Tensorflow 编译加速器 XLA 源码深入解读

Tensorflow 编译加速器 XLA 源码深入解读

Will Zhang
机器学习框架开发者
关注他

XLA是Tensorflow内置的用于加速的编译器，但在实践中，对于不了解其机制的同学来说，往往得不到正收益，甚至经常得到负收益。而本文的目的则是通过讲解XLA内部代码实现来解明其机制。

1. Overview

与XLA相关的代码都在 github.com/tensorflow/t...

其中分为了多个目录，也对应了多个模块，如下

模块	路径
aot	github.com/tensorflow/t...
jit	github.com/tensorflow/t...
tf2xla	github.com/tensorflow/t...
xla/client	github.com/tensorflow/t...
xla/service	github.com/tensorflow/t...

使用有JIT和AOT两种方式，以我见到的场景来看，JIT的用法要更普遍一些，而理解了JIT的内容，再看AOT也比较简单，因此本文只会讲解JIT.

JIT是指，在Tensorflow运行时，无论是训练还是推理，从Tensorflow Graph中切割一部分子图交由XLA编译并运行。JIT的好处就是对用户的负担小，Tensorflow用户只要打开一个开关即可享受到加速的收益。

JIT的驱动方式是向Tensorflow注册了多个优化PASS，定义在这个文件中
github.com/tensorflow/t...

按照执行顺序列出这些PASS

1. IntroduceFloatingPointJitterPass
2. EncapsulateXlaComputationsPass
3. CloneConstantsForBetterClusteringPass
4. ClusterScopingPass
5. MarkForCompilationPass
6. ForceXlaConstantsOnHostPass
7. IncreaseDynamismForAutoJitPass
8. PartiallyDeclusterPass
9. ReportClusteringInfoPass
10. EncapsulateSubgraphsPass
11. BuildXlaOpsPass

这里面比较核心的是EncapsulateXlaComputationsPass, MarkForCompilationPass, EncapsulateSubgraphsPass, BuildXlaOpsPass这四个Pass

EncapsulateXlaComputationsPass服务于jit_compile这种使用方式，Tensorflow提供这种使用方式的目的是为了让用户精细化地控制计算图中哪部分启用XLA. 而EncapsulateXlaComputationsPass做的事情比较简单，识别Python端设置的_xla_compile_id属性，将具有同样_xla_compile_id属性值的算子融合为一个XlaLaunch. 而XlaLaunch算子运行时做的事情也比较简单，将子图交给XLA编译，缓存其编译结果并运行。

EncapsulateXlaComputationsPass之后的所有Pass都服务于autoclustering这种用法，这种使用方式是为了让用户能够一键开启XLA优化。AutoClustering的意思就是自动寻找子图，将单个子图称为Cluster.

对于AutoClustering

1. CloneConstantsForBetterClusteringPass, ClusterScopingPass都是一些准备工作，方便MarkForCompilationPass的处理
2. MarkForCompilationPass负责寻找合适的Cluster, 并为找到的同一个Cluster内所有节点设置同样的_xla_compile_id属性
3. ForceXlaConstantsOnHostPass, IncreaseDynamismForAutoJitPass, PartiallyDeclusterPass则是对MarkForCompilationPass的结果进行微调
4. EncapsulateSubgraphsPass将每个Cluster内的多个节点替换为单个节点，但在单个节点内记录了Cluster内子图的信息
5. 将EncapsulateSubgraphsPass融合的节点替换为XlaCompileOp + XlaRunOp两个算子

XlaCompileOp承载Cluster的所有输入以及子图信息，在运行时进行编译（编译存在缓存）得到XlaExecutableClosure传递给XlaRunOp，而XlaRunOp则运行XlaExecutableClosure.

