再谈AI前端---挑挑LazyTensor的刺

https://zhuanlan.zhihu.com/p/392630428

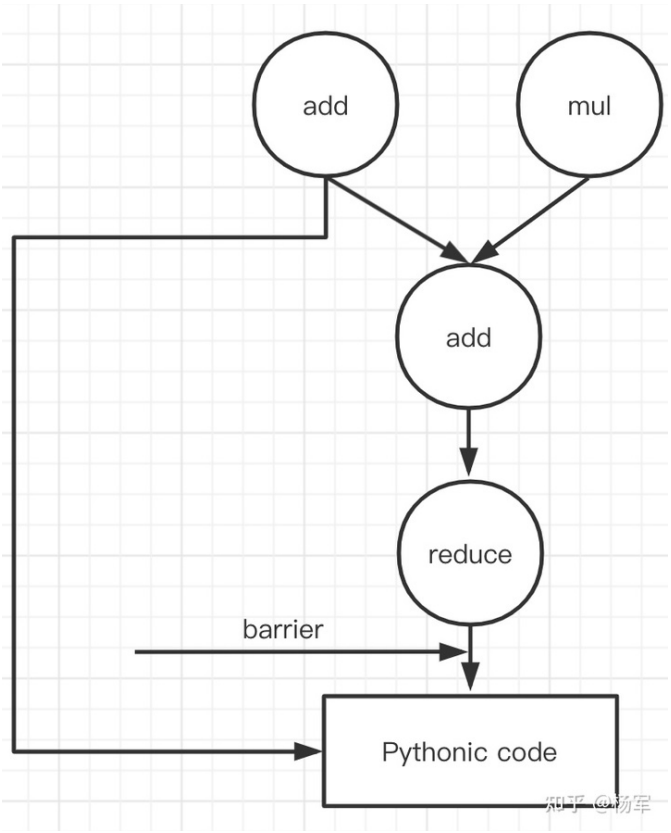
None Thu Sep, 09 00:42

这篇文章是上一篇文章的接续，继续讨论一下AI前端设计可能给后端对接带来的影响。

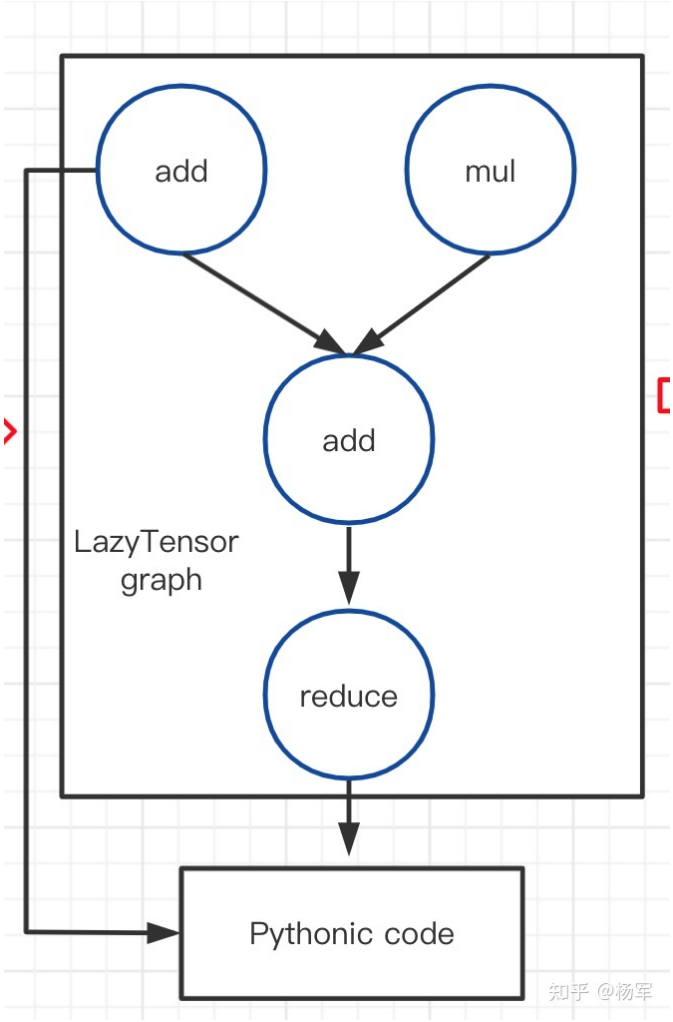
LazyTensor是一个比较有特色的工作。其核心idea如某位朋友所说，并不是新鲜事，在Minera和MxNet里都有类似的机制，只不过被Facebook和Google的同学用来给PyTorch和Swift TF对接XLA backend，有了更大一些的影响力。

