AI编译优化--业务实践

作者：PAI团队

> *进入正题前，还是先打个招聘小广告，欢迎对我们工作感兴趣的同学联系我们，细节参见这里，可以直接邮件muzhuo.yj@alibaba-inc.com。*

本文是AI编译优化系列连载的第四篇，总纲请移步：

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

在之前的文章中，我们介绍了PAI团队在计算密集算子方面的优化工作。

在这篇文章里，我们会介绍一个AI编译优化技术在阿里内部实际业务场景的落地case。AI编译技术在业务落地的过程中，不仅完善了自身的完备性和优化效果，同时在和业务结合的过程中，加深了我们对于线上环境复杂度的认知，理解了分布式训练中带来的各种挑战和问题，在过程中也积累了一些获取最优模型迭代性能的best practice。

机器翻译模型训练是我们的一个典型的落地场景，在这个场景中，我们将访存密集算子优化+计算密集算子优化+分布式+IO优化多项技术进行了结合，形成了组合优化的效果，帮助业务方提升了模型迭代的效率，**模型收敛时间由最初的三天半(85h)训练时间下降到了一天(25h)左右**，同时结合落地过程中业务方同学的使用反馈我们也进行了性能和可用性的改进。

模型介绍

业务方使用的Alitranx-Transformer2.0翻译质量大幅提升，追平了目前世界上最好的开源NMT系统Marian。是业务方结合Transformer模型和阿里特有的电商场景，进行了大量模型创新之后的模型结构(部分工作也被AI顶会工作接受)。使用这个模型，在WMT2018 国际机器翻译大赛上，阿里机器翻译团队共提交5项结果，并全数获得冠军，Transformer-2.0模型在其中功不可没。

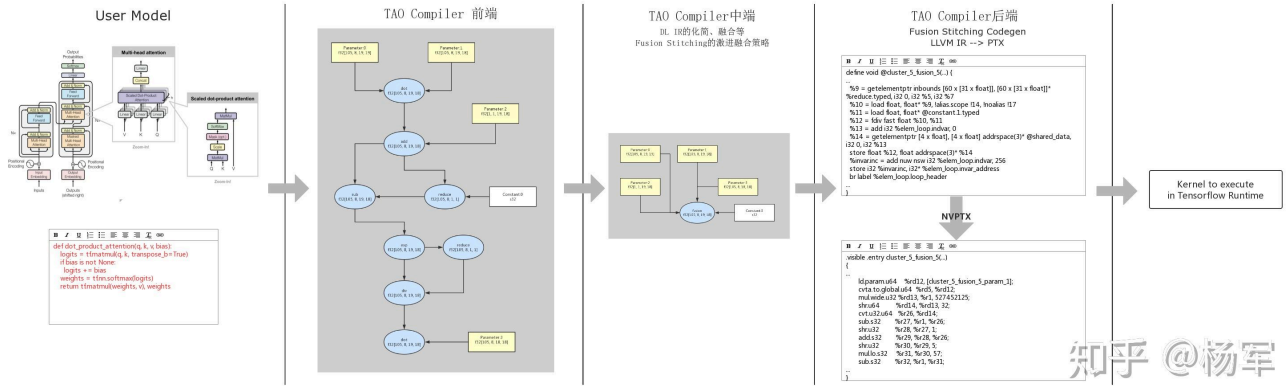
AI编译技术应用介绍

针对业务方的Transformer 2.0模型的训练优化需求，我们进行了访存密集算子优化，计算密集算子优化，以及和TF社区的Distribution Strategy自动分布式工作配合的联合优化，在下面会逐一进行介绍。

访存密集算子优化

宏观来看，AI编译器与其他编译器架构没有本质区别，分别由前端、中端、后端三个部分构成。前端负责完成TensorFlow Op计算子图到DL IR子图的翻译转换工作，中端会在DL IR子图层面执行一系列图优化动作，包括逻辑化简、以及IR融合等等，后端会进行Codegen以及可执行码的构建。

下图中我们以业务方使用的Transformer模型中的基本组成结构，*DotProductAttention*为例，具体说明AI编译器的整个工作流程。在我们获取了用户的构图代码之后，前端会将原始Op构图描述(*DotProductAttention*主要为 *matmul*，*bias add*和*softmax*op)，转换成更细粒度的DL IR表示(下图中可以看到包含了细粒度的*dot*，*add*，*reduce*等指令)，经过中端将这些DL IR表示进行融合，可以看到融合之后DL IR指令数目大大减少，最后由后端生成优化后的高效fusion kernel，由TensorFlow的Runtime调用执行。