- CompileToLocalExecutable
- XlaCompileOp::Compute

在XlaCompilationCache::CompileStrict中，其主要分为两步

1. 使用tf2xla模块提供的CompileFunction处理Cluster得到XlaCompilationResult, 在XlaCompilationResult中的主要成员是XlaComputation, 而XlaComputation又是HloModuleProto的一层封装
2. 使用xla/client模块提供的LocalClient::Compile处理XlaCompilationResult, 也就是HloModuleProto, 得到LocalExecutable

而上面两步的产物XlaCompilationResult和LocalExecutable则是XlaExecutableClosure的两个主要成员，最终由XlaCompileOp传递给XlaRunOp进行执行。

至此，JIT的主要职责完成，关于XlaCompilationResult和LocalExecutable则在后面的模块中详解。

3. tf2xla

tf2xla模块的主要职责是对外提供CompileFunction接口，将Tensorflow Graph转为XlaCompilationResult（也就是HloModuleProto）.

同样，我们看一下CompileFunction的核心调用栈（从下往上调用）

- GraphCompiler::Compile
- ExecuteGraph
- XlaCompiler::CompileGraph
- XlaCompiler::CompileFunction

GraphCompiler::Compile的主要逻辑如下

1. 拓扑序遍历所有节点 github.com/tensorflow/t...
2. 为每个Node中的Op创建其对应的XlaOpKernelgithub.com/tensorflow/t...
3. 执行每个Kernel的Compute github.com/tensorflow/t...

这里的XlaOpKernel与Tensorflow中普通的OpKernel不太一样，虽然XlaOpKernel继承自OpKernel.

我们可以看一下XlaOpKernel的实现

```
class XlaOpKernel : public OpKernel {
 public:
  explicit XlaOpKernel(OpKernelConstruction* construction);

  // Subclasses should implement Compile(), much as standard OpKernels implement
  // Compute().
  virtual void Compile(XlaOpKernelContext* context) = 0;

 private:
```

```
  Compile(&xla_context);
}
```

其主要是封装上下文，并让子类实现Compile接口。于是GraphCompiler::Compile中执行每个Kernel的Compute也就是执行每个XlaOpKernel的Compile.

tf2xla中定义的XlaOpKernel都定义在目录 github.com/tensorflow/t...

如果某个Tensorflow Operator在tf2xla中找不到对应的XlaOpKernel, 也就意味着XLA暂时还不支持该Tensorflow Operator.

回到XlaOpKernel::Compile, 为了解释其原理，我们可以找到简单的LeakyReluOp来说明

```
class LeakyReluOp : public XlaOpKernel {
 public:
  explicit LeakyReluOp(OpKernelConstruction* ctx) : XlaOpKernel(ctx) {
    OP_REQUIRES_OK(ctx, ctx->GetAttr("alpha", &alpha_));
  }
  void Compile(XlaOpKernelContext* ctx) override {
    auto features = ctx->Input("features");
    auto prod_with_alpha = features * xla::ScalarLike(features, alpha_);
    auto gt_zero = xla::Gt(features, xla::ScalarLike(features, 0));
    auto output = xla::Select(gt_zero, features, prod_with_alpha);
    ctx->SetOutput(0, output);
  }
  float alpha_;
};
```

在 github.com/tensorflow/t... 中定义了一些基本操作，而LeakyReluOp 调用这些基本操作描述LeakyRelu的计算过程。

需要注意，这里没有真正的数据在计算，仅仅是定义如何计算。

最后调用xla/client模块提供的XlaBuilder::Build得到XlaComputation, 也就是新的计算图的表达方式，HloModuleProto

至此，tf2xla的主要职责也已完成

4. xla/client

xla/client有三个主要的对外接口

1. XlaBuilder配合tf2xla模块的XlaOpKernel将Tensorflow Graph转化为HloModuleProto
2. LocalClient::Compile配合JIT模块将HloModuleProto转为LocalExecutable. 查看实现可知这是xla/service模块的LocalService::CompileExecutables的一层封装
3. LocalExecutable供JIT模块真正执行。查看实现可知这是xla/service模块的Executable::ExecuteAsyncOnStreamWrapper的一层封装

