【我与TVM二三事 前篇（9）】relay codegen

TVM常规的使用方式是直接用relay.build生成最终的tir表达形式，relay optimize（链接。。。）解析中为了方便代码跟读，只用了optimize，本篇直接用build走完整流程，追读optimize之后的codegen部分。跟读使用如下代码

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

入口：RelayBuildModule::Build -> RelayBuildModule::BuildRelay

前置过程：Optimize(优化relay，略)

调用流程：GraphCodegen

1 GraphCodegen::Init -> CompileEngineImpl(构建全局实例)

2 GraphCodegen::Codegen

2.1 relay.backend.GraphPlanMemory(分配内存) -> StorageAllocator::Plan

2.1.1 StorageAllocaInit::GetInitTokenMap

2.1.1.1 CollectDeviceInfo (记录device\_copy算子)

2.1.1.2 StorageAllocaInit::Run(记录Const和Call需要的存储信息)

2.1.2 StorageAllocator::Run(模拟runtime，分配内存)

2.1.2.1 Request(检查可复用内存) or/and Alloc(分配)

2.1.3 Map<Expr, Array<IntegerArray> > smap(绑定信息)

2.2 GraphExecutorCodegen::VisitExpr(遍历进行Lower,存储json描述)

2.2.1 CCacheKey && CachedFunc(构建LowerFunc)

2.2.1.1 CCacheKey(构建Key，保存source\_func和target)

2.2.1.2 CachedFunc -> CompileEngineImpl::LowerInternal

2.2.1.2.1 CreateSchedule -> ScheduleGetter::Create

2.2.1.2.1.1 (python)relay.backend.lower\_call -> select\_implementation(选择最优实现)

2.2.1.2.1.2 anchor\_implementation\_.Schedule(构建Schedule)

2.2.1.2.2 tvm::lower -> (python) lower(lower过程)

2.2.2 GraphAddCallNode -> AddNode(记录json形式的节点)

2.3 LoweredOutput ret(填充LoweredOutput信息)

本篇跟读的部分为 : @ tvm/src/relay/backend/build\_module.cc

class RelayBuildModule : public runtime::ModuleNode {

…

void BuildRelay(IRModule relay\_module,

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

…

relay\_module = Optimize(relay\_module, targets\_, params);

…

graph\_codegen\_ = std::unique\_ptr<GraphCodegen>(new GraphCodegen());

graph\_codegen\_->Init(nullptr, targets\_);

graph\_codegen\_->Codegen(func);

ret\_.graph\_json = graph\_codegen\_->GetJSON();

ret\_.params = graph\_codegen\_->GetParams();

auto lowered\_funcs = graph\_codegen\_->GetIRModule();

…

}

…

}

# Step1 初始化GraphCodegen

此处用法比较奇特，GraphCodegen作用基本等于GraphExecutorCodegenModule的一个接口，通过CallFunc调用GraphExecutorCodegenModule对应的函数

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

struct GraphCodegen {

public:

GraphCodegen() {

auto pf = GetPackedFunc("relay.build\_module.\_GraphExecutorCodegen");

mod = (\*pf)();

}

…

protected:

tvm::runtime::Module mod;

template <typename R, typename... Args>

R CallFunc(const std::string& name, Args... args) {

auto pf = mod.GetFunction(name, false);

return pf(std::forward<Args>(args)...);

}

…

}

-> @ tvm/src/relay/backend/graph\_executor\_codegen.cc

virtual PackedFunc GetFunction(const std::string& name, const ObjectPtr<Object>& sptr\_to\_self) {

if (name == "init") {

return PackedFunc([sptr\_to\_self, this](TVMArgs args, TVMRetValue\* rv) {

…

void\* mod = args[0];

Map<Integer, tvm::Target> tmp = args[1];

…

codegen\_ = std::make\_shared<GraphExecutorCodegen>(reinterpret\_cast<runtime::Module\*>(mod),

targets);

});

}

…

}

…

class GraphExecutorCodegen : public backend::MemoizedExprTranslator<std::vector<GraphNodeRef>> {

public:

