【我与TVM二三事 前篇（3）】relay optimize

此文会跟读tvm.IRModule构建完成到优化之后的过程，梳理相关知识点。

Relay.build会完成IRModule。跟读使用以下测试代码：

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

#optimize the mod之前的部分已经解析过，参照relay\_parse部分解读。本次讲解会追踪到optimize过程结束，也就是常规定义的编译器中端过程。

调用流程：

1 tvm.target.Target(创建Target，用于codegen过程，此篇不涉及)

1.1 Target::Target -> TargetInternal::FromString -> TargetInternal::FromRawString

1.2 GetTargetKind(从名字得到Kind模版)

1.3 TargetInternal::ParseType(解析添加额外参数)

1.4 TargetInternal::FromConfig(从配置得到target)

2 tvm.transform.PassContext(构建优化用的Context，存储信息)

2.1 PassContext::Create

2.2 PassConfigManager::Global()->Legalize(基础检查)

2.3 PassContext::Internal::EnterScope

2.4 PassContext::EnterWithScope(当前ctx入栈)

3 relay.Optimize(执行优化)

3.1 autotvm.tophub.context(用于schedule查找最优配置)

3.1.1 ApplyHistoryBest(构建一个Context种类)

3.1.2 ApplyHistoryBest.load(从文件加载历史配置)

3.2 bld\_mod.optimize(执行优化)

3.2.0 BindParamsByName(绑定参数)

3.2.1 SimplifyInference(优化normalize系列操作)

3.2.2 ToANormalForm(使用赋值格式修改IR结构)

3.2.3 FoldScaleAxis(折叠Scale类型算子)

3.2.4 FoldConstant(常量折叠)

3.2.5 AlterOpLayout(转换算子layout，更换strategy)

3.2.6 FuseOps(算子聚类)

# Step 1 创建Target

@demo.py

target = tvm.target.Target("llvm", host="llvm")

通过调用链

Target::Target ->

TargetInternal::FromString ->

TargetInternal::FromRawString

构造Target，Target是用于记录device信息的载体

@ tvm/src/target/target.cc

ObjectPtr<Object> TargetInternal::FromRawString(const String& target\_str) {

…

std::unordered\_map<String, ObjectRef> config = {{"kind", String(name)}};

TargetKind kind = GetTargetKind(name);

for (size\_t iter = 0, end = options.size(); iter < end;) {

std::string key, value;

…

try {

…

config[key] = TargetInternal::ParseType(value, TargetInternal::FindTypeInfo(kind, key));

} catch (const Error& e) {

throw Error(": Error when parsing target[\"" + key + "\"]" + e.what());

}

}

return TargetInternal::FromConfig(config);

}

简单概括此过程就是GetTargetKind并填充target的属性，GetTargetKind调用TargetKinf::Get并最终调用注册过的target，注册过程使用了AttrRegistry:

using TargetKindRegistry = AttrRegistry<TargetKindRegEntry, TargetKind>;

此类行详细说明可以查看[深入理解TVM：RELAY\_REGISTER\_OP](https://zhuanlan.zhihu.com/p/369433448)

获得注册的target以及注册target代码如下

@ tvm/src/target/target\_kind.cc

Optional<TargetKind> TargetKind::Get(const String& target\_kind\_name) {

const TargetKindRegEntry\* reg = TargetKindRegistry::Global()->Get(target\_kind\_name);

…

return reg->kind\_;

}

…

TVM\_REGISTER\_TARGET\_KIND("llvm", kDLCPU)

.add\_attr\_option<Array<String>>("mattr")

.add\_attr\_option<String>("mcpu")

…

.set\_default\_keys({"cpu"});

查看最后的target，可以看到其中有一些额外的设定。

target : llvm -keys=cpu -link-params=0(<class 'tvm.target.target.Target'>)

# Step 2 构建PassContext

@demo.py

with tvm.transform.PassContext(opt\_level=3)

PassContext作用类似优化过程的上下文环境，详细的介绍可以参照这里：

[TVM代码走读（三） 图优化1--初识PASS](https://zhuanlan.zhihu.com/p/149988448)

初始化阶段会调用PassConfigManager::Legalize进行PassContext的配置

@ tvm/src/transform.cc

void Legalize(Map<String, ObjectRef>\* config) {

…

for (auto kv : \*config) {

…

if (kv.second->IsInstance<Map<String, ObjectRef>::ContainerType>()) {

ObjectRef converted =

reflection->CreateObject(info.type\_key, Downcast<Map<String, ObjectRef>>(kv.second));

update.emplace\_back(kv.first, converted);

} …

}

for (auto&& kv : update) {

config->Set(kv.first, kv.second);

}

}

主要过程就是检查config中的key和注册的type（注册config通过TVM\_REGISTER\_PASS\_CONFIG\_OPTION(name,type);）类型是否一致。

Enter过程存储当前的PassContext到栈顶,Exit则是出栈

@ tvm/src/transform.cc

void PassContext::EnterWithScope() {

PassContextThreadLocalEntry\* entry = RelayPassContextThreadLocalStore::Get();

entry->context\_stack.push(\*this);

}

…

void PassContext::ExitWithScope() {

PassContextThreadLocalEntry\* entry = RelayPassContextThreadLocalStore::Get();

…

entry->context\_stack.pop();

}

查看\_\_enter\_\_和\_\_exit\_\_对应的最终调用，可以看到核心作用是在栈顶记录了PassContext，以便在当前Scope下Pass使用栈顶的PassContext。

# Step 3 relay.optimize

一般解析得到relay IRModule之后会直接调用relay.build完成 optimize，lower和codegen过程，由于后两部分内容较多，本篇先使用relay.optimize进行跟读。

