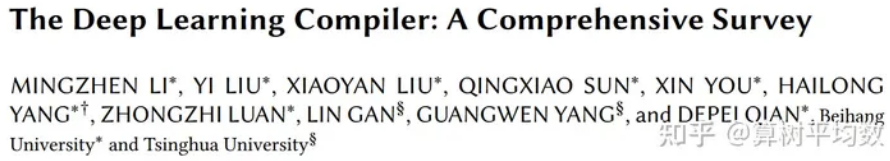
<五> 深度学习编译器综述：Frontend Optimizations

<五> 深度学习编译器综述：Frontend Optimizations

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

The Deep Learning Compiler: A Comprehensive Survey


The Deep Learning Compiler: A Comprehensive Survey

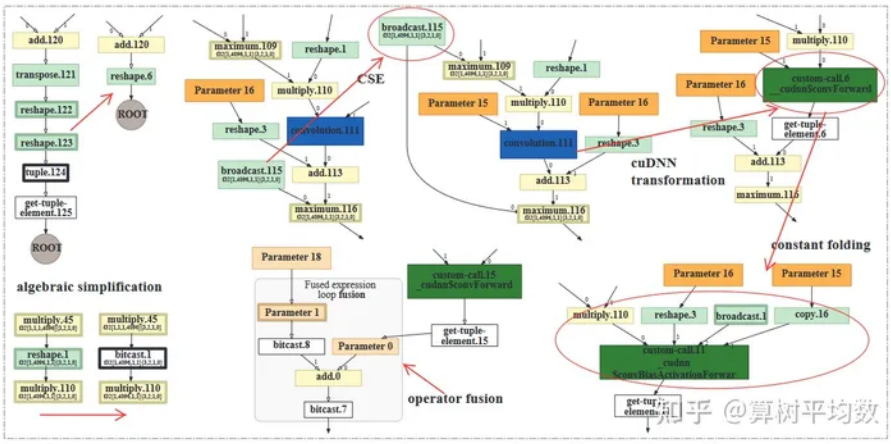
MINGZHEN LI*, YI LIU*, XIAOYAN LIU*, QINGXIAO SUN*, XIN YOU*, HAILONG YANG*†, ZHONGZHI LUAN*, LIN GAN§, GUANGWEN YANG§, and DEPEI QIAN*, Beihang University* and Tsinghua University§

1. Frontend Optimizations

深度学习编译器的**前端优化（Frontend Optimization）**是指在将深度学习模型从高级深度学习框架（如TensorFlow、PyTorch等）转换为可执行的机器码或中间表示（Intermediate Representation，IR）之前，对模型进行一系列的优化操作。

构建计算图后，前端应用图级优化。许多优化更容易在图级别上识别和执行，因为图提供了计算的全局视图。

这些优化操作旨在改善模型的性能、降低计算成本、减少资源占用和提高执行效率。



前端优化通常由passes定义，通过遍历计算图的节点并执行图转换实现。

> 在编译器领域，"pass"（通常称为编译器通行证或优化通行证）是指编译器的一个阶段或一个模块，它对输入程序或代码执行一系列特定的转换、优化或分析操作。
>
> 每个pass通常执行一组相关的任务，以改变程序的形式、提高代码质量、优化性能或进行其他编译任务。
>
> pass的目的是将源代码或中间表示（IR）从一个状态转换到另一个状态，以便后续pass可以继续执行更高级别的优化或代码生成任务。通行证之间的顺序和数量通常取决于编译器的设计和优化策略。
>
> 以下是一些pass可能执行的任务示例：

