【手撕LLM-KVCache】显存刺客的前世今生--文末含代码

小冬瓜AIGC
原创课程➡️公众号：手撕LLM

来自专栏 · 手撕LLM

> 我是小冬瓜AIGC，原创超长文知识分享，原创课程已帮助多名同学速成上岸LLM赛道：手撕LLM+RLHF
> 研究方向：LLM、RLHF、Safety、Alignment

本文摘自[高效Attention]系列：【手撕LLM-KVCache】显存刺客的前世今生

阅读本文前推荐阅览：小冬瓜AIGC:【手撕LLM-Generation】Top-K+重复性惩罚

前言

什么是KV-Cache?

`KV-Cache` 是一种加速推理的技术, 现代的LLM使用，包括 `LLaMA`
`KV-Cache` 在长文本输入的情况下，占用的显存线性随句长增长，此时需要做 `cache` 优化
可以说系统掌握KV-Cache是做LLM推理加速的基础，另一条线路是纯GPU算法优化如FlashAttention

阅读完本文可以掌握：

- `LLM` 推理中的 `KV-Cache` 作用、原理、流程
- `KV-Cache` 的显存占用量分析
- `KV-Cache` 的优化手段
- 手撕 `KV-Cache`
- 现代 `LLM` 架构，不同的 `KV-Cache` 优化实现手段

1. Generate时的Next Token推理

```
# 无KV-Cache
idx = torch.randn([1,10])
for i in range(max_length):
    logist = model(idx)
    next_idx = softmax(argmax(logist))
    idx = cat(idx, next_idx)
```

```
# 有KV-Cache
idx = torch.randn([1,10])
for i in range(max_length):
    logist = model(idx)
    next_idx = softmax(argmax(logist))
    idx = next_idx
```


1.1 无KV-Cache生成代码

- 无 `KV-Cache` 推理

```
import torch
import torch.nn.functional as F
from transformers import LlamaModel, LlamaConfig, LlamaForCausalLM

# 加载模型
```

```python
config = LlamaConfig(vocab_size = 100,
                    hidden_size = 256,
                    intermediate_size = 512,
                    num_hidden_layers = 2,
                    num_attention_heads = 4,
                    num_key_value_heads = 4,
                    )
model = LlamaForCausalLM(config)

# 创建数据、不使用tokenizer
X = torch.randint(0, 100, (1,10))
print(X.shape)

#
idx={}
idx['input_ids'] = X
for i in range(4):
    print(f"\nGeneration第{i}个时的输入{idx['input_ids'].shape}: ")
    print(f"Generation第{i}个时的输入{idx['input_ids']}: ")
    output = model(**idx)
    logits = output['logits'][:,-1,:]
    idx_next = torch.argmax(logits , dim=1)[0]

    idx['input_ids'] = torch.cat((idx['input_ids'], idx_next.unsqueeze(0).unsqueeze(1)), dim=-1)
```

输出结果

```
torch.Size([1, 10])

Generation第0个时的输入torch.Size([1, 10]):
Generation第0个时的输入tensor([[48,  8, 96,  3,  1,  3, 65, 85, 18, 25]]):

Generation第1个时的输入torch.Size([1, 11]):
Generation第1个时的输入tensor([[48,  8, 96,  3,  1,  3, 65, 85, 18, 25,  1]]):

Generation第2个时的输入torch.Size([1, 12]):
Generation第2个时的输入tensor([[48,  8, 96,  3,  1,  3, 65, 85, 18, 25,  1, 66]]):

Generation第3个时的输入torch.Size([1, 13]):
Generation第3个时的输入tensor([[48,  8, 96,  3,  1,  3, 65, 85, 18, 25,  1, 66,  3]]):
```

1.2 有KV-Cache生成代码

- 在推理时，计算 `Qi@K^T` 形式的 `Attention`，如下图
- `Qi` 对应预测 `Token_{i+1}`

```python
    # this code generate With KV Cache
    i = 0
    T = idx.size(0)
    T_new = T+max_new_tokens
    empty = torch.empty(T_new, dtype=dtype, device=device)
    empty[:T] = idx
    idx = empty
    input_pos = torch.arange(0, T, device=device)
    max_new_tokens = 10
    for _ in range(max_new_tokens):
        x = idx.index_select(0, input_pos).view(1, -1)
        print(f"input_t{i}: ", x.int())
        i += 1
        # forward
```

