AdaptiveLLM: 基于张量交换和张量重算的大语言模型推理优化技术

梁绪宁[1]，王思琪[1]，杨海龙[1]，栾钟治[1]，刘轶[1]，钱德沛[1]

[1] 北京航空航天大学，北京 100191

(liangxuning@126.com,lethean1@buaa.edu.cn,hailong.yang@buaa.edu.cn,07680@buaa.edu.cn,yi.liu@buaa.edu.cn,depeiq@buaa.edu.cn)

AdaptiveLLM: Efficient LLM inference based on swapping and re-computation

Liang Xuning[1]，Wang Siqi[1]，Yang Hailong[1]，Luan Zhongzhi[1]，Liu Yi[1]，Qian Depei[1]

[1] (Beihang University, Beijing 100191)

Abstract Large Language Models(LLMs) come with an extremely high amount of parameters, posing significant challenges for inference tasks. Traditional LLM inference services employ swapping and re-computation techniques, guaranteeing the success of generation at the cost of performance on limited GPU memory. However, existing LLM serving systems fail to search memory management schemes adaptively based on runtime information, leading to a sub-optimal performance. Furthermore, these works are inferior in the trade-off between throughput and latency, preferring on only one and compromising the other. To address the above issues, we propose *AdaptiveLLM*, an efficient LLM serving framework for inference tasks based on swapping and re-computation. Specifically, *AdaptiveLLM* implements an overhead predictor for swapping and re-computation, with an error rate lower than 2% and 4% respectively. *AdaptiveLLM* also adopts a cost-aware memory optimization algorithm and a fairness-based request scheduling algorithm based on the overhead predictor. The former improves the throughput on the server, and the latter reduces latency for the client, making a enhancement on the real-time performance. On typical LLMs and datasets, our evaluation shows that the cost-aware memory optimization algorithm improves the throughput by 10% to 40%, and the fairness-based request scheduling algorithm reduces the average weighted around time by 20% to 40%, compared with the vLLM. In conclusion, *AdaptiveLLM* achieves efficient LLM inference by making a better trade off between throughput and latency.

Key words LLM; inference; swapping; re-computation

摘要 大语言模型（*LLMs*）拥有极高的参数量，为推理任务带来 *GPU* 内存瓶颈。传统 *LLM* 推理框架引入张量交换和张量重算等技术，在有限的 *GPU* 内存上牺牲性能完成推理。然而，已有研究工作无法根据推理任务运行时信息自适应地选择内存优化技术，导致推理任务的性能无法进一步提升。同时这些工作没有实现整体吞吐率与单请求延时之间的权衡，常以二者之一作为优化目标。针对以上问题，本文面向 *LLM* 推理服务场景，提出 *AdaptiveLLM*，一款基于张量交换和张量重算的 *LLM* 推理框架。*AdaptiveLLM* 实现了张量重算和张量交换开销预测，其预测误差分别在 *2%* 和 *4%* 以下。在此基础上引入基于开销感知的内存优化策略，实现服务器端处理加速。同时引入基于公平性的用户请求调度策略，实现客户端实时请求响应。本文在常见 *LLM* 和数据集上开展实验，以 *vLLM* 作为基准程序进行对比评估。结果表明，基于开销感知的内存优化策略带来 *10%* 到 *40%* 的整体吞吐率提升，基于公平性的用户请求调度策略使平均带权周转时间降低 *20%* 至 *40%*。由此证明 *AdaptiveLLM* 在优化过程中权衡整体吞吐率与单请求延时，实现 *LLM* 高效推理。

关键词 大语言模型；推理；张量交换；张量重算

中图法分类号 TP391

基金项目：目前无人捐助此项目

1. 引言

从人脸识别 [?]、个性化推荐 [?]、到智能家居 [?]、无人驾驶 [?] 等应用领域，深度学习 [?]（Deep Learning，DL）相关技术已经融入到社会的方方面面，为人类的生产生活带来了极大的便利。自然语言处理 [?]（Natural Language Processing，NLP）作为深度学习领域的重要研究方向，长期以来备受人们关注。近年来，随着 GPU 算力的不断提升，各种语言模型也朝着复杂化，多功能化的方向迅猛发展。

大语言模型 [?]（Large Language Models，LLM）是自然语言处理领域的一个分支。LLM 通常拥有十亿级别，甚至万亿级别的参数量，因此需要海量的文本数据进行训练。同时，LLM 在多种类型的任务中展现出卓越性能，如文本摘要 [?]、机器翻译 [?]、代码生成 [?] 以及对话问答 [?] 等，拥有巨大的科研价值与商业价值。2021 年 GPT-3 模型 [? ?] 的问世标志着 LLM 领域的里程碑，自此，各大科研机构纷纷投入到相关研究中，各种 LLM 层出不穷，使得该领域的热度空前高涨。

复杂的结构和爆炸式增长的参数量为 LLM 带来了卓越的应用效果，但却为众多的科研工作者们带来了巨大难关。LLM 在训练时消耗资源多，花费时间长，失败风险高，性能要求严，使得 LLM 的训练成本急剧上涨。推理任务的成本相对较低，但极高的内存占用也为推理任务的高效执行带来了独特的挑战。例如，GPT-175B 模型仅在权重加载环节就需要消耗 325GB 的 GPU 内存空间 [?]，在传统的 LLM 推理框架下，需要使用至少 5 个 NVIDIA A100 GPU（80GB），且需要引入复杂的并行化策略。因此，降低运行时资源消耗对 LLM 推理任务至关重要。

为了应对较高参数量带来的 GPU 内存瓶颈，本文提出了一款基于张量交换 [?]（Swapping）和张量重算 [?]（Recomputation）的 LLM 推理服务框架，引入先进的显存优化策略和用户请求调度策略实现 LLM 的高效推理。该框架针对服务器端实现高吞吐，针对客户端实现实时请求处理，进一步解决整体吞吐率与单请求延时在性能优化上长期以来存在的矛盾。

传统的 LLM 推理框架拥有诸多可改进之处，下面列举其中最为显著的两点。

首先，通过张量交换和张量重算等内存优化技术，可以在有限的 GPU 内存空间中增加批处理大小。然而，上述两项内存优化技术对 LLM 推理性能的影响十分复杂，取决于服务器硬件环境（如 GPU 计算能力、GPU-CPU 传输带宽）、用户设置（如生成新 token 时的采样方式）、LLM 与数据集选取、以及推理任务的运行时信息等等。已有的 LLM 推理框架 [? ? ?] 在面对 GPU 内存瓶颈时固定调用张量交换或张量重算技术，而无法根据上述信息选择更优者，显著影响推理任务的性能。

