AI编译优化--Dynamic Shape Compiler

https://zhuanlan.zhihu.com/p/305546437

None Mon Jun, 14 16:10

作者：PAI团队

背景

本文主要介绍PAI团队在AICompiler中新上线的一套Dynamic Shape Compiler框架，作为AICompiler技术栈中原有的Static Shape Compiler框架的重要补充。团队更多在AICompiler方向的过往工作请关注总纲：

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

从团队三年前启动深度学习编译器方向的工作以来，“Dynamic Shape”问题一直是阻碍实际业务落地的严重问题之一。彼时，包括XLA在内的主流深度学习框架，都是基于Static Shape语义的编译器框架。即，just-in-time运行的编译器，会在运行时捕捉待编译子图的实际输入shape组合，并且为每一个输入shape组合生成一份编译结果。

Static Shape Compiler的优势显而易见，编译期完全已知静态shape信息的情况下，Compiler可以作出更好的优化决策并得到更好的CodeGen性能，同时也能够得到更好的显存/内存优化plan和调度执行plan；然而，Static Shape Compiler的缺点也十分明显，具体包括：

- 编译开销的增加。对于训练业务，编译开销导致训练迭代速度不稳定，训练初期显著负优化，甚至整个训练过程的时间开销负优化；对于Inference业务，很多业务实际部署和迭代时不允许出现性能抖动，而离线的预编译预热又会使得部署的过程变复杂。
- 内存显存占用的增加。除编译开销的问题之外，当shape变化范围特别大的时候，编译缓存额外占用的内存显存，经常导致实际部署环境下的内存/显存OOM，直接阻碍业务的实际落地。
- 对于一部分业务场景，shape变化范围可能非常大甚至是趋于无穷的，比较常见的包括广告推荐类业务中常见的稀疏化模型，还有例如分布式训练下的embedding切片等等。在这种情况下，编译缓存永远也无法收敛，用户也就不可能通过compiler获取到性能收益了。
- 上述问题在部分情况下，可以通过人工干预Compiler的圈图过程来缓解，即，将shape变化剧烈的子图排除在编译范围之外。然而，这种解决办法对用户非常不友好，大大降低了Compiler应用的通用性和透明性，这要求做部署和优化的同学同时对模型结构和compiler非常了解，且每一次模型结构迭代时，都需要花费额外的工作量来调整圈图获得可以接受的性能效果。

关于这一问题，曾经出现过若干类解决方案，包括，

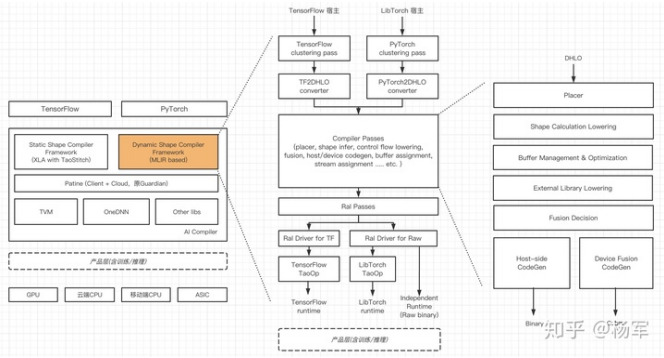
- 对Compiler在圈图过程中的自动化干预；
- 在编译期内部自动对变化维度做bucketing补齐并将子图计算结果做自动的slicing。

然而这些解决方案都存在各自的局限，例如前者只能适配于小部分子图shape变化剧烈的情况，后者在很多模型上都无法得到自动slicing的完备数学推导。

为彻底解决这一问题，我们选择基于MLIR（Multi Layer Intermediate Representation），结合团队过往对AICompiler中积累的部分经验，打造一套完备支持Dynamic Shape语义的AI编译器，希望能够彻底解决深度学习编译器在这部分对灵活性要求较高的业务中无法落地应用的问题。

