Efficient Memory Management for Large Language Model Serving with *PagedAttention*

Woosuk Kwon[1,*] Zhuohan Li[1,*] Siyuan Zhuang[1] Ying Sheng[1,2] Lianmin Zheng[1] Cody Hao Yu[3]
Joseph E. Gonzalez[1] Hao Zhang[4] Ion Stoica[1]
[1]UC Berkeley [2]Stanford University [3]Independent Researcher [4]UC San Diego

Abstract

High throughput serving of large language models (LLMs) requires batching sufficiently many requests at a time. However, existing systems struggle because the key-value cache (KV cache) memory for each request is huge and grows and shrinks dynamically. When managed inefficiently, this memory can be significantly wasted by fragmentation and redundant duplication, limiting the batch size. To address this problem, we propose PagedAttention, an attention algorithm inspired by the classical virtual memory and paging techniques in operating systems. On top of it, we build vLLM, an LLM serving system that achieves (1) near-zero waste in KV cache memory and (2) flexible sharing of KV cache within and across requests to further reduce memory usage. Our evaluations show that vLLM improves the throughput of popular LLMs by 2-4× with the same level of latency compared to the state-of-the-art systems, such as FasterTransformer and Orca. The improvement is more pronounced with longer sequences, larger models, and more complex decoding algorithms. vLLM's source code is publicly available at https://github.com/vllm-project/vllm.

1 Introduction

The emergence of large language models (*LLMs*) like GPT [5, 37] and PaLM [9] have enabled new applications such as programming assistants [6, 18] and universal chatbots [19, 35] that are starting to profoundly impact our work and daily routines. Many cloud companies [34, 44] are racing to provide these applications as hosted services. However, running these applications is very expensive, requiring a large number of hardware accelerators such as GPUs. According to recent estimates, processing an LLM request can be 10× more expensive than a traditional keyword query [43]. Given these high costs, increasing the throughput—and hence reducing the cost per request—of *LLM serving* systems is becoming more important.

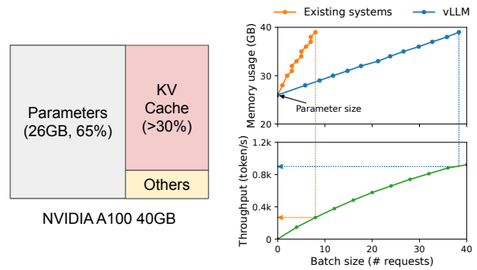


Figure 1. *Left:* Memory layout when serving an LLM with 13B parameters on NVIDIA A100. The parameters (gray) persist in GPU memory throughout serving. The memory for the KV cache (red) is (de)allocated per serving request. A small amount of memory (yellow) is used ephemerally for activation. *Right:* vLLM smooths out the rapid growth curve of KV cache memory seen in existing systems [31, 60], leading to a notable boost in serving throughput.

At the core of LLMs lies an autoregressive Transformer model [53]. This model generates words (tokens), *one at a time*, based on the input (prompt) and the previous sequence of the output's tokens it has generated so far. For each request, this expensive process is repeated until the model outputs a termination token. This sequential generation process makes the workload *memory-bound*, underutilizing the computation power of GPUs and limiting the serving throughput.

Improving the throughput is possible by batching multiple requests together. However, to process many requests in a batch, the memory space for each request should be efficiently managed. For example, Fig. 1 (left) illustrates the memory distribution for a 13B-parameter LLM on an NVIDIA A100 GPU with 40GB RAM. Approximately 65% of the memory is allocated for the model weights, which remain static during serving. Close to 30% of the memory is used to store the dynamic states of the requests. For Transformers, these states consist of the key and value tensors associated with the attention mechanism, commonly referred to as *KV cache* [41], which represent the context from earlier tokens to generate new output tokens in sequence. The remaining small

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10.
https://doi.org/10.1145/3600006.3613165

*Equal contribution.

面向大语言模型的高效内存管理：基于<样式 id='1'>分页注意力</样式>

吴肃坤[1]<样式 id='2'>，</样式>* 李柱浩[1]<样式 id='6'>，</样式>* 庄思源[1] 盛颖[1,2] 郑连民[1] 郝宇Cody[3]约瑟夫·E·冈萨雷斯[1] 张浩[4] 伊恩·斯托伊卡[1]加州大学伯克利分校 [2]斯坦福大学 [3]独立研究员 [4]加州大学圣地亚哥分校

摘要

大语言模型（LLM）的高吞吐量服务需要同时批处理足够多的请求。然而，现有系统面临挑战，因为每个请求的键值缓存（KV缓存）内存巨大且动态变化。如果管理不当，这种内存会因碎片化和冗余重复而显著浪费，从而限制批处理规模。为解决此问题，我们提出了分页注意力（PagedAttention）算法，该算法受操作系统中的经典虚拟内存和分页技术的启发。在此基础上，我们构建了vLLM系统，该系统实现了（1）KV缓存内存近乎零浪费和（2）请求内外的KV缓存灵活共享，以进一步降低内存使用。评估显示，与FasterTransformer和Orca等最先进系统相比，vLLM在相同延迟水平下将流行LLM的吞吐量提升了2-4× 倍。随着序列更长、模型更大、解码算法更复杂，改进效果越明显。vLLM的源代码已公开发布在https://github.com/vllm-project/vllm。

1 简介

大语言模型（大语言模型）如GPT [5,37] 和PaLM [9] 的出现，催生了新的应用，例如编程助手 [6, 18] 和通用聊天机器人 [19, 35]，这些应用开始深刻影响我们的工作和日常生活。许多云公司 [34, 44] 正竞相提供这些应用作为托管服务。然而，运行这些应用成本很高，需要大量硬件加速器，如GPU。据最新估计，处理一个LLM请求的成本可能 10× 高于传统关键词查询 [43]。鉴于这些高昂的成本，提高吞吐量——即降低每请求成本——对于大语言模型服务系统来说正变得越来越重要。

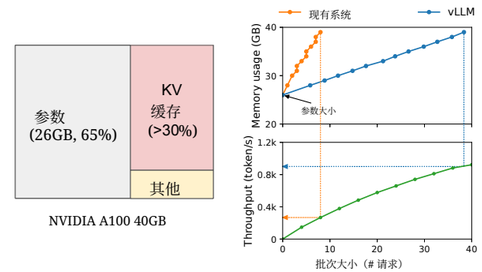


图1。 左： 为NVIDIA A100上服务一个具有13B参数的大语言模型时的内存布局。参数（灰色）在整个服务过程中持续存在于GPU内存中。KV缓存的内存（红色）根据服务请求进行（分）配。一小部分内存（黄色）用于临时激活。右： vLLM 使现有系统中 KV 缓存内存的快速增长曲线变得平滑 [31, 60]，从而显著提升了服务吞吐量。

LLMs的核心是一个自回归Transformer模型 [53]。该模型根据输入（提示）以及到目前为止它已经生成的输出的标记序列，逐个生成单词（标记）。对于每个请求，这个昂贵的过程会重复进行，直到模型输出一个终止标记。这种顺序生成过程使工作负载成为内存限制，未能充分利用GPU的计算能力，并限制了服务吞吐量。

通过将多个请求组合在一起，可以提高吞吐量。然而，为了批量处理许多请求，每个请求的内存空间应该得到高效管理。例如，图1（左）展示了在具有40GB RAM的NVIDIA A100 GPU上运行的13B参数大语言模型的内存分布情况。大约65%的内存用于分配模型权重，这些权重在服务过程中保持静态。将近30%的内存用于存储请求的动态状态。对于Transformer模型，这些状态由与注意力机制相关的键和值张量组成，通常被称为*KV缓存* [41]，它们表示从早期标记中获取上下文以生成序列中的新输出标记。剩余的小部分内存

所有者/作者。*SOSP '23，2023年10月23日至26日，科布伦茨，德国*© 2023 版权所有者/作者。ACM ISBN 979-8-4007-0229-7/23/10。
https://doi.org/10.1145/3600006.3613165

*平等贡献。

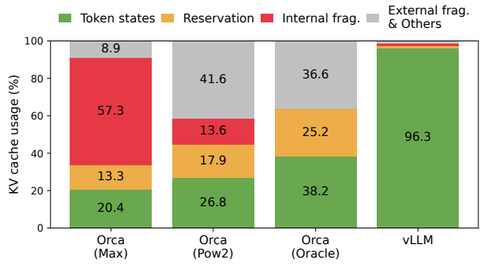


Figure 2. Average percentage of memory wastes in different LLM serving systems during the experiment in §6.2.

percentage of memory is used for other data, including activations – the ephemeral tensors created when evaluating the LLM. Since the model weights are constant and the activations only occupy a small fraction of the GPU memory, the way the KV cache is managed is critical in determining the maximum batch size. When managed inefficiently, the KV cache memory can significantly limit the batch size and consequently the throughput of the LLM, as illustrated in Fig. 1 (right).

In this paper, we observe that existing LLM serving systems [31, 60] fall short of managing the KV cache memory efficiently. This is mainly because they store the KV cache of a request in contiguous memory space, as most deep learning frameworks [33, 39] require tensors to be stored in contiguous memory. However, unlike the tensors in the traditional deep learning workloads, the KV cache has unique characteristics: it dynamically grows and shrinks over time as the model generates new tokens, and its lifetime and length are not known a priori. These characteristics make the existing systems' approach significantly inefficient in two ways:

First, the existing systems [31, 60] suffer from internal and external memory fragmentation. To store the KV cache of a request in contiguous space, they *pre-allocate* a contiguous chunk of memory with the request's maximum length (e.g., 2048 tokens). This can result in severe internal fragmentation, since the request's actual length can be much shorter than its maximum length (e.g., Fig. 11). Moreover, even if the actual length is known a priori, the pre-allocation is still inefficient: As the entire chunk is reserved during the request's lifetime, other shorter requests cannot utilize any part of the chunk that is currently unused. Besides, external memory fragmentation can also be significant, since the pre-allocated size can be different for each request. Indeed, our profiling results in Fig. 2 show that only 20.4% - 38.2% of the KV cache memory is used to store the actual token states in the existing systems.

Second, the existing systems cannot exploit the opportunities for memory sharing. LLM services often use advanced decoding algorithms, such as parallel sampling and beam search, that generate multiple outputs per request. In these scenarios, the request consists of multiple sequences that can partially share their KV cache. However, memory sharing is not possible in the existing systems because the KV cache of the sequences is stored in separate contiguous spaces.

To address the above limitations, we propose *PagedAttention*, an attention algorithm inspired by the operating system's (OS) solution to memory fragmentation and sharing: *virtual memory with paging*. PagedAttention divides the request's KV cache into blocks, each of which can contain the attention keys and values of a fixed number of tokens. In PagedAttention, the blocks for the KV cache are not necessarily stored in contiguous space. Therefore, we can manage the KV cache in a more flexible way as in OS's virtual memory: one can think of blocks as pages, tokens as bytes, and requests as processes. This design alleviates internal fragmentation by using relatively small blocks and allocating them on demand. Moreover, it eliminates external fragmentation as all blocks have the same size. Finally, it enables memory sharing at the granularity of a block, across the different sequences associated with the same request or even across the different requests.

In this work, we build *vLLM*, a high-throughput distributed LLM serving engine on top of PagedAttention that achieves near-zero waste in KV cache memory. vLLM uses block-level memory management and preemptive request scheduling that are co-designed with PagedAttention. vLLM supports popular LLMs such as GPT [5], OPT [62], and LLaMA [52] with varying sizes, including the ones exceeding the memory capacity of a single GPU. Our evaluations on various models and workloads show that vLLM improves the LLM serving throughput by 2-4× compared to the state-of-the-art systems [31, 60], without affecting the model accuracy at all. The improvements are more pronounced with longer sequences, larger models, and more complex decoding algorithms (§4.3). In summary, we make the following contributions:

- We identify the challenges in memory allocation in serving LLMs and quantify their impact on serving performance.
- We propose PagedAttention, an attention algorithm that operates on KV cache stored in non-contiguous paged memory, which is inspired by the virtual memory and paging in OS.
- We design and implement vLLM, a distributed LLM serving engine built on top of PagedAttention.
- We evaluate vLLM on various scenarios and demonstrate that it substantially outperforms the previous state-of-the-art solutions such as FasterTransformer [31] and Orca [60].

2 Background

In this section, we describe the generation and serving procedures of typical LLMs and the iteration-level scheduling used in LLM serving.

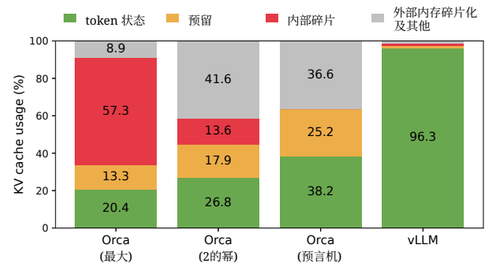


图2。 实验§6.2中不同大语言模型服务系统内存浪费的平均百分比。

用于其他数据，包括激活数据——在评估大语言模型时创建的临时张量。由于模型权重是常量，而激活数据仅占用GPU内存的一小部分，因此KV缓存的管理方式对于确定最大批量大小至关重要。当管理效率低下时，KV缓存内存会显著限制批量大小，进而影响大语言模型的吞吐量，如图1（右）所示。

在本文中，我们观察到现有的大语言模型服务系统 [31, 60] 在高效管理 KV 缓存内存方面存在不足。这主要是因为它们将请求的 KV 缓存存储在连续内存空间中，而大多数深度学习框架 [33, 39] 要求张量存储在连续内存中。然而，与传统深度学习工作负载中的张量不同，KV 缓存具有独特的特征：随着模型生成新的标记，它会在一段时间内动态增长和缩小，并且其生命周期和长度是事先未知的。这些特征使得现有系统的这种方法在两个方面存在显著低效性：

首先，现有系统 [31, 60] 存在内部和外部内存碎片化问题。为了将请求的 KV 缓存存储在连续空间中，它们 预先分配 了一块具有请求最大长度（例如，2048个标记）的连续内存块。这可能导致严重的内部碎片化，因为请求的实际长度可能远小于其最大长度（例如，图11）。此外，即使实际长度是事先已知的，预先分配仍然效率低下：在请求的生命周期内，整个块被保留，其他较短的请求无法利用当前未使用的任何部分。此外，外部内存碎片化也可能非常显著，因为每个请求的预分配大小可能不同。确实，我们图2中的分析结果表明，现有系统中只有 20.4% - 38.2% 的 KV 缓存内存用于存储实际 token 状态。

其次，现有系统无法利用内存共享的机会。大语言模型服务通常使用高级这些算法每次请求会生成多个输出。在这些场景中，请求包含多个可以部分共享其 KV 缓存的序列。然而，由于序列的 KV 缓存存储在不同的连续空间中，现有系统无法实现内存共享。

为解决上述限制，我们提出了分页注意力，一种受操作系统（OS）解决内存碎片化和共享方案启发的注意力算法：分页虚拟内存。分页注意力将请求的KV缓存划分为块，每个块可以包含固定数量标记的注意力键和值。在分页注意力中，KV缓存的块不一定存储在连续空间中。因此，我们可以像操作系统虚拟内存一样更灵活地管理KV缓存：可以将块视为页，标记视为字节，请求视为进程。这种设计通过使用相对较小的块并按需分配来缓解内部碎片化。此外，它消除了外部碎片化，因为所有块大小相同。最后，它实现了以块为粒度的内存共享，跨同一请求关联的不同序列，甚至跨不同请求。

在本工作中，我们构建了 *vLLM*，一个基于分页注意力的、高吞吐量的分布式LLM服务引擎，它在KV缓存内存上实现了近乎零的浪费。vLLM采用块级内存管理和预占式请求调度，这些设计与分页注意力协同设计。vLLM支持不同大小的流行LLM，如GPT [5],OPT [62], 和LLaMA [52]，包括那些超过单个GPU内存容量的模型。我们在各种模型和工作负载上的评估表明，与最先进的系统 [31, 60], 相比，vLLM将LLM服务吞吐量提高了2-4×，且完全不影响模型精度。随着序列更长、模型更大以及解码算法更复杂（§4.3），改进效果更为显著。总之，我们做出了以下贡献：

- 我们识别了在服务LLM时内存分配的挑战，并量化了它们对服务性能的影响。
- 我们提出了分页注意力，一种在非连续分页内存中存储的KV缓存上运行的注意力算法，该算法受操作系统中虚拟内存和分页的启发。
- 我们设计和实现了vLLM，一个基于分页注意力的分布式大语言模型服务引擎。
- 我们在各种场景下评估了vLLM，并证明其显著优于之前的最先进解决方案，例如FasterTransformer [31] 和Orca [60]。

2 背景

在本节中，我们描述了典型大语言模型的生成和服务流程，以及大语言模型服务中使用的迭代级调度。

2.1 Transformer-Based Large Language Models

The task of language modeling is to model the probability of a list of tokens $(x_1, \ldots, x_n)$. Since language has a natural sequential ordering, it is common to factorize the joint probability over the whole sequence as the product of conditional probabilities (a.k.a. *autoregressive decomposition* [3]):

$$P(x) = P(x_1) \cdot P(x_2 \mid x_1) \cdots P(x_n \mid x_1, \ldots, x_{n-1}). \quad (1)$$

Transformers [53] have become the de facto standard architecture for modeling the probability above at a large scale. The most important component of a Transformer-based language model is its *self-attention* layers. For an input hidden state sequence $(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$, a self-attention layer first applies linear transformations on each position i to get the query, key, and value vectors:

$$q_i = W_q x_i,\ k_i = W_k x_i,\ v_i = W_v x_i. \quad (2)$$

Then, the self-attention layer computes the attention score a_{ij} by multiplying the query vector at one position with all the key vectors before it and compute the output o_i as the weighted average over the value vectors:

$$a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^{i} \exp(q_i^\top k_t / \sqrt{d})},\ o_i = \sum_{j=1}^{i} a_{ij} v_j. \quad (3)$$

Besides the computation in Eq. 4, all other components in the Transformer model, including the embedding layer, feed-forward layer, layer normalization [2], residual connection [22], output logit computation, and the query, key, and value transformation in Eq. 2, are all applied independently position-wise in a form of $y_i = f(x_i)$.