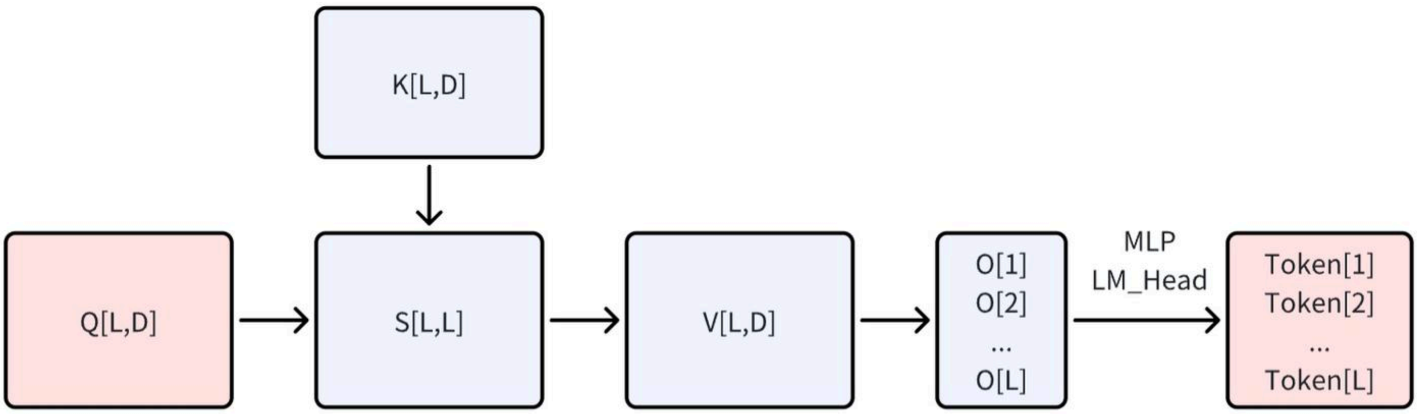
```
        logits = model(x, max_seq_length, input_pos)
        probs = torch.nn.functional.softmax(logits, dim=-1)
        idx_next = torch.multinomial(probs, num_samples=1).to(dtype=dtype)
        # advance
        input_pos = input_pos[-1:] + 1
        # concatenate the new generation
        idx = idx.index_copy(0, input_pos, idx_next)

    return idx
```

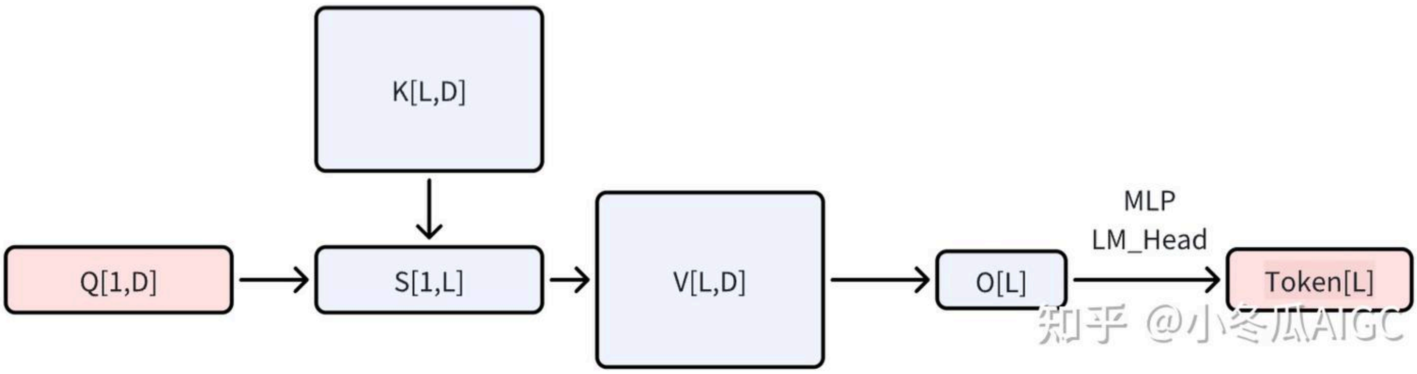
结果

```
input_t0:  tensor([[ 1,   3,  13,   9,  20,   4,   3,   9,   5]], dtype=torch.int32)
input_t1:  tensor([[3]], dtype=torch.int32)
input_t2:  tensor([[8]], dtype=torch.int32)
input_t3:  tensor([[4]], dtype=torch.int32)
input_t4:  tensor([[3]], dtype=torch.int32)
input_t5:  tensor([[6]], dtype=torch.int32)
input_t6:  tensor([[13]], dtype=torch.int32)
input_t7:  tensor([[6]], dtype=torch.int32)
input_t8:  tensor([[15]], dtype=torch.int32)
input_t9:  tensor([[23]], dtype=torch.int32)
```

2. LLM 训练和推理图解

- 训练时，`QKV` 满维度算 `Attention` 得出所有的 `next_token`
- 推理时，要预测下个 `token` 只需要当前最尾的一个 `q`
- 我们可以得出一个结论, `Q[-1],K[:],V[:]` 就可以计算 `next_token`

