【我与TVM二三事 前篇（4）】strategy初体验

上一篇（链接。。。）解析了relay优化的过程和一些demo中用到的Pass，但是遗漏了一个知识点，就是涉及到AlterOpLayout这个优化的时候，会调用

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

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

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

…

这个函数，上一篇没有追读这个过程，只看了结果。这个函数的作用是将一个算子替换为另一个（conv2d -> contrib\_conv2d\_nchwc），并更新workload对应的config。这部分没有涉及到Stmt的部分，但用到了lower过程中常用的一些概念，例如strategy，compute，schedule等等，可以通过代码跟读对这些结构有一个基础了解。

# Part1 strategy

开始追读\_alter\_conv2d\_layout 代码之前，有必要了解一下strategy的机制，因为选择strategy是这个函数的基础。TVM中每个op都可以注册FTVMStrategy函数，注册过程和（贴链接。。。prepare）中提到的FTVMAlterOpLayout注册过程很类似，也分两部分

准备过程（import tvm时调用）：

1 override\_native\_generic\_func(conv2d\_strategy) (将conv2d\_strategy变为函数holder并设定默认函数)

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

3 strategy.add\_implementation -> OpStrategy::AddImplementation(添加实现，需要compute和schedule函数)

注册过程（import tvm时调用）：

1 reg.register\_strategy

2 tvm.ir.register\_op\_attr

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

注册过程和之前FTVMAlterOpLayout注册的没有太大区别，首先看一下准备过程

@ tvm/python/tvm/relay/op/strategy/generic.py

@override\_native\_generic\_func("conv2d\_strategy")

def conv2d\_strategy(attrs, inputs, out\_type, target):

…

override\_native\_generic\_func和上一篇FTVMAlterOpLayout注册中（链接。。。。。）追读的generic\_func作用相同，但会把包裹的函数注册为default函数。

得到函数holder后，同样可以使用register进行信息收集

@ tvm/python/tvm/relay/op/strategy/x86.py

@conv2d\_strategy.register("cpu")

def conv2d\_strategy\_cpu(attrs, inputs, out\_type, target):

…

if groups == 1:

if layout == "NCHW":

if topi.x86.is\_int8\_hw\_support(data.dtype, kernel.dtype):

…

else:

strategy.add\_implementation(

wrap\_compute\_conv2d(topi.x86.conv2d\_nchw),

wrap\_topi\_schedule(topi.x86.schedule\_conv2d\_nchw),

name="conv2d\_nchw.x86",

)

…

strategy.add\_implementation会调用

@ tvm/src/relay/ir/op\_strategy.cc

TVM\_REGISTER\_GLOBAL("relay.op.\_OpStrategyAddImplementation")

.set\_body([](TVMArgs args, TVMRetValue\* rv) {

OpStrategy strategy = args[0];

FTVMCompute compute = args[1];

FTVMSchedule schedule = args[2];

…

strategy.AddImplementation(compute, schedule, name, plevel);

});

…

void OpStrategy::AddImplementation(FTVMCompute fcompute, FTVMSchedule fschedule, String name,

int plevel) {

…

ObjectPtr<OpSpecializationNode> n = make\_object<OpSpecializationNode>();

n->condition = curr\_cond;

op\_spec = OpSpecialization(n);

op\_spec.AddImplementation(fcompute, fschedule, std::move(name), plevel);

self->specializations.push\_back(op\_spec);

}

对name对应的FTVMCompute和FTVMSchedule进行注册，每对FTVMCompute: FTVMSchedule注册到一个OpSpecialization结构中，再由OpImplementation记录fcompute和fschefule。使用的时候通过specializations数组进行取用，最终通过

@ tvm/src/relay/ir/op\_strategy.cc

Array<te::Tensor> OpImplementation::Compute(const Attrs& attrs, const Array<te::Tensor>& inputs,

const Type& out\_type) {

return (\*this)->fcompute(attrs, inputs, out\_type);

}

te::Schedule OpImplementation::Schedule(const Attrs& attrs, const Array<te::Tensor>& outs,

const Target& target) {

return (\*this)->fschedule(attrs, outs, target);

}

调用FTVMCompute和FTVMSchedule函数。

# Part2 FTVMCompute

调用前：autotvm.register\_topi\_compute

1 \_register\_task\_compute(TASK\_TABLE填充任务)

2 task\_env.add\_task(为autotvm添加task)

3 DispatchContext.current.query(找到合适的配置)

调用中：fcompute(cfg,\*args),cfg是<调用前：3>得到的结果

调用后：

1 attrs["workload"] = workload(为op添加合适的ID:workload)

2 tvm.te.\_ffi\_api.ComputeOp(制作ComputeOp，为构建schedule做准备)

