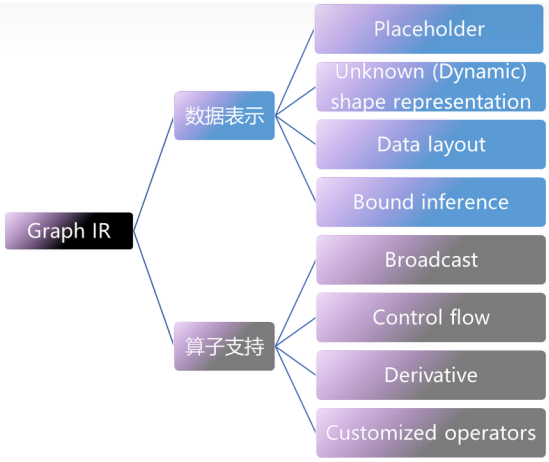
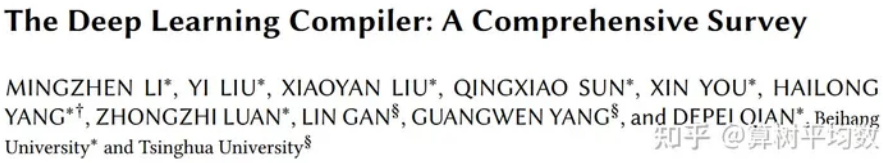


<三>深度学习编译器综述：High-Level IR (2)

算树平均数
昏昏沉沉工程师 (寻职中)
关注他

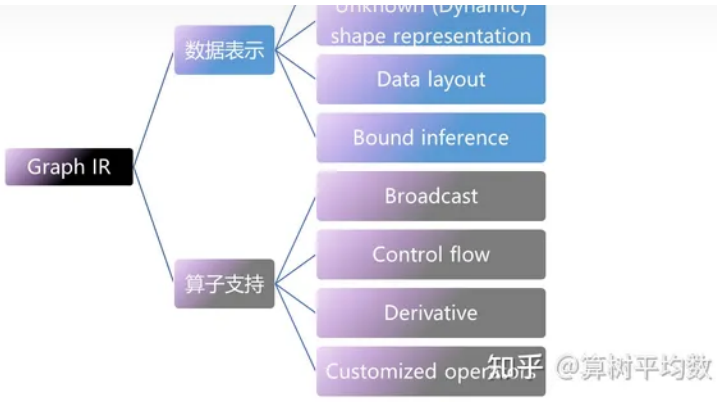
The Deep Learning Compiler: A Comprehensive Survey


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 DEPEI QIAN*, Beihang University* and Tsinghua University§

上次内容介绍了High-Level IR中Graph IR的类别和Tensor 计算方式.

本文将深入Graph IR的实现，了解Graph IR的**Data representation（数据表示，管理）**和**Operators supported（支持的操作，算子）**。



Implementation of Graph IR

Data representation

深度学习编译器中的数据（例如输入、权重和中间数据）通常以tensor表示。

DL编译器可以直接通过内存指针操作tensor，或者通过Placeholder以更灵活的方式表示。

Placeholder:

placeholder广泛用于符号编程（例如 Lisp 、Tensorflow ）。placeholder只是一个具有显式形状信息（例如，每个维度的大小）的变量，并且将在计算的后期阶段用值填充。

它允许程序员描述操作并构建计算图，而无需关心确切的数据元素，这有助于将计算定义与深度学习编译器中的确切执行分开。

此外，程序员可以方便地使用placeholder来改变输入/输出和其他相应中间数据的形状，而无需改变计算定义。

> 解释一下，这两种方式在处理张量数据的表示和传递上有不同的特点。
>
> **内存指针直接表示：**
> 当DL编译器使用内存指针直接表示张量数据时，它会将张量数据的实际值存储在内存中，并使用指针来引用这些内存位置。
> 这种方式效率高，适用于已知形状和数据值的情况，但可能不够灵活，无法处理动态形状或未知数据的情况。
>
> **Placeholder表示：**
> Placeholder是一种更灵活的数据表示方式。在这种方式中，编译器并不直接存储张量的实际值，而是创建一个Placeholder，表示这个张量的数据将在运行时动态地提供。
> 这对于模型的输入、输出以及未知形状的数据非常有用。Placeholder允许在运行时灵活地传入实际的张量数据，使得编译器能够适应不同的输入和情境。

假设我们有一个深度学习模型，涉及输入张量和权重张量。我们可以通过以下两种方式来表示这些张量数据

内存指针直接表示：

```
import numpy as np

# 定义输入张量和权重张量的数据
```

```
# 使用内存指针直接表示数据
input_tensor = input_data.ctypes.data
weight_tensor = weight_data.ctypes.data

# 编译器可以直接使用指针访问数据
```

