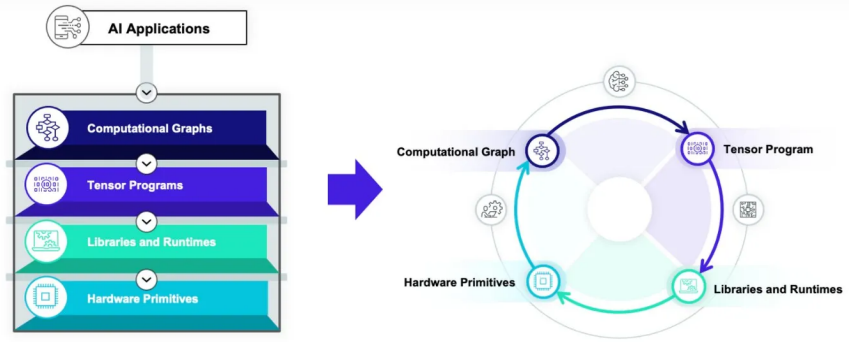


新一代深度学习编译技术变革和展望

陈天奇
CS. PhD 机器学习系统

已关注

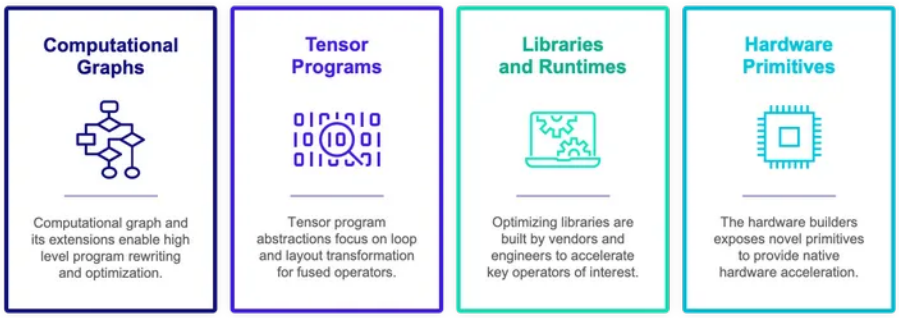
你收藏过 深度学习 相关内容

2021年12月16日，第四次TVMCon如期在线举行。从大约四年多前行业的起点开始，深度学习编译技术也从初期萌芽阶段到了现在行业广泛参与的状况。我们也看到了许多关于编译和优化技术有趣的讨论。深度学习编译技术生态的蓬勃发展也使得我们有了许多新的思考。TVM作为最早打通的深度学习编译技术框架已经可以给大家提供不少价值。但是就好像深度学习框架本身会经历从从第一代（caffe）到下两代（TF, pytorch）的变化一样，我们非常清楚深度学习编译技术本身也需要经历几代的演化。因此从两年前开始我们就开始问这样一个问题：“目前深度学习编译技术的瓶颈在哪里，下一代技术是什么“。

比较幸运的是，作为最早打通编译流程的架构我们可以比较早地直接观察到只有尝试整合才可以看到的经验。而目前开始繁荣的生态本身 (MLIR的各种dialects，XLA, ONNX, TorchScript) 也开始给我们不少可以学习参考的目标。本文是对过去两年全面深度学习编译和硬件加速生态的总结思考，也是我们对于深度学习新一代编译技术的技术展望，希望对大家有一些参考价值。本文的内容大部分来自于今年我们在TVMCon的主题报告。