首先介绍我们研发的大尺度fusion codegen框架Fusion Stitching。

Fusion Stitching

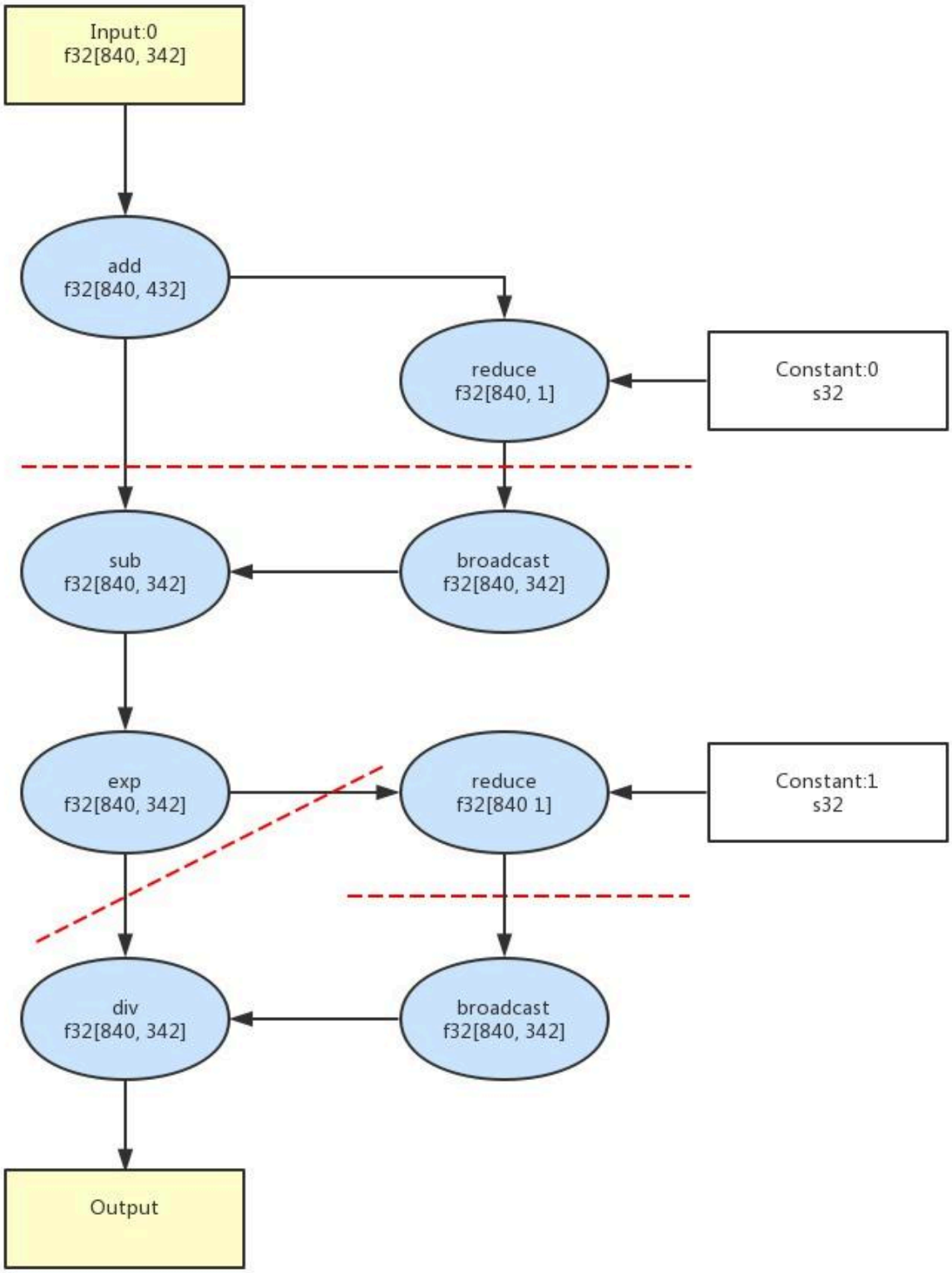
我们针对NVIDIA GPU硬件平台，原创性地提出了通过片上共享内存进行数据中转从而大幅提升编译图融合力度的Fusion Stitching codegen框架，在编译器中进行了大量的精细优化设计，详细技术细节参见这里。

这个框架会同时涉及到编译器中端和后端的动作。中端负责激进力度的计算子图融合，后端基于融合后的子图完成对应的代码生成动作。

编译器中端改进

Fusion Stitching不再受限于OpFusion CodeGen的模版的设计理念，在Op Fusion的颗粒度上采用了一种比较激进的策略，*只要是有生产消费关系的且类型被支持的节点即可fuse在一起*。

我们依然以业务方机器翻译模型使用的 *DotProductionAttention*为例，下图为其中的 *softmax* 前向计算部分的DL IR表示，使用社区XLA的编译优化，这部分将产生 *4* 个Kernel，而基于我们的Stitching Fusion Pass，这个计算图可以全部合并为 **1** 个Kernel,从而可以显著节省包括kernel launch以及显存访问的性能开销。