GraphExecutorCodegen(runtime::Module\* mod, const TargetsMap& targets) : mod\_(mod) {

compile\_engine\_ = CompileEngine::Global();

targets\_ = targets;

}

…

}

-> @ tvm/src/relay/backend/compile\_engine.cc

CompileEngine& CompileEngine::Global() {

…

static CompileEngine\* inst = new CompileEngine(make\_object<CompileEngineImpl>());

return \*inst;

}

Codegen过程实际使用的是CompileEngineImpl的全局实例和targets配合进行代码生成。

# Step2 Codegen

使用Step1 初始化GraphCodegen构建的GraphExecutorCodegenModule继续调用

@ tvm/src/relay/backend/graph\_executor\_codegen.cc

class GraphExecutorCodegenModule : public runtime::ModuleNode {

virtual PackedFunc GetFunction(const std::string& name, const ObjectPtr<Object>& sptr\_to\_self) {

if (name == "init") {

…

} else if (name == "codegen") {

return PackedFunc([sptr\_to\_self, this](TVMArgs args, TVMRetValue\* rv) {

Function func = args[0];

this->output\_ = this->codegen\_->Codegen(func);

});

}

}

…

}

…

class GraphExecutorCodegen : public backend::MemoizedExprTranslator<std::vector<GraphNodeRef>> {

…

LoweredOutput Codegen(relay::Function func) {

auto pf = GetPackedFunc("relay.backend.GraphPlanMemory");

storage\_device\_map\_ = (\*pf)(func);

…

heads\_ = VisitExpr(func->body);

…

LoweredOutput ret;

…

for (auto& kv : lowered\_funcs\_) {

if (ret.lowered\_funcs.count(kv.first) == 0) {

ret.lowered\_funcs.Set(kv.first, IRModule(Map<GlobalVar, BaseFunc>({})));

}

auto& mod = ret.lowered\_funcs[kv.first];

mod->Update(kv.second);

ret.lowered\_funcs.Set(kv.first, mod);

}

ret.external\_mods = compile\_engine\_->LowerExternalFunctions();

return ret;

}

…

}

这一段就是CodeGen的完整流程，其中涉及较多子流程，分步骤说明

## Step2.1 GraphPlanMemory

分配内存，实现在 @ tvm/src/relay/backend/graph\_plan\_memory.cc

class StorageAllocator : public StorageAllocaBaseVisitor {

public:

…

// Run storage allocation for a function.

Map<Expr, Array<IntegerArray> > Plan(const Function& func) {

prototype\_ = StorageAllocaInit(&arena\_).GetInitTokenMap(func);

this->Run(func);

…

for (const auto& kv : token\_map\_) {

std::vector<Integer> storage\_ids;

std::vector<Integer> device\_types;

for (StorageToken\* tok : kv.second) {

if (tok->device\_type) {

num\_annotated\_nodes++;

}

num\_nodes++;

storage\_ids.push\_back(tok->storage\_id);

device\_types.push\_back(tok->device\_type);

}

smap.Set(GetRef<Expr>(kv.first), Array<IntegerArray>({storage\_ids, device\_types}));

}

…

return smap;

}

}

过程依然不少，继续拆解

### Step 2.1.1 StorageAllocaInit::GetInitTokenMap

@ tvm/src/relay/backend/graph\_plan\_memory.cc

class StorageAllocaInit : protected StorageAllocaBaseVisitor {

…

std::unordered\_map<const ExprNode\*, std::vector<StorageToken\*> > GetInitTokenMap(

const Function& func) {

node\_device\_map\_ = CollectDeviceInfo(func);

this->Run(func);

return std::move(token\_map\_);

}

…

}

CollectDeviceInfo调用链为CollectDeviceInfo -> DeviceInfo::GetDeviceMap

@ tvm/src/relay/transforms/device\_annotation.cc

class DeviceInfo {

public:

static Map<Expr, Integer> GetDeviceMap(const Expr& expr) {

DeviceInfo device\_info;

device\_info.post\_visitor\_ = PostDfsOrderVisitor();

device\_info.post\_visitor\_.Visit(expr);

if (device\_info.post\_visitor\_.num\_device\_copy\_ops\_ > 0) {

device\_info.PropagateDeviceId();

return device\_info.device\_map\_;