## Step 3.1 创建tophub\_context

@ tvm/python/tvm/autotvm/tophub.py

def context(target, extra\_files=None):

…

best\_context = ApplyHistoryBest([])

…

for tgt in targets:

…

for name in possible\_names:

name = \_alias(name)

if name in all\_packages:

if not check\_backend(tophub\_location, name):

continue

filename = "%s\_%s.log" % (name, PACKAGE\_VERSION[name])

best\_context.load(os.path.join(AUTOTVM\_TOPHUB\_ROOT\_PATH, filename))

break # only load one file to avoid some fallback template mismatch problem

…

return best\_context

核心部分功能就是使用ApplyHistoryBest对象对存储的配置进行加载，存储的配置在log文件中，内容如下：

# This is the pre-tuned parameters for x86 cpu backends

# TVM downloaded this during compilation

{"input": ["llvm -mcpu=skylake-avx512", "conv2d\_NCHWc.x86", [["TENSOR", [1, 3, 224, 224], "float32"], ["TENSOR", [64, 3, 7, 7], "float32"], [2, 2], [3, 3, 3, 3], [1, 1], "NCHW", "NCHW", "float32"], {}], "config": {"index": 193, "code\_hash": null, "entity": [["tile\_ic", "sp", [1, 3]], ["tile\_oc", "sp", [2, 32]], ["tile\_ow", "sp", [14, 8]], ["unroll\_kw", "ot", false]]}, "result": [[9.593498664366054e-05], 0, 1.757516860961914, 1539972731.352305], "version": 0.2, "tvm\_version": "0.7.dev0"}

{"input": ["llvm -mcpu=skylake-avx512", "conv2d\_NCHWc.x86", [["TENSOR", [1, 64, 56, 56], "float32"], ["TENSOR", [64, 64, 3, 3], "float32"], [1, 1], [1, 1, 1, 1], [1, 1], "NCHW", "NCHW", "float32"], {}], "config": {"index": 188, "code\_hash": null, "entity": [["tile\_ic", "sp", [1, 64]], ["tile\_oc", "sp", [2, 32]], ["tile\_ow", "sp", [8, 7]], ["unroll\_kw", "ot", true]]}, "result": [[0.00010744781060336864], 0, 1.8370168209075928, 1539973676.429802], "version": 0.2, "tvm\_version": "0.7.dev0"}

…

每一行记录了对应一种schedule的历史最优配置（schedule在后续文章中会介绍，可以简单理解为一种生成计算实现时候的参数配置）。ApplyHistoryBest.load执行之后会更新self.best\_by\_targetkey和self.best\_by\_model，以供lower过程使用。本节不会使用到配置，后续章节在进行深入研究。

## Step 3.2 relay优化

@ tvm/python/tvm/relay/build\_module.py

def optimize(mod, target=None, params=None):

…

with tophub\_context:

bld\_mod = BuildModule()

mod, params = bld\_mod.optimize(mod, target, params)

TVM中tvm::runtime::ModuleNode派生的类基本都用做在python中当作class的主体使用，并使用GetFunction和python中的直接调用产生联系，所以python的BuildModule中self.\_optimize = self.mod["optimize"]最终调用为

@ tvm/src/relay/backend/build\_module.cc

IRModule Optimize(IRModule relay\_module, const TargetsMap& targets,

const std::unordered\_map<std::string, runtime::NDArray>& params) {

…

Array<Pass> pass\_seqs;

pass\_seqs.push\_back(transform::RemoveUnusedFunctions(entry\_functions));

…

pass\_seqs.push\_back(transform::SimplifyExpr());

…

transform::Pass seq = transform::Sequential(pass\_seqs);

if (targets.size() == 1) {

const auto& it = targets.begin();

With<Target> tctx((\*it).second);

relay\_module = seq(relay\_module);

} else {

relay\_module = seq(relay\_module);

}

…

relay\_module = transform::FuseOps()(relay\_module);

…

return relay\_module;

}

Optimize的过程就是使用transform::XXX函数对IRModule进行修改，可以使用transform::Sequential进行链式执行，也可以直接使用transform::XXX()(relay\_module)进行单步调用。关于Pass的解读文章比较多，列举几个写的比较简明易懂的：

【[从零开始学深度学习编译器】七，万字长文入门TVM Pass](https://zhuanlan.zhihu.com/p/378739411)

[【从零开始学深度学习编译器】八，TVM的算符融合以及如何使用TVM Pass Infra自定义Pass](https://zhuanlan.zhihu.com/p/380630801)

[TVM代码走读（三） 图优化1--初识PASS](https://zhuanlan.zhihu.com/p/149988448)

[TVM代码走读（五） 图优化3-- Constant Folding](https://zhuanlan.zhihu.com/p/151815380)

直接调用optimize会自动选择需要的pass，如果要单独查看每个pass的效果，可以使用demo中的debug\_optimze的方式，注意此处优化前需要先调用bind\_params\_by\_name，否则会对fold\_const这个pass产生影响。

**注意：debug模式只是用于理解Pass的过程，部署的时候不能这么用，因为改变过程中params可能已经发生改变。**

接下来追读一下此demo涉及到的优化过程。为了辅助查看结构的改变，简单写了一个RelayVisualizer，配合Netron使用 <https://github.com/Archermmt/tvm_walk_through/blob/master/relay_optimize/visualize.py>

此处感觉到Relay一个不太友好的地方，每个节点都没有ID，tensor的名字是按照increase顺序确定，所以查看优化前后效果的时候对不上节点名称，不知道TVM是不是已经添加了节点ID之类有辨识度的名字（翻了半天没找到），可以让优化过程的debug友好一些。