编译器后端Codegen

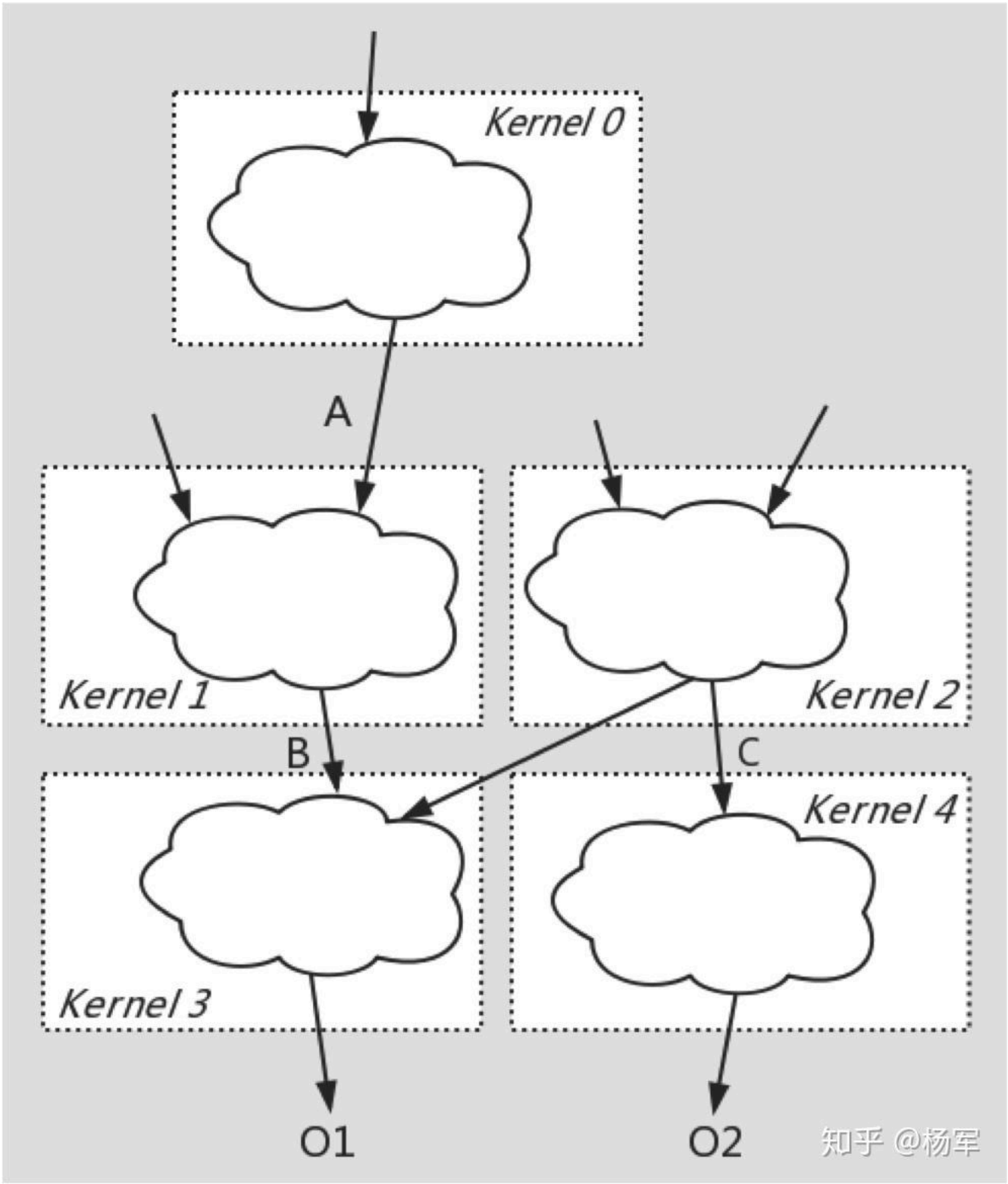
社区XLA CodeGen对每个Codegen出的Kernel基于一个单一的Parallel Loop模板，即每个cuda thread处理tensor中的一个element，这种简单的模板在处理线性连接的elementwise计算图时能够

解决问题，但当计算图的连接关系变复杂，同时计算图中出现Reduce / Broadcast等复杂计算节点时，在实际业务中效果并不好，特别是对于包含了后向处理部分的训练过程计算图更是如此。

