



<二> 深度学习编译器综述: High-Level IR (1)

赞同 3



分享

<二> 深度学习编译器综述: High-Level IR (1)

算树平均数
昏昏沉沉工程师 (寻职中)

[关注他](#)

★ 你收藏过 深度学习 相关内容

[<一> 深度学习编译器综述: Abstract & Introduction](#)

[<二> 深度学习编译器综述: High-Level IR \(1\)](#)

[<三> 深度学习编译器综述: High-Level IR \(2\)](#)

[<四> 深度学习编译器综述: Low-Level IR](#)

[<五> 深度学习编译器综述: Frontend Optimizations](#)

[<六> 深度学习编译器综述: Backend Optimizations\(1\)](#)

[<七> 深度学习编译器综述: Backend Optimizations\(2\)](#)

The Deep Learning Compiler: A Comprehensive Survey

The Deep Learning Compiler: A Comprehensive Survey

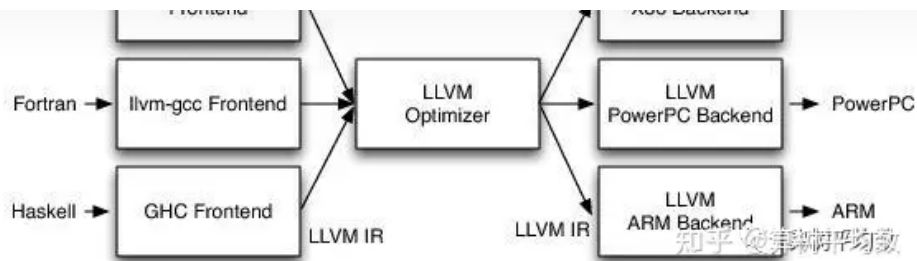
MINGZHEN LI*, YI LIU*, XIAOYAN LIU*, QINGXIAO SUN*, XIN YOU*, HAILONG YANG*[†], ZHONGZHI LUAN*, LIN GAN[§], GUANGWEN YANG[§], and DEPEIQIAN*, Beihang University* and Tsinghua University[§]

知乎 @算树平均数

DL Compiler综述解读的第二Part将围绕的IR (Intermediate Representation, 中间表示), 与传统编译器一样, 也是作为编译器在将高级源代码翻译为底层机器代码的过程中使用的一种抽象表示形式。

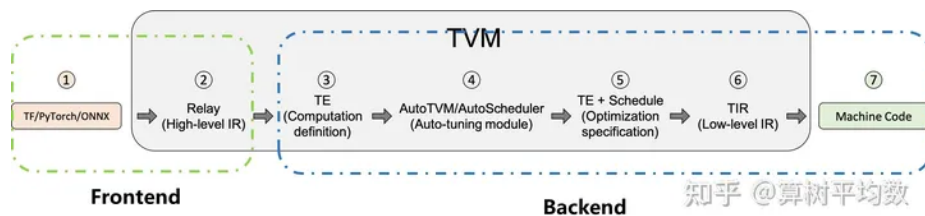
IR 扮演着连接源代码和目标机器代码之间的桥梁。

下图是LLVM IR在LLVM中的地位: **连接Frontend和 Backend**, 关于传统编译器就不展开了, 需要相关补充知识推荐看看编译原理 (龙书)



关于DL Compiler，我们先提出引例，先知道大概每一步做什么了工作，再自顶向下地详细了解各个部件。

以下图TVM架构为例，简单介绍深度学习模型怎么通过DL Compiler进行优化，最终形成Machine Code。



- (1) 输入 (TF/Pytorch/ONNX): 深度学习框架导出的模型文件
- (2): 首先解析模型文件转化为一种更易于优化的中间表示 (如Relay、TE等)，这个中间表示能够表示模型的操作、结构和计算图，Relay 用于描述整个深度学习模型的结构和操作。
- (3)(4)(5)而 TE 用于描述张量计算的操作，执行一些底层优化。
- (6)TIR: 作为lower的IR，充当了连接前端 (如TE、Relay) 和后端之间的桥梁，在TIR生成后，可以在TIR层面执行一些优化，如循环优化、内存访问模式优化等。这些优化可以在TIR表示上直接操作，以提高生成代码的性能。
- (7)Machine Code: TVM的后端会将TIR表示转化为底层硬件平台上的机器代码，过程包括：**指令选择，寄存器分配，指令调度，指令编码，后处理，生成目标文件。**