### Step 3.2.1 relay.transform.SimplifyInference

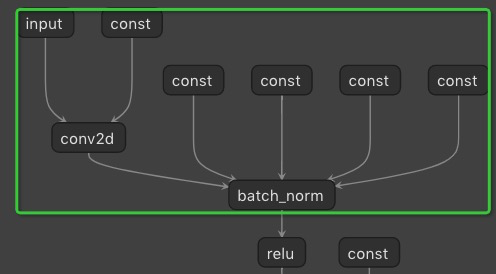
调用过程：transform::SimplifyInference

1 InferenceSimplifier::Mutate (ExprMutator::Mutate) -> InferenceSimplifier::VisitExpr\_(Call 和 TupleGetItemNode触发变形)

2 BatchNormToInferUnpack/GroupNormToInferUnpack…(执行变形)

这个过程的作用是把一些不好优化的算子拆成基础运算，常见的有batch\_norm，layer\_norm等等。先看优化前后的区别

Base Relay IR：



Netron查看完整信息

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

Relay IR after SimplifyInference：



Netron查看完整信息

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

直觉上看，这个Pass似乎增加了很多计算量，但是从处理逻辑来看，大部分后端engine对于基础算子支持的很好，拆分之后更好处理，并且拆分不增加额外运算（查看pytorch或者tensorflow的底层会发现，batch\_norm的底层基本就是按照缠粉出来的算子进行计算），在分block的时候只要block设置合理，也可以省去访存开销。这个过程的起作用的部分如下

@ tvm/src/relay/transforms/simplify\_inference.cc

Expr VisitExpr\_(const TupleGetItemNode\* n) final {

…

if (const auto\* call = new\_n->tuple.as<CallNode>()) {

if (call->op == batch\_norm\_op\_) {

return BatchNormToInferUnpack(call->attrs, call->args[0], call->args[1], call->args[2],

call->args[3], call->args[4], ty\_map\_.at(call->args[0]));

} …

}

return new\_e;

}

Expr VisitExpr\_(const CallNode\* n) {

auto new\_n = ExprMutator::VisitExpr\_(n);

if (n->op == batch\_norm\_op\_) {

ty\_map\_[new\_n.as<CallNode>()->args[0]] = n->args[0]->checked\_type();

}…

return new\_n;

}

注意batch\_norm节点输出是3个，所以遍历的时候节点类型是TupleGetItemNode。class InferenceSimplifier : public ExprMutator说明此优化类会遍历节点并进行变形。本demo中只有batch\_norm节点触发了Pass，代码如下

@ tvm/src/relay/transforms/simplify\_inference.cc

Expr BatchNormToInferUnpack(const Attrs attrs, Expr data, Expr gamma, Expr beta, Expr moving\_mean,

Expr moving\_var, Type tdata) {

…

Expr epsilon = MakeConstantScalar(ttype->dtype, static\_cast<float>(param->epsilon));

Expr var\_add\_eps = Add(moving\_var, epsilon);

…

scale = ExpandBiasToMatchAxis(scale, ndim, {axis});

shift = ExpandBiasToMatchAxis(shift, ndim, {axis});

Expr out = Multiply(data, scale);

out = Add(out, shift);

return out;

}

其实就是使用Expr表达式重写batch\_norm的计算过程，可以使用的Expr定义在

@ tvm/src/relay/transforms/pattern\_utils.h

…

inline Expr Add(Expr lhs, Expr rhs) {

static const Op& op = Op::Get("add");

return Call(op, {lhs, rhs}, Attrs(), {});

}

inline Expr Subtract(Expr lhs, Expr rhs) {

static const Op& op = Op::Get("subtract");

return Call(op, {lhs, rhs}, Attrs(), {});

}

…

大部分仍然是Call节点，所以最后打印的结果中才会出现很多的call。

### Step 3.2.2 ToANormalForm

调用过程：transform::ToANormalForm

1 DependencyGraph::Create(创建DependencyGraph)

2 CalcScope(设定图中每个节点的scope，只有在存在new\_scope的时候才会重新设定)

3 Fill::ToANormalForm(对一个scope中的所有节点构建赋值链)

4 Fill::VisitExpr && Fill::Compound(访问节点，添加链接)

这个Pass没有被显示声明为默认的pass，此demo中用到的地方只有和FoldConstant一起使用的时候做常量的数值计算。A-norm-form找了一些资料，但实在是由于个人编译水平有限看不懂，只能跟读一下代码了解个皮毛。关于DependencyGraph的梳理参照（上一节链接）

使用relay.transform.ToANormalForm作为唯一的pass设定demo中的debug\_optimize过程，效果如下

Base

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

%0 = nn.conv2d(%input0, meta[relay.Constant][0], strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7]);

%1 = nn.batch\_norm(%0, meta[relay.Constant][1], meta[relay.Constant][2], meta[relay.Constant][3], meta[relay.Constant][4]);

…

add(%91, meta[relay.Constant][101])

}

After optimize

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

let %x = meta[relay.Constant][0];

let %x1 = nn.conv2d(%input0, %x, strides=[2, 2], padding=[3, 3, 3, 3], channels=64, kernel\_size=[7, 7]);

let %x2 = meta[relay.Constant][1];

let %x3 = meta[relay.Constant][2];

let %x4 = meta[relay.Constant][3];

let %x5 = meta[relay.Constant][4];

let %x6 = nn.batch\_norm(%x1, %x2, %x3, %x4, %x5)；

…

let %x194 = add(%x192, %x193);