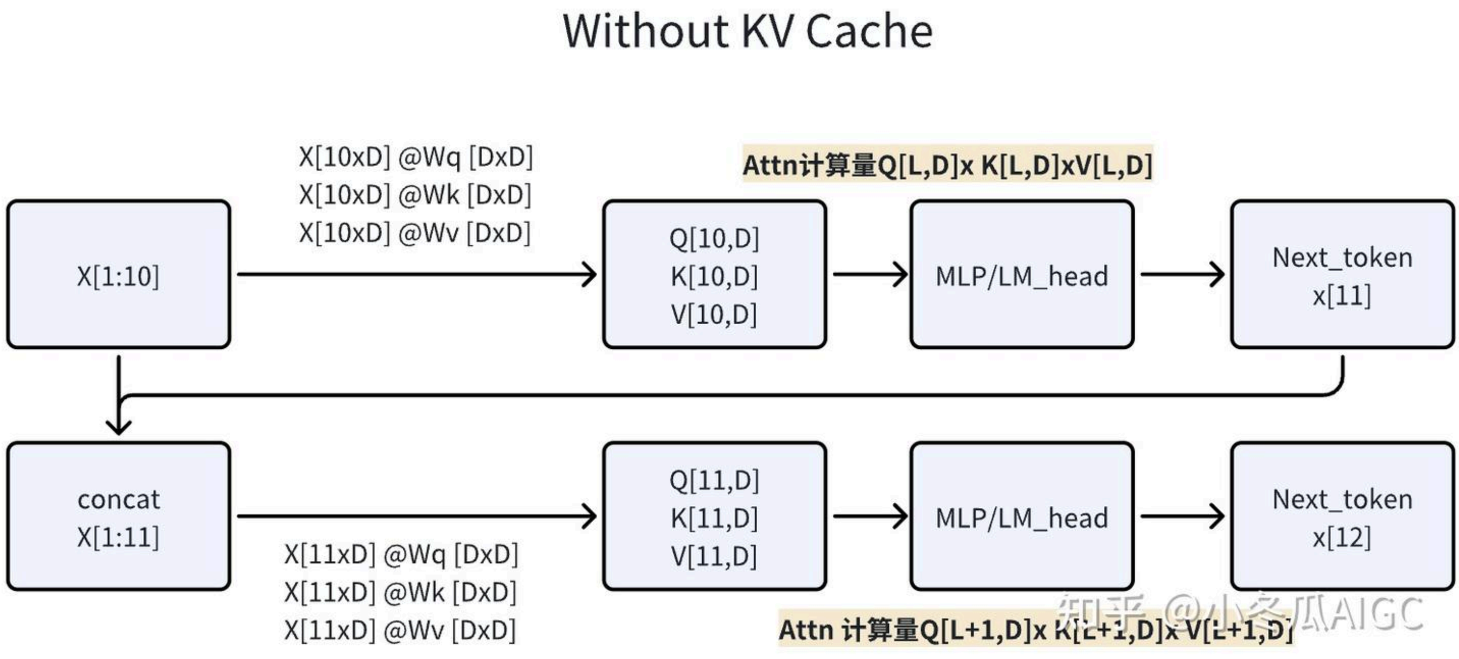


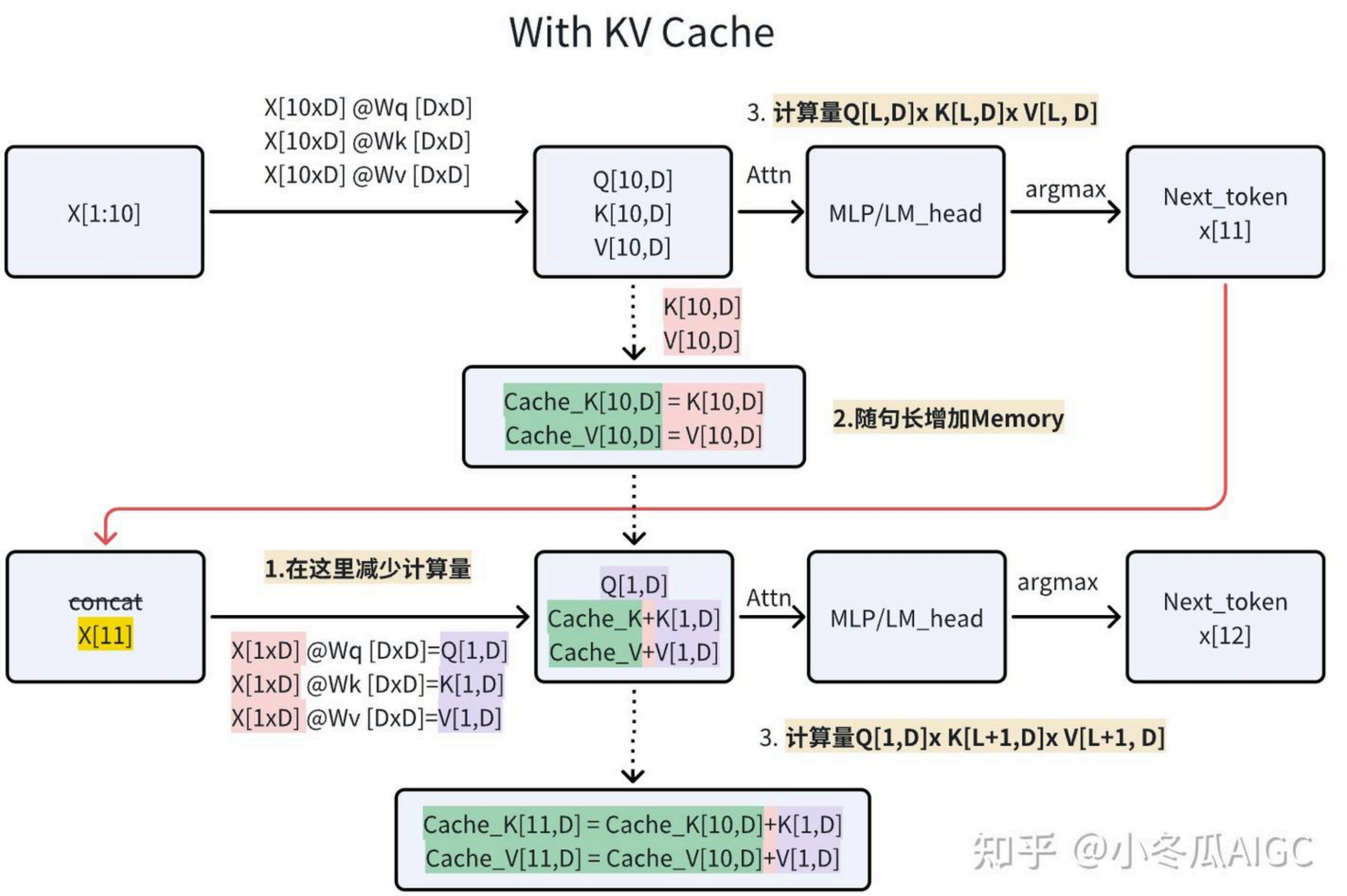


3. KV-Cache流程

- Without `KV-Cache` 每次需要计算全 `Wq(X),Wk(X), Wv(X)`，每次需要计算全量 `Attn`

- With `KV-Cache`，第一步计算完整 `Attn`，将 `KV` 保存成 `KV_cache`
- With `KV-Cache`，第二步，取第一步的 `Next Token` 计算 `QKV`， 将 `[KV_cache,KV]` 联合，计算出 `QKV`
- `KV-Cache` 每个循环累增， memory 量：2*(N层*L长度*D维度)