其次，在针对 LLM 推理任务进行性能优化时，传统工作 [? ? ?] 或者以整体吞吐率为单一导向，或者以单请求延时为单一导向，而没有在二者间进行权衡。整体吞吐率是面向服务器端的性能优化指标，体现了服务器端的处理效率。单请求平均延时是面向客户端的性能优化指标，体现了用户请求处理的实时性。二者均在 LLM 应用程序中占有重要地位。

针对传统工作的不足之处，本文设计了 AdaptiveLLM, 一款基于张量交换和张量重算的 LLM 推理服务框架。具体而言，本文开展了以下工作：

- **本文设计了一款张量重算分析器，实现张量重算开销的精准预测。**

  通过算子粒度的计算复杂度分析识别张量重算开销的影响因素，并建立回归预测模型获取单步迭代执行时间。实验表明，张量重算开销的预测误差在 2% 以内。

- **本文设计了一款张量交换分析器，实现张量交换开销的精准预测。**

  利用用户请求 KV Cache 的内存占用和 GPU-CPU 间通信效率对张量交换的数据传输开销进行了预估。实验表明，张量交换开销的计算误差在 4% 以内。

- **本文设计了一个基于张量交换和张量重算的自适应 LLM 推理优化器。**

  该优化器引入基于开销感知的内存优化策略，提升整体吞吐率，实现了面向服务器端的处理加速。同时引入基于公平性的用户请求调度策略，降低单请求平均延时和带权周转时间，实现了面向客户端的实时请求处理。

- **本文搭建了一款 LLM 推理服务框架 AdaptiveLLM，并进行实验评估。**

  AdaptiveLLM 基于 vLLM 框架实现上述设计的张量重算分析器、张量交换分析器与自适应 LLM 推理优化器等功能模块，并在典型 LLM（OPT [?]、Llama [?]）和数据集（Summary [?]、Chatbot [?]、Alpaca [?]）上开展实验。结果表明，本文能够实现 10% 到 40% 的整体吞吐率提升，且将用户请求平均带权周转时间降低 20% 至 40%。以此证明本文的优化技术实现了整体吞吐率与单请求延时的权衡，完成 LLM 高效推理。

2. 背景知识

本章介绍有关 AdaptiveLLM 的背景知识。由于 AdaptiveLLM 主要面向 LLM 推理过程中产生的 KV Cache 内存占用进行优化，因此本章将在第一节阐明 KV Cache 在 LLM 推理任务中的功能，在第二节论述传统工作中面向 KV Cache 的内存优化技术。

2.1. KV Cache 的提出

LLM 推理任务以 token 作为输入与输出的基本单位。对于生成式推理任务，每次前向传播计算仅生成一个新 token。一般来说，其包含两个阶段：prefill 阶段读取用户输入的 token 序列，生成第一个 token；decode 阶段分为多步进行，依次生成后续 token，直至得到终止 token。在推理过程中，每个 token 拥有一个 key-value 张量对，为自注意力机制下的编码结果。

在 decode 阶段中，每个 token 的计算均依赖于前序 token 的 key 值和 value 值。如果每次计算前都重新调用自注意力机制来获取前序 token 的 key-value 张量，则会产生大量不必要的计算开销。主流 LLM 推理服务 [? ? ? ?] 框架普遍采用 KV Cache 数据结构来保存这些 token 的 key-value 张量，方便后续 token 的生成，避免重复计算，其工作原理如图1所示。

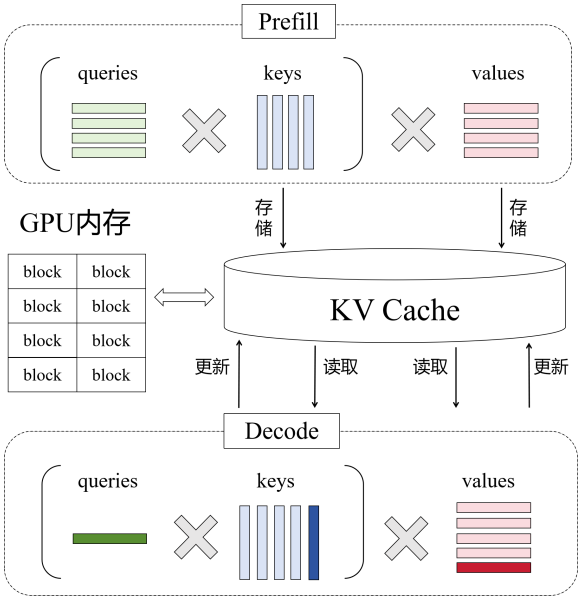

图 1 KV Cache 的功能示意图

然而，随着后续 token 的不断生成，KV Cache 迅速扩展，产生推理内存瓶颈。例如，在 OPT-13B 模型中，对于一个长度为 100 的用户请求，其 KV Cache 能够占用 39.1MB 的内存空间。有限的 GPU 内存将批处理大小限制在较低水平，阻碍推理并发度的进一步提升，进而限制吞吐率。

不同于 LLM 参数张量，KV Cache 占用的内存空间在对应用户请求推理完毕后被释放。其内存占用量大、动态性高，拥有较大的优化空间，因此 AdaptiveLLM 的内存优化策略将针对 KV Cache 实现。

2.2. KV Cache 的内存优化

KV Cache 的引入方便了计算过程，却带来内存瓶颈，使得 LLM 推理性能的提升无法达到预期水平。下面介绍针对 KV Cache 内存占用的一些优化工作。

(1) 内存碎片优化

在传统 LLM 推理服务框架 [?] 中，内存管理器按照用户定义的序列长度上限，为每个请求设置一块固定大小的 GPU 内存来存储 KV Cache。但用户请求长度的差异性导致内碎片的大量产生。为了解决该问题，部分 LLM 推理框架 [?] 能够基于历史信息来预测输出长度，并按照预测值分配内存。然而，预测误差会导致输出截断，且旧请求的完成与新请求的加入使得内存中产生很多外碎片。随着新请求的不断到来，内碎片与外碎片在内存中积累，严重影响了内存空间的高效使用。基于这些问题，vLLM 框架 [?] 引入了 Paged Attention 机制，基于 OS 页式内存管理思想，将 GPU 内存划分成块，并通过维护块表来支持 KV Cache 在内存空间中的不连续存储。该机制基本消除了内碎片和外碎片现象，大大提升内存利用率。