%x194

}

大致感觉是每一个会被消耗的tensor都变成了一个LetNode节点。然后根据Scope的level构建赋值链（let\_list）。核心部分逻辑如下：

@ tvm/src/relay/transforms/to\_a\_normal\_form.cc

Expr Fill::ToBasicBlockNormalForm(const Expr& e, const DependencyGraph& dg,

NodeScopeMap\* node\_scope, ExprSet\* lifted) {

Fill fi(dg, node\_scope, lifted);

auto var = fi.VisitExpr(e);

return fi.GetScope(e)->let\_list->Get(var);

}

…

Expr Fill::VisitExpr(const Expr& e, const Var& v) {

if (memo.count(e) == 0) {

memo.insert({e, ExprFunctor<Expr(const Expr&, const Var&)>::VisitExpr(e, v)});

} else if (v.defined()) {

GetScope(e)->let\_list->Push(v, memo.at(e));

}

auto ret = memo.at(e);

// if no include\_set is specified, every expression should be atomic.

if (include\_set\_ == nullptr) ICHECK(IsAtomic(ret));

return ret;

}

Expr Fill::VisitExpr(const Expr& e) { return this->VisitExpr(e, Var()); }

LetList的Push操作+Get操作会构建Let节点

@ tvm/src/relay/transforms/let\_list.h

class LetList {

public:

…

Var Push(Var pv, Expr expr) {

…

lets\_.emplace\_back(std::make\_pair(pv, expr));

return pv;

}

…

Expr Get(const Expr& body) {

ICHECK(!used\_);

Expr ret = body;

for (auto rit = lets\_.rbegin(); rit != lets\_.rend(); ++rit) {

ret = Let(std::get<0>(\*rit), std::get<1>(\*rit), ret);

}

used\_ = true;

return ret;

}

…

}

### Step 3.2.3 relay.transform.FoldScaleAxis

调用过程：transform::FoldScaleAxis

1 BackwardFoldScaleAxis

1.1 BackwardTransformerNode::Fold

1.2 BackwardPrep::Prepare(准备变换信息)

1.2.1 ExprVisitor::VisitExpr

1.2.2 Op::GetAttrMap<FBackwardPrep>("FScaleAxisBackwardPrep")(执行准备)

1.3 BackwardTransformerNode::Mutate(执行变换)

1.3.1 BackwardTransformerNode::Transform

1.3.2 Op::GetAttrMap<FBackwardTransform>("FScaleAxisBackwardTransform")(执行变换)

2 ForwardFoldScaleAxis

3 FoldConstant

此优化一般和SimplifyInference一起使用，作用是折叠Scale相关的操作到conv2d，add，substract，multiply等节点，这个Pass实际上是一个Sequential，包含了三个按顺序执行的子Pass。

@ tvm/src/relay/transforms/fold\_scale\_axis.cc

Pass FoldScaleAxis() {

// FoldScaleAxis pass contains the following three passes. Therefore, we can

// register it as a sequential pass.

Pass pass = Sequential({BackwardFoldScaleAxis(), ForwardFoldScaleAxis(), FoldConstant()},

"FoldScaleAxis");

return pass;

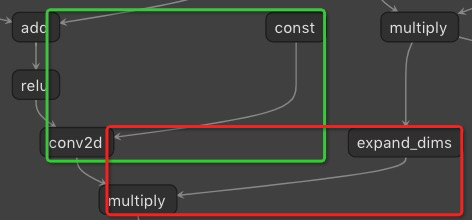
}

所以实际过程是三个Pass

### Step 3.2.3.1 BackwardFoldScaleAxis

折叠合适的CallNode之前的ScaleAxis操作，效果如下

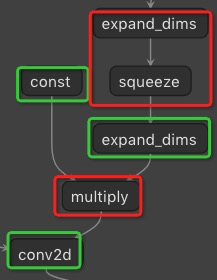
Relay IR after SimplifyInference：



Netron查看完整信息

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

Relay IR after BackwardFoldScaleAxis：



Netron查看完整信息

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

Pass由Prepare和Mutate两步组成。对应不同的Call算子由不同的Prepare函数（准备信息，信息用Message结构保存）和Transform函数（进行变形）组成，例如

@ tvm/src/relay/transforms/fold\_scale\_axis.cc

Message Conv2DBackwardPrep(const Call& call, const Array<Message>& in\_messages) {

…

Layout kernel\_layout(param->kernel\_layout);

Layout out\_layout(param->out\_layout == "" ? param->data\_layout : param->out\_layout);

…

if (param->groups == 1 || is\_depthwise\_conv2d) {

…

return Message(arr, false);

}

}

return NullValue<Message>();

}

Expr Conv2DBackwardTransform(const Call& call, const Message& message, const Expr& scale,

const BackwardTransformer& transformer) {

…

const auto\* param = call->attrs.as<Conv2DAttrs>();

…

Expr data = transformer->Transform(call->args[0], NullValue<Message>(), NullValue<Expr>());

Expr weight = transformer->Transform(call->args[1], NullValue<Message>(), NullValue<Expr>());

…

if (is\_simple) {

wscale = ExpandBiasToMatchAxis(scale, kernel\_layout.ndim(), {big\_ko\_axis});

} …

weight = Multiply(weight, wscale);

return Call(call->op, {data, weight}, call->attrs, call->type\_args);