} else {

return Map<Expr, Integer>();

}

}

…

}

主要就是判断是否存在device\_copy算子，本demo中由于没有存在所以返回空Map。device\_copy算子判断方法为：

@ tvm/src/relay/transforms/device\_annotation.cc

bool IsDeviceCopyNode(const ExprNode\* node) {

if (!node->IsInstance<CallNode>()) return false;

const auto\* call\_node = static\_cast<const CallNode\*>(node);

return call\_node->attrs.as<DeviceCopyAttrs>();

}

是对一个属性的判断，并不是一种具体的算子类型。

CollectDeviceInfo返回空map之后继续执行StorageAllocaInit::Run

@ tvm/src/relay/backend/graph\_plan\_memory.cc

class StorageAllocaBaseVisitor : public ExprVisitor {

…

void Run(const Function& func) {

for (Var param : func->params) {

CreateToken(param.operator->(), false);

}

// must always keep output alive.

for (StorageToken\* tok : GetToken(func->body)) {

tok->ref\_counter += 1;

}

}

…

}

…

class StorageAllocaInit : protected StorageAllocaBaseVisitor {

void CreateToken(const ExprNode\* op, bool can\_realloc) final {

…

if (const auto\* tuple\_type = op->checked\_type().as<TupleTypeNode>()) {

…

} else {

const auto\* ttype = op->checked\_type().as<TensorTypeNode>();

ICHECK(ttype);

StorageToken\* token = arena\_->make<StorageToken>();

token->ttype = ttype;

token->device\_type = device\_type;

tokens.push\_back(token);

}

token\_map\_[op] = tokens;

}

}

基本过程就是遍历记录需要分配内存的数据（此demo中涉及Const节点和Call节点），分配一个token来记录Tensor信息、被消费次数（ref\_counter）、设备信息（device\_type）等。

最终结果是得到prototype\_ 一个记录<节点：存储信息>的map

### Step 2.1.2 StorageAllocator::Run

Run函数和Step 2.1.1 StorageAllocaInit::GetInitTokenMap中的Run函数一样，关键来看CreateToken这个核心函数

@ tvm/src/relay/backend/graph\_plan\_memory.cc

class StorageAllocator : public StorageAllocaBaseVisitor {

void CreateToken(const ExprNode\* op, bool can\_realloc) final {

…

std::vector<StorageToken\*> tokens;

for (StorageToken\* tok : it->second) {

if (can\_realloc) {

tokens.push\_back(Request(tok));

} else {

// Allocate a new token,

StorageToken\* allocated\_tok = Alloc(tok, GetMemorySize(tok));

allocated\_tok->device\_type = tok->device\_type;

// ensure it never get de-allocated.

allocated\_tok->ref\_counter += 1;

tokens.push\_back(allocated\_tok);

}

}

token\_map\_[op] = tokens;

}

…

void VisitExpr\_(const CallNode\* op) final {

std::vector<StorageToken\*> args;

// for each input, visit argument token.

for (Expr arg : op->args) {

for (StorageToken\* tok : GetToken(arg)) {

args.push\_back(tok);

}

}

// create token for the call node.

CreateToken(op, true);

// check if there is orphaned output that can be released immediately.

for (StorageToken\* tok : token\_map\_.at(op)) {

CheckForRelease(tok);

}

for (StorageToken\* tok : args) {

tok->ref\_counter -= 1;

CheckForRelease(tok);

}

}

}

访问顺序和GetInitTokenMap相同，使用GetInitTokenMap中得到的prototype\_对需要分配内存的数据进行分配，如果可以重新分配（can\_realloc）就使用Request（一个搜索函数，用来找合适的使用完的内存）找到一块最合适的被释放的内存进行重新分配，否者直接使用Alloc（malloc的一个抽象）非配一段新的内存。Call节点访问之后要对args的引用计数减一并检查是否需要free。

