The Deep Learning Compiler- A Comprehensive Survey 深度学习编译器综述 （二）

The Deep Learning Compiler- A Comprehensive Survey 深度学习编译器综述 （一）

4. KEY COMPONENTS OF DL COMPILERS

4.1 High-level IR

为了克服传统编译器中采用的IR的限制，因其限制了在DL模型中使用的复杂计算的表达，现有的DL编译器利用了高级IR（也称为图形IR），并进行了特殊设计以实现高效的代码优化。为了更好地理解DL编译器中使用的图形IR，我们将以下列方式描述图形IR的表示和实现。

4.1.1 Representation of Graph IR

图形IR的表示形式影响图形IR的表达能力，并决定了DL编译器分析图形IR的方式。

DAG-based IR

基于DAG（有向无环图）的IR是编译器构建计算图的最传统方式之一，其中节点和边被组织为一个有向无环图。在DL编译器中，DAG的节点表示原子的DL运算符（如卷积、池化等），而边表示张量。该图是无循环的，没有循环存在，这与通用编译器的数据依赖图（DDG）[50]不同。借助DAG计算图，DL编译器可以分析各种运算符之间的关系和依赖关系，并利用它们来指导优化。在DDG上已经有了许多优化技术，例如公共子表达式消除（CSE）和死代码消除（DCE）等。通过结合DL的领域知识和这些算法，可以对DAG计算图进行进一步的优化，这将在第4.3节中详细介绍。基于DAG的IR由于其简洁性，在编程和编译方面非常方便。然而，它也存在一些缺陷，例如由于计算范围缺失的定义而引起的语义模糊。

DAG-based IR有它的优点，比如自动微分非常容易实现，但也有缺点，比如内部无法包含控制流，即if或其他控制语句。

```
A = f(B)
C = g(D, A)
E = h(C)
```

Let-binding-based IR

通过提供let表达式来解决语义模糊是一种解决方法，许多高级编程语言如Javascript [30]、F# [76]和Scheme [3]都使用了这种方法。使用let关键字定义一个表达式时，会生成一个let节点，然后该节点指向表达式中的运算符和变量，而不仅仅是在变量之间建立计算关系，这区别

于只构建基于DAG的计算图。在基于DAG的编译器中，当一个过程需要获取一个表达式的返回值时，它首先访问相应的节点并搜索相关节点，也称为递归下降技术。相比之下，基于let-binding的编译器计算出let表达式中的所有变量的结果，并建立一个变量映射。当需要特定的结果时，编译器查找该映射以确定表达式的结果。在DL编译器中，TVM的Relay IR [78]同时采用了基于DAG的IR和基于let-binding的IR，以获得双方的优点。

相比于DAG-based IR，let-binding-based IR的表达能力更强：基于let-binding的IR可以更自然地表达复杂的计算逻辑和数据依赖关系，尤其适用于函数式编程语言。它可以轻松地表示函数调用、递归、条件表达式等复杂的计算结构。基于let-binding的IR更容易进行静态类型推断，因为所有的变量和表达式都明确地绑定了类型。

```
let
   A = f(B)
   C = g(D, A)
   E = h(C)
in
   E
```

Representing Tensor Computation

不同的图形IR有不同的方法来表示张量上的计算。各种DL框架的运算符根据这些特定的表示形式被转化为图形IR。而定制化的运算符也需要在这样的表示中进行编程。张量计算的表示可以分为以下三个类别。

1) Function-based

函数式表示是一种提供封装运算符的表示方法，这种方法被Glow、nGraph和XLA采用。以XLA的IR（High-Level Optimizer，HLO）为例，它由一组符号编程中的函数组成，其中大部分函数没有副作用。指令被组织成三个层次，包括HloModule（整个程序）、HloComputaion（一个函数）和HloInstruction（操作）。XLA使用HLO IR来表示图形IR和操作IR，因此HLO的操作范围从数据流级别一直到操作符级别。

2) Lambda expression

Lambda表达式是一个索引公式表达式，通过变量绑定和替换来描述计算过程。使用Lambda表达式，程序员可以快速定义计算过程，而不需要实现一个新的函数。TVM使用基于Lambda表达式的张量表达式来表示张量计算。在TVM中，张量表达式中的计算运算符是由输出张量的形状和计算规则的Lambda表达式定义的。

3) Einstein notation

爱因斯坦求和符号，也被称为求和约定，是一种表示求和的记法。它的编程简洁性优于Lambda表达式。以TC为例，临时变量的指标无需定义。IR可以根据爱因斯坦求和符号中未定义变量的出现来确定实际的表达式。在爱因斯坦求和符号中，操作符需要满足关联性和交换性。这种限制条件确保了归约操作可以按任意顺序执行，从而实现进一步的并行化。