引例结束，现在你应该知道了TVM做了什么工作了吧。

回归论文综述，开始今天的正题：**IR (Intermediate Representation, 中间表示)**

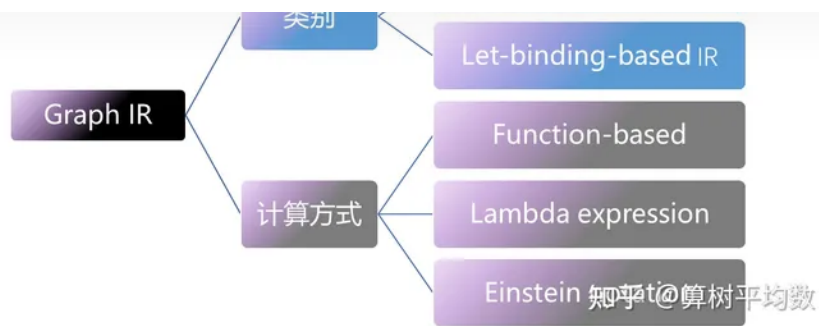
KEY COMPONENTS OF DL COMPILERS

1. High-level IR

为了克服传统编译器中采用的 IR 限制了 DL 模型中使用的复杂计算的表达，现有的 DL 编译器利用高级 IR (称为) 和特殊设计来实现高效的代码优化。

1.1 Representation of Graph IR

Graph IR的表示影响Graph IR的表达能力，也决定DL编译器分析图IR的方式。



1. DAG-based IR:

DAG-Based IR (Directed Acyclic Graph-Based Intermediate Representation, 有向无环图中间表示) 是一种编译器中常用的中间表示形式。

在DAG-Based IR中, 节点表示操作 (卷积、池化等), 边表示Tensor。

下图为简单的示例, 描述了一个加法和乘法操作的DAG-Based IR



DAG-Based IR也是无循环的非循环图, 与通用编译器的数据依赖图 (Data Dependence Graphs, DDG) 相比, 深度学习编译器可以分析各种算子之间的关系和依赖关系, 并用它们来指导优化。

DDG 上已经有很多优化, 例如公共子表达式消除 (CSE) 和死代码消除 (DCE), 通过将DL的领域知识与这些算法相结合, 可以将进一步的优化应用于 DAG-Based IR。

- 优点: 简单, 便于编程和编译
- 缺点: 但存在计算范围定义缺失导致语义模糊等缺陷

计算范围定义缺失导致语义模糊:

在一些情况下, DAG-Based IR可能无法清晰地表达操作在代码中的计算范围。

例如, 在一个循环结构内部, 操作的计算范围可能因循环的迭代次数而变化, 但DAG-Based IR可能无法直接表示这种情况。这可能导致在进行一些优化时, 无法精确地理解操作的计算范围, 从而影响了优化的有效性。

这种模糊性可能需要在DAG-Based IR之上引入其他表示层, 或者进行额外的分析, 以便准确地定义操作的计算范围。

一些编译器可能会使用更高级的中间表示 (如SSA形式、静态单赋值形式等) 来更好地定义操作的作用域, 以便在后续的优化中能够更精确地分析和转换代码。

2. Let-binding-based IR:

或表达式。

在编程中, "Let-binding" 允许您创建一个变量, 将其初始化为一个值, 并在特定的范围内使用这个变量。这有助于提高代码的可读性和可维护性。

在Python中, 您可以使用 "Let-binding" 的概念通过赋值语句来创建和初始化变量。例如:

```
x = 10
y = x + 5
result = y * 2
```

在上面的代码中, 我们使用 "Let-binding" 的方式通过赋值语句引入了变量 x、y 和 result, 并将它们绑定到不同的值和表达式。这样, 我们可以在后续的代码中使用这些变量, 而不需要重复计算值。

类似地, 在函数式编程语言中, "Let-binding" 可以更为明确地表示为:

```
let x = 10
    y = x + 5
    result = y * 2
in
    result
```

在这个Haskell语言的示例中, "Let-binding" 通过使用 "let" 关键字引入了变量 x、y 和 result, 并在 "in" 子句中使用这些变量。这种方式使得代码的计算过程和依赖关系更加清晰。总之, "Let-binding" 是一种通过赋值或绑定的方式引入变量, 并将其绑定到特定的值或表达式的编程概念。

它有助于提高代码的可读性和可维护性, 同时也在中间表示中引入了一个中间变量来明确表示操作的计算过程。

"Let-binding"是解决语义歧义的一种方法, 当使用let关键字定义表达式时, 会生成一个let节点, 然后它指向表达式中的运算符和变量, 而不仅仅是像DAG一样构建变量之间的计算关系。

