# Paged Attention

# 1. LLM 推理的两阶段

一般分为prefill-decode两个阶段。

## 1.1. Prefill

在prefill阶段，我们把prompt一次性喂给LLM。这样LLM就能为prompt中的所有token生成K，V矩阵：
（数学公式后补）
如果使用了KVcache，那么这一步计算的KV矩阵就会存下来，这样在后续（prompt后的第2个token起）就能直接用了。因为在自回归过程中，attention机制还是需要回看prompt的内容，在这个过程中，对于prompt中的$token_i$, 其第一层attention是$token_{1:i}$的加权和，这一点在自回归生成后续token时是不变的，是KVcache复用的基础。

- 一个KVcache块里装了什么？

## 1.2. Decode

在Decode中，我们自回归地生成prompt的后续token（也即输出），在这个过程中，我们可以不断复用之前$token$已经存下的KVcache，同时当当前token生成完毕后，我们也将它的KV存入cache，方便下一个token的生成。

由于Decode阶段的是逐一生成token的，因此它不能像prefill阶段那样能做大段prompt的并行计算，所以在LLM推理过程中，Decode阶段的耗时一般是更大的。

## 1.3. 两阶段中的KVcache

所以推理过程中，KVcache的大小基本上与token序列的长度成正比，也因此KVcache所需的存储空间也与token序列的长度成比例，随着每次prompt和生成长度的变化而变化。

这导致vLLM论文中提到的问题，也即难以提前为KVcache预留合适的空间，使得既能满足推理需求又不至于浪费。

# 2. 为KVcache分配空间的方式

LLM在实际应用中的部署方式一般是隐藏细节，暴露接口，接受request，用其中的prompt进行推理，并返回结果。

一般对于一种模型，我们不会部署多个同样的模型（因为成本很高），而是在一个运行着的模型上同时进行多个推理。因而有两个概念:

- *batch_size*, 也即同时在这个模型实例上运行的推理的个数（prompt的条数）
- *max_sequence_length*, 一条prompt的最大允许长度（必须限制，因为最大开销是此长度的平方）

## 2.1. 完整内存分配

对于每一个prompt，可预测的最大开销就是 $batch\_size * max\_sequence\_length$ 个KVcache块。这样的坏处在于内存浪费，因为并不是每个prompt的长度都接近上限。

另外一个坏处在于，这样的分配策略使得只有长度足够的*完整*内存块能被利用，即使有容量相同的碎片内存，我们也无法使用。

## 2.2. 随用随分配

在需要更多内存的时候才分配更多内存。vLLM的做法正是这样，具体的策略就是*Paged_Attention*. 这个概念是继承自操作系统的。

为了避免各个进程间的打扰，操作系统使用了“虚拟化”的思想，将实体内存通过管理单元分为多个连续存储空间，对于每片空间，将其映射为一块完整的虚拟内存提供给进程，进程可以完全安全且自由地使用这一块内存。

这个方法也会碰到我们之前KVcache存储的一个痛点：因为需要连续内存，即使有容量相同的碎片内存，我们也无法使用。所以操作系统进而使用“分页管理”的机制。

分页管理的本质是（1）将物理内存分为更小的块，虚拟内存可以由不连续的小块组成。（2）不直接为进程分配一大块内存，而是动态的为进程现在在运行的部分分配内存，用完即删，再加载要运行的其他部分。 由于把进程需要的内存也分成了更小的块，所以我们可以更加细粒度的管理内存的分配。

## 2.3. Paged Attention

本质上就是把KVcache也进行分页管理。由于LLM实例一次要处理多个prompt，所以实际上就像是操作系统要处理多个进程的内存分配一样。

vLLM把内存也分成若干个小块（一个块的大小大约是16个token的KV值）。这些物理内存可以是离散的（一般在GPU上），然后由管理器把它们映射为连续的虚拟内存，提供给推理加速框架（vLLM）用来存储KVcache。只有当token序列不断扩展，当前块存满，框架需要使用新的虚拟内存块，系统才分配一个新块给框架。