}

此demo中触发变形操作的是nn.conv2d和multiply节点，此处追读的时候比较绕，因为这两个是协同变换的，参照变化图示中绿色和红色部分，multiply的变形添加squeeze，conv2d的变形添加expand\_dims，并改变multiply的位置。这个Pass中进行对一个节点进行变形的时候会触发args的Transform进行递归，最终的变形顺序仍然是后向DFS，不过在访问到multiply节点的时候会先进行squeeze的插入，然后继续向上递归变形。

### Step 3.2.3.2 ForwardFoldScaleAxis

折叠合适的CallNode之前的ScaleAxis操作，此demo中没有改变IR，不进行跟读。大致逻辑和Step 3.2.3.1 BackwardFoldScaleAxis中相同，由于是top-down的顺序，读起来也会相对顺一些。

### Step 3.2.4 FoldConstant

调用过程：transform::FoldConstant

1 ConstantFolder::Mutate(MixedModeMutator::Mutate) -> ConstantFolder::Rewrite\_(Call 节点触发)

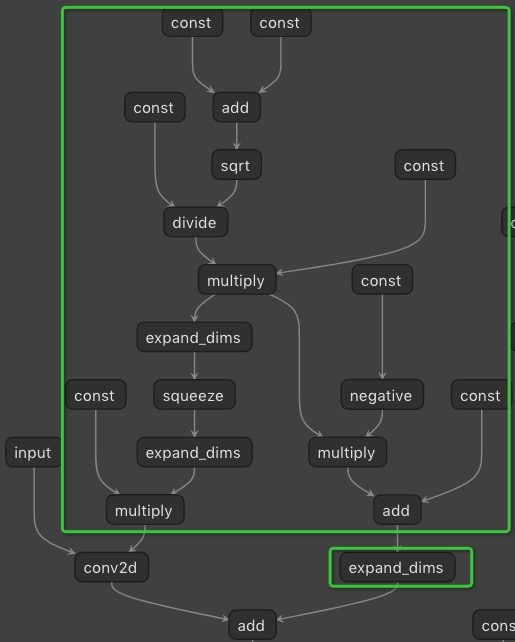
2 ConstantChecker::Check && ConstantFolder::ConstEvaluate(检查const节点，通过的话进行常量折叠)

3 transform::Sequential(对const节点进行变换)

4 CreateInterpreter && ObjectToExpr(JIT执行)

这个Pass在FoldScaleAxis被调用，但是独立使用的机会更多，所以单开一小节进行分析。FoldConstant应该是TVM中最常用的也是效果最好的Pass没有之一，作用相当于放了一个AOE清场，先看看效果：

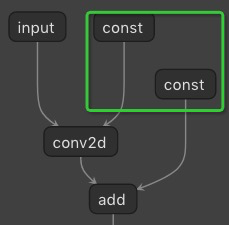
Relay IR after BackwardFoldScaleAxis：



Netron查看完整信息

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

Relay IR after FoldConstant：



Netron查看完整信息

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

顿时感觉世界都清净了。常量折叠应该是几乎所有编译器都会使用的优化手段，包括我之前独立架构的DLRouter，算法大同小异，按照const的依赖关系拓展frontier，对于所有依赖都是Const的节点进行折叠转义成Const，最终的结果就是所有完全依赖于Const的节点都会被折叠（例如Relay IR after BackwardFoldScaleAxis中绿色圈起来的部分，用无依赖的const进行保守拓展可以访问到绿色框中所有的节点）。

本demo中所有触发转换的节点都会先被评估是否是const的完全依赖节点，方式是对args进行检查

@ tvm/src/relay/transforms/fold\_constant.cc

class ConstantChecker : private ExprVisitor {

public:

bool Check(const Expr& expr) {

…

if (expr.as<ConstantNode>()) {

return true;

}

const auto it = memo\_.find(expr);

if (it != memo\_.end()) return it->second;

VisitExpr(expr);

return memo\_[expr]; // return memoized result or the default value false

}

private:

std::unordered\_map<Expr, bool, ObjectPtrHash, ObjectPtrEqual> memo\_;

void VisitExpr\_(const TupleNode\* n) final {

bool result = true;

for (const auto& field : n->fields) {

if (!Check(field)) {

result = false;

break;

}

}

memo\_[GetRef<Tuple>(n)] = result;

}

};

即const节点或者tuple(const,const,…)形式的节点被认为是属于const输入，判断成功后直接折叠成Const

@ tvm/src/relay/transforms/fold\_constant.cc

Expr ConstEvaluate(Expr expr) {

std::vector<transform::Pass> passes = {transform::FuseOps(0), transform::ToANormalForm(),

transform::InferType()};

Function func;

…

auto seq = transform::Sequential(passes);

mod = seq(mod);

…

With<PassContext> fresh\_build\_ctx(PassContext::Create());

FInterpreter executor = CreateInterpreter(mod, dev, target);

return ObjectToExpr(executor(expr));

}

这里引入了一种eager处理数据的机制FInterpreter以及一些依赖的Pass，ToANormalForm解析见Step 3.2.2 ToANormalForm节，FuseOps在下一节进行跟读。接下来主要看一下CreateInterpreter这个解释器的使用方式。

@ tvm/src/relay/backend/interpreter.cc

TypedPackedFunc<ObjectRef(Expr)> CreateInterpreter(IRModule mod, Device device, Target target) {

if (mod.defined()) {

…

}

auto intrp = std::make\_shared<Interpreter>(mod, device, target);

auto packed = [intrp](Expr expr) {

auto f = DetectFeature(expr);

ICHECK(f.is\_subset\_of(FeatureSet::All() - fGraph));

return intrp->Eval(expr);

};

return TypedPackedFunc<ObjectRef(Expr)>(packed);