1. **词法分析和语法分析**：这是编译器的前端阶段，它将源代码解析成语法树或中间表示，以便后续pass能够理解代码的结构。
2. **语义分析**：pass可以执行类型检查、作用域分析、错误检测等任务，以确保源代码的语义正确性。
3. **优化pass**：这些pass执行各种代码优化操作，例如常量折叠、死代码消除、循环展开、操作融合等，以提高程序性能。
4. **中间表示生成**：pass可以将优化后的代码转换为中间表示（IR），这是编译器用来生成目标代码的内部表示形式。
5. **代码生成**：pass将中间表示转换为目标平台的机器代码或汇编代码。
6. **链接**：如果编译器处理多个源文件或库，pass会将它们组合成一个可执行程序或库。

pass之间的顺序和数量可以因编译器的设计而异。

优化pass通常在前端pass后，但在代码生成之前。

编译器开发者可以根据编译器的目标和优化策略来确定pass的组织和执行顺序。

这些pass是编译器内部的模块，它们协同工作以实现源代码到目标代码的转换过程。

在本节中，我们将前端优化分为三类：

1. **Node-level optimizations 节点级别**
2. **Block-level optimizations 块级别**
3. **Dataflow-level optimizations 数据流级别**

2. Node-Level Optimizations

节点级优化包括：

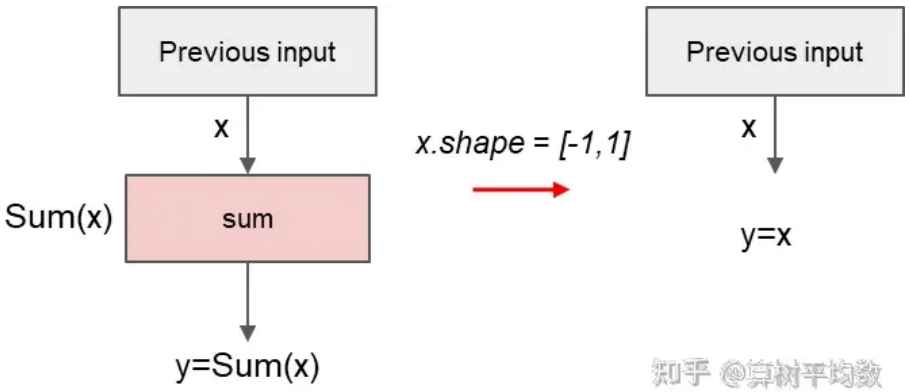
- **节点消除（消除不必要的节点）**
- **节点替换（用其他成本较低的节点替换节点）**

在通用编译器中，Nop Elimination 删除占用少量空间但不指定任何操作的无操作指令。

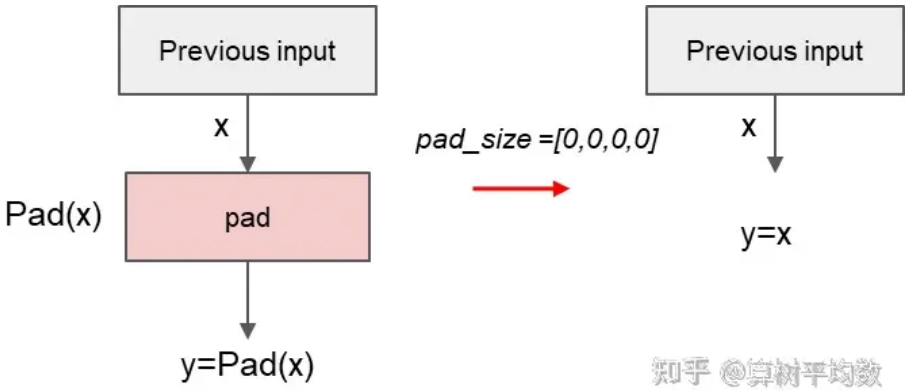
在深度学习编译器中，Nop Elimination 负责消除缺乏足够输入的操作。

