AI编译优化---访存密集算子优化

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。*

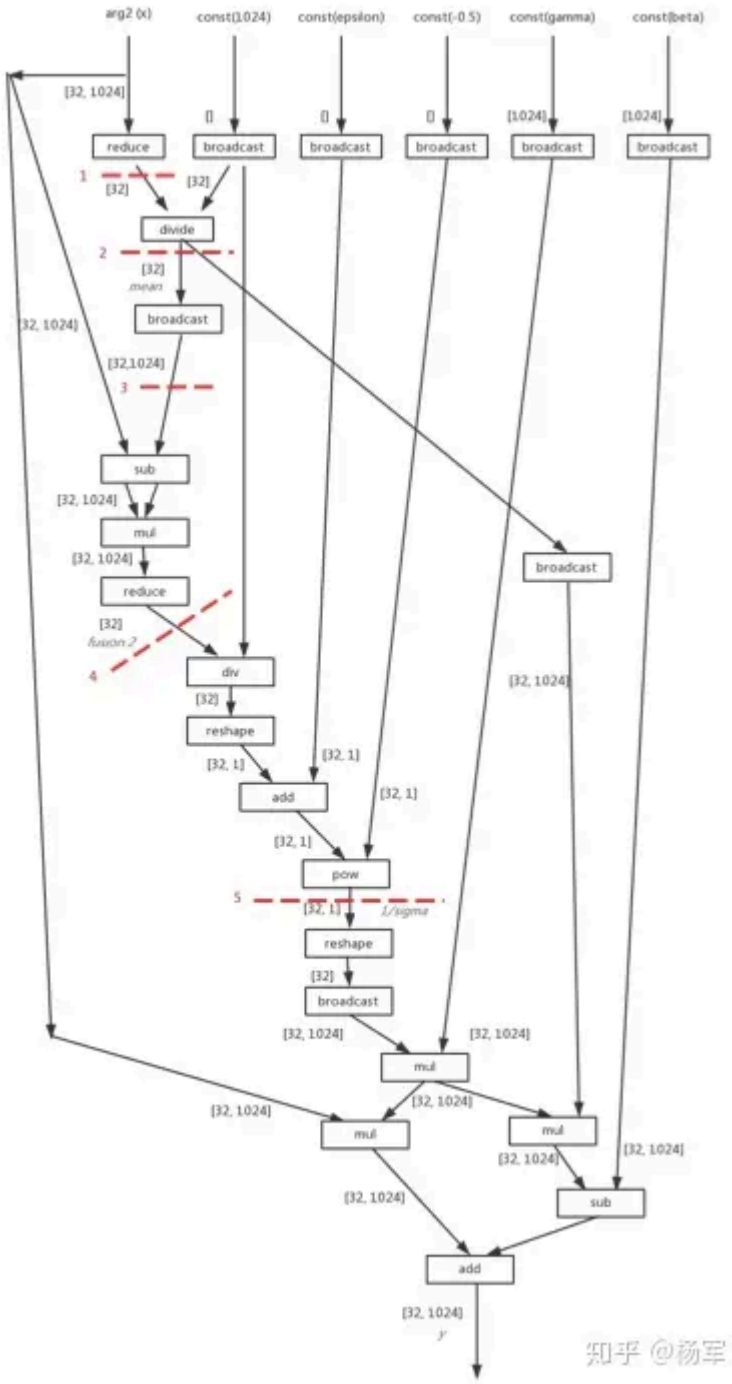
本文是AI编译优化系列连载的第二篇，总纲请移步：

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

PAI团队在访存密集算子上以XLA为基点开展了大量的工作，工作内容涵盖前端、中端、后端以及执行引擎。这一系列工作先后经历了V1和V2两代演进。

如前文所说，XLA在GPU backend上的主要收益来源是对访存密集型算子的自动Op Fusion CodeGen。催生我们做这些尝试的原因是，我们在实际业务中发现，社区XLA在最核心的CodeGen环节还有很大的问题和改进空间。

例如下图为一个LayerNorm模块的前向计算子图，手工优化的话，它可以很容易被写成一个CUDA kernel，本应该是很适合编译器通过自动op fusion来获取性能收益。但我们发现XLA实际并没有做的非常好。经过细致的分析后我们认为这里的本质问题是社区的CodeGen模版过于简单，因为模版非常简单限制了中端Fusion pass的灵活度，被迫做了非常保守的op fusion策略，图中每一条红线都有一个细节的原因导致社区XLA无法把这些算子fuse到一起，也就无法拿到节省相应访存开销所能得到的优化收益。



分析之后我们认为造成这些问题的根本原因为：

- 单一Parallel Loop的比较简单的CodeGen模版；
- 由于CodeGen模版比较简单，为保证性能而被迫保守的Op Fusion策略。

这种简单的CodeGen策略在fuse线性的Elementwise算子时足够得到理想的结果，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。

当然，我们可以直接采用复杂的CodeGen模版，通过牺牲通用性的方式换取更好的性能收益。但在Op节点类型以及计算图的拓扑结构千变万化的情况下，这种方式并不能够在编译器中通过编译的方式有效的节省人力成本。于是就引申出了我们工作里的设计理念。