整个过程可以理解为对runtime过程的模拟，运行到每一个function或者constant的时候需要把数据放到内存中，计算之后结果再存到内存中。为了最大限度的复用内存，对于不再使用的内存不使用eager的方式进行free，而是等模拟到最后看是不是有可能被复用。模拟一遍之后就可以做到最大限度的内存复用。如果早起有魔改过kaldi并且读过其中gpu内存申请代码的人对这个过程应该不陌生。

打印了一下得到的token\_map\_里面的内容，直观感受一下

meta[relay.Constant][0] /\* ty=Tensor[(1000), float32] \*/

has storages :

id 48, size 4000, device\_type 0;

…

meta[relay.Constant][0] /\* ty=Tensor[(100, 512, 10), float32] \*/

has storages :

id 47, size 2048000, device\_type 0;

### Step 2.1.3 信息完善

@ tvm/src/relay/backend/graph\_plan\_memory.cc

class StorageAllocator : public StorageAllocaBaseVisitor {

…

Map<Expr, Array<IntegerArray> > Plan(const Function& func) {

…

for (const auto& kv : token\_map\_) {

std::vector<Integer> storage\_ids;

std::vector<Integer> device\_types;

for (StorageToken\* tok : kv.second) {

if (tok->device\_type) {

num\_annotated\_nodes++;

}

num\_nodes++;

storage\_ids.push\_back(tok->storage\_id);

device\_types.push\_back(tok->device\_type);

}

smap.Set(GetRef<Expr>(kv.first), Array<IntegerArray>({storage\_ids, device\_types}));

}

…

return smap;

}

…

}

逻辑比较简单，就是把Step 2.1.2 StorageAllocator::Run中得到的token\_map\_和节点关联起来。

## Step2.2 Lower

内存非配完毕之后开始进行lower\_func的生成，调用代码为

@ tvm/src/relay/backend/graph\_executor\_codegen.cc

class GraphExecutorCodegen : public backend::MemoizedExprTranslator<std::vector<GraphNodeRef>> {

public:

…

LoweredOutput Codegen(relay::Function func) {

…

for (auto param : func->params) {

auto node\_ptr = GraphInputNode::make\_node\_ptr(param->name\_hint(), GraphAttrs());

var\_map\_[param.get()] = AddNode(node\_ptr, param);

}

heads\_ = VisitExpr(func->body);

…

}

…

std::vector<GraphNodeRef> VisitExpr\_(const CallNode\* op) override {

Expr expr = GetRef<Expr>(op);

Function func;

…

auto pf0 = GetPackedFunc("relay.backend.\_make\_CCacheKey");

auto pf1 = GetPackedFunc("relay.backend.\_CompileEngineLower");

…

CCacheKey key = (\*pf0)(func, target);

CachedFunc lowered\_func = (\*pf1)(compile\_engine\_, key);

…

lowered\_funcs\_[target->str()]->Update(lowered\_func->funcs);

return GraphAddCallNode(op, \_GetUniqueName(lowered\_func->func\_name), lowered\_func->func\_name);

}

…

}

以遍历的方式访问所有节点，除了Call节点其他都是简单的信息记录，Call节点是核心处理逻辑，使用relay.backend.\_make\_CCacheKey生成一个用于存储lower\_func的key，relay.backend.\_CompileEngineLower则用于生成需要的lower\_func。再追一下lower\_func的生成过程

@ tvm/src/relay/backend/compile\_engine.cc

TVM\_REGISTER\_GLOBAL("relay.backend.\_CompileEngineLower")

.set\_body\_typed([](CompileEngine self, CCacheKey key) { return self->Lower(key); });

…

class CompileEngineImpl : public CompileEngineNode {

private:

// implement lowered func

CCacheValue LowerInternal(const CCacheKey& key) {

…

With<Target> target\_scope(key->target);

…

auto cfunc = CreateSchedule(key->source\_func, key->target);

auto cache\_node = make\_object<CachedFuncNode>(\*(cfunc.operator->()));

…

Array<te::Tensor> all\_args = cache\_node->inputs;

for (te::Tensor arg : cache\_node->outputs) {

all\_args.push\_back(arg);

}

// lower the function

if (const auto\* f = runtime::Registry::Get("relay.backend.lower")) {

cache\_node->funcs = (\*f)(cfunc->schedule, all\_args, cache\_node->func\_name, key->source\_func);

