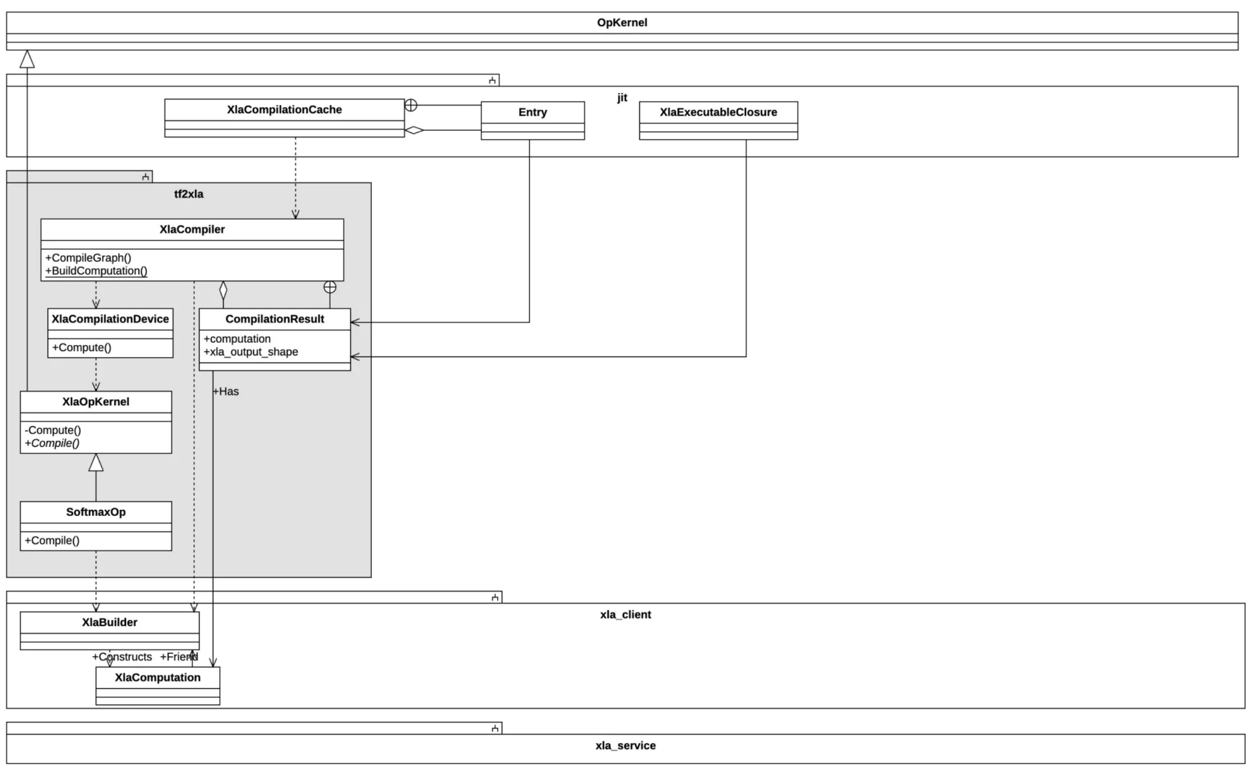
Tensorflow XlaOpKernel | tf2xla 机制详解

compiler/aot/	以AOT的方式将tf2xla/接入TF引擎
compiler/jit/	以JIT的方式将tf2xla/接入TF引擎， 核心是9个优化器和3个tfop，其中XlaCompileOp调用tf2xla的“编译”入口完成功能封装，XlaRunOp调用xla/client完成“运行”功能。
compiler/tf2xla/	对上提供xla_compiler.cc:XlaCompiler::CompileFunction()供jit:compile_fn()使用将cluster转化为XlaComputation。核心是利用xla/client提供的接口，实现对XlaOpKernel的“Symbolic Execution”功能。每个XlaOpKernel子类均做的以下工作: **从XlaOpKernelContext中取出XlaExpression或XlaOp, 调用xla/client/xla_buidler.h提供的方法完成计算, 将计算结果的XlaOp存入XlaKernelContext.**
compiler/xla/client/	对上提供xla_builder.cc:Builder等供CompileFunction()使用，将Graph由Op表达转化为HloModuleProto:HloComputationProto:HloInstructionProto表达并保存在XlaComputation中。 对上提供local_client.cc:LocalClient::Compile()，作为编译入口供jit：BuildExecutable()使用，将已经得到的XlaComputation交给service并进一步编译为二进制。 对上提供local_client.cc:LocalExecutable::Run()，作为运行入口供jit/kernels/xla_ops.cc:XlaRunOp使用，通过Key找到相应的二进制交给service层处理
compiler/xla/service/	对上提供local_service.cc:LocalService::BuildExecutable()供LocalClient::Compile()使用实现真正的编译，承接XlaComputation封装的HloProto, 将其转化为HloModule:HloComputation:HloInstruction表达, 对其进行优化之后, 使用LLVM后端将其编译为相应Executable后端的二进制代码 对上提供executable.cc:Executable::ExecuteOnStream()供LocalExecutable::Run()使用实现真正的执行二进制。

- 

从Kernel的视角, XLA并不会新增Op, 而是针对已有的Op, 新增了基于XLA的另一个版本的Kernel: XlaOpKerne。在TF引擎中, OpKernel在软件栈上已是底层, 即最终的计算过程都要在OpKernel中实现. 但在XLA中, XlaOpKernel只是编译的入口, 大量的实际工作都交给了更下层的XLA引擎去完成.XLA相关的代码在tensorflow/compiler中.

tf2xla/负责XlaOpKernel的构造, 注册. 虽然XLA与TF引擎不在一层, 但二者面临的问题有很多有相似之处, 比如都需要对Kernel和Device保持易扩展性, 都需要维持前驱/后继Kernel的数据流和控制流关系. 基于类似的种种原因, XLA内部实现的注册XlaOpKernel的接口与TF引擎中注册OpKernel的风格十分相似, 同时, 其内部实现又有本质的不同, 而这些”不同”, 正是我们需要关注的.

要理解XlaOpKernel与OpKernel的不同, 关键在于了解”**Symbolic Execution**“.
先来看TF引擎, 它的OpKernel::Compute()方法要: OpKernelContext.Input()取输入数据 ==> 计算 ==> OpKernelContext.SetOutput()存输出数据, 计算结果继续通过OpKernelContext流入后继Opkernel, 其中流动的是真正的训练数据, 暂且将这个过程称之为”Execution”.
对比之下, XLA中的”Symbolic Execution”中的”Symbolic”即是说, XlaOpKernel的设计目的不在于去处理训练数据, 而在于去生成能够正确的处理数据的代码. 整个JIT类似于python解释器，先将程序编译为二进制，再运行二进制，XlaOpKernel执行的”Symbolic Execution”就类似其中的”编译为二进

制”的过程. 具体地, 在XlaOpKernel::Compile()中: XlaOpKernelContext.Input()以XlaOp形式取输入 ==> 调用xla/client/xla_buidler.h提供的方法实现Op该有的功能, 实际上是生成一组能处理数据的HloInstruction ==> XlaOpKernelContext.SetOutput()存储XlaOp形式的结果, 计算结果继续通过XlaOpKernelContext流入后继XlaOpkernel, 其中流动的都是以XlaOp表征的对训练数据的处理方法.