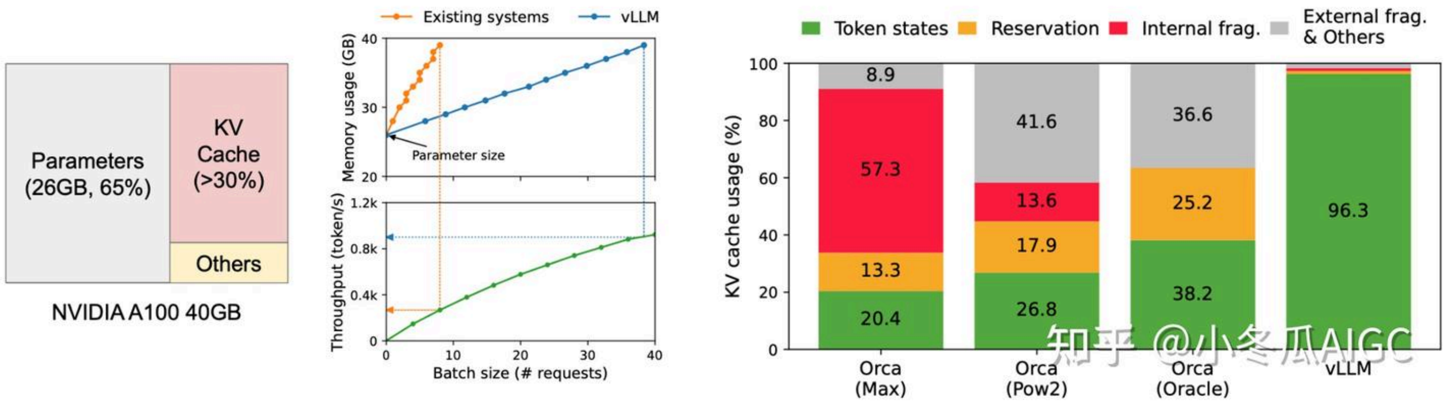




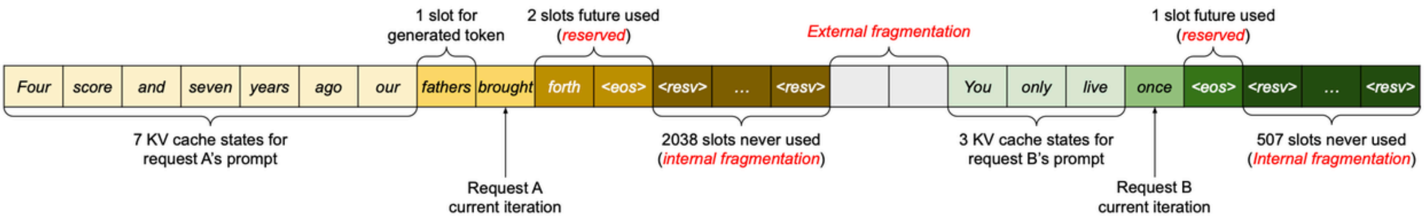
4. KV-Cache占用分析

- `KV-Cache` 计算量
  - `2 x L x batch_size x [d x n_kv_heads] x Layers x k-bits`
- Existing System Memory中 `KV cache` >30%占用
- vLLM 在40GB能跑30+个 `batchsize`， 800+ `token/s` 输出，相较baseline `<10 batchsize`， `300+token/s` 输出

- 现 `GPU` `KV Cache` 中有效存储率低



这里可以看出在连续的存储空间里有大量的 `KV-Cache` 存储碎片



关于 `KV-Cache` 的计算量分析，进一步阅读参考这篇博文Transformer Inference Arithmetic

5. KV Cache推理优化

根据公式总结有四类方式

- 2 x Length x batch_size x [d x n_kv_heads] x Layers x k-bits x 内存模型

1. `n_kv_heads` : `MQA` / `GQA` 通过减少KV的头数减少显存占用
2. `Length` ：通过减少长度 `L` , 以减少 `KV` 显存占用，如使用循环队列管理窗口 `KV`
3. `KV-Cache` 的管理：从 `OS` (操作系统)的内存管理角度，减少碎片，如Paged Attention
4. `K-bits` : 从量化角度减少 `KV cache` 的宽度，如使用 `LLM-QAT` 进行量化

5.1 减少头数 MQA/GQA

- 一种手段是减少KV heads的数量，如果以 `MQA(Multi-Query-Attention)` 来说， `KV head 8->1` 之间节省7倍存储量
- 对于 `GQA(Grouped-Query_Attention)` 来说，平衡精度将KV head 8 -> N, 1<N<8之间trading off精度和速度
- 2 x L x batch_size x D[d x n_kv_heads] x Layers x k-bits