现状是什么：当前深度学习编译解决方案和瓶颈

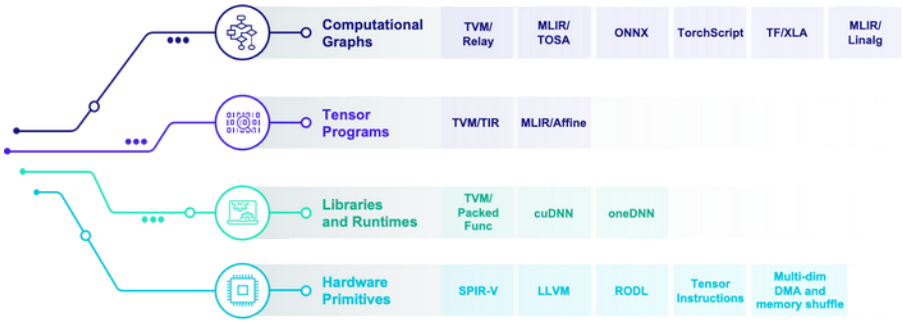
四类抽象



当然深度学习编译加速生态已经从萌芽阶段到开始繁荣生长的阶段。但是抛开具体实现而言，现在深度学习编译生态围绕着**四类抽象**展开:

- 计算图表示(computational graph)：计算图可以把深度学习程序表示成DAG，然后进行类似于算子融合，改写，并行等高级优化。Relay, 》

- 张量程序表示(tensor program): 在这个级别我们需要对子图进行循环优化，对于DSA支持还要包含张量化和内存搬移的优化。
- 算子库和运行环境(library and runtime): 算子库本身依然是我们快速引入专家输入优化性能的方式。同时运行环境快速支持数据结构运行库。
- 硬件专用指令 (hardware primitive) ： 专用硬件和可编程深度学习加速器也引入了我们专用硬件张量指令的需求。

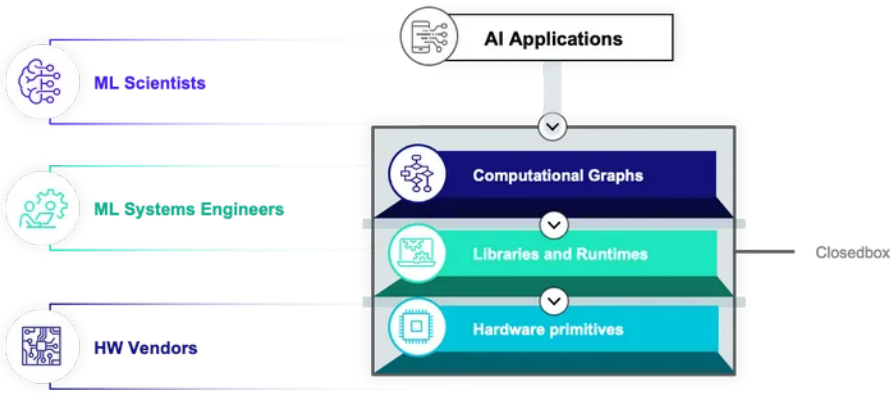


当然的深度学习编译生态基本上也是围绕着各种抽象的实现展开。上图是对于当前生态的一个不完全总结。值得指出的是，算子库和运行环境等本身未必直接和编译技术相关。但是因为我们的目标是新硬件部署和运行加速，运行环境抽象和硬件指令也是生态的重要一环。

我们需要多层抽象本身基本是深度学习编译和优化领域大家达成的一种共识。但是为了真正地支持机器学习，光依赖于各个组件本身是远远不够的。我们发现最大的问题是“如何设计各个层级的抽象，并且对它们进行有效的整合”。

目前的解决方案

如果我们仔细地研究目前的整个生态，包括深度学习框架，编译框架(包括基于MLIR，ONNX或者是TVM的解决方案）。大家都遵循一种叫做多层渐进优化(Multi-stage lowering)的方式。这种构建方式的大致思路是我们在每一层抽象中采用一个（有时多个）中间表示。我们会在每一个层级(dialect, abstraction）做一些内部优化，然后把问题丢给下一个层级继续进行优化。



因为生态本身设计和其他一些原因，当然的解决方案基本有以下特点：每一个层级抽象基本由一个比较独立的团体维护。层和层之间往往比较松耦合。最后解决方案本身往往是以一个黑盒工具的方式呈现给用户。

很多人的一个愿景是只要每一层的优化做的好，把东西拼起来，我们就可以组合成一个满足需求的解决方案。虽然这样的做法的确可以达到一定的效果，但是我们也需要反问，multi-staging lowering真的可足以解决深度学习优化的问题吗?

