【我与TVM二三事 前篇（2）】relay optimize准备知识

本来是准备写relay的优化过程，但是追读代码的过程中发现有一些基础的结构很重要，作用不清楚的话会很容易被优化过程的代码劝退，所以先进行一些梳理

# Part1 DependencyGraph

构建过程：

1 DependencyGraph::Create

2 DependencyGraph::Creator::Create

3 MixedModeVisitor::VisitExpr

4 DependencyGraph::Creator::VisitLeaf(添加) && DependencyGraph::Creator::Depend(链接）

这个结构在ToANormalForm这个Pass中用到了，很多打印函数也都有使用，有必要梳理一下结构。可以采用如下代码进行测试

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/test_dependency_graph.py>

这段测试代码会在print调用PrettyPrint的时候构建dependency\_graph，为了方便查看结构，我写了一个可视化的function，需要和下面的代码一起修改

@ tvm/src/printer/relay\_text\_printer.cc

Doc RelayTextPrinter::PrintFinal(const ObjectRef& node) {

if (node.defined() && node->IsInstance<BaseFuncNode>() &&

!node->IsInstance<relay::FunctionNode>()) {

…

} else if (node.as<ExprNode>()) {

Expr expr = Downcast<Expr>(node);

dg\_ = DependencyGraph::Create(&arena\_, expr);

//add visualize after add function by Archermmt

dg\_.visualize(“dependency.prototxt”)；

}

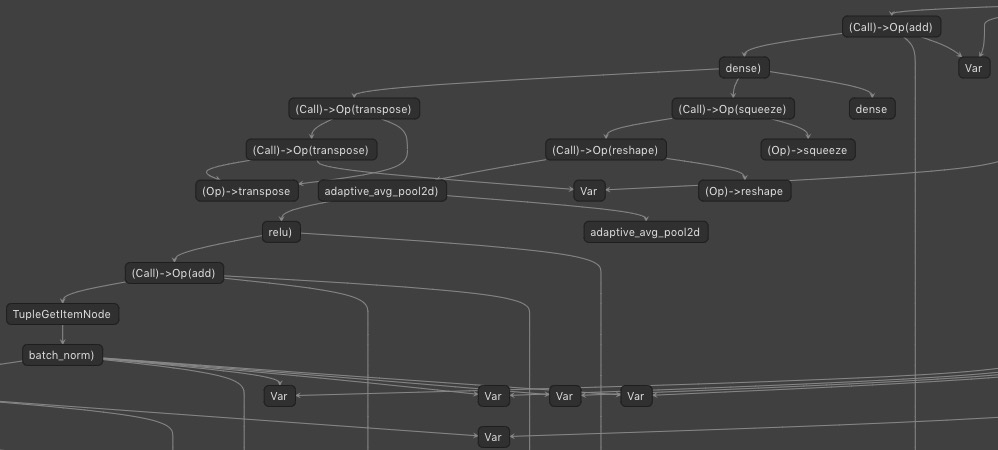
…

}

@ tvm/src/relay/analysis/dependency\_graph.h 中添加

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualize.cpp#L2>

运行之后结果如下



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/dependcy_graph.prototxt>

DependencyGraph的结构就是DAG图，每个节点是典型的图节点形式（parents，children，信息三元素），其中添加和链接的核心函数为

@ tvm/src/relay/analysis/dependency\_graph.cc

void Depend(DependencyGraph::Node\* parent, DependencyGraph::Node\* child) {

auto\* parent\_link = arena\_->make<LinkNode<DependencyGraph::Node\*> >();

parent\_link->value = parent;

child->parents.Push(parent\_link);

auto\* child\_link = arena\_->make<LinkNode<DependencyGraph::Node\*> >();

child\_link->value = child;

parent->children.Push(child\_link);

}

…

void VisitLeaf(const Expr& e) override {

if (visited\_.count(e) == 0) {

if (graph\_.expr\_node.count(e) == 0) {

graph\_.expr\_node[e] = NewNode(false);

}

visited\_.insert(e);

MixedModeVisitor::VisitLeaf(e);

graph\_.post\_dfs\_order.push\_back(graph\_.expr\_node[e]);