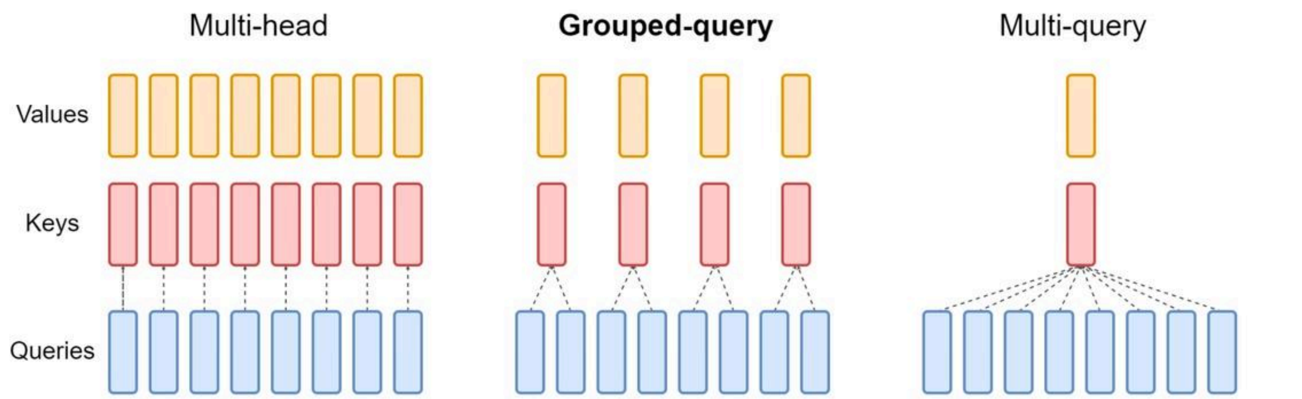


Figure 2: Overview of grouped-query method. Multi-head attention has H query, key, and value heads. Multi-query attention shares single key and value heads across all query heads. Grouped-query attention instead shares single key and value heads for each *group* of query heads, interpolating between multi-head and multi-query attention.

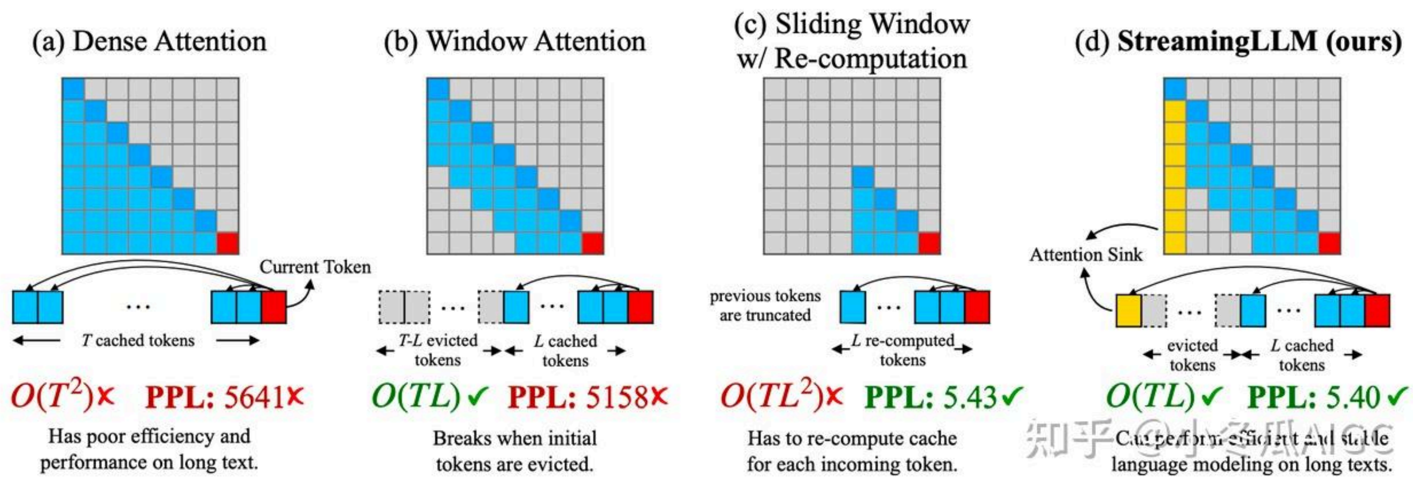
看下 `LLaMA` 的Cache存储实现, 提前开辟cache的存储空间

- `KV` 有单独的头数 `n_kv_heads`

```
# llama/llama/model.py
class Attention(nn.Module):
        self.wq = ColumnParallelLinear(
            args.dim,
            args.n_heads * self.head_dim,
        )
        self.wk = ColumnParallelLinear(
            args.dim,
            self.n_kv_heads * self.head_dim,  # 与Q的头数不一样
        )
        self.wv = ColumnParallelLinear(
            args.dim,
            self.n_kv_heads * self.head_dim,
        )
        self.cache_k = torch.zeros(
            (
                args.max_batch_size,
                args.max_seq_len,
                self.n_local_kv_heads,
                self.head_dim,
            )
        ).cuda()
        self.cache_v = torch.zeros(
            (
                args.max_batch_size,
                args.max_seq_len,
                self.n_local_kv_heads,
                self.head_dim,
            )
        ).cuda()
```