两种隔阂

我们大约在三年前完成了基于multi-staging lowering的解决方案。当我们把全栈解决方案搭建起来并且不断实践之后我们发现有两种隔阂阻碍整个行业的发展。

其中的第一种**竖向隔阂**阻隔了手工优化的方案和自动编译优化的方案。如果我们看当前的深度学习运行框架和编译框架。我们可以发现有两种流派：一类是以手工算子优化为主的算子库驱动方案。这一类方案一般可以比较容易地让更多的优化专家加入，但是本身也会带来比较多的工程开销。另一类方案是以自动优化为主的编译方案。编译核心方案往往可以带来更多的自动的效果，但是有时候也比较难以引入领域知识。大部分当前的框架基本都只为两者中的其中一个设计。而我们往往发现实际比较好的解决方案其实同时需要机器学习工程师的输入和自动化。并且随着领域的发展，最优的解决方案也会发生变化。怎么样打破这样的竖向墙，让手工优化，机器学习优化专家的知识和自动优化做有机整合，也是目前行业面临的一个大的问题。

除了竖向高墙之外，第二类隔阂则一定程度上和multi-stage lowering的生态直接相关。因为我们往往会把不同层级的抽象分开来设计，虽然在抽象内部可以做比较灵活的优化。但是在一个抽象到另外一个抽象的转换时候往往需要通过translator或者lowering批量转换。这样导致了我们很多困难都开始集中在一类抽象和另外一类抽象的**边界**上，这样导致了如果我们想要在边界上面做一些分步的优化（如把其中一部分子图交给一类编译逻辑，剩下的交给其他编译如逻辑）我们就必须在边界上面引入大量的工程。另外一个常见的现象是这类转换往往是**单向**的，我们一般会在高阶的抽象如计算图上面做一些优化，然后传递给张量计算层级。但是张量计算或者硬件层级的信息往往难以反馈给更高的层级。举个例子，其实很多时候张量程序的优化本身可以反过来指导计算图层级的算子融合和数据排布，但是当前的单向架构比较难自然地利用这一类反馈。

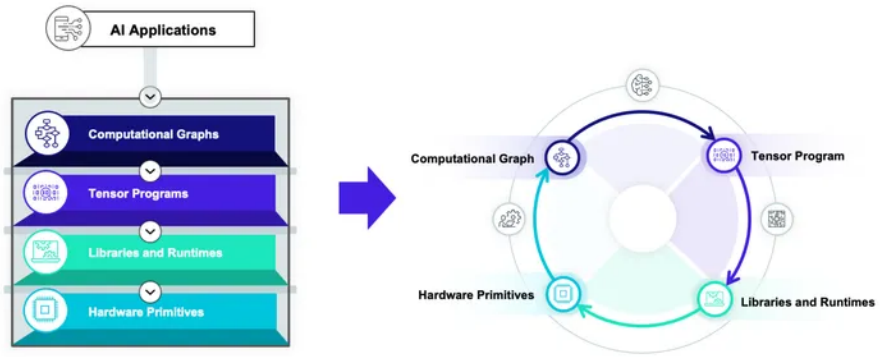
总结一下我们的经验。深度学习编译和优化本身不是一个一个层级可以全部完成优化的问题。解决相关问题需要各个层级抽象之间的联动。随着TVM和MLIR一类基础架构的出现，我们其实已经可以比较容易地搭建出某一个层级的抽象或者dialect并且让它们之间通过multi-stage lowering的方式从高到地级抽象进行逐层变换和转换。但是现在的困难点往往出现在抽象的转换边界上。不论是在边界引入更多的可模块化整合变换，或者是进行尝试反馈迭代，multi-stage lowering本身是远远不够的。不仅如此，随着抽象层级的增加，如果希望加入自定义算子，我们往往需要针对每个抽象层级进行进一步的架构，其中的开销变得更加庞大。