为了了解XlaBuilder, 首先可以看HloModuleProto, 其包含多个HloComputationProto, 且存在一

```
  reserved "root_name";

  string name = 1;

  // The array of instructions is always in a valid dependency order, where
  // operands appear before their users.
  repeated HloInstructionProto instructions = 2;

  // The program shape (with layout) of this computation.

  xla.ProgramShapeProto program_shape = 4;

  // The id of this computation.
  int64 id = 5;

  // The id of the root of the computation.
  int64 root_id = 6;
}
message HloModuleProto {
  string name = 1;
  string entry_computation_name = 2;
  int64 entry_computation_id = 6;

  // The array of computations is always in a valid dependency order, where
  // callees appear before their callers.
  repeated HloComputationProto computations = 3;

  // The host program shape (with layout) of the entry computation.
  xla.ProgramShapeProto host_program_shape = 4;

  // The id of this module.
  int64 id = 5;

  // The schedule for this module.
  HloScheduleProto schedule = 7;

  // Describes alias information between inputs and outputs.
  HloInputOutputAliasProto input_output_alias = 8;

  DynamicParameterBindingProto dynamic_parameter_binding = 9;

  repeated CrossProgramPrefetch cross_program_prefetches = 10;

  // True if the module contains dynamic computation.
  bool is_dynamic = 11;
}
```

对于HloModuleProto/HloComputationProto/HloInstructionProto的关系可以大致理解为程序/函数/指令。

在tf2xla一节中我们提到，XlaOpKernel使用xla_builder提供的基本原语来描述其计算过程，那么我们接下来仔细看一下基本原语内部做了什么，以Transpose为例如下

```
  HloInstructionProto instr;
  *instr.mutable_shape() = shape.ToProto();
  for (int64_t dim : permutation) {
    instr.add_dimensions(dim);
  }
  return AddInstruction(std::move(instr), HloOpcode::kTranspose, {operand});
}
XlaOp XlaBuilder::Transpose(XlaOp operand,
                            absl::Span<const int64_t> permutation) {
  return ReportErrorOrReturn([&]() -> StatusOr<XlaOp> {
    TF_ASSIGN_OR_RETURN(const Shape* operand_shape, GetShapePtr(operand));
    TF_ASSIGN_OR_RETURN(Shape shape, ShapeInference::InferTransposeShape(
                                         *operand_shape, permutation));
    return TransposeInternal(shape, operand, permutation);
  });
}
XlaOp Transpose(const XlaOp operand, absl::Span<const int64_t> permutation) {
  return operand.builder()->Transpose(operand, permutation);
}
```

在XlaBuilder中有两个重要成员

```
std::vector<HloInstructionProto> instructions_;
std::map<int64, HloComputationProto> embedded_;
```

XlaOpKernel调用的xla_builder的原语主要功能是插入描述的指令(HloInstructionProto)以及调用的函数(HloComputationProto)到这两个成员中。

而添加完成后，最终XlaBuilder::Build创建一个称为entry的HloComputationProto作为HloModuleProto的入口，并将其存储的instructions_打包入entry中，再将entry以及所有的embedded_打包到HloModuleProto中。

至此，XlaBuilder的使命完成。

5. xla/service

xla/service的核心职责是将HloModuleProto编译为能够真正执行的Executable

我们看一下LocalService::CompileExecutables的核心调用栈（从下往上调用）

- LLVMCompiler::Compile
- Compiler::Compile
- Service::BuildExecutables
- LocalService::CompileExecutables

可以看到，最终会调用到LLVMCompiler中，我们细看一下LLVMCompiler::Compile代码

```
StatusOr<std::vector<std::unique_ptr<Executable>>> LLVMCompiler::Compile(
    std::unique_ptr<HloModuleGroup> module_group,
```