例如，

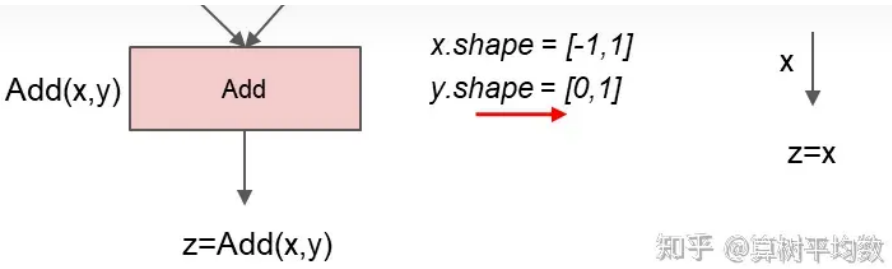
- 可以消除只有一个输入张量的 sum 节点



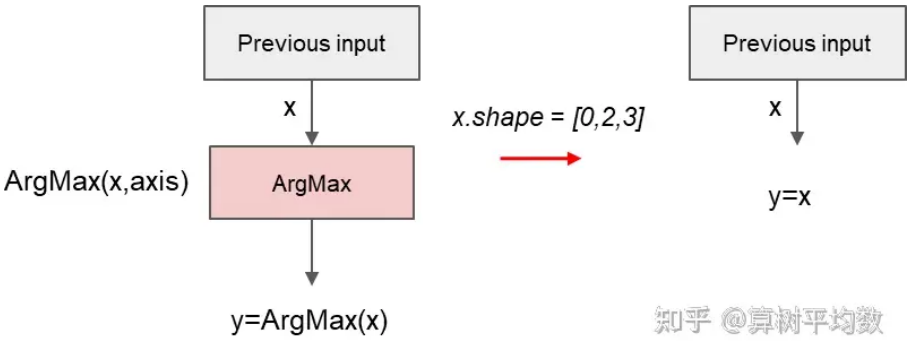
- 可以消除填充宽度为零的 padding 节点。



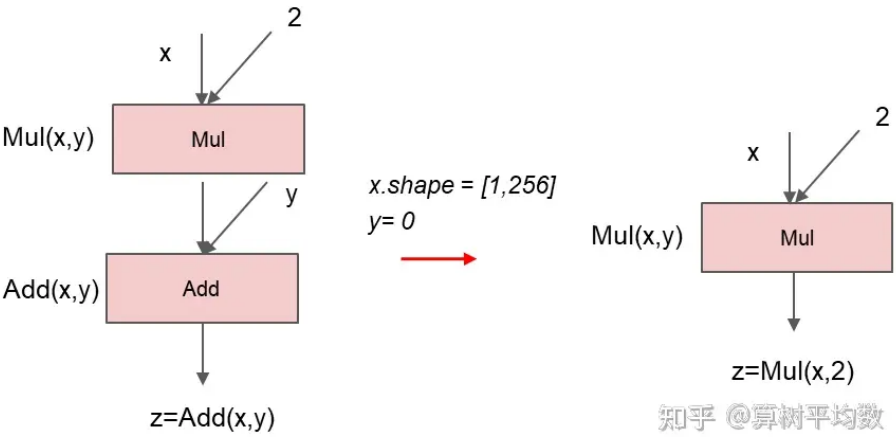
- 假设A是零维张量，B是常量张量，那么A和B的求和运算节点可以替换为已经存在的常量节点B，而不影响正确性。



- 假设C是3维张量，但一维形状为零，如{0,2,3}，因此C没有元素，可以消除argmin/argmax运算节点。



> 我们使用tvm实现一个节点消除的例子：




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


def add_example(shape):
a = relay.var("a", relay.TensorType(shape, "float32"))
y = relay.multiply(y,relay.const(2,"float32"))
y = relay.add(y,relay.const(0,"float32"))
return relay.Function([a,b],y)


shape1 = (1, 256)
```

```
print(mod)
'''
def @main(%a: Tensor[(1, 256), float32]) {
%0 = multiply(%a, 2f);
add(%0, 0f)
}

'''