}

后端优化的Pass EtaExpand在此demo中没有起作用，暂时略过。起作用的代码为

auto intrp = std::make\_shared<Interpreter>(mod, device, target);

intrp->Eval(expr);

此demo中调用链为Interpreter::Eval->Interpreter::VisitExpr\_(LetNode)->Interpreter::VisitExpr\_(CallNode)->Interpreter::VisitExpr\_(LetNode)->Interpreter::Invoke->Interpreter:: InvokePrimitiveOp

之后会调用CompileEngine::JIT相关的函数。这部分已经进入编译后端范畴，不属于relay优化过程，后面的章节再继续深入解析。

### Step 3.2.5 AlterOpLayout

调用过程：transform::AlterOpLayout

1 ForwardRewrite

1.1 ForwardRewriter::Rewrite -> MixedModeMutator::VisitExpr

1.1.1 ForwardRewriter::Rewrite\_(Call节点触发，调用frewrite)

1.2 LayoutRewriter<AlterTransformMemorizer>(frewrite的实现)

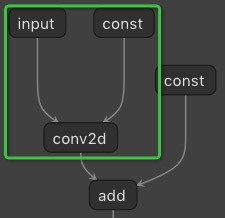
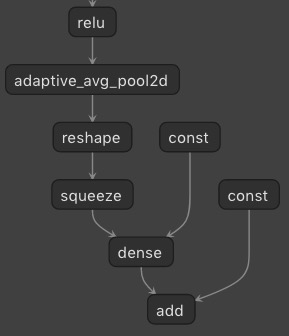
1.2.1 InferCorrectLayouts

1.2.2 AlterTransformMemorizer::CallWithNewLayouts

1.2.2.1 Op::GetAttrMap<FTVMAlterOpLayout>("FTVMAlterOpLayout")

用于将layout转换成硬件更友好的layout。这个Pass优化后可能产生新的待优化算子例如expand\_dims，所以一般使用后会再添加FoldConstant Pass。此处感觉如果Pass直接的关系可以用一种依赖Tree的形式管理会好一点，优化过程循环遍历整个Tree，直到IR不再改变。先看一下优化效果

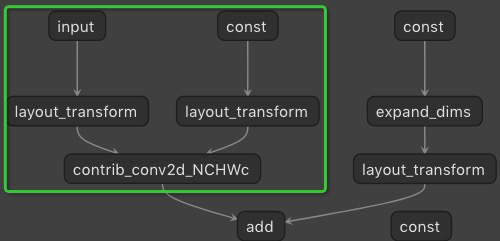
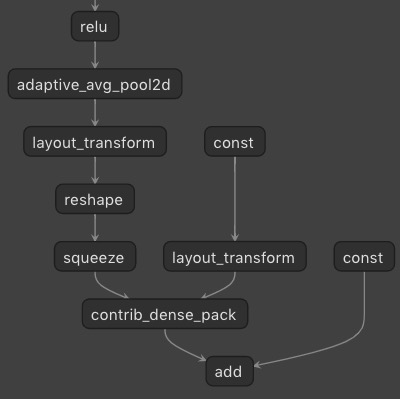
Relay IR after FoldConstant：

Netron查看完整信息

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

Relay IR after AlterOpLayout：

Netron查看完整信息

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

变形过程和上面几个Pass大同小异，这个Pass有一个需要仔细分析地方就是使用python对C++中的Op进行属性/方法的注册。先追踪一下这个Pass的调用逻辑：

@ tvm/src/relay/transforms/alter\_op\_layout.cc

Expr AlterOpLayout(const Expr& expr) {

// TODO(@icemelon9): need to rerun type inference after applying an alter op.

AlterTransformMemorizer alterMemorizer(make\_object<AlterTransformMemorizerNode>());

auto fcontext = [&](const Call& call) -> ObjectRef { return alterMemorizer; };

return ForwardRewrite(expr, LayoutRewriter<AlterTransformMemorizer>, fcontext);

}

可以看出由两部分共同实现，ForwardRewrite驱动LayoutRewriter进行整个IR的重写。

ForwardRewrite调用链为ForwardRewriter::Rewrite->ForwardRewriter::VisitExpr，进行后向DFS访问（MixedModeMutator::VisitExpr进行算子重写）

其中比较常用的部分为访问Call节点的Rewrite\_函数

@ tvm/src/relay/transforms/forward\_rewrite.cc

Expr Rewrite\_(const CallNode\* call\_node, const Expr& post) final {

const Call& ref\_call = GetRef<Call>(call\_node);

PackedFunc frewrite;

…

if (frewrite != nullptr) {

Expr res = frewrite(ref\_call, call\_args,

fcontext\_ != nullptr ? fcontext\_(ref\_call) : ObjectRef(nullptr));

if (res.defined()) return res;

…

}

}

…

}

比较重要的部分就是用frewrite函数对象处理Call节点，此对象对应的LayoutRewriter是一个模版函数，实现如下

@ tvm/src/relay/transforms/transform\_layout.h

template <class TransformMemorizerT>

Expr LayoutRewriter(const Call& ref\_call, const Array<Expr>& new\_args, const ObjectRef& ctx) {

…

// old\_in, old\_out = op.infer(old\_in)

bool success = false;

std::tie(old\_in, old\_out, success) =

InferCorrectLayouts(ref\_call, Array<Layout>(nullptr), old\_in, types);

…

// new\_op = alter(op)

Call new\_call = memorizer.CallWithNewLayouts(ref\_call, normal\_new\_args);

…

// new\_in2, new\_out = op.infer(new\_in)

if (new\_call->op->IsInstance<OpNode>()) {

success = false;

std::tie(new\_in2, new\_out, success) = InferCorrectLayouts(new\_call, new\_in, old\_in, types);