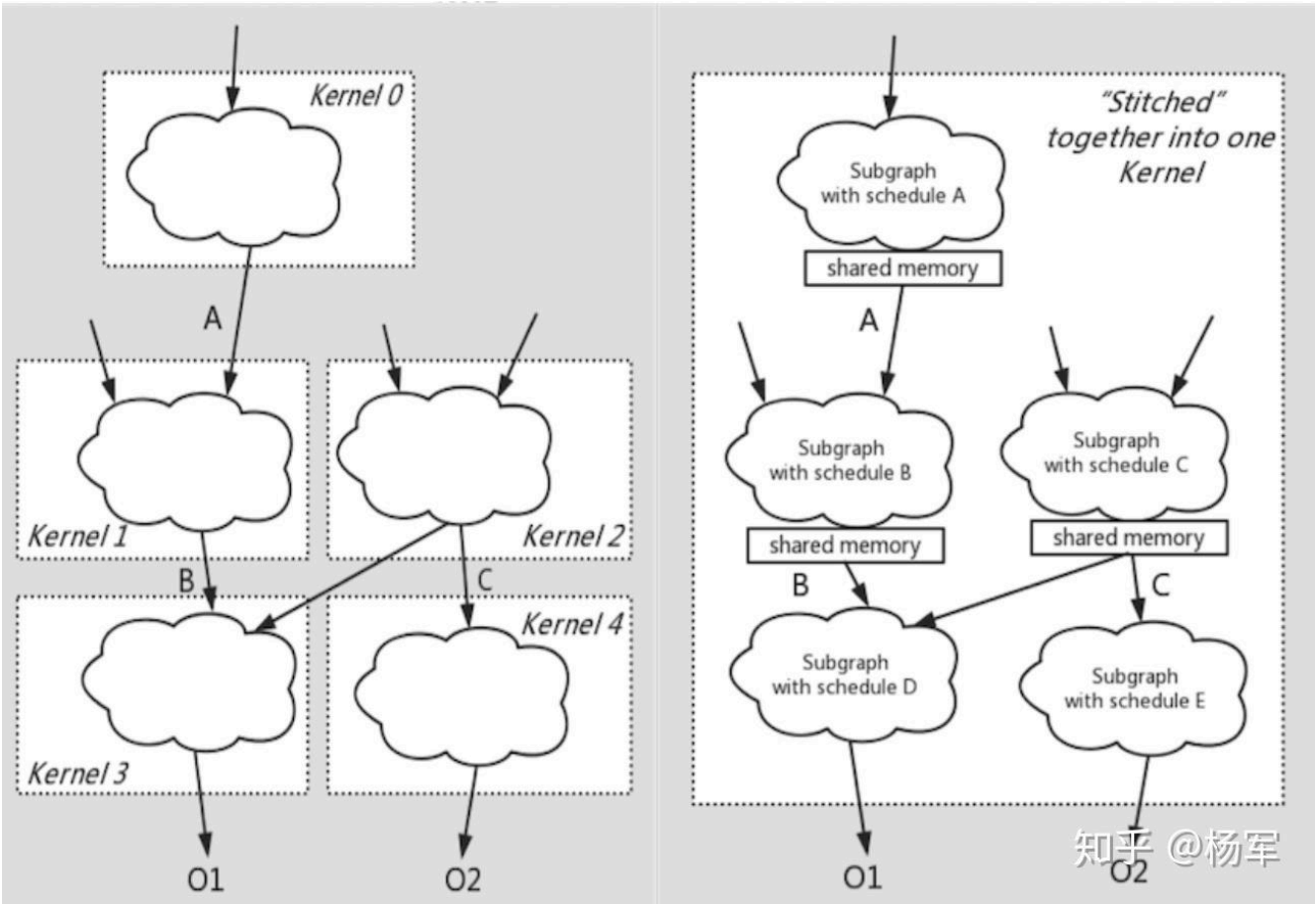
V1

V1的核心原理可以概括为：

- 借助于GPU硬件中低访存开销的shared memory，缝合不同schedule的计算子图，实现多个parallel loop复合
- 支持不同shape的多个tensor输出
- 保证CodeGen性能的前提下，实现更加激进的Op fusion策略

在GPU的体系结构中，SM内的shared memory的访存开销远远低于global memory的访存开销，可以帮助我们来把多个本来独立的kernel粘合在一起。这样的方案下，被粘合的每一个子图仍然可以基于自己的简单的CodeGen模版来运作，他们不必担心因为各自的pattern被fuse在一起而需要更复杂的CodeGen模版，也一定程度上不必担心因为fuse的颗粒度太大而产生过多的冗余的计算。V1的核心工作涉及中端和后端两个部分，解决了多种不同细节原因导致的细粒度算子无法fuse到一起的问题，主要是基于这个核心思路。

下面的两张图分别为社区XLA Codegen的情况和V1的情况。V1的CodeGen将社区做法中难以fuse在一起的计算pattern，通过低访存开销的shared memory作为桥接，将多个kernel缝合在一起，同时又保留被缝合的每一个部分都依然采用简单的CodeGen模版。这种方案在通用性和性能之间取得一个更好的平衡点。



V1通过这样的方式，在一定程度上保证CodeGen性能的基础上，大幅提高了op fusion的颗粒度，以LSTM前向和LayerNorm前向这两个计算pattern为例:

- LSTM前向图的TensorFlow结点数包括18个原子op，LayerNorm前向图包括42个原子Op
- 社区XLA完成fusion之后，LSTM前向图由4条fusion指令组成，LayerNorm前向图由6条fusion指令组成
- V1完成fusion之后，LSTM前向图只剩下2条fusion指令，LayerNorm前向图则可以完全fuse成一条指令

V2

在做V1的推广落地的同时，我们抽取了一些业务进行了比较精细的分析，观察优化距离性能的极限还有多大的距离，结论是虽然凭借在op fusion颗粒度上的优势，在很多case上可以做到明显优于社区XLA，但离硬件性能的峰值还是有比较明显的gap的。所以我们抽取了一些有代表性的kernel，分析了这些gap的具体原因。我们发现每个case各自都有不同的原因导致没有能够接近访存的峰值性能。这些原因比较细节，在此不再展开，但宏观来说问题来自于目前的CodeGen基于Rule-Based的策略，而Rule-Based的策略本身存在其固有的局限性。

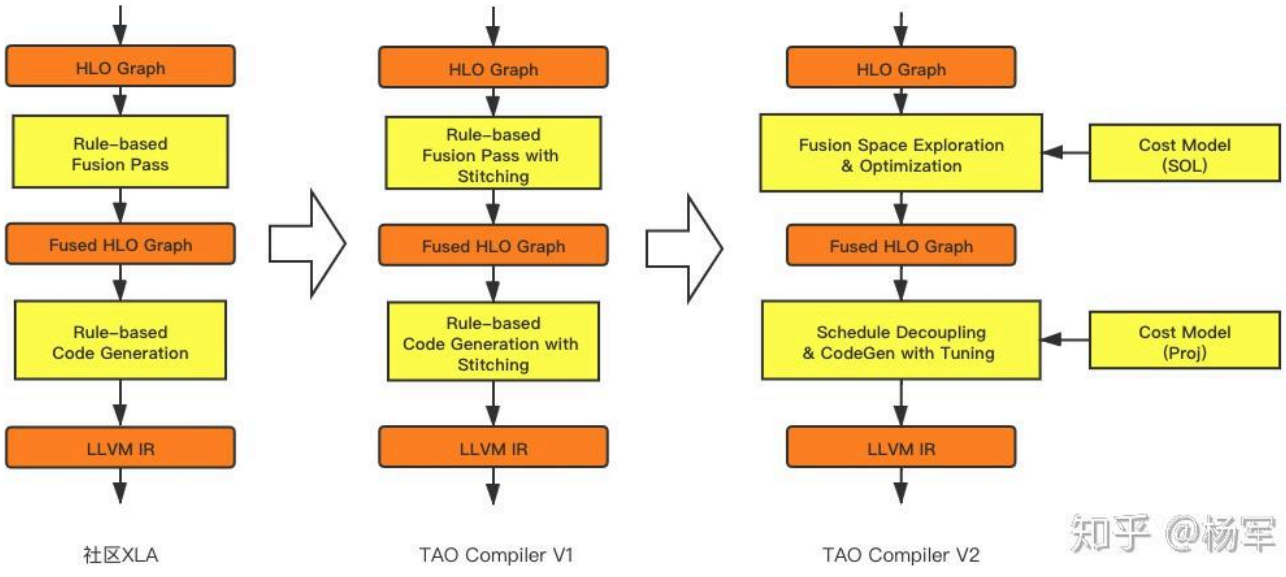
具体到compiler的各个环节：

- 在中端Op Fusion层面上，基于简单规则下局部合理的Op Fusion决策可能并非全局最优的决策；
- 在后端CodeGen层面上，基于简单规则下，同一Kernel内的不同节点的模版选择和CodeGen行为决策也可能并非整体最优的决策。

