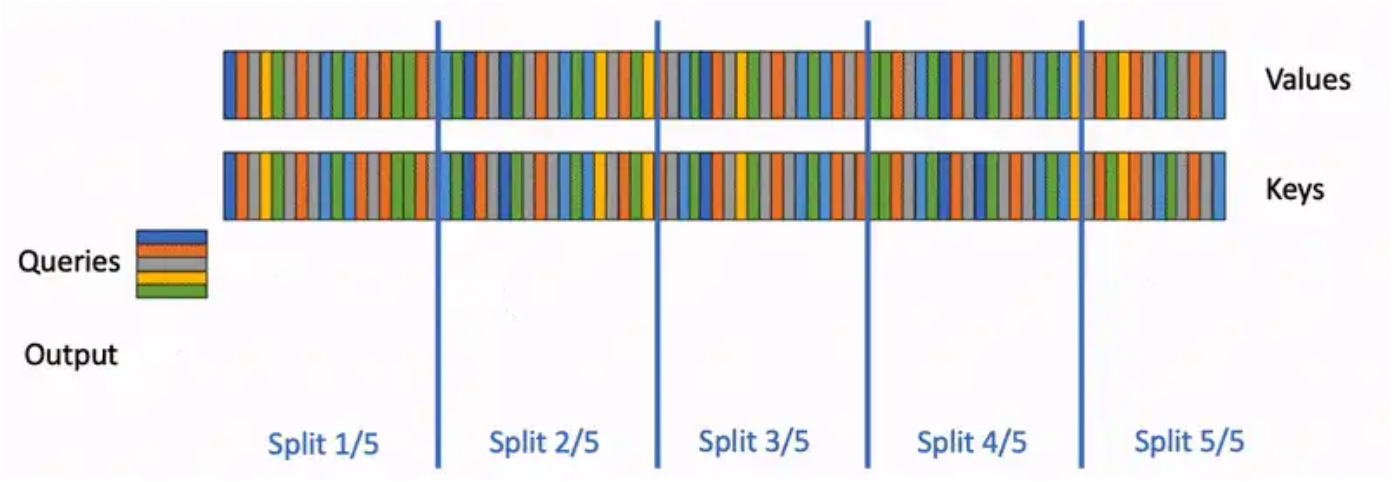
FlashAttenion-V3- Flash Decoding详解

目录

1. Motivation
2. Multi-head attention for decoding
3. A faster attention for decoding: Flash-Decoding
4. Benchmarks on CodeLlama 34B
5. Using Flash-Decoding
6. 个人总结



最新FlashDecoding++

Austin：【FlashAttention-V4，非官方】FlashDecoding++

Flash Attention V1和V2的作者又推出了Flash Decoding，真是太强了！

Flash-Decoding借鉴了FlashAttention的优点，将并行化维度扩展到keys/values序列长度。这种方法几乎不收序列长度影响（这对LLM模型能力很重要），可以充分利用GPU，即使在batch size较小时（inference特点），也可以极大提高了encoding速度。

相关背景知识先推荐阅读：

Austin：FlashAttention图解（如何加速Attention）

Austin：FlashAttention2详解（性能比FlashAttention提升200%）

Motivation

最近，像ChatGPT或Llama这样的LLM模型受到了空前的关注。然而，它们的运行成本却非常高昂。虽然单次回复的成本约为0.01美元（例如在AWS 8块A100上运行几秒钟），但是当扩展到数十亿用户的多次交互时，成本会迅速上升。而且一些场景的成本更高，例如代码自动补全，因为只要用户输入一个新字符就会执行。由于LLM应用非常广泛且还在迅速增长，即使稍微提升其运行效率也会产生巨大的收益。

LLM inference（或称为decoding）是一个迭代的过程：预测的tokens是逐个生成的。如果生成的句子有N个单词，那么模型需要进行N次forward。一个常用的优化技巧是KV Cache，该方法缓存了之前forward的一些中间结果，节约了大部分运算（如MatMul），但是attention操作是个例外。随着输出tokens长度增加，attention操作的复杂度也极具上升。