mod1 = relay.transform.SimplifyExpr()(mod)
print(mod1)
'''
def @main(%a: Tensor[(1, 256), float32] /* ty=Tensor[(1, 256), float32] */) -> Tensor[
multiply(%a, 2f /* ty=float32 */) /* ty=Tensor[(1, 256), float32] */
}
'''
```

可以看到，当add操作的y变量为0时候，add节点被消除了

3. Block-Level Optimizations

Block-Level Optimizations主要包括：

1. **Algebraic Simplification 代数简化**
2. **Operator Fusion 算子融合**
3. **Operator Sinking 算子下沉**

3.1 Algebraic Simplification 代数简化

一个深度学习模型在前端表示一般都是一个DAG图，各种算子可以利用等价计算的方式进行计算量优化，其中代数简化主要包括：

1. **代数识别（Algebraic Identification）**：这一优化技术旨在识别和简化计算图中的代数表达式。通过识别节点之间的代数关系，编译器可以将一系列节点替换为更简单的等效形式，从而减少计算的复杂性。

> **合并相同权重的操作**：如果在计算图中多次使用相同的权重参数进行卷积操作，编译器可以将这些操作合并为一个，以减少计算的复杂性。
> **消除冗余操作**：如果计算图中包含冗余的操作，例如相同的激活函数应用多次，编译器可以消除其中一些操作，以减少计算量。
> **GEMM优化例子**：
> 1.有两个输入矩阵 A 和 B。
> 2.对两个输入矩阵进行转置操作，分别得到 AT 和 BT。
> 3.然后将 AT 和 BT 相乘，得到结果矩阵 C。
>
> 这种方法在数学上是正确的，但它涉及两次矩阵的转置操作，这可能会导致性能下降，尤其是在大规模矩阵计算时，因为矩阵的转置需要额外的计算和内存访问。
>
> 优化的思路是改变操作的顺序，如下：
> 1.有两个输入矩阵 A 和 B。
> 2.将矩阵 B 与矩阵 A 直接相乘，得到结果矩阵 C。
> 3.如果需要，再对矩阵 C 进行转置操作，得到 CT。
> 这种优化的关键在于，通过改变操作顺序，我们只需要在最后一步才进行一次转置操作，而不是在两个输入矩阵上都进行转置操作，从而减少了计算和内存访问的开销

> **用移位操作代替乘法**：将乘法操作替换为位移操作，特别是在权重是2的幂次方时，可以显著提高计算速度。例如，将 `x * 8` 替换为 `x << 3`。
> **用累加操作代替多次加法**：如果计算图中包含多次相同的加法操作，编译器可以将它们替换为累加操作，从而减少加法的次数。

1. **常量折叠（Constant Folding）**：常量折叠是一种将常量表达式替换为其计算结果的优化技术。如果计算图中包含了大量的常量节点，编译器可以在编译时计算这些常量表达式的值，并将其替换为结果，从而减少计算的复杂性和运行时开销。

> **计算常量表达式**：如果计算图中包含常量操作，编译器可以在编译时计算这些常量表达式的值。例如，将 `3 * 4` 替换为 `12`。
> **移除无用的常量**：如果计算图中包含未使用的常量节点，编译器可以将其移除以减少计算图的复杂性。

这些代数简化优化技术考虑了节点序列，并利用不同类型节点之间的可交换性、可结合性和可分配性等代数性质，以简化计算。

3.2 Operator Fusion 算子融合

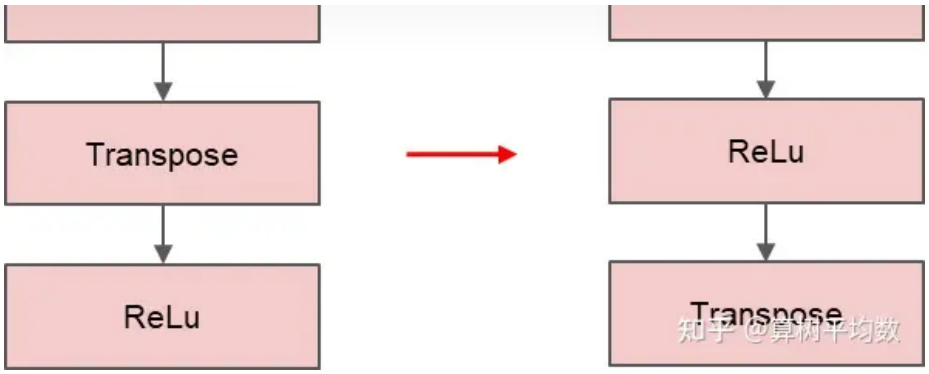
算子融合（Operator Fusion）是一项至关重要的优化技术。它的目标是将多个神经网络操作或算子合并成一个更大的算子，以提高计算效率和减少资源消耗。

1. **更好的计算共享**：通过将多个操作融合成一个，可以减少计算之间的数据传输和中间结果的存储需求。这提高了计算资源的利用率，特别是在GPU等硬件加速器上。
2. **消除中间分配**：运算符融合通常会减少或消除不必要的中间分配和内存操作。这有助于减少内存占用和提高内存带宽的效率。
3. **进一步优化的便利性**：融合后的运算符通常更容易进行进一步的优化。例如，可以更容易地对融合的运算符应用常量传播、代数化简、强度降级等优化技术，以减少计算开销。
4. **减少启动和同步开销**：在某些硬件上，启动和同步运算符的开销可以很显著。通过融合运算符，可以减少启动和同步操作的次数，从而提高计算效率。

> 算子融合通常在深度学习计算图的不同操作之间执行。
> 例如，卷积操作、激活函数操作和池化操作可以融合成一个单一的运算符，称为卷积层（Convolution Layer），这有助于减少计算和数据传输的复杂性。
>
> 以下是一个简单的示例，说明运算符融合如何提高计算效率：
> 考虑以下计算图片段：
> ```
> Input -> Convolution -> ReLU -> Pooling -> Output
> ```
>
> 在算子融合之前，这个计算图中有四个独立的运算符。
> 但是，通过算子融合，可以将这些算子合并为一个算子，如下所示：
>
> ```
> Input -> Conv-Relu-Pool -> Output
> ```
>