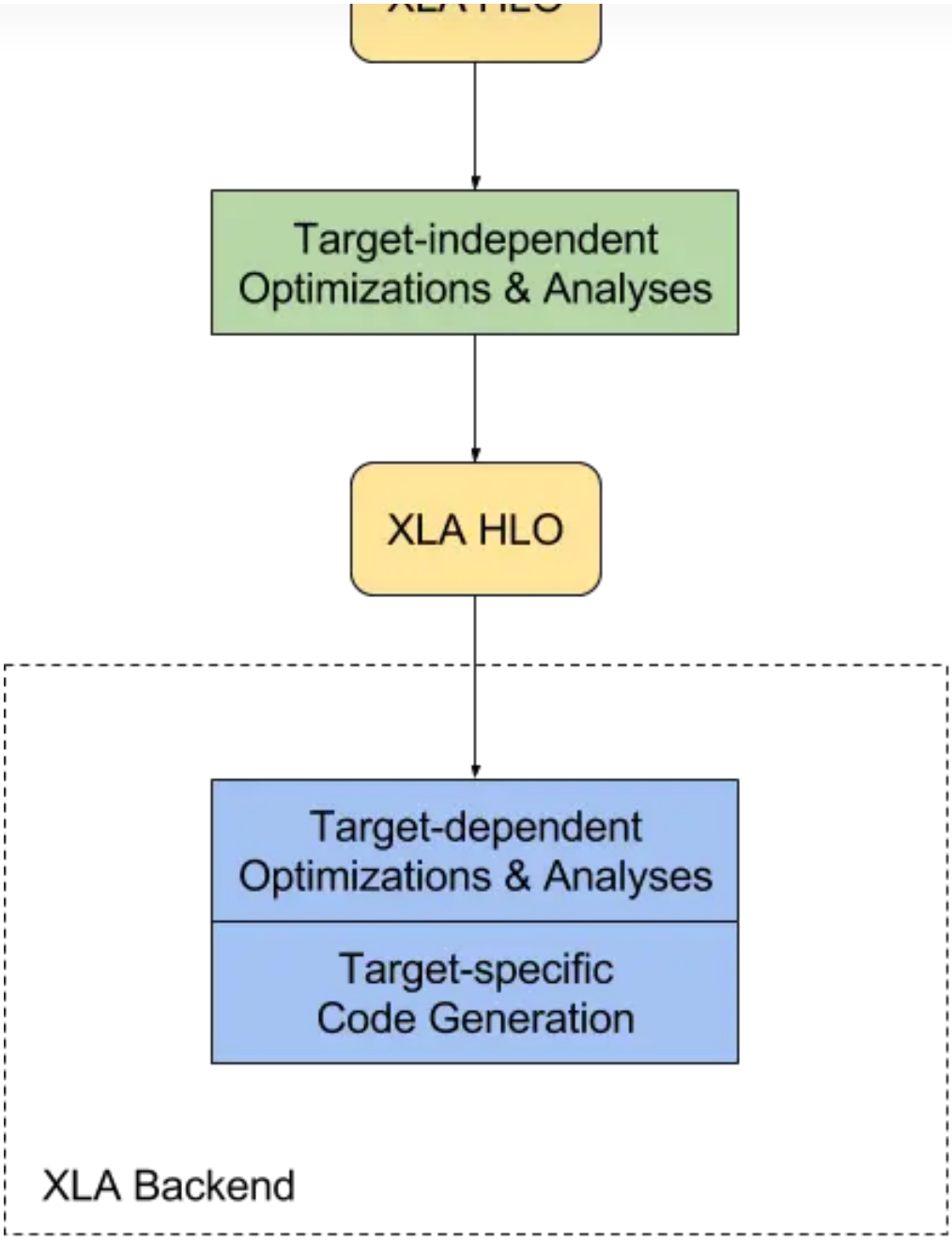
```
  //  - Denormals are zero (DAZ): roughly, operations treat denormal floats as
  //    zero.
  //  - Flush denormals to zero (FTZ): roughly, operations produce zero instead
  //    of denormal floats.
  //
  // In theory enabling these shouldn't matter since the compiler should ideally
  // not leak its environment into generated code, but we turn off DAZ and FTZ
  // to get some defense-in-depth.
  tensorflow::port::ScopedDontFlushDenormal dont_flush_denormals;

  std::vector<std::unique_ptr<Executable>> result;
  std::vector<std::unique_ptr<HloModule>> modules =
      module_group->ConsumeModules();
  for (size_t i = 0; i < modules.size(); i++) {
    TF_ASSIGN_OR_RETURN(modules[i],
                        RunHloPasses(std::move(modules[i]), stream_execs[i][0],
                                     options.device_allocator));
    TF_ASSIGN_OR_RETURN(std::unique_ptr<Executable> executable,
                        RunBackend(std::move(modules[i]), stream_execs[i][0],
                                   options.device_allocator));
    result.push_back(std::move(executable));
  }

  return {std::move(result)};
}
```