…

} …

…

}

核心分成两部分：InferCorrectLayouts 和 CallWithNewLayouts

#### Step 3.2.5.1 InferCorrectLayouts

在CallWithNewLayouts前后分别调用了一次InferCorrectLayouts，前调用没有传入new\_in\_layouts，作为检查节点推断是否可以成功；后调用是为了推断新节点的layout和属性。

@ tvm/src/relay/transforms/infer\_layout\_utils.h

static inline std::tuple<Array<Layout>, Array<Layout>, bool> InferCorrectLayouts(

const Call& call, const Array<Layout>& new\_in\_layouts, const Array<Layout>& old\_in\_layouts,

const Array<tvm::relay::Type>& old\_in\_types) {

static auto finfer\_layout = Op::GetAttrMap<FInferCorrectLayout>("FInferCorrectLayout");

…

if (finfer\_layout.count(op)) {

Array<Array<Layout>> inferred\_layouts;

inferred\_layouts = finfer\_layout[op](call->attrs, new\_in\_layouts, old\_in\_layouts, old\_in\_types);

…

return std::make\_tuple<>(inferred\_layouts[0], inferred\_layouts[1], true);

} …

}

依然是使用Op绑定的FInferCorrectLayout函数进行layout的推断，同时会更改参数，例如concat节点的infer\_layout函数

@ tvm/src/relay/op/tensor/transform.h

static inline Array<Array<Layout>> ConcatenateLayout(const Attrs& attrs,

const Array<Layout>& new\_in\_layouts,

const Array<Layout>& old\_in\_layouts,

const Array<tvm::relay::Type>& old\_in\_types) {

ConcatenateAttrs\* param = const\_cast<ConcatenateAttrs\*>(attrs.as<ConcatenateAttrs>());

…

if (new\_in\_layouts.defined()) {

const auto& concate\_dim = old\_in\_layouts[0][axis];

if (all\_input\_layouts\_same) {

auto new\_index = new\_in\_layouts[0].IndexOf(concate\_dim);

ret = new\_in\_layouts[0];

param->axis = new\_index;

is\_new\_layout\_selected = true;

}…

}…

}

#### Step 3.2.5.2 CallWithNewLayouts

Call new\_call = memorizer.CallWithNewLayouts(ref\_call, normal\_new\_args);对节点进行变形，代码为

Call CallWithNewLayouts(const Call& ref\_call, const std::vector<Expr>& new\_args) override {

static auto falter\_layout = Op::GetAttrMap<FTVMAlterOpLayout>("FTVMAlterOpLayout");

Op op = Downcast<Op>(ref\_call->op);

…

if (falter\_layout.count(op)) {

…

Expr altered\_value =

falter\_layout[op](ref\_call->attrs, new\_args, tinfos, ref\_call->checked\_type());

…

}

…

const CallNode\* new\_call = new\_e.as<CallNode>();

ICHECK(new\_call) << "Can only replace the original operator with another call node";

return GetRef<Call>(new\_call);

}

通过static auto falter\_layout = Op::GetAttrMap<FTVMAlterOpLayout>("FTVMAlterOpLayout")找到对应的函数。FTVMAlterOpLayout属性是通过 reg.register\_alter\_op\_layout("XXX")实现的，注册机制参考上一篇（链接。。。）

### Step 3.2.6 FuseOps

调用过程：transform::FuseOps

1 FuseMutator::Transform

2 IndexedForwardGraph::Create(构建IndexedForardGraph)

3 GraphPartitioner::Partition (算子分组)

3.1 GraphPartitioner::InitGroups(每个节点分成一组) ->

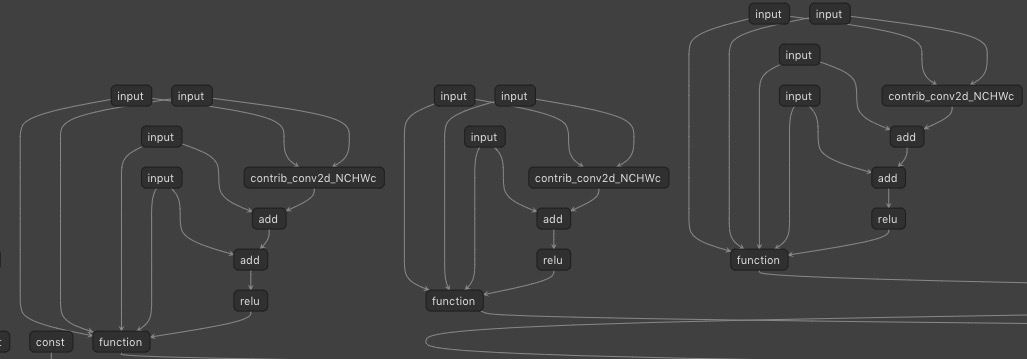
DominatorTree::PostDom(构建依赖树) ->

GraphPartitioner::RunFuse(进行分组)

4 FuseMutator::Mutate(MixedModeMutator::Mutate)

5 FuseMutator::Rewrite\_ && FuseMutator::MakeNewFunction (一组算子包成Function)

After FuseOps:



Netron查看完整信息

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

此Pass是常用的Pass之一，作用是把inplace类型的算子和可以融合的算子放在一起，调度的时候减少访存开销，是打击访存密集型的算子的一种手段。上一节中梳理了这个Pass需要使用到的基本结构IndexedForwardGraph，GraphPartitioner和DominatorTree，具体梳理见。。。（链接）。