在基于DAG的编译器中, 当计算需要获取某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。

相反, 基于Let-binding的编译器计算出let表达式中变量的所有结果并构建变量映射。当需要特定结果时, 编译器会查找此映射来决定表达式的结果。

在DL编译器中, TVM的Relay IR同时采用了**DAG-based IR**和**Let-binding-based IR**, 以获得两者的优点。

Representing Tensor Computation:

不同的Grap IR 有不同的方式来表示张量的计算。根据这种特定的表示, 不同的深度学习框架的算子被转换为图IR。并且定制的算子也需要以这种表示形式进行编程。张量计算的表示可以分为以下三类。

1 . Function-based::

基于函数的表示只是提供封装的运算符, Glow、nGraph和XLA均采用这种表示。以高级优化器(HLO, XLA的IR)为例, 它由一组符号编程的函数组成, 并且大多数没有副作用。

指令分为三个级别, 包括 HloModule (整个程序)、HloComputaion (函数) 和 HloInstruction (操作) 。

XLA使用HLO IR来表示图IR和操作IR, 使得HL

我们可以用XLA的 "Function-based" 表示来表示这个操作:

```
import jax.numpy as jnp
from jax import jit

# 定义一个函数，表示矩阵相加并乘以标量的操作
@jit
def matrix_operation(a, b, scalar):
    result = (a + b) * scalar
    return result

# 输入数据
matrix_a = jnp.array([[1, 2], [3, 4]])
matrix_b = jnp.array([[5, 6], [7, 8]])
scalar = 2

# 调用函数执行操作
output = matrix_operation(matrix_a, matrix_b, scalar)

print(output)
```

2. Lambda expression:

Lambda表达式是一种索引公式表达式，描述变量绑定和替换的计算。使用 lambda 表达式，程序员可以快速定义计算，而无需实现新函数。TVM表示使用张量表达式进行张量计算，该表达式基于lambda表达式。

在TVM中，张量表达式中的计算运算符由输出张量的形状和计算规则的lambda表达式定义。

让我通过一个简单的例子来展示TVM中的Lambda表达式：

假设我们有两个输入矩阵 A 和 B，我们想要将它们相加，并将结果存储在输出矩阵 C 中。

在TVM中，我们可以使用Lambda表达式来表示这个相加的操作：

```
import tvm
from tvm import te

# 定义输入矩阵维度
M, N = 2, 2

# 创建TVM计算图上的符号变量
A = te.placeholder((M, N), name='A')
B = te.placeholder((M, N), name='B')

# 使用Lambda表达式定义相加操作
C = te.compute((M, N), lambda i, j: A[i, j] + B[i, j], name='C')

# 创建TVM的调度器
s = te.create_schedule(C.op)

# 编译计算图并执行
func = tvm.build(s, [A, B, C], "llvm")
ctx = tvm.Device("llvm", 0)
a = tvm.nd.array([[1, 2], [3, 4]], ctx)
b = tvm.nd.array([[5, 6], [7, 8]], ctx)
c = tvm.nd.empty((M, N), ctx)
func(a, b, c)
print(c.asnumpy())
...
```

爱因斯坦符号又称为求和约定，是一种表达求和的符号。它的编程简单性优于lambda表达式。以TC为例，临时变量的索引不需要定义。

IR可以根据爱因斯坦表示法，通过未定义变量的出现来计算出实际的表达式。在爱因斯坦表示法中，运算符需要具有结合性和交换性。这个限制保证了归约运算符可以按任何顺序执行，从而可以进一步并行化

让我通过一个简单的例子来展示在TC中使用Einstein表示法：
假设我们有两个矩阵 A 和 B，我们想要计算它们的矩阵乘法并将结果存储在矩阵 C 中。在TC中，我们可以使用Einstein表示法来定义这个矩阵乘法操作：

```
def matmul(A: float[N, K], B: float[K, M]) -> float[N, M]:  
    C(n, m) +=! A(n, k) * B(k, m)
```

编辑于 2024-01-04 17:43 · IP 属地中国香港

[深度学习 \(Deep Learning\)](#) [编译器](#) [深度学习编译器](#)

发布一条带图评论吧




还没有评论，发表第一个评论吧


文章被以下专栏收录

深度学习编译器 [深度学习编译器](#)

推荐阅读

编译器和解释器之间有什么区别
[编译器和解释器之间有什么区别](#)

[推荐几个好用的编译器](#)
下面重点介绍几个好用的编译器。
online gdb这个在线编译器就比较强大了，主要特点有：支持gdb在线调试支持控制台输入(这个特点几

深入理解编译器
[深入理解编译器](#)