(2) 张量交换与张量重算

为了攻克推理内存瓶颈，传统框架引入了张量交换技术 [? ? ?]，将暂时不会使用的 KV Cache 传输至 CPU 中，在计算需要时重新传输至 GPU 中。然而，CPU-GPU 间有限的 PCIe 带宽使得换出和换入过程产生不可忽略的通信开销，限制吞吐率，降低推理性能。部分研究提出 [?]，当张量交换带来的开销超过重新调用自注意力机制的开销时，应选择后者来获取所有前序 token 的 key-value 张量，也称张量重算。具体来说，内存管理器直接删除重算请求对应的 KV Cache，在其被调度时执行一次 prefill 阶段来代替原本应该执行的 decode 阶段。重算与交换的联合使用缓解了通信开销问题，然而，当 GPU 内存不足时，如何在二者中进行选择成为了新的困境。AdaptiveLLM 针对此问题设计了基于开销感知的内存优化策略，能够预测二者的开销，并选择开销小的过程执行。

3. 优化设计

本章介绍了本文工作的具体设计。第一节给出 AdaptiveLLM 的整体设计方案，后面的章节将分别介绍 AdaptiveLLM 中不同的功能模块。

3.1. 整体架构

AdaptiveLLM 实现了三个主要功能模块，包括张量重算分析器、张量交换分析器和自适应 LLM 推理优化器，其整体架构如图2所示。

其中，张量重算分析器基于用户请求长度、模型隐藏维度、模型层数和批处理大小来预测重算开销。张量交换分析器基于 KV Cache 内存占用和 GPU-CPU 双向传输带宽来预测交换开销。自适应 LLM 推理优化器包含内存优化决策器和用户请求调度器。具体来说，当 GPU 内存不足时，内存优化决策器引入基于开销感知的内存优化策略，选择优先级最低的用户请求，收集张量重算分析器提供的重算开销预测值与张量交换分析器提供的交换开销预测值。选择开销较小的内存优化方式，而后交付相应的执行器。该过程也称为“抢占调度”。当 GPU 内存空余时，用户请求调度器使用基于公平性的用户请求调度策略，在满足公平性的前提下尽可能多地调度剩余用户请求，避免 GPU 资源浪费。该过程也称为“启动调度”。在推理过程中，内存优化决策器与用户请求调度器共享运行时内存占用信息。二者高效协同，实现整体吞吐率与单请求延时的权衡。

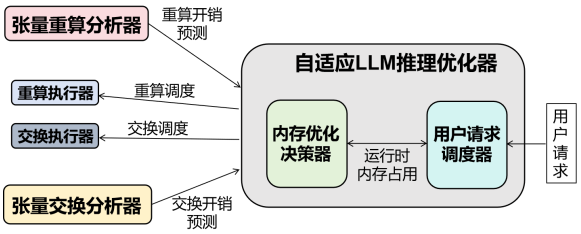

图 2 整体设计架构

3.2. 张量重算分析器

张量重算技术的时间线流程如图3所示。抢占调度时，重算执行器在内存中删除用户请求的 KV Cache 张量。启动调度时，执行一次 prefill 阶段来恢复被删除的数据。因此，张量重算引入的额外开销等于被抢占请求执行 prefill 阶段的时间。

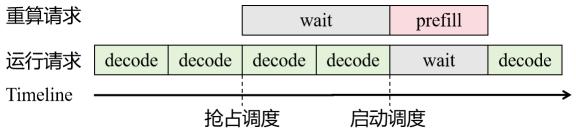

图 3 张量重算时间线流程

本文以 OPT 和 Llama 模型为例，通过算子粒度复杂度分析来识别单步推理时间的影响因素。

(1) 算子粒度开销分析

OPT 和 Llama 模型中包含 5 种不同的算子：ReLU、Norm、Linear、SiluAndMul 和 Attention，其计算流程如图4所示。图中 X_i,Y_i 是由用户输入决定的张量维度；$input\_dim$,$output\_dim$,$head\_size$ 是由算子本身决定的张量维度。

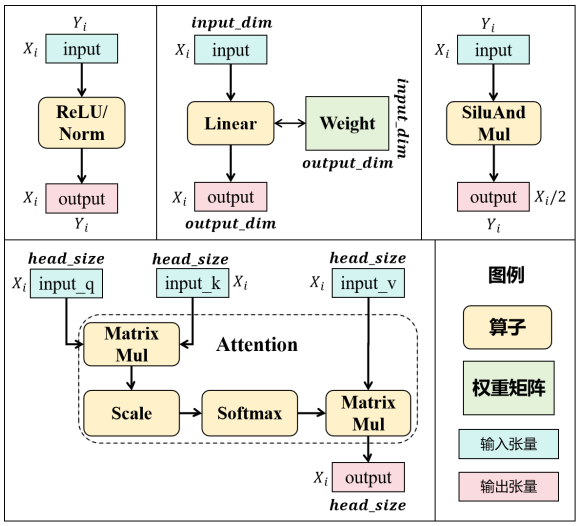

图 4 四种算子的计算流程

下面分别对这些算子进行复杂度分析。

- **ReLU 算子**：逐位调用激活函数进行计算，其时间复杂度为 $O(X_i * Y_i)$。
- **Norm 算子**：是 LayerNorm、RMSNorm（仅在 Llama 模型中）等多种归一化算子的统称，其时间复杂度为 $O(X_i * Y_i)$。
- **Linear 算子**：是 RowParallelLinear，ColumnParallelLinear 等多种线性层算子的统称，将输入向量从 $input\_dim$ 维空间映射到 $output\_dim$ 维空间中，其计算复杂度为 $O(X_i * input\_dim * output\_dim)$。
- **SiluAndMul 算子**：该算子仅出现在 Llama 模型的 MLP 层中，将输入向量的指定维度减半，其时间复杂度为 $O(X_i * Y_i)$。
- **Attention 算子**：属于复合操作，由矩阵乘法、缩放和 Softmax 激活等底层算子组成，整体计算过程如公式1，其时间复杂度为 $O(X_i^2 * head\_size)$。

$$Attention(Q,K,V) = softmax(\frac{Q \times K^T}{\sqrt{h}} \times V) \quad (1)$$

根据算子粒度复杂度分析，可以识别出 4 项有关 LLM 单步推理执行时间的影响因素，分别为：LLM

层数、LLM 隐藏维度、单请求需要处理的 token 数量、和批处理大小。

(2) 单步推理开销预测模型

单步迭代执行时间预测是一项拥有 4 个输入变量，1 个输出变量的回归预测任务。根据算子粒度时间复杂度分析可知，输出变量与输入变量之间存在多项式依赖关系。因此，本文共选用了 8 个回归模型，包括线性回归模型、决策树回归模型、随机森林回归模型、岭回归模型、套索回归模型、弹性回归模型、梯度提升回归模型、和 K-临近回归模型。针对每种回归模型，对不同的多项式拟合次数（1 到 5）进行测试。选择在测试集上预测误差最小的配置，并将其部署到 AdaptiveLLM 的张量重算分析器中。