晚上和一位朋友讨论LazyTensor这个工作的特点，最后收敛的结论是：

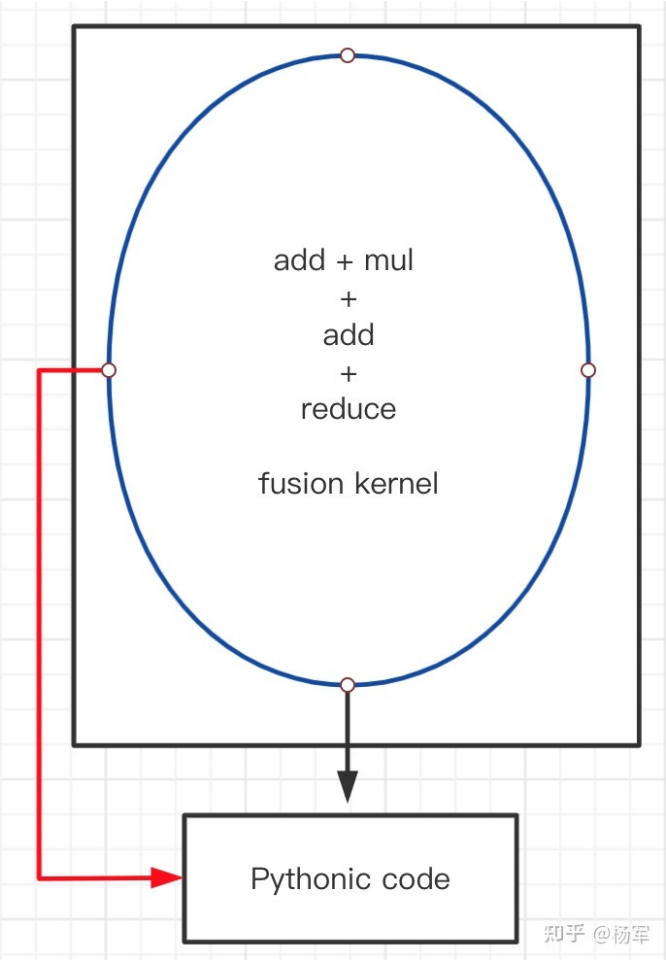
- LazyTensor本质上解决的是对接新硬件后端的有无问题，比如为PyTorch加入TPU的支持；
- LazyTensor在一定程度上能够带来优化收益，比如在GPU上通过LazyTensor的机制接入XLA backend，可以享受到kernel fusion的收益。甚至通过LazyTensor的机制想去对接TVM，享受TVM对一些计算密集算子的specialized codegen tuning收益也是符合其设计原则的。
- LazyTensor同时也引入了潜在的performance overhead。其核心是可能会影响到eager mode下，kernel发射和kernel执行overlap的异步程度。举个例子。下面是原始的一个PyTorch的执行示意图：



在**reduce**结点和**Pythonic code**结点之间就是LazyTensor的barrier，即LazyTensor的执行机制走到这个地方，会把计算图生成出来，喂给后端的AI编译backend，触发JIT编译过程。编译的标的如下图黑框里圈出来的子图。