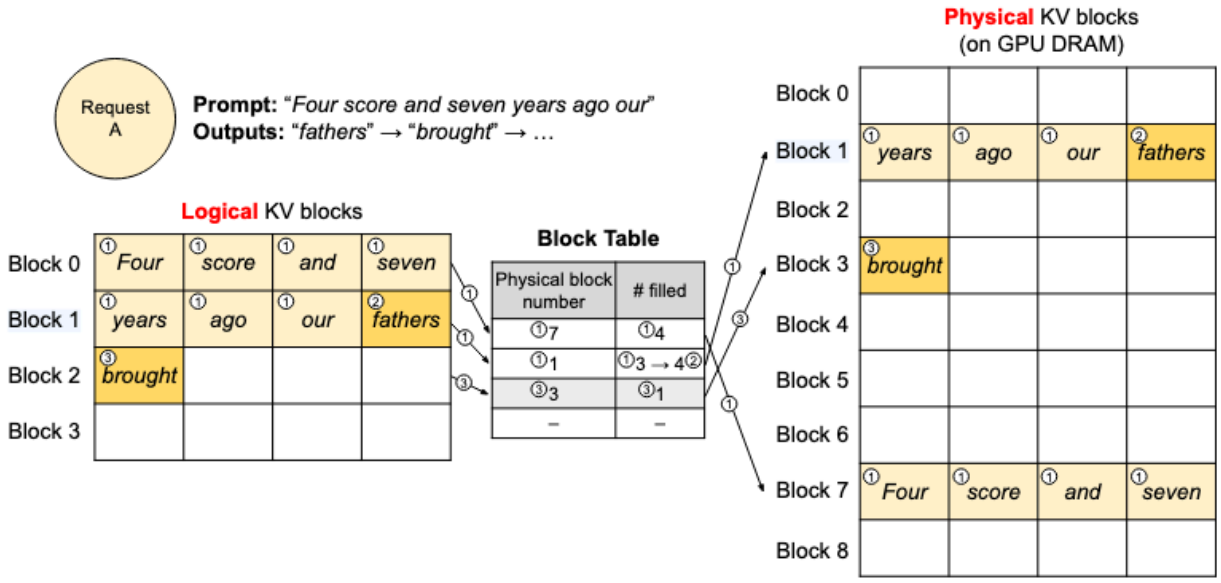


**Figure 6.** Block table translation in vLLM.

参照上图，带圈序号（1）代表prefill阶段，这里我们并行计算prompt里各个token的KV值并存进内存块（本图中一个block可以存4个token的KV值）。（2）是decode阶段，我们自回归生成token并把对应token的KV存起来以准备复用。

vLLM的PagedAttention在一次推理中的运行步骤如下：

- （1）划分逻辑块
  根据prefill时token序列的长度决定需要几个虚拟内存块（本图中 $7\%4+1=2$）。其中第二个内存块还有一个 token的空余。
- （2）映射到物理块
  我们已经知道需要几个虚拟的内存块，接下来就在物理内存上分配这些。映射时我们通过block table来记录虚拟-物理内存的映射关系。我们需要记录（i）虚拟内存对应的物理内存的位置（ii）物理内存上已经填满的槽位。对本例而言第一块四个都满，第二个前三个满了。
- （3）进入decode阶段开始生成
  复用存储的KVcache进行计算。把新token对应的KV存到上一个虚拟内存块没填满的槽位上。此时逻辑块都装满了，所以vLLM再创建一个新的虚拟内存块，同样的，block table中会记录这个新内存块地址以及填满的槽位（此时都是空的，所以下个词要存入第一个槽位）
- （4）循环生成token，复用KV直到<EOS>出现