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
//compiler/xla/client/xla_builder.h
// This represents an instruction that has been enqueued using the XlaBuilder.
// This is used to pass to subsequent computations that depends upon the
// instruction as an operand.
class XlaOp {
```

//compiler/xla/client/xla_builder.h // This represents an instruction that has been enqueued using the XlaBuilder. // This is used to pass to subsequent computations that depends upon the // instruction as an operand. class XlaOp {

```
//compiler/xla/client/xla_builder.h
// This represents an instruction that has been enqueued using the XlaBuilder.
// This is used to pass to subsequent computations that depends upon the
// instruction as an operand.
class XlaOp {
```


至于真正处理数据的时机, 就要交给XLA引擎, 它来负责后续的”编译”和”执行”, 具体地, 在JIT中, XlaCompileOp会在所有的XlaOpKernel::Compile()执行完毕之后, 继续调用xla/service中相应的方法将这些所有生成的HloInstruction编译生成二进制并进一步交给XlaRunOp来执行.

XlaOpKernel定义

`"XlaOpKernel(compiler/tf2xla/xla_op_kernel.h)"`继承自`"OpKernel"`(当前XlaOpKernel不支持AsynOpKernel类似的异步机制), 通过这种继承, XlaOpKernel与OpKernel注册框架有天然的相容性, 同时, 又针对XLA的设计要求作了以下处理, 来实现XLA需要的Symbolic Execution: **一个XlaOpKernel子类不再实现以OpKernelContext为参数的Compute()方法, 而要实现以XlaOpKernelContext为参数的Compile()方法.**

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
//compiler/tf2xla/xla_op_kernel.h
class XlaOpKernel : public OpKernel {
// Subclasses should implement Compile(), much as standard OpKernels implement
// Compute().
virtual void Compile(XlaOpKernelContext* context) = 0;
void Compute(OpKernelContext* context) final;
};
//compiler/tf2xla/xla_op_kernel.cc
void XlaOpKernel::Compute(OpKernelContext* context) {
XlaOpKernelContext xla_context(context);
Compile(&xla_context);
}
```

//compiler/tf2xla/xla_op_kernel.h class XlaOpKernel : public OpKernel { // Subclasses should implement Compile(), much as standard OpKernels implement // Compute(). virtual void Compile(XlaOpKernelContext* context) = 0; void Compute(OpKernelContext* context) final; }; //compiler/tf2xla/xla_op_kernel.cc void XlaOpKernel::Compute(OpKernelContext* context) { XlaOpKernelContext xla_context(context); Compile(&xla_context); }

```
//compiler/tf2xla/xla_op_kernel.h
class XlaOpKernel : public OpKernel {
  // Subclasses should implement Compile(), much as standard OpKernels implement
  // Compute().
  virtual void Compile(XlaOpKernelContext* context) = 0;
  void Compute(OpKernelContext* context) final;
};

//compiler/tf2xla/xla_op_kernel.cc
void XlaOpKernel::Compute(OpKernelContext* context) {
  XlaOpKernelContext xla_context(context);
  Compile(&xla_context);
}
```


同时, 依前文所述, XlaOpKernel的运行上下文, 输入输出与OpKernel有很大不同, 的这里, XLA重新封装了一个上下文类: `"XlaOpkernelContext"`, 要注意到, 生成处理数据的XlaOp本身就和数据的形状, 类型等有关系, 即数据的元数据, 这些信息又存储在`"OpKernelContext"`中, 所以使用了”**关联**“OpKernelContext的方式来构造XlaOpKernelContext

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
//compiler/tf2xla/xla_op_kernel.h
// The context passed to the Compile() method of XlaOpKernel. An
// XlaOpKernelContext is a variant of the standard OpKernel class, tailored for
// implementing operators that perform symbolic execution as part of the XLA
// compiler. The key difference is that XlaOpKernelContext produces and consumes
// data as XLA computations, rather than as standard Tensors.
//
// Under the hood, symbolic execution communicates using special Tensors that
// wrap XlaExpression objects, however this is an implementation detail that
// this class hides. The *only* correct way to allocate a Tensor during
// compilation is using the XlaOpKernelContext methods, since they ensure there
// is a valid XlaExpression backing the tensor. No Op should ever call
// allocate_output or allocate_temp directly on the underlying OpKernelContext.
```

class XlaOpKernelContext {
public:
explicit XlaOpKernelContext(OpKernelContext* context);
XlaContext* xla_context() const;
// Returns input `index` as a XlaOp. Unlike
// OpKernelContext::Input returns a symbolic value rather than a concrete
// Tensor.
xla::XlaOp Input(int index);
//compiler/tf2xla/xla_op_kernel.h // The context passed to the Compile() method of XlaOpKernel. An // XlaOpKernelContext is a variant of the standard OpKernel class, tailored for // implementing operators that perform symbolic execution as part of the XLA // compiler. The key difference is that XlaOpKernelContext produces and consumes // data as XLA computations, rather than as standard Tensors. // // Under the hood, symbolic execution communicates using special Tensors that // wrap XlaExpression objects, however this is an implementation detail that // this class hides. The *only* correct way to allocate a Tensor during // compilation is using the XlaOpKernelContext methods, since they ensure there // is a valid XlaExpression backing the tensor. No Op should ever call // allocate_output or allocate_temp directly on the underlying OpKernelContext. class XlaOpKernelContext { public: explicit XlaOpKernelContext(OpKernelContext* context); XlaContext* xla_context() const; // Returns input `index` as a XlaOp. Unlike // OpKernelContext::Input returns a symbolic value rather than a concrete // Tensor. xla::XlaOp Input(int index);

```
//compiler/tf2xla/xla_op_kernel.h
// The context passed to the Compile() method of XlaOpKernel. An
// XlaOpKernelContext is a variant of the standard OpKernel class, tailored for
// implementing operators that perform symbolic execution as part of the XLA
// compiler. The key difference is that XlaOpKernelContext produces and consumes
// data as XLA computations, rather than as standard Tensors.
//
// Under the hood, symbolic execution communicates using special Tensors that
// wrap XlaExpression objects, however this is an implementation detail that
// this class hides. The *only* correct way to allocate a Tensor during
// compilation is using the XlaOpKernelContext methods, since they ensure there
// is a valid XlaExpression backing the tensor. No Op should ever call
// allocate_output or allocate_temp directly on the underlying OpKernelContext.
class XlaOpKernelContext {
 public:
  explicit XlaOpKernelContext(OpKernelContext* context);

  XlaContext* xla_context() const;
  // Returns input `index` as a XlaOp. Unlike
  // OpKernelContext::Input returns a symbolic value rather than a concrete
  // Tensor.
  xla::XlaOp Input(int index);
```

XLA已经实现的OpKernel在 tensorflow/compiler/tf2xla/kernels/中, 实现一个新的XlaOpKernel, 子类需要实现"`Compile()`"方法, 并通过"`REGISTER_XLA_OP`"注册到系统中, 举个例子:
Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter
//compiler/tf2xla/kernels/relu_op.cc
class ReluOp : public XlaOpKernel {
public:
explicit ReluOp(OpKernelConstruction* ctx) : XlaOpKernel(ctx) {}
void Compile(XlaOpKernelContext* ctx) override {
xla::XlaBuilder* builder = ctx->builder();
auto zero = XlaHelpers::Zero(builder, input_type(0));
ctx->SetOutput(0, xla::Max(zero, ctx->Input(0)));
}
};
REGISTER_XLA_OP(Name("Relu"), ReluOp);
//compiler/tf2xla/kernels/relu_op.cc class ReluOp : public XlaOpKernel { public: explicit ReluOp(OpKernelConstruction* ctx) : XlaOpKernel(ctx) {} void Compile(XlaOpKernelContext* ctx) override { xla::XlaBuilder* builder = ctx->builder(); auto zero = XlaHelpers::Zero(builder, input_type(0)); ctx->SetOutput(0, xla::Max(zero, ctx->Input(0))); } }; REGISTER_XLA_OP(Name("Relu"), ReluOp);

```
//compiler/tf2xla/kernels/relu_op.cc
class ReluOp : public XlaOpKernel {
 public:
  explicit ReluOp(OpKernelConstruction* ctx) : XlaOpKernel(ctx) {}
  void Compile(XlaOpKernelContext* ctx) override {
    xla::XlaBuilder* builder = ctx->builder();
    auto zero = XlaHelpers::Zero(builder, input_type(0));
    ctx->SetOutput(0, xla::Max(zero, ctx->Input(0)));
  }
};
REGISTER_XLA_OP(Name("Relu"), ReluOp);
```

XlaOpKernel 注册

和TF引擎中的OpKernel机制类似, XLA内部也使用了**regsitry->registrar->registration->create_fn()**的结构管理旗下的XlaOpKernel.
Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter
//compiler/tf2xla/xla_op_registry.h
#define REGISTER_XLA_OP(NAME, OP) \
REGISTER_XLA_OP_UNIQ_HELPER(__COUNTER__, NAME, OP)
class XlaOpRegistrationBuilder {
public:
// Starts an operator registration chain.
static XlaOpRegistrationBuilder Name(absl::string_view name);