因为各个抽象之间的竖向和横向隔阂。不论我们如何做好一层抽象内部本身，我们依然难以做好端到端的整体优化。需要注意的是，这些隔阂和问题的存在和基础架构的选择无关，不论是基于MLIR，ONNX或者是TVM的方案，一旦采用了multi-stage lowering都会不可避免地面对这个问题。相信在领域里面努力的小伙伴不管采取什么基础架构，在把方案打通之后或多或少都会碰到这个本质的问题。

未来在哪里：从箭头到圈

这一节我们会介绍对于这一个问题经过两年多
系统的核心技术路线，我们把这一路线叫做**TV**

我们可以注意到，几乎所有的难点都由边界隔阂产生，因此我们需要解决的重点是抓住关键点，消除边界隔阂。我们的目标是把一个单向箭头的multi-stage lowering方案，演化成一个可以让各个抽象之间有机交互的一个圈。

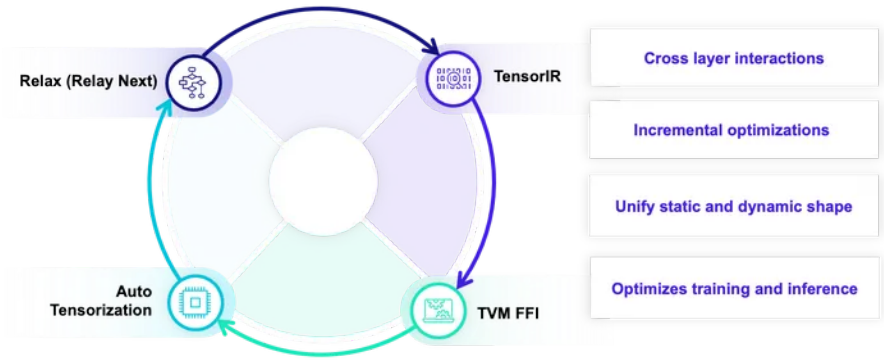


整个技术路线总结下来包含三大关键点:

- Unify: 统一多层抽象
- Interact: 交互开放迭代
- Automate: 自动优化整合

Unify: 统一多层抽象

为了打破抽象直接隔阂，我们需要完成的第一步是统一抽象。这当然并不意味着我们需要设计一个唯一的层级来解决所有问题 – 架构上面稍有不慎，我们可能会设计出一个整合了所有抽象短板的复杂表示。在这里我们依然需要承认每一类抽象的重要性，但是我们需要在不同类别的抽象之间进行协同设计，并且可以让每一个层级和其它层级进行相互交互。



具体到TVM而言，TVM主要的重点放在四个抽象上。AutoTensorization用来解决硬件指令生命和张量程序对接，TVM FFI（PackedFunc）机制使得我们可以灵活地引入任意的算子库和运行库函数并且在各个编译模块和自定义模块里面相互调用。TensorIR负责张量级别程序和硬件张量指令的整合。Relax (Relax Next) 会引入relay的进一步迭代，直接引入first class symbolic shape的支持。但是就和一开始提到的一样，这里的关键点并不只在各个抽象层级本身，而是抽象之间的相互交互和联合优化。

```python
import tvm.script
from tvm.script import tir as T, relax as R

@tvm.script.ir_module
class MyIRModule:
    @T.prim_func
    def tir_mm(X: T.Buffer[(n, d), "float32"],
               W: T.Buffer[(d, m), "float32"],
               Y: T.Buffer[(n, m), "float32"]):
        for i, j, k  in T.grid(n, m, d):
            with T.block("body"):
                vi, vj, vk = T.axis.remap("SSR", [i, j, k])
                with T.init():
                    Y[vi, vj] = 0
                Y[vi, vj] += X[vi, vk] * W[vk, wj]

    @R.function
    def relax_func(x: R.Tensor[(n, d), "float32"], w: R.Tensor[(d, m), "float32"]):
        with R.dataflow()
            lv0: R.Tensor[(n, m), "float32"] = R.call_dps((n, m), tir_mm, [x, w])
            lv1: R.Tensor[(n * m,), "float32"] = R.flatten(lv0)
            gv0: R.Tensor[lv2, "float32"] = R.exp(lv1)
            R.output(gv0)

        R.call_packed("custom_inplace_update", gv0)
        return gv0
```