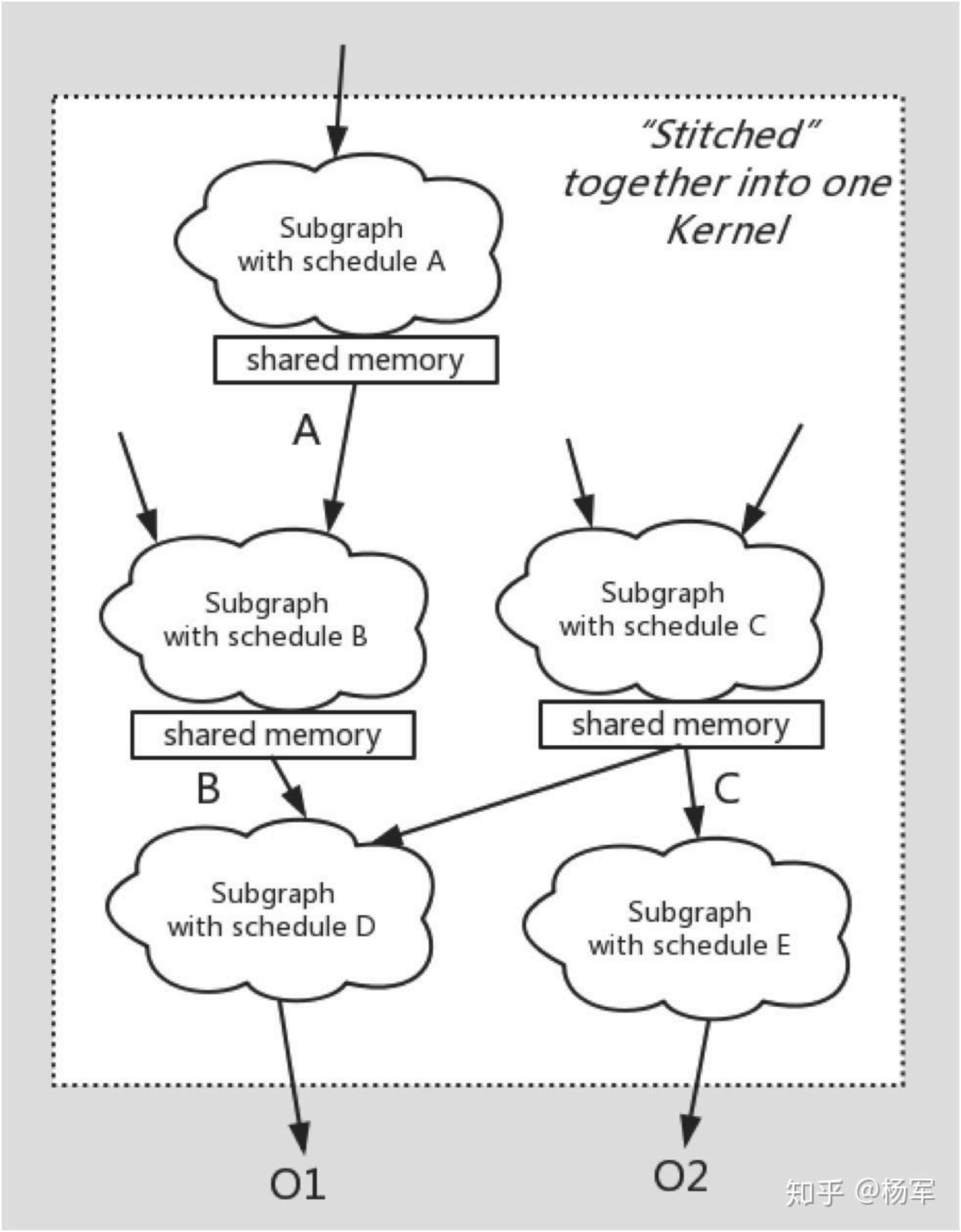
而我们的工作将一个Kernel的计算图通过shared memory做隔离划分为多个子图，每个部分为一个独立计算的Parallel Loop，可以根据实际shape需要决定自己的schedule（即如何为每个cuda thread划分工作量）。

下面的两张图分别为社区XLA Codegen的情况和FusionStitching Codegen的情况，可以看到社区的fusion做法中没有涉及到shared memory的使用，而我们的codegen将社区做法中难以fuse在一起的计算 *通过低访存开销的shared memory作为桥接，将多个Kernel缝合在一起*，这就是FusionStitching的含义来源。

社区XLA Codegen的情况



Fusion Stitching CodeGen的情况如下：



其他优化

在整个研发过程中，我们还进行了对前端的优化，解决了不合理的编译子图划分引发不必要的memcpy，同时我们根据实际业务场景中遇到的需求，还提供了更完备的Op支持。在中端，我

们完善了由于过于激进的fusion策略可能带来的潜在的可用性和性能问题。在后端，我们通过对于codegen指令优化以及根据硬件和kernel计算自适应的调节launch dimension的改进，都在真实的workload中拿到了可观的收益。

自动混合精度

自动混合精度可以使得用户无需经过繁琐的模型改写即可实现混合精度训练、充分利用NVIDIA V100 GPU TensorCore来优化计算密集型kernel（比如Transformer模型中大量存在的矩阵乘计算）的性能、提升模型迭代速度。而灵活易用的loss scale接口，使得用户可以在使用混合精度进行训练的过程中方便的对可能存在的精度问题进行控制，可以做到和fp32训练出的模型精度无损。同时，自动混合精度训练出来的模型和全精度之前可以无缝的进行restore加载，进行finetune。