```
...
}
class XlaOpRegistrar {
public:
XlaOpRegistrar(std::unique_ptr<XlaOpRegistry::OpRegistration> registration);
};
#define REGISTER_XLA_OP_UNIQ_HELPER(COUNTER, BUILDER, OP) \
REGISTER_XLA_OP_UNIQ(COUNTER, BUILDER, OP)
#define REGISTER_XLA_OP_UNIQ(CTR, BUILDER, OP) \
static ::tensorflow::XlaOpRegistrar xla_op_registrar__body__##CTR##__object( \
::tensorflow::XlaOpRegistrationBuilder::BUILDER.Build( \
[](::tensorflow::OpKernelConstruction* context) \
-> ::tensorflow::OpKernel* { return new OP(context); }));
//compiler/tf2xla/xla_op_registry.cc
XlaOpRegistry& XlaOpRegistry::Instance() {
static XlaOpRegistry* r = new XlaOpRegistry;
return *r;
}
std::unique_ptr<XlaOpRegistry::OpRegistration> XlaOpRegistrationBuilder::Build(
XlaOpRegistry::Factory factory) {
registration_->factory = factory;
return std::move(registration_);
}
XlaOpRegistrar::XlaOpRegistrar(
std::unique_ptr<XlaOpRegistry::OpRegistration> registration) {
XlaOpRegistry& registry = XlaOpRegistry::Instance();
mutex_lock lock(registry.mutex_);
auto& existing_ops = registry.ops_[registration->name]; // std::unordered_map<string, std::vector<std::unique_ptr<OpRegistration>>> ops_
for (auto& existing : existing_ops) {
if (!XlaOpRegistry::IsCompatible(*existing, *registration)) {
LOG(FATAL)
<< "XLA op registration " << registration->name
<< " is incompatible with existing registration of the same name.";
}
}
existing_ops.emplace_back(std::move(registration)); //将registration注册到registry
}
```

//compiler/tf2xla/xla_op_registry.h #define REGISTER_XLA_OP(NAME, OP) \ REGISTER_XLA_OP_UNIQ_HELPER(__COUNTER__, NAME, OP) class XlaOpRegistrationBuilder { public: // Starts an operator registration chain. static XlaOpRegistrationBuilder Name(absl::string_view name); ... } class XlaOpRegistrar { public: XlaOpRegistrar(std::unique_ptr<XlaOpRegistry::OpRegistration> registration); }; #define REGISTER_XLA_OP_UNIQ_HELPER(COUNTER, BUILDER, OP) \ REGISTER_XLA_OP_UNIQ(COUNTER, BUILDER, OP) #define REGISTER_XLA_OP_UNIQ(CTR, BUILDER, OP) \ static ::tensorflow::XlaOpRegistrar xla_op_registrar__body__##CTR##__object(\ ::tensorflow::XlaOpRegistrationBuilder::BUILDER.Build(\ [](::tensorflow::OpKernelConstruction* context) \ -> ::tensorflow::OpKernel* { return new OP(context); })); //compiler/tf2xla/xla_op_registry.cc XlaOpRegistry& XlaOpRegistry::Instance() { static XlaOpRegistry* r = new XlaOpRegistry; return *r; } std::unique_ptr<XlaOpRegistry::OpRegistration> XlaOpRegistrationBuilder::Build(XlaOpRegistry::Factory factory) { registration_->factory = factory; return std::move(registration_); } XlaOpRegistrar::XlaOpRegistrar(std::unique_ptr<XlaOpRegistry::OpRegistration> registration) { XlaOpRegistry& registry = XlaOpRegistry::Instance(); mutex_lock lock(registry.mutex_); auto& existing_ops = registry.ops_[registration->name]; // std::unordered_map<string, std::vector<std::unique_ptr<OpRegistration>>> ops_ for (auto& existing : existing_ops) { if (!XlaOpRegistry::IsCompatible(*existing, *registration)) { LOG(FATAL) << "XLA op registration " << registration->name << " is incompatible with existing registration of the same name."; } } existing_ops.emplace_back(std::move(registration)); //将registration注册到registry }

```
//compiler/tf2xla/xla_op_registry.h
#define REGISTER_XLA_OP(NAME, OP) \
  REGISTER_XLA_OP_UNIQ_HELPER(__COUNTER__, NAME, OP)
class XlaOpRegistrationBuilder {
 public:
  // Starts an operator registration chain.
  static XlaOpRegistrationBuilder Name(absl::string_view name);
  ...
}
class XlaOpRegistrar {
 public:
  XlaOpRegistrar(std::unique_ptr<XlaOpRegistry::OpRegistration> registration);
};
#define REGISTER_XLA_OP_UNIQ_HELPER(COUNTER, BUILDER, OP) \
  REGISTER_XLA_OP_UNIQ(COUNTER, BUILDER, OP)
#define REGISTER_XLA_OP_UNIQ(CTR, BUILDER, OP)                                 \
  static ::tensorflow::XlaOpRegistrar xla_op_registrar__body__##CTR##__object( \
      ::tensorflow::XlaOpRegistrationBuilder::BUILDER.Build(                   \
          [](::tensorflow::OpKernelConstruction* context)                      \
              -> ::tensorflow::OpKernel* { return new OP(context); }));

//compiler/tf2xla/xla_op_registry.cc
XlaOpRegistry& XlaOpRegistry::Instance() {
  static XlaOpRegistry* r = new XlaOpRegistry;
  return *r;
}
std::unique_ptr<XlaOpRegistry::OpRegistration> XlaOpRegistrationBuilder::Build(
    XlaOpRegistry::Factory factory) {
  registration_->factory = factory;
  return std::move(registration_);
}
XlaOpRegistrar::XlaOpRegistrar(
    std::unique_ptr<XlaOpRegistry::OpRegistration> registration) {
  XlaOpRegistry& registry = XlaOpRegistry::Instance();
  mutex_lock lock(registry.mutex_);
  auto& existing_ops = registry.ops_[registration->name];   //  std::unordered_map<string, std::vector<std::unique_ptr<OpRegistration>>> ops_
  for (auto& existing : existing_ops) {
```

```
    if (!XlaOpRegistry::IsCompatible(*existing, *registration)) {
      LOG(FATAL)
          << "XLA op registration " << registration->name
          << " is incompatible with existing registration of the same name.";
    }
  }
  existing_ops.emplace_back(std::move(registration));  //将registration注册到registry
}
```

如此, 就添加了新的XlaOpKernel(实际上是XlaOpKernelRegistry::OpRegistration)注册到了XlaOpRegistry中, 但事情还远没有结束, 和TF引擎一样, XLA同样需要检索大量的XlaOpKernel, 但XLA无意重新实现一遍TF引擎已经实现过的Regsitration的管理机制, 而为了复用TF的实现, 除了继承OpKernel, 还需要在保留create_fn()的基础上, 将XlaOpKernelRegitry::OpRegistration转换为KernelRegistration, 如此就可以使用TF引擎的OpKernel管理机制. 具体的, 在JIT中, 由Optimization: MarkForCompilationPass中完成这种转换:


```
MarkForCompilationPassImpl::Run() //compiler/jit/mark_for_compilation_pass.cc
RegisterCompilationKernels() //compiler/tf2xla/xla_op_registry.cc
for ops in registry.ops_:
std::vector<std::unique_ptr<OpRegistration>>& op_registrations = ops.second
for op_registration in op_registrations:
for backends in registry::backends_:
std::unique_ptr<KernelDef> kdef(new KernelDef);
kdef->set_op(op_registration->name);
kdef->set_device_type(backend.first);
for type_attr in type_attrs:
...
registry.kernel_registrars_.emplace_back(new kernel_factory::OpKernelRegistrar(new KernelDef(*kdef), "XlaJitOp",op_registration->factory));
```

MarkForCompilationPassImpl::Run() //compiler/jit/mark_for_compilation_pass.cc RegisterCompilationKernels() //compiler/tf2xla/xla_op_registry.cc for ops in registry.ops_: std::vector<std::unique_ptr<OpRegistration>>& op_registrations = ops.second for op_registration in op_registrations: for backends in registry::backends_: std::unique_ptr<KernelDef> kdef(new KernelDef); kdef->set_op(op_registration->name); kdef->set_device_type(backend.first); for type_attr in type_attrs: ... registry.kernel_registrars_.emplace_back(new kernel_factory::OpKernelRegistrar(new KernelDef(*kdef), "XlaJitOp",op_registration->factory));