4.1.2 Implementation of Graph IR

在DL编译器中，Implementation of Graph IR实现了对数据和操作的管理。

Data representation

在DL编译器中，数据（如输入、权重和中间数据）通常以张量的形式进行组织，也被称为多维数组。DL编译器可以通过内存指针直接表示张量数据，或者以更灵活的方式使用占位符进行表示。占位符包含张量每个维度的大小。另外，张量的维度大小也可以被标记为未知。为了进行优化，DL编译器需要数据布局信息。此外，迭代器的范围也需要根据占位符进行推断。

1) Placeholder

占位符在符号编程（例如Lisp和TensorFlow）中被广泛使用。占位符是一个带有明确形状信息（例如每个维度的大小）的变量，它将在计算的后期阶段被填充具体的值。它允许程序员描述操作并构建计算图，而无需关注具体的数据元素，这有助于将计算定义与DL编译器中的确切执行分离开来。此外，使用占位符，程序员可以方便地通过改变输入/输出和其他对应的中间数据的形状来修改张量的形状，而无需更改计算定义。

2) Unknown (Dynamic) shape representation

通常在声明占位符时，支持未知维度大小。例如，TVM使用Any表示未知维度（例如，Tensor⟨(Any,3), fp32⟩）；XLA使用None来实现同样的目的（例如，tf.placeholder("float", [None, 3])）；nGraph使用其PartialShape类。未知形状表示对于支持动态模型是必要的。然而，要完全支持动态模型，必须放宽边界推断和维度检测，同时还需要实施额外的机制来确保内存的有效性。

3) Data layout

数据布局描述了张量在内存中的组织方式，通常是从逻辑索引到内存索引的映射关系。数据布局通常包括维度的顺序（例如NCHW和NHWC）、切片、填充、步长等信息。TVM和Glow将数据布局表示为操作符的参数，并要求此类信息用于计算和优化。然而，将数据布局信息与操作符而不是张量结合在一起，能够直观地实现某些操作符，并减少编译开销。XLA将数据布局表示为与其后端硬件相关的约束条件。Relay和MLIR正计划将数据布局信息添加到它们的张量类型系统中。

4) Bound inference

在DL编译器中编译DL模型时，边界推断被应用于确定迭代器的边界。尽管DL编译器中的张量表示对于描述输入和输出很方便，但它也为推断迭代器的边界带来了特殊的挑战。边界推断通常根据计算图和已知的占位符进行递归或迭代的执行。例如，在TVM中，迭代器形成了一个有向无环超图，其中图的每个节点表示一个迭代器，每个超边表示两个或多个迭代器之间的关系（例如，分割、融合或重新基准）。一旦根迭代器的边界根据占位符的形状确定，可以根据关系递归地推断其他迭代器的边界。

Operators supported

DL编译器支持的操作符负责表示DL工作负载，并且它们是计算图的节点。这些操作符通常包括代数运算符（例如+、×、exp和topK）、神经网络运算符（例如卷积和池化）、张量运算符（例如改变形状、调整大小和复制）、广播和约简运算符（例如min和argmin），以及控制流运算符（例如条件语句和循环）。在这里，我们选择三个在不同的DL编译器中经常使用的代表性操作符进行说明。此外，我们还讨论了自定义操作符的情况。

1) Broadcast

广播操作符可以复制数据并生成具有兼容形状的新数据。如果。例如，对于一个加法操作

符，期望输入张量具有相同的形状。一些编译器（如XLA和Relay）通过提供**广播**操作符来放宽这种限制。例如，XLA允许通过复制向量直到其形状与矩阵匹配，来对矩阵和向量进行逐元素相加。

2) Control flow

在表示复杂和灵活的模型时，需要使用控制流。模型例如循环神经网络（RNN）和强化学习（RL）依赖于递归关系和数据相关的条件执行[103]，这就需要使用控制流。在DL编译器的图形IR中不支持控制流的情况下，这些模型必须依赖主机语言（例如Python中的if和while）的控制流支持或静态展开，这会降低计算效率。Relay注意到任意的控制流可以通过递归和模式来实现，这已经在函数式编程中得到了证明[78]。因此，它提供了if操作符和递归函数来实现控制流。相反，XLA通过特殊的HLO操作符（如while和conditional）来表示控制流。