整体架构

Dynamic Shape Compiler的整体架构，及其在AICompiler中的上下文关系如下图所示。



Compiler部分

MLIR Infrastructure

MLIR是由Google在2019年发起的项目，MLIR 的核心是一套灵活的多层IR基础设施和编译器实用工具库，深受 LLVM 的影响，并重用其许多优秀理念。

这里我们选择基于MLIR的主要原因包括：

- **比较丰富的基础设施支持，**使得完成编译器的常规开发工作更为便捷，效率更好。TableGen，以及编写常规pattern matching的graph optimization pass的简化等。
- **Open for Extension的模块化设计架构，**这里的核心是其Dialect抽象的设计。除Dialect的concept本身，在架构设计上，基于LLVM在传统编译期领域的成功经验，MLIR团队还是展现出了老练的架构设计能力，将整个MLIR架构的设计变得很具模块化。
- **MLIR的胶水能力，**使得其可以比较灵活方便地与已经存在的优化手段进行集成，而非拒斥。

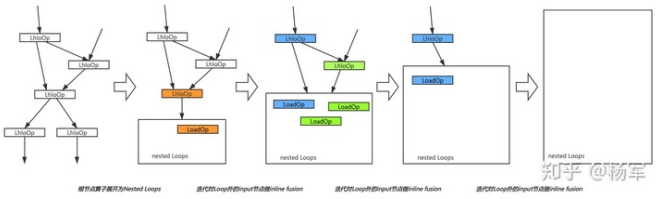
具体实现

MLIR框架的上述特性，使得我们可以比较方便的有选择性的leverage部分社区已有组件，避免完全的重新造轮子，也一定程度上避免从头彻底重构XLA代码带来的巨大工作量。

这里我们根据过往对AI编译器的理解，选择了4层比较主要的中间层抽象，包括：

- DHLO Dialect：能够完备表达动态shape语义的算子层计算图抽象，它的主要作用是能够用有限数量的算子类型来描述不同前端框架的大量算子定义，且表达足够灵活。
- DLHLO Dialect：引入Buffer语义的计算图抽象，用于在编译器流程中进行内存/显存的管理和优化。
- Loop Dialect：用于将算子层的计算描述基于Loop等展开为指令集的计算描述，我们在这一层上完成了算子fusion的CodeGen。
- GPU Dialect：为GPU编程模型中的kernel launching及各种底层原语提供中间层抽象。

下图展示了我们基于MLIR的Loop Dialect等基础设施，在CodeGen中实现最简单的Input fusion的基本原理。对比XLA中只有高层的HLO和底层的llvm两层中间表示，MLIR提供的Loop Dialect抽象可以直接在中间层完成fusion，很好的简化了开发的复杂度。



篇幅原因，我们在次不在赘述Compiler部分其它各个模块的具体实现细节，请感兴趣的同学请移步MLIR社区中发起的相关细节讨论：RFC，以及会议讨论。

此处想着重介绍下对比于XLA，Dynamic Shape Compiler需要额外考虑的一些问题，包括：

- **DHLO IR**，我们在XLA的HLO IR基础上，扩展了一套具有完备动态shape表达能力的IR。静态场景下，HLO IR中的shape表达会被静态化，所有的shape计算会被固化为编译时常量保留在编译结果中；而在动态shape场景下，IR本身需要有足够的能力表达shape计算和动态shape信息的传递。
- **Placer模块**，对于Dynamic Shape Compiler来说，计算可以分为shape计算和data计算两类，对于GPU backend而言，通常shape计算的计算量较小，launch和拷贝开销相比较大因此通常更适合在host侧完成计算。我们实现了一个简单的单卡分图策略，对host侧和device侧计算执行不同的lowering pipeline。
- **Buffer管理及Buffer优化模块**，有别于静态Shape编译期能够比较容易通过liveness分析，实现Buffer的复用等优化，而在动态shape语境下，由于Buffer Size未知编译期则不容易做到完全一致的优化。我们目前使用的是动态的Buffer申请和释放，优化申请和释放的时间点，同时后台使用应用层包含Cache的Allocator，来达到性能和灵活性之间的平衡。后续可考虑在IR中充分表达Shape Constraint信息的情况下，来尝试在编译期做精细的Buffer复用优化。