}

}

由VisitLeaf进行节点添加以及拓展，Depend进行链接。注意此处的VisitLeaf不是一般树结构中的叶节点，而是children被完全拓展之后的节点。

构建过程对Node的new\_scope也进行了设定。New\_scope标明此节点是否开启了一段新scope，可以查看设定new\_scope=true的Node类型，有IfNode，FunctionNode，LetNode和MatchNode，都是涉及到分支或者block的初始节点，相当于C++中出现{}的时候标定一下。

# Part2 IndexedForwardGraph

构建过程：

1 IndexedForwardGraph::Create

2 IndexedForwardGraph::Creator::Prepare

3 IndexedForwardGraph::Creator::Update(添加根节点)

4 ExprVisitor::VisitExpr

5 IndexedForwardGraph::Creator::AddNode && IndexedForwardGraph::Creator::Update(添加图节点)

FuseOps的时候用到，这个结构涉及到一些有用的算法，并且可能在写其他Pass的时候用到，所以也在此梳理一下。这个结构目前只有在使用FuseOps这个Pass的时候会被调用，而FuseOps在实验过程中发现不支持batch\_norm所以采用进行SimplifyInference之后的IR进行调试，采用如下代码跟读

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/test_index_graph.py>

IndexedForwardGraph不会被独立使用，而是在fuse\_ops这个Pass中使用到。为了方便查看结构，同样添加了一个简单的可视化方法，需要和以下代码一起修改

@ tvm/src/relay/transforms/fuse\_ops.cc

Expr Transform(const Expr& body, int fuse\_opt\_level, size\_t max\_fuse\_depth) {

// setup the group map.

auto graph = IndexedForwardGraph::Create(&arena\_, body);

// visualize the indexed forward graph, added by Archermmt

graph.visualize("indexed\_forward\_graph.prototxt");

…

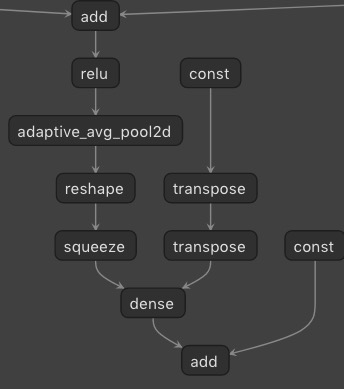
}

@ tvm/src/relay/transforms/fuse\_ops.cc 中添加

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualize.cpp#L98>

可视化之后结构可以和Relay IR的拓扑图比较，会发现大体结构相同，只是IndexedForwardGraph多了一些边的信息（由于没有弄明白Netron边上信息怎么添加，把pattern添加到了点上）。Pattern对应的就是一个出边的类型，后续过程的GraphPartitioner会按照这个信息进行分组。

Relay IR



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/simplify_inference.prototxt>

对应的IndexedForwardGraph



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/indexed_forward_graph.prototxt>

# Part3 GraphPartitioner

构建过程：

1 GraphPartitioner::Partition

2 GraphPartitioner::InitGroups(每个节点初始化为一个Group)

3 DominatorTree::PostDom(构建依赖树作为辅助结构)

4 GraphPartitioner::RunFuse(使用LCA进行依赖查找，对节点分组)

在tvm relay优化过程中，GraphPartitioner和IndexedForwardGraph共同使用对计算图算子进行分组，以达到减少访存次数的效果。此结构不会单独使用，会在调用FuseOps这个Pass的时候被用到

@ tvm/src/relay/transforms/fuse\_ops.cc

std::vector<GraphPartitioner::Group\*> GraphPartitioner::Partition(

const IndexedForwardGraph& graph) {

this->InitGroups(graph);

if (opt\_level\_ == 0) return std::move(groups\_);

// get post dominator tree

auto post\_dom\_tree = DominatorTree::PostDom(arena\_, graph);

// run fusion algorithm.

for (int phase = 0; phase < 3; ++phase) {

this->RunFuse(graph, post\_dom\_tree, phase);

}

return std::move(groups\_);

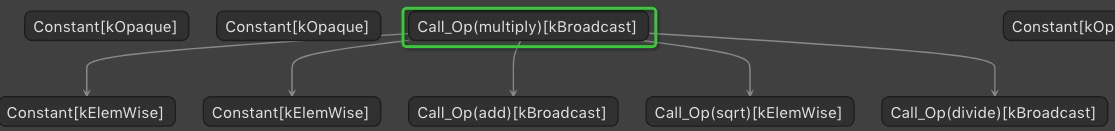
}

InitGroups过程比较简单，就是将IndexedForwardGraph中的每个节点包裹成Group，其中pattern为kOutEWiseFusable的节点设定anchor\_ref。为了可视化，添加如下代码