```
MarkForCompilationPassImpl::Run() //compiler/jit/mark_for_compilation_pass.cc
  RegisterCompilationKernels()    //compiler/tf2xla/xla_op_registry.cc
    for ops in registry.ops_:
      std::vector<std::unique_ptr<OpRegistration>>& op_registrations = ops.second
      for op_registration in op_registrations:
        for backends in registry::backends_:
          std::unique_ptr<KernelDef> kdef(new KernelDef);
          kdef->set_op(op_registration->name);
          kdef->set_device_type(backend.first);
          for type_attr in type_attrs:
            ...
          registry.kernel_registrars_.emplace_back(new kernel_factory::OpKernelRegistrar(new KernelDef(*kdef), "XlaJitOp",op_registration->factory));
```


-9- 将device_type信息存储在KernelDef, XlaOpKernel就是靠device_type的不同才能与"`GlobalKernelRegistry()`"中OpKernel区分开, 在当前版本(1.14)中, XLA共注册3个backend, 所以此处的kdef取"`DEVICE_GPU_XLA_JIT`"等下述3个值之一, 关于"`::tensorflow::XlaBackendRegistrar` "我另文详述, 这里只需了解这些Backend的类型最终将作为XlaOpKernel的kdef.device_type_.


```
//compiler/tf2xla/xla_op_registry.h
REGISTER_XLA_BACKEND(DEVICE_CPU_XLA_JIT, kCpuAllTypes, CpuOpFilter);
REGISTER_XLA_BACKEND(DEVICE_INTERPRETER_XLA_JIT, kExecAllTypes, OpFilter);
REGISTER_XLA_BACKEND(DEVICE_GPU_XLA_JIT, kGpuAllTypes, GpuOpFilter);
```

//compiler/tf2xla/xla_op_registry.h REGISTER_XLA_BACKEND(DEVICE_CPU_XLA_JIT, kCpuAllTypes, CpuOpFilter); REGISTER_XLA_BACKEND(DEVICE_INTERPRETER_XLA_JIT, kExecAllTypes, OpFilter); REGISTER_XLA_BACKEND(DEVICE_GPU_XLA_JIT, kGpuAllTypes, GpuOpFilter);


```
//compiler/tf2xla/xla_op_registry.h
REGISTER_XLA_BACKEND(DEVICE_CPU_XLA_JIT, kCpuAllTypes, CpuOpFilter);
REGISTER_XLA_BACKEND(DEVICE_INTERPRETER_XLA_JIT, kExecAllTypes, OpFilter);
REGISTER_XLA_BACKEND(DEVICE_GPU_XLA_JIT, kGpuAllTypes, GpuOpFilter);
```


-12- 根据*"Tensorflow OpKernel机制详解"*一文, "`kernel_factory::OpKernelRegistrar()`"会调用"`InitInternal()`"将KEY和作为VALUE的"`KernelRegistration`"注册到"`GlobalKernelRegistry()`", 这里, factory即是REGISTER_XLA_OP`时**create_fn(): 一个用于new 一个XlaOpKernel实例的方法**. 至此, 我们注册了的XlaOpKernel就进入到了了GlobalKernelRegistry(), 这些XlaOpKernel可以通过TF引擎的通用接口**"FindKernelRegistration()"**来构造并获取. 此时, 在TF引擎中, 一个Op就有个多个Kernel: OpKernel + 多个适配了不同device的XlaOpKernel, Kernel之间彼此通过device_type进行区分.

在1.14版本中，已经有集成的debug工具查看当前运行时已经注册的OpKernel和XlaOpKernel, 可以看到，即便不考虑OpKernel支持更多硬件设备，仅从Op种类上看，也只有**229/1062**个Op支持XLA，这方面还有很多工作要做。相应的Op明细我列在了文末，感兴趣的同学可以查看。

XlaOpKernel 替换 OpKernel

JIT将所有的XlaOpKernel注册到TF引擎, 那真正运行的时候如何找到相应的XlaOpKernel呢?这就涉及到了刚才一直在强调的问题: 注册到TF引擎时, 每一个OpKernelRegistrar都使用了JIT内部的设备类型. JIT在系统初始化的时候即注册了3个DeviceFactory:

Open code in new window
EnlighterJS 3 Syntax Highlighter

```
//compiler/jit/xla_interpreter_device.cc
REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_INTERPRETER, XlaInterpreterDeviceFactory, 40);
//compiler/jit/xla_cpu_device.cc
REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_CPU, XlaCpuDeviceFactory);
//compiler/jit/xla_gpu_device.cc
REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_GPU, XlaGpuDeviceFactory);
//compiler/jit/xla_device.cc
XlaGpuDeviceFactory::CreateDevices(devices)
for i in gpu_ids:
options.compilation_device_name = DEVICE_GPU_XLA_JIT
device = absl::make_unique<XlaDevice>(session_options, options)
//XlaDevice::XlaDevice():
xla_metadata_(DeviceType(options.compilation_device_name)
device_type_(device_type)
jit_device_name_(options.compilation_device_name),
devices->push_back(std::move(device))
```

//compiler/jit/xla_interpreter_device.cc REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_INTERPRETER, XlaInterpreterDeviceFactory, 40); //compiler/jit/xla_cpu_device.cc REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_CPU, XlaCpuDeviceFactory); //compiler/jit/xla_gpu_device.cc REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_GPU, XlaGpuDeviceFactory); //compiler/jit/xla_device.cc XlaGpuDeviceFactory::CreateDevices(devices) for i in gpu_ids: options.compilation_device_name = DEVICE_GPU_XLA_JIT device = absl::make_unique<XlaDevice>(session_options, options) //XlaDevice::XlaDevice(): xla_metadata_(DeviceType(options.compilation_device_name) device_type_(device_type) jit_device_name_(options.compilation_device_name), devices->push_back(std::move(device))


```
//compiler/jit/xla_interpreter_device.cc
REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_INTERPRETER, XlaInterpreterDeviceFactory, 40);
//compiler/jit/xla_cpu_device.cc
REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_CPU, XlaCpuDeviceFactory);
//compiler/jit/xla_gpu_device.cc
REGISTER_LOCAL_DEVICE_FACTORY(DEVICE_XLA_GPU, XlaGpuDeviceFactory);

//compiler/jit/xla_device.cc
XlaGpuDeviceFactory::CreateDevices(devices)
  for i in gpu_ids:
    options.compilation_device_name = DEVICE_GPU_XLA_JIT
    device = absl::make_unique<XlaDevice>(session_options, options)
      //XlaDevice::XlaDevice():
      xla_metadata_(DeviceType(options.compilation_device_name)
        device_type_(device_type)
      jit_device_name_(options.compilation_device_name),
    devices->push_back(std::move(device))
```


根据Tensorflow的设计, 所有注册的DeviceFactory最终都会被Tensorflow执行引擎在初始化阶段调用"`CreateDevices()`"以**工厂模式”生产”**相应的device实例, JIT注册的这三个也不例外, 以JIT内的GPU为例, 可以看到, 每一个device.device_type_都是"`DEVICE_GPU_XLA_JIT`".在OpKernel执行阶段, XlaCompileOp执行编译的时候结合上述的DEVICE_GPU_XLA_JIT等属性, 从GlobalKernelRegistry()中检索所需的Kernel;

Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter

```
tensorflow::XlaCompileOp::Compute()
tensorflow::CompileToLocalExecutable()
BuildCompilationCache(XlaPlatformInfo& platform_info, {
*cache = new XlaCompilationCache(platform_info.xla_device_metadata()->jit_device_type());
//XlaCompilationCache()
device_type_(std::move(device_type))
//compiler/jit/xla_device.cc XlaDevice::Metadata::jit_device_type()
return device_type_;
XlaCompiler::Options options
options.device_type = cache->device_type();
//XlaCompilationCache::Compile
return cache->Compile(options)
XlaCompilationCache::CompileImpl(options
XlaCompiler compiler(options);
//XlaCompiler::XlaCompiler(XlaCompiler::Options options):
options_(options),
device_(new XlaCompilationDevice(SessionOptions(), options_.device_type))
LocalDevice(options, Device::BuildDeviceAttributes(absl::StrCat("/device:",type.type()...
device_mgr_(absl::WrapUnique(device_))
local_pflr_.reset(new ProcessFunctionLibraryRuntime(&device_mgr_
for d in device_mgr->ListDevices():
flr_map_[d] = NewFunctionLibraryRuntime()
return std::unique_ptr<FunctionLibraryRuntime>(new FunctionLibraryRuntimeImpl(device_mgr, env, device
pflr_.reset(new ProcessFunctionLibraryRuntime(&device_mgr_,)
local_flib_runtime_ = local_pflr_->GetFLR(device_->name())
flib_runtime_ = pflr_->GetFLR(device_->name());
tensorflow::XlaCompiler::CompileFunction()
tensorflow::XlaCompiler::CompileGraph()
xla::XlaBuilder builder(name);
XlaContext* context = new XlaContext(this, &builder)
ExecuteGraph(context, std::move(graph), device_, flib_runtime_)
device->resource_manager()->Create(
GraphCompiler graph_compiler(device, graph.get(), flib, step_container.get());
//tensorflow::GraphCompiler::Compile()
graph_compiler.Compile()
```

```
for (Node* n : topo_sorted_nodes):
//core/common_runtime/function.cc tensorflow::FunctionLibraryRuntimeImpl::CreateKernel()
flib_->CreateKernel(n->def(), &op_kernel_raw);
tensorflow::CreateNonCachedKernel(device_)
device_type = DeviceType(device->attributes().device_type());
//core/framework/op_kernel.cc
tensorflow::CreateOpKernel(device_type)
Status s = OpRegistry::Global()->LookUpOpDef(node_def.op(),&op_def);
FindKernelRegistration(device_type)
FindKernelRegistration(device_type)
string key = Key(node_op, device_type, label);
KernelRegistry* typed_registry = GlobalKernelRegistryTyped();
auto regs = typed_registry->registry.equal_range(key)
for iter in regs:
KernelAttrsMatch(iter->second.def, node_attrs, &match)
*reg = &iter->second
// Everything needed for OpKernel construction.
OpKernelConstruction context(...)
//tensorflow::XlaOpKernel::XlaOpKernel()
*kernel = registration->factory->Create(&context);
std::unique_ptr<OpKernel> op_kernel(op_kernel_raw);
OpKernelContext op_context()
*flib_->GetFunctionLibraryDefinition(), *n)
device_->Compute(CHECK_NOTNULL(params.op_kernel), &op_context)
op_kernel->Compute(context)
BuildComputation()
entry->compilation_status = compile_fn(
entry->compilation_status = BuildExecutable(
*out_compilation_result = &entry->compilation_result;
*out_executable = entry->executable.get()
```

tensorflow::XlaCompileOp::Compute() tensorflow::CompileToLocalExecutable() BuildCompilationCache(XlaPlatformInfo& platform_info, { *cache = new XlaCompilationCache(platform_info.xla_device_metadata()->jit_device_type()); //XlaCompilationCache() device_type_(std::move(device_type)) //compiler/jit/xla_device.cc XlaDevice::Metadata::jit_device_type() return device_type_; XlaCompiler::Options options options.device_type = cache->device_type(); //XlaCompilationCache::Compile return cache->Compile(options) XlaCompilationCache::CompileImpl(options XlaCompiler compiler(options); //XlaCompiler::XlaCompiler(XlaCompiler::Options options): options_(options), device_(new XlaCompilationDevice(SessionOptions(), options_.device_type)) LocalDevice(options, Device::BuildDeviceAttributes(absl::StrCat("/device:",type.type()... device_mgr_(absl::WrapUnique(device_)) local_pflr_.reset(new ProcessFunctionLibraryRuntime(&device_mgr_ for d in device_mgr->ListDevices(): flr_map_[d] = NewFunctionLibraryRuntime() return std::unique_ptr<FunctionLibraryRuntime>(new FunctionLibraryRuntimeImpl(device_mgr, env, device pflr_.reset(new ProcessFunctionLibraryRuntime(&device_mgr_,) local_flib_runtime_ = local_pflr_->GetFLR(device_->name()) flib_runtime_ = pflr_->GetFLR(device_->name()); tensorflow::XlaCompiler::CompileFunction() tensorflow::XlaCompiler::CompileGraph() xla::XlaBuilder builder(name); XlaContext* context = new XlaContext(this, &builder) ExecuteGraph(context, std::move(graph), device_, flib_runtime_) device->resource_manager()->Create(GraphCompiler graph_compiler(device, graph.get(), flib, step_container.get()); //tensorflow::GraphCompiler::Compile() graph_compiler.Compile() for (Node* n : topo_sorted_nodes): //core/common_runtime/function.cc tensorflow::FunctionLibraryRuntimeImpl::CreateKernel() flib_->CreateKernel(n->def(), &op_kernel_raw); tensorflow::CreateNonCachedKernel(device_) device_type = DeviceType(device->attributes().device_type()); //core/framework/op_kernel.cc tensorflow::CreateOpKernel(device_type) Status s = OpRegistry::Global()->LookUpOpDef(node_def.op(),&op_def); FindKernelRegistration(device_type) FindKernelRegistration(device_type) string key = Key(node_op, device_type, label); KernelRegistry* typed_registry = GlobalKernelRegistryTyped(); auto regs = typed_registry->registry.equal_range(key) for iter in regs: KernelAttrsMatch(iter->second.def, node_attrs, &match) *reg = &iter->second // Everything needed for OpKernel construction. OpKernelConstruction context(...) //tensorflow::XlaOpKernel::XlaOpKernel() *kernel = registration->factory->Create(&context); std::unique_ptr<OpKernel> op_kernel(op_kernel_raw); OpKernelContext op_context() *flib_->GetFunctionLibraryDefinition(), *n) device_->Compute(CHECK_NOTNULL(params.op_kernel), &op_context) op_kernel->Compute(context) BuildComputation() entry->compilation_status = compile_fn(entry->compilation_status = BuildExecutable(*out_compilation_result = &entry->compilation_result; *out_executable = entry->executable.get()

```
tensorflow::XlaCompileOp::Compute()
  tensorflow::CompileToLocalExecutable()
      BuildCompilationCache(XlaPlatformInfo& platform_info, {
      *cache = new XlaCompilationCache(platform_info.xla_device_metadata()->jit_device_type());
        //XlaCompilationCache()
        device_type_(std::move(device_type))
        //compiler/jit/xla_device.cc XlaDevice::Metadata::jit_device_type()
        return device_type_;
    XlaCompiler::Options options
    options.device_type = cache->device_type();
    //XlaCompilationCache::Compile
    return  cache->Compile(options)
      XlaCompilationCache::CompileImpl(options
        XlaCompiler compiler(options);
        //XlaCompiler::XlaCompiler(XlaCompiler::Options options):
          options_(options),
          device_(new XlaCompilationDevice(SessionOptions(), options_.device_type))
            LocalDevice(options, Device::BuildDeviceAttributes(absl::StrCat("/device:",type.type()...
          device_mgr_(absl::WrapUnique(device_))
          local_pflr_.reset(new ProcessFunctionLibraryRuntime(&device_mgr_
            for d in device_mgr->ListDevices():
              flr_map_[d] = NewFunctionLibraryRuntime()
                return std::unique_ptr<FunctionLibraryRuntime>(new FunctionLibraryRuntimeImpl(device_mgr, env, device
          pflr_.reset(new ProcessFunctionLibraryRuntime(&device_mgr_,)
          local_flib_runtime_ = local_pflr_->GetFLR(device_->name())
          flib_runtime_ = pflr_->GetFLR(device_->name());
        tensorflow::XlaCompiler::CompileFunction()
          tensorflow::XlaCompiler::CompileGraph()
            xla::XlaBuilder builder(name);
            XlaContext* context = new XlaContext(this, &builder)
            ExecuteGraph(context, std::move(graph), device_, flib_runtime_)
              device->resource_manager()->Create(
              GraphCompiler graph_compiler(device, graph.get(), flib, step_container.get());
              //tensorflow::GraphCompiler::Compile()
              graph_compiler.Compile()
                for (Node* n : topo_sorted_nodes):
```