2.2 LLM Service & Autoregressive Generation

Once trained, LLMs are often deployed as a conditional generation service (e.g., completion API [34] or chatbot [19, 35]). A request to an LLM service provides a list of *input prompt* tokens $(x_1, \ldots, x_n)$, and the LLM service generates a list of output tokens $(x_{n+1}, \ldots, x_{n+T})$ according to Eq. 1. We refer to the concatenation of the prompt and output lists as *sequence*.

Due to the decomposition in Eq. 1, the LLM can only sample and generate new tokens one by one, and the generation process of each new token depends on all the *previous tokens* in that sequence, specifically their key and value vectors. In this sequential generation process, the key and value vectors of existing tokens are often cached for generating future tokens, known as *KV cache*. Note that the KV cache of one token depends on all its previous tokens. This means that the KV cache of the same token appearing at different positions in a sequence will be different.

Given a request prompt, the generation computation in the LLM service can be decomposed into two phases:

The prompt phase takes the whole user prompt $(x_1, \ldots, x_n)$ as input and computes the probability of the first new token $P(x_{n+1} \mid x_1, \ldots, x_n)$. During this process, also generates the key vectors $k_1, \ldots, k_n$ and value vectors $v_1, \ldots, v_n$. Since prompt tokens $x_1, \ldots, x_n$ are all known, the computation of the prompt phase can be parallelized using matrix-matrix multiplication operations. Therefore, this phase can efficiently use the parallelism inherent in GPUs.

The autoregressive generation phase generates the remaining new tokens sequentially. At iteration t, the model takes one token x_{n+t} as input and computes the probability $P(x_{n+t+1} \mid x_1, \ldots, x_{n+t})$ with the key vectors $k_1, \ldots, k_{n+t}$ and value vectors $v_1, \ldots, v_{n+t}$. Note that the key and value vectors at positions 1 to $n + t - 1$ are cached at previous iterations, only the new key and value vector k_{n+t} and v_{n+t} are computed at this iteration. This phase completes either when the sequence reaches a maximum length (specified by users or limited by LLMs) or when an end-of-sequence (*<eos>*) token is emitted. The computation at different iterations cannot be parallelized due to the data dependency and often uses matrix-vector multiplication, which is less efficient. As a result, this phase severely underutilizes GPU computation and becomes memory-bound, being responsible for most portion of the latency of a single request.

2.3 Batching Techniques for LLMs

The compute utilization in serving LLMs can be improved by batching multiple requests. Because the requests share the same model weights, the overhead of moving weights is amortized across the requests in a batch, and can be overwhelmed by the computational overhead when the batch size is sufficiently large. However, batching the requests to an LLM service is non-trivial for two reasons. First, the requests may arrive at different times. A naive batching strategy would either make earlier requests wait for later ones or delay the incoming requests until earlier ones finish, leading to significant queueing delays. Second, the requests may have vastly different input and output lengths (Fig. 11). A straightforward batching technique would pad the inputs and outputs of the requests to equalize their lengths, wasting GPU computation and memory.

To address this problem, fine-grained batching mechanisms, such as cellular batching [16] and iteration-level scheduling [60], have been proposed. Unlike traditional methods that work at the request level, these techniques operate at the iteration level. After each iteration, completed requests are removed from the batch, and new ones are added. Therefore, a new request can be processed after waiting for a single iteration, not waiting for the entire batch to complete. Moreover, with special GPU kernels, these techniques eliminate the need to pad the inputs and outputs. By reducing the queueing delay and the inefficiencies from padding, the fine-grained batching mechanisms significantly increase the throughput of LLM serving.

2.1 基于Transformer的大语言模型

语言建模的任务是建模一系列标记（x_1，…，x_n）。由于语言具有自然的顺序性，通常将整个序列的联合概率分解为条件概率的乘积（即<code>自回归分解</code> [3]):

$$P(x) = P(x_1) \cdot P(x_2 \mid x_1) \cdots P(x_n \mid x_1, \ldots, x_{n-1}). \quad (1)$$

Transformer [53]已成为在大规模上建模概率的法定标准架构。基于Transformer的语言模型最重要的组件是其 自注意力 层。对于输入隐藏状态序列 $(x_1, \ldots, x_n) \in \mathbb{R}^{n \times d}$，自注意力层首先对每个位置 i 应用线性变换以获得查询向量、键向量和值向量：

$$q_i = W_q x_i,\ k_i = W_k x_i,\ v_i = W_v x_i. \quad (2)$$

然后，自注意力层通过将一个位置的查询向量与它之前的所有键向量相乘来计算注意力分数 a_{ij}，并将输出 o_i 计算为值向量的加权平均值：

$$a_{ij} = \frac{\exp(q_i^\top k_j / \sqrt{d})}{\sum_{t=1}^{i} \exp(q_i^\top k_t / \sqrt{d})},\ o_i = \sum_{j=1}^{i} a_{ij} v_j. \quad (3)$$

除了等式4中的计算外，Transformer模型中的所有其他组件，包括嵌入层、前馈层、层归一化 [2]，残差连接 [22]，输出逻辑计算，以及在等式2中的查询、键和值变换，都以 $y_i = f(x_i)$。的形式独立地逐位置应用。

2.2 大语言模型服务 & 自回归生成

训练完成后，大语言模型通常被部署为条件生成服务（例如，完成API [34] 或聊天机器人 [19, 35]）。对大语言模型服务的请求会提供一系列 输入提示 标记 $(x_1, \ldots, x_n)$，而大语言模型服务会根据公式1生成一系列输出标记 $(x_{n+1}, \ldots, x_{n+T})$。我们将提示和输出列表的连接称为 序列。

由于公式1中的分解，大语言模型只能逐个采样和生成新标记，并且每个新标记的生成过程依赖于该序列中所有 先前标记，具体是它们的键值向量。在此顺序生成过程中，现有标记的键值向量通常被缓存以生成后续标记，这被称为 *KV*缓存。请注意，一个标记的KV缓存依赖于该序列中所有其先前的标记。这意味着，在序列中不同位置出现的相同标记的KV缓存将是不同的。

给定一个请求提示，LLM 服务中的生成计算可以被分解为两个阶段：

提示阶段 以整个用户提示 $(x_1, \ldots, x_n)$ 作为输入，并计算第一个新标记 $P(x_{n+1} \mid x_1, \ldots, x_n)$ 的概率。在此过程中，还会生成键向量 $k_1, \ldots, k_n$ 和值向量 $v_1, \ldots, v_n$。由于提示标记 $x_1, \ldots, x_n$ 都是已知的，因此提示阶段的计算可以使用矩阵-矩阵乘法操作进行并行化。因此，此阶段可以有效地利用 GPU 中固有的并行性。

自回归生成阶段依次生成剩余的新标记。在迭代 t中，模型将一个标记 x_{n+t} 作为输入，并使用键向量 $k_1, \ldots, k_{n+t}$ 和值向量 $v_1, \ldots, v_{n+t}$ 计算概率 $P(x_{n+t+1} \mid x_1, \ldots, x_{n+t})$。请注意，位置 1 到 $n + t - 1$ 的键值向量在之前的迭代中已被缓存，本次迭代仅计算新的键向量 k_{n+t} 和值向量 v_{n+t}。此阶段完成时，序列达到最大长度（由用户指定或受大语言模型限制）或发出序列结束标记(*<eos>*)。由于数据依赖性，不同迭代的计算无法并行化，且通常使用矩阵向量乘法，效率较低。因此，此阶段严重浪费GPU计算资源，成为内存限制瓶颈，导致单次请求的延迟大部分由该阶段产生。

2.3 大语言模型的批次技术

在为大语言模型提供服务时，通过将多个请求批次处理可以提高计算利用率。由于请求共享相同的模型权重，权重传输的开销会在批次中的请求之间分摊，当批次大小足够大时，这种开销会被计算开销所淹没。然而，将请求批次处理给大语言模型服务具有两个非平凡的挑战。首先，请求可能在不同时间到达。一种简单的批次处理策略要么让先到的请求等待后到的请求，要么延迟新到的请求直到先到的请求完成，从而导致显著的排队延迟。其次，请求的输入和输出长度可能差异很大（图11）。一种直接的批次处理技术会填充请求的输入和输出以使它们的长度相等，从而浪费GPU计算和内存资源。

为解决此问题，已提出细粒度批处理机制，如细胞批处理 [16]和迭代级调度 [60]，。与传统的工作在请求级别的方法不同，这些技术工作在迭代级别。每次迭代后，已完成的请求会从批次中移除，并添加新的请求。因此，一个新的请求可以在等待一个迭代后进行处理，而不是等待整个批次完成。此外，通过特殊的 GPU 内核，这些技术消除了对输入和输出的填充需求。通过减少排队延迟和填充带来的低效性，细粒度批处理机制显著提高了大语言模型服务的吞吐量。

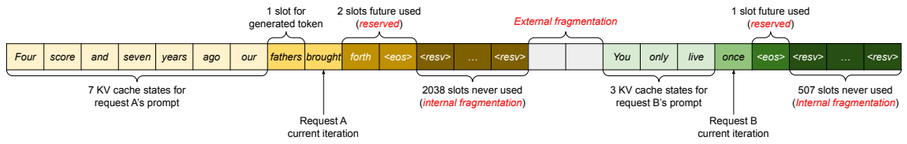


Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

3 Memory Challenges in LLM Serving

Although fine-grained batching reduces the waste of computing and enables requests to be batched in a more flexible way, the number of requests that can be batched together is still constrained by GPU memory capacity, particularly the space allocated to store the KV cache. In other words, the serving system's throughput is *memory-bound*. Overcoming this memory-bound requires addressing the following challenges in the memory management:

Large KV cache. The KV Cache size grows quickly with the number of requests. As an example, for the 13B parameter OPT model [62], the KV cache of a single token demands 800 KB of space, calculated as 2 (key and value vectors) × 5120 (hidden state size) × 40 (number of layers) × 2 (bytes per FP16). Since OPT can generate sequences up to 2048 tokens, the memory required to store the KV cache of one request can be as much as 1.6 GB. Concurrent GPUs have memory capacities in the tens of GBs. Even if all available memory was allocated to KV cache, only a few tens of requests could be accommodated. Moreover, inefficient memory management can further decrease the batch size, as shown in Fig. 2. Additionally, given the current trends, the GPU's computation speed grows faster than the memory capacity [17]. For example, from NVIDIA A100 to H100, The FLOPS increases by more than 2x, but the GPU memory stays at 80GB maximum. Therefore, we believe the memory will become an increasingly significant bottleneck.

Complex decoding algorithms. LLM services offer a range of decoding algorithms for users to select from, each with varying implications for memory management complexity. For example, when users request multiple random samples from a single input prompt, a typical use case in program suggestion [18], the KV cache of the prompt part, which accounts for 12% of the total KV cache memory in our experiment (§6.3), can be shared to minimize memory usage. On the other hand, the KV cache during the autoregressive generation phase should remain unshared due to the different sample results and their dependence on context and position. The extent of KV cache sharing depends on the specific decoding algorithm employed. In more sophisticated algorithms like beam search [49], different request beams can share larger portions (up to 55% memory saving, see §6.3) of their KV cache, and the sharing pattern evolves as the decoding process advances.

Scheduling for unknown input & output lengths. The requests to an LLM service exhibit variability in their input and output lengths. This requires the memory management system to accommodate a wide range of prompt lengths. In addition, as the output length of a request grows at decoding, the memory required for its KV cache also expands and may exhaust available memory for incoming requests or ongoing generation for existing prompts. The system needs to make scheduling decisions, such as deleting or swapping out the KV cache of some requests from GPU memory.

3.1 Memory Management in Existing Systems

Since most operators in current deep learning frameworks [33, 39] require tensors to be stored in contiguous memory, previous LLM serving systems [31, 60] also store the KV cache of one request as a contiguous tensor across the different positions. Due to the unpredictable output lengths from the LLM, they statically allocate a chunk of memory for a request based on the request's maximum possible sequence length, irrespective of the actual input or eventual output length of the request.

Fig. 3 illustrates two requests: request A with 2048 maximum possible sequence length and request B with a maximum of 512. The chunk pre-allocation scheme in existing systems has three primary sources of memory wastes: *reserved* slots for future tokens, *internal fragmentation* due to over-provisioning for potential maximum sequence lengths, and *external fragmentation* from the memory allocator like the buddy allocator. The external fragmentation will never be used for generated tokens, which is known before serving a request. Internal fragmentation also remains unused, but this is only realized after a request has finished sampling. They are both pure memory waste. Although the reserved memory is eventually used, reserving this space for the entire request's duration, especially when the reserved space is large, occupies the space that could otherwise be used to process other requests. We visualize the average percentage of memory wastes in our experiments in Fig. 2, revealing that the actual effective memory in previous systems can be as low as 20.4%.

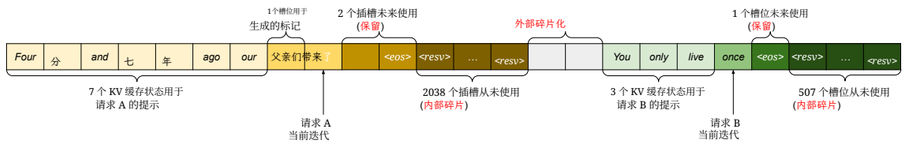


图3。 现有系统中KV缓存内存管理。存在三种类型的内存浪费——预留、内部碎片和外部碎片——这些浪费阻止其他请求装入内存。每个内存槽位中的标记代表其KV缓存。注意，相同标记在不同位置时可以有不同的KV缓存。

大语言模型服务中的3个内存挑战

尽管细粒度批处理减少了计算的浪费，并使请求能够更灵活地进行批处理，但可以一起批处理的请求数量仍然受到GPU内存容量的限制，特别是分配给存储KV缓存的空间。换句话说，服务器的吞吐量是内存限制。克服这种内存限制需要解决内存管理中的以下挑战：

大KV缓存。 随着请求数量的增加，KV缓存大小迅速增长。例如，对于13B参数的OPT模型 [62]，，单个标记的KV缓存需要800 KB的空间，计算方式为2（键值向量）× 5120（隐藏状态大小）× 40 （层数）× 2 （FP16每层的字节数）。由于OPT可以生成长达2048标记的序列，因此存储一个请求的KV缓存所需的内存可能高达1.6GB。并发GPU的内存容量在几十GB之间。即使所有可用内存都分配给KV缓存，也只能容纳几十个请求。此外，低效的内存管理会进一步降低批次大小，如图2所示。此外，鉴于当前趋势，GPU的计算速度增长速度比内存容量 [17]更快。例如，从NVIDIA A100到H100，FLOPS增加了2倍以上，但GPU内存最大仍为80GB。因此，我们认为内存将成为一个日益重要的瓶颈。

复杂的解码算法。 大语言模型服务为用户提供多种解码算法可供选择，每种算法对内存管理复杂性的影响各不相同。例如，当用户从单个输入提示中请求多个随机样本时，这在程序建议 [18]，中是一个典型用例，提示部分的 KV 缓存可以被共享，在我们的实验中（§6.3），这部分缓存占总 KV 缓存内存的 12%，从而最小化内存使用。另一方面，自回归生成阶段的 KV 缓存应保持不共享，因为不同的样本结果及其对上下文和位置的依赖性不同。KV 缓存共享的程度取决于所采用的特定解码算法。在更复杂的算法如束搜索 [49]，中，不同的请求波束可以共享更大的部分（内存节省高达 55%，参见§6.3）的 KV 缓存，并且共享模式随着解码过程的推进而演变。

调度未知输入和输出长度。 对大语言模型服务的请求在输入和输出长度上表现出可变性。这要求内存管理系统能够适应广泛的提示长度。此外，随着请求的输出长度在解码时增长，其KV缓存的内存需求也会增加，并可能耗尽用于传入请求或现有提示的持续生成的可用内存。系统需要做出调度决策，例如从GPU内存中删除或交换出某些请求的KV缓存。

3.1 现有系统中的内存管理

由于当前深度学习框架中的大多数算子[33, 39]需要张量存储在连续内存中，之前的LLM服务系统 [31, 60] 也把一个请求的KV缓存作为连续张量存储在不同的位置。由于LLM的输出长度不可预测，它们基于请求的最大可能序列长度静态分配一块内存给请求，而不管请求的实际输入或最终输出长度。

图3展示了两个请求：请求A具有2048个最大可能的序列长度，请求B的最大值为512。现有系统中，块预分配方案存在三个主要的内存浪费来源：预留用于未来标记的槽位、由于为潜在最大序列长度过度配置而导致的内部碎片，以及来自内存分配器（如伙伴分配器）的外部碎片。外部碎片永远不会用于生成的标记，这些标记在服务请求之前就已知晓。内部碎片也未被使用，但只有在请求完成采样后才能意识到这一点。它们都是纯粹的内存浪费。尽管预留的内存最终会被使用，但为整个请求的持续时间预留此空间，尤其是在预留空间很大时，会占用本可用于处理其他请求的空间。我们在图2中可视化了实验中的内存浪费平均百分比，揭示出先前系统中实际有效内存可能低至20.4%。

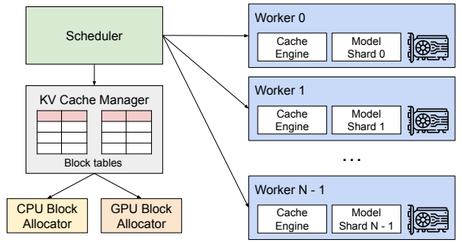


Figure 4. vLLM system overview.