FTVMCompute对象代表的是一种计算过程，就是一个op是怎么计算的。以本例中调用到的函数为例

@ tvm/python/tvm/topi/x86/conv2d.py

def conv2d\_nchw(data, kernel, strides, padding, dilation, out\_dtype):

layout = "NCHW"

packed\_out = conv2d\_NCHWc(data, kernel, strides, padding, dilation, layout, layout, out\_dtype)

return unpack\_NCHWc\_to\_nchw(packed\_out, out\_dtype)

…

@autotvm.register\_topi\_compute("conv2d\_NCHWc.x86")

def conv2d\_NCHWc(cfg, data, kernel, strides, padding, dilation, layout, out\_layout, out\_dtype):

…

return nn.conv2d\_NCHWc(data, kernel, strides, padding, dilation, layout, out\_layout, out\_dtype)

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

def conv2d\_NCHWc(data, kernel, stride, padding, dilation, layout, out\_layout, out\_dtype="float32"):

…

return te.compute(

oshape,

lambda n, oc\_chunk, oh, ow, oc\_block: te.sum(

data\_pad[

n,

idxdiv(ic, ic\_bn),

oh \* HSTR + kh \* dilation\_h,

ow \* WSTR + kw \* dilation\_w,

idxmod(ic, ic\_bn),

].astype(out\_dtype)

\* kernel[oc\_chunk, idxdiv(ic, ic\_bn), kh, kw, idxmod(ic, ic\_bn), oc\_block],

axis=[ic, kh, kw],

),

name="conv2d\_NCHWc",

tag="conv2d\_NCHWc",

)

…

def unpack\_NCHWc\_to\_nchw(packed\_out, out\_dtype):

…

unpacked\_out = te.compute(

oshape,

lambda n, c, h, w: packed\_out[n, idxdiv(c, oc\_bn), h, w, idxmod(c, oc\_bn)].astype(

out\_dtype

),

name="output\_unpack",

tag=tag.INJECTIVE + ",unpack\_nchwc",

)

return unpacked\_out

可见返回的最后调用的是te.compute，compute结构会在后续章节进行到tvm\_lower的时候讲到，暂时不深入解析。主要看一下register\_topi\_compute这个包装器

@ tvm/python/tvm/autotvm/task/topi\_integration.py

def register\_topi\_compute(task\_name, func=None):

…

def \_decorate(topi\_compute):

@\_register\_task\_compute(task\_name)

def wrapper(\*args, \*\*kwargs):

…

if task\_env is not None and task\_env.tracing:

task\_env.add\_task(task\_name, args)

workload = args\_to\_workload(args, task\_name)

tgt = Target.current()

cfg = DispatchContext.current.query(tgt, workload)

node = topi\_compute(cfg, \*args)

…

attrs["workload"] = workload

if isinstance(op, tensor.ComputeOp):

op = tvm.te.\_ffi\_api.ComputeOp(op.name, op.tag, attrs, op.axis, op.body)

…

if isinstance(node, tensor.Tensor):

return op.output(0)

return [op.output(i) for i in range(len(node))]

return wrapper

if func:

return \_decorate(func)

return \_decorate

-> @ tvm/python/tvm/autotvm/task/task.py

def \_register\_task\_compute(name, func=None):

…

def \_do\_reg(f):

…

tmpl = TASK\_TABLE[name]

…

tmpl.fcompute = f

return f

…

这个包装器主要做了三件事：

1. 通过\_register\_task\_compute把当前的compute和task\_name添加到全局TABLE中，此函数在autotvm的时候会用到
2. 通过task\_env.add\_task添加task，被记录的task在autotvm的时候会用到
3. 重新设定attrs，包括workload属性。Workload就是对compute节点输入参数的序列化描述，作为一个ID使用，例如：

args (Tensor(shape=[1, 3, 224, 224], op.name=placeholder), Tensor(shape=[64, 3, 7, 7], op.name=placeholder), (2, 2), (3, 3, 3, 3), (1, 1), 'NCHW', 'NCHW', 'float32')

workload ('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')

# Part3 FTVMSchedule

调用前：autotvm.register\_topi\_schedule

1 \_register\_task\_schedule(TASK\_TABLE填充任务)

2 DispatchContext.current.query(找到合适的配置)

调用中：fschedule(cfg,\*args),cfg是<调用前：2>得到的结果

FTVMSchedule需要和FTVMCompute共同使用，代表的是计算过程的配置，例如内存怎么设定，怎么循环访问内存等等。以本例中调用的函数为例

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

@autotvm.register\_topi\_schedule("conv2d\_NCHWc.x86")

def schedule\_conv2d\_NCHWc(cfg, outs):

"""Create schedule for tensors"""

outs = [outs] if isinstance(outs, te.tensor.Tensor) else outs

s = te.create\_schedule([x.op for x in outs])

def \_callback(op):

…

traverse\_inline(s, outs[0].op, \_callback)

return s

最终返回的是te.create\_schedule得到的对象，在后续章节中会解析。此处追读一下常用的函数register\_topi\_schedule

@ tvm/python/tvm/autotvm/task/topi\_integration.py

def register\_topi\_schedule(task\_name, func=None):

def \_decorate(topi\_schedule):