3) Derivative

一个操作符Op的导数操作符将Op的输出梯度和Op的输入数据作为输入，并计算Op的梯度。虽然一些DL编译器（如TVM和TC）支持自动微分[88]，但在应用链式法则时，它们要求在高级IR中得到所有操作符的导数。TVM正在努力提供代数运算符和神经网络运算符的导数操作符。程序员可以使用这些导数操作符构建自定义操作符的导数。相反，PlaidML可以自动生成导数操作符，即使对于自定义操作符也可以。值得注意的是，无法支持导数操作符的DL编译器无法提供模型训练的能力。

4) Customized operators

自定义操作符的支持可以让程序员为特定的目的定义他们自己的操作符。支持自定义操作符可以提高DL编译器的可扩展性。例如，在Glow中定义新的操作符时，程序员需要实现逻辑和节点封装。另外，如果需要的话，还需要进行降低步骤、操作IR生成和指令生成等额外的工作。而TVM和TC除了描述计算实现之外，所需的程序编写工作较少。具体来说，TVM的用户只需要描述计算和调度，声明输入输出张量的形状即可。此外，自定义操作符通过钩子集成了Python函数，进一步减轻了程序员的负担。

4.1.3 Discussion

几乎所有的DL编译器都有自己独特的高级IR。然而，它们共享相似的设计理念，例如使用DAG和let-binding来构建计算图。此外，它们通常为程序员提供方便的方法来表示张量计算。高级IR中数据和操作符的设计足够灵活和可扩展，以支持多样化的DL模型。更重要的是，高级IR是与硬件无关的，因此可以与不同的硬件后端进行搭配使用。

4.2 Low-level IR

4.2.1 Implementation of Low-Level IR

低级IR以比高级IR更细粒度的表示方式描述了DL模型的计算过程，通过提供调整计算和内存访问的接口，使得可以进行面向目标的优化。在本节中，我们将常见的低级IR实现分类为三类：**基于Halide的IR、基于多面体的IR和其他独特的IR。**

Halide-based IR

Halide最初被提出用于并行化图像处理，并且在DL编译器（如TVM）中证明了其可扩展性和高效性。Halide的基本理念是**将计算和调度分离**。与直接给出一个特定方案不同，采用Halide的编译器尝试各种可能的调度，并选择最佳方案。在Halide中，内存引用和循环嵌套的边界被限制为与轴对齐的有界框。因此，Halide不能表示具有复杂模式的计算（例如非矩形）。幸运的是，DL中的计算非常规律，非常适合使用Halide完美地表示。此外，Halide可以轻松地对这些边界进行参数化，并将它们暴露给tuning机制。将Halide的原始IR应用于DL编译器的后端时需要进行修改。例如，Halide的输入形状是无限的，而DL编译器需要知道数据的确切形状，以便将操作符映射到硬件指令上。一些编译器，如**TC(Tensor Comprehensions)**，要求数据具有固定的大小，以确保张量数据的时间局部性更好。

TVM通过以下努力将Halide IR改进为一个独立的符号性IR。它去除了对LLVM的依赖，并对Halide的项目模块和IR设计进行了重构，追求更好的组织性和对图形IR和前端语言（如Python）的可访问性。它还提高了可重用性，通过实现运行时调度机制方便地添加自定义操作符。TVM将变量定义从字符串匹配简化为指针匹配，**确保每个变量具有单一的定义位置（静态单赋值，SSA）**[22]。

Polyhedral-based IR

多面体模型是DL编译器中采用的重要技术。它使用线性规划、仿射变换和其他数学方法来优化具有静态控制流的循环代码，以及边界和分支。与Halide相比，在多面体模型中，内存引用和循环嵌套的边界可以是任意形状的多面体。这种灵活性使得多面体模型在通用编译器中被广泛使用。然而，这种灵活性也阻碍了与调优（tuning）机制的集成。尽管如此，由于可以处理深度嵌套的循环，许多DL编译器（如TC和PlaidML（作为nGraph的后端））已经采用了多面体模型作为它们的低级IR。基于多面体的IR使得应用各种多面体变换（如融合、切片、下沉和映射）变得容易，包括设备相关和设备无关的优化。有许多多面体模型编译器借用的工具链，如isl [96]、Omega [48]、PIP [23]、Polylib [60]和PPL [9]。

TC在低级IR中具有其独特的设计，结合了Halide和多面体模型。**它使用基于Halide的IR来表示计算，并采用基于多面体的IR来表示循环结构**。TC通过抽象实例来呈现详细的表达式，并引入特定的节点类型。简而言之，TC使用domain节点来指定索引变量的范围，使用context节点来描述与硬件相关的新迭代变量。filter节点表示一个与语句实例组合的迭代器。set和sequence是用于指定filters执行类型（并行和串行执行）的关键字。此外，TC使用extension节点来描述代码生成中的其他必要指令，例如内存移动。