Placeholder表示：

```
import tvm
from tvm import te
import numpy as np
# 定义占位符
input_tensor = te.placeholder((2, 2), name='input', dtype='float32')
weight_tensor = te.placeholder((2, 2), name='weight', dtype='float32')

# 定义计算操作
output = te.compute((2, 2), lambda i, j: input_tensor[i, j] * weight_tensor[i, j])

# 编译图形中间表示
s = te.create_schedule(output.op)
func = tvm.build(s, [input_tensor, weight_tensor, output], "llvm")

# 在运行时传入实际的数据
ctx = tvm.device('llvm')
input_data = np.array([[1.0, 2.0], [3.0, 4.0]],dtype=np.float32)
weight_data = np.array([[0.5, 0.5], [0.5, 0.5]],dtype=np.float32)
input_tvm = tvm.nd.array(input_data,ctx)
weight_tvm = tvm.nd.array(weight_data,ctx)
output_tvm = tvm.nd.empty((2, 2),ctx)
func(input_tvm, weight_tvm, output_tvm)

# 可以获取输出结果
```

Unknown (Dynamic) shape representation

声明placeholder时通常支持未知的维度大小。

例如，

- TVM 使用 Any 来表示未知维度(例如，Tensor ‹(Any, 3), fp32›)；
- XLA 使用 None 来实现相同的目的(例如，tf .placeholder（“f loat”，[None, 3]))
- nGraph 使用其 PartialShape 类。

未知的形状表示对于支持动态模型是必要的。然而，为了完全支持动态模型，应该放宽约束推理和维度检查。此外，还应该实现额外的机制来保证内存的有效性。

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。

数据布局通常包括维度序列（例如，NCHW 和 NHWC）、tiling, padding, striding等。

TVM 和 Glow 将数据布局表示为算子参数，并需要此类信息进行计算和优化。

> 在 TVM 中，数据布局信息通常作为操作符
> 一个或多个输入张量，每个张量都有自己的

```
import tvm
from tvm import te

# 定义占位符
A = te.placeholder((3, 4), name='A', dtype='float32')

# 定义计算操作，指定数据布局
C = te.compute((3, 4), lambda i, j: A[i, j] * 2, name='C')

# 编译图形中间表示
s = te.create_schedule(C.op)
func = tvm.build(s, [A, C], "llvm")
```

然而，将数据布局信息与运算符而不是张量相结合可以实现某些运算符的直观实现并减少编译开销。

XLA 将数据布局表示为与其后端硬件相关的约束。

Relay 和 MLIR 将在其张量类型系统中添加数据布局信息。

Bound inference:

在深度学习编译器中编译深度学习模型时，边界推断用于确定迭代器的边界。

尽管深度学习编译器中的张量表示可以方便地描述输入和输出，但它给推断迭代器边界带来了特殊的挑战。

绑定推理通常根据计算图和已知占位符以递归或迭代方式执行。

例如，在 TVM 中，迭代器形成有向非循环超图，其中图的每个节点代表一个迭代器，每个超边代表两个或多个迭代器之间的关系（例如，分割、融合或变基）。

一旦根据占位符的形状确定了根迭代器的边界，就可以根据关系递归地推断出其他迭代器

> 假设我们有一个计算操作，计算输入张量 A 的元素的平方和，并将结果保存在输出张量 B 中。

```
import tvm
from tvm import te

# 定义占位符
A = te.placeholder((3, 4), name='A', dtype='float32')
B = te.compute(A.shape, lambda i, j: A[i, j] * A[i, j], name='B')
s = te.create_schedule(B.op)
# 进行 Bound Inference
bounds = tvm.te.schedule.InferBound(s)
# 打印边界信息
print(bounds)

# print info
#{iter_var(j, range(min=0, ext=4)): range(min=0, ext=4), iter_var(i, range(min=0, ext=
```

> 通过这个函数，我们可以获取张量 A 和 B 的维度的最小索引值和范围

Operators supported

深度学习编译器支持的算子负责表示深度学习

- 代数算子（+, ×, exp and topK）
- 神经网络算子（convolution and pooling）
- 张量算子（reshape, resize and copy）
- 广播和归约算子（例如，min and argmin）
- 控制流运算符（conditional and loop）