#### Step 3.2.6.1 RunFuse

跟读一下核心部分RunFuse，填一下上一节留的坑。

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

void RunFuse(const IndexedForwardGraph& graph, const DominatorTree& post\_dom\_tree, int phase) {

for (size\_t nid = 0; nid < groups\_.size(); ++nid) {

// the group of current node has been specified already.

auto\* graph\_node = graph.post\_dfs\_order[nid];

auto\* dom\_node = post\_dom\_tree.nodes[nid];

…

if (phase == 2) {

…

if (CheckPath(graph\_node, dom\_node->parent->gnode, fcond)) {

CommitFuse(graph\_node, dom\_node->parent->gnode);

}

…

continue;

}

…

if (group\_node->pattern == kOutEWiseFusable) {

…

if (CheckPath(graph\_node, dom\_node->parent->gnode, fcond)) {

CommitFuse(graph\_node, dom\_node->parent->gnode);

}

…

} …

}

}

看起来比较复杂，但仔细查看会发现其实所有的过程都是用CheckPath对节点到最近邻祖先路径上所有节点进行检查，如果所有检查都通过则使用CommitFuse进行合并。CheckPath的代码逻辑比较简单

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

template <typename F>

bool CheckPath\_(IndexedForwardGraph::Node\* src, IndexedForwardGraph::Node\* sink, F fcond) {

…

for (auto link = src->outputs.head; link != nullptr; link = link->next) {

if (!CheckPath\_(link->value.node, sink, fcond)) return false;

}

return true;

}

…

template <typename F>

bool CheckPath(IndexedForwardGraph::Node\* src, IndexedForwardGraph::Node\* sink, F fcond) {

…

for (auto link = src->outputs.head; link != nullptr; link = link->next) {

if (!CheckPath\_(link->value.node, sink, fcond)) return false;

}

return true;

}

核心就是路径检查，常用编译器中的算子组合查找经常会用到的一种算法。

CommitFuse的核心部分是

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

void MergeFromTo(Group\* child, Group\* parent) {

child = child->FindRoot();

parent = parent->FindRoot();

if (child == parent) return;

// update the number of nodes of the parent group

parent->num\_nodes += child->num\_nodes;

child->parent = parent;

…

}

也就是对合格的节点的root进行赋值，并对num\_nodes进行修改，标定此融合之后root节点代表的子图中节点总数。融合的具体规则由于时间有限没有仔细追读，等以后需要用到的时候再来填坑。

#### Step 3.2.6.2 构建Function

分组完毕之后直接把一组包装成一个Function，例如对Call的处理方式中

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

Expr Rewrite\_(const CallNode\* call, const Expr& post) {

if (call->op.as<OpNode>()) {

…

auto\* ret\_group = gmap\_.at(call)->FindRoot();

Array<Expr> new\_args = GetNewArguments(call->args, ret\_group);

auto new\_call = Call(call->op, new\_args, call->attrs, call->type\_args, call->span);

if (ret\_group->root\_ref == call) {

…

return MakeNewFunction(ret\_group, call->checked\_type(), new\_call);

}…

}…

}

GetNewArguments 和 MakeNewFunction部分逻辑比较简洁，分别用来构建arguments和构建Function

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

Expr MakeNewFunction(GraphPartitioner::Group\* group, Type ret\_type, Expr body) {

// If the function has no call, it is not a primitive function.

struct HasCallVisitor : ExprVisitor {

bool has\_call = false;

void VisitExpr\_(const CallNode\* op) final { has\_call = true; }

} visitor;

visitor(body);

const GroupInfo& ginfo = ginfo\_[group];

auto func = Function(ginfo.params, body, ret\_type, {});

func = WithAttr(std::move(func), attr::kPrimitive, tvm::Integer(visitor.has\_call));

return Call(func, ginfo.arguments, Attrs());

}

Array<Expr> GetNewArguments(const tvm::Array<Expr>& args,

GraphPartitioner::Group\* current\_group) {

Array<Expr> new\_args;

for (auto arg : args) {

auto\* arg\_group = gmap\_.at(arg.get())->FindRoot();

auto type = arg->checked\_type();

Expr new\_arg = this->Mutate(arg);

if (current\_group != arg\_group) {

Var param = ginfo\_[current\_group].GetOrAllocParam(new\_arg, type);

new\_args.push\_back(param);

} else {

new\_args.push\_back(new\_arg);

}

}

return new\_args;

}

# 总结

优化过程准备工作比较简单，核心在于各种Pass，其中每个Pass都值得仔细研究一遍。由于精力和时间有限，本人只追读了在此demo中起作用的部分。大体梳理为：

SimplifyInference：将BatchNorm，LayerNorm，GroupNorm，InstanceNorm操作变成基础数学运算

FoldScaleAxis：三合一Pass，实际执行BackwardFoldScaleAxis，ForwardFoldScaleAxis，FoldConstant，前两个对conv2d，leaky\_relu，elemwise族算子进行变换，在合适的地方添加expand\_dims

FoldConstant：常量折叠，对每个需要折叠的常量使用FuseOps，ToANormalForm，InferType Pass进行修改，得到a-normal-form的表达式，再使用Interpreter的JIT进行常量计算，JIT部分后续章节再解析。

AlterOpLayout：部分与tir的优化重合，关键结构在于使用注册的FTVMAlterOpLayout函数进行算子替换，注册的机制核心是tvm.ir.register\_op\_attr->\_ffi\_api.RegisterOpAttr，这种机制在后续章节中会广泛用到。

FuseOps：算子融合，准确的讲应该是算子分组，对可以不保留中间参数的算子进行Function的封装，计算的行为和顺序上没有变化。