@ tvm/src/relay/transforms/fuse\_ops.cc 中添加

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualize.cpp#L223>

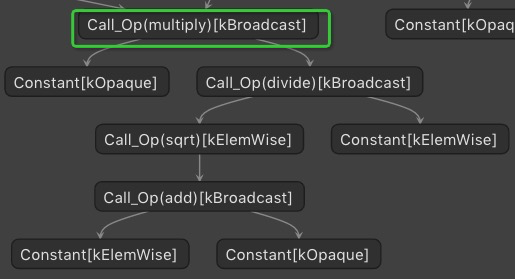
分组之后的部分结构



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/graph_paritioner.prototxt>

对应原始结构



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/simplify_inference.prototxt>

以及对应打印出来的代码部分

fn (%input0: Tensor[(1, 3, 224, 224), float32]) -> Tensor[(1, 1000), float32] {

%149 = fn (%p021: Tensor[(64), float32], %p118: Tensor[(64), float32], Primitive=1) -> Tensor[(64), float32] {

%146 = add(%p021, 1e-05f /\* ty=float32 \*/) /\* ty=Tensor[(64), float32] \*/;

%147 = sqrt(%146) /\* ty=Tensor[(64), float32] \*/;

%148 = divide(1f /\* ty=float32 \*/, %147) /\* ty=Tensor[(64), float32] \*/;

multiply(%148, %p118) /\* ty=Tensor[(64), float32] \*/

};

…

%316(%314, %315, meta[relay.Constant][101] /\* ty=Tensor[(1000), float32] \*/) /\* ty=Tensor[(1, 1000), float32] \*/

}

结果的结构不难理解，就是对原来的计算图进行了分块，计算过程并没有改变。下一节会梳理一下DominatorTree::Postdom这个结构，核心方法RunFuse中没有涉及到新的结构，会在下一节relay optimize的时候进行解读。

## Part3.1 DominatorTree

构建过程：

1 DominatorTree::PostDom

2 DominatorTree::GetNode(得到LCA节点)

3 DominatorTree::LeastCommonAncestor(搜索LCA) && DominatorTree::CombinePattern(决定节点pattern)

对IndexedForwardGraph进行遍历，对每个节点定义两个参数：1.depth，取决于LCA算法找到的parent。2.设定pattern，等于节点入边中最大的pattern

其中depth相关的LCA算法实现为：

@ tvm/src/relay/transforms/fuse\_ops.cc

static Node\* LeastCommonAncestor(Node\* lhs, Node\* rhs, OpPatternKind\* edge\_pattern) {

while (lhs != rhs) {

if (lhs == nullptr) return nullptr;

if (rhs == nullptr) return nullptr;

if (lhs->depth < rhs->depth) {

edge\_pattern[0] = CombinePattern(edge\_pattern[0], rhs->pattern);

rhs = rhs->parent;

} else if (rhs->depth < lhs->depth) {

edge\_pattern[0] = CombinePattern(edge\_pattern[0], lhs->pattern);

lhs = lhs->parent;

} else {

edge\_pattern[0] = CombinePattern(edge\_pattern[0], lhs->pattern);

edge\_pattern[0] = CombinePattern(edge\_pattern[0], rhs->pattern);

lhs = lhs->parent;

rhs = rhs->parent;

}

}

return lhs;

}

…

static OpPatternKind CombinePattern(OpPatternKind lhs, OpPatternKind rhs) {

if (lhs > rhs) return lhs;

return rhs;

}

核心逻辑并不复杂，就是所有的parents进行depth比较，向上遍历直到寻找到同一个祖先节点。LCA过程中还涉及到确定pattern，实现就是比较pattern的大小，返回大值。通过CombinePattern可以看出OpPatternKind的枚举是有优先级的设计的，值的大小决定fuse的难易度和一些规则。