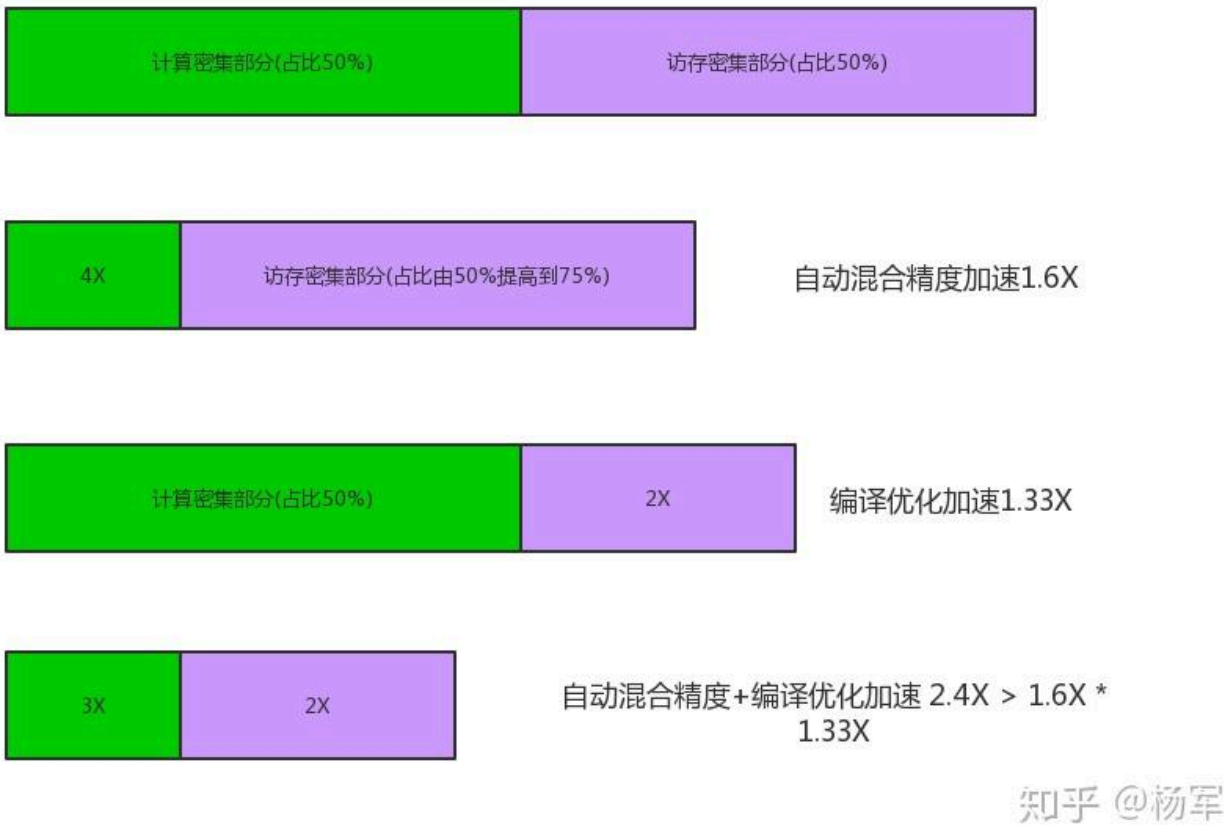
混合精度使用Best Practice

业务方在新的Transformer 2.0的训练中已经使用了自动混合精度的功能。但是取得的加速比并不明显，经过profiling发现，并不是所有的fp16的GEMM计算都使用了TensorCore的高效实现，这是由于TensorCore计算kernel实现的一个限制，要求在一个矩阵计算中，其两个输入矩阵的尺寸M、N、K都是8的整数倍时才会启用TensorCore的kernel。

在用户的模型里面中一般隐层的数目都是类似于 *512/1024* 这样是8的倍数的magic number，已经满足是8的整数倍的要求。根据Transformer模型的特性，矩阵乘计算中尺寸的MNK有一个值或者两个值就是隐层数目的大小，而另外的值的大小都是和计算中输入的batch size相关的。基于模型的这个特性，我们提供了一个在用户侧轻巧修改的best practice方法，控制输入的batch size是8的整数倍（如下代码所示，用户使用的TensorFlow中的***GroupByWindowDataset***，我们修改了其中控制batch size的window_size_func），从而保证了矩阵乘计算能够使用TensorCore充分加速。混合精度的部分加速比由之前的**1.18X**提升到了**1.48X**。

```
def window_size_func(key):
  key += 1  # For bucket_width == 1, key 0 is unassigned.
  size = (num_gpus * batch_size_words // (key * bucket_width))
  size = size - size % (8 * num_gpus)
  return tf.to_int64(size)
```

混合精度和访存密集算子优化的结合



在上图中，我们将GPU计算分成计算密集型的部分和访存密集部分，在Transformer模型中，计算密集型部分占比约为**50%**，主要是矩阵乘计算，这个计算部分可以通过自动混合精度的使用进行加速，剩余的访存密集型部分则由Fusion Stitching进行优化。我们可以看到，单独自动混合精度的性能提升是**1.48倍**，单独Fusion Stitching优化的提升是**1.33倍**，而我们同时使用两种优化手段，得到了**2.4倍**左右的性能提升，**取得了1+1>2的组合优化效果**。一方面是由于使用了混合精度之后，提升了访存密集计算的比例，扩展了编译优化的可优化空间；同时在进行自动混合精度过程中，为了控制精度而插入的cast节点和loss scale等计算，也可以被fusion优化，消除了这些节点带来的额外开销。

自动分布式

业务方的Transfromer-2.0模型使用了TF社区提供的MirroredStrategy，便捷高效的实现了单机8卡的训练(除了在社区已有的Distribution Strategy进行完善强化之外，PAI团队在分布式方面也开展了大量的自研性质的工作，后续也会在不同渠道有相关的文章进行分享，也敬请大家期待)。在8卡情况下，由于我们充分利用了NVLink提供的高速卡间互联进行数据通信，同时随着自动混合精度和Fusion Stitching优化的加入随着计算的粒度变大和计算效率提高，通信和IO方面的压力突显出来，对我们提出了更多的挑战。

这里着重讲一下对于跨device control dependency依赖的聚合简化和对数据IO的优化，在这两个优化的共同作用下，我们消除了数据IO的bottleneck，使得我们计算优化能够充分发挥其威力，进而提升业务方同学的模型迭代效率。

Dependency Optimizer

在profiling过程中发现，单机8卡情况下，存在着大量的*Send/Recv/Identity/Const Op*的launch阻塞了GPU设备侧计算kernel的发射时机，这就影响到了GPU计算性能的表现。首先我们分析一下这些Op的来源。