在V2与V1以及社区XLA相比最大的区别可以概括为：

- 更多的考虑全局最优，而非基于贪婪的局部最优；
- Rule-Based到Cost-Based的演进。

这两点区别同时反映在中端的Op Fusion策略以及后端的CodeGen策略上(下图中出现的TAO Compiler是我们内部的一个codename，后面也会出现，不再赘述)。



我们引入了多个不同等级的cost model，其主要区别如下：

- SOL(speed) of light) cost model：开销小，准确度相对低，主要考虑理论仿存量和理论计算量等；
- PROJ(projected) cost model：开销适中，准确度上对较高，会对GPU体系结构进行适度建模；
- Profiling cost model：开销大，准确度最高，用于编译后实际Profiling测速。

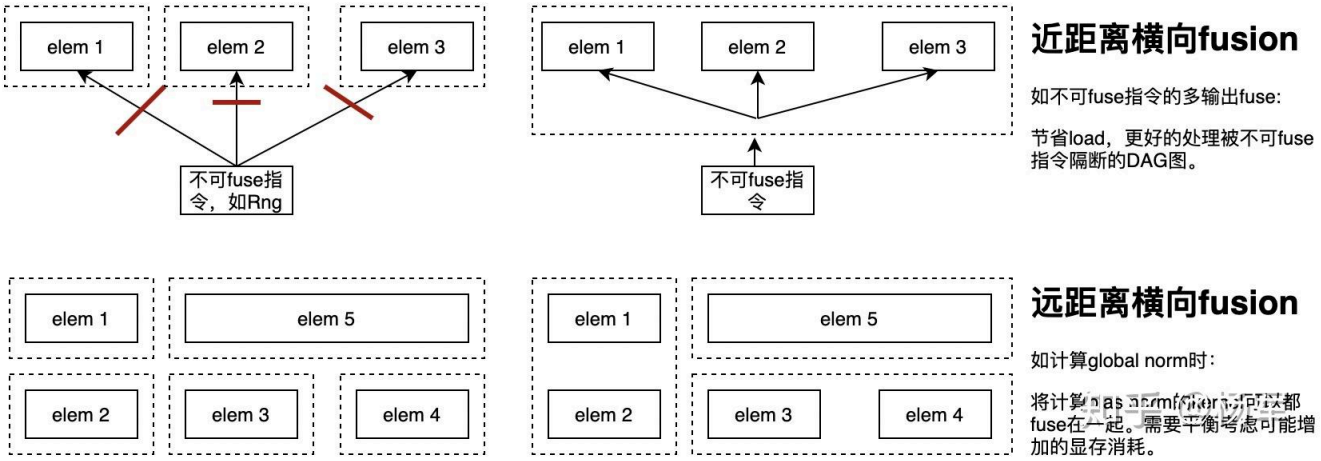
中端在Graph范围内做Op Fusion决策，由于探索范围较大，选择接入开销较低的SOL cost model；后端在Kernel范围内做tuning，可以接入开销相对较高的Proj Cost Model。

中端Op Fusion决策部分

目前XLA和V1中解决HLO instruction之间的Auto Fusion问题时采用的是完全基于规则的策略。这些规则都是基于人工经验确定的，往往难以覆盖到所有的情况，导致可能会出现各种corner case以及为了解决这些corner case而打上的一系列补丁。从长期来看这种方法并不能充分的挖缺潜在的性能优化空间，也会带来越来越沉重的维护成本。为了使用更加系统性的方法解决Auto Fusion的问题，我们提出了Cost Based Fusion Strategy。

新的策略主要是对V1中以下两个大的方面进行改进：

- **兼容性度量。**兼容性度量是一种用来评估一个给定的Fusion动作是否有收益的方法。目前V1中使用的是一条简单的规则来进行兼容性度量，即只要融合之后的kernel的thread block level的并行度没有下降到1则认为融合具有收益。该策略在很多实际业务上发挥了很好的作用，但也遇到了不少Corner case。比如该规则对于大shape的pattern表现为过于激进，融合后可能因并行性大幅下降而导致性能变差，而对于小shape的pattern则表现的过于保守，丢掉了一部分融合的机会。在V2中我们基于cost model来更系统化的考虑访存量，kernel launch次数，并行度改变对于最终性能的影响等，可以更准确的评估一个融合动作的效果以指导Fusion Plan的探索。
- **Fusion的探索空间。**一个计算图对应的Fusion Plan由多个Fusion Pattern组成。一个Fusion Pattern中如果只包含具有生产消费关系指令则是纵向fusion，反之如果包含有不含生产消费关系的指令否则为横向fusion。一个计算图可能存在着多种不同的Fusion Plan。目前V1中是基于一个启发式的贪心策略确定性的生成一个Fusion Plan，而再不进行其他可能的探索。这种策略好处是具有很高的效率，但缺点是会漏掉很多可能性。另一方面目前社区XLA及V1中只会考虑纵向fusion而未考虑横向fusion。这里所说的横向fusion即不存在直接生产消费关系的节点之间的fusion，如下图所示。除其中近距离横向fusion可同样带来仿存节省的收益外，在更多的场景下可通过节省kernel launch开销带来收益，同时当使用空分并行的方式对横向fusion之后的kernel做CodeGen时还可能提升GPU的资源利用率。下图中展示了一些横向fusion的例子。V2的中端部分，对包含以上两种fusion的解空间进行了更多探索，以期找到更好的全局最优决策。



具体来说fusion空间探索是一个迭代式的过程，每一次迭代由以下几个步骤构成：

- **搜索高价值fusion pattern。**这里的高价值指的是基于cost model评估会带来更大收益的pattern。直接暴力枚举进行搜索是一个指数级别的算法，我们使用的是一种启发式的算法，在稀疏图场景下（一般指令间的连接图满足这个前提）是一个接近线性的算法。具体而言我们首先构造计算图指令间的一个拓扑排序，依次遍历每条指令并按照递推的方式生成以该指令结尾的top k候选fusion pattern。递推的过程中我们会进行局部的充分枚举（扩张），对整个空间进行更多的探索，同时也会及时进行剪枝（收缩），确保可以不会组合爆炸。
- **构造fusion plan。**即寻找Fusion Pattern的一个有效组合以构成一个完整的Fusion Plan。这里的有效组合指的是各个Fusion Pattern之间不相交且相应的融合动作不会引入环。 这里我们尝试了多种不同的策略，比如贪心策略，BFS+剪枝策略。目前使用的是每次选择兼容的且收益最大的fusion pattern的贪心策略。
- **apply fusion plan。**也即根据Fusion Plan执行相应的融合动作。

以上的过程可以对整个优化空间进行更大程度的探索，同时又有效的控制了搜索的复杂性，在一组评测应用集合（这也是进行编译器开发经常性需要的工具支撑了）上的实验表明，该策略工作良好，符合预期。

