AI编译优化--工程篇：编译解耦及全量打开

https://zhuanlan.zhihu.com/p/305539613

None Mon Jun, 14 16:00

作者：PAI团队

背景

本文主要是介绍阿里PAI团队的AICompiler在工程落地方面的一些工作，包括推动AICompiler和Tensorflow解耦以及在PAI各种产品上的全量打开覆盖。团队更多在AICompiler方向的过往工作请参考总纲：

https://zhuanlan.zhihu.com/p/163717035

AICompiler最开始是基于Tensorflow中的XLA进行深度定制和拓展，因此和特定的tensorflow版本绑定的比较紧，这点也给我们在对接不同形式的业务时带来不少的挑战，比如：

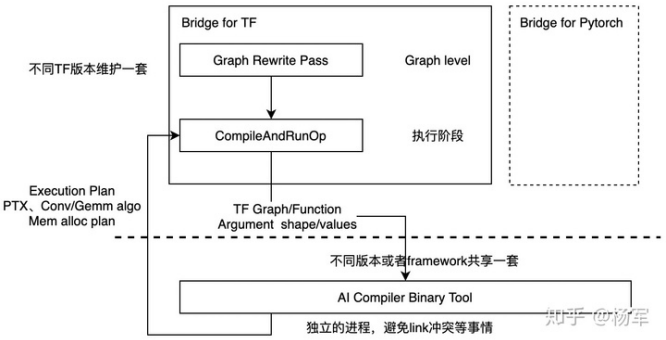
- 用户使用了更高或更低版本的TF完成模型的建模，为了使用AICompiler需要迁移模型实现；
- 用户自己基于开源TF进行了定制，希望在此基础上叠加AICompiler的优化能力；

为了解决这些问题，我们推动了AICompiler和tensorflow解耦。完成解耦之后的AICompiler支持以**插件**的形式嵌入到用户**预装**的tensorflow中，在无需修改tensorflow源码及重编译的情况下，便可以向用户提供通用透明的性能优化能力，大幅降低了部署应用AICompiler的门槛，提高了覆盖的产品场景和业务范围。我们将在第二章详细介绍我们解耦的工程方案。

在解耦的基础上，我们进一步推动了AICompiler在生产集群（含训练及推理作业）上的默认打开。全量打开，一方面可以进一步降低用户使用的门槛，另一方面也对AICompiler的稳定性和鲁棒性提出了更高的要求。举个例子，Google从2017年初推出XLA以来，到目前为止也还没有成为一个默认打开的feature，这主要是因为AI作业的多样性以及编译自身的复杂性导致可能有bad case。为了解决这个问题，我们设计了一套系统化的兜底机制，保证了在上线之后覆盖的**数万作业中基本零故障**。我们将在第三章中介绍我们配合AICompiler全量打开的系统化兜底机制的设计。

插件化设计的解耦方案

为了实现和TensorFlow的解耦，我们将AICompiler整体划分成了如下图所示的两个层次：AICompiler Binary Tool 和 AICompiler Bridge Layer。



AICompiler Binary Tool。Binary tool以一个预编译的二进制工具存在，是AICompiler的主体，负责接收特定格式的计算图，完成编译优化并输出优化之后的结果。该binary tool本身不依赖于tensorflow或者pytorch，相对中立，可以在不同的Tensorflow版本乃至不同的框架间共享。

AICompiler Bridge Layer。Bridge layer主要负责圈定并导出可编译的计算子图，调用AICompiler Binary Tool完成编译优化，并解释执行binary tool输出的结果。Bridge layer需要和宿主（TF，PyTorch）直接交互，因此需要对不同的版本或者框架做适配，但Bridge layer本身并不负责优化，整体相对轻量，所以维护的成本相比整个AI Compiler做多版本适配的情况显著降低。

通过框架接入层和主体优化层的拆分，AICompiler的主体功能将独立于特定的框架或者版本，可以按照自己的节奏进行迭代，同时又可以灵活的适配不同的使用场景。

全量打开的系统化机制

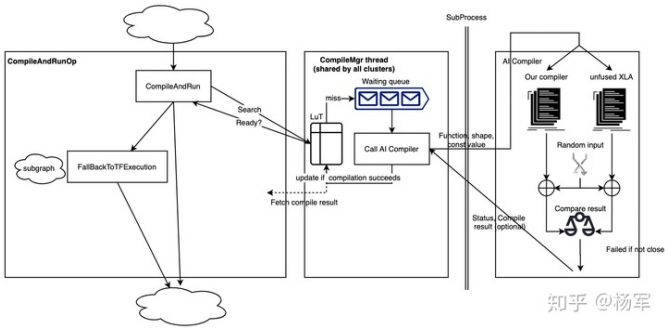
受限于编译本身的复杂性，目前AICompiler在稳定性及性能等方面还存在bad case。一方面我们会持续的改进编译本身，提升稳定性及性能；另一方面，结合一定的系统保护机制，我们可以尽量规避各种可能的稳定性和性能方面的问题，从而可以达到全面打开的效果。通过全量打开我们可以更加充分的评估编译本身的效果，同时收集到的反馈也可以更好的指导编译方面下一步优化的方向。

系统化兜底机制的主要目标为：

1. 原来可以正常运行的任务，在打开编译优化后，不能有性能/正确性方面的问题；
2. 原本在打开编译优化有加速效果的任务，不会因为增加系统化兜底机制之后而加速效果减弱；