5.2 减少Length长度

- 减少 `KV-Cache` 的长度， base窗口： `Mistral-7B, StreamingLLM, LongLoRA`



以下是 `Mistral` 里采用循环队列

- 将 `Cache` 长度固定在 `sliding-window` 长度，借助 `RollingBuffer` ，不需要频繁“移位”

```
# mistral-src/mistral/cache.py
# ...
```

```python
class RotatingBufferCache:
    def __init__(self, n_layers: int, max_batch_size: int, sliding_window: int, n_kv_heads: int, head_dim: int):

        self.sliding_window = sliding_window
        self.n_kv_heads = n_kv_heads
        self.head_dim = head_dim

        self.cache_k = torch.empty((
            n_layers,
            max_batch_size,
            sliding_window, # 窗口大小的cache
            n_kv_heads,
            head_dim
        ))
        self.cache_v = torch.empty((
            n_layers,
            max_batch_size,
            sliding_window,
            n_kv_heads,
            head_dim
        ))
        # holds the valid length for each batch element in the cache
        self.kv_seqlens = None
```

- 另外 `StreamingLLM` 采用 `torch.tensor` `split/cat` 操作实现

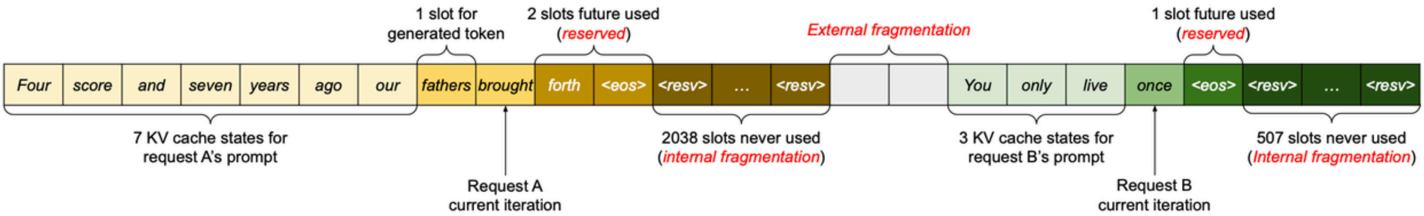
```python
# streaming-llm/streaming-llm/kv_cache.py
class StartRecentKVCache:
    ...
    def __call__(self, past_key_values):  # 实现sink kv cache
        ...
    def evict_for_space(self, past_key_values, num_coming):
        ...
    def evict_range(self, past_key_values, start, end):
        ...
```

- 在 `lit-LLama` 中采用" `rolling` "算子管理 `cache` 的更新

```python
# lit-llama/lit_llama/model.py
        if kv_cache is not None:
            cache_k, cache_v = kv_cache
            # check if reached token limit
            if input_pos[-1] >= max_seq_length:
                input_pos = torch.tensor(max_seq_length - 1, device=input_pos.device)
                # shift 1 position to the left
                cache_k = torch.roll(cache_k, -1, dims=2)
                cache_v = torch.roll(cache_v, -1, dims=2)
            k = cache_k.index_copy(2, input_pos, k)
            v = cache_v.index_copy(2, input_pos, v)
            kv_cache = k, v
```