编译完成之后，大体会产出这样的结果：



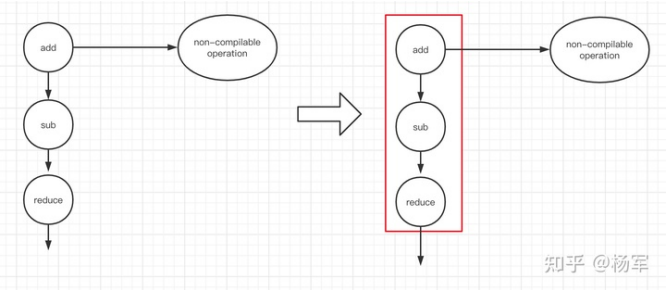
把这三张图放在一起对比就会注意到，**在Pythonic code和add这个操作之间存在一条data dependency，不过这条data dependency的通路足够的长，所以通常情况下是可以被通路中间执行的算子的耗时给摊平掉。但是在LazyTensor里，因为LazyTensor圈出的计算子图在实际触发JIT编译之前，是不会被执行的，所以*add-->Pythonic code*之间的data dependency只能被delay到计算子图完成编译，生成kernel，并完成kernel执行，结果写回之后才能resolve Pythonic code操作的data dependency，这就impede了eager mode下原本的kernel发射和执行可以overlap的异步性，带来了性能penalty。**

如果有JIT编译cache，能够解决这个问题么？

答案是不完全能。其原因是LazyTensor的机制只有在看到barrier结点，能够把可编译子图cluster出来以后，才可能去检索编译cache看是否可能复用上一轮编译的结果，而这已然错失了一些时机了。

同时，在AI编译后端，往往会推崇比较aggressive的fusion优化，fusion优化仅从编译子图的角度，能够带来性能收益，**但目前主流的AI编译框架里，kernel fusion都会带来一个side effect，fused kernel下游的消费者，只有等到这个fused kernel完整执行完以后，才能够resolve自己所需的data dependency**。某些场景下，这就可能会破坏了操作之间并行执行的可能性。举个例子，在下

面的示例里，左图里add结点执行完，**non-compilable operation**就可以往下执行了，当我们把**add**, **sub**, **reduce**这三个结点fuse成一个kernel以后，就需要等到这个fused kernel完整执行完以后，**non-compilable operation**才能够继续执行。在分布式场景里，fusion和分布式的计算通信overlap的矛盾也是这方面的一个典型例子。



所以LazyTensor其实在解决了一些问题的同时，也引入了一些新的问题，简单总结如下：

- LazyTensor的'**Lazy mode**'影响到了Eager mode原本kernel发射和kernel执行overlap的异步程度；
- LazyTensor对接的AI编译后端，在带来fusion收益的同时，也可能会因为fusion kernel执行的原子特性，影响到下游结点kernel发射的速率。

而LazyTensor会引入这些问题，其实也是有其trade-off考虑的。我们可以稍微列一下PyTorch对接AI编译后端的几种作法。

Trace、**Script**以及**LazyTensor**。

Script理论上说最干净，但麻烦在于总会遇到导出script不顺利的模型，因为AI编译后端支持的IR体系往往不能cover host侧Python建模的语法全集，这会限制其可用性。

Trace比较简洁，根据主动提供的输入Tensor，沿着执行踪迹生成一张计算图，但麻烦在于不能很好地cover dynamic control flow的问题，存在语义不正确的风险。

而**LazyTensor**在设计原理上正好能够解决Script和Trace在可用性方面的limitation。一位朋友说**LazyTensor**有些像是一种**被动的，adaptive的trace机制**，我觉得这也是一个形象的概括。

针对这些问题，有没有更好的解法呢？随便brain-storming一下：

- 设计更好的建模前端，从后端优化角度让建模行为对后端优化更友好。Jax算是这方面的尝试。不过这条路涉及到复杂的生态问题，周期很长。以及有些垂直属性的AI业务公司，会通过组织手段约束其建模团队的模型写法，来适配AI框架平台的优化布署需求，也算是某种形式的“建模前端”的改进，可以类比于Verilog的可综合语法子集的约束。
- 对后端实现进行改造。比如前面提到的大尺度fusion影响下游kernel发射速率的问题，是可能通过编译器本体，运行时的改良来进行解决的。在Google的TFRT的项目里也看到了解

决这个问题的一些思考尝试。但是对于LazyTensor的'Lazy mode'，除了建模前端的改进以外，目前我还没有想到更好的解法，也欢迎和同行们一起讨论碰撞。

发布于 07-24

文章被以下专栏收录