 0 else float('inf')
    return tpot
```

2. TTFT(Time to First Token)
   该指标测量从请求开始到生成第一个token的时间，反映系统的响应速度：

```
def get_ttft(content: str):
    """测量首个token的生成时间 (Time to First Token - TTFT)"""
    start_time = time.time()
    chat_response = client.chat.completions.create(
        model="Qwen/Qwen2.5-3B",
        messages=[
            {"role": "system", "content": "You are Qwen, created by Alibaba Cloud. You are
a helpful assistant."},
            {"role": "user", "content": content},
        ],
        temperature=0.7,
        top_p=0.8,
        max_tokens=2048,
        stream=True,
    )

    for chunk in chat_response:
        ttft = time.time() - start_time
        break

    chat_response.close()
    return ttft
```

2.2 负载场景设计

我们设计了三种不同的负载场景来测试系统性能:

1. 轻负载场景

- 并发用户数: 8
- 平均请求数: 5/用户
- 请求频率: 0.05次/秒

2. 中负载场景

- 并发用户数: 16
- 平均请求数: 10/用户
- 请求频率: 0.2次/秒

3. 重负载场景

- 并发用户数: 24
- 平均请求数: 20/用户
- 请求频率: 0.5次/秒

3. 调度策略实现与对比

3.1 调度策略实现

1. 原生FCFS策略

```
def _schedule_default(self) -> SchedulerOutputs:
    """默认的FCFS调度策略实现

    主要步骤:
    1. 优先调度等待队列中的prefill请求
    2. 如果没有prefill,则调度running队列中的decode请求
    3. 如果资源充足,从swapped队列中换入请求
    """
    budget = SchedulingBudget(
        token_budget=self.scheduler_config.max_num_batched_tokens,
        max_num_seqs=self.scheduler_config.max_num_seqs,
    )

    if not swapped:
        prefills = self._schedule_prefills(budget, curr_loras=None)

    if len(prefills.seq_groups) == 0:
        running = self._schedule_running(budget, curr_loras=None)

        if len(running.preempted) + len(running.swapped_out) == 0:
            swapped = self._schedule_swapped(budget, curr_loras=None)

    return SchedulerOutputs(...)
```

2. 优先级调度(Priority Scheduling)

我们对原生vLLM的调度器进行了改进,增加了主动优先级调度机制:

优先级调度策略的核心思想是根据请求的优先级对任务进行排序和调度。在我们的实现中:

- 通过_get_priority函数计算每个请求的优先级,优先级由请求的等待时间和预期执行时间共同决定
- 维护了一个按优先级排序的等待队列,确保高优先级请求优先获得执行
- 实现了主动抢占机制,当出现更高优先级的请求时,可以暂停当前正在执行的低优先级任务
- 被抢占的任务会被临时交换到CPU内存,等待后续重新调度

```
def _should_active_preempt(self, is_swapped) -> bool:
    """检查是否需要主动抢占

    1. 对等待队列和交换队列按优先级排序
    2. 比较最高优先级请求
    3. 决定是否需要抢占
    """
    waiting_queue = deque(sorted(self.waiting, key=self._get_priority))
    swapped_queue = deque(sorted(self.swapped, key=self._get_priority))

    if is_swapped:
```

```
        # 检查等待队列中是否有更高优先级的请求
        if waiting_queue and self._get_priority(waiting_queue[0]) > \
            self._get_priority(swapped_queue[0]):
            return True
    else:
        return True
    return False
```

3. 最短作业优先(Shortest Job First)

SJF调度策略通过预测请求的完成时间来进行任务调度。该策略的主要特点:

- 使用_get_finished_time函数估算每个请求的预期完成时间
- 完成时间的估算考虑了请求的token长度、模型的生成速度等因素
- 优先调度预期完成时间最短的请求,以减少平均等待时间
- 对于长请求可能会产生"饥饿"现象,但由于短任务处理十分快速，实际测试中表现良好

```
# 在waiting队列中选择预计完成时间最短的请求
if self.scheduler_config.policy == 'sjf':
    waiting_queue = deque(
        sorted(self.waiting, key=self._get_finished_time))
```

3.2 性能对比

在不同请求长度场景下,我们对比了三种调度策略的性能指标:

1. 短请求场景(tokens < 500):

调度策略	请求数量	平均长度(tokens)	TTFT(s)	TPOT(token/s)
FCFS	30	101.0	6.327	33.379
Priority	22	100.5	16.147	30.865
SJF	18	104.0	3.379	42.497

2. 中等请求场景(500-5000 tokens):

调度策略	请求数量	平均长度(tokens)	TTFT(s)	TPOT(token/s)
FCFS	79	2900.6	12.487	31.832
Priority	111	3231.5	14.009	28.850
SJF	94	2887.5	3.641	39.338

3. 长请求场景(tokens > 5000):

调度策略	请求数量	平均长度(tokens)	TTFT(s)	TPOT(token/s)
FCFS	113	10921.8	16.212	27.544
Priority	136	11365.1	18.983	25.186
SJF	153	11235.1	19.299	23.339

3.3 结果分析

1. 短请求场景

- SJF策略在短请求场景下展现出最快的首次响应速度(TTFT=3.379s),这得益于其优先处理完成时间短的请求的特性
- Priority策略虽然TTFT较高(16.147s),但每个token的平均生成速度最快(TPOT=30.865token/s)
- FCFS策略在两个指标上都处于中等水平(TTFT=6.327s, TPOT=33.379token/s)

2. 中等请求场景

- SJF策略继续保持着出色的首次响应性能(TTFT=3.641s),但token生成速度较慢(TPOT=39.338token/s)
- Priority策略在token生成速度上表现最佳(TPOT=28.850token/s),说明其资源利用效率较高
- FCFS策略的TTFT(12.487s)相对较高,但TPOT(31.832token/s)表现尚可

3. 长请求场景

- 在长请求场景下,所有策略的TTFT都显著增加,这主要是由于长请求占用资源时间较长所致
- SJF策略展现出最好的token生成效率(TPOT=23.339token/s),说明其在处理长请求时资源利用更为合理
- Priority策略的性能指标(TTFT=18.983s, TPOT=25.186token/s)与SJF接近
- FCFS策略虽然TTFT较低(16.212s),但token生成效率最差(TPOT=27.544token/s)

4. 总结

综上，对于vllm中默认的fcfs调度策略，是一种被动的抢占策略，也是Swap队列中任务优先级大于Wait队列中任务优先级（由于遵守先来先服务策略）。我们在本任务中主要做的是学习LLM的Prefill和Decode两阶段请求，包括LLM的推理流程，理清并掌握vllm的调度框架。在以上基础上，我们将默认的被动策略修改为主动的抢占策略，让到来的高优先级（或者任务完成时间短优先）策略让这些任务在资源充足情况下也能发生主动抢占，并设计了不同的workload并进行了分析。最终发现各个策略在不同的情况下各有千秋。