Although compaction [54] has been proposed as a potential solution to fragmentation, performing compaction in a performance-sensitive LLM serving system is impractical due to the massive KV cache. Even with compaction, the pre-allocated chunk space for each request prevents memory sharing specific to decoding algorithms in existing memory management systems.

4 Method

In this work, we develop a new attention algorithm, *PagedAttention*, and build an LLM serving engine, *vLLM*, to tackle the challenges outlined in §3. The architecture of vLLM is shown in Fig. 4. vLLM adopts a centralized scheduler to coordinate the execution of distributed GPU workers. The *KV cache manager* effectively manages the KV cache in a paged fashion, enabled by PagedAttention. Specifically, the KV cache manager manages the physical KV cache memory on the GPU workers through the instructions sent by the centralized scheduler.

Next, We describe the PagedAttention algorithm in §4.1. With that, we show the design of the KV cache manager in §4.2 and how it facilitates PagedAttention in §4.3, respectively. Then, we show how this design facilitates effective memory management for various decoding methods (§4.4) and handles the variable length input and output sequences (§4.5). Finally, we show how the system design of vLLM works in a distributed setting (§4.6).

4.1 PagedAttention

To address the memory challenges in §3, we introduce *PagedAttention*, an attention algorithm inspired by the classic idea of *paging* [25] in operating systems. Unlike the traditional attention algorithms, PagedAttention allows storing continuous keys and values in non-contiguous memory space. Specifically, PagedAttention partitions the KV cache of each sequence into *KV blocks*. Each block contains the key and value vectors for a fixed number of tokens,[1] which we denote as *KV block size* (B). Denote the key block $K_j = (k_{(j-1)B+1}, \dots, k_{jB})$ and value block $V_j = (v_{(j-1)B+1}, \dots, v_{jB})$. The attention computation in Eq. 4 can be transformed into the following block-wise computation:

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t \mathbf{1} / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \qquad (4)$$

where $A_{ij} = (a_{i,(j-1)B+1}, \dots, a_{i,jB})$ is the row vector of attention score on j-th KV block.

During the attention computation, the PagedAttention kernel identifies and fetches different KV blocks separately. We show an example of PagedAttention in Fig. 5: The key and value vectors are spread across three blocks, and the three blocks are not contiguous on the physical memory. At each time, the kernel multiplies the query vector q_i of the query token ("*forth*") and the key vectors K_j in a block (e.g., key vectors of "*Four score and seven*" for block 0) to compute the attention score A_{ij}, and later multiplies A_{ij} with the value vectors V_j in a block to derive the final attention output o_i.

In summary, the PagedAttention algorithm allows the KV blocks to be stored in non-contiguous physical memory, which enables more flexible paged memory management in vLLM.

[1] In Transformer, each token has a set of key and value vectors across layers and attention heads within a layer. All the key and value vectors can be managed together within a single KV block, or the key and value vectors at different heads and layers can each have a separate block and be managed in separate block tables. The two designs have no performance difference and we choose the second one for easy implementation.

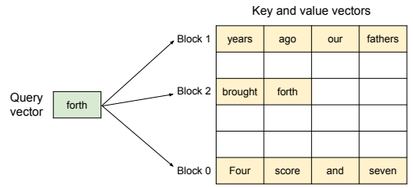


Figure 5. Illustration of the PagedAttention algorithm, where the attention key and values vectors are stored as non-contiguous blocks in the memory.

4.2 KV Cache Manager

The key idea behind vLLM's memory manager is analogous to the *virtual memory* [25] in operating systems. OS partitions memory into fixed-sized *pages* and maps user programs' logical pages to physical pages. Contiguous logical pages can correspond to non-contiguous physical memory pages, allowing user programs to access memory as though it were contiguous. Moreover, physical memory space needs not to be fully reserved in advance, enabling the OS to dynamically allocate physical pages as needed. vLLM uses the ideas behind virtual memory to manage the KV cache in an LLM service. Enabled by PagedAttention, we organize the KV cache as fixed-size KV blocks, like pages in virtual memory.

A request's KV cache is represented as a series of *logical KV blocks*, filled from left to right as new tokens and their KV cache are generated. The last KV block's unfilled positions are reserved for future generations. On GPU workers, a *block engine* allocates a contiguous chunk of GPU DRAM and

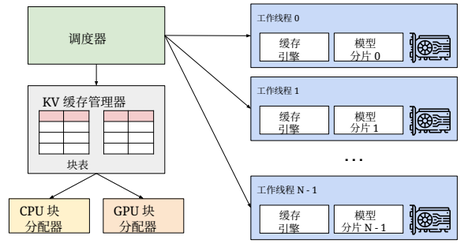


图4。 vLLM系统概述。

尽管紧缩 [54] 已被提出作为解决碎片化的潜在方案，但由于巨大的KV缓存，在性能敏感的大语言模型服务系统中执行紧缩是不切实际的。即使进行紧缩，每个请求预分配的块空间也阻止了现有内存管理系统中对解码算法的特定内存共享。

4 方法

在本工作中，我们开发了一种新的注意力算法，分页注意力，并构建了一个大语言模型服务引擎，*vLLM*，以应对 §3 中概述的挑战。vLLM 的架构如图 4 所示。vLLM 采用一个集中式调度器来协调分布式 GPU 工作者的执行。*KV 缓存管理器*有效地以分页方式管理 KV 缓存，这得益于分页注意力。具体来说，KV 缓存管理器通过集中式调度器发送的指令来管理 GPU 工作者上的物理 KV 缓存内存。

接下来，我们在 §4.1 中描述分页注意力算法。随后，我们在 §4.2 中展示 KV 缓存管理器的设计，并在 §4.3 中分别展示它如何促进分页注意力。然后，我们展示该设计如何促进各种解码方法的有效内存管理（§4.4）以及如何处理可变长度的输入和输出序列（§4.5）。最后，我们展示 vLLM 的系统设计在分布式环境下的工作原理（§4.6）。

4.1 分页注意力

为解决§3中提到的内存挑战，我们引入了<样式 id='1'>分页注意力</样式>，这是一种受操作系统经典<样式 id='3'>分页</样式> [25] 思想启发的注意力算法。与传统注意力算法不同，分页注意力允许在非连续的内存空间中存储连续的键和值。具体来说，分页注意力将每个序列的KV缓存划分为<样式 id='6'>KV块</样式>。每个块包含固定数量标记[1] 的键和值向量，我们将其表示为<样式 id='9'>KV</样式>

[1] 在 Transformer 中，每个 token 在层内和层内的注意力头之间拥有一组键值向量。所有键值向量可以统一管理在一个 KV 块内，或者不同头和层的键值向量可以各自拥有一个独立的块，并在各自的块表中管理。这两种设计没有性能差异，我们选择后者以方便实现。

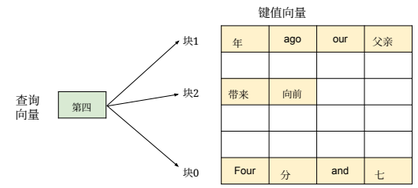


图 5。 分页注意力算法的示意图，其中注意力键和值向量以非连续块的形式存储在内存中。

<样式 id='1'>块大小</样式>(B)。记键块 $K_j = (k_{(j-1)B+1}$<样式 id='6'>，...，</样式> k_{jB})和值块 $V_j = (v$<样式 id='11'>(</样式>$_{j-1}$<样式 id='14'>)</样式>$_{B+1}$<样式 id='17'>，...，</样式> v_{jB})<样式 id='20'>。</样式>公式4中的注意力计算可以转换为以下分块计算：

$$A_{ij} = \frac{\exp(q_i^\top K_j / \sqrt{d})}{\sum_{t=1}^{\lceil i/B \rceil} \exp(q_i^\top K_t \mathbf{1} / \sqrt{d})}, \quad o_i = \sum_{j=1}^{\lceil i/B \rceil} V_j A_{ij}^\top, \qquad (4)$$

其中 $A_{ij} = (a_{i,(j-1)B+1}, \dots, a_{i,jB})$ 是第 j 个 KV 块上的注意力分向量。

在注意力计算过程中，分页注意力内核会分别识别和获取不同的KV块。我们在图5中展示了分页注意力的一个例子：键值向量分布在三个块中，并且这三个块在物理内存上不是连续的。每次，内核都会将查询标记（"*forth*"）的查询向量 q_i 与一个块中的键向量 K_j（例如块0的"*Four score and seven*"键向量）相乘，以计算注意力分数 A_{ij}，然后 later 将 A_{ij} 与一个块中的值向量 V_j 相乘，以导出最终的注意力输出 o_i.

总而言之，分页注意力算法允许KV块存储在非连续物理内存中，这使vLLM中的分页内存管理更加灵活。

4.2 KV 缓存管理器

vLLM内存管理器背后的关键思想类似于操作系统的虚拟内存 [25]。操作系统将内存划分为固定大小的页，并将用户程序的逻辑页映射到物理页。连续的逻辑页可以对应非连续的物理内存页，允许用户程序像访问连续内存一样访问内存。此外，物理内存空间无需预先完全保留，使操作系统能够根据需要动态分配物理页。vLLM使用虚拟内存背后的思想来管理LLM服务中的KV缓存。借助分页注意力，我们将KV缓存组织为固定大小的KV块，就像虚拟内存中的页一样。

一个请求的KV缓存表示为一系列<样式 id='1'>逻辑KV块</样式>，随着新标记及其KV缓存的生成，从左到右填充。最后一个KV块的未填充位置被保留用于未来的生成。在GPU工作者上，一个<样式 id='3'>块引擎</样式>分配一块连续的GPU显存，并将其

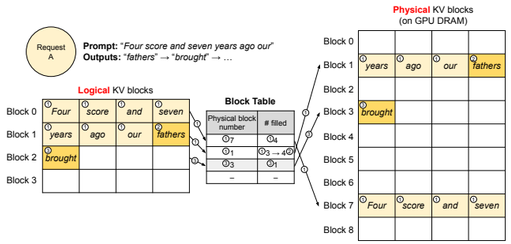


Figure 6. Block table translation in vLLM.

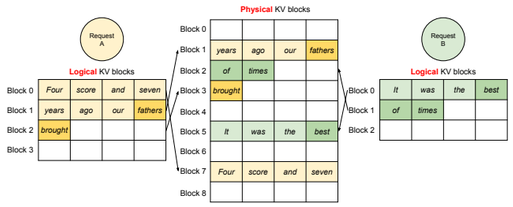


Figure 7. Storing the KV cache of two requests at the same time in vLLM.

divides it into *physical KV blocks* (this is also done on CPU RAM for swapping; see §4.5). The *KV block manager* also maintains *block tables*—the mapping between logical and physical KV blocks of each request. Each block table entry records the corresponding physical blocks of a logical block and the number of filled positions. Separating logical and physical KV blocks allows vLLM to dynamically grow the KV cache memory without reserving it for all positions in advance, which eliminates most memory waste in existing systems, as in Fig. 2.

4.3 Decoding with PagedAttention and vLLM

Next, we walk through an example, as in Fig. 6, to demonstrate how vLLM executes PagedAttention and manages the memory during the decoding process of a single input sequence: ① As in OS's virtual memory, vLLM does not require reserving the memory for the maximum possible generated sequence length initially. Instead, it reserves only the necessary KV blocks to accommodate the KV cache generated during prompt computation. In this case, The prompt has 7 tokens, so vLLM maps the first 2 logical KV blocks (0 and 1) to 2 physical KV blocks (7 and 1, respectively). In the prefill step, vLLM generates the KV cache of the prompts and the first output token with a conventional self-attention algorithm (e.g., [13]). vLLM then stores the KV cache of the first 4 tokens in logical block 0 and the following 3 tokens in logical block 1. The remaining slot is reserved for the subsequent autoregressive generation phase. ② In the first autoregressive decoding step, vLLM generates the new token with the PagedAttention algorithm on physical blocks 7 and 1. Since one slot remains available in the last logical block, the newly generated KV cache is stored there, and the block table's #filled record is updated. ③ At the second decoding step, as the last logical block is full, vLLM stores the newly generated KV cache in a new logical block; vLLM allocates a new physical block (physical block 3) for it and stores this mapping in the block table.

Globally, for each decoding iteration, vLLM first selects a set of candidate sequences for batching (more in §4.5), and allocates the physical blocks for the newly required logical blocks. Then, vLLM concatenates all the input tokens of the current iteration (i.e., all tokens for prompt phase requests and the latest tokens for generation phase requests) as one sequence and feeds it into the LLM. During LLM's computation, vLLM uses the PagedAttention kernel to access the previous KV cache stored in the form of logical KV blocks and saves the newly generated KV cache into the physical KV blocks. Storing multiple tokens within a KV block (block size > 1) enables the PagedAttention kernel to process the KV cache across more positions in parallel, thus increasing the hardware utilization and reducing latency. However, a larger block size also increases memory fragmentation. We study the effect of block size in §7.2.

Again, vLLM dynamically assigns new physical blocks to logical blocks as more tokens and their KV cache are generated. As all the blocks are filled from left to right and a new physical block is only allocated when all previous blocks are full, vLLM limits all the memory wastes for a request within one block, so it can effectively utilize all the memory, as shown in Fig. 2. This allows more requests to fit into memory for batching—hence improving the throughput. Once a request finishes its generation, its KV blocks can be freed to store the KV cache of other requests. In Fig. 7, we show an example of vLLM managing the memory for two sequences. The logical blocks of the two sequences are mapped to different physical blocks within the space reserved by the block engine in GPU workers. The neighboring logical blocks of both sequences do not need to be contiguous in physical GPU memory and the space of physical blocks can be effectively utilized by both sequences.

4.4 Application to Other Decoding Scenarios

§4.3 shows how PagedAttention and vLLM handle basic decoding algorithms, such as greedy decoding and sampling, that take one user prompt as input and generate a single output sequence. In many successful LLM applications [18, 34], an LLM service must offer more complex decoding scenarios that exhibit complex accessing patterns and more opportunities for memory sharing. We show the general applicability of vLLM on them in this section.