这些目标中第一条强调的是鲁棒性，也即“兜的住”，第二条强调的是轻量化，也即系统兜底本身的开销越小越好。

为了实现这些目标，我们设计了如下图所示的机制。主体上可以分为回退模块，异步编译模块，沙箱校验模块。



回退模块。该模块的核心目标是**确保检查到编译存在稳定性/正确性/性能regression等问题之后可以平滑的回退到TF进行解释执行。**具体做法是在圈定可编译子图的阶段，保留一份可编译子图对应的原图，并在需要回退的情况下通过调用tf function的方式执行原始图。

异步编译模块。在原本XLA的实现中编译是一个阻塞动作，也即在编译的过程中对应的cluster不会进行计算，导致编译时间体现在端到端执行时间上。对于一些端到端执行时间偏短的任务，即使编译本身之后的计算是有加速的，但考虑到编译时间之后的端到端时间上反而可能是负优化。为了解决这类问题，我们引入了异步编译的机制，也即**在编译还没有完成之前，通过回退到TF进行执行，掩盖编译本身的开销，在编译完成之后再切换到编译后的版本执行**。

沙箱校验模块。在解耦之后，编译的过程是在子进程中完成，因此天然的就规避了编译core dump的问题，但仍然存在正确性和性能方面的校验的问题。

- **正确性校验**。在子进程编译的过程中，对所有fused codegen kernel做正确性校验，当计算误差在预定义范围之外时触发回退。在fused kernel级别进行的正确性校验一方面是为了控制校验本身的开销，另一方也是因为非fuse kernel的实现种类相对固定，不存在长尾现象，日常的测试足够覆盖。
- **性能校验。**在子进程编译成功之后（包括功能和正确性验证已经通过），我们在CompileAndRunOp层面还引入了一层cluster粒度的性能校验模块，通过和TF baseline的计时比较，决定是否选择启用编译的版本。

沙箱校验模块在实现上还需要满足很多工程约束：

- **显存约束。**编译在子进程中完成，在正确性校验的过程中需要进行真实的计算，为了避免和主进程争抢显存资源，我们使用了虚拟显存的技术，确保子进程消耗尽可能少的物理显存（一个cuda context的开销）；
- **正确性校验ground truth选取问题**。如果选择cpu的单线程的版本作为ground truth，实现上固然可行，但却导致正确性校验的时间大幅拉长，不符合轻量化的目标。经过各种权衡，我们选择了unfused版本计算的结果作为baseline，兼顾时间开销和正确性验证有效性；
- **正确性校验误差标准**。误差标准定义的过高会导致大面积的编译失败，导致原本编译优化可以生效的任务没有了加速，反之如果正确性标准定义的过低，则存在漏检的问题。为此

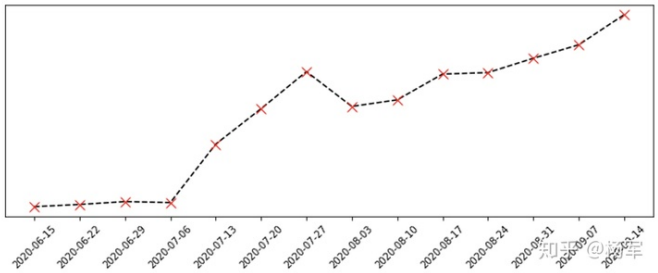
我们采用数据驱动的方式，通过收集大量真实的误差数据，综合考虑绝对误差/相对误差/超过误差范围元素的比例来定义一个误差判断标准；

- **平滑的纠错机制。**当一个fused kernel的误差超标时，简单的回退整个编译子图会导致我们对于局部的错误容忍度很低，最终会让编译可以生效的比例大幅降低。为此当fused kernel的正确性不达标时，我们首先采用自我纠偏（也即回退到unfused版本），而不是直接回退整个cluster。
- **测试激励**。在正确性验证时输入数据我们目前采用了随机生成的方式，这种方式存在bad case （比如index类型的输入，随机产生的值可能会越界），但整体上比例不高，且得益于平滑的纠错机制，局部假阳性不影响大局。

部署效果

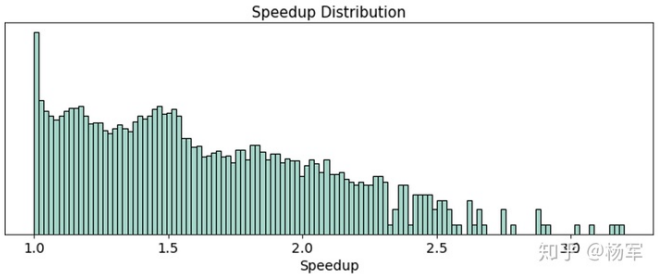
采用以上系统化兜底设计的AICompiler已经在多个PAI的产品中全量上线，测试的结果验证了这套方案的有效性。

以下是我们在一个大规模生产集群中灰度上线之后的统计数据（灰度比例逐步增加，9月底到达全量），截止9月25号累计覆盖超过几万个生产作业，无明显bad case。



单机作业每周调用量数据统计

(考虑数据脱敏，这里只给出了作业数的时间增长趋势，未给出作业详细数字)



所有作业的端到端加速情况统计

（ 考虑数据脱敏，这里只给出了作业加速比的相对作业数，未给出详细作业数字）

总结

通过解耦和全量打开中开发的系统化兜底机制，我们为AICompiler奠定了相对扎实的工程基础，初步打通了新feature开发，上线验证，持续优化迭代的闭环，为编译优化这种相对长期的工作赢得更多的时间。

目前我们正在持续推进以MLIR为基础，兼容多框架（Tensorflow/PyTorch），支持多Device（GPU/CPU/ASIC），多种优化手段融合（静态shape/动态shape）的AICompiler框架建设，欢迎感兴趣的同学加入我们一起共建，通过编译的手段为构建一流的AI infra而努力。感兴趣的同学欢迎邮件联系 muzhuo.yj@alibaba-inc.com。