> 这个融合后的运算符执行相同的功能，但在计算上更高效，减少了计算和数据传输的开销。
> 总之，深度学习编译器前端中的块级优化的运算符融合是一项关键的优化技术，可以显著提高深度学习模型的执行效率，减少资源消耗，并为后续的优化提供更好的基础。
> 它是实现高性能深度学习推断和训练的重要组成部分。

3.3 Operator Sinking 算子下沉

这种优化将Transpose等操作下沉到BN、ReLU、sigmoid 和Channel Shuffle等操作之下。通过这种优化，许多类似的运算彼此更加接近，为代数简化创造了更多机会。



> 我们使用tvm实现constant floding 和 operator fusion的例子：

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

def example():
shape = (1, 64, 54, 54)
c_data = np.empty(shape).astype("float32")
c = relay.const(c_data)
weight = relay.var("weight", shape=(64, 64, 3, 3))
x = relay.var("x", relay.TensorType((1, 64, 56, 56), "float32"))
conv = relay.nn.conv2d(x, weight)
y = relay.add(c, c)
y = relay.multiply(y, relay.const(2, "float32"))
y = relay.add(conv, y)
z = relay.add(y, c)
z1 = relay.add(y, c)
z2 = relay.add(z, z1)
return relay.Function([x, weight], z2)


f = example()
mod = tvm.IRModule.from_expr(f)
print(mod)
'''
def @main(%x: Tensor[(1, 64, 56, 56), float32], %weight: Tensor[(64, 64, 3, 3), float3
%0 = add(meta[relay.Constant][0], meta[relay.Constant][0]);
%1 = nn.conv2d(%x, %weight, padding=[0, 0, 0, 0]);
%2 = multiply(%0, 2f);
%3 = add(%1, %2);
%4 = add(%3, meta[relay.Constant][0]);
%5 = add(%3, meta[relay.Constant][0]);
add(%4, %5)
}

'''

mod1 = relay.transform.FoldConstant()(mod)
print(mod1)

'''

def @main(%x: Tensor[(1, 64, 56, 56), float32] /* ty=Tensor[(1, 64, 56, 56), float32]
%0 = nn.conv2d(%x, %weight, padding=[0, 0, 0, 0]) /* ty=Tensor[(1, 64, 54, 54), float3
%1 = add(%0, meta[relay.Constant][0] /* ty=Tensor[(1, 64, 54, 54), float32] */) /* ty=
```

```
add(%2, %3) /* ty=Tensor[(1, 64, 54, 54), float32] */
}

'''
mod2 = relay.transform.FuseOps()(mod1)
print(mod2)

'''
def @main(%x: Tensor[(1, 64, 56, 56), float32] /* ty=Tensor[(1, 64, 56, 56), float32]
%4 = fn (%p0: Tensor[(1, 64, 56, 56), float32] /* ty=Tensor[(1, 64, 56, 56), float32]
%0 = nn.conv2d(%p0, %p1, padding=[0, 0, 0, 0]) /* ty=Tensor[(1, 64, 54, 54), float32]
%1 = add(%0, %p2) /* ty=Tensor[(1, 64, 54, 54), float32] */;
%2 = add(%1, %p3) /* ty=Tensor[(1, 64, 54, 54), float32] */;
%3 = add(%1, %p3) /* ty=Tensor[(1, 64, 54, 54), float32] */;
add(%2, %3) /* ty=Tensor[(1, 64, 54, 54), float32] */
} /* ty=fn (Tensor[(1, 64, 56, 56), float32], Tensor[(64, 64, 3, 3), float32], Tensor[
%4(%x, %weight, meta[relay.Constant][0] /* ty=Tensor[(1, 64, 54, 54), float32] */, met
}

'''
```