Parallel sampling. In LLM-based program assistants [6, 18], an LLM generates multiple sampled outputs for a single input prompt; users can choose a favorite output from various candidates. So far we have implicitly assumed that a request

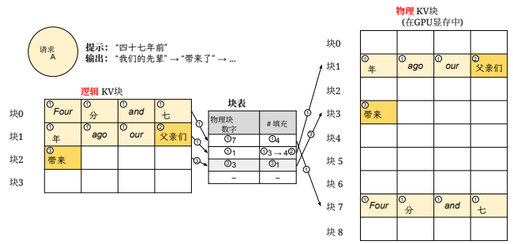


图6。 vLLM中的块表翻译。

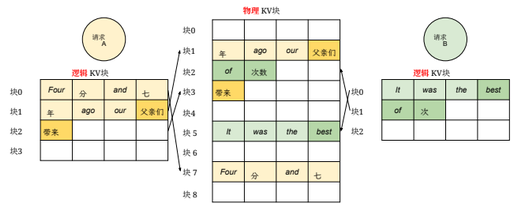


图7。 vLLM中同时存储两个请求的KV缓存。

划分为<样式 id='1'>物理KV块</样式>（这在CPU内存中为交换也执行；参见§4.5）。<样式 id='3'>KV块管理器</样式>还维护<样式 id='5'>块表</样式>——即每个请求的逻辑KV块与物理KV块之间的映射。每个块表条目记录逻辑块对应的物理块以及已填充位置的数量。分离逻辑和物理KV块允许vLLM动态扩展KV缓存内存，而无需预先为所有位置保留空间，从而消除了现有系统中大多数内存浪费，如图2所示。

4.3 使用分页注意力和vLLM进行解码

接下来，我们通过图6中的示例，演示vLLM如何执行分页注意力和在单个输入序列的解码过程中管理内存：① 与操作系统虚拟内存类似，vLLM最初不需要为最大可能的生成序列长度预留内存。相反，它仅预留必要的KV块来容纳提示计算期间生成的KV缓存。在这种情况下，提示有7个标记，因此vLLM将前2个逻辑KV块（0和1）映射到2个物理KV块（7和1，分别）。在预填充步骤中，vLLM使用传统的自回归算法（例如，[13]）生成提示和第一个输出标记的KV缓存。然后，vLLM将前4个标记的KV缓存存储在逻辑块0中，将后续3个标记存储在逻辑块1中。剩余的槽位为后续的自回归生成阶段保留。 ② 在第一个自回归解码步骤中，vLLM在物理块7和1上使用分页注意力算法生成新标记。由于最后一个逻辑块中还有一个槽位可用，新生成的KV缓存被存储在那里，并且块表的#filled记录被更新。 ③ 在第二步解码中，由于最后一个逻辑块已满，vLLM将新生成的KV缓存存储在一个新的逻辑块中；vLLM为其分配一个新的物理块（物理块3），并将此映射存储在块表中。

全球范围内，对于每次解码迭代，vLLM 首先选择一批候选序列进行批次处理（更多内容见 §4.5），并为新需要的逻辑块分配物理块。然后，vLLM 将当前迭代的所有输入标记（即提示阶段的所有标记）进行连接

请求和生成阶段最新标记请求）作为一条序列，并将其输入 LLM。在 LLM 的计算过程中，vLLM 使用分页注意力内核访问以逻辑 KV 块形式存储的前一个 KV 缓存，并将新生成的 KV 缓存保存到物理 KV 块中。在 KV 块（块大小 > 1）内存储多个标记，使分页注意力内核能够并行处理 KV 缓存中的更多位置，从而提高硬件利用率并减少延迟。然而，更大的块大小也会增加内存碎片化。我们研究块大小的影响，见 §7.2。

同样地，vLLM会动态地为逻辑块分配新的物理块，随着更多标记及其KV缓存被生成。由于所有块都是从左到右填满的，并且只有当所有前一个块都填满时才会分配新的物理块，vLLM将每个请求的内存浪费限制在一个块内，因此它可以有效利用所有内存，如图2所示。这允许更多请求被放入内存进行批次处理——从而提高吞吐量。当一个请求完成生成后，其KV块可以被释放，用于存储其他请求的KV缓存。在图7中，我们展示了vLLM管理两个序列内存的一个示例。两个序列的逻辑块被映射到GPU工作者中块引擎预留空间内的不同物理块。两个序列的相邻逻辑块在物理GPU内存中不需要连续，并且物理块的空间可以被两个序列有效利用。

4.4 应用到其他解码场景

§4.3 展示了分页注意力和vLLM如何处理基本的解码算法，例如贪婪解码和采样，这些算法以一个用户提示为输入并生成单个输出序列。在许多成功的LLM应用中 [18, 34],一个LLM服务必须提供更复杂的解码场景，这些场景表现出复杂的访问模式，并提供了更多内存共享的机会。我们在本节展示了vLLM在这些场景中的通用适用性。

并行采样。 InLLM-basedprogramassistants [6, 18],一个LLM为单个输入提示生成多个采样输出；用户可以从各种候选中选择最喜欢的输出。到目前为止，我们一直隐含地假设一个请求

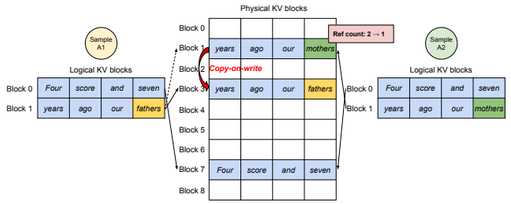


Figure 8. Parallel sampling example.

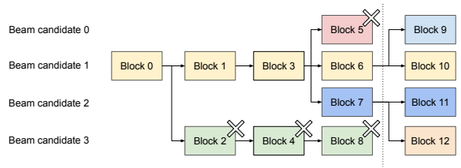


Figure 9. Beam search example.

generates a single sequence. In the remainder of this paper, we assume the more general case in which a request generates multiple sequences. In parallel sampling, one request includes multiple samples sharing the same input prompt, allowing the KV cache of the prompt to be shared as well. Via its PagedAttention and paged memory management, vLLM can realize this sharing easily and save memory.

Fig. 8 shows an example of parallel decoding for two outputs. Since both outputs share the same prompt, we only reserve space for one copy of the prompt's state at the prompt phase; the logical blocks for the prompts of both sequences are mapped to the same physical blocks: the logical block 0 and 1 of both sequences are mapped to physical blocks 7 and 1, respectively. Since a single physical block can be mapped to multiple logical blocks, we introduce a *reference count* for each physical block. In this case, the reference counts for physical blocks 7 and 1 are both 2. At the generation phase, the two outputs sample different output tokens and need separate storage for KV cache. vLLM implements a *copy-on-write* mechanism at the block granularity for the physical blocks that need modification by multiple sequences, similar to the copy-on-write technique in OS virtual memory (e.g., when forking a process). Specifically, in Fig. 8, when sample A1 needs to write to its last logical block (logical block 1), vLLM recognizes that the reference count of the corresponding physical block (physical block 1) is greater than 1; it allocates a new physical block (physical block 3), instructs the block engine to copy the information from physical block 1, and decreases the reference count to 1. Next, when sample A2 writes to physical block 1, the reference count is already reduced to 1; thus A2 directly writes its newly generated KV cache to physical block 1.

In summary, vLLM enables the sharing of most of the space used to store the prompts' KV cache across multiple output samples, with the exception of the final logical block, which is managed by a copy-on-write mechanism. By sharing physical blocks across multiple samples, memory usage can be greatly reduced, especially for *long input prompts*.

Beam search. In LLM tasks like machine translation [59], the users expect the top-k most appropriate translations output by the LLM. Beam search [49] is widely used to decode the most probable output sequence from an LLM, as it mitigates the computational complexity of fully traversing the sample space. The algorithm relies on the *beam width* parameter k, which determines the number of top candidates retained at every step. During decoding, beam search expands each candidate sequence in the beam by considering all possible tokens, computes their respective probabilities using the LLM, and retains the top-k most probable sequences out of $k \cdot |V|$ candidates, where $|V|$ is the vocabulary size.

Unlike parallel decoding, beam search facilities sharing not only the initial prompt blocks but also other blocks across different candidates, and the sharing patterns dynamically change as the decoding process advances, similar to the process tree in the OS created by compound forks. Fig. 9 shows how vLLM manages the KV blocks for a beam search example with $k = 4$. Prior to the iteration illustrated as the dotted line, each candidate sequence has used 4 full logical blocks. All beam candidates share the first block 0 (i.e., prompt). Candidate 3 digresses from others from the second block. Candidates 0-2 share the first 3 blocks and diverge at the fourth block. At subsequent iterations, the top-4 probable candidates all originate from candidates 1 and 2. As the original candidates 0 and 3 are no longer among the top candidates, their logical blocks are freed, and the reference counts of corresponding physical blocks are reduced. vLLM frees all physical blocks whose reference counts reach 0 (blocks 2, 4, 5, 8). Then, vLLM allocates new physical blocks (blocks 9-12) to store the new KV cache from the new candidates. Now, all candidates share blocks 0, 1, 3; candidates 0 and 1 share block 6, and candidates 2 and 3 further share block 7.

Previous LLM serving systems require frequent memory copies of the KV cache across the beam candidates. For example, in the case shown in Fig. 9, after the dotted line, candidate 3 would need to copy a large portion of candidate 2's KV cache to continue generation. This frequent memory copy overhead is significantly reduced by vLLM's physical block sharing. In vLLM, most blocks of different beam candidates can be shared. The copy-on-write mechanism is applied only when the newly generated tokens are within an old shared block, as in parallel decoding. This involves only copying one block of data.

Shared prefix. Commonly, the LLM user provides a (long) description of the task including instructions and example inputs and outputs, also known as *system prompt* [36]. The description is concatenated with the actual task input to form the prompt of the request. The LLM generates outputs based

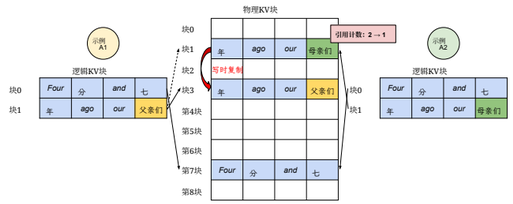


图8。 并行采样示例。

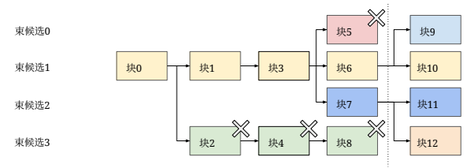


图9。 束搜索示例。

生成一个序列。在本文的其余部分，我们假设更一般的情况，即一个请求生成多个序列。在并行采样中，一个请求包含多个样本，它们共享相同的输入提示，因此提示的KV缓存也可以被共享。通过其分页注意力和分页内存管理，vLLM可以轻松实现这种共享并节省内存。

图8展示了对两个输出的并行解码示例。由于两个输出共享相同的提示，我们在提示阶段仅预留提示状态的一个副本的空间；两个序列的提示逻辑块映射到相同的物理块：两个序列的逻辑块0和1分别映射到物理块7和1。由于单个物理块可以映射到多个逻辑块，我们为每个物理块引入一个引用计数。在这种情况下，物理块7和1的引用计数均为2。在生成阶段，两个输出采样不同的输出标记，需要为KV缓存分配单独的存储空间。vLLM为需要被多个序列修改的物理块在块粒度上实现了写时复制机制，类似于操作系统虚拟内存中的写时复制技术（例如，在创建进程时）。具体来说，在图8中，当示例A1需要写入其最后一个逻辑块（逻辑块1）时，vLLM识别到相应物理块（物理块1）的引用计数大于1；它分配一个新的物理块（物理块3），指示块引擎从物理块1复制信息，并将引用计数减少到1。接下来，当示例A2写入物理块1时，引用计数已经减少到1；因此A2直接将其新生成的KV缓存写入物理块1。

总之，vLLM使得用于存储提示的KV缓存的大部分空间能够在多个输出样本之间共享，最后的逻辑块除外，它由写时复制机制管理。通过在多个样本之间共享物理块，内存使用可以大大减少，特别是对于长输入提示。

束搜索。 在大语言模型任务（如机器翻译 [59],）中，用户期望得到大语言模型输出的最-k 合适的翻译。束搜索 [49] 被广泛用于从大语言模型解码最可能的输出序列，因为它可以

门控计算复杂度完全遍历样本空间。

该算法依赖于<样式 id='1'>束宽度</样式>参数 k，该参数决定每一步保留的顶部候选数量。在解码过程中，束搜索通过考虑所有可能的标记扩展束中的每个候选序列，使用大语言模型计算它们的概率，并保留 $k \cdot |V|$ 个候选中概率最高的前k 个序列，其中 $|V|$ 是词汇量。

与并行解码不同，束搜索不仅支持跨不同候选共享初始提示块，还支持共享其他块，且共享模式会随着解码过程的推进动态变化，类似于操作系统通过复合分叉创建的过程树。图9展示了vLLM如何管理束搜索示例中的KV块（使用 $k = 4$）。在虚线所示迭代之前，每个候选序列已使用4个完整的逻辑块。所有束候选共享第一个块0（即提示）。候选3从第二个块开始偏离其他候选。候选0-2共享前3个块，并在第4个块处分叉。在后续迭代中，前4个概率最高的候选均源自候选1和2。由于原始候选0和3不再属于前候选，其逻辑块被释放，相应物理块的引用计数减少。vLLM释放所有引用计数为0的物理块（块2、4、5、8）。然后，vLLM为新的候选分配新的物理块（块9-12）以存储新的KV缓存。现在，所有候选共享块0、1、3；候选0和1共享块6，候选2和3进一步共享块7。

之前的LLM服务系统需要在束候选之间频繁复制KV缓存。例如，在图9所示的情况下，在虚线之后，候选3需要复制候选2的大量KV缓存才能继续生成。vLLM的物理块共享显著减少了这种频繁的内存复制开销。在vLLM中，不同束候选的大部分块可以共享。只有在新生成的标记位于旧共享块内时（如在并行解码中），才会应用写时复制机制。这仅涉及复制一个数据块。

共享前缀。 通常，LLM用户会提供一个（长）任务描述，包括指令和示例输入和输出，也称为系统提示 [36]。该描述与实际任务输入连接起来形成请求的提示。LLM根据

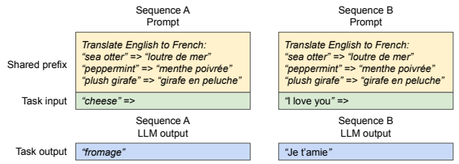


Figure 10. Shared prompt example for machine translation. The examples are adopted from [5].

on the full prompt. Fig. 10 shows an example. Moreover, the shared prefix can be further tuned, via prompt engineering, to improve the accuracy of the downstream tasks [26, 27].

For this type of application, many user prompts share a prefix, thus the LLM service provider can store the KV cache of the prefix in advance to reduce the redundant computation spent on the prefix. In vLLM, this can be conveniently achieved by reserving a set of physical blocks for a set of predefined shared prefixes by the LLM service provider, as how OS handles shared library across processes. A user input prompt with the shared prefix can simply map its logical blocks to the cached physical blocks (with the last block marked copy-on-write). The prompt phase computation only needs to execute on the user's task input.

Mixed decoding methods. The decoding methods discussed earlier exhibit diverse memory sharing and accessing patterns. Nonetheless, vLLM facilitates the simultaneous processing of requests with different decoding preferences, which existing systems *cannot* efficiently do. This is because vLLM conceals the complex memory sharing between different sequences via a common mapping layer that translates logical blocks to physical blocks. The LLM and its execution kernel only see a list of physical block IDs for each sequence and do not need to handle sharing patterns across sequences. Compared to existing systems, this approach broadens the batching opportunities for requests with different sampling requirements, ultimately increasing the system's overall throughput.

4.5 Scheduling and Preemption

When the request traffic surpasses the system's capacity, vLLM must prioritize a subset of requests. In vLLM, we adopt the first-come-first-serve (FCFS) scheduling policy for all requests, ensuring fairness and preventing starvation. When vLLM needs to preempt requests, it ensures that the earliest arrived requests are served first and the latest requests are preempted first.

LLM services face a unique challenge: the input prompts for an LLM can vary significantly in length, and the resulting output lengths are not known a priori, contingent on both the input prompt and the model. As the number of requests and their outputs grow, vLLM can run out of the GPU's physical blocks to store the newly generated KV cache. There are two classic questions that vLLM needs to answer in this context: (1) Which blocks should it evict? (2) How to recover evicted blocks if needed again? Typically, eviction policies use heuristics to predict which block will be accessed furthest in the future and evict that block. Since in our case we know that all blocks of a sequence are accessed together, we implement an all-or-nothing eviction policy, i.e., either evict all or none of the blocks of a sequence. Furthermore, multiple sequences within one request (e.g., beam candidates in one beam search request) are gang-scheduled as a *sequence group*. The sequences within one sequence group are always preempted or rescheduled together due to potential memory sharing across those sequences. To answer the second question of how to recover an evicted block, we consider two techniques:

Swapping. This is the classic technique used by most virtual memory implementations which copy the evicted pages to a swap space on the disk. In our case, we copy evicted blocks to the CPU memory. As shown in Fig. 4, besides the GPU block allocator, vLLM includes a CPU block allocator to manage the physical blocks swapped to CPU RAM. When vLLM exhausts free physical blocks for new tokens, it selects a set of sequences to evict and transfer their KV cache to the CPU. Once it preempts a sequence and evicts its blocks, vLLM stops accepting new requests until all preempted sequences are completed. Once a request completes, its blocks are freed from memory, and the blocks of a preempted sequence are brought back in to continue the processing of that sequence. Note that with this design, the number of blocks swapped to the CPU RAM never exceeds the number of total physical blocks in the GPU RAM, so the swap space on the CPU RAM is bounded by the GPU memory allocated for the KV cache.

Recomputation. In this case, we simply recompute the KV cache when the preempted sequences are rescheduled. Note that recomputation latency can be significantly lower than the original latency, as the tokens generated at decoding can be concatenated with the original user prompt as a new prompt—their KV cache at all positions can be generated in one prompt phase iteration.

The performances of swapping and recomputation depend on the bandwidth between CPU RAM and GPU memory and the computation power of the GPU. We examine the speeds of swapping and recomputation in §7.3.