后端CodeGen部分

在介绍V2后端CodeGen的具体工作之前，我们先对XLA和TVM做一个简单的横向比较：

- XLA是一种比较务实的方案，用基于规则的简单方式去做收益空间最大的访存密集型pattern的Automatic CodeGen，它的优势是自动化和比较小的编译开销，缺点是如果想进一步提升性能的话会非常局限；
- TVM的目标是去胜任更复杂的CodeGen，基于规则的策略无法胜任情况下，它在Halide工作的基础上，提出了一个比较先进的Schedule抽象。这个Schedule的空间非常大，如何才能找到最好的Schedule，一方面需要基于用户的经验写出比较适合硬件体系结构的Schedule，一方面需要巨大的编译开销为代价在Schedule之间进行tuning。

这里我们需要解决的问题是：**需要找到一种方式，能够自动/对用户透明的，在有限的编译开销的基础上，提升性能并进一步接近硬件的性能峰值极限。**

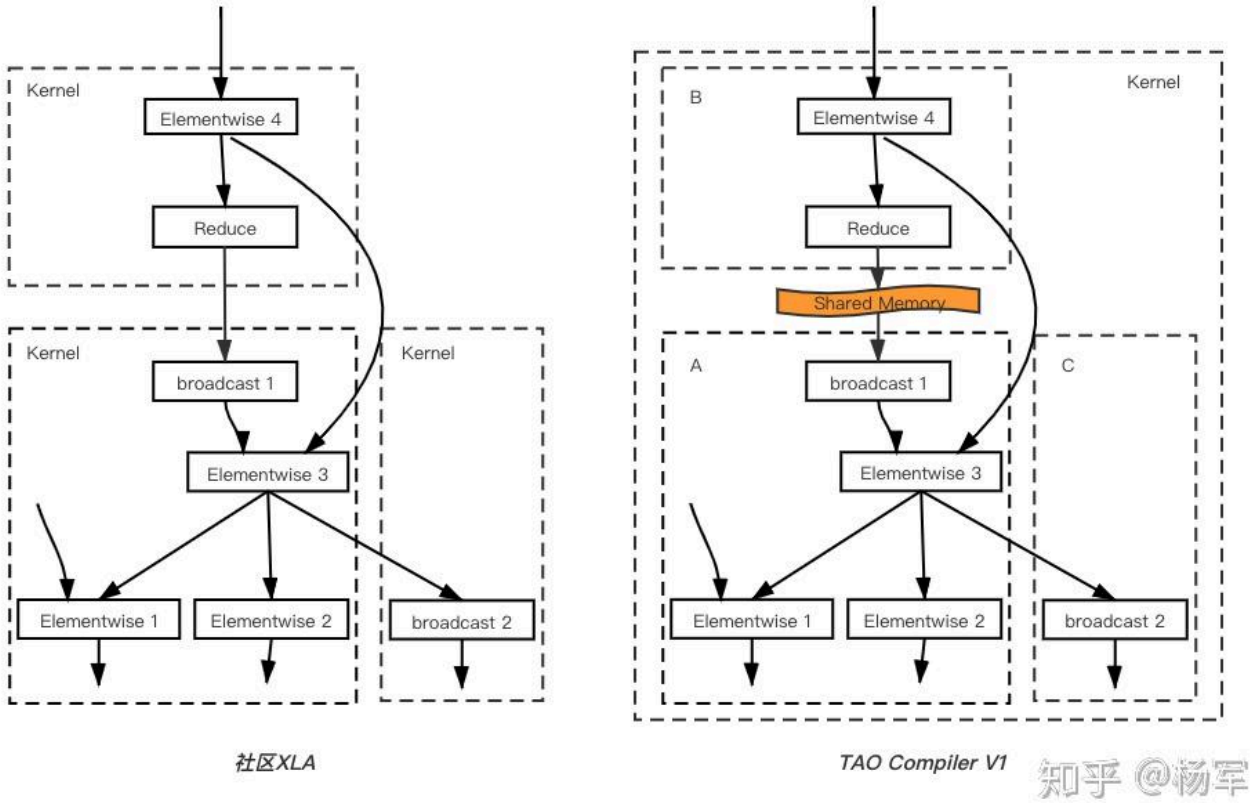
V2的工作在XLA的基础上，一定程度上试图借鉴TVM的Schedule抽象的核心想法，但又有本质的不同。根据我们过往工作的经验，对于访存密集型pattern来说，为了达到或接近硬件的峰值性能极限，所需要的Schedule的枚举空间通常并不需要非常大。因此我们试图提炼一套比TVM的schedule抽象要简化很多的CodeGen Plan抽象，希望这套抽象能够代表手工写相应Pattern的Kernel时所需要考虑的全部问题空间。同时需要保证CodeGenPlan的枚举空间是足够小的，能够满足用户对编译开销的需求。此外，我们使用Proj Cost Model而非TVM中目前基于实际profiling结果来tuning的方式，其原因也是希望能在编译开销和性能之间取得更好的平衡点。

在V1的工作的基础上，V2在CodeGen部分的主要区别是：

- CodeGen Plan抽象，尽量对全部的CodeGen可能性做抽象和枚举；
- Index计算与主体计算分离，引入*index_cache*与*value_cache*，最小化冗余计算；
- 与CodeGen Plan抽象相对应的Implementation抽象，方便对不同节点增加新的算子实现模版。

篇幅原因我们不再详细介绍CodeGen Plan抽象的具体细节。下面仅试图以一个pattern为例，来说明V2与V1及社区XLA之间，在CodeGen结果上的主要区别。

下图左图为社区XLA的CodeGen结果，由于简单模版的限制会生成3个简单Schedule的kernel；右图是V1的结果，借助Shared Memory的粘合，可以生成一个kernel。