以上这个程序样例就展示了我们统一抽象的设计目标。这是一个通过TVMScript表示的IRModule。MyIRModule是包含了两种函数。其中tir_mm是一个TensorIR级别的函数。新一代的TensorIR的设计目标是实现 张量程序和硬件专用指令之间的通过自动张量化的联动。再看relax_func这个函数。其中有几个关键点。R.call_dps((n, m), tir_mm, [x, w])是在计算图级别直接对于TensorIR函数tir_mm的直接调用。并且我们通过特殊的调用形式使得张量程序依然以和图算子一样的方式出现在计算图中。支持这一特性意味着计算图需要和张量程序层级表示进行联合设计，使得计算图的优化可以利用张量程序层级的信息。最后R.call_packed("custom_inplace_update", gv0) 允许计算图直接和TVM FFI函数进行交互。

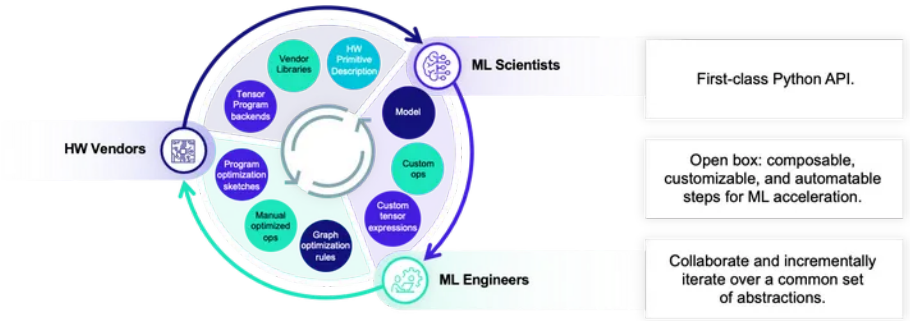
多层抽象之间的相互统一整合虽然带来了更多的一些设计考虑，但是也带来了解决隔阂之后的不少优势。举个例子，假设我们现在有一个快速的算子优化思路，在传统的编译流程中常见的做法是引入在每个层级引入新的改写规则。但是在统一抽象下，我们可以直接引入一个pass把一个局部算子先改写成call_packed调用一个手写的外部算子。在确认性能合理的情况下再考虑如何把一个局部改写变成更加自动化的方案。同样我们也可以把手工算子和自动代码生成的方案更加有机地整合在一起。最后，因为对于TensorIR程序的调用直接被表示再图程序中，我们可以直接把对于TensorIR的改写优化变成对于对应计算图的调用改写，使得张量级别变换的信息可以反馈到图级别的变换中去。

最后，值得注意的是样例中关于动态shape的支持采用了symbolic shape的方案。样例中的(n, m)和(n * m)中的n和m贯穿整个程序。使得我们可以在编译优化时利用这些更多的信息来做更多动态相关优化。并且关于symbolic表达式的支持也完美地和TensorIR层级的symbolic shape统一在了一起。这一特性也很大地体现和协同抽象设计的重要性。