然而我们希望LLM能处理长上下文。增加了上下文长度，LLM可以输出更长的文档、跟踪更长的对话，甚至在编写代码之前处理整个代码库。例如，2022年大多数LLM的上下文长度最多为2k（如GPT-3），但现在LLM上下文长度可以扩展到32k（Llama-2-32k），甚至最近达到了100k（CodeLlama）。在这种情况下，attention操作在推理过程中占据了相当大的时间比例。此外，当batch size增加时，即使在相对较小的上下文中，attention操作也可能成为瓶颈。这是因为该操作需要对内存的访问会随着batch size增加而增加，而模型中其他操作只和模型大小相关。

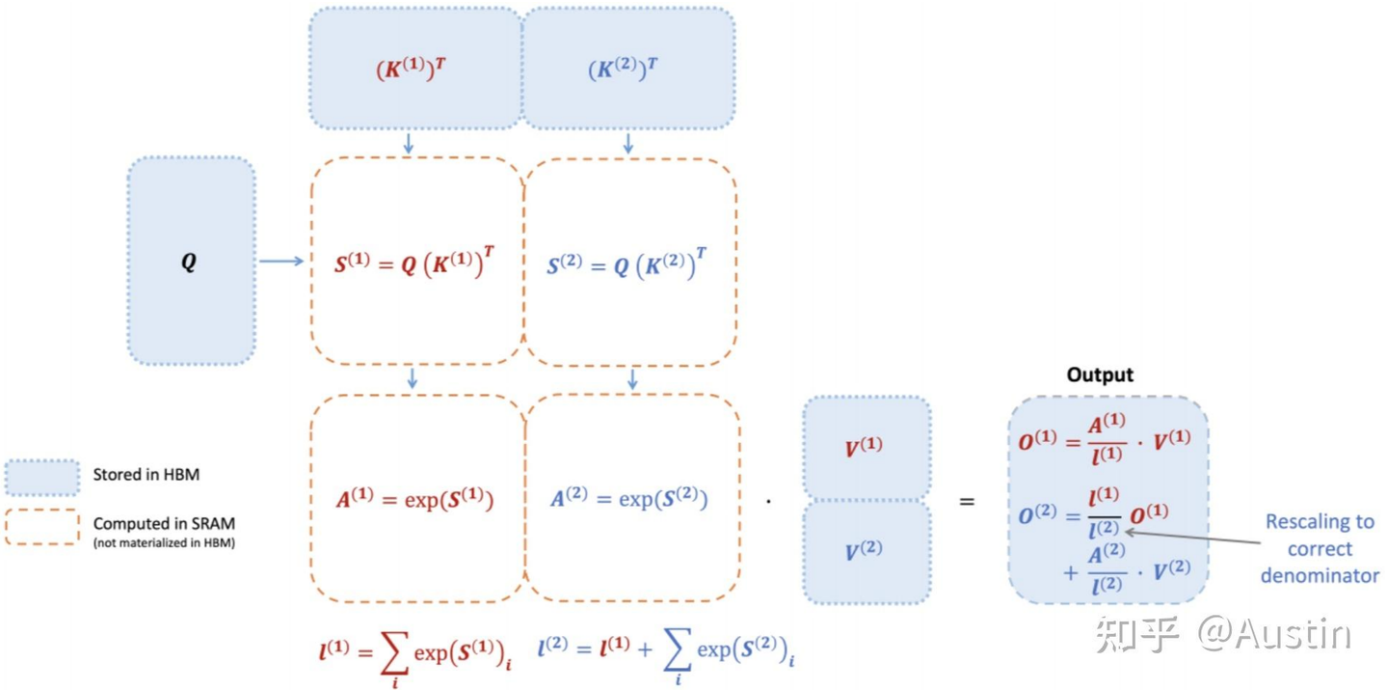
因此，本文提出了Flash-Decoding，可以推理过程中显著加速attention操作（例如长序列生成速度提高8倍）。其主要思想是最大化并行加载keys和values的效率，通过重新缩放组合得到正确结果。

Multi-head attention for decoding

在decoding过程中，每个生成的新token需要与先前的tokens合并后，才能继续执行attention操作，即$\mathrm{softmax}(Q \times K^T) \times V$。Attention操作在训练过程的瓶颈主要卡在访问内存读写中间结果（例如$Q \times K^T$）的带宽，相关加速方案可以参考FlashAttention和FlashAttention2。