@\_register\_task\_schedule(task\_name)

def wrapper(outs, \*args, \*\*kwargs):

…

workload = get\_workload(outs, task\_name)

…

cfg = DispatchContext.current.query(tgt, workload)

return topi\_schedule(cfg, outs, \*args, \*\*kwargs)

return wrapper

…

-> @ tvm/python/tvm/autotvm/task/task.py

def \_register\_task\_schedule(name, func=None):

def \_do\_reg(f):

…

tmpl = TASK\_TABLE[name]

…

tmpl.fschedule = f

return f

…

核心逻辑就是为TASK\_TABLE添加一个schedule的函数，key是指定的name，执行schedule的时候会从DispatchContext查找适配当前tgt的最优配置，也就是schedule\_conv2d\_NCHWc(cfg, outs)中的cfg参数，进行schedule的构建。

# Part4 \_alter\_conv2d\_layou函数执行

调用过程：

1 select\_implementation

1.1 get\_valid\_implementations(运行当前op所有strategy得到实现)

1.2 fcompute && autotvm.task.get\_workload(对每个impl执行fcompute得到outs和workload)

1.3 dispatch\_ctx.query(查询历史记录得到workload对应的最好配置)

1.4 best\_autotvm\_impl = impl(更新最优记录)

2 dispatch\_ctx.update(更新配置)

回过头来看一下\_alter\_conv2d\_layout的代码，感受一下strategy和compute的使用过程，schedule在这个函数调用过程中没有用到，后续章节填坑。

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

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

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

…

if isinstance(dispatch\_ctx, autotvm.task.ApplyGraphBest):

…

else:

impl, outs = relay.backend.compile\_engine.select\_implementation(

relay.op.get("nn.conv2d"), attrs, tinfos, out\_type, target

)

…

if topi\_tmpl == "conv2d\_NCHWc.x86":

# we only convert conv2d\_NCHW to conv2d\_NCHWc for x86

if data\_layout == "NCHW" and kernel\_layout == "OIHW":

…

dispatch\_ctx.update(target, new\_workload, cfg)

…

\_alter\_conv2d\_layout过程可以分成两个部分：得到implement和更新workload

## Part4.1 select\_implementation：得到implement

@ tvm/python/tvm/relay/backend/compile\_engine.py

def select\_implementation(op, attrs, inputs, out\_type, target, use\_autotvm=True):

…

all\_impls = get\_valid\_implementations(op, attrs, inputs, out\_type, target)

best\_plevel\_impl = max(all\_impls, key=lambda x: x.plevel)

…

for impl in all\_impls:

outs = impl.compute(attrs, inputs, out\_type)

outputs[impl] = outs

workload = autotvm.task.get\_workload(outs)

…

cfg = dispatch\_ctx.query(target, workload)

…

if best\_cfg is None or best\_cfg.cost > cfg.cost:

best\_autotvm\_impl = impl

best\_cfg = cfg

…

if best\_autotvm\_impl:

…

return best\_autotvm\_impl, outputs[best\_autotvm\_impl]

…

首先使用get\_valid\_implementations得到op注册的所有FTVMStrategy。

@ tvm/python/tvm/relay/backend/compile\_engine.py

def get\_valid\_implementations(op, attrs, inputs, out\_type, target):

fstrategy = op.get\_attr("FTVMStrategy")

with target:

strategy = fstrategy(attrs, inputs, out\_type, target)

…

然后按顺序调用compute函数（对于此example，compute函数就是Part2 FTVMCompute中追读的conv2d\_nchw函数），此函数调用的时候添加workload属性作为计算过程的id。最后dispatch\_ctx.query则通过target和workload进行查找看有没有最优的配置，如果有则得到best\_autotvm\_impl，返回此配置。若果涉及到多种实现，则按照cost比较每个workload的好坏，返回最优的配置。

## Part4.2更新workload

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

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

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

…

if topi\_tmpl == "conv2d\_NCHWc.x86":

# we only convert conv2d\_NCHW to conv2d\_NCHWc for x86

if data\_layout == "NCHW" and kernel\_layout == "OIHW":

…

dispatch\_ctx.update(target, new\_workload, cfg)

就是用上一步得到的最优配置重新配置入参，并构建和新的入参对应的workload，更新dispatch\_ctx中的配置数据。

# 总结

\_alter\_conv2d\_layout函数主要作用是对算子进行变换，同时更新workload和配置。

其中涉及通过select\_implementation得到最优配置这个过程，此机制在lower过程中还会用到。

此过程会从注册的strategy中选择最优实现，选择的时候调用compute函数会记录task给autotvm使用，同时得到workload作为计算过程的id。接下来dispatch\_ctx会根据target和workload查找最优配置，最后经过评估得到最优配置