4.6 Distributed Execution

Many LLMs have parameter sizes exceeding the capacity of a single GPU [5, 9]. Therefore, it is necessary to partition them across distributed GPUs and execute them in a model parallel fashion [28, 63]. This calls for a memory manager capable of handling distributed memory. vLLM is effective in distributed settings by supporting the widely used Megatron-LM style tensor model parallelism strategy on Transformers [47]. This strategy adheres to an SPMD (Single Program Multiple Data) execution schedule, wherein the linear layers are partitioned

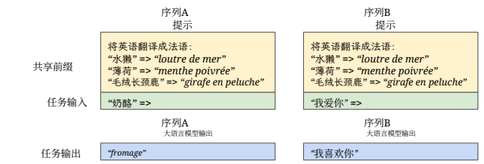


图10。 机器翻译共享提示示例。示例来自 [5]。

在完整提示上。图10显示了一个示例。此外，共享前缀可以通过提示工程进一步调整，以提高下游任务的准确性 [26, 27]。

对于这类应用，许多用户提示共享前缀，因此大语言模型服务提供者可以提前存储前缀的KV缓存，以减少在前缀上花费的冗余计算。在vLLM中，这可以通过大语言模型服务提供者为预定义的一组共享前缀预留一组物理块来实现，就像操作系统处理跨进程的共享库一样。带有共享前缀的用户输入提示可以将其逻辑块简单地映射到缓存的物理块（最后一个块标记为写时复制）。提示阶段的计算只需要在用户的任务输入上执行。

混合解码方法。 前面讨论的解码方法表现出不同的内存共享和访问模式。尽管如此，vLLM促进了具有不同解码偏好的请求的同时处理，而现有系统无法 高效地做到这一点。这是因为vLLM通过一个将逻辑块映射到物理块的通用映射层隐藏了不同序列之间复杂的内存共享。大语言模型及其执行内核只看到每个序列的一组物理块ID，并且不需要处理跨序列的共享模式。与现有系统相比，这种方法扩展了具有不同采样需求的请求的批处理机会，最终提高了系统的整体吞吐量。

4.5 调度与抢占

当请求流量超过系统容量时，vLLM 必须优先处理一部分请求。在 InvLLM 中，我们采用先来先服务（FCFS）的调度策略处理所有请求，确保公平性并防止饥饿。当 vLLM 需要抢占请求时，它确保最早到达的请求优先处理，而最新到达的请求优先被抢占。

大语言模型服务面临一个独特的挑战：大语言模型的输入提示长度可能差异很大，而输出长度则取决于输入提示和模型，且事先未知。随着请求数量和输出量的增长，vLLM 可能会耗尽 GPU 的物理块来存储新生成的 KV 缓存。vLLM 需要在这种情况下回答两个经典问题：

（1）应该驱逐哪些块？（2）如果需要再次使用，如何恢复被驱逐的块？通常，驱逐策略使用启发式方法来预测未来最可能被访问的块，并驱逐该块。由于在我们的情况下我们知道一个序列的所有块会一起被访问，我们实现了一种全有或全无的驱逐策略，即要么驱逐序列的所有块，要么一个都不驱逐。此外，一个请求内的多个序列（例如，一次束搜索请求中的束候选）被作为为一个序列组进行成组调度。由于这些序列之间可能存在内存共享，一个序列组内的序列总是会被一起抢占或重新调度。为了回答第二个问题，即如何恢复被驱逐的块，我们考虑了两种技术：

交换。 这是大多数虚拟内存实现使用的经典技术，它将驱逐的页复制到磁盘上的交换空间。在我们的案例中，我们将驱逐的块复制到 CPU 内存。如图 4 所示，除了 GPU 块分配器，vLLM 还包括一个 CPU 块分配器来管理被交换到 CPU RAM 的物理块。当 vLLM 耗尽用于新标记的空闲物理块时，它会选择一组序列进行驱逐，并将它们的 KV 缓存转移到 CPU。一旦它抢占一个序列并驱逐其块，vLLM 会停止接受新请求，直到所有被抢占的序列完成。当一个请求完成时，其块会被释放，而被抢占序列的块会重新加载到内存中以继续处理该序列。请注意，根据此设计，交换到 CPU RAM 的块数量永远不会超过 GPU RAM 中总物理块的数量，因此 CPU RAM 上的交换空间受分配给 KV 缓存的 GPU 内存限制。

重新计算。 在这种情况下，当被抢占的序列被重新调度时，我们只需重新计算KV缓存。请注意，重新计算延迟可以显著低于原始延迟，因为解码时生成的标记可以与原始用户提示连接起来作为一个新的提示——它们在所有位置上的KV缓存可以在一个提示阶段迭代中生成。

交换和重新计算的性能取决于CPU内存和GPU内存之间的带宽以及GPU的计算能力。我们在§7.3中考察了交换和重新计算的速率。

4.6 分布式执行

许多大语言模型的参数大小超过了单个GPU的容量 [5, 9]。因此，有必要将它们划分到分布式GPU上，并以模型并行方式执行 [28, 63]。这需要一个能够处理分布式内存的内存管理器。vLLM在分布式环境中效果不佳，因为它支持Transformer上广泛使用的Megatron-LM风格张量模型并行策略 [47]。该策略遵循SPMD（单程序多数据）执行调度，其中线性层被划分

Table 1. Model sizes and server configurations.

Model size	13B	66B	175B
GPUs	A100	4×A100	8×A100-80GB
Total GPU memory	40 GB	160 GB	640 GB
Parameter size	26 GB	132 GB	346 GB
Memory for KV cache	12 GB	21 GB	264 GB
Max. # KV cache slots	15.7K	9.7K	60.1K

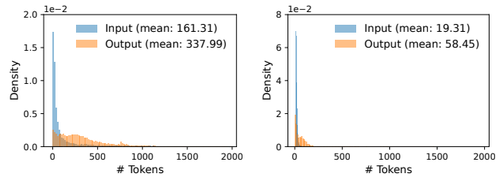
to perform block-wise matrix multiplication, and the the GPUs constantly synchronize intermediate results via an all-reduce operation. Specifically, the attention operator is split on the attention head dimension, each SPMD process takes care of a subset of attention heads in multi-head attention.

We observe that even with model parallel execution, each model shard still processes the same set of input tokens, thus requiring the KV Cache for the same positions. Therefore, vLLM features a single KV cache manager within the centralized scheduler, as in Fig. 4. Different GPU workers share the manager, as well as the mapping from logical blocks to physical blocks. This common mapping allows GPU workers to execute the model with the physical blocks provided by the scheduler for each input request. Although each GPU worker has the same physical block IDs, a worker only stores a portion of the KV cache for its corresponding attention heads.

In each step, the scheduler first prepares the message with input token IDs for each request in the batch, as well as the block table for each request. Next, the scheduler broadcasts this control message to the GPU workers. Then, the GPU workers start to execute the model with the input token IDs. In the attention layers, the GPU workers read the KV cache according to the block table in the control message. During execution, the GPU workers synchronize the intermediate results with the all-reduce communication primitive without the coordination of the scheduler, as in [47]. In the end, the GPU workers send the sampled tokens of this iteration back to the scheduler. In summary, GPU workers do not need to synchronize on memory management as they only need to receive all the memory management information at the beginning of each decoding iteration along with the step inputs.

5 Implementation

vLLM is an end-to-end serving system with a FastAPI [15] frontend and a GPU-based inference engine. The frontend extends the OpenAI API [34] interface, allowing users to customize sampling parameters for each request, such as the maximum sequence length and the beam width k. The vLLM engine is written in 8.5K lines of Python and 2K lines of C++/CUDA code. We develop control-related components including the scheduler and the block manager in Python while developing custom CUDA kernels for key operations such as PagedAttention. For the model executor, we implement popular LLMs such as GPT [5], OPT [62], and LLaMA [52] using



(a) ShareGPT **(b)** Alpaca

Figure 11. Input and output length distributions of the (a) ShareGPT and (b) Alpaca datasets.

PyTorch [39] and Transformers [58]. We use NCCL [32] for tensor communication across the distributed GPU workers.

5.1 Kernel-level Optimization

Since PagedAttention introduces memory access patterns that are not efficiently supported by existing systems, we develop several GPU kernels for optimizing it. (1) *Fused reshape and block write.* In every Transformer layer, the new KV cache are split into blocks, reshaped to a memory layout optimized for block read, then saved at positions specified by the block table. To minimize kernel launch overheads, we fuse them into a single kernel. (2) *Fusing block read and attention.* We adapt the attention kernel in FasterTransformer [31] to read KV cache according to the block table and perform attention operations on the fly. To ensure coalesced memory access, we assign a GPU warp to read each block. Moreover, we add support for variable sequence lengths within a request batch. (3) *Fused block copy.* Block copy operations, issued by the copy-on-write mechanism, may operate on discontinuous blocks. This can lead to numerous invocations of small data movements if we use the `cudaMemcpyAsync` API. To mitigate the overhead, we implement a kernel that batches the copy operations for different blocks into a single kernel launch.

5.2 Supporting Various Decoding Algorithms

vLLM implements various decoding algorithms using three key methods: `fork`, `append`, and `free`. The `fork` method creates a new sequence from an existing one. The `append` method appends a new token to the sequence. Finally, the `free` method deletes the sequence. For instance, in parallel sampling, vLLM creates multiple output sequences from the single input sequence using the `fork` method. It then adds new tokens to these sequences in every iteration with `append`, and deletes sequences that meet a stopping condition using `free`. The same strategy is also applied in beam search and prefix sharing by vLLM. We believe future decoding algorithms can also be supported by combining these methods.

6 Evaluation

In this section, we evaluate the performance of vLLM under a variety of workloads.

表1。 模型大小和服务器配置。

模型大小	13B	66B	175B
GPUs	A100	4×A100	8×A100-80GB
总GPU内存	40 GB	160 GB	640 GB
参数大小	26 GB	132 GB	346 GB
KV缓存内存	12 GB	21 GB	264 GB
最大 KV 缓存槽数量	15.7K	9.7K	60.1K

以执行块状矩阵乘法，并且GPU通过全归约操作不断同步中间结果。具体来说，注意力算子按注意力头维度进行划分，每个SPMD进程负责多头注意力中的一部分注意力头。

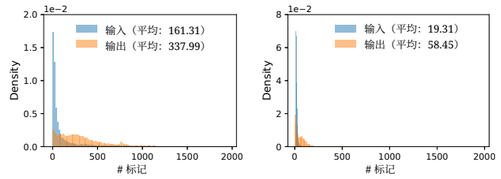
我们观察到，即使采用模型并行执行，每个模型分片仍然会处理相同的输入标记集，因此需要为相同的位置使用KV缓存。因此，vLLM在集中式调度器内部集成了一个KV缓存管理器，如图4所示。不同的GPU工作者共享该管理器，以及从逻辑块到物理块的映射。这种公共映射允许GPU工作者使用调度器为每个输入请求提供的物理块来执行模型。尽管每个GPU工作者具有相同的物理块ID，但每个工作者仅存储其对应注意力头的部分KV缓存。

在每个步骤中，调度器首先为批次中的每个请求准备包含输入标记ID的消息，以及每个请求的块表。接下来，调度器将此控制消息广播给GPU工作者。然后，GPU工作者开始使用输入标记ID执行模型。在注意力层中，GPU工作者根据控制消息中的块表读取KV缓存。在执行期间，GPU工作者在没有调度器协调的情况下，使用all-reduce通信原语同步中间结果，如图 [47]所示。最后，GPU工作者将本次迭代的采样标记发送回调度器。总之，GPU工作者不需要在内存管理上进行同步，因为他们只需要在每个解码迭代的开始时接收所有内存管理信息以及步骤输入。

5 实现

vLLM 是一个端到端的推理系统，拥有 FastAPI [15]前端和基于 GPU 的推理引擎。前端扩展了 OpenAI API [34]接口，允许用户为每个请求自定义采样参数，例如最大序列长度和束宽度 k。vLLM 引擎使用 8.5K 行 Python 和 2K 行 C++/CUDA 代码编写。我们开发控制相关组件在-

包括调度器和块管理器，在开发自定义CUDA内核以实现关键操作（如分页注意力）时使用Python。对于模型执行器，我们使用PyTorch [5],和Transformer [62],实现了GPT [5]、OPT [62],和LLaMA [52]等流行的大语言模型。



(a) ShareGPT **(b)** Alpaca

图11。 ShareGPT和Alpaca数据集的输入和输出长度分布。

我们使用NCCL [32]进行分布式GPU工作节点间的张量通信。

5.1 内核级优化

由于分页注意力引入的内存访问模式现有系统无法高效支持，我们开发了多个GPU内核来优化它。(1) 融合重形状和块写入。 在每个Transformer层中，新的KV缓存被分成块，重形状为针对块读取优化的内存布局，然后保存在块表指定的位置。为了最小化内核启动开销，我们将它们融合成一个内核。(2) 融合块读取和注意力。我们调整了 FasterTransformer [31] 中的注意力内核，使其根据块表读取KV缓存，并实时执行注意力操作。为了确保合并内存访问，我们为每个块分配一个GPU warp。此外，我们增加了对请求批次内可变序列长度的支持。(3) 融合块复制。 写时复制机制发出的块复制操作可能作用于不连续的块。如果我们使用 cudaMemcpyAsync API，这会导致大量小数据移动的调用。为了降低开销，我们实现了一个内核，将不同块的复制操作批量合并到一个内核启动中。

5.2 支持多种解码算法

vLLM 通过三种关键方法实现多种解码算法：fork、append 和 free。fork 方法从现有序列创建新序列。append 方法将新标记附加到序列。最后，free 方法删除序列。例如，在并行采样中，vLLM 使用 fork 方法从单个输入序列创建多个输出序列。然后，它使用 append 在每次迭代中向这些序列添加新标记，并使用 free 删除满足停止条件的序列。vLLM 在束搜索和前缀共享中也应用相同策略。我们认为未来解码算法也可以通过组合这些方法来支持。

6 评估

在本节中，我们评估了 vLLM 在各种工作负载下的性能。

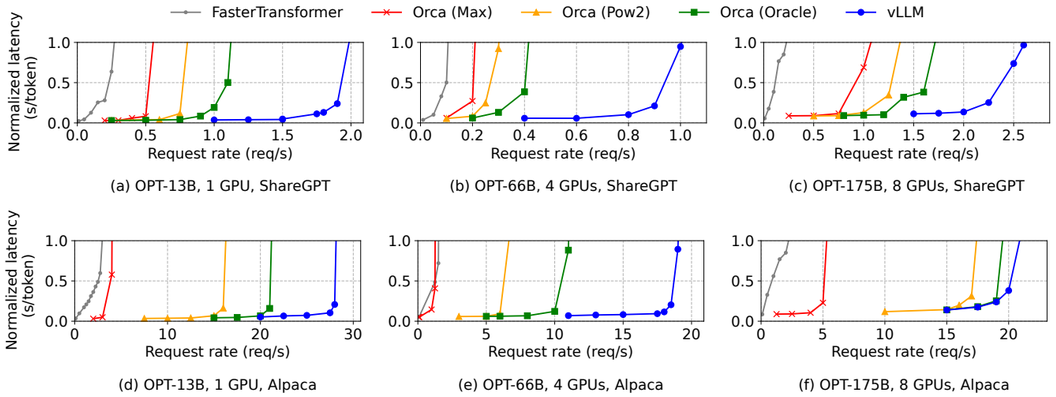


Figure 12. Single sequence generation with OPT models on the ShareGPT and Alpaca dataset

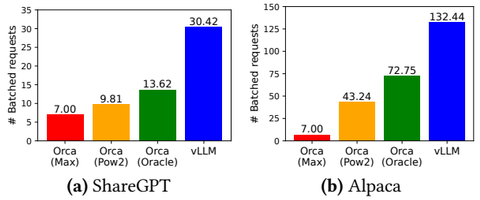


Figure 13. Average number of batched requests when serving OPT-13B for the ShareGPT (2 reqs/s) and Alpaca (30 reqs/s) traces.

6.1 Experimental Setup

Model and server configurations. We use OPT [62] models with 13B, 66B, and 175B parameters and LLaMA [52] with 13B parameters for our evaluation. 13B and 66B are popular sizes for LLMs as shown in an LLM leaderboard [38], while 175B is the size of the famous GPT-3 [5] model. For all of our experiments, we use A2 instances with NVIDIA A100 GPUs on Google Cloud Platform. The detailed model sizes and server configurations are shown in Table 1.

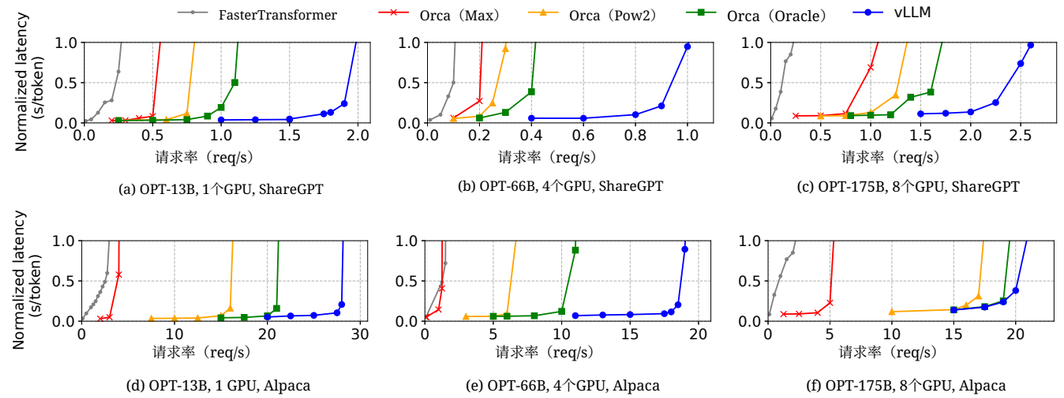
Workloads. We synthesize workloads based on ShareGPT [51] and Alpaca [50] datasets, which contain input and output texts of real LLM services. The ShareGPT dataset is a collection of user-shared conversations with ChatGPT [35]. The Alpaca dataset is an instruction dataset generated by GPT-3.5 with self-instruct [57]. We tokenize the datasets and use their input and output lengths to synthesize client requests. As shown in Fig. 11, the ShareGPT dataset has 8.4× longer input prompts and 5.8× longer outputs on average than the Alpaca dataset, with higher variance. Since these datasets do not include timestamps, we generate request arrival times using Poisson distribution with different request rates.

Baseline 1: FasterTransformer. FasterTransformer [31] is a distributed inference engine highly optimized for latency. As FasterTransformer does not have its own scheduler, we implement a custom scheduler with a dynamic batching mechanism similar to the existing serving systems such as Triton [30]. Specifically, we set a maximum batch size B as large as possible for each experiment, according to the GPU memory capacity. The scheduler takes up to B number of earliest arrived requests and sends the batch to FasterTransformer for processing.