ControlDependency来源

业务方模型训练中使用的是AdamOptimizer。在TensorFlow的AdamOptimizer实现中可以看到，**AdamOptimizer需要在等到所有的update操作完成之后才会进行最后的** \beta1,\beta2 **乃至于global step的更新。**

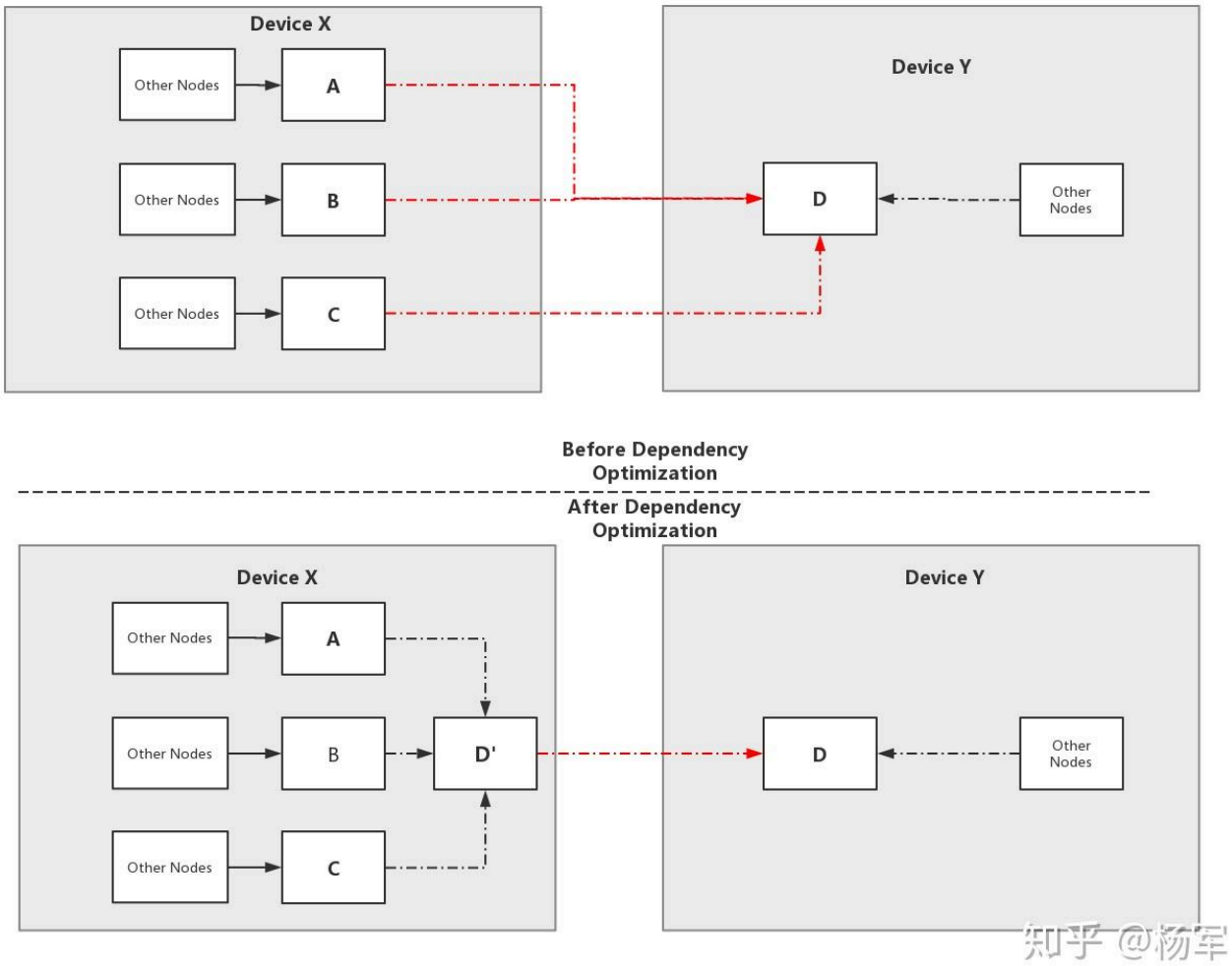
由于单机8卡情况下，我们使用MirroredStrategy作为同步更新的机制，因此需要等到所有卡上的Variable更新都完成之后才会进行最终的global step等的更新。而Transformer2.0模型一共有198个Variable，因此每张卡上的global_step更新都要依赖于当前卡以及其他卡上的Variable更新完成，**因此产生了7 * 198个跨device的依赖边**，8张卡每张卡上Variable都有这样的跨device依赖，因此一共产生了**8 * 7 * 198**个跨device依赖边。

TensorFlow实现机制中会将1个跨device control dependency依赖边转换为1个Send，1个Recv，1个Const以及1个Identity 4个Op，这部分引入了一共**44,352**个Op，这些op的launch不能被overlap隐藏起来，因此其造成了后续计算的阻塞。

ControlDependency聚合

假设我们存在**A->D, B->D,C->D**这样的依赖边，同时A、B、C在同一个device X上，而C在另外一个device Y上，则我们可以在device X上引入一个相同device上的D‘节点，使得A、B和C都依赖于D’，将原始的依赖关系等价转换为了