可以看到分为两个步骤

1. RunHloPasses, 对应于下图的硬件无关优化
2. RunBackend, 对应于下图的硬件相关优化以及Codegen



GPU上的RunHloPasses最终会调用到GpuCompiler::OptimizeHloModule, 在其中会调用非常多的pass来做优化，这里列举一下其主要阶段，每个阶段包含了多个pass

1. SPMD
2. optimization
3. collective-optimizations
4. OptimizeHloConvolutionCanonicalization
5. layout assignment
6. OptimizeHloPostLayoutAssignment
7. fusion
8. horizontal_fusion
9. post-fusion

1. CompileModuleToLlvmIrImpl将HloModule编译为LLVM IR
2. CompileToTargetBinary将LLVM IR编译成cubin

而CompileModuleToLlvmIrImpl主要逻辑将调用到IrEmitterUnnested::EmitLmhloRegion，如下

```
Status IrEmitterUnnested::EmitLmhloRegion(mlir::Region* region) {
  for (mlir::Operation& op : llvm::make_early_inc_range(region->front())) {
    TF_RETURN_IF_ERROR(EmitOp(&op));
  }
  return Status::OK();
}
```

EmitOp就是给每个OP做Codegen, 如果对Codegen不了解，可以看我之前的文章 使用LLVM实现一门语言了解一下

接下来看CompileToTargetBinary, 其核心逻辑在调用NVPTXCompiler::CompileTargetBinary，代码如下

```
StatusOr<std::pair<std::string, std::vector<uint8>>>
NVPTXCompiler::CompileTargetBinary(const HloModuleConfig& module_config,
                                   llvm::Module* llvm_module,
                                   GpuVersion gpu_version,
                                   se::StreamExecutor* stream_exec,
                                   bool relocatable,
                                   const HloModule* debug_module) {
  std::string libdevice_dir;
  {
    tensorflow::mutex_lock lock(mutex_);

    // Find the directory containing libdevice.  To avoid searching for it every
    // time, we have a one-element cache, keyed on the module's config's
    // cuda_data_dir.
    if (cached_libdevice_dir_.empty()) {
      cached_libdevice_dir_ = GetLibdeviceDir(module_config);
    }
    libdevice_dir = cached_libdevice_dir_;
  }
  VLOG(2) << "Libdevice dir = " << libdevice_dir << "\n";
  std::unique_ptr<llvm::Module> loaded_module =
      MaybeLoadLLVMFromFile(debug_module, llvm_module);
  llvm::Module* selected_module = nullptr;
  if (loaded_module) {
    selected_module = loaded_module.get();
  } else {
    selected_module = llvm_module;
  }

  string ptx;
  if (!(debug_module &&
        MaybeLoadPtxFromFile(module_config, debug_module, &ptx))) {
```

```
  }

  std::vector<uint8> cubin = CompileGpuAsmOrGetCachedResult(
      stream_exec, ptx, absl::get<se::CudaComputeCapability>(gpu_version),
      module_config, relocatable);

  return std::pair<std::string, std::vector<uint8>>(std::move(ptx),
                                                    std::move(cubin));
}
```

代码里先编译出PTX, 再编译出我们想要的最终结果cubin

6. 结语

至此，XLA的源码梳理完成。

本文如有问题，欢迎指正挑错，欢迎多交流（线上线下均可），共同学习共同进步，下一篇文章见。

编辑于 2021-10-30 06:51

TensorFlow 学习　编译器　深度学习编译优化

24 条评论　默认　最新

raymond
感谢作者，这里有个问题，就是针对spmd的代码分发和执行，是如何做的？比如在jit编译阶段如何体现spmd?执行阶段是如何把executable程序分发到各个device执行的?
08-13　回复　1

夜.月
很赞，梳理的很清晰。
其实大多数人了解怎样添加优化PaaS就够了，少数芯片厂商需要了解HLO对接。
2021-11-29　回复　1

圈圈虫
Bangdai 君tql
2021-11-02　回复　1