Interact: 交互开放迭代

除了抽象的整合之外，一个非常重要的课题是如何让不同的人之间进行相互协作。深度学习编译优化是一个涉及到各种各样工程师人群的领域。算法专家希望开发新的模型和自定义算子，机器学习系统工程师需要开发系统优化，硬件厂商需要迭代自己的硬件。每一类人都会变对不同层级的抽象。深度学习领域的繁荣很大程度地归功于深度学习框架的开放生态。任何人都可以以python的方式编写我们的模型并且把它们和其人写的模块进行整合。传统的编译器领域往往会一个更加封闭的方式呈现出来。在一个multi-stage lowering的架构下把框架搭好，然后提供一个命令行接口给用户。

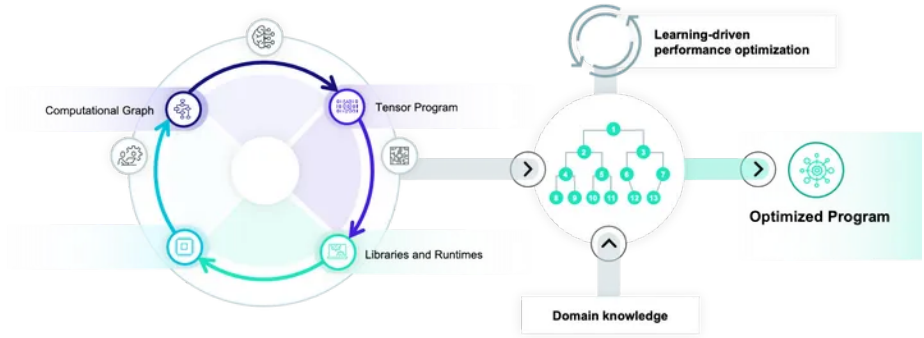
虽然一个美好的愿景是我们可以把每一个层级的抽象做好最好再，然后再把火箭的零件全部拼接起来。但是这样的愿望往往是不现实的，实际的情况是每一层都还是会有自己的问题。如何通过系统工程的方式做好整合互补，并且可以让不同的人群可以在一起相互快速合作迭代出新的解决方案才是我们需要考虑的问题。



而这个时候允许不同的人之间进行协作，交互开发和迭代就是一个我们需要首要考虑的话题。在这个层面上，我们遵循以下几点原则: python-first，通过TVMscript和直接多语言整合的架构让大家都可以通过python API相互协作。开放开发不论是哪个抽象层级之间都可以整合交流。协作迭代，让大家可以一起合作达到更好的效果。举个例子，一个机器学习专家可以利用计算表达式来写一个自定义算子，但这个自定义算子可以被系统专家写的自动转换规则优化，而其中自动转换规则本身又利用的硬件厂商所提供的张量指令特性。当然这个整个链路的每一环都可以在一起快速联动的时候，我们就可以快速地根据需求迭代出想要的解决方案。