A->D', B->D', C'->D‘，D'->D，将原有的3个跨device的依赖边，聚合之后只存在一个跨device依赖边，如下图所示，图中的红色虚线就是跨device依赖边。



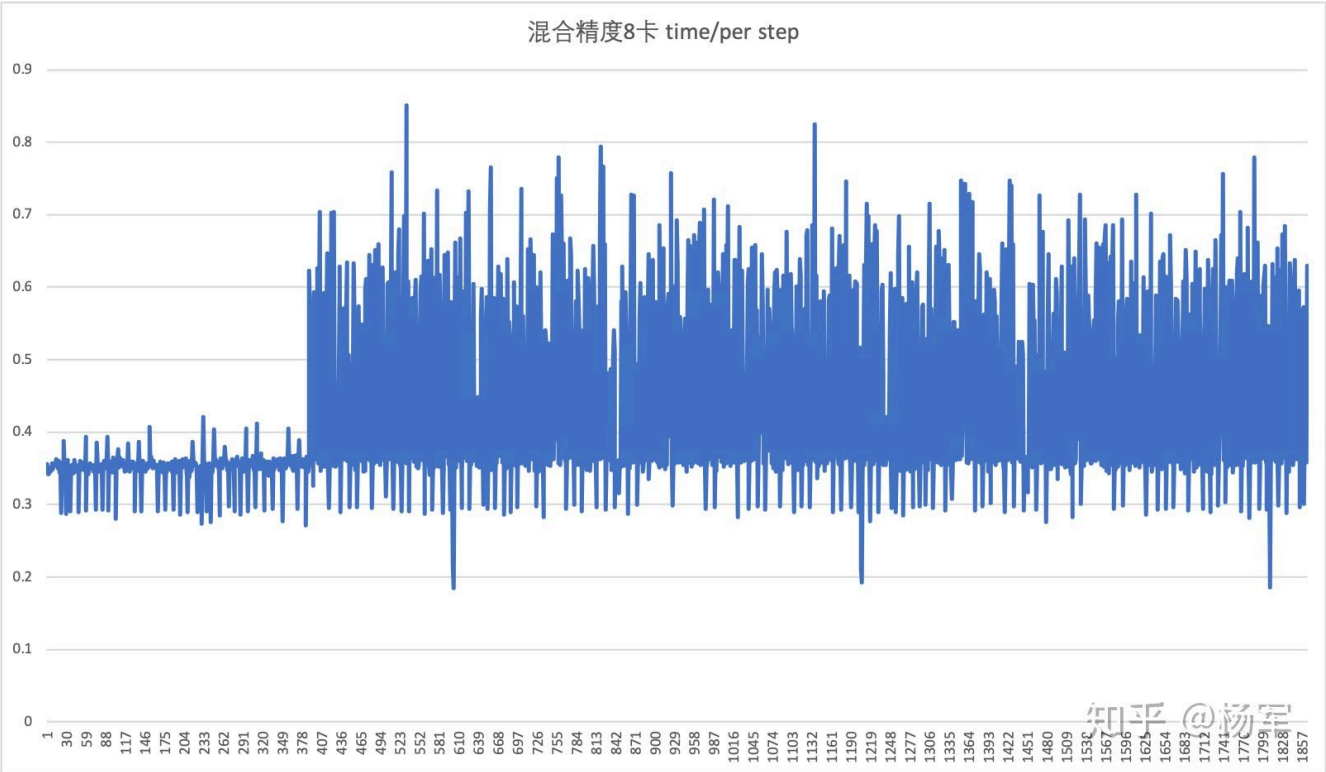
效果和性能提升

经过Dependency Optimizer对Control Dependency的合并简化，**减少了44,128个冗余op的使用**。我们通过跨device依赖边的聚合优化，获得了约**10%**的性能提升，同时由于op launch的大大减少，也减轻了CPU的开销。

数据预处理

业务方的模型训练中的数据预处理流程是经典的文本数据bucketing的做法。输入是原始文本数据，在预处理流程中转换为数字id，并且按照数据长度组成各个bucket，每次读取数据时，会从一个有数据的bucket中获取一个batch的数据。

数据预处理阻塞计算



图中的step time上可以看出300步之前，每个step时间比较平均，但是300步之后，平均时间开始增加，且伴随着比较大的时间抖动。我们profiling也发现了，经过约300个step计算的之后，数据的预取队列就变为空，意味着后续的计算中，每次取数据的时候，因为预取队列为空，获取数据时都要等数据处理好，再进行计算，即数据供给部分阻塞了计算的进行。

经过分析发现，计算数据吞吐速率高于数据预处理时候的吞吐速率，因此出现的现象是计算等待数据，根据木桶效应，此时的短板就在于数据预处理的效率，因为只有解决掉解决掉数据预处理的短板，才能充分发挥计算的性能。

数据预处理中的性能热点及优化

经过profiling我们发现，IO预处理部分pipeline中的瓶颈一是在vocabulary lookup和异常样本过滤部分，另外一个是在最后bucketing的组batch部分，这两个部分中，第一个部分是可以转到离线过程中的，这样就可以消除数据预处理对于计算的阻塞。

将数据IO性能热点中的，vocabulary lookup和异常样本过滤转移到离线环节中，业务方同学使用UDF和SQL完成了这两个步骤，并加入到了训练的pipeline当中。