Baseline 2: Orca. Orca [60] is a state-of-the-art LLM serving system optimized for throughput. Since Orca is not publicly available for use, we implement our own version of Orca. We assume Orca uses the buddy allocation algorithm to determine the memory address to store KV cache. We implement three versions of Orca based on how much it over-reserves the space for request outputs:

- **Orca (Oracle).** We assume the system has the knowledge of the lengths of the outputs that will be actually generated for the requests. This shows the upper-bound performance of Orca, which is infeasible to achieve in practice.
- **Orca (Pow2).** We assume the system over-reserves the space for outputs by at most 2×. For example, if the true output length is 25, it reserves 32 positions for outputs.
- **Orca (Max).** We assume the system always reserves the space up to the maximum sequence length of the model, i.e., 2048 tokens.

Key metrics. We focus on serving throughput. Specifically, using the workloads with different request rates, we measure *normalized latency* of the systems, the mean of every request's end-to-end latency divided by its output length, as in Orca [60]. A high-throughput serving system should retain low normalized latency against high request rates. For most experiments, we evaluate the systems with 1-hour traces. As an exception, we use 15-minute traces for the OPT-175B model due to the cost limit.



Fi图12。在ShareGPT和Alpaca数据集上使用OPT模型进行单序列生成

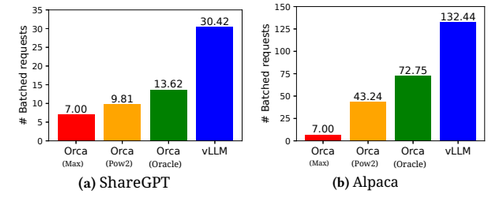


图13.在服务OPT-13B时，ShareGPT（2 reqs/s）和Alpaca（30 reqs/s）的 traces 的平均批次请求数量。

6.1 实验设置

模型和服务器配置。 我们使用 OPT [62] 模型，参数量分别为 13B、66B 和 175B，以及参数量为 13B 的 LLaMA [52] 进行评估。13B 和 66B 是 LLMs 中常见的规模，如 LLM 排行榜 [38], 所示，而 175B 是著名 GPT-3 [5] 模型的大小。在我们的所有实验中，我们使用 Google Cloud Platform 上的 A2 实例和 NVIDIA A100 GPU。详细的模型大小和服务器配置如表1所示。

工作负载。 我们基于ShareGPT [51]和Alpaca [50] 数据集合成工作负载，这些数据集包含真实大语言模型服务的输入和输出文本。ShareGPT数据集是用户共享的与ChatGPT [35]的对话集合。Alpaca数据集是由GPT-3.5使用self-instruct [57]生成的指令数据集。我们标记化这些数据集，并使用它们的输入和输出长度来合成客户端请求。如图11所示，ShareGPT数据集的平均输入提示比Alpaca数据集长8.4×，输出长5.8×，且方差更高。由于这些数据集不包含时间戳，我们使用具有不同请求率的泊松分布生成请求到达时间。

基准1：FasterTransformer。FasterTransformer [31]是一个针对延迟高度优化的分布式推理引擎。由于FasterTransformer没有自己的调度器，我们实现了一个具有动态批处理机制的定制调度器，该机制与现有的服务系统（如Triton [30]）相似。具体来说，我们根据GPU内存容量为每个实验设置尽可能大的最大批次大小 B。调度器最多处理 B 个最早到达的请求，并将批次发送给FasterTransformer进行处理。

基准2：Orca。 Orca [60]是一个针对吞吐量优化的最先进的大语言模型服务系统。由于Orca没有公开可用，我们实现了自己的Orca版本。我们假设Orca使用伙伴分配算法来确定存储KV缓存的内存地址。我们根据Orca为请求输出预留空间过度的程度，实现了三种Orca版本：

- **Orca（Oracle）**。 我们假设系统知道请求实际生成的输出长度。这显示了Orca的上限性能，但在实际中无法实现。
- **Orca（Pow2）**。 我们假设系统最多按 2×的倍数预留输出空间。例如，如果真实输出长度是25，它会为输出预留32个位置。
- **Orca（Max）**。 我们假设系统总是预留模型的最大序列长度空间，即2048个标记。

关键指标。 我们专注于吞吐量。具体来说，使用具有不同请求率的任务，我们测量系统的 归一化延迟，即每个请求端到端延迟的平均值除以其输出长度，如Orca [60]所示。高吞吐量服务系统应能在高请求率下保持低归一化延迟。对于大多数实验，我们使用1小时的跟踪记录进行评估。作为例外，由于成本限制，我们使用15分钟的跟踪记录评估OPT-175B模型。

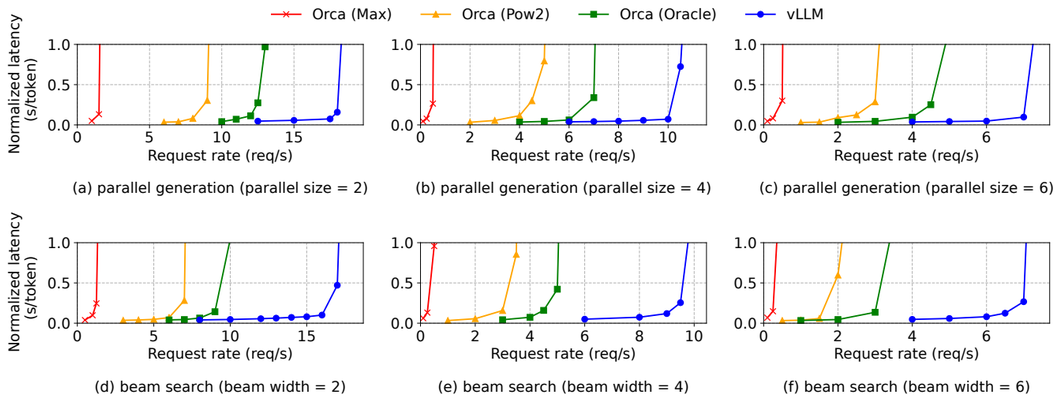


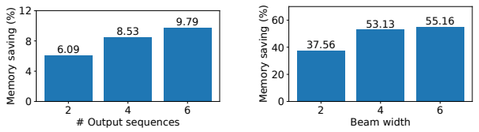
Figure 14. Parallel generation and beam search with OPT-13B on the Alpaca dataset.

6.2 Basic Sampling

We evaluate the performance of vLLM with basic sampling (one sample per request) on three models and two datasets. The first row of Fig. 12 shows the results on the ShareGPT dataset. The curves illustrate that as the request rate increases, the latency initially increases at a gradual pace but then suddenly explodes. This can be attributed to the fact that when the request rate surpasses the capacity of the serving system, the queue length continues to grow infinitely and so does the latency of the requests.

On the ShareGPT dataset, vLLM can sustain 1.7×–2.7× higher request rates compared to Orca (Oracle) and 2.7×–8× compared to Orca (Max), while maintaining similar latencies. This is because vLLM's PagedAttention can efficiently manage the memory usage and thus enable batching more requests than Orca. For example, as shown in Fig. 13a, for OPT-13B vLLM processes 2.2× more requests at the same time than Orca (Oracle) and 4.3× more requests than Orca (Max). Compared to FasterTransformer, vLLM can sustain up to 22× higher request rates, as FasterTransformer does not utilize a fine-grained scheduling mechanism and inefficiently manages the memory like Orca (Max).

The second row of Fig. 12 and Fig. 13b shows the results on the Alpaca dataset, which follows a similar trend to the ShareGPT dataset. One exception is Fig. 12 (f), where vLLM's advantage over Orca (Oracle) and Orca (Pow2) is less pronounced. This is because the model and server configuration for OPT-175B (Table 1) allows for large GPU memory space available to store KV cache, while the Alpaca dataset has short sequences. In this setup, Orca (Oracle) and Orca (Pow2) can also batch a large number of requests despite the inefficiencies in their memory management. As a result, the performance of the systems becomes compute-bound rather than memory-bound.



(a) Parallel sampling **(b)** Beam search

Figure 15. Average amount of memory saving from sharing KV blocks, when serving OPT-13B for the Alpaca trace.

6.3 Parallel Sampling and Beam Search

We evaluate the effectiveness of memory sharing in PagedAttention with two popular sampling methods: parallel sampling and beam search. In parallel sampling, all parallel sequences in a request can share the KV cache for the prompt. As shown in the first row of Fig. 14, with a larger number of sequences to sample, vLLM brings more improvement over the Orca baselines. Similarly, the second row of Fig. 14 shows the results for beam search with different beam widths. Since beam search allows for more sharing, vLLM demonstrates even greater performance benefits. The improvement of vLLM over Orca (Oracle) on OPT-13B and the Alpaca dataset goes from 1.3× in basic sampling to 2.3× in beam search with a width of 6.

Fig. 15 plots the amount of memory saving, computed by the number of blocks we saved by sharing divided by the number of total blocks without sharing. We show 6.1% - 9.8% memory saving on parallel sampling and 37.6% - 55.2% on beam search. In the same experiments with the ShareGPT dataset, we saw 16.2% - 30.5% memory saving on parallel sampling and 44.3% - 66.3% on beam search.

6.4 Shared prefix

We explore the effectiveness of vLLM for the case a prefix is shared among different input prompts, as illustrated in

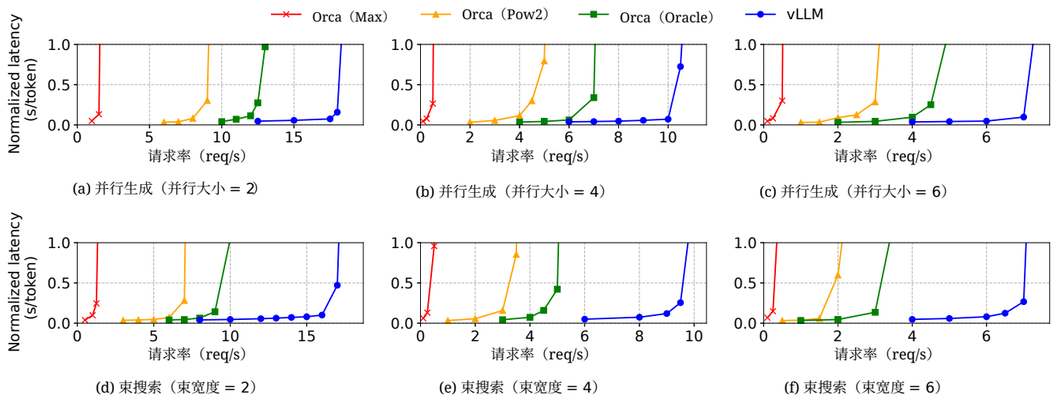


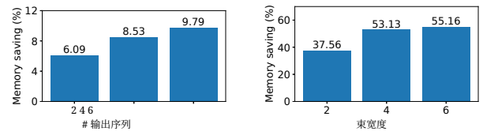
图14。 在Alpaca数据集上使用OPT-13B的并行生成和束搜索。

6.2 基本采样

我们评估了vLLM在三种模型和两个数据集上使用基本采样（每个请求一个样本）的性能。图12的第一行显示了在ShareGPT数据集上的结果。曲线表明，随着请求率的增加，延迟最初以缓慢的速度增加，然后突然爆炸。这可以归因于当请求率超过服务系统的容量时，队列长度会无限增长，请求的延迟也随之增加。

在ShareGPT数据集上，vLLM可以比Orca（Oracle）维持 1.7×–2.7×更高的请求率，比Orca（Max）维持 2.7×–8×更高的请求率，同时保持相似的延迟。这是因为vLLM的分页注意力可以有效地管理内存使用，从而能够处理比Orca更多的批次请求。例如，如图13a所示，对于OPT-13B，vLLM同时处理的请求比Orca（Oracle）多 2.2× ，比Orca（Max）多 4.3× 。与FasterTransformer相比，vLLM可以维持高达 22× 更高的请求率，因为FasterTransformer没有使用细粒度调度机制，并且像Orca（Max）一样低效地管理内存。

图12和图13b的第二行展示了在Alpaca数据集上的结果，该数据集与ShareGPT数据集的趋势相似。一个例外是图12(f)，其中vLLM相对于Orca（Oracle）和Orca（Pow2）的优势不那么明显。这是因为OPT-175B（表1）的模型和服务器配置允许使用较大的GPU内存空间来存储KV缓存，而Alpaca数据集的序列较短。在这种设置下，尽管Orca（Oracle）和Orca（Pow2）的内存管理效率不高，但它们也能批量处理大量请求。因此，系统的性能变得计算受限，而不是内存限制。



(a)并行采样 **(b)**束搜索

图15. 共享KV块时，在为Alpaca轨迹服务OPT-13B时的平均内存节省量。

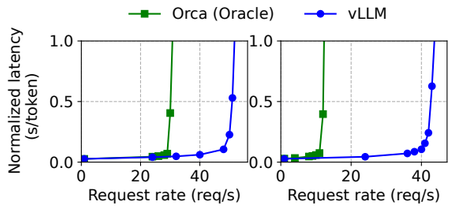
6.3 并行采样和束搜索

我们使用两种流行的采样方法——并行采样和束搜索——评估了PagedAttention中内存共享的有效性。在并行采样中，请求中的所有并行序列可以共享提示的KV缓存。如图14的第一行所示，随着采样序列数量的增加，vLLM相对于Orca基线带来了更多改进。同样，图14的第二行展示了不同束宽度下的束搜索结果。由于束搜索允许更多共享，vLLM表现出更大的性能优势。vLLM在OPT-13B和Alpaca数据集上相对于Orca（Oracle）的改进从基本采样的 1.3× 提升到束宽度为6时的 2.3× 。

图15绘制了内存节省量，计算方式为通过共享节省的块数除以不共享时的总块数。我们在并行采样上展示了6.1%-9.8%的内存节省，在束搜索上展示了37.6% - 55.2%。在相同的ShareGPT数据集实验中，我们在并行采样上看到了16.2% - 30.5%的内存节省，在束搜索上看到了44.3% - 66.3%。

6.4 共享前缀

我们探讨了vLLM在多个输入提示共享前缀情况下的有效性，如图所示



(a) 1-shot prefix prompt (b) 5-shot prefix prompt

Figure 16. Translation workload where the input prompts share a common prefix. The prefix includes (a) 1 example with 80 tokens or (b) 5 examples with 341 tokens.

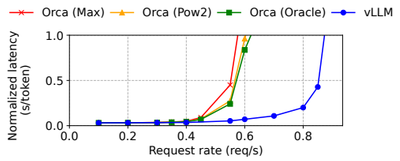


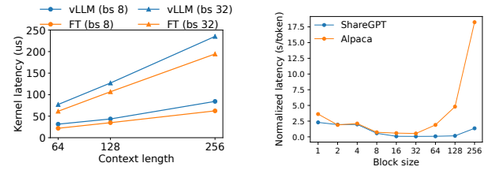
Figure 17. Performance on chatbot workload.

Fig. 10. For the model, we use LLaMA-13B [52], which is multilingual. For the workload, we use the WMT16 [4] English-to-German translation dataset and synthesize two prefixes that include an instruction and a few translation examples. The first prefix includes a single example (i.e., one-shot) while the other prefix includes 5 examples (i.e., few-shot). As shown in Fig. 16 (a), vLLM achieves 1.67× higher throughput than Orca (Oracle) when the one-shot prefix is shared. Furthermore, when more examples are shared (Fig. 16 (b)), vLLM achieves 3.58× higher throughput than Orca (Oracle).

6.5 Chatbot

A chatbot [8, 19, 35] is one of the most important applications of LLMs. To implement a chatbot, we let the model generate a response by concatenating the chatting history and the last user query into a prompt. We synthesize the chatting history and user query using the ShareGPT dataset. Due to the limited context length of the OPT-13B model, we cut the prompt to the last 1024 tokens and let the model generate at most 1024 tokens. We do not store the KV cache between different conversation rounds as doing this would occupy the space for other requests between the conversation rounds.

Fig. 17 shows that vLLM can sustain 2× higher request rates compared to the three Orca baselines. Since the ShareGPT dataset contains many long conversations, the input prompts for most requests have 1024 tokens. Due to the buddy allocation algorithm, the Orca baselines reserve the space for 1024 tokens for the request outputs, regardless of how they predict the output lengths. For this reason, the three Orca baselines behave similarly. In contrast, vLLM can effectively



(a) Latency of attention kernels. **(b)** End-to-end latency with different block sizes.

Figure 18. Ablation experiments.

handle the long prompts, as PagedAttention resolves the problem of memory fragmentation and reservation.

7 Ablation Studies

In this section, we study various aspects of vLLM and evaluate the design choices we make with ablation experiments.

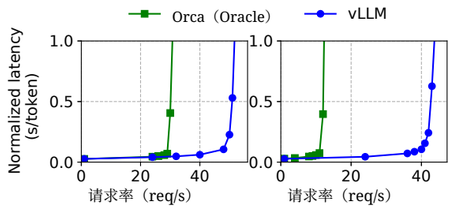
7.1 Kernel Microbenchmark

The dynamic block mapping in PagedAttention affects the performance of the GPU operations involving the stored KV cache, i.e., block read/writes and attention. Compared to the existing systems, our GPU kernels (§5) involve extra overheads of accessing the block table, executing extra branches, and handling variable sequence lengths. As shown in Fig. 18a, this leads to 20–26% higher attention kernel latency, compared to the highly-optimized FasterTransformer implementation. We believe the overhead is small as it only affects the attention operator but not the other operators in the model, such as Linear. Despite the overhead, PagedAttention makes vLLM significantly outperform FasterTransformer in end-to-end performance (§6).