```
          //core/common_runtime/function.cc tensorflow::FunctionLibraryRuntimeImpl::CreateKernel()
          flib_->CreateKernel(n->def(), &op_kernel_raw);
            tensorflow::CreateNonCachedKernel(device_)
              device_type = DeviceType(device->attributes().device_type());
              //core/framework/op_kernel.cc
              tensorflow::CreateOpKernel(device_type)
                Status s = OpRegistry::Global()->LookUpOpDef(node_def.op(),&op_def);
                FindKernelRegistration(device_type)
                  FindKernelRegistration(device_type)
                    string key = Key(node_op, device_type, label);
                    KernelRegistry* typed_registry = GlobalKernelRegistryTyped();
                    auto regs = typed_registry->registry.equal_range(key)
                    for iter in regs:
                      KernelAttrsMatch(iter->second.def, node_attrs, &match)
                      *reg = &iter->second
                // Everything needed for OpKernel construction.
                OpKernelConstruction context(...)
                //tensorflow::XlaOpKernel::XlaOpKernel()
                *kernel = registration->factory->Create(&context);
          std::unique_ptr<OpKernel> op_kernel(op_kernel_raw);
          OpKernelContext op_context()
          *flib_->GetFunctionLibraryDefinition(), *n)
          device_->Compute(CHECK_NOTNULL(params.op_kernel), &op_context)
            op_kernel->Compute(context)
    BuildComputation()
entry->compilation_status = compile_fn(
entry->compilation_status = BuildExecutable(
*out_compilation_result = &entry->compilation_result;
*out_executable = entry->executable.get()
```

-1- JIT编译执行入口, 初始化结束后, Tensorflow执行引擎执行XlaCompileOp进而进入"编译"阶段
-4-8- 将device中获取到的DEVICE_GPU_XLA_JIT 存入XlaCompilationCache
-10- 用XlaCompilationCache中的device_type构造XlaCompiler::Options, 进一步用于构造XlaCompiler
-18- 构造XlaCompiler::device_对象, 可以看到, 构造DeviceAttr使用的device_type_就是刚才传入的DEVICE_GPU_XLA_JIT
-19,21,23- 将存有DEVICE_GPU_XLA_JIT的Device laCompiler::device_存入XlaCompiler::flib_runtime_
-33- 构造GraphCompiler, 传入的flib即XlaCompiler::flib_runtime_, 存入GraphCompiler::flib_
-39- 此处使用的device追根溯源, 就是系统初始化CreateDevices()时构造的device, 即 DEVICE_GPU_XLA_JIT
-18,40- 遥相呼应
-46- 根据DEVICE_GPU_XLA_JIT构造检索用的key
-54,52- 执行create_fn(), 在XLA中即是构造XlaOpKernel.
-59- 执行OpKernel的入口
-61- 这里就是XlaOpKernel::Compute(), 如前文所述, 实质就是执行Compile()

另外, 关于使用device区别不同Kernel的讨论, 可以看到, TF引擎为OpKernel只提供了三种设备:`"DEVICE_CPU"`, `"DEVICE_GPU"`和`"DEVICE_SYCL"` (`core/framework/types.h`), 可见, JIT通过构造虚拟设备的方式将两类Kernel巧妙的融合在一个数据结构中, 设计可谓巧妙. 类似的很多Linux内核程序也是借助了设备驱动接口注册虚拟设备实现和用户态的交互, 优秀的代码设计大体相似, 但垃圾的代码却各有各的垃圾法.
Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter
//core/common_runtime/executor.cc
for iter in graphs:
Device* device;
TF_RETURN_IF_ERROR(device_mgr_->LookupDevice(partition_name, &device));
//core/common_runtime/executor.cc for iter in graphs: Device* device; TF_RETURN_IF_ERROR(device_mgr_->LookupDevice(partition_name, &device));

```
//core/common_runtime/executor.cc
for iter in graphs:
  Device* device;
  TF_RETURN_IF_ERROR(device_mgr_->LookupDevice(partition_name, &device));
```


XlaOpKernel 调试

和KernelRegistry一样, XlaOpRegistry也没有提供很多调试接口, 目前的版本只有这一个, 毕竟, 既然是开发人员, 就不能奢求太多
Plain text
Copy to clipboard
Open code in new window
EnlighterJS 3 Syntax Highlighter
//tf2xla/xla_registry.h
// Returns all operations for which there are XLA kernels on any device.
static std::vector<string> GetAllRegisteredOps();
//tf2xla/xla_registry.h // Returns all operations for which there are XLA kernels on any device. static std::vector<string> GetAllRegisteredOps();