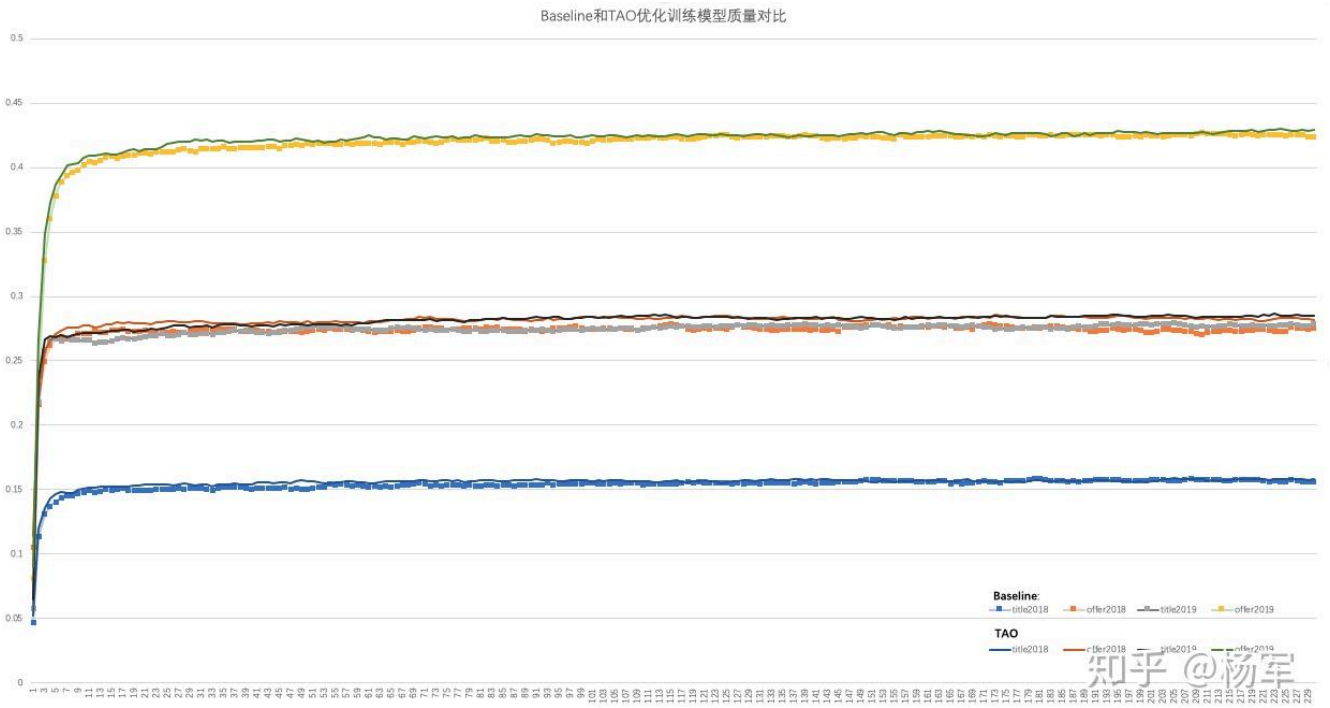
最终优化效果

我们将访存密集算子优化，计算密集算子优化以及分布式优化相结合，改造了IO通路之后，目前的训练过程已经集成到了业务生产训练平台中，端到端加速比为**3.65X**，训练收敛时间由之前的85小时缩短到了25小时，显著提升了业务模型训练的迭代效率。

值得一提的是，这里的加速结果，**是在使用相同模型配置，数据以及相同的资源情况下所得到的**，相当于大幅提升了现有硬件资源的使用效率，从而达到PAI作为AI基础设施提供方所一直

倡导的“**使用更少的硬件，支持更多业务更快完成迭代**“的宗旨。

下面的曲线为训练过程中bleu值随着迭代的step的变化曲线，可以看出上述所有优化打开之后，训练过程中收敛趋势和baseline基本一致，做到了精度的无损。



总结

在对机器翻译团队模型训练进行加速的过程中，我们将访存密集算子优化，计算密集算子优化以及分布式/IO通用优化手段相结合，形成优化的组合拳，获得了理想的加速效果。在业务落地过程中解决的可用性和性能问题，都已经沉淀到了PAI的训练工具中，作为通用优化手段的一部分，在其他的workload上面也得到了充分的检验，助力我们不断完善AI编译的技术建设。

在机器翻译模型使用编译技术加速训练过程中，我们几乎遇到了所有的训练任务的常见问题，编译优化的可用性问题，性能问题，通信问题，IO问题，甚至还遇到了一个不合理的padding带来的精度损失问题，在这个过程中，不仅仅锤炼了AI编译技术的成熟度和可用性，我们也对未来的发力点有了进一步的认知。希望通过AI编译优化工作的持续推进，可以让业务方算法团队的同学能够更加专注于模型和算法本身，真正的实现让天下没有难train的model。

这篇文章分享的其实是18年底团队的一个业务落地工作，在过去的近两年时间，我们在AI编译技术以及落地上有了更进一步的发展，也在不断启动一个又一个更exciting的项目，欢迎对我们的工作感兴趣的同学联系我们，一起来推进AI编译技术的建设打造。