Automate: 自动优化整合

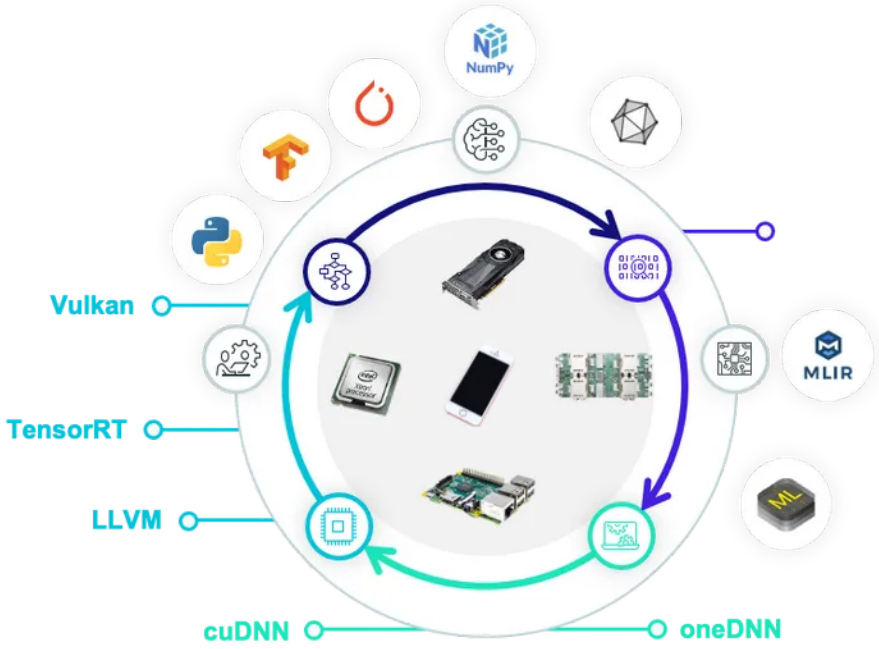
自动优化始终是深度学习编译器血液里面的东西，也是实现Unity的一个关键环节。



很多传统的自动优化方案是排他的，也就是只有整个优化方案全部采用对应的模型的做法（如一些polyhedral model）才可以解决好，一旦尝试引入领域知识，或者想要的优化跳出了原有的范畴，自动化就难以发挥优势。我们需要改变这一观点，重点思考如何在自动优化的过程中有效地整合领域专家知识。使得我们真正可以把自动化和高手之间的力量整合在一起。社区基于TensorIR的MetaSchedule正是往这个方向前进的重要一步。

整体生态整合

前面的几节我们介绍了TVM Unity的三个关键技术点。当然unity本身设计的目标本身并不是解决所有问题。我们非常清楚只有当把这个圈和整个机器学习和硬件生态一起整合的时候我们才可以发挥最大的效率。而抽象的整合使得我们可以更加容易地提供更多种整合方式。我们可以通过子图整合把TVM整合入已有的框架中，也可以把常见的硬件后端整合入TVM本身。统一的抽象可以使得这样的整合以相对统一的方式发生在各个层级。



TVM FFI 等灵活的接口也让我们可以和整个生态做灵活的对接。并且把不同的后端生态通过自动化做更加有效的整合。

总结和未来展望

这篇文章总结了我们对于深度学习编译领域在过去两年的思考和未来的展望。对于新一代架构的推动一直是我们核心关注的主题，这里提到的各个特性也都已经重构完成或者在进行中。TVM FFI在去年已经逐渐成熟，TensorIR本身刚被合并到主干，后续的metaschedule也会陆续进入主干。Relax本身也在社区进行开放开发。TVM Unity作为社区的核心主题也会是接下来一年的重点，并且在开发过程中的一些元素已经可以提供不少好处。当社区逐渐向这一技术方向靠拢对接，整合的优势也会越来越明显。

过去一年中常常看到一些关于深度学习编译基础架构的讨论。其实从核心上面来看，基础架构本身当然是重要的一环（scala之于spark一样），不过不论MLIR, TVM或者其他编译框架基础架构本身也都在相互学习中趋近成熟。真正的瓶颈还是在于抽象的设计和更进一步真正地进行协同整合。当然其实三个技术关键点的选择也的确影响了我们对于基础架构的思考。比如TVM FFI作为比较重要的基础架构支撑了交互和python-first的特性。

不论基础架构，我们面对的真正问题是如何解决这些关键的隔阂问题，multi-stage lowering本身的缺陷导致了现有的方案必须有所突破才能演化到新一代的深度学习编译系统。本文探讨了这一演化的方向和具体的技术路线。希望可以对整个领域有所启发。

在新一代的架构前进的道路上，我们不再是一个人在努力。很多的关键组件设计都是由各个同学一起合作完成。我们也欢迎更多的同学一起加入到社区中来，一起推动实现新一代深度学习编译系统的共同建设中来。

编辑于 2021-12-18 15:30

深度学习（Deep Learning）　TVM　深度学习编译优化

发布一条带图评论吧