3.3. 张量交换分析器

张量交换的时间线流程如图5所示。抢占调度时，交换执行器将用户请求的 KV Cache 从 GPU 传输到 CPU 中（换出阶段）。启动调度时，将其 KV Cache 传输回 GPU 中（换入阶段）。因此，张量交换引入的额外开销等于被抢占请求的换出时间与换入时间之和。

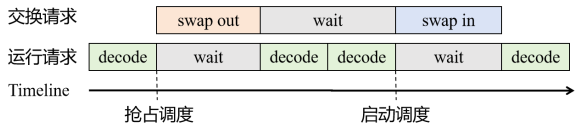

图 5 张量交换时间线流程

换出开销与换入开销的计算方式如公式2所示。

$$\begin{aligned} SwapOut\_Time &= \frac{KVCach\_Mem}{DtoH - bandwidth} \\ SwapIn\_Time &= \frac{KVCache\_Mem}{HtoD - bandwidth} \end{aligned} \tag{2}$$

其中 $DtoH - bandwidth$ 是 GPU 传输数据到 CPU 的带宽，$HtoD - bandwidth$ 是 CPU 传输数据到 GPU 的带宽。AdaptiveLLM 继承了 vLLM 所采用的 Paged Attention 技术，在 GPU 和 CPU 内存中划分大小固定的 Block，用于存储 KV Cache。每个 Block 的内存占用如公式3所示，其中 $block\_size$ 是用户定义的参数，用于调整 Block 大小。

$$\begin{aligned} block\_mem = 2 \times num\_layers \times hidden\_size \\ \times block\_size \times sizeof(float16) \end{aligned} \tag{3}$$

因此，假设一个用户请求的长度为 n，占用 GPU block 的数量为 $block\_num$，则其 KV Cache 占用的总内存空间如公式4所示。

$$\begin{aligned} KVCache &= block\_mem \times block\_num \\ &= block\_mem \times \lceil \frac{n}{block\_size} \rceil \end{aligned} \tag{4}$$

由此可以计算出张量交换引入的额外开销。在上述公式中，换入换出传输带宽是由实验环境所决定的，在传输数据量较大时基本保持稳定。而 $block\_size$ 与 $block\_mem$ 在推理任务中均保持不变。因此对于不同的用户请求，其区别仅在于序列长度 n 的不同。

3.4. 内存优化决策器

当 GPU 内存不足时，需要调用内存优化策略。AdaptiveLLM 中的内存优化策略分为张量交换和张量重算两种。根据上文的分析，张量交换引入的额外开销等于 KV Cache 的换出开销与换入开销之和；张量重算引入的额外开销等于 prefill 过程的开销。

Algorithm 1 Mem_Schedule

Input: 运行队列 $running$, 重算兼等待队列 $waiting$, 交换队列 $swapped$
Output: 无

```
sorted(running, key =< priority >, order = asc)
while require_mem(running) > avail_gpu_mem() do
    req ← running.pop()                    // 优先级最低的用户请求
    if kvcache_mem(req) <= avail_cpu_mem() then
        recomp_time ← GET_RECOMP_TIME(req)
        swap_time ← GET_SWAP_TIME(req)
        if swap_time < recomp_time then
            preempt_mode ← SWAP
        else
            preempt_mode ← RECOMP
        end if
    else
        preempt_mode ← RECOMP
    end if
    if preempt_mode == SWAP then
        SWAP(req)                          // 交付张量交换执行器
        swapped.append(req)
    else
        RECOMP(req)                        // 交付张量重算执行器
        waiting.append(req)
    end if
end while
```

张量交换和张量重算带来的额外开销成为阻拦用户请求并发度进一步提升的瓶颈，因此内存优化方式的选择尤为重要。在不同的运行环境中，应该使用不同的内存优化策略，减少额外开销。然而，vLLM 在内存优化策略的选择上并未考虑开销问题。针对使用贪心采样策略的用户请求，其执行张量重算。针对使用并行采样或束搜索采样策略的用户请求，其

执行张量交换。因此在面对 GPU 内存瓶颈时难以有效地压缩开销，进而无法提升吞吐率。AdaptiveLLM 则对两种内存优化方式的开销进行比较，选择更优者执行。内存优化决策器的工作流程如算法1所示。

当剩余的 GPU 内存空间不足以存放运行队列在下一次迭代中产生的 KV Cache 时（第 2 行），内存优化决策器进入工作状态。选择运行队列中优先级最低的用户请求（第 3 行），调用张量交换分析器和张量重算分析器来预测其张量交换和张量重算开销（第 5、6 行）。如果交换开销小于重算开销，则将该请求交付交换执行器处理（第 7、8 行），否则交付重算执行器处理（第 9、10 行）。以上过程循环执行，直至运行队列在下一次迭代中产生的 KV Cache 能够全部存放到 GPU 内存中。此外，当 CPU 内存不足时，内存优化决策器将直接调用张量重算技术，而跳过开销预测和比较过程（第 12，13 行）。

3.5. 用户请求调度器

AdaptiveLLM 维护三个用户请求队列：*waiting* 队列、*running* 队列与 *swapped* 队列。*waiting* 队列存储初次进入调度系统，还未执行过，或者因张量重算而失去 KV Cache 的用户请求；*running* 队列存储正在运行（执行 decode 阶段）的用户请求；*swapped* 队列存储被换出到 CPU 中的用户请求。这三个队列之间拥有以下调度规则：

- *running* 队列中的用户请求运行完毕后会返回客户端，否则继续运行。
- 当 GPU 内存条件允许时，*swapped* 队列中的用户请求可以直接转移至 *running* 队列中。
- 当 GPU 内存条件允许时，*waiting* 队列中的用户请求可以转移至 *running* 队列中，但需要先执行 prefill 阶段。

如果剩余的 GPU 内存空间不足以存储 *running* 队列在下一次迭代中产生的 KV Cache，则需要内存优化决策器进行抢占调度。如果剩余的 GPU 空间足够，则考虑扩充 *running* 队列，以避免浪费 GPU 资源。在扩充 *running* 队列时，用户请求调度器将部分请求从 *swapped* 队列或 *waiting* 队列中转移至 *running* 队列中。但由于两种转移方式存在较大差别（是否需要执行 prefill 阶段），因此每次扩充 *running* 队列时，或者仅从 *swapped* 队列进行调度，或者仅从 *waiting* 队列进行调度，而无法同时调度两个队列。用户请求调度器的工作流程如算法2所示。