## 2.4. 处理多个请求时的情况

vLLM并行地为收到的prompt执行以上的内容，注意的是，vLLM中的batch不像神经网络训练中的batch是一同输入，一同处理完的，而是先处理完的先输出，并且把这个prompt占据的内存块释放。

# 3. KVcache 复用的几种情况

以上介绍的机制是处理一条promtpt时其内部KVcache的复用情况，基本思想就是在生成靠后的token时可以复用靠前的token的KVcache。

然而，在有些情况下，不同的prompt（或者一条prompt后续的不同分支之间）也是可以复用KVcache的。

## 3.1. Parallel Sampling

也即对于同一个prompt，让LLM生成多种不同的回答。一种做法是将这个prompt服制成两份，作为两个不同的prompt独立处理。但由于prompt完全一

样，其tokenize之后的token序列也完全一样，这部分token的KVcache是可以复用的。

vLLM这样处理Parallel Sampling：

- （1）Prefill阶段
  我们依旧为两个独立的prompt分配虚拟内存块。但由于token完全一样，所以我们只需要在物理内存上分配一块内存以存储这些token的KVcache，再将两个虚拟内存块都映射到这一块内存上就行。
- （2）Decode阶段
  这一阶段LLM开始生成新的token，而新的token可能不同。vLLM使用了“copy-on-write”机制。当LLM生成了不一样的token时，我们显然不能再复用这个token，而因为KVcache存储的最小单位是块，所以第一个不同token所在的块我们也不能够复用。
  在写入第一个不同token时，我们在物理内存上开辟一个新的块，存储这个不同token（mothers）所在的块，并把对应的prompt后续分支的虚拟内存块映射到这个新块上（物理块block1）。另一个prompt后续分支中包含不同token（fathers）的虚拟内存块则映射在之前的块（物理block3）上。换言之就是从第一个不同token所在的块开始二者“解耦”了。

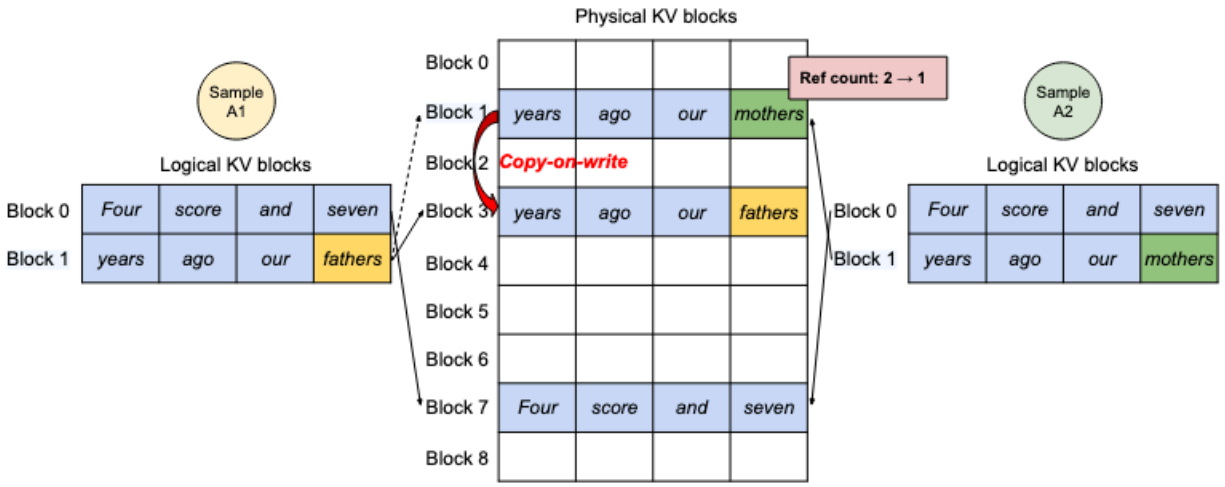


**Figure 8.** Parallel sampling example.