7.2 Impact of Block Size

The choice of block size can have a substantial impact on the performance of vLLM. If the block size is too small, vLLM may not fully utilize the GPU's parallelism for reading and processing KV cache. If the block size is too large, internal fragmentation increases and the probability of sharing decreases.

In Fig. 18b, we evaluate the performance of vLLM with different block sizes, using the ShareGPT and Alpaca traces with basic sampling under fixed request rates. In the ShareGPT trace, block sizes from 16 to 128 lead to the best performance. In the Alpaca trace, while the block size 16 and 32 work well, larger block sizes significantly degrade the performance since the sequences become shorter than the block sizes. In practice, we find that the block size 16 is large enough to efficiently utilize the GPU and small enough to avoid significant internal fragmentation in most workloads. Accordingly, vLLM sets its default block size as 16.



(a) 1-shot前缀提示 (b) 5-shot前缀提示

图16。输入提示共享公共前缀的翻译工作负载。前缀包括 (a) 1个包含80个标记的示例或 (b) 5个包含341个标记的示例。

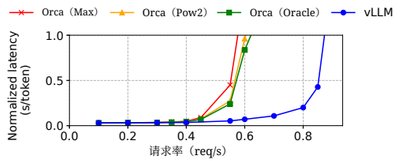


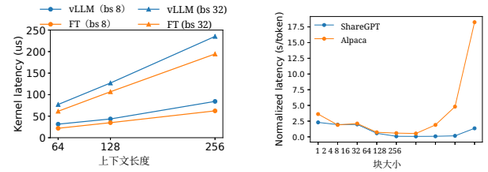
图17。聊天机器人工作负载的性能。

图10。对于模型，我们使用LLaMA-13B [52]，它支持多语言。对于工作负载，我们使用WMT16 [4] 英德翻译数据集，并合成两个包含指令和少量翻译示例的前缀。第一个前缀包含单个示例（即单样本），而另一个前缀包含5个示例（即少样本）。如图16 (a)所示，当单样本前缀被共享时，vLLM的吞吐量 1.67× 高于Orca（Oracle）。此外，当共享更多示例时（图16 (b)），vLLM的吞吐量 3.58×也高于Orca（Oracle）。

6.5 聊天机器人

聊天机器人 [8, 19, 35] 是大语言模型最重要的应用之一。为了实现聊天机器人，我们通过将聊天历史和最后一个用户查询连接成一个提示，让模型生成响应。我们使用ShareGPT数据集合成聊天历史和用户查询。由于OPT-13B模型的上下文长度有限，我们将提示截断到最后1024个标记，并让模型最多生成1024个标记。我们不存储不同对话轮次之间的KV缓存，因为这样做会占用对话轮次之间其他请求的空间。

图17显示，与三个Orca基线相比，vLLM能够维持 2× 更高的请求率。由于ShareGPT数据集中包含许多长对话，大多数请求的输入提示包含1024个标记。由于使用了伙伴分配算法，Orca基线为请求输出预留了1024个标记的空间，无论它们如何预测输出长度。因此，这三个Orca基线表现相似。相比之下，vLLM能够有效地



(a) 注意力核的延迟。 **(b)** 不同块大小的端到端延迟。不同的块大小。

图18。消融实验。

处理长提示，因为分页注意力解决了内存碎片化和预留的问题。

7 消融研究

在本节中，我们研究vLLM的各个方面，并通过消融实验评估我们做出的设计选择。

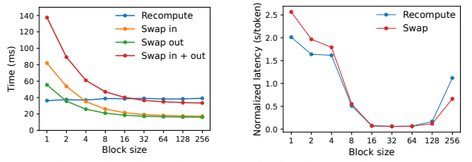
7.1 内核微基准测试

分页注意力中的动态块映射会影响GPU操作的性能，这些操作涉及存储的KV缓存，即块读写和注意力。与现有系统相比，我们的GPU内核（§5）在访问块表、执行额外分支和处理可变序列长度方面存在额外的开销。如图18a所示，这导致注意力内核延迟比高度优化的FasterTransformer实现高20–26%。我们认为开销很小，因为它只影响注意力算子，而不影响模型中的其他算子，例如Linear。尽管存在开销，分页注意力使vLLM在端到端性能（§6）方面显著优于FasterTransformer。

7.2 块大小的影响

块大小的选择会对vLLM的性能产生显著影响。如果块大小太小，vLLM可能无法充分利用GPU在读取和处理KV缓存时的并行性。如果块大小太大，内部碎片化会增加，共享的概率会降低。

在图18b中，我们使用ShareGPT和Alpaca追踪，在固定请求率下，评估vLLM在不同块大小下的性能。在ShareGPT追踪中，从16到128的块大小导致最佳性能。在Alpaca追踪中，虽然块大小16和32效果良好，但更大的块大小会显著降低性能，因为序列的长度会短于块大小。在实践中，我们发现块大小16足够有效地利用GPU，同时又足够小，以避免在大多数工作负载中发生显著的内部碎片。因此，vLLM将其默认块大小设置为16。



(a) Microbenchmark **(b)** End-to-end performance

Figure 19. (a) Overhead of recomputation and swapping for different block sizes. (b) Performance when serving OPT-13B with the ShareGPT traces at the same request rate.

7.3 Comparing Recomputation and Swapping

vLLM supports both recomputation and swapping as its recovery mechanisms. To understand the tradeoffs between the two methods, we evaluate their end-to-end performance and microbenchmark their overheads, as presented in Fig. 19. Our results reveal that swapping incurs excessive overhead with small block sizes. This is because small block sizes often result in numerous small data transfers between CPU and GPU, which limits the effective PCIe bandwidth. In contrast, the overhead of recomputation remains constant across different block sizes, as recomputation does not utilize the KV blocks. Thus, recomputation is more efficient when the block size is small, while swapping is more efficient when the block size is large, though recomputation overhead is never higher than 20% of swapping's latency. For medium block sizes from 16 to 64, the two methods exhibit comparable end-to-end performance.

8 Discussion

Applying the virtual memory and paging technique to other GPU workloads. The idea of virtual memory and paging is effective for managing the KV cache in LLM serving because the workload requires dynamic memory allocation (since the output length is not known a priori) and its performance is bound by the GPU memory capacity. However, this does not generally hold for every GPU workload. For example, in DNN training, the tensor shapes are typically static, and thus memory allocation can be optimized ahead of time. For another example, in serving DNNs that are not LLMs, an increase in memory efficiency may not result in any performance improvement since the performance is primarily compute-bound. In such scenarios, introducing the vLLM's techniques may rather degrade the performance due to the extra overhead of memory indirection and non-contiguous block memory. However, we would be excited to see vLLM's techniques being applied to other workloads with similar properties to LLM serving.

LLM-specific optimizations in applying virtual memory and paging. vLLM re-interprets and augments the idea of virtual memory and paging by leveraging the application-specific semantics. One example is vLLM's all-or-nothing swap-out policy, which exploits the fact that processing a request requires all of its corresponding token states to be stored in GPU memory. Another example is the recomputation method to recover the evicted blocks, which is not feasible in OS. Besides, vLLM mitigates the overhead of memory indirection in paging by fusing the GPU kernels for memory access operations with those for other operations such as attention.

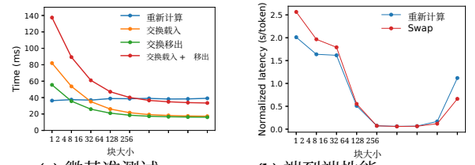
9 Related Work

General model serving systems. Model serving has been an active area of research in recent years, with numerous systems proposed to tackle diverse aspects of deep learning model deployment. Clipper [11], TensorFlow Serving [33], Nexus [45], InferLine [10], and Clockwork [20] are some earlier general model serving systems. They study batching, caching, placement, and scheduling for serving single or multiple models. More recently, DVABatch [12] introduces multi-entry multi-exit batching. REEF [21] and Shepherd [61] propose preemption for serving. AlpaServe [28] utilizes model parallelism for statistical multiplexing. However, these general systems fail to take into account the auto-regressive property and token state of LLM inference, resulting in missed opportunities for optimization.

Specialized serving systems for transformers. Due to the significance of the transformer architecture, numerous specialized serving systems for it have been developed. These systems utilize GPU kernel optimizations [1, 29, 31, 56], advanced batching mechanisms [14, 60], model parallelism [1, 41, 60], and parameter sharing [64] for efficient serving. Among them, Orca [60] is most relevant to our approach.

Comparison to Orca. The iteration-level scheduling in Orca [60] and PagedAttention in vLLM are complementary techniques: While both systems aim to increase the GPU utilization and hence the throughput of LLM serving, Orca achieves it by scheduling and interleaving the requests so that more requests can be processed in parallel, while vLLM is doing so by increasing memory utilization so that the working sets of more requests fit into memory. By reducing memory fragmentation and enabling sharing, vLLM runs more requests in a batch in parallel and achieves a 2-4× speedup compared to Orca. Indeed, the fine-grained scheduling and interleaving of the requests like in Orca makes memory management more challenging, making the techniques proposed in vLLM even more crucial.

Memory optimizations. The widening gap between the compute capability and memory capacity of accelerators has caused memory to become a bottleneck for both training and inference. Swapping [23, 42, 55], recomputation [7, 24] and their combination [40] have been utilized to reduce the peak memory of training. Notably, FlexGen [46] studies how to swap weights and token states for LLM inference with



(a) 微基准测试 **(b)** 端到端性能

图19.(a) 不同块大小的重新计算和交换开销。 (b) 在相同请求率下使用ShareGPT跟踪服务OPT-13B的性能。

7.3 比较重新计算与交换

vLLM 支持重新计算和交换作为其恢复机制。为了理解这两种方法的权衡，我们评估了它们的端到端性能，并对其开销进行了微基准测试，如图 19 所示。我们的结果表明，在小块大小的情况下，交换会产生过高的开销。这是因为小块大小通常会导致 CPU 和 GPU 之间进行大量的小数据传输，这限制了有效的 PCIe 带宽。相比之下，重新计算的开销在不同块大小下保持不变，因为重新计算不利用 KV 块。因此，当块大小较小时，重新计算更高效，而当块大小较大时，交换更高效，尽管重新计算的开销永远不会高于交换延迟的 20%。对于 16 到 64 的中等块大小，这两种方法表现出相当端到端性能。

8 讨论

将虚拟内存和分页技术应用于其他GPU工作负载。 虚拟内存和分页的概念对于管理LLM服务中的KV缓存是有效的，因为该工作负载需要动态内存分配（因为输出长度是事先未知的）并且其性能受限于GPU内存容量。然而，这并不适用于所有GPU工作负载。例如，在DNN训练中，张量形状通常是静态的，因此内存分配可以提前优化。再例如，在服务非LLM的DNN时，内存效率的提高可能不会带来任何性能提升，因为性能主要受计算受限。在这种情况下，引入vLLM的技术可能会由于内存间接寻址和非连续块内存的额外开销而降低性能。然而，我们很期待看到vLLM的技术被应用于具有与LLM服务相似特性的其他工作负载。

针对大语言模型的具体优化，应用于虚拟内存和分页。vLLM 通过利用特定应用的语义，重新诠释并增强了虚拟内存和分页的概念。一个例子是 vLLM 的全有或全无交换策略，该策略利用了处理请求需要将其所有对应的 token 状态都存储在 GPU 内存中的事实。另一个例子是用于恢复被驱逐块的重新计算方法，这在操作系统中是不可行的。此外，vLLM 通过将用于内存访问操作的 GPU 内核与用于注意力等其他操作的 GPU 内核融合，减轻了分页中内存间接寻址的开销。

vLLM 通过利用特定应用的语义，重新诠释并增强了虚拟内存和分页的概念。一个例子是 vLLM 的全有或全无交换策略，该策略利用了处理请求需要将其所有对应的 token 状态都存储在 GPU 内存中的事实。另一个例子是用于恢复被驱逐块的重新计算方法，这在操作系统中是不可行的。此外，vLLM 通过将用于内存访问操作的 GPU 内核与用于注意力等其他操作的 GPU 内核融合，减轻了分页中内存间接寻址的开销。

9 相关工作

通用模型服务系统。 模型服务是近年来一个活跃的研究领域，提出了许多系统来解决深度学习模型部署的各个方面。Clipper [11],TensorFlow Serving [33], Nexus [45], InferLine [10], 和 Clockwork [20] 是一些较早的通用模型服务系统。它们研究用于服务单个或多个模型的批次、缓存、放置和调度。最近，DVABatch [12] 引入了多入口多出口批次。REEF [21] 和 Shepherd [61] 提出了用于服务的抢占。AlpaServe [28]利用模型并行进行统计复用。然而，这些通用系统未能考虑到LLM推理的自回归特性和token状态，导致错失了优化的机会。

针对 Transformer 的专用服务系统。 由于 Transformer 架构的重要性，已开发出众多针对它的专用服务系统。这些系统利用 GPU 内核优化 [1, 29, 31, 56], 高级批处理机制 [14, 60], 模型并行 [1,41, 60]，和参数共享 [64] 进行高效服务。其中，Orca [60] 与我们的方法最相关。

与 Orca 的比较。 Orca [60] 中的迭代级调度和vLLM中的分页注意力是互补的技术：虽然这两个系统都旨在提高GPU利用率，从而提升LLM服务的吞吐量，但Orca通过调度和交错请求来实现这一点，以便更多请求可以并行处理，而vLLM则是通过提高内存利用率，使更多请求的工作集能够适应内存。通过减少内存碎片化并实现共享，vLLM能够并行处理更多请求，与Orca相比实现了2-4×倍的性能提升。实际上，像Orca那样对请求进行细粒度调度和交错处理，使得内存管理更具挑战性，这使得vLLM中提出的技术更加关键。

内存优化。 加速器的计算能力与内存容量之间的差距日益扩大，导致内存成为训练和推理的瓶颈。通过交换 [23, 42, 55], 重新计算 [7, 24]及其组合 [40] 已被用于降低训练的峰值内存。值得注意的是，FlexGen [46] 研究如何在有限的GPU内存下为LLM推理交换权重和token状态，但它并未针对在线服务设置。

limited GPU memory, but it does not target the online serving settings. OLLA [48] optimizes the lifetime and location of tensors to reduce fragmentation, but it does not do fine-grained block-level management or online serving. FlashAttention [13] applies tiling and kernel optimizations to reduce the peak memory of attention computation and reduce I/O costs. This paper introduces a new idea of block-level memory management in the context of online serving.

10 Conclusion

This paper proposes PagedAttention, a new attention algorithm that allows attention keys and values to be stored in non-contiguous paged memory, and presents vLLM, a high-throughput LLM serving system with efficient memory management enabled by PagedAttention. Inspired by operating systems, we demonstrate how established techniques, such as virtual memory and copy-on-write, can be adapted to efficiently manage KV cache and handle various decoding algorithms in LLM serving. Our experiments show that vLLM achieves 2-4× throughput improvements over the state-of-the-art systems.

Acknowledgement

We would like to thank Xiaoxuan Liu, Zhifeng Chen, Yanping Huang, anonymous SOSP reviewers, and our shepherd, Lidong Zhou, for their insightful feedback. This research is partly supported by gifts from Andreessen Horowitz, Anyscale, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

References

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. 2022. DeepSpeed Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale. *arXiv preprint arXiv:2207.00032* (2022).

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[3] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2000. A neural probabilistic language model. *Advances in neural information processing systems* 13 (2000).

[4] Ond rej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. 2016. Findings of the 2016 Conference on Machine Translation. In *Proceedings of the First Conference on Machine Translation*. Association for Computational Linguistics, Berlin, Germany, 131–198. http://www.aclweb.org/anthology/W/W16/W16-2301

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[7] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[8] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. https://lmsys.org/blog/2023-03-30-vicuna/

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[10] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 477–491.

[11] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 613–627.

[12] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.

[13] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[14] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 389–402.

[15] FastAPI. 2023. FastAPI. https://github.com/tiangolo/fastapi.

[16] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[17] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. 2021. Ai and memory wall. *RiseLab Medium Post* 1 (2021), 6.

[18] Github. 2022. https://github.com/features/copilot

[19] Google. 2023. https://bard.google.com/

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[21] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent {GPU-accelerated}{DNN} Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[23] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.

[24] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. 2020. Checkmate: Breaking the memory wall with optimal tensor rematerialization.