此外，我们注意到在动态shape语境会为编译期的性能performance带来一些有趣的新挑战：

- **部分优化决策后置到运行期**，以Implicit Broadcast为例，目前主流的前端AI框架都支持implicit broadcast语义，而在动态shape语义下，编译期无法充分知道LHS/RHS是否需要执行Broadcast操作。为保证完备性，如果所有情况下都稳妥的执行Broadcast计算的话，则会带来比较严重的冗余计算/Fusion颗粒度等问题。其它与之类似问题还包括GPU Kernel的Launch Dimension选择等，我们解决这一问题的做法是编译期做多版本编译，运行期根据实际shape来选择最优实现，保证灵活性的同时，缓解灵活性带来的性能损耗。
- **Shape约束信息的使用**，我们发现在Dynamic Shape Compiler中，即使Tensor的Shape信息未知，但Shape之间的约束信息，例如两个Tensor之间的某两个维度的size是否相等等信息，仍然会对编译结果的性能产生比较重要的影响。主要原因包括：在图层面，这些信息带来了更大的图优化空间，而在CodeGen层面，这些信息能够更有效的指导低层Lowering做CSE等传统编译器优化，减少冗余的计算指令数。

多前端框架支持

随着近年来PyTorch用户数量的持续增加，对PyTorch作业的性能优化需求也正在变得越来越重要。AICompiler框架在设计时也包含了扩展支持不同前端框架的考虑，关于解耦框架的考虑请移步这里，本文不再详述。

从IR lowering的角度，这里我们选择相比于HLO更具泛化表达能力的DHLO Dialect作为不同前端框架的统一接入IR，而在PyTorch侧选择用户部署时导出的TorchScript IR，通过实现一个轻量的Converter将TorchScript转换为DHLO IR实现了对PyTorch Inference作业的覆盖。MLIR相对完备的IR基础设施也为Converter的实现提供了便利。

RAL (Runtime Abstraction Layer)

除编译本身的问题之外，我们还面临其它一些问题，例如如何将编译的结果能够配合TensorFlow/LibTorch等宿主在各自的运行环境上下文中执行起来，如何管理运行时IR层不易表达的状态信息等等。我们希望为不同的运行时环境实现一套统一的Compiler架构，为此我们引入了运行时抽象层，即RAL层。RAL层主要负责解决如下问题：

Compile Once and Run Anywhere

RAL实现了多种运行环境的适配支持，用户可以根据需要进行选择。

- **全图编译，独立运行**。当整个计算图都支持编译时，RAL提供了一套简易的runtime以及在此之上RAL Driver的实现，使得compiler编译出来结果可以脱离框架直接运行，减少框架overhead，比如我们在支持某语音ASR模型（类transformer网络）推理优化时，使用全图编译将框架开销从TF的22ms减小到4ms；
- **TF中子图编译运行**。RAL目前实现了TF Driver，可以支持在训练/推理场景中对圈出的子图进行编译执行；
- **Pytorch中子图编译运行**。RAL目前实现了Libtorch Driver，可以支持在推理场景中对圈出子图进行编译执行；

以上环境中在诸如资源（e.g. memory）管理，API语义等上存在差异，希望能够引入一层抽象对compiler侧屏蔽这些差异。**RAL通过抽象出一套最小集合的API（RAL中称为Driver），并清晰的定义出它们的语义，这样compiler和runtime就可以在一定层度上隔离开来，简化compiler的开发，同时通过提供这套API在不同环境下的实现，来达到在不同的环境中都能够执行编译出来的结果的目的**。

Stateless编译

dynamic shape compiler完成一个计算图的编译之后，编译的结果可能被多次执行，而有些op的执行是带状态的：

- 在device（e.g. gpu）上执行时，对const op希望只在第一次执行时加载并常驻device，而不是每次都引入一次host-to-device的拷贝；
- 对于需要根据具体shape信息进行tuning的op（e.g. gemm/conv），tuning cache需要一个地方存储；

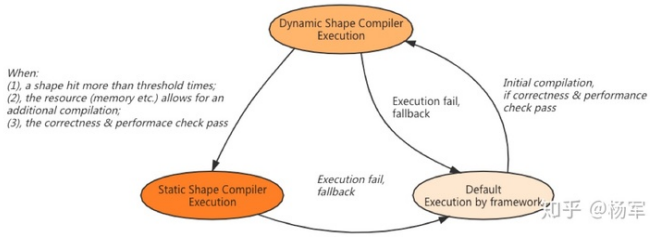
RAL将**资源初始化等带状态的部分抽取出来，封装成context来管理生命周期。在代码生成的过程中，通过简单的注入context，将状态量隐藏在context之后，使得compiler侧看到的是一个纯计算的过程。**无状态的设计一方面简化了代码生成的复杂度，另一方面也更容易支持多线程并发执行（比如推理）的场景，同时在错误处理，回滚方面也更加容易支持。