}…

value->cached\_func = CachedFunc(cache\_node);

return value;

}

…

}

这里又分了两个主要的过程

### Step 2.2.1 CreateSchedule

最终调用ScheduleGetter::Create

@ tvm/src/relay/backend/compile\_engine.cc

class ScheduleGetter : public backend::MemoizedExprTranslator<Array<te::Tensor>> {

public:

…

CachedFunc Create(const Function& prim\_func) {

auto cache\_node = make\_object<CachedFuncNode>();

cache\_node->target = target\_;

…

cache\_node->outputs = this->VisitExpr(prim\_func->body);

…

te::Schedule schedule;

// No need to register schedule for device copy op.

if (anchor\_attrs\_.as<DeviceCopyAttrs>() == nullptr) {

…

if (!schedule.defined()) {

ICHECK(anchor\_implementation\_.defined());

schedule = anchor\_implementation\_.Schedule(anchor\_attrs\_, tensor\_outs, target\_);

}

…

}

cache\_node->schedule = std::move(schedule);

return CachedFunc(cache\_node);

}

…

Array<te::Tensor> VisitExpr\_(const CallNode\* call\_node) final {

static auto flower\_call = tvm::runtime::Registry::Get("relay.backend.lower\_call");

…

Op op = Downcast<Op>(call\_node->op);

…

if (op == device\_copy\_op\_) {

…

} else {

LoweredOutput lowered\_out = (\*flower\_call)(GetRef<Call>(call\_node), inputs, target\_);

outputs = lowered\_out->outputs;

impl = lowered\_out->implementation;

}

…

return outputs;

}

…

}

发现有分成了两个过程，继续拆解。

### Step 2.2.1.1 get lower implement

对应cache\_node->outputs = this->VisitExpr(prim\_func->body);这一行代码，核心过程依然是遍历，并且单独关照CallNode类型的节点。对CallNode的Lower过程使用relay.backend.lower\_call实现

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

@tvm.\_ffi.register\_func("relay.backend.lower\_call")

def lower\_call(call, inputs, target):

op = call.op

…

if not is\_dyn:

best\_impl, outputs = select\_implementation(op, call.attrs, inputs, ret\_type, target)

else:

…

…

return LoweredOutput(outputs, best\_impl)

最终得到LoweredOutput用来存储outputs和best\_impl。这个过程核心部分逻辑select\_implementation在（贴链接。。。）这一篇解析过，简单来说就是使用compute函数构建计算过程并提取workload，然后根据环境中保存的记录查找最适合workload的配置，两者组合就是最优实现best\_impl。

### Step 2.2.1.1 get schedule

对应schedule = anchor\_implementation\_.Schedule(anchor\_attrs\_, tensor\_outs, target\_);这一行代码。

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

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

const Target& target) {

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

}

终于调用到了schedule函数，此处查看一下source\_func对应的schedule stages：

Primfunc:

fn (%p0: Tensor[(1, 512), float32], %p1: Tensor[(100, 512, 10), float32], %p2: Tensor[(1000), float32], Primitive=1) -> Tensor[(1, 1000), float32] {

%0 = nn.contrib\_dense\_pack(%p0, %p1, units=None, out\_dtype="float32") /\* ty=Tensor[(1, 1000), float32] \*/;

add(%0, %p2) /\* ty=Tensor[(1, 1000), float32] \*/

}

Stages:

0 stage(placeholder, placeholder(placeholder, 0x7f8458b15ea0))

1 stage(placeholder, placeholder(placeholder, 0x7f8458b23380))

2 stage(compute.global, compute(compute.global, body=[reduce(combiner=comm\_reducer(result=[(x + y)], lhs=[x], rhs=[y], identity\_element=[0f]), source=[(placeholder[y.c, k]\*placeholder[floordiv(x.c, 10), k, floormod(x.c, 10)])], init=[], axis=[iter\_var(k, range(min=0, ext=512))], where=(bool)1, value\_index=0)], axis=[iter\_var(y.c, range(min=0, ext=1)), iter\_var(x.c, range(min=0, ext=1000))], reduce\_axis=[iter\_var(k, range(min=0, ext=512))], tag=dense\_pack, attrs={"workload": ["dense\_pack.x86", ["TENSOR", [1, 512], "float32"], ["TENSOR", [100, 512, 10], "float32"], (nullptr), "float32"]}))