PlaidML使用基于**多面体的IR**（称为Stripe）来表示张量操作。它通过扩展并行多面体块的嵌套到多个级别来创建一层层可并行化的代码层次结构。此外，它允许将嵌套的多面体分配给嵌套的内存单元，以提供一种将计算与内存层次结构匹配的方式。在Stripe中，硬件配置独立于内核代码。Stripe中的tags（在其他编译器中称为passes）不会改变内核结构，但提供了有关优化pass的硬件目标附加信息。Stripe将深度学习运算符拆分为适合本地硬件资源的tiles。

Other unique IR

有一些深度学习编译器使用自定义的低级中间表示（low-level IRs），而没有使用Halide和多面体模型。在这些自定义的低级IR上，它们应用特定于硬件的优化并lowers为LLVM IR。

Glow中的低级IR是基于指令的表达式，它对由地址引用的张量进行操作[79]。Glow低级IR中有两种类型的**基于指令的函数：声明和程序**。第一个函数声明了在程序的整个生命周期中存在的常量内存区域的数量（例如输入、权重、偏置）。第二个函数是一个包含函数（例如卷积和池化）和临时变量的列表，以在本地分配的区域中执行操作。指令可以在全局内存区域或本

地分配的区域上运行。此外，每个操作数都带有其中一个限定符：@in表示操作数从缓冲区读取；@out表示操作数向缓冲区写入；@inout表示操作数对缓冲区进行读写。这些指令和操作数限定符帮助Glow确定何时可以执行某些内存优化。

MLIR受到LLVM的很大影响，它是一个比LLVM更纯粹的编译器基础设施。MLIR在LLVM中复用了许多思想和接口，它位于模型表示和代码生成之间。MLIR拥有一个灵活的类型系统，允许多个抽象级别，并引入了方言来表示这些多个抽象级别。每个方言由一组定义的不可变操作组成。目前，MLIR的方言包括TensorFlow IR、XLA HLO IR、实验性多面体IR、LLVM IR和TensorFlow Lite。还支持方言之间的灵活转换。此外，MLIR可以创建新的方言来与新的低级编译器连接，为硬件开发人员和编译器研究人员铺平道路。

XLA的HLO IR可以同时被视为高级IR和低级IR，因为HLO足够细粒度地表示硬件特定信息。此外，HLO支持特定于硬件的优化，并可用于生成LLVM IR。

4.2.2 Code Generation based on Low-Level IR

大多数深度学习编译器采用的低级IR最终可以转换为LLVM IR，并受益于LLVM成熟的优化器和代码生成器。此外，LLVM还可以从零开始明确设计专用加速器的自定义指令集。然而，传统的编译器直接生成LLVM IR时可能会生成较差的代码。为了避免这种情况，**深度学习编译器采用了两种方法来实现硬件相关的优化：1）在LLVM的上层IR（例如基于Halide的IR和基于多面体的IR）中执行特定于目标的循环转换，以及2）为优化passes提供有关硬件目标的额外信息。**大多数深度学习编译器都同时应用了这两种方法，但重点不同。一般来说，更注重前端用户的深度学习编译器（例如TC、TVM、XLA和nGraph）可能会重点关注方法1），而更倾向于后端开发人员的深度学习编译器（例如Glow、PlaidML和MLIR）可能会重点关注方法2）。

在深度学习编译器中，compilation scheme主要可以分为两类：即时编译（JIT）和提前编译（AOT）。对于JIT编译器，它可以即时生成可执行代码，并且可以通过更好的运行时知识对代码进行优化。AOT编译器首先生成所有可执行二进制文件，然后再执行它们。因此，它们在静态分析方面的范围比JIT编译更大。此外，AOT方法可以与嵌入式平台的交叉编译器（例如C-GOOD [46]）一起使用，也可以在远程机器上进行执行（TVM RPC）并支持定制加速器。

4.2.3 Discussion

在深度学习编译器中，低级IR是对深度学习模型的细粒度表示，它反映了深度学习模型在各种硬件上的详细实现。低级IR包括基于Halide的IR、基于多面体的IR和其他独特的IR。尽管它们在设计上有所不同，但它们利用成熟的编译器工具链和基础设施，提供定制的硬件特定优化和代码生成接口。低级IR的设计还可以影响新的深度学习加速器的设计（例如TVM HalideIR和Inferentia，以及XLA HLO和TPU）。