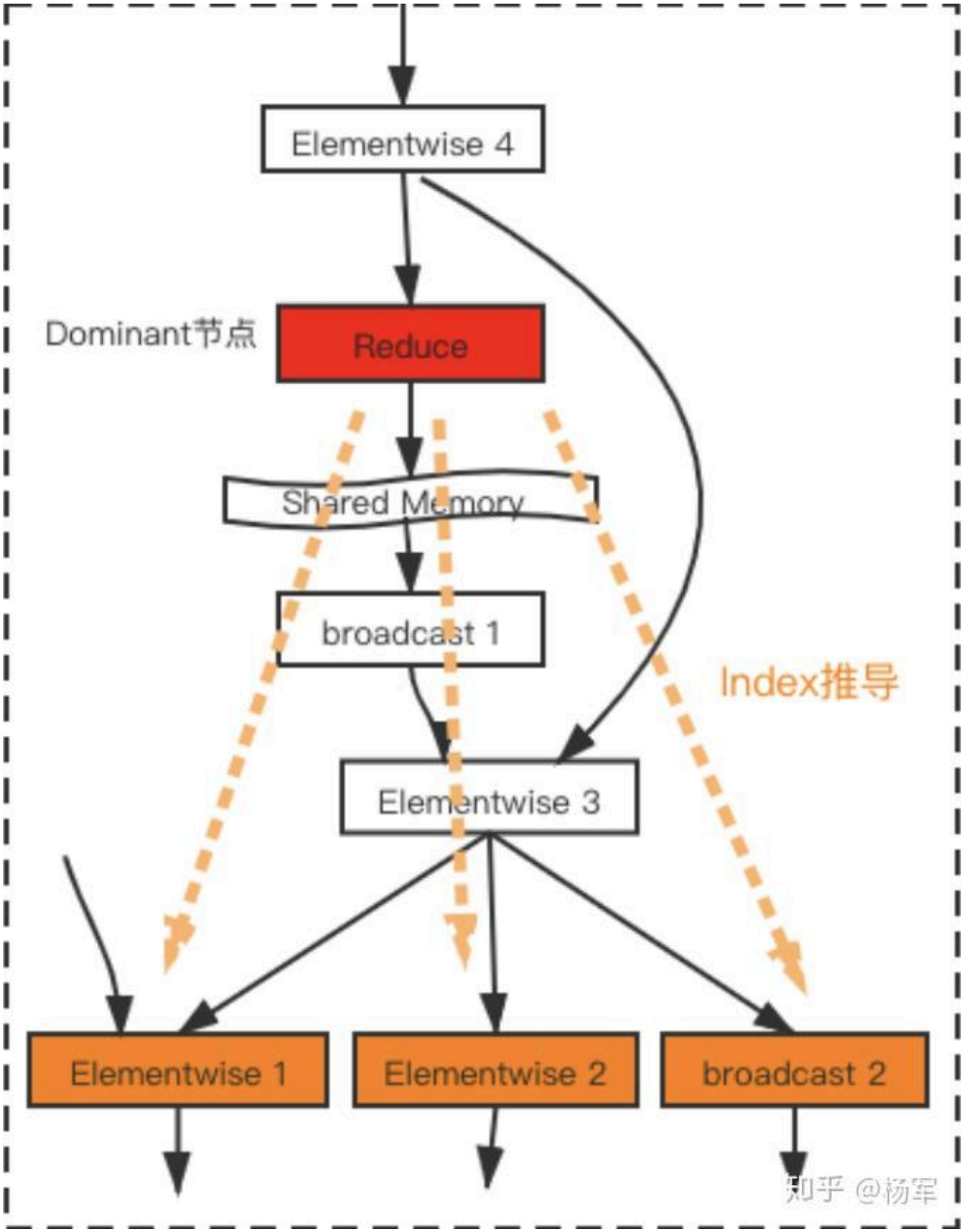
社区XLA　　TAO Compiler V1

V1距离硬件性能峰值之间可能存在差距的主要问题来自于：

- A/B/C三个子部分的CodeGen Schedule是彼此独立的，在此例中，A/B/C三个部分的计算在同一个GPU thread内可能依赖于Elementwise 4节点处的不同Element，产生冗余计算。

- A/B/C三个子部分的CodeGen过程相对独立，Elementwise 3等节点处的值和Index的计算结果无法被有效Cache，进一步增加了冗余计算指令。
- A/B/C三个子部分可能各自有多种CodeGen模版，但可能只有一部分子模版的组合可以在全局上最好的利用GPU的并发度（即GPU Occupancy）。

当冗余计算指令的数量（或者Occupancy带来的问题）达到一定临界值后，本应该是仿存Bound的计算pattern，性能热点便会迁移为计算或Occupancy，导致无法达到硬件仿存性能峰值的性能。



V2经过CodeGen Plan的枚举和基于Cost Model的tuning之后，会找到一个认为相对最优的CodeGen Plan。例如，在上图这个pattern中，选择Reduce节点做为整个Kernel的Dominant节点可能是更好的方案，其它节点处，在同一个thread内需要计算的Index会由Dominant节点的Index以辐射传播的方式推导计算得到，Index的传播过程和数值的计算过程在整个kernel范围内进行Cache，通过以上机制最大化的避免冗余的Index计算和数值计算。此外，CostModel还会对枚举

的CodeGen Plan中的Launch Dimension等因素对Occupancy的影响进行建模，得到对于整体性能最优的CodeGen行为。

Cost Model

中端不同的fusion策略以及不同后端CodeGen生成方式会影响最终模型实现的效率。所以很自然的想到利用cost model帮助中端、后端CodeGen探索最优的实现。这里主要的挑战是：

- cost model需要高效的对成百上千种不同实现做实时的评估，需要减少其本身的overhead；
- cost model需要准确的区分不同实现的性能差异；
- 不同型号的gpu拥有不同的硬件架构（memory hierarchy，latency以及throughput指标），cost model需要对不同型号的gpu建模。

所以我们会将workload输入映射成不同的资源需求，同时对device内部流水线stage的resource、latency以及throughput做了抽象和表示。最终通过建立latency模型以及throughput模型估计当前实现的计算耗时。

Cost Model分为3个level： SOL Cost Model, Projected Cost Model以及Profiling Cost Model。

- SOL Cost Model 只考虑硬件spec，不考虑具体实现、架构特点（例如假设100% cache hit ）和编译器效率。对于CodeGen，主要用memory cycles以及device throughput（例如：处理 1000T MACs, V100 125 Tflops vs 4x TPU 180 Tflops vs new ASIC）来区分。其特点是建模速度快，误差较大， 所以可以用于粗粒度性能筛选以及不同硬件性能快速评估。因此,我们在中端OP fusion探索决策时引入SOL cost model评估不同Op fusion的cost并保留下cost 最小的Op fusion Plan。
- Projected Cost Model需要考虑具体kernel实现、架构特点（如hit rate）和编译器特点。对于CodeGen，主要用kernel latency来区分。特点是建模速度慢，但误差较小，所以可以用于细粒度性能筛选。因此,我们在做后端CodeGen时，引入Proj cost model区分同一条fused instrcution在不同CodeGen Plan下因为指令数不同以及寄存器复用情况差异造成的性能差别。
- Profiling Cost Model目的是作为cost model的ground-truth， 从profiler上得到实际运行性能结果。其特点是执行耗时长，所以可以用于构造offline cost model以及修正SOL level与Proj Level cost model。

Cost Model主要由三个模块组成： Workload Model, Device Model, 以及 Cost Evaluation。简单来说 cost=\frac{workload_model}{device_spec} 。可以举一个简单的例子，假设任务的workload由10条硬件指令组成，芯片的处理能力是每个cycle处理2 条硬件指令，那么该workload需要5个cycle才能被处理完。

Workload model的准确性影响了整个cost model建模的准确性，因此80%以上的研发时间用于建模workload model。实践中，我们会估计某条fused instruction在当前CodeGen Plan下的寄存器消耗、各个硬件单元的硬件指令需求、memory 消耗以及shared memory消耗。