3 stage(compute, compute(compute, body=[compute.global[y, x]], axis=[iter\_var(y, range(min=0, ext=1)), iter\_var(x, range(min=0, ext=1000))], reduce\_axis=[], tag=dense\_pack, attrs={"workload": ["dense\_pack.x86", ["TENSOR", [1, 512], "float32"], ["TENSOR", [100, 512, 10], "float32"], (nullptr), "float32"]}))

4 stage(placeholder, placeholder(placeholder, 0x7f84577fb6a0))

5 stage(T\_add, compute(T\_add, body=[(compute[ax0, ax1] + placeholder[ax1])], axis=[iter\_var(ax0, range(min=0, ext=1)), iter\_var(ax1, range(min=0, ext=1000))], reduce\_axis=[], tag=broadcast, attrs={}))

其中contrib\_dense\_pack绑定的compute的schedule是

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

@autotvm.register\_topi\_compute("dense\_pack.x86")

def dense\_pack(cfg, data, weight, bias=None, out\_dtype=None):

…

C = te.compute(

(M, N),

lambda y, x: te.sum(

data[y, k].astype(out\_dtype)

\* packw[idxdiv(x, packw\_bn), k, idxmod(x, packw\_bn)].astype(out\_dtype),

axis=k,

),

tag="dense\_pack",

)

if bias is not None:

C = te.compute((M, N), lambda i, j: C[i, j] + bias[j].astype(out\_dtype), tag=tag.BROADCAST)

return C

…

@autotvm.register\_topi\_schedule("dense\_pack.x86")

def schedule\_dense\_pack(cfg, outs):

"""Create the schedule for dense\_pack"""

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

def \_callback(op):

if "dense\_pack" in op.tag:

\_schedule\_dense\_pack\_template(cfg, s, op.output(0), outs[0])

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

return s

…

def \_schedule\_dense\_pack\_template(cfg, s, C, O):

A, packedB = s[C].op.input\_tensors

CC = s.cache\_write(C, "global")

y, x = s[C].op.axis

(k,) = s[CC].op.reduce\_axis

…

return s

可以发现使用的就是ComputeOp构建Schedule，并配合config使用了基础的构建优化手段，这个过程的梳理参照（链接。。。。。）

### Step 2.2.2 tvm::lower

@ tvm/src/relay/backend/compile\_engine.cc

TVM\_REGISTER\_GLOBAL("relay.backend.\_CompileEngineLower")

.set\_body\_typed([](CompileEngine self, CCacheKey key) { return self->Lower(key); });

…

class CompileEngineImpl : public CompileEngineNode {

private:

// implement lowered func

CCacheValue LowerInternal(const CCacheKey& key) {

…

// lower the function

if (const auto\* f = runtime::Registry::Get("relay.backend.lower")) {

cache\_node->funcs = (\*f)(cfunc->schedule, all\_args, cache\_node->func\_name, key->source\_func);

}…

value->cached\_func = CachedFunc(cache\_node);

return value;

}

…

}

relay.backend.lower 直接调用lower @ tvm/python/tvm/driver/build\_module.py，后面的故事大家就都知道了，参照（链接。。。。。。）。直接看这一步的结果（Stmt结构）：

func fused\_nn\_contrib\_dense\_pack\_add

primfn(placeholder\_3: handle, placeholder\_4: handle, placeholder\_5: handle, T\_add\_1: handle) -> ()

attr = {"global\_symbol": "fused\_nn\_contrib\_dense\_pack\_add", "tir.noalias": True}

buffers = {T\_add: Buffer(T\_add\_2: Pointer(float32), float32, [1, 1000], []),

placeholder\_2: Buffer(placeholder\_6: Pointer(float32), float32, [1000], []),

placeholder\_1: Buffer(placeholder\_7: Pointer(float32), float32, [100, 512, 10], []),