客户端发送的用户请求进入 *waiting* 队列中，而 *running* 队列和 *swapped* 队列最初为空（第 1-3 行）。

Algorithm 2 Req_Schedule

Input: 大模型 LLM, 待执行的用户请求队列 L
Output: 无

```
1:  w ← L                                          // 初始化 waiting 队列
2:  r ← empty_list                                 // 初始化 running 队列
3:  s ← empty_list                                 // 初始化 swapped 队列
4:  while ¬(w.is_empty() ∧ s.is_empty() ∧ r.is_empty()) do
5:      MemSchedule(r, w, s)                       // (内存不足时) 抢占调度
6:      s_sche ← SWAP_IN_SCHE()                    // 换入队列构建
7:      w_sche ← RECOMP_SCHE()                     // 重算队列构建
8:      if GET_PRI(w_sche) ≤ GET_PRI(s_sche) then
9:          r = r + s_sche                         // 换入
10:         s = s − s_sche
11:     else
12:         LLM.PREFILL(w_sche)                    // 重算
13:         r = r + w_sche
14:         w = w − w_sche
15:         continue
16:     end if
17:     LLM.DECODE(r)                              // 单次推理迭代
18:     for req in r do
19:         if req.is_finished() then
20:             r.remove(req)                      // 移除完成的请求
21:         end if
22:     end for
23: end while
```

客户端发送的用户请求进入 *waiting* 队列中，而 *running* 队列和 *swapped* 队列最初为空（第 1-3 行）。当 GPU 内存不足时，调用内存优化算法进行抢占调度（第 5 行），否则扩充 *running* 队列。

用户请求调度器尽可能多地寻找能从 *swapped* 队列转移至 *running* 队列的用户请求（第 6 行），和能从 *waiting* 队列转移至 *running* 队列的用户请求（第 7 行）。对它们进行优先级比较（第 8 行），若前者的优先级均值较高，则将其直接转移到 *running* 队列中（第 9-10 行）；若后者的优先级均值较高，则其执行 prefill 阶段后（第 12 行）转移至 *running* 队列中（第 13-14 行），同时直接进入下一轮迭代（第 15 行）。需要注意的是，当 GPU 内存不足时，无法实现从 *swapped* 队列或 *waiting* 队列向 *running* 队列的调度，即 *w_sche* 和 *s_sche* 队列均为空，此时也就不存在后续的优先级比较过程了。

在以上调度操作完成后，*running* 队列应当为非空的，否则推理过程无法继续。*running* 队列执行 decode 阶段（第 17 行），将已完成的用户请求移除后进入下一次迭代（第 18-22 行）。

对于一个用户请求，定义其优先级等于处理时间除以序列长度，其中处理时间等于当前时刻减去该用户请求初次进入 *waiting* 队列的时刻。定义用户请求队列的优先级等于所有用户请求优先级的平均值。当用户请求初次进入 *waiting* 队列时，其序列长度较

短，优先级增长较为迅速，能够被很快处理。而在等待过程中，其优先级在不断提升，避免了饥饿现象。

4. 实验验证

本章介绍实验部分。第一节为实验平台软硬件配置。第二节介绍 LLM 与数据集的选取，以及实验参数设置。第三节针对基于开销感知的内存优化策略，进行吞吐率测试。第四节针对基于公平性的用户请求调度策略，进行实时性测试。第五节分析张量交换与张量重算预测误差。第六节进行其它测试工作。

4.1. 实验环境

本文开展实验使用的服务器软硬件配置如表1所示。使用 Intel(R) Xeon(R) CPU 和 NVIDIA A800 80GB GPU 作为硬件环境，使用 CUDA-11.8、PyTorch-2.0.1、Ray-2.7.1 以及 vLLM-0.2.5 作为底层框架进行开发。服务器使用 PCIe 连接实现 GPU-CPU 通信。

表 1 实验平台的软硬件配置

软件/硬件	型号/版本
CPU	Intel(R) Xeon(R) CPU @ 2.60GHz
GPU	NVIDIA A800 PCIE 80GB
OS	CentOS Linux 7 (Core)
CUDA	11.8
PyTorch	2.0.1
Ray	2.7.1
vLLM	0.2.5

4.2. 实验设置

本文选用 OPT [?]（OPT-13B、OPT-30B）和 Llama [?]（Llama-13B、Llama-32.5B）作为实验模型，在三个常见数据集（Chatbot [?]、Alpaca [?]、Summary [?]）上进行测试。数据集信息如表2所示。

表 2 实验数据集选取

数据集	样本总数	平均输入长度	任务类型
Chatbot	258064	17.02	对话类
Alpaca	68912	19.66	指令类
Summary	1799	340.48	摘要类

三个数据集的样本序列长度分布曲线如图6所示。Chatbot 和 Alpaca 中大多数序列长度较短，而 Summary 中序列长度展现出很大差异性，且包含长序列。它们涵盖了 LLM 应用程序面临的大部分场景。

实验过程中的参数设置模拟 LLM 应用程序在多任务并发场景下的运行状态。具体来说，本文将 GPU Block 数量设置为 128，将 CPU Block 数量设置为 64。针对 12 个实验组，在相应数据集中使用简单随机抽样法选取 1000 个样本进行后续测试。

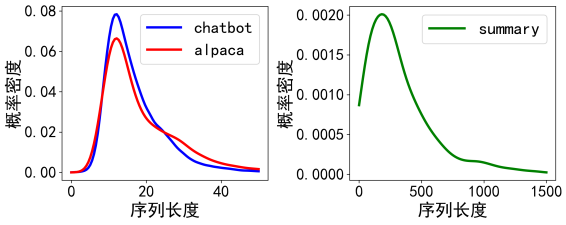

图 6 序列长度分布曲线

4.3. 吞吐率测试

本文以 vLLM 作为基准框架，针对 AdaptiveLLM 进行吞吐率测试。同时，对 vLLM 框架稍加修改形成 vLLM_s，即内存管理器在 GPU 内存不足时固定调用张量交换技术。

图7展示了 12 个实验组在推理任务中的整体吞吐率测试结果，其横坐标为序列最大输出长度。表3给出了最大输出长度为 64 时，AdaptiveLLM 相对于 vLLM 和 vLLM_s 的具体加速比（左边数字为 vLLM，右边数字为 vLLM_s）。结果表明，和 vLLM 相比，AdaptiveLLM 实现了最高 1.40× 的整体吞吐加速；和 vLLM_s 相比，AdaptiveLLM 实现了最高 2.55× 的整体吞吐加速。