Device Model建模了GPU各个硬件单元的throughput、bandwidth以及latency。可以从公开资料或者论文中得到.在device model中，我们主要建模了芯片内部各个unit的数量、SM内各个单元的throughput和latency、L2的throughput（用于建模atom指令）以及 Memory bandwidth。

Cost Evaluation根据workload model以及device model 估计最终生成kernel的Memory cycle、Occupancy（并行度）、Wave number、Kernel Latency、Performance Limiter以及SM内部各个unit的cycles。

Cost Model主要有两种建模方式：Throughput Model以及Latency Model。

Throughput Model首先估计fused instruction所需各机器指令的数量（例如需要X条FFMA, Y条MOV, Z条IMAD等）。并对机器指令根据其对应的pipe做聚合（例如FFMA和IMAD都是发射到FMA Pipe的）。在获得当前GPU各个pipe的throughput后，用pipe workload（硬件指令）的数量除以这个pipe的throughput就是其所需要的cycle数。最后在所有pipe中找cycle最大的pipe，这个pipe就是这个fused instruction在当前gpu上的performance limiter，并且这个pipe所需要的时间就是kernel的时间。

Latency Model首先估计fused instruction所需的register数量，并计算出它的occupancy，再计算出总共需要的wave数量（波数）。然后估计fused instruction所需各硬件指令的数量（同Throughput Model中的第一条）。根据device model获得当前gpu各个硬件指令的latency。根据硬件指令数以及指令的latency计算 warp latency，用warp latency乘以wave数量得到最后的kernel latency。

对于V1来说，由于大部分fused instruction已经达到或者接近GPU math或memory的理论上限，此类情况下，throughput model能很好地对fused instruction做建模。 对于V2，由于fuse了更大的尺度，目前大部分fused instruction距离硬件极限还有点距离，所以用throughput model做区分是不合理的。需要用latency model。

中后端之外的编译器框架开发

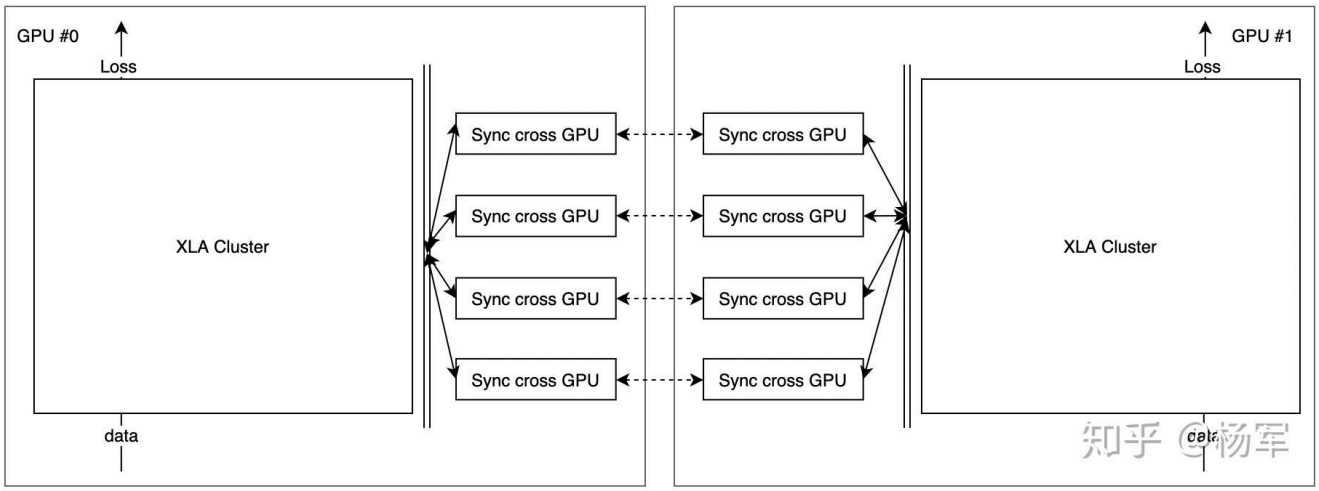
前面介绍的V1/V2的工作主要涵盖了我们在AI编译器中端和后端的工作内容，而在实际业务推进过程中，除了中后端工作以外，还存在若干编译前端，以及执行引擎层面的工作需要配套起来，才可能充分发挥出编译器的优化效能，这也体现出编译器研发“食不厌精”的特性。下面列举了两个有代表性的工作例子。

Output Proxy

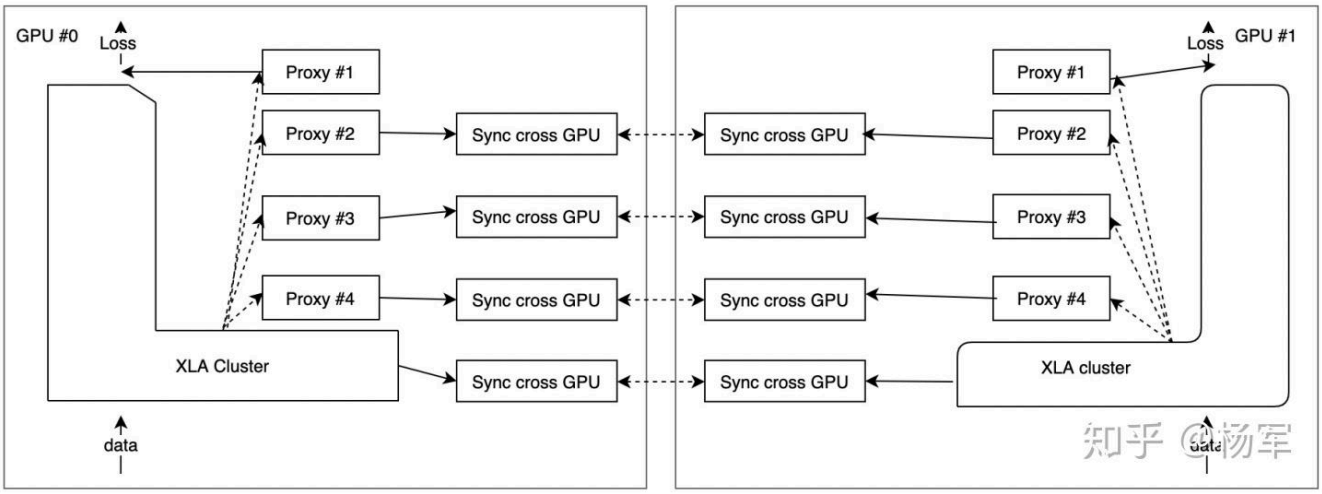
一般而言，由于更大的编译子图拥有更多的上下文信息，因而对应有更大的潜在编译优化空间，也即大尺度的cluster对于编译优化而言更加友好。另一方面，目前TF runtime将一个完整的cluster看成是一个op，这意味着所有依赖于该cluster的其他op，即使它只使用了该cluster的部分输出，也必须等待该cluster全部执行完成之后才能够开始执行。上述的设计对于单GPU的执行环境而言一般不会产生副作用，**但在分布式的环境下大尺度的cluster则会打破计算与通信之间的相互掩盖，从而导致潜在的性能下降，也即大尺度的cluster对于分布式执行不友好。**