```
//tf2xla/xla_registry.h
  // Returns all operations for which there are XLA kernels on any device.
  static std::vector<string> GetAllRegisteredOps();
```


XlaOpKernel VS OpKernel

NR	Op supported by XLA	Op not supported by XLA	
1	_ArrayToList	__MklDummyConv2DBackpropFilterWithBias	DatasetToGraph

2	_ListToArray	__MklDummyConv2DWithBias	DatasetToSingleElement
3	AddN	__MklDummyPadWithConv2D	DebugGradientIdentity
4	AdjustContrastv2	__MklDummyPadWithFusedConv2D	DebugGradientRefIdentity
5	AdjustHue	_FusedConv2D	DebugIdentity
6	AdjustSaturation	_FusedMatMul	DebugNanCount
7	All	_HostCast	DebugNumericSummary
8	Any	_HostRecv	DecodeAndCropJpeg
9	ApproximateEqual	_HostSend	DecodeBase64
10	ArgMax	_If	DecodeBmp
11	ArgMin	_MklAddN	DecodeCompressed
12	Assert	_MklAvgPool	DecodeCSV
13	AssignVariableOp	_MklAvgPool3D	DecodeGif
14	AvgPool	_MklAvgPool3DGrad	DecodeJpeg
15	AvgPool3D	_MklAvgPoolGrad	DecodeJSONExample
16	BatchMatMul	_MklConcat	DecodePaddedRaw
17	BatchMatMulV2	_MklConcatV2	DecodePng
18	BatchToSpace	_MklConv2D	DecodeProtoV2
19	BatchToSpaceND	_MklConv2DBackpropFilter	DecodeRaw
20	BiasAdd	_MklConv2DBackpropFilterWithBias	DecodeWav
21	BiasAddGrad	_MklConv2DBackpropInput	DeepCopy
22	BiasAddV1	_MklConv2DWithBias	DeleteIterator
23	BroadcastArgs	_MklConv2DWithBiasBackpropBias	DeleteSessionTensor
24	BroadcastGradientArgs	_MklConv3D	DenseToDenseSetOperation
25	BroadcastTo	_MklConv3DBackpropFilterV2	DenseToSparseSetOperation
26	Bucketize	_MklConv3DBackpropInputV2	Dequantize
27	Case	_MklDepthwiseConv2dNative	DeserializeIterator
28	Cast	_MklDepthwiseConv2dNativeBackpropFilter	DeserializeManySparse
29	CheckNumerics	_MklDepthwiseConv2dNativeBackpropInput	DeserializeSparse
30	ClipByValue	_MklDequantize	DestroyResourceOp
31	Concat	_MklElu	DestroyTemporaryVariable
32	ConcatOffset	_MklEluGrad	Dilation2D
33	ConcatV2	_MklFusedBatchNorm	Dilation2DBackpropFilter
34	ConjugateTranspose	_MklFusedBatchNormGrad	Dilation2DBackpropInput
35	Const	_MklFusedBatchNormGradV2	Div

36	ControlTrigger	_MklFusedBatchNormV2	DrawBoundingBoxes
37	Conv2D	_MklFusedConv2D	DrawBoundingBoxesV2
38	Conv3D	_MklIdentity	DynamicPartition
39	Cross	_MklInputConversion	EditDistance
40	Cumprod	_MklLeakyRelu	EncodeBase64
41	Cumsum	_MklLeakyReluGrad	EncodeJpeg
42	DepthToSpace	_MklLRN	EncodeJpegVariableQuality
43	DepthwiseConv2dNative	_MklLRNGrad	EncodePng
44	DepthwiseConv2dNativeBackpropFilter	_MklMaxPool	EncodeProto
45	DepthwiseConv2dNativeBackpropInput	_MklMaxPool3D	EncodeWav
46	Diag	_MklMaxPool3DGrad	EnsureShape
47	DiagPart	_MklMaxPoolGrad	Enter
48	DynamicStitch	_MklPadWithConv2D	Equal
49	Elu	_MklPadWithFusedConv2D	EuclideanNorm
50	EluGrad	_MklQuantizedAvgPool	Exit
51	Empty	_MklQuantizedConcatV2	ExperimentalAssertNextDataset
52	EmptyTensorList	_MklQuantizedConv2D	ExperimentalAutoShardDataset
53	ExpandDims	_MklQuantizedConv2DAndRelu	ExperimentalBytesProducedStatsDataset
54	ExtractImagePatches	_MklQuantizedConv2DAndReluAndRequantize	ExperimentalChooseFastestDataset
55	FakeParam	_MklQuantizedConv2DAndRequantize	ExperimentalCSVDataset
56	FakeQuantWithMinMaxArgs	_MklQuantizedConv2DPerChannel	ExperimentalDatasetCardinality
57	FakeQuantWithMinMaxArgsGradient	_MklQuantizedConv2DWithBias	ExperimentalDatasetToTFRecord
58	FakeQuantWithMinMaxVars	_MklQuantizedConv2DWithBiasAndRelu	ExperimentalDenseToSparseBatchDatase
59	FakeQuantWithMinMaxVarsGradient	_MklQuantizedConv2DWithBiasAndReluAndRequantize	ExperimentalDirectedInterleaveDataset
60	FFT	_MklQuantizedConv2DWithBiasAndRequantize	ExperimentalGroupByReducerDataset
61	FFT2D	_MklQuantizedConv2DWithBiasSignedSumAndReluAndRequantize	ExperimentalGroupByWindowDataset
62	FFT3D	_MklQuantizedConv2DWithBiasSumAndRelu	ExperimentalIgnoreErrorsDataset
63	Fill	_MklQuantizedConv2DWithBiasSumAndReluAndRequantize	ExperimentalIteratorGetDevice
64	FusedBatchNorm	_MklQuantizedDepthwiseConv2D	ExperimentalLatencyStatsDataset
65	FusedBatchNormGrad	_MklQuantizedDepthwiseConv2DWithBias	ExperimentalLMDBDataset
66	FusedBatchNormGradV2	_MklQuantizedDepthwiseConv2DWithBiasAndRelu	ExperimentalMapAndBatchDataset
67	FusedBatchNormGradV3	_MklQuantizedDepthwiseConv2DWithBiasAndReluAndRequantize	ExperimentalMapDataset
68	FusedBatchNormV2	_MklQuantizedMaxPool	ExperimentalMatchingFilesDataset
69	FusedBatchNormV3	_MklQuantizeV2	ExperimentalMaxIntraOpParallelismData

70	Gather	_MklRelu	ExperimentalNonSerializableDataset
71	GatherNd	_MklRelu6	ExperimentalParallelInterleaveDataset
72	GatherV2	_MklRelu6Grad	ExperimentalParseExampleDataset
73	HSVToRGB	_MklReluGrad	ExperimentalPrivateThreadPoolDataset
74	IdentityN	_MklReshape	ExperimentalRandomDataset
75	If	_MklSlice	ExperimentalRebatchDataset
76	IFFT	_MklSoftmax	ExperimentalScanDataset
77	IFFT2D	_MklTanh	ExperimentalSetStatsAggregatorDataset
78	IFFT3D	_MklTanhGrad	ExperimentalSleepDataset
79	InTopKV2	_MklToTf	ExperimentalSlidingWindowDataset
80	InvertPermutation	_NcclBroadcastRecv	ExperimentalSqlDataset
81	IRFFT	_NcclBroadcastSend	ExperimentalStatsAggregatorHandle
82	IRFFT2D	_NcclReduceRecv	ExperimentalStatsAggregatorSummary
83	IRFFT3D	_NcclReduceSend	ExperimentalTakeWhileDataset
84	L2Loss	_ParallelConcatStart	ExperimentalThreadPoolDataset
85	LeakyRelu	_ParallelConcatUpdate	ExperimentalThreadPoolHandle
86	LeakyReluGrad	_ReadVariablesOp	ExperimentalUnbatchDataset
87	LinSpace	_Recv	ExperimentalUniqueDataset
88	ListDiff	_ScopedAllocator	ExtractGlimpse
89	LogSoftmax	_ScopedAllocatorConcat	ExtractJpegShape
90	LRN	_ScopedAllocatorSplit	ExtractVolumePatches
91	LRNGrad	_Send	Fact
92	MatMul	_UnaryOpsComposition	FakeQuantWithMinMaxVarsPerChannel
93	MatrixBandPart	_VarHandlesOp	FakeQuantWithMinMaxVarsPerChannel(
94	MatrixDiag	_While	FakeQueue
95	MatrixDiagPart	; group	FIFOQueue
96	MatrixSetDiag	; idx:	FIFOQueueV2
97	Max	Abort	FilterByLastComponentDataset
98	MaxPool	Abs	FilterDataset
99	MaxPool3D	AccumulatorApplyGradient	Fingerprint
100	MaxPool3DGrad	AccumulatorNumAccumulated	FixedLengthRecordDataset
101	MaxPool3DGradGrad	AccumulatorSetGlobalStep	FixedLengthRecordDatasetV2
102	MaxPoolGrad	AccumulatorTakeGradient	FixedLengthRecordReader
103	MaxPoolGradGrad	Add	FixedLengthRecordReaderV2

104	MaxPoolGradGradV2	AddManySparseToTensorsMap	FixedUnigramCandidateSampler
105	MaxPoolGradV2	AddSparseToTensorsMap	FlatMapDataset
106	MaxPoolV2	AddV2	FloorDiv
107	Mean	AdjustContrast	FloorMod
108	Min	AllCandidateSampler	FlushSummaryWriter
109	MirrorPad	Angle	For
110	Multinomial	AnonymousIterator	FractionalAvgPool