在这里，我们选择在不同的深度学习编译器中经常使用的三个代表性算子进行说明。另外，我们讨论一些算子自定义的case。

Broadcast

Broadcast可以复制数据并生成具有兼容形状的新数据。如果没有Broadcast算子，输入张量形状会受到更多限制。

例如，对于加法运算符，输入张量应具有相同的形状。一些编译器（例如 XLA 和 Relay）通过提供Broadcast来放宽此类限制。

例如，XLA 允许对矩阵和向量进行逐元素加法，方法是复制矩阵和向量，直到其形状与矩阵匹配。

> 在 Relay 中，广播操作可以通过广播函数 `relay.broadcast_to` 来实现，我们使用 `relay.broadcast_to` 函数将输入张量 `x` 广播为与 `y` 相同的形状，然后使用 `relay.add` 函数进行逐元素相加操作

```
import tvm
from tvm import relay
import numpy as np
# 创建输入变量
x = relay.var("x", shape=(3, 1), dtype="float32")
y = relay.var("y", shape=(3, 4), dtype="float32")

# 进行广播操作
broadcasted_x = relay.broadcast_to(x, shape=(3, 4))
result = relay.add(broadcasted_x, y)

# 创建 Relay 函数
func = relay.Function([x, y], result)

# 编译 Relay 函数
mod = tvm.IRModule.from_expr(func)
target = "llvm"
compiled_func = relay.create_executor(mod = mod)

# 输入数据
input_x = tvm.nd.array(np.array([[1], [2], [3]],dtype=np.float32))
input_y = tvm.nd.array(np.array([[4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]],dtype

# 执行函数
output = compiled_func.evaluate()(input_x, input_y)
print(output)
#[[ 5.  6.  7.  8.]
#[10. 11. 12. 13.]
#[15. 16. 17. 18.]]
```

Control flow

表示复杂且灵活的模型时需要控制流。 RNN 和
的条件执行，这需要控制流。

Relay 注意到任意控制流可以通过递归和模式来实现，这已经通过函数式编程得到了证明，因此，它提供了if运算符和递归函数来实现控制流。

相反，XLA 表示特殊 HLO 运算符（例如 while 和If）的控制流。

> 我们使用 `relay.If` 操作来实现一个简单的条件语句。
> 如果输入值 `x` 小于 10，则执行 `then_branch`，将 `x` 乘以 2；
> 否则，执行 `else_branch`，将 `x` 乘以 3。

```
import tvm
from tvm import relay


'''
x = input()
if x < 10:
        x * = 2
else
        x * = 3

'''

# 创建输入变量
x = relay.var("x", shape=(), dtype="float32")

# 创建条件语句
condition = relay.less(x, relay.const(10, "float32"))
then_branch = relay.multiply(x, relay.const(2, "float32"))
else_branch = relay.multiply(x, relay.const(3, "float32"))
result = relay.If(condition, then_branch, else_branch)

# 创建 Relay 函数
func = relay.Function([x], result)

# 编译 Relay 函数
mod = tvm.IRModule.from_expr(func)
target = "llvm"
compiled_func = relay.create_executor(mod = mod)

# 输入数据
input_data = tvm.nd.array(np.array(5., dtype=np.float32))

# 执行函数
output = compiled_func.evaluate()(input_data)
print(output)

##
# 10.0
```

Derivate

算子Op的导数算子将Op的输出梯度和输入数据作为输入，然后计算Op的梯度。

尽管一些DL编译器（例如TVM和TC）支持自动微分，但当应用链式规则时，它们需要高级IR中所有运算符的导数。

相反，PlaidML 可以自动生成导数算子，甚至是定制算子。

值得注意的是，无法支持导数算子的深度学习编译器无法提供模型训练的能力。

> 以下将使用tvm实现函数求导$f(x) = x^2$

```
from tvm import relay
import numpy as np
import tvm

mod = tvm.IRModule()
x = relay.var('x', shape=())
y = relay.multiply(x, x)
mod['main'] = relay.Function([x], y)
mod = relay.transform.InferType()(mod)
grad_ir = relay.transform.gradient(mod['main'], mode = 'first_order')
b_mod = tvm.IRModule.from_expr(grad_ir)
print(mod['main'])
'''
fn (%x: float32 /* ty=float32 */) -> float32 {
multiply(%x, %x) /* ty=float32 */
} /* ty=fn (float32) -> float32 */
'''
lib =relay.build(b_mod,'llvm')
m = tvm.contrib.graph_executor.GraphModule(lib['default'](tvm.device('llvm',0)))

input_data = tvm.nd.array(np.array(5.,dtype="float32"))
m.set_input('x',input_data)
m.run()
res = m.get_output(0,tvm.nd.empty(input_data.shape)).numpy()
grad = m.get_output(1,tvm.nd.empty(input_data.shape)).numpy()
print(f"result is : {res}, gradient is : {grad}")
'''
result is : 25.0, gradient is : 10.0
'''
```

Customized operators

它允许程序员为特定目的定义其运算符。提供对自定义运算符的支持提高了 DL 编译器的可扩展性。

例如，在Glow中定义新的算子时，程序员需要实现逻辑和节点的封装。此外，如果需要的话，还需要额外的努力，例如降低步骤、操作IR生成和指令生成。

而 TVM 和 TC 除了描述计算实现之外，需要较少的编程工作。具体来说，TVM的用户只需要描述计算和时间表并声明输入/输出张量的形状。而且定制的算子通过hook的方式集成了Python函数，进一步减轻了程序员的负担。

> 如何在TVM自定义自己的算子，详看tvm.hyper.ai/docs/dev/h...

编辑于 2024-01-04 17:45 · IP 属地中国香港

编译器　深度学习（Deep Learning）　深度学习编译器