对用户透明的编译模式切换

我们对于Dynamic Shape Compiler在AICompiler中的定位是：与原Static Shape Compiler并列的一套框架，在允许适度牺牲性能的情况下，提供对于强Dynamic Shape类业务的通用透明支持。

然而从用户的角度来说，通常并不容易判断一个Workload的更适合Dynamic Shape Compiler还是Static Shape Compiler，为此我们结合接耦和全量打开[link]中的工作，设计了一套编译模式自动切换的状态机。其基本思路是，在任务初期先选择较为安全的Dynamic Shape Compiler，结合后台编译让用户能够在运行时尽早得到有性能提升的编译执行，并在后续执行过程中结合资源的实际占用情况和实际运行时的shape变化范围来有选择性的切换到Static Shape Compiler的执行。

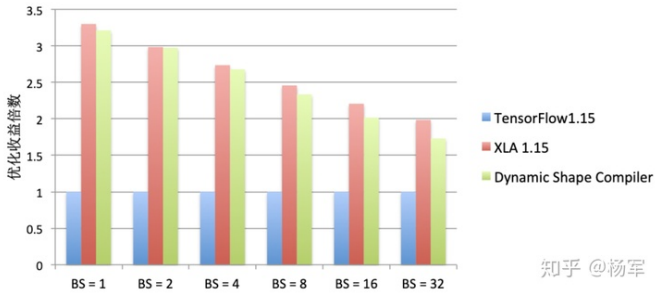
两套compiler在运行时的切换关系如下图所示：



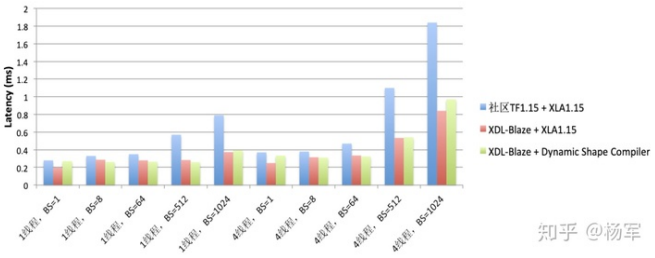
性能结果

以内部一个基于Transformer结构的ASR语音识别模型为例。过往我们为这个业务提供的优化主要基于Static Shape Compiler，但因为shape变化范围较大，只能采用离线预编译的方式来完成部署，部署过程较为繁琐。

下图展示了基于Dynamic Shape Compiler在不同batchsize下的实际性能结果，其中纵轴为latency的提升倍数。整体编译次数从之前的几千次降低到1次。从数字上来看，在只有一次编译的较小编译开销下，性能十分接近Static Shape Compiler的性能优化结果。

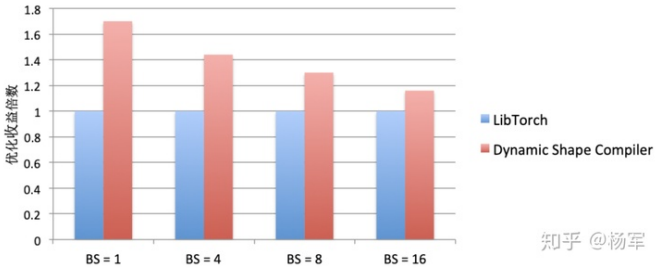


下面这个例子是PAI团队和阿里妈妈XDL广告技术团队合作攻坚的一款排序模型，过往需要通过batching/手工干预圈图等方式才能将编译次数控制到可接受范围内，造成每次模型迭代后部署过程较为繁琐。从性能结果上看，在业务方原有优化工作的基础上，Dynamic shape compiler与基于XLA的代码生成与优化性能基本持平，与业务方的其它优化工作可以正交叠加，并在此基础上大幅简化业务部署难度，通过合作创造更大价值。



具体细节会在和阿里妈妈广告团队的后续合作技术文章中进行详述，此处不再展开。

下图是一个Dynamic Shape Compiler对接PyTorch前端的例子，在某风控场景语音识别模型上的性能优化结果：



当前实现的这套Dynamic Shape Compiler目前虽初具雏形，但仍然可以看到非常多的提升空间。在更多backend硬件支持，Fusion/CodeGen性能，Buffer管理优化，稀疏化算子支持，运行时调度优化等等诸多环节还存在很大提升空间，欢迎各位有兴趣的同学加入进来一起探索，在编译方向上建设一流的AI基础架构！