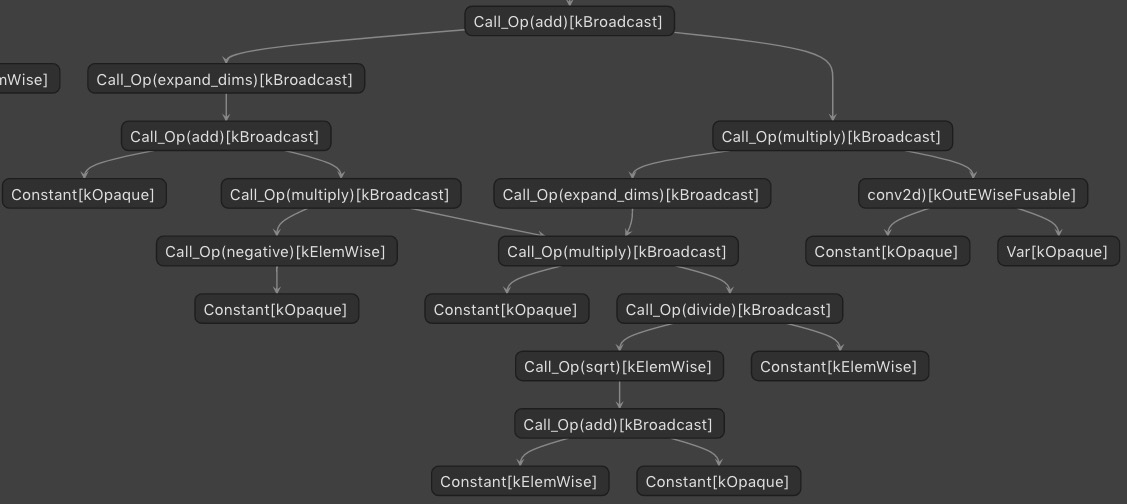
同样为了可视化方便添加部分代码：

@ tvm/src/relay/transforms/fuse\_ops.cc 中添加

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualize.cpp#L162>

对比PostDomTree结构和IndexedForwardGraph结构如下

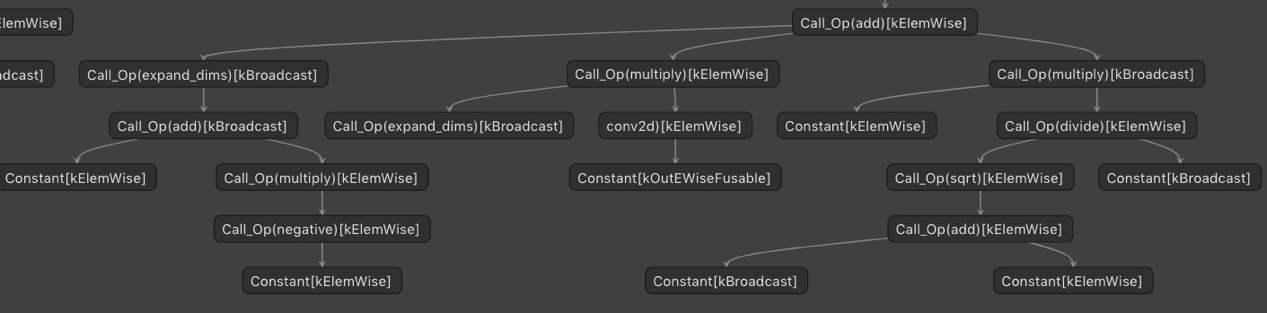
IndexedForwardGraph：



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/indexed_forward_graph.prototxt>

PostDomTree



Netron查看完整信息

<https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualizes/post_dom_tree.prototxt>

节点的parent是此节点的最近邻祖先(LCA)，pattern是所有此节点到LCA的边中最大的pattern，大概理解就是数据从LCA到此节点过程中融合难度最大的种类。

# Part4 FTVMAlterOpLayout注册机制

FTVMAlterOpLayout可以注册为Op的属性，这个属性会在AlterOpLayout这个Pass调用的时候读取并用来做算子变形。这个属性的注册涉及到了TVM里面一种python和C++交互的注册机制，比较绕。这一节用llvm下conv2d的此属性注册过程为例梳理一下。

注册过程可以拆成两个部分：准备信息和注册属性

准备过程（import tvm时调用）：

1 tvm.target.generic\_func(conv2d\_alter\_layout) (将conv2d\_alter\_layout变为函数holder)

2 conv2d\_alter\_layout.register(对不同target添加处理函数)

注册过程（import tvm时调用）：

1 reg.register\_alter\_op\_layout

2 tvm.ir.register\_op\_attr

3 \_ffi\_api.RegisterOpAttr(注册到Op的AttrMap中)