> 可以看出，经过constant floding 和 operator fusion，最后只剩下一个算子 `%4 = fn`

4.Dataflow-Level Optimizations

数据流级别优化关注的是计算图中不同块之间的数据流和依赖关系。它旨在优化数据的传输和处理流程以提高整个计算图的效率。

Dataflow-Level Optimizations主要包括：

1. **Common sub-expression elimination (CSE) 共同子表达式消除**
2. **Dead code elimination (DCE) 死代码消除**
3. **Static memory planning 静态内存规划**
4. **Layout Transformation 布局变换**

4.1 Common sub-expression elimination (CSE) 共同子表达式消除

共同子表达式消除是一种优化技术，它旨在消除计算图中重复计算相同表达式的情况。

当多个操作需要计算相同的中间结果时，只需计算一次，并在需要时重复使用这个结果，而不必重复计算

> 考虑以下计算图片段：

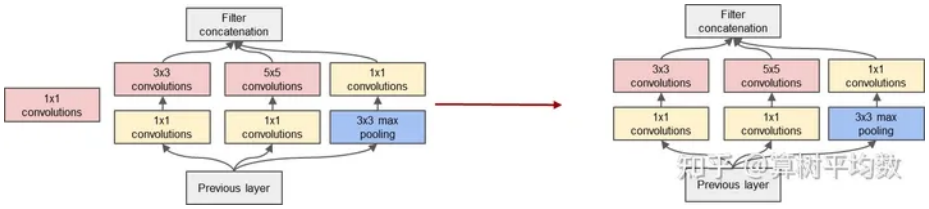
```
A = B + C
D = B + C
```

> 这里的表达式 `B + C` 在两个地方都计算了两次。通过共同子表达式消除，可以将其计算一次，然后重用结果：

```
A = B + C
D = B + C
```