表 3 AdaptiveLLM 相对于基准框架的加速比

LLM-数据集	Chatbot	Alpaca	Summary
OPT-13B	1.38/2.46	1.36/2.55	1.15/2.50
OPT-30B	1.27/1.98	1.22/1.99	1.11/1.64
Llama-13B	1.28/2.28	1.40/2.37	1.17/2.12
Llama-32.5B	1.28/1.95	1.23/2.13	1.09/2.18

由于 Summary 数据集的平均序列长度和方差均明显高于 Alpaca 和 Chatbot 数据集，因此在相同条件下，其推理吞吐率低于 Alpaca 和 Chatbot。尽管如此，以 vLLM 作为基准框架时，AdaptiveLLM 在 Summary 数据集上也达到了 1.1× 加速比。

表4给出了序列最大输出长度为 64 时，不同框架推理过程中的抢占行为次数。

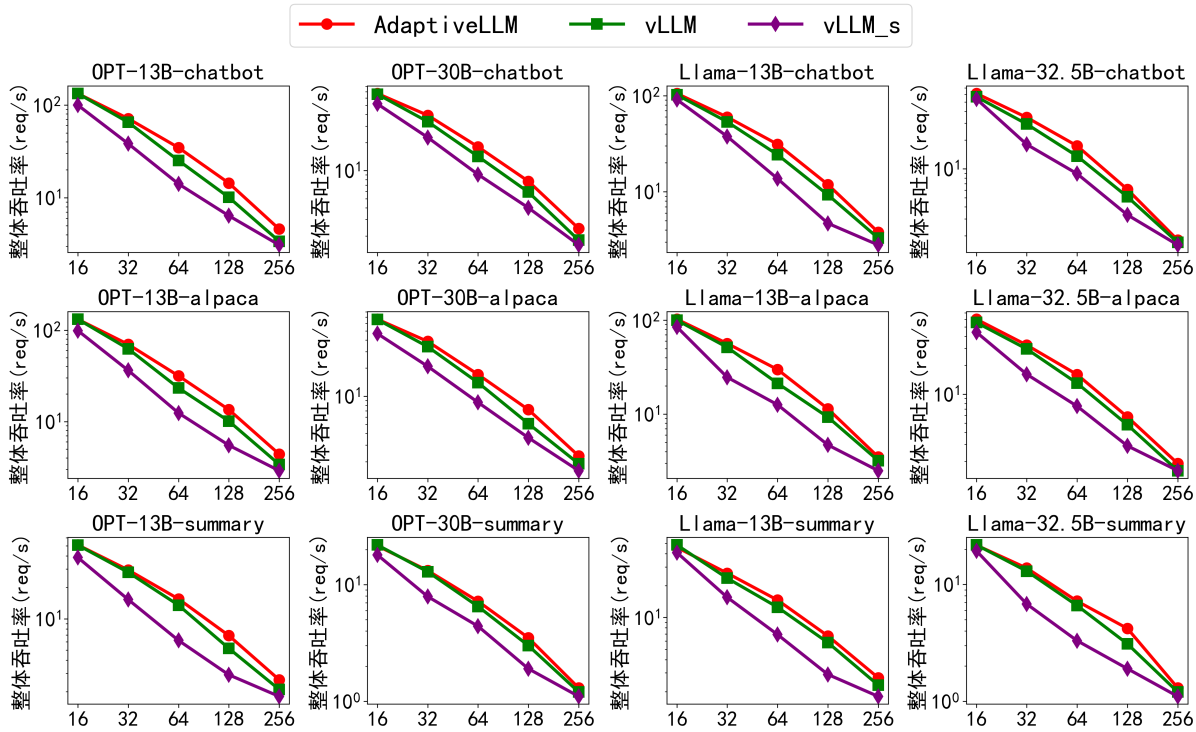

图 7 推理任务吞吐率

表 4 推理任务抢占行为记录

实验组	AdaptiveLLM		vLLM	vLLM_s
抢占行为（千次）	重算	交换	重算	交换
OPT-13B-chatbot	0.11	1.13	1.77	0.78
OPT-13B-alpaca	0.10	1.17	1.82	0.99
OPT-13B-summary	0.10	0.56	0.58	0.26
OPT-30B-chatbot	0.08	1.10	1.64	0.68
OPT-30B-alpaca	0.10	1.05	1.61	0.59
OPT-30B-summary	0.09	0.43	0.47	0.32
Llama-13B-chatbot	0.12	1.02	1.57	0.83
Llama-13B-alpaca	0.08	1.03	1.55	0.87
Llama-13B-summary	0.15	0.55	0.57	0.36
Llama-32.5B-chatbot	0.07	1.04	1.53	0.20
Llama-32.5B-alpaca	0.10	1.00	1.57	0.55

本文研究的推理场景使用贪心采样策略生成新 token。在 GPU 内存不足时，vLLM 调用张量重算技术，vLLM_s 调用张量交换技术，而 AdaptiveLLM 能够从二者中选择开销较小的内存优化方式。CPU 内存的限制使 vLLM_s 的批处理大小低于 vLLM 和 AdaptiveLLM，因此吞吐率也较低。当序列最大输出长度限制在较低水平时，每个请求执行推理任务所需的迭代次数较少，资源需求量低，抢占鲜有发生，此时 AdaptiveLLM 和 vLLM 的性能差距不大。随着最大输出长度的增加，有限的 GPU 内存无法满足需求，AdaptiveLLM 调用基于开销感知的内存优化策略，展现性能优势。当最大输出长度过大时，无论是 AdaptiveLLM 还是 vLLM，其批处理大小均限制在较低水平，但 AdaptiveLLM 仍具有明显优势（当序列最大输出长度为 256 时，AdaptiveLLM 在 vLLM 的基础上实现最高 1.3× 的加速比）。

在 Chatbot 和 Alpaca 数据集的推理任务中，序列长度较短，批处理大小高。根据预测器给出的结果可知，此时张量交换开销小于张量重算开销。然而随着新 token 的不断生成，大量用户请求需要被换出，导致 CPU 内存不足。因此 AdaptiveLLM 执行了大量张量交换操作和少量张量重算操作。

在 Summary 数据集的推理任务中，其序列长度较大，批处理大小低。张量交换发生频率较低，极少出现 CPU 内存不足的现象，此时张量重算操作的执行大部分来源于开销比较的结果。

综上所述，在基于开销感知的内存优化策略下，AdaptiveLLM 在 GPU 内存不足时预测张量交换与张量重算开销，并选择开销较小的内存优化技术执行，进而大幅度提升推理任务整体吞吐率。