111	NoOp	AnonymousIteratorV2	FractionalAvgPoolGrad
112	OneHot	ApplyAdadelta	FractionalMaxPool
113	OnesLike	ApplyAdagrad	FractionalMaxPoolGrad
114	Pack	ApplyAdagradDA	FusedPadConv2D
115	Pad	ApplyAdam	FusedResizeAndPadConv2D
116	PadV2	ApplyAdaMax	GenerateVocabRemapping
117	PartitionedCall	ApplyAddSign	GeneratorDataset
118	PlaceholderWithDefault	ApplyCenteredRMSProp	GetSessionHandle
119	PreventGradient	ApplyFtrl	GetSessionHandleV2
120	Prod	ApplyFtrlV2	GetSessionTensor
121	Qr	ApplyGradientDescent	Greater
122	QuantizeAndDequantizeV2	ApplyMomentum	GreaterEqual
123	QuantizeAndDequantizeV3	ApplyPowerSign	GuaranteeConst
124	RandomShuffle	ApplyProximalAdagrad	HashTable
125	RandomStandardNormal	ApplyProximalGradientDescent	HashTableV2
126	RandomUniform	ApplyRMSProp	HistogramFixedWidth
127	RandomUniformInt	Assign	HistogramSummary
128	Range	AssignAdd	HostConst
129	Rank	AssignAddVariableOp	Identity
130	ReadVariableOp	AssignSub	IdentityReader
131	Relu	AssignSubVariableOp	IdentityReaderV2
132	Relu6	AsString	Imag
133	Relu6Grad	AudioSpectrogram	ImageSummary
134	ReluGrad	AudioSummary	ImmutableConst
135	Reshape	AudioSummaryV2	ImportEvent
136	ResizeBilinear	AvgPool3DGrad	InitializeTable
137	ResizeBilinearGrad	AvgPoolGrad	InitializeTableFromTextFile

138	ResizeNearestNeighbor	Barrier	InitializeTableFromTextFileV2
139	ResourceApplyAdadelta	BarrierClose	InitializeTableV2
140	ResourceApplyAdagrad	BarrierIncompleteSize	InplaceAdd
141	ResourceApplyAdagradDA	BarrierInsertMany	InplaceSub
142	ResourceApplyAdam	BarrierReadySize	InplaceUpdate
143	ResourceApplyAdaMax	BarrierTakeMany	InterleaveDataset
144	ResourceApplyAddSign	Batch	InTopK
145	ResourceApplyFtrl	BatchDataset	IsBoostedTreesEnsembleInitialized
146	ResourceApplyFtrlV2	BatchDatasetV2	IsBoostedTreesQuantileStreamResourceI
147	ResourceApplyMomentum	BatchFFT	IsVariableInitialized
148	ResourceApplyPowerSign	BatchFFT2D	Iterator
149	ResourceApplyProximalGradientDescent	BatchFFT3D	IteratorFromStringHandle
150	ResourceApplyRMSProp	BatchFunction	IteratorFromStringHandleV2
151	ResourceGather	BatchIFFT	IteratorGetNext
152	ResourceScatterUpdate	BatchIFFT2D	IteratorGetNextAsOptional
153	ResourceStridedSliceAssign	BatchIFFT3D	IteratorGetNextSync
154	Reverse	BatchMatrixBandPart	IteratorToStringHandle
155	ReverseSequence	BatchMatrixDiag	IteratorV2
156	ReverseV2	BatchMatrixDiagPart	KMC2ChainInitialization
157	RFFT	BatchMatrixSetDiag	KmeansPlusPlusInitialization
158	RFFT2D	BatchNormWithGlobalNormalization	LearnedUnigramCandidateSampler
159	RFFT3D	BatchNormWithGlobalNormalizationGrad	LeftShift
160	RGBToHSV	Betainc	Less
161	Select	Bincount	LessEqual
162	SelectV2	BitwiseAnd	LMDBReader
163	SelfAdjointEigV2	BitwiseOr	LoadAndRemapMatrix
164	Selu	BitwiseXor	LogicalAnd
165	SeluGrad	BoostedTreesAggregateStats	LogicalNot
166	Shape	BoostedTreesBucketize	LogicalOr
167	ShapeN	BoostedTreesCalculateBestFeatureSplit	LogUniformCandidateSampler
168	Size	BoostedTreesCalculateBestGainsPerFeature	LookupTableExport
169	Slice	BoostedTreesCenterBias	LookupTableExportV2
170	Snapshot	BoostedTreesCreateEnsemble	LookupTableFind
171	Softmax	BoostedTreesCreateQuantileStreamResource	LookupTableFindV2

172	SoftmaxCrossEntropyWithLogits	BoostedTreesDeserializeEnsemble	LookupTableImport
173	SpaceToBatch	BoostedTreesExampleDebugOutputs	LookupTableImportV2
174	SpaceToBatchND	BoostedTreesGetEnsembleStates	LookupTableInsert
175	SpaceToDepth	BoostedTreesMakeQuantileSummaries	LookupTableInsertV2
176	SparseMatMul	BoostedTreesMakeStatsSummary	LookupTableRemoveV2
177	SparseSoftmaxCrossEntropyWithLogits	BoostedTreesPredict	LookupTableSize
178	SparseToDense	BoostedTreesQuantileStreamResourceAddSummaries	LookupTableSizeV2
179	Split	BoostedTreesQuantileStreamResourceDeserialize	LoopCond
180	SplitV	BoostedTreesQuantileStreamResourceFlush	LowerBound
181	Squeeze	BoostedTreesQuantileStreamResourceGetBucketBoundaries	Lu
182	StackCloseV2	BoostedTreesSerializeEnsemble	MakeIterator
183	StackPopV2	BoostedTreesTrainingPredict	MapClear
184	StackPushV2	BoostedTreesUpdateEnsemble	MapDataset
185	StatefulPartitionedCall	CacheDataset	MapDefun
186	StatefulStandardNormalV2	ChooseFastestBranchDataset	MapIncompleteSize
187	StatefulTruncatedNormal	CloseSummaryWriter	MapPeek
188	StatefulUniform	CollectiveBcastRecv	MapSize
189	StatefulUniformFullInt	CollectiveBcastSend	MapStage
190	StatefulUniformInt	CollectiveGather	MapUnstage
191	StatelessIf	CollectiveReduce	MapUnstageNoKey
192	StatelessMultinomial	CombinedNonMaxSuppression	MatchingFiles
193	StatelessRandomNormal	CompareAndBitpack	Maximum
194	StatelessRandomUniform	Complex	MaxPoolGradGradWithArgmax
195	StatelessRandomUniformInt	ComputeAccidentalHits	MaxPoolGradWithArgmax
196	StatelessTruncatedNormal	ConcatenateDataset	MaxPoolWithArgmax
197	StatelessWhile	ConditionalAccumulator	MemmappedTensorAllocator
198	StopGradient	Conj	Merge
199	StridedSlice	ConsumeMutexLock	MergeSummary
200	StridedSliceGrad	Conv2DBackpropFilter	MergeV2Checkpoints
201	Sum	Conv2DBackpropInput	Mfcc
202	TensorArrayCloseV3	Conv3DBackpropFilter	Minimum
203	TensorArrayConcatV3	Conv3DBackpropFilterV2	MirrorPadGrad
204	TensorArrayGatherV3	Conv3DBackpropInput	Mod
205	TensorArrayGradV3	Conv3DBackpropInputV2	ModelDataset

206	TensorArrayReadV3	Copy	Mul
207	TensorArrayScatterV3	CopyHost	MultiDeviceIterator
208	TensorArraySizeV3	Cosh	MultiDeviceIteratorFromStringHandle
209	TensorArraySplitV3	CountUpTo	MultiDeviceIteratorGetNextFromShard
210	TensorArrayV3	CreateSummaryDbWriter	MultiDeviceIteratorInit
211	TensorArrayWriteV3	CreateSummaryFileWriter	MultiDeviceIteratorToStringHandle
212	TensorListElementShape	CropAndResize	MutableDenseHashTable
213	TensorListGetItem	CropAndResizeGradBoxes	MutableDenseHashTableV2
214	TensorListLength	CropAndResizeGradImage	MutableHashTable
215	TensorListPopBack	CTCBeamSearchDecoder	MutableHashTableOfTensors
216	TensorListPushBack	CTCGreedyDecoder	MutableHashTableOfTensorsV2
217	TensorListReserve	CTCLoss	MutableHashTableV2
218	TensorListSetItem	CudnnRNN	MutexLock
219	TensorListStack	CudnnRNNBackprop	MutexV2
220	Tile	CudnnRNNBackpropV2	Name:
221	TopKV2	CudnnRNNBackpropV3	NcclAllReduce
222	Transpose	CudnnRNNCanonicalToParams	NcclBroadcast
223	TruncatedNormal	CudnnRNNParamsSize	NcclReduce
224	Unpack	CudnnRNNParamsToCanonical	NearestNeighbors
225	VariableShape	CudnnRNNV2	Neg
226	VarIsInitializedOp	CudnnRNNV3	NegTrain
227	While	DataFormatDimMap	NextAfter
228	ZerosLike	DataFormatVecPermute	NextIteration