举例而言，假设一次训练使用两块GPU，做数据并行，使用allreduce算法交换卡间梯度的场景。在这个场景中，假设不使用XLA，则当GPU在计算gradient #i时，由于gradient #i+1已经计算完成，所以可以并行进行gradient #i+1的allreduce操作，从而可以充分的利用GPU间的数据传输带宽，提高端到端的性能。

作为对比，下图展示的是在同样的设置下打开编译优化时的情形。具体而言，在这个例子中所有的前向和反向计算都被划分到同一个cluster之中，而由于allreduce op目前并不支持编译，因而并不能与计算op划分在一起。如此造成的结果是，所有的allreduce操作都只能在前向和反向操作全部完成之后才能开始，打破了原有的计算和通信之间的相互掩盖，从而导致潜在的端到端的性能下降（即使由于打开了编译优化提升了计算的速度，但通信的时间会相应的延长，从而可能导致端到端的时间反而延长）。



Output Proxy是我们针对这个问题设计的一种方案用来保证在大尺度cluster的前提下（也即充分利用编译优化的技术提升计算性能）解决由于粒度过粗导致分布式环境中计算与通信不能有效的相互掩盖的问题。如下图所示：

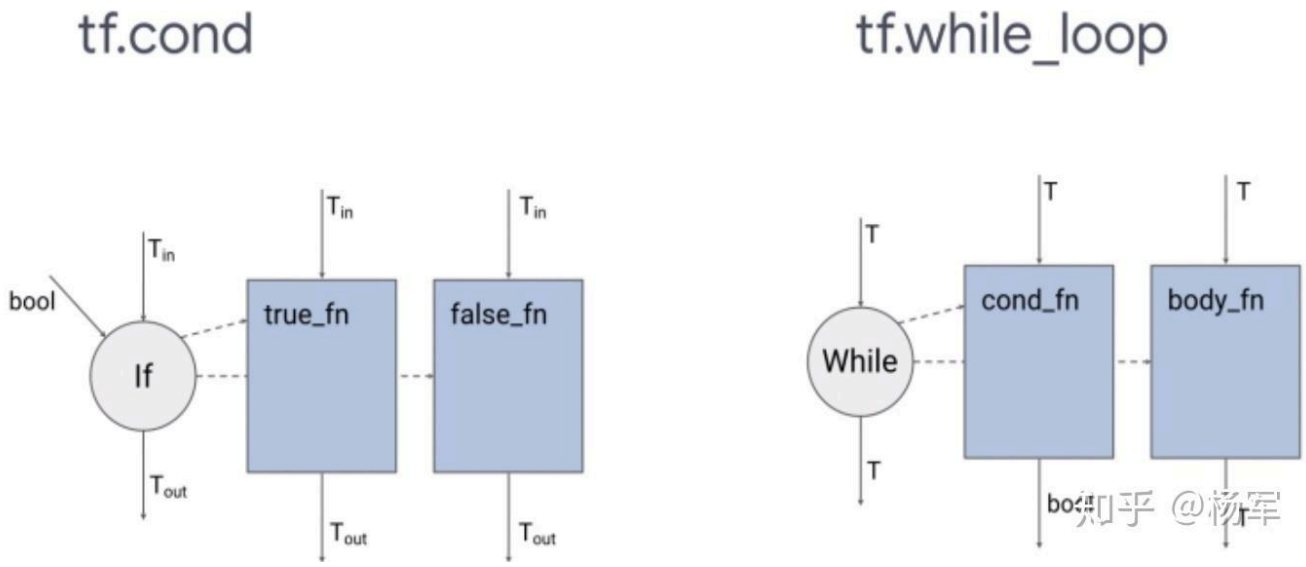


打破大尺度cluster引发的false sync，基本思想是需要实现on-demand ouput, 也即在编译cluster的内部，一边执行计算一边将已经就绪的输出结果带回到TF runtime，以便TF runtime调度后续依赖于这些输出的其他op（主要是针对数据通信op）尽快开始执行。*

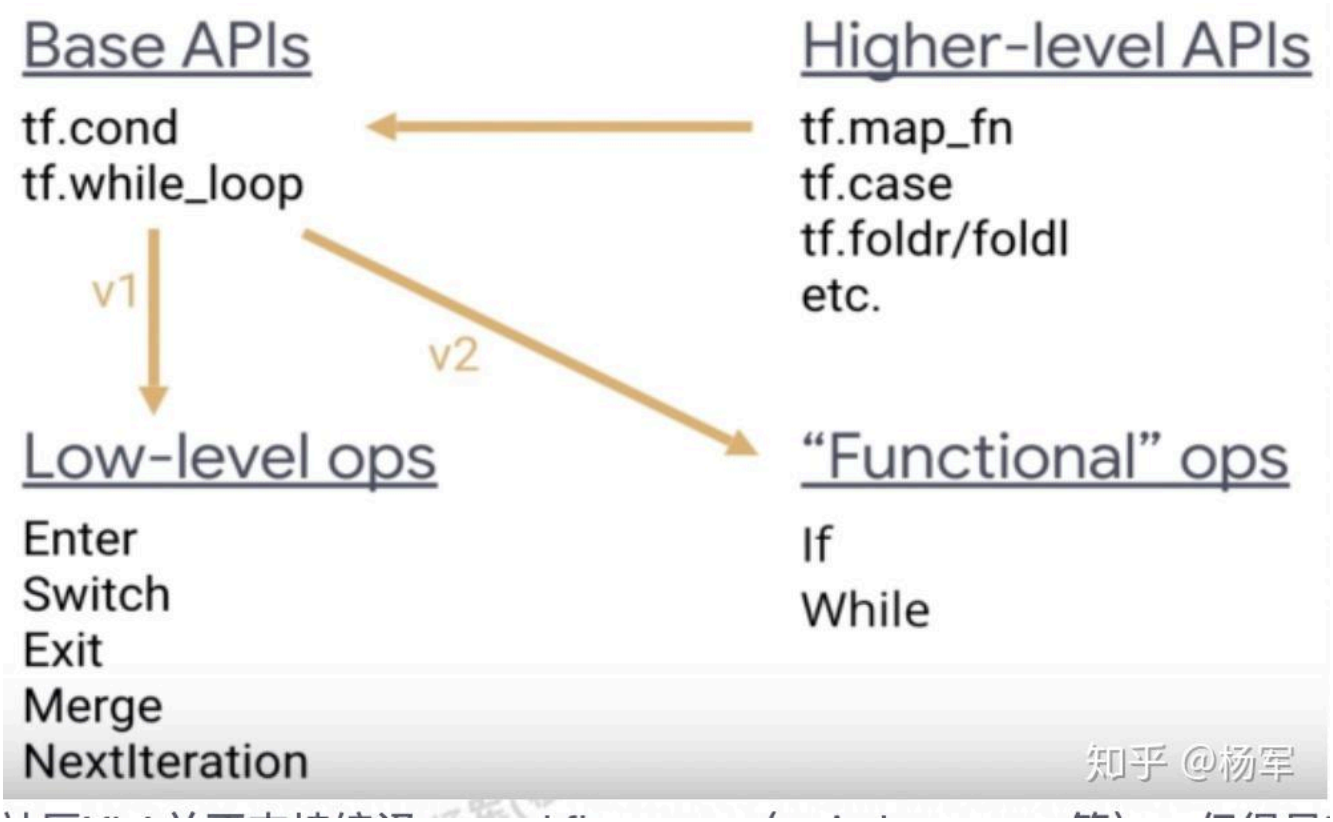
然而在目前TF runtime中一个op输出的动作是一个原子操作。也就是说，从TF的角度看，一个op的所有输出要么都处于没有就绪的状态，要么都处于就绪状态。为了解决这个问题，我们需要将原来的一个xlaLaunchOp (对应一个编译 cluster)进行拆分，增加一些**占位**节点，用来代理输出相应的计算结果。具体而言，我们将原来的一个XlaLaunchOp按照如上图所示的方式进行拆分。实际测试表明OutputProxy具有较低的运行时开销，且可以有效的解决cluster过大导致通信效率下降的问题。