然而，上述优化不适合直接应用于推理过程。因为在训练过程中，FlashAttention对batch size和query length进行了并行化加速。**而在推理过程中，query length通常为1，这意味着如果batch size小于GPU上的SM数量（例如A100上有108个SMs），那么整个计算过程只使用了GPU的一小部分！** 特别是当上下文较长时，通常会减小batch size来适应GPU内存。例如batch size = 1时，FlashAttention对GPU利用率小于1%！

下面展示了FlashAttention的计算示意图，该示例将keys和values分为了2个block：



FlashAttention示意图

对应的计算公式：

$$
\begin{aligned}
m^{(1)} &= \text{rowmax}\left(\mathbf{S}^{(1)}\right) \in \mathbb{R}^{B_r} \\
\ell^{(1)} &= \text{rowsum}\left(e^{\mathbf{S}^{(1)}-m^{(1)}}\right) \in \mathbb{R}^{B_r} \\
\tilde{\mathbf{P}}^{(1)} &= \text{diag}\left(\ell^{(1)}\right)^{-1} e^{\mathbf{S}^{(1)}-m^{(1)}} \in \mathbb{R}^{B_r \times B_c} \\
\mathbf{O}^{(1)} &= \tilde{\mathbf{P}}(1)\mathbf{V}^{(1)} = \text{diag}\left(\ell^{(1)}\right)^{-1} e^{\mathbf{S}^{(1)}-m^{(1)}}\mathbf{V}^{(1)} \in \mathbb{R}^{B_r \times d} \\
m^{(2)} &= \max\left(m^{(1)}, \text{rowmax}\left(\mathbf{S}^{(2)}\right)\right) = m \\
\ell^{(2)} &= e^{m^{(1)}-m^{(2)}}\ell^{(1)} + \text{rowsum}\left(e^{\mathbf{S}^{(2)}-m^{(2)}}\right) = \text{rowsum}\left(e^{\mathbf{S}^{(1)}-m}\right) + \text{rowsum}\left(e^{\mathbf{S}^{(2)}-m}\right) = \ell \\
\tilde{\mathbf{P}}^{(2)} &= \text{diag}\left(\ell^{(2)}\right)^{-1} e^{\mathbf{S}^{(2)}-m^{(2)}} \\
\mathbf{O}^{(2)} &= \text{diag}\left(\ell^{(1)}/\ell^{(2)}\right)^{-1}\mathbf{O}^{(1)} + \tilde{\mathbf{P}}^{(2)}\mathbf{V}^{(2)} = \text{diag}\left(\ell^{(2)}\right)^{-1} e^{s^{(1)}-m}\mathbf{V}^{(1)} + \text{diag}\left(\ell^{(2)}\right)^{-1} e^{s^{(2)}-m}\mathbf{V}^{(2)} = \mathbf{O}
\end{aligned}
$$