## Part4.1 准备信息

首先对需要注册到AttrMap中的函数进行信息收集

@ tvm/python/tvm/topi/nn/conv2d.py

@tvm.target.generic\_func

def conv2d\_alter\_layout(attrs, inputs, tinfos, out\_type):

…

-> @ tvm/python/tvm/target/generic\_func.py

def generic\_func(fdefault):

…

def register(key, func=None, override=False):

…

def \_do\_reg(myf):

key\_list = [key] if isinstance(key, str) else key

for k in key\_list:

…

dispatch\_dict[k] = myf

return myf

…

def dispatch\_func(func, \*args, \*\*kwargs):

…

for k in target.keys:

if k in dispatch\_dict:

return dispatch\_dict[k](\*args, \*\*kwargs)

return func(\*args, \*\*kwargs)

fdecorate = decorate(fdefault, dispatch\_func)

fdecorate.register = register

fdecorate.fdefault = fdefault

fdecorate.dispatch\_dict = dispatch\_dict

return fdecorate

generic\_func这个包装器把conv2d\_alter\_layout从函数变成了一个对象，对象的register方法用来填充dispatch\_dict这个字典，而真正执行的时候会调用dispatch\_func从dispatch\_dict取出适配target的函数。使用register填充的过程可以参考

@ tvm/python/tvm/topi/x86/conv2d\_alter\_op.py

@conv2d\_alter\_layout.register("cpu")

def \_alter\_conv2d\_layout(attrs, inputs, tinfos, out\_type):

…

在各个模块初始化之后，conv2d\_alter\_layout这个对象的dispatch\_dict就包含了针对不同target的function。打印如下

conv2d\_alter\_layout dispatch\_dict ： {

'cpu': <function \_alter\_conv2d\_layout at 0x1508d8840>,

'cuda': <function \_alter\_conv2d\_layout at 0x1509e0268>,

'gpu': <function \_alter\_conv2d\_layout at 0x1509e0268>,

'arm\_cpu': <function \_alter\_conv2d\_layout at 0x150a2b8c8>,

'mali': <function \_alter\_conv2d\_layout at 0x150a442f0>,

'bifrost': <function \_alter\_conv2d\_layout at 0x150a4ed90>,

'intel\_graphics': <function \_alter\_conv2d\_layout at 0x150a58d08>}

接下来需要把这些函数注册到C++的AttrRegistry里面，供Op::GetAttrMap<T>调用。

## Part4.2 注册属性

注册行为的触发在：@ tvm/python/tvm/relay/op/nn/\_nn.py

@reg.register\_alter\_op\_layout("nn.conv2d")

def alter\_op\_layout\_conv2d(attrs, inputs, tinfos, out\_type):

"""Alternate the layout of conv2d"""

return topi.nn.conv2d\_alter\_layout(attrs, inputs, tinfos, out\_type)

-> @ tvm/python/tvm/relay/op/op.py

def register\_alter\_op\_layout(op\_name, alter\_layout=None, level=10):

…

return tvm.ir.register\_op\_attr(op\_name, "FTVMAlterOpLayout", alter\_layout, level)

-> @ tvm/python/tvm/ir/op.py

def register\_op\_attr(op\_name, attr\_key, value=None, level=10):

…

def \_register(v):

\_ffi\_api.RegisterOpAttr(op\_name, attr\_key, v, level)

return v

return \_register(value) if value is not None else \_register

至此调用RegisterOpAttr完成了注册。

# 总结

分析了四种结构，作用如下：

DependencyGraph (ToANormalForm用到)：构建计算图，结构等同于常用模型编译器里面的DAG结构，但信息量要大得多。

IndexedForwardGraph (FuseOps用到)：类似DependencyGraph，只包含计算节点，且顺序和DependencyGraph相反。

GraphPartitioner (FuseOps用到)：辅助结构，用于标定节点和“组长”（一个分组里面的root节点，姑且这么称呼比较好记）的关系

DominatorTree (FuseOps用到)：服务结构，用于表明节点和其LCA的链接关系。

FTVMAlterOpLayout的注册 (AlterOpLayout用到) ：主要涉及到了TVM里面python和C++交互的机制。tvm.generic类函数包裹的函数变成函数信息载体，并通过register收集不同target的处理函数，收集完成后通过tvm.ir.register\_op\_attr绑定到Op上。类似机制的注册在TVM中被广泛使用