4.2 Dead code elimination (DCE) 死代码消除

这些节点通常是由于模型重构或其他原因而变得无效的。



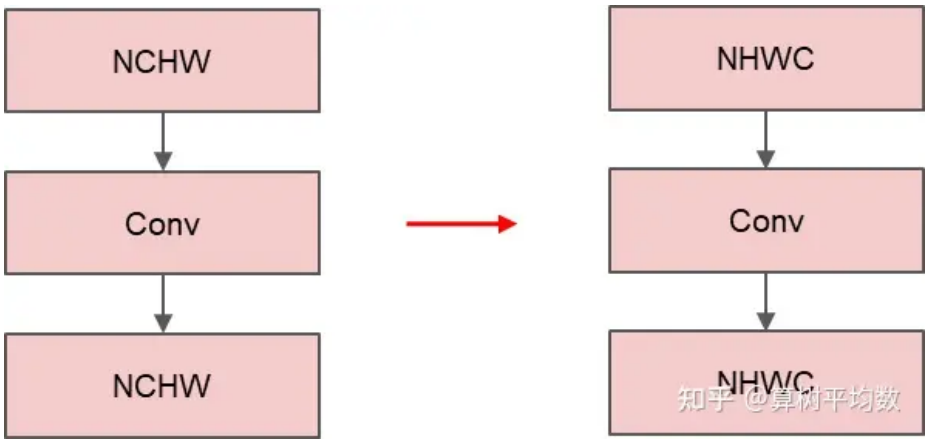
4.3 Static memory planning 静态内存规划

静态内存规划是指在模型编译期间为模型的中间结果分配内存空间，以减少在运行时的内存分配和释放开销。这有助于提高执行效率。

1. **In-Place Memory Sharing（原地内存共享）**：
2. **解释**：不同的操作或层次可以共享相同的内存空间，以减少内存占用。这意味着在模型的计算过程中，相同的内存区域可以用于不同操作的输入和输出，而不必每次都为它们分配新的内存。
3. **优点**：原地内存共享可以减少内存占用和内存分配开销，提高内存使用效率。它特别适用于内存有限的设备，如边缘设备。
4. **示例**：在卷积神经网络中，输入特征图和输出特征图的内存可以被多个卷积操作共享，因为它们的尺寸和数据类型相同。这可以通过指定输入和输出张量的内存布局来实现。
5. **Standard Memory Sharing（标准内存共享）**：
6. **解释**：不同操作或层次可以共享相同的内存空间，但在计算过程中需要确保不会互相干扰。这通常需要使用额外的同步和管理机制来确保数据的正确性。
7. **优点**：标准内存共享可以减少内存占用和内存分配开销，但与原地内存共享不同，它更注重数据的安全性和正确性。
8. **示例**：在多线程或多设备环境中，标准内存共享可能会用于确保多个操作之间的数据共享和同步。例如，在分布式深度学习训练中，不同的计算节点可以共享模型参数，但需要使用同步机制确保参数的一致性。

4.4 Layout Transformation 布局变换

布局转换是指在计算图中的不同操作之间进行数据布局的变换，以适应不同硬件或加速器的需求。例如，将数据从行优先布局转换为列优先布局以提高内存访问效率



> 我们使用tvm实现CSE 和 DSE的例子：

```
from tvm import te
import tvm.relay as relay

def add_example(shape):
a = relay.var("a", relay.TensorType(shape, "float32"))
b = relay.add(a, relay.const(1,"float32"))
d = relay.add(a, relay.const(100,"float32"))
y1 = relay.multiply(b,relay.const(2,"float32"))
c = relay.add(a, relay.const(1,"float32"))
y2 = relay.multiply(c,relay.const(3,"float32"))
y = relay.add(y1,y2)
return relay.Function([a],y)


shape1 = (1, 256)
f = add_example(shape1)
mod = tvm.IRModule.from_expr(f)
print(mod)
'''

def @main(%a: Tensor[(1, 256), float32]) {
%0 = add(%a, 1f);
%1 = add(%a, 1f);
%2 = multiply(%0, 2f);
%3 = multiply(%1, 3f);
add(%2, %3)
}

'''

mod1 = relay.transform.EliminateCommonSubexpr()(mod)
print(mod1)
'''
def @main(%a: Tensor[(1, 256), float32] /* ty=Tensor[(1, 256), float32] */) -> Tensor[
%0 = add(%a, 1f /* ty=float32 */) /* ty=Tensor[(1, 256), float32] */;
%1 = multiply(%0, 2f /* ty=float32 */) /* ty=Tensor[(1, 256), float32] */;
%2 = multiply(%0, 3f /* ty=float32 */) /* ty=Tensor[(1, 256), float32] */;
add(%1, %2) /* ty=Tensor[(1, 256), float32] */
}

'''
```

> 可以看出：
> `d = relay.add(a, relay.const(100,"float32"))` 作为一个死代码被消除了，其中 `b` , `c` 都是 `a + 1` 的操作进行CSE优化

5. Discussion

前端是深度学习编译器中最重要的组件之一，负责从深度学习模型到高级IR（例如计算图）的转换以及基于高级IR的与硬件无关的优化。

尽管前端的实现可能在不同 DL 编译器的高级 IR 的数据表示和运算符定义方面有所不同，但与硬件无关的优化集中在三个级别：节点级、块级和数据流级。

每个级别的优化方法都利用了 DL 特定的以及通用的编译优化技术，这减少了计算冗余并提高了 DL 模型在计算图级别的性能。

编辑于 2024-01-04 17:44 · IP 属地中国香港