4.4. 实时性测试

为了消除整体吞吐率变化对实时性测试的影响，本文选取平均带权周转时间作为测试指标。用户请求带权周转时间等于客户端响应时间除以服务器端处理时间，如公式5所示。对于某一用户请求来说，

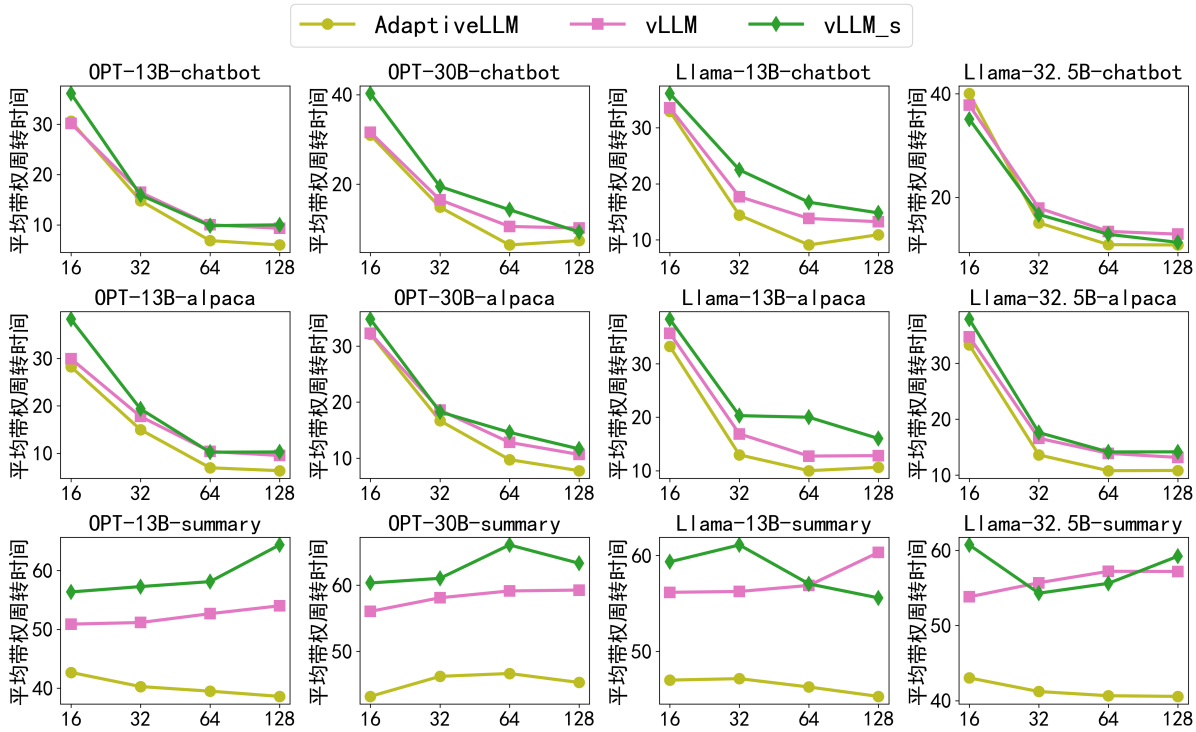

图 8 用户请求平均带权周转时间

$finish\_t$ 是其处理完毕时刻，$send\_t$ 是其从客户端发送至服务器端的时刻，$sche\_t$ 是其被 AdaptiveLLM 初次调度的时刻。平均带权周转时间越低，说明用户请求的排队时间越短。

$$w\_around\_t = \frac{finish\_t - send\_t}{finish\_t - sche\_t} \tag{5}$$

图8展示了平均带权周转时间随批处理大小上限的变化情况。在不同批处理大小设置下，基于公平性的用户请求调度策略均能使平均带权周转时间显著下降。当批处理大小较大时（64 或 128），AdaptiveLLM 的平均带权周转时间相比于 vLLM 下降了 20% 至 40%，相比于 vLLM_s 下降了 20% 至 60%。

对于序列较短的 Chatbot 和 Alpaca 数据集而言，随着批处理大小的上升，GPU 利用更加充分，因此平均带权周转时间下降。在实际运行中，当批处理大小到达 64 至 128 时，GPU 产生内存瓶颈，此时批处理大小无法继续提升，平均带权周转时间达到最小值。

对于序列较长的 Summary 数据集而言，其处理并发度被限制在较低水平（10 以下），无法达到用户设置的批处理大小上限。因此平均带权周转时间呈稳定状态。AdaptiveLLM 中高效的调度策略展现优势，使用户请求等待时间显著低于 vLLM 和 vLLM_s。

综上所述，基于公平性的用户请求调度策略使得用户请求从客户端发送至服务器端后能够很快开始处理，不会出现长时间等待现象。

4.5. 开销预测

(1) 张量重算预测误差

张量重算开销由张量重算分析器根据 LLM 层数、LLM 隐藏维度、单请求需要处理的 token 数量、以及批处理大小预测得到。表5和表6分别展示了 OPT 模型和 Llama 模型单步推理执行时间的预测效果。OPT 执行时间预测任务共有 6.4w 条训练数据和 1.6w 条测试数据，结果表明，随机森林回归模型性能最佳，其在拟合 2 次多项式时能够达到 1.76% 的预测误差。Llama 执行时间预测任务共有 6.8 条训练数据和 1.7w 条测试数据，结果表明，随机森林模型同样性能最佳，其在拟合 2 次多项式时能够达到 1.30% 的预测误差。

表 5 OPT 模型单步迭代执行时间预测误差

模型-拟合次数	1	2	3	4	5
线性回归模型	46.52	46.65	28.75	11.86	9.32
决策树	1.81	1.81	1.81	1.81	1.81
随机森林	1.77	1.76	1.77	1.77	1.78
岭回归模型	46.52	46.37	28.45	11.51	7.36
lasso 回归模型	40.22	25.53	27.38	26.08	25.49
弹性回归模型	111.89	123.62	91.67	87.59	86.48
梯度提升模型	15.57	16.05	14.80	15.09	14.68
KNN 回归模型	2.55	2.80	2.89	3.00	3.05

表 6 LLama 模型单步迭代执行时间预测误差

模型-拟合次数	1	2	3	4	5
线性回归模型	76.41	69.44	39.61	12.91	9.18
决策树	1.33	1.32	1.33	1.33	1.34
随机森林	1.31	1.30	1.31	1.31	1.31
岭回归模型	76.41	69.01	39.18	12.73	7.72
lasso 回归模型	69.23	33.57	34.42	35.16	31.58
弹性回归模型	127.18	139.7	100.18	94.94	93.51
梯度提升模型	22.42	21.97	19.42	19.99	19.38
KNN 回归模型	2.24	2.36	2.48	2.63	2.68