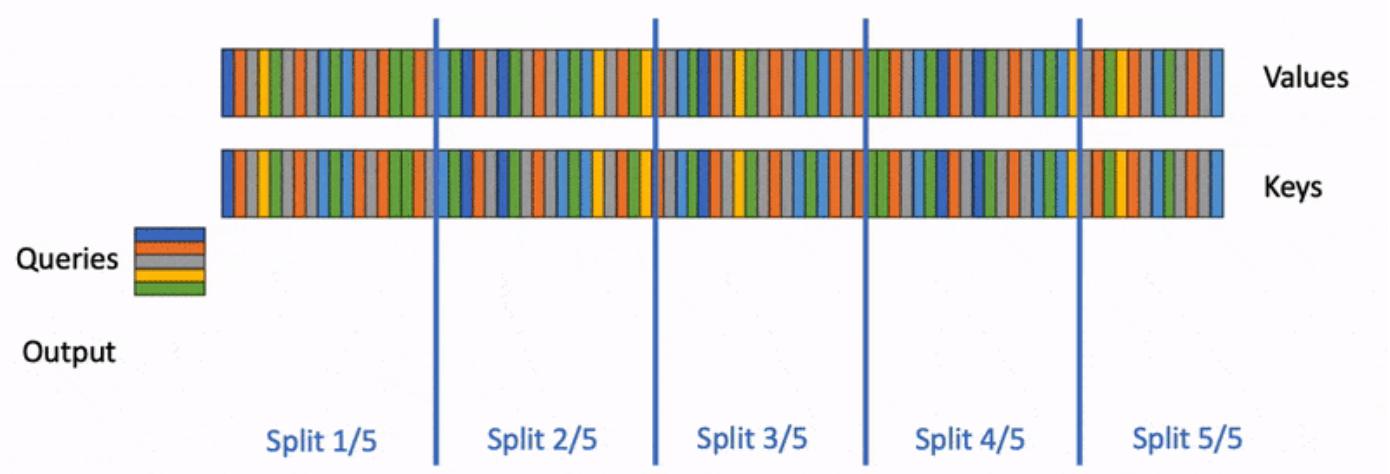
FlashAttention示意图对应的计算公式

注意$\mathbf{O}^{(2)}$的计算过程依赖$\mathbf{O}^{(1)}$，从下图也可以看出，FlashAttention是按顺序更新output的，其实当时我在看FlashAttention这篇文章时就觉得这个顺序操作可以优化的，因为反正都要rescale，不如最后统一rescale，没必要等之前block计算完（为了获取上一个block的max值）

flashattention计算过程

A faster attention for decoding: Flash-Decoding

上面提到FlashAttention对batch size和query length进行了并行化加速，**Flash-Decoding在此基础上增加了一个新的并行化维度：keys/values的序列长度** 。即使batch size很小，但只要上下文足够长，它就可以充分利用GPU。与FlashAttention类似，Flash-Decoding几乎不用额外存储大量数据到全局内存中，从而减少了内存开销。



flashdecoding计算过程

Flash Decoding主要包含以下三个步骤（可以结合上图来看）：

1. 将keys和values分成较小的block
2. **使用FlashAttention并行计算query与每个block的注意力（这是和FlashAttention最大的区别）** 。对于每个block的每行（因为一行是一个特征维度），Flash Decoding会额外记录attention values的log-sum-exp（标量值，用于第3步进行rescale）

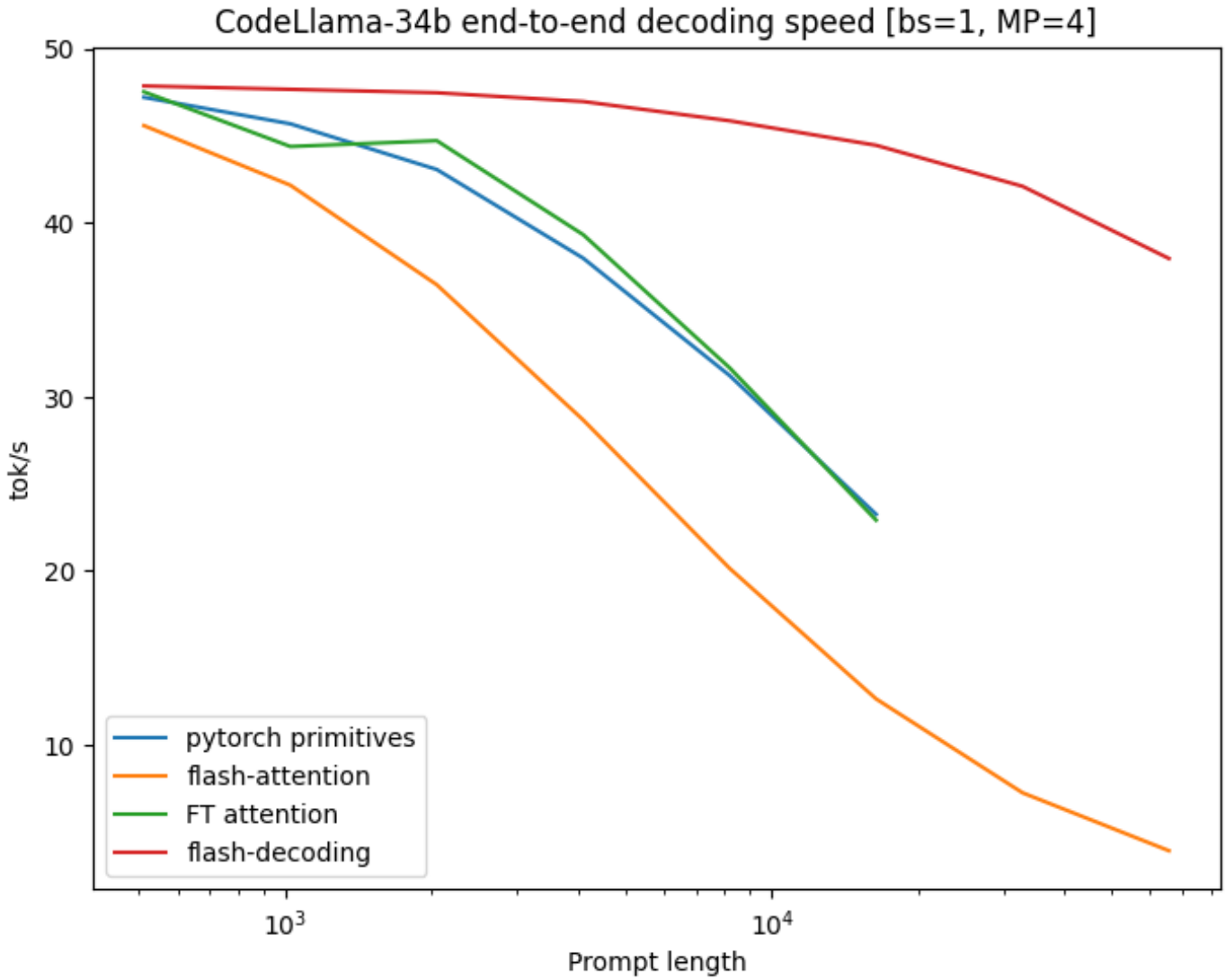
3. 对所有output blocks进行reduction得到最终的output，需要用log-sum-exp值来重新调整每个块的贡献