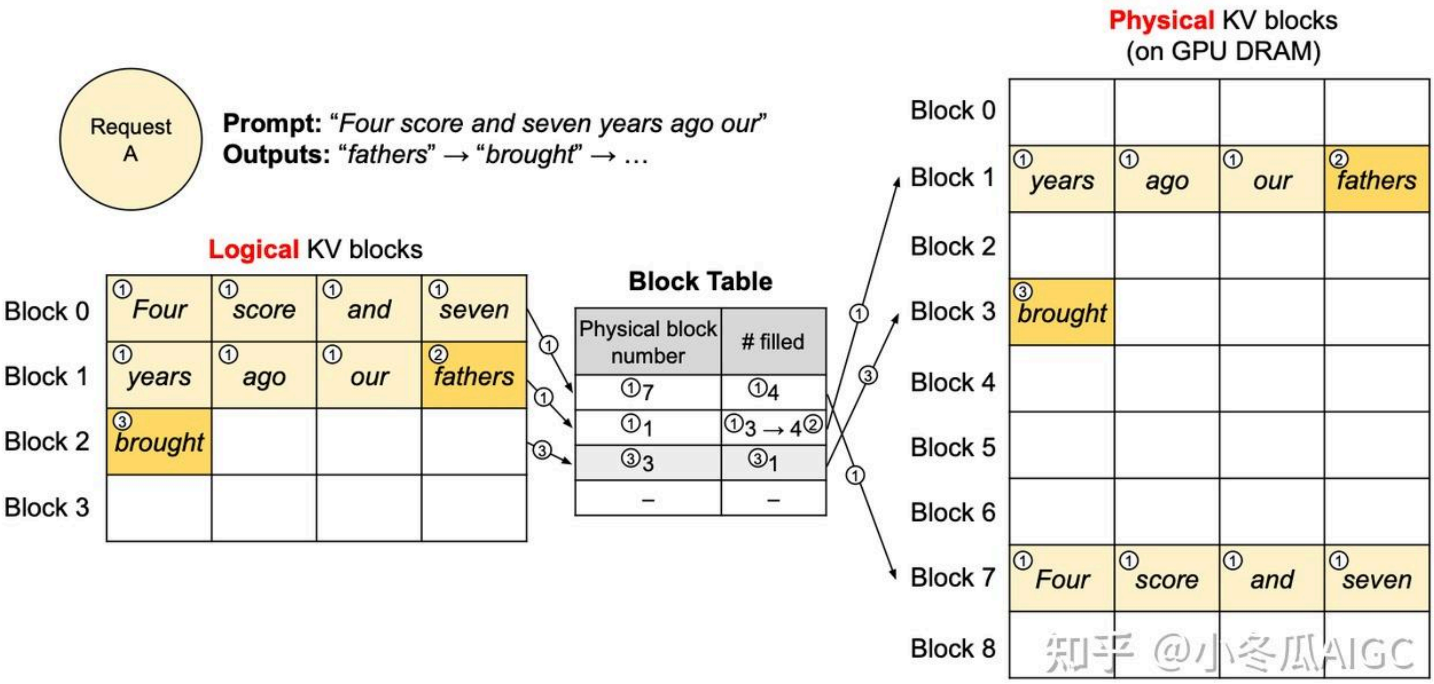
5.3 KV-Cache的管理，减少碎片

- 优化 `KV-Cache` 存储模型： `PagedAttention`
- 对于以下连续存储模型，如果将 `Page Length` 改成4
- `PagedAttention` 不是去改变 `Attention` 的计算，而是改变 `KV-cache` 的存取方式



Page1	Four	Score	and	Seven
Page2	years	ago	our	
Page3	you	only	live	
Page4				

1. 当完成分页后，消除External Fragmentation
2. 原来的2038+507的内部碎片，现在为1+1
3. 每个Page可以分散的存储在memory中
4. 而page内部的数据存储是连续的



`vLLM` 构建了 `CacheEngine` 先申请一块大的连续 `GPU` 存储，再自己统一做内存管理

```
# vllm/worker/cache_engine.py
# ...
class CacheEngine:
    def __init__(
        self,
        cache_config: CacheConfig,
        model_config: ModelConfig,
        parallel_config: ParallelConfig,
    ) -> None:
        # ...
        # Initialize the cache.
        self.gpu_cache = self.allocate_gpu_cache()
        self.cpu_cache = self.allocate_cpu_cache()
        # ...
```

5.4 减少bits数，量化模型

- `LLM-QAT` 在量化训练过程，将 `KV-Cache` 也做 `Quantization`
- 在部署时 `KV-Cache` 如果是 `16bit` ，量化成 `4bit` 的话，显存直接减少4倍

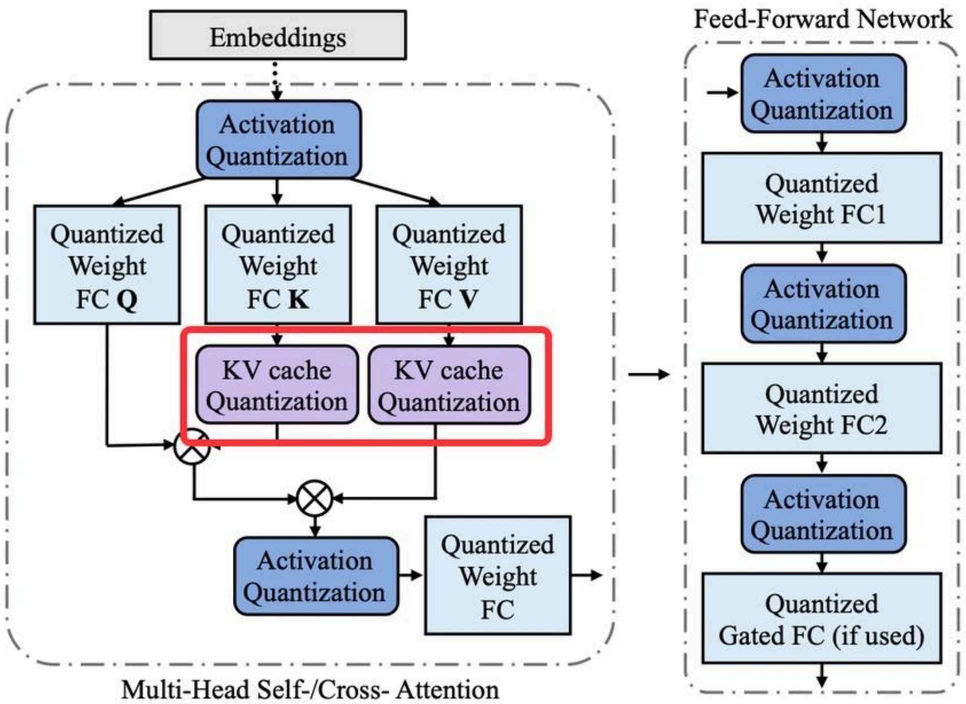


Figure 2: Overview of the quantized transformer in LLM-QAT. We quantize all the weights and input activations in fully-connected linear layers. The KV cache is also quantized if specified.