OLLA [48]优化张量的生命周期和位置以减少碎片化，但它不进行细粒度的块级管理或在线服务。FlashAttention [13] 应用tiling和内核优化以减少注意力计算的峰值内存并降低I/O成本。本文介绍了在线服务背景下的一种新的块级内存管理思路。

10 结论

本文提出了分页注意力（PagedAttention）这一新型注意力算法，允许注意力键值对存储在非连续的分页内存中，并介绍了vLLM，这是一个通过分页注意力实现高效内存管理的高吞吐量大语言模型（LLM）服务系统。受操作系统的启发，我们展示了如何将虚拟内存和写时复制等成熟技术应用于高效管理LLM服务中的KV缓存和处理各种解码算法。我们的实验表明，vLLM相较于现有最佳系统实现了2-4× 的吞吐量提升。

致谢

我们感谢刘晓轩、陈志峰、黄艳平、匿名的SOSP审稿人以及我们的Shepherd周立东提供的富有见地的反馈。这项研究部分得到了Andreessen Horowitz、Anyscale、Astronomer、Google、IBM、Intel、Lacework、Microsoft、阿联酋人工智能大学、三星SDS、Uber和VMware的资助。

参考文献

[1] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, 等. 2022. DeepSpeed Inference: 实现前所未有的规模下 Transformer 模型的高效推理. *arXiv* 预印本 *arXiv:2207.00032* (2022). [2] Jimmy Lei Ba, Jamie Ryan Kiros, 和 Geoffrey E Hinton. 2016. Layer normalization. *arXiv* 预印本 *arXiv:1607.06450* (2016). [3] Yoshua Bengio, Réjean Ducharme, 和 Pascal Vincent. 2000. A neural probabilistic language model. 神经信息处理系统进展 13 (2000). [4] Ondrej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, 和 Marcos Zampieri. 2016. 2016 年机器翻译会议发现. 收录于 第一届机器翻译会议论文集. 计算语言学协会, 德国柏林, 131–198. http://www.aclweb.org/anthology/W/W16/W16-2301[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, 等. 2020. 语言模型是少样本学习器. 神经信息处理系统进展 33 (2020), 1877–1901.[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas

约瑟夫、格雷格·布罗克曼等人。2021年。评估在代码上训练的大型语言模型。*arXiv*预印本 *arXiv:2107.03374* (2021)。[7] 天齐·陈、徐冰、张驰元和卡洛斯·盖斯特林。2016年。以次线性内存成本训练深度网络。*arXiv*预印本 *arXiv:1604.06174* (2016)。[8] 魏林·蒋、李朱浩、林子、盛英、吴张浩、张浩、郑连明、庄思源、庄永浩、约瑟夫·E·冈萨雷斯、伊翁·斯托伊卡和埃里克·P·辛。2023年。Vicuna：一个开源聊天机器人，以90%*的ChatGPT质量令人印象深刻GPT-4。https://lmsys. org/blog/2023-03-30-vicuna/[9] 阿卡纳施卡·乔杜里、沙兰·纳兰、雅各布·德夫林、马arten·博斯马、高尔瓦夫·米什拉、亚当·罗伯茨、保罗·巴姆、海因·温·丘恩、查尔斯·萨顿、塞巴斯蒂安·盖尔曼等人。2022年。Palm：使用路径扩展语言建模。*arXiv*预印本 *arXiv:2204.02311* (2022)。[10] 丹尼尔·克朗肖、古拉埃·塞拉、莫祥熙、科尔·祖马、伊翁·斯托伊卡、约瑟夫·冈萨雷斯和阿列克谢·图曼诺夫。2020年。InferLine：针对预测服务管道的延迟感知提供和扩展。在第*11*届*ACM*云计算研讨会。477–491.[11] 丹尼尔·克朗肖、王欣、朱利奥·周、迈克尔·J·富兰克林、约瑟夫·E·冈萨雷斯和伊翁·斯托伊卡。2017年。Clipper：一个低延迟的在线预测服务系统。在第*14*届*USENIX*网络系统设计与实现研讨会（*NSDI 17*）。613–627.[12] 崔伟豪、赵寒、陈泉、魏浩、李子睿、曾德、李超和郭明毅。2022年。DVABatch：针对GPU上高效处理DNN服务的多样化感知多入口多出口批处理。在*2022 USENIX*年度技术会议（*USENIX ATC 22*）。183–198。[13] 崔特·Dao、付丹、斯特凡诺·埃尔蒙、阿蒂·鲁德拉和克里斯托弗·雷。2022年。Flashattention：具有io感知的快速和内存高效的精确注意力。神经信息处理系统进展 35 (2022)，16344–16359。[14] 方嘉瑞、余杨、赵成都和周杰。2021年。TurboTransformers：一个用于Transformer模型的GPU服务系统。在第*26*届*ACM SIGPLAN*关于并行编程原理和实践的研讨会。389–402。[15] FastAPI。2023年。FastAPI。https://github.com/tiangolo/fastapi。[16] 高平、余令帆、吴永伟和李金阳。2018年。使用细胞批处理进行低延迟RNN推理。在第十三届*EuroSys*会议。1–15。[17] 阿米尔·戈拉米、姚哲伟、金世雄、迈克尔·W·马霍尼和库尔特·基茨。2021年。人工智能和内存墙。*RiseLab Medium*文章 1 (2021)，6。[18] Github。2022年。https://github.com/features/copilot[19] 谷歌。2023年。https://bard.google.com/[20] 阿潘·古贾拉蒂、雷扎·卡里米、萨菲亚·阿尔扎亚特、魏浩、安东尼·考夫曼、Ymir·维格弗森和乔纳森·M·麦克。2020年。像时钟一样服务{DNNs}：从底层开始的性能可预测性。在第*14*届*USENIX*操作系统设计与实施研讨会（*OSDI 20*）。443–462。[21] 韩明聪、张汉泽、陈荣和陈海波。2022 年。微秒级抢占，用于并发{GPU加速}{DNN}推理。在第*16*届*USENIX*操作系统设计与实施研讨会（*OSDI 22*）。539–558。[22] 何凯明、张祥宇、任少卿和孙剑。2016年。深度残差学习用于图像识别。在*IEEE*计算机视觉和模式识别会议。770–778。[23] 黄建钦、金吉和李金阳。2020年。Swapadvisor：通过智能交换将深度学习推向GPU内存极限之外。在第*25*届国际计算机辅助语言和操作系统会议。1341–1355。[24] 帕拉斯·贾因、阿吉·贾因、安鲁达·N·鲁西马哈、阿米尔·戈拉米、皮特·阿贝尔、约瑟夫·冈萨雷斯、库尔特·基茨和伊翁·斯托伊卡。2020年。Checkmate：通过最优张量重材料化打破内存墙。

Proceedings of Machine Learning and Systems 2 (2020), 497–511.
[25] Tom Kilburn, David BG Edwards, Michael J Lanigan, and Frank H Sumner. 1962. One-level storage system. *IRE Transactions on Electronic Computers* 2 (1962), 223–235.
[26] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).
[27] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).
[28] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. *arXiv preprint arXiv:2302.11665* (2023).
[29] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 881–897.
[30] NVIDIA. [n. d.]. Triton Inference Server. https://developer.nvidia.com/nvidia-triton-inference-server.
[31] NVIDIA. 2023. FasterTransformer. https://github.com/NVIDIA/FasterTransformer.
[32] NVIDIA. 2023. NCCL: The NVIDIA Collective Communication Library. https://developer.nvidia.com/nccl.
[33] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).
[34] OpenAI. 2020. https://openai.com/blog/openai-api
[35] OpenAI. 2022. https://openai.com/blog/chatgpt
[36] OpenAI. 2023. https://openai.com/blog/custom-instructions-for-chatgpt
[37] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
[38] LMSYS ORG. 2023. Chatbot Arena Leaderboard Week 8: Introducing MT-Bench and Vicuna-33B. https://lmsys.org/blog/2023-06-22-leaderboard/.
[39] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).
[40] Shishir G Patil, Paras Jain, Prabal Dutta, Ion Stoica, and Joseph Gonzalez. 2022. POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging. In *International Conference on Machine Learning*. PMLR, 17573–17583.
[41] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. *arXiv preprint arXiv:2211.05102* (2022).
[42] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training.. In *USENIX Annual Technical Conference*. 551–564.
[43] Reuters. 2023. https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/
[44] Amazon Web Services. 2023. https://aws.amazon.com/bedrock/
[45] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 322–337.
[46] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. 2023. High-throughput Generative Inference of Large Language Models with a Single GPU. *arXiv preprint arXiv:2303.06865* (2023).
[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).
[48] Benoit Steiner, Mostafa Elhoushi, Jacob Kahn, and James Hegarty. 2022. OLLA: Optimizing the Lifetime and Location of Arrays to Reduce the Memory Usage of Neural Networks. (2022). https://doi.org/10.48550/arXiv.2210.12924
[49] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).
[50] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[51] ShareGPT Team. 2023. https://sharegpt.com/
[52] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[54] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for {Log-Structured}{Key-Value} Store on Persistent Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 773–788.
[55] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.
[56] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. LightSeq: A High Performance Inference Library for Transformers. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers*. 113–120.
[57] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-Instruct: Aligning Language Model with Self Generated Instructions. *arXiv preprint arXiv:2212.10560* (2022).
[58] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.
[59] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).
[60] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for {Transformer-Based} Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 521–538.
[61] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

机器学习和系统研讨会论文 2 (2020)，497–511。[25] 汤姆·基尔本、大卫·B·G·爱德华兹、迈克尔·J·兰igan和弗兰克·H·萨默纳。1962年。一级存储系统。*IRE*电子计算机交易 2 (1962)，223–235。[26] 布莱恩·莱斯特、拉米·阿尔-拉夫、诺亚·康斯坦特。2021年。规模的力量，用于参数高效的提示调整。*arXiv*预印本 *arXiv:2104.08691* (2021)。[27] 李思嘉和李佩西。2021年。前缀调整：优化生成用连续提示。*arXiv*预印本 *arXiv:2101.00190* (2021)。[28] 李朱浩、郑连明、钟银民、刘文、盛英、金欣、黄艳平、陈志峰、张浩、约瑟夫·E·冈萨雷斯等人。2023年。AlpaServe：具有模型并行的统计复用，用于深度学习服务。*arXiv*预印本 *arXiv:2302.11665* (2023)。[29] 马凌晓、谢志强、杨志、薛继龙、苗友山、崔伟豪、胡文祥、杨帆、张林涛和周立冬。2020年。Rammer：使用rtasks实现整体深度学习编译器优化。在第*14*届*USENIX*操作系统设计与实施会议。881–897。[30] 英伟达。[n。d。]Triton推理服务器。https://developer.nvidia.com/ nvidia-triton-inference-server。[31] 英伟达。2023年。FasterTransformer。https://github.com/NVIDIA/FasterTransformer。[32] 英伟达。2023年。NCCL：英伟达集体通信库。https://developer.nvidia.com/nccl。[33] 克里斯托弗·奥尔斯顿、诺亚·菲德尔、基里尔·戈罗沃伊、杰里迈亚·哈姆斯滕、劳利、李方伟、维努·拉贾谢克哈、苏克里蒂·拉梅什和乔丹·索伊克。2017年。Tensorflow-serving：灵活、高性能的ml服务。*arXiv*预印本 *arXiv:1712.06139* (2017)。[34] OpenAI。2020年。https://openai.com/blog/openai-api [35] OpenAI。2022年。https://openai.com/blog/chatgpt [36] OpenAI。2023年。https://openai.com/blog/custom-instructions-for- chatgpt
[37] 2023 4 2303 08774 [ ]
[38] 2023 8
OpenAI。. GPT技术报告。arXiv: . cs.CL LMSYS ORG。. Chatbot Arena排行榜周：介绍MT-Bench和Vicuna-33B。 https://lmsys.org/blog/2023-06-22- leaderboard/。[39]亚当·帕斯克、山姆·格罗斯、弗朗西斯科·马萨、亚当·莱勒、詹姆斯·布拉德伯里、格雷戈里·卡南、特雷弗·基利恩、李增明、娜塔莉亚·吉梅尔申、卢卡·安蒂加等人。2019年。Pytorch：一个命令式风格、高性能的深度学习库。神经信息处理系统进展 32 (2019)。[40] 希什尔·G·帕蒂尔、帕拉斯·贾因、普拉巴尔·杜塔、伊翁·斯托伊卡和约瑟夫·冈萨雷斯。2022年。POET：在集成重材料化和分页的微型设备上训练神经网络。在国际机器学习会议。PMLR，17573–17583。[41] 雷纳·波普、肖尔托·道格拉斯、阿卡纳施卡·乔杜里、雅各布·德夫林、詹姆斯·布拉德伯里、安塞尔姆·列夫斯卡娅、乔纳森·希克、肖凡·肖、阿格拉瓦尔和杰夫·迪恩。2022年。高效扩展Transformer推理。*arXiv*预印本 *arXiv:2211.05102* (2022)。[42] 任杰、山米亚姆·拉贾班达里、雷扎·亚兹达尼·阿明阿巴迪、奥拉廷吉·鲁瓦塞、杨双燕、张敏嘉、李东和何宇雄。2021年。ZeRO-Offload：使十亿级模型训练民主化.. 在 *USENIX*年度技术会议。551–564。[43] 路透社。2023年。https://www.reuters.com/technology/tech-giants-ai-like-bing-bard-poses-billion-dollar-search-problem-2023-02-22/ [44] 亚马逊网络服务。2023年。https://aws.amazon.com/bedrock/[45] 沈海晨、陈乐群、金宇晨、赵亮宇、孔冰雨、马太·菲利普斯、阿文德·克里希纳穆尔蒂和拉维·苏达рам。2019年。Nexus：一个用于加速基于DNN的视频分析的GPU集群引擎。在第*27*届*ACM*操作系统原理研讨会。322–337。[46] 盛英、郑连明、袁彬航、李朱浩、马克斯·拉宾宁、付丹·余、谢志强、陈北迪、克拉克·巴雷特、约瑟夫·E·冈萨雷斯等人。2023年。大型生成推理的高吞吐量

单GPU语言模型。*arXiv* 预印本 *arXiv:2303.06865* (2023)。[47] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, 和 Bryan Catanzaro。2019。Megatron-lm：使用模型并行训练十亿参数语言模型。*arXiv* 预印本 *arXiv:1909.08053* (2019)。[48] Benoit Steiner, Mostafa Elhoushi, Jacob Kahn, 和 James Hegarty。2022。OLLA：优化数组的生命周期和位置以减少神经网络的内存使用。(2022)。https://doi.org/10.48550/arXiv.2210.12924[49] Ilya Sutskever, Oriol Vinyals, 和 Quoc V Le。2014。使用神经网络的序列到序列学习。神经信息处理系统进展 27 (2014)。[50] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, 和 Tatsunori B. Hashimoto。2023。Stanford Alpaca：一个指令跟随LLaMA模型。https:// github.com/tatsu-lab/stanford_alpaca。[51] ShareGPT 团队。2023。https://sharegpt.com/ [52] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie- Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, 等等。2023。Llama：开放高效的基座语言模型。*arXiv* 预印本 *arXiv:2302.13971* (2023)。[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, 和 Illia Polosukhin。2017。注意力就是全部你需要。神经信息处理系统进展 30 (2017)。[54] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, 和 Jiwu Shu。2022。Pacman：持久内存上{日志结构}{键值}存储的高效压缩方法。在 *2022 USENIX* 年度技术会议 *(USENIX ATC 22)*。773–788。[55] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuai-wen Leon Song, Zenglin Xu, 和 Tim Kraska。2018。Superneurons：用于训练深度神经网络的动态GPU内存管理。在 第*23*届*ACM SIGPLAN*并行编程原理与实践研讨会。 41–53。 [56] Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, 和 Lei Li。2021。LightSeq：Transformer的高性能推理库。在 第*2021*届北美计算语言学协会会议：人机语言技术：工业论文。 113–120.[57] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, 和 Hannaneh Hajishirzi。2022。Self-Instruct：使用自生成指令对齐语言模型。*arXiv* 预印本 *arXiv:2212.10560* (2022)。[58]Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, 等等。2020。Transformer：最先进的自然语言处理。在 第*2020*届自然语言处理经验方法会议：系统演示。 38–45.[59] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, 等等。2016。Google的神经机器翻译系统：弥合人与机器翻译之间的差距。*arXiv* 预印本 *arXiv:1609.08144* (2016)。 [60] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, 和 Byung-Gon Chun。2022。Orca：基于{Transformer}生成模型的分布式服务系统。在 第*16*届*USENIX*操作系统设计与实现研讨会 *(OSDI 22)*。521–538。 [61] Hong Zhang, Yupeng Tang, Anurag Khandelwal, 和 Ion Stoica。2023。SHEPHERD：在野外服务DNNs。在 第*20*届 *USENIX*网络系统设计与实现研讨会 *(NSDI 23)*。 USENIX Association, Boston, MA, 787–808。 https://www.usenix.org/conference/nsdi23/presentation/zhang-hong

[62] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[63] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. 2022. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 559–578.

[64] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. 2022. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 489–504.

[62] 苏珊·张，斯蒂芬·罗勒，纳曼·高oyal，米克尔·阿特克塞，茅亚晨，陈舒慧，克里斯托弗·迪万，莫娜·迪亚布，李先，林希维多利亚，等。2022年。OPT：开放预训练的Transformer语言模型。*arXiv*预印本 *arXiv:2205.01068*(2022)。[63] 郑连民，李朱浩，张浩，庄永浩，陈志峰，黄艳平，王怡达，徐元忠，朱道丹

Eric P Xing 等. 2022. Alpa: 自动化跨与跨算子并行，用于分布式深度学习. 收录于 第*16*届*USENIX*操作系统设计与实现会议 *(OSDI 22)*. 559–578. [64] Zhe Zhou, Xuechao Wei, Jiejing Zhang 和 Guangyu Sun. 2022. PetS: 一个统一的参数高效Transformer服务框架. 收录于 第*22*届*USENIX*年度技术会议 *(USENIX ATC 22)*. 489–504.