实际应用中，第1步中的数据分块不涉及GPU操作（因为不需要在物理上分开），只需要对第2步和第3步执行单独的kernels。虽然最终的reduction操作会引入一些额外的计算，但在总体上，Flash-Decoding通过增加并行化的方式取得了更高的效率。

Benchmarks on CodeLlama 34B

作者对CodeLLaMa-34b的decoding throughput进行了基准测试。该模型与Llama 2具有相同的架构。作者在各种序列长度（从512到64k）上测试了decoding速度，并比较了多种attention计算方法：

- PyTorch：使用纯PyTorch primitives运行注意力计算（不使用FlashAttention）。
- FlashAttention v2（v2.2之前的版本）。
- FasterTransformer：使用FasterTransformer attention kernel
- Flash-Decoding
- 将从内存中读取整个模型和KV Cache所需的时间作为上限



Untitled

从上图可以看出，Flash-Decoding在处理非常大的序列时速度可以提高8倍，**并且比其他方法具有更好的可扩展性** 。所有方法在处理small prompts时表现相似，但随着序列长度从512增加到64k，其他方法的性能都变差了，而Flash-Decoding对序列长度的增加并不敏感（下图也是很好的证明）

Setting \ Algorithm	PyTorch Eager	Flash-Attention v2.0.9	Flash-Decoding
B=256, seqlen=256	3058.6	390.5	63.4
B=128, seqlen=512	3151.4	366.3	67.7
B=64, seqlen=1024	3160.4	364.8	77.7
B=32, seqlen=2048	3158.3	352	58.5
B=16, seqlen=4096	3157	401.7	57
B=8, seqlen=8192	3173.1	529.2	56.4
B=4, seqlen=16384	3223	582.7	58.2
B=2, seqlen=32768	3224.1	1156.1	60.3
B=1, seqlen=65536	1335.6	2300.6	64.4
B=1, seqlen=131072	2664	4592.2	106.6

micro-benchmark on A100

Using Flash-Decoding

作者还通了Flash-Decoding使用方式：

1. 基于FlashAttention package，从版本2.2开始。
2. xFormers，在版本0.0.22中提供了`xformers.ops.memory_efficient_attention`模块

作者也提供了LLaMa v2/CodeLLaMa的repo1和xFormers repo2。此外，作者还提供了一个针对LLaMa v1/v2的最小示例。

个人总结

Flash-Decoding对LLM在GPU上inference进行了显著加速（尤其是batch size较小时），并且在处理长序列时具有更好的可扩展性。