(2) 张量交换预测误差

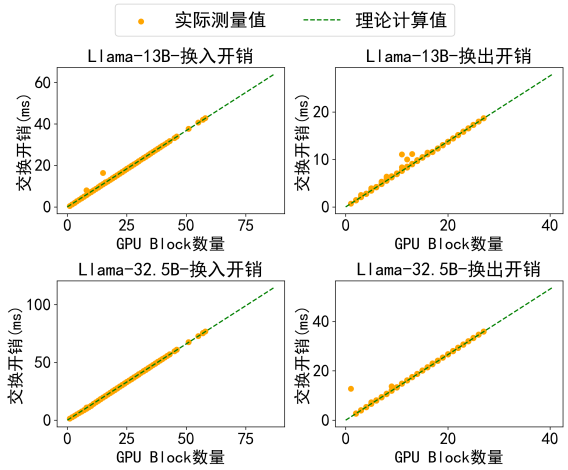

图 9 交换开销预测误差

张量交换预测开销由张量交换分析器根据用户请求 KV Cache 的内存占用和 GPU-CPU 双向传输带宽而计算得到。本文针对模型 Llama-13B 和 Llama-32.5B 进行测试，其结果如图9所示。两个模型换入开销预测 MAPE 误差分别为 1.5% 和 1.1%，换出开销预测 MAPE 误差分别为 1.0% 和 1.2%。因此，张量交换开销总预测误差低于 4%。

4.6. 其它测试

基于开销感知的内存优化策略在获取张量重算和张量交换开销时，会带来新的预测开销。本文设计如下对照实验获取预测过程的开销：

在吞吐率测试过程中，当 GPU 内存不足时调用开销比较过程，但最终使用 vLLM 提供的固定式内存优化策略（张量重算）。观察此情景下推理任务的总用时可知，预测开销在推理任务中仅占 0.1% 至 1%。

5. 相关工作

传统 LLM 推理框架采取了很多显存优化技术。

(1) 早期工作

Hugging Face Accelerate [?] 实现了张量交换技术，但换出与换入的张量仅限于 LLM 的参数张量。DeepSpeed ZeRO-Inference [?] 实现了 LLM 的分布式推理，能够利用数据并行性与张量并行性来实现 LLM 推理加速。但以上两个框架均无法针对 KV Cache 实现张量交换技术，也没有实现张量重算。

(2) 张量交换的提出

FlexGen [?] 首次提出了“自适应内存优化”的概念，并将张量交换的范围由参数张量扩展至所有张量。通过线性规划建模在交换方案的可行域内进行搜索，在给定的时间内找到较优解。同时，FlexGen 还实现了张量压缩技术，相比于 Hugging Face Accelerate 和 DeepSpeed ZeRO-Inference，实现了较大的吞吐率提升。然而，FlexGen 假设运行队列中的所有用户请求拥有相同的输出长度。在实际情况下，输出长度具有很大的差异性，使得相关理论无法推广。

(3) 调度粒度的转变

ORCA [?] 将批处理调度的粒度从单个用户请求转化为单次推理迭代，化解了用户请求相互等待的性能瓶颈。vLLM [?] 在 ORCA 的基础上实现了张量重算技术。基于 OS 页式内存管理思想，引入 Paged Attention 机制来实现。vLLM 相比于 OCRA，大幅度提升显存利用率，并增加批处理大小上限，进而提升推理任务的整体吞吐率。同时，vLLM 设计了规范且友好的用户接口，开发者能够根据任务需求来定义各种参数，极大地方便了有关推理优化的深入研究。

(4) 投机推理技术

SpecInfer [?] 引入了投机推理技术（Speculative Sampling），根据小型 LLM 的输出来预测大型 LLM 的输出，在大幅度提升推理吞吐率的同时保障了输出质量。DistillSpec [?] 在 SpecInfer 的基础上实现了知识蒸馏技术（Knowledge Distillation，KD），使得输出预测准确率显著提升。

6. 未来工作

在未来一段时间内，本文将针对以下部分完善 AdaptiveLLM 的设计。

(1) 张量压缩技术

有限的 PCIe 传输带宽使数据的 GPU-CPU 传输带来无法忽略的通信开销。随着 LLM 参数量和批处理大小的增加，通信开销成为主要性能瓶颈。近期工作中 [?]，张量压缩技术常与张量交换技术联合使用，通过矩阵变换等数学方式减少传输参数量。高效

的压缩技术能够在不损失张量精度的前提下减小通信开销，进一步提升批处理大小上限，提升吞吐率。

(2) 内存优化与前向传播的并行

- **张量交换与前向传播的并行**：张量交换的本质是 GPU-CPU 通信传输过程，而前向传播的本质是 GPU 计算过程。二者在传统模式下串行执行。AdaptiveLLM 计划在内存优化决策器中设计一个交换线程和一个计算线程，并行完成两项任务，进一步减少张量交换带来的额外开销。
- **张量重算与前向传播的并行**：SARATHI [**?**] 框架研发了 chunk-prefill 技术，实现 prefill 阶段与 decode 阶段共置运行。由于张量重算的本质是 prefill 过程，因此若将该技术移植到 AdaptiveLLM 中，可以实现张量重算与前向传播的并行。

(3) 张量并行与流水线并行 [**?**]

AdaptiveLLM 目前仅针对张量并行度与流水线并行度均为 1 的场景进行优化，本文将在未来实现张量并行技术与流水线并行技术。

7. 结论

本文设计了 AdaptiveLLM，一款基于张量交换和张量重算的 LLM 推理服务框架。AdaptiveLLM 实现了张量重算开销预测与张量交换开销预测，其预测误差分别在 2% 和 4% 以下。AdaptiveLLM 研发了基于开销感知的内存优化策略和基于公平性的用户请求调度策略。基于开销感知的内存优化策略在 GPU 内存不足时，执行开销较小的内存优化方式来保证推理任务的顺利完成。基于公平性的用户请求调度策略则能够在 GPU 内存充足时调度更多的用户请求。实验表明，以 vLLM 框架作为基准程序时，AdaptiveLLM 有 10%-40% 的整体吞吐率提升，实现面向服务器端的处理加速。同时能够以合理的方式调度用户请求，将平均带权周转时间降低 20%-40%，实现面向客户端的实时处理。综上所述，AdaptiveLLM 权衡整体吞吐率与单请求延时，化解二者在优化实现上的矛盾。