Control Flow 编译

TF中一开始对于control flow的支持使用的是data flow的思想，通过swith/merge/enter/exit/nextiteration等ops来搭建对与if和while的支持，相关的信息可以参见这篇论文。TF后续演进的过程中逐步增加了对functional while/if的支持。functional while/if在图层面是一个单独的op，不再被lower成上述的switch/merge等节点。如图：



由于性能原因目前TF执行时默认使用的仍然是control flow这套实现，但在图层面提供有control flow实现和functional while/if实现之间的相互转换pass。



社区XLA并不支持编译control flow ops （switch、merge等），但很早就支持functional while/If op的编译。其原理类似functional ops的实现，也即将控制逻辑转变为host端的控制流逻辑。

支持编译control flow ops有很多的好处，包括有：

- 节省TF解释执行swith/merge/enter/exit/nextiteration等ops的开销。我们的实验结果表明，在一些具有dynamic encode/decode等结构的模型做推理时这部分开销是主要开销之一。
- 支持control flow编译可以使得编译cluster的粒度变得更大，因而具有更大的潜在性能优化空间。

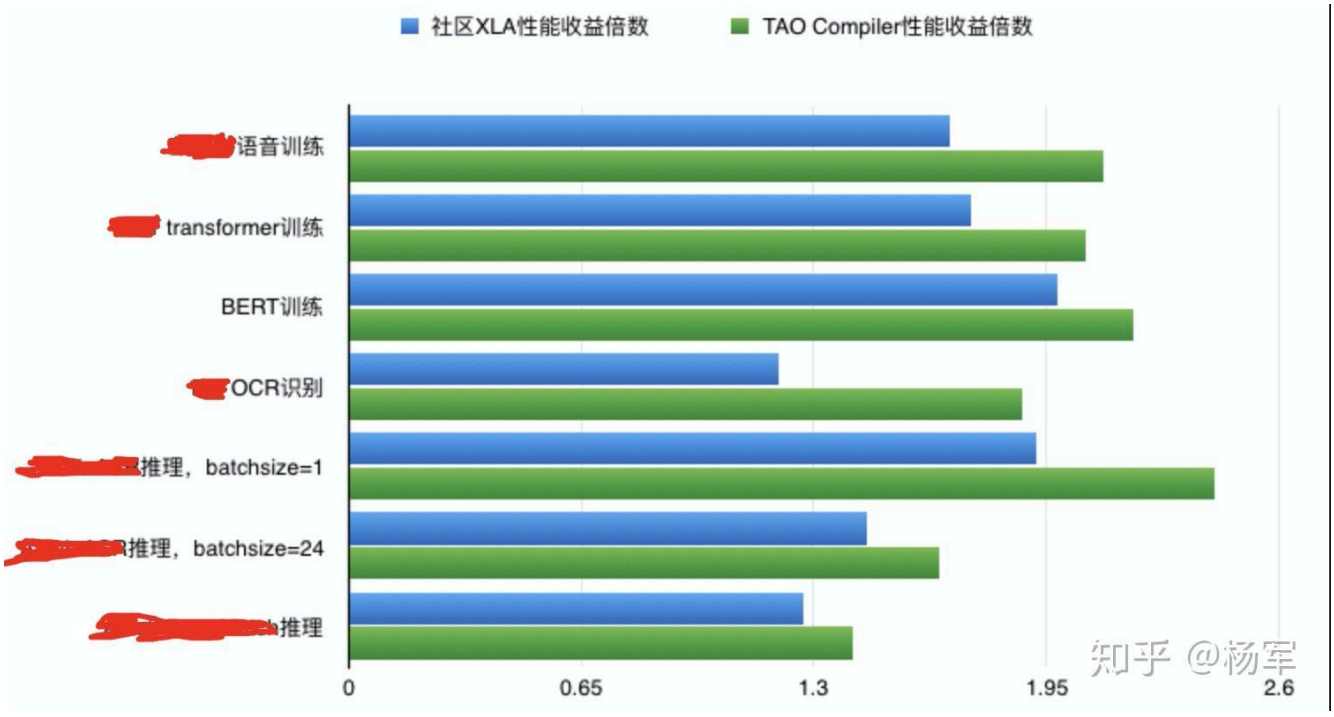
但由于XLA本身compile-time static shape的特性，可以被编译的functional While/If op需要满足以下条件：

- functional while所对应的body function和condition function中的节点（或者functional if所对应的true function和false function中的节点）必须都支持编译；
- functional if所对应的true function和false function的输出的shape需要相等；
- functional while的body function的compile-time-const-input必须是loop invariant；
- functional while的body/cond function中的节点每次迭代时shape需保持不变；

其中1，3可以在图改写阶段静态决议，但shape不变性则需要在runtime阶段检测。我们支持control flow编译的主要工作是在于在改图阶段以及运行时阶段对上述条件进行检查，并将不符合条件的functional while/if ops平滑的回退到普通的control flow ops (swith/merge等)的实现。实验测试表明我们的工作在训练以及推理场景下都取得了符合预期的表现，在一些control flow是主要瓶颈之一的推理模型上取得了30-70%的性能提升。

与社区XLA性能对比

下图为基于本文中所述工作，与社区XLA的性能数字对比，其中性能收益数字，均为相同计算精度下的收益数字。



本篇整体性的介绍了PAI团队在访存密集算子上过往的一些工作，这些工作已经在我们的训练及推理的生产场景大规模打开启用，也取得了不错的业务效果。当前我们还在推进一些更有意思的工作，感兴趣的同学欢迎联系我们，一起来进行AI编译技术的建设打造：）。