6 手撕KV-Cache

参照该讲解 【手撕LLM-KVCache】显存刺客的前世今生

手撕 `KV-Cache` 的推理过程, 可以直接.py或jupyter运行

- 该代码实现一个从 `embedding` -> `attention` -> `lm_head` 网络
- 该网络带 `KV-Cache`
- `generation` 可debug `KV-Cache shape`

```
# author: xiaodongguaAIGC
# KV-Cache + Generation + decoder

import torch
import torch.nn.functional as F
from transformers import LlamaModel, LlamaConfig, LlamaForCausalLM

D = 128 # single-head-dim
V = 64  # vocab_size

class xiaodonggua_kv_cache(torch.nn.Module):
    def __init__(self, D, V):
        super().__init__()
        self.D = D
        self.V = V
        self.Embedding = torch.nn.Embedding(V,D)
        self.Wq = torch.nn.Linear(D,D)
        self.Wk = torch.nn.Linear(D,D)
```

```
        self.Wv = torch.nn.Linear(D,D)
        self.lm_head = torch.nn.Linear(D,V) # LM_head
        self.cache_K = self.cache_V = None  # initial

    def forward(self,X):
        X = self.Embedding(X)
        Q,K,V = self.Wq(X),self.Wk(X),self.Wv(X)
        print("input_Q:", Q.shape)
        print("input_K:", K.shape)
        print("input_V:", V.shape)

        # Easy KV_Cache
        if self.cache_K == None:
            self.cache_K = K
            self.cache_V = V
        else:
            self.cache_K = torch.cat((self.cache_K, K), dim = 1)
            self.cache_V = torch.cat((self.cache_V, V), dim = 1)
            K = self.cache_K
            V = self.cache_V

        print("cache_K:", self.cache_K.shape)
        print("cache_V:", self.cache_K.shape)

        # ignore proj/MLP/scaled/mask/multi-head when calculate Attention
        attn =Q@K.transpose(1,2)@V

        # output
        output=self.lm_head(attn)
        return output

model = xiaodonggua_kv_cache(D,V)

# 创建数据、不使用tokenizer
X = torch.randint(0, 64, (1,10))
print(X.shape)

for i in range(4):
    print(f"\nGeneration {i} step input_shape: {X.shape}: ")
    output = model.forward(X)
    print(output.shape)
    next_token = torch.argmax(F.softmax(output, dim = -1),-1)[:,-1]
    print(next_token.shape)
    X = next_token.unsqueeze(0)
```

结果为

```
torch.Size([1, 10])

Generation 0 step input_shape: torch.Size([1, 10]):
input_Q: torch.Size([1, 10, 128])
input_K: torch.Size([1, 10, 128])
input_V: torch.Size([1, 10, 128])
cache_K: torch.Size([1, 10, 128])
cache_V: torch.Size([1, 10, 128])
torch.Size([1, 10, 64])
torch.Size([1])

Generation 1 step input_shape: torch.Size([1, 1]):
input_Q: torch.Size([1, 1, 128])
input_K: torch.Size([1, 1, 128])
input_V: torch.Size([1, 1, 128])
```

```
cache_K: torch.Size([1, 11, 128])
cache_V: torch.Size([1, 11, 128])
torch.Size([1, 1, 64])
torch.Size([1])

Generation 2 step input_shape: torch.Size([1, 1]):
input_Q: torch.Size([1, 1, 128])
input_K: torch.Size([1, 1, 128])
input_V: torch.Size([1, 1, 128])
cache_K: torch.Size([1, 12, 128])
cache_V: torch.Size([1, 12, 128])
torch.Size([1, 1, 64])
torch.Size([1])

Generation 3 step input_shape: torch.Size([1, 1]):
input_Q: torch.Size([1, 1, 128])
input_K: torch.Size([1, 1, 128])
input_V: torch.Size([1, 1, 128])
cache_K: torch.Size([1, 13, 128])
cache_V: torch.Size([1, 13, 128])
torch.Size([1, 1, 64])
torch.Size([1])
```

7. KV-Cache总结

- `KV-Cache` 的前身可以追溯到 `Transformer` 的 `Encoder` 出来的 `KV` 值，用于做 `cross-attention`
- 从 `Generate` 看出， `KV-Cache` 存在的必要性，此时能准确计算出 `KV-Cache` 的显存占用逻辑
- 在 `KV-Cache` 里优化有4点思路：减少长度，减少头数、减少 `bits` 数、增加 `cache` 的管理

Reference

- 本文摘自[高效Attention]系列：【手撕LLM-KVCache】显存刺客的前世今生
- LLaMA
- LLaMA2
- MQA
- GQA
- vLLM
- Mistral
- LongLoRA
- Streaming-LLM
- FlashAttention
- LLM-QAT

《手撕RLHF》 解析如何系统的来做LLM对齐工程

小冬瓜AIGC：【手撕RLHF-Safe RLHF】带着脚镣跳舞的PPO

小冬瓜AIGC：【手撕RLHF-Rejection Sampling】如何优雅的从SFT过渡到PPO

《手撕LLM》 系列文章+原创课程：LLM原理涵盖Pretrained/PEFT/RLHF/高性能计算

小冬瓜AIGC：【手撕LLM-QLoRA】NF4与双量化-源码解析

小冬瓜AIGC：【手撕LLM-RWKV】重塑RNN 效率完爆Transformer

小冬瓜AIGC：【手撕LLM-FlashAttention】从softmax说起，保姆级超长文！！

小冬瓜AIGC:【手撕LLM-Generation】Top-K+重复性惩罚

小冬瓜AIGC：【手撕LLM-KVCache】显存刺客的前世今生--文末含代码

小冬瓜AIGC：【手撕LLM-FlashAttention2】只因For循环优化的太美

《手撕Agent》从代码和工程角度，探索能够通向AGI的Agent方法

小冬瓜AIGC：【手撕Agent-ReAct】想清楚再行动、减轻LLM幻觉

> 我是小冬瓜AIGC，原创超长文知识分享，原创课程已帮助多名同学速成上岸LLM赛道：手撕LLM+RLHF
> 研究方向：LLM、RLHF、Safety、Alignment

编辑于 2024-05-23 01:22・广东

LLM 大模型 模型部署