placeholder: Buffer(placeholder\_8: Pointer(float32), float32, [1, 512], [])}

buffer\_map = {placeholder\_3: placeholder, placeholder\_4: placeholder\_1, placeholder\_5: placeholder\_2, T\_add\_1: T\_add} {

for (ax1.outer.ax0.outer.fused: int32, 0, 25) "parallel" {

attr [compute: Pointer(float32x10)] "storage\_scope" = "global";

allocate(compute, float32x10, [4]);

attr [compute.global: Pointer(float32x10)] "storage\_scope" = "global";

allocate(compute.global, float32x10, [1]) {

for (y.inner.outer.x.inner.outer.fused: int32, 0, 4) {

compute.global[ramp(0, 1, 10)] = broadcast(0f32, 10)

for (k.outer: int32, 0, 512) {

compute.global[ramp(0, 1, 10)] = ((float32x10\*)compute.global[ramp(0, 1, 10)] + (broadcast((float32\*)placeholder\_8[k.outer], 10)\*(float32x10\*)placeholder\_7[ramp((((ax1.outer.ax0.outer.fused\*20480) + (y.inner.outer.x.inner.outer.fused\*5120)) + (k.outer\*10)), 1, 10)]))

}

compute[ramp((y.inner.outer.x.inner.outer.fused\*10), 1, 10)] = (float32x10\*)compute.global[ramp(0, 1, 10)]

}

for (ax1.inner.outer: int32, 0, 4) {

T\_add\_2[ramp(((ax1.outer.ax0.outer.fused\*40) + (ax1.inner.outer\*10)), 1, 10)] = ((float32x10\*)compute[ramp((ax1.inner.outer\*10), 1, 10)] + (float32x10\*)placeholder\_6[ramp(((ax1.outer.ax0.outer.fused\*40) + (ax1.inner.outer\*10)), 1, 10)])

}

}

}

}

## Step 2.3 填充LoweredOutput信息

每个Function节点的Stmt表达构建完成之后，剩下的收尾工作就是填充信息，这部分逻辑比较简洁。

@ tvm/src/relay/backend/graph\_executor\_codegen.cc

class GraphExecutorCodegen : public backend::MemoizedExprTranslator<std::vector<GraphNodeRef>> {

…

LoweredOutput Codegen(relay::Function func) {

…

LoweredOutput ret;

…

for (auto& kv : lowered\_funcs\_) {

if (ret.lowered\_funcs.count(kv.first) == 0) {

ret.lowered\_funcs.Set(kv.first, IRModule(Map<GlobalVar, BaseFunc>({})));

}

auto& mod = ret.lowered\_funcs[kv.first];

mod->Update(kv.second);

ret.lowered\_funcs.Set(kv.first, mod);

}

ret.external\_mods = compile\_engine\_->LowerExternalFunctions();

return ret;

}

…

}

主要做了两件事：

1. 填充params信息，每个param的存储信息被保存到ret.params结构中
2. 填充lowered\_funcs信息，前置过程中每个Function节点得到的Stmt结构会被记录到ret.lowered\_funcs中

# 总结

Codegen过程十分繁琐，而且前置知识很多，读起来很有一种分分钟被劝退的感觉，好在这部分应该算是整个tvm流程中最难读的部分，基本上算是最难的一关。

Codegen过程输入是Relay IR，输出是一个LoweredOutput结构，其中核心部分lowered\_funcs为Map<String, IRModule>，每个IRModule中保存了Stmt结构。整个过程大体分成

1. 分配内存：首先查看是否涉及device\_op，此类算子还没遇到过，但个人理解在拆分计算图进行异构部署的时候会遇到；然后对整个计算过程进行模拟，最大化内存复用，最终得到内存最优分配的信息。
2. 遍历整个计算图，对每个节点进行Lower，lower的核心过程依然是compute->schedule->lower，在之前的章节中已经梳理过（链接。。。。）。最终会用CCacheKey保存原始信息，CachedFunc保存Stmt结构。
3. 保存一些debug或者查看用的信息，绑定每个Stmt结构和对应的计算节点。