TensorFlow技术内幕（七）：模型优化之XLA（上）

本章中我们分析一下TensorFlow的XLA（Accelerated Linear Algebra 加速线性代数)的内核实现。代码位置在tensorflow/compiler.

XLA

在XLA技术之前，TensorFlow中计算图的执行是由runtime(运行时)代码驱动的：runtime负责加载计算图定义、创建计算图、计算图分区、计算图优化、分配设备、管理节点间的依赖并调度节点kernel的执行；计算图是数据部分，runtime是代码部分。在第五章session类的实现分析中，我们已经比较详细的分析了这个过程。在XLA出现之后，我们有了另一个选择，计算图现在可以直接被编译成目标平台的可执行代码，可以直接执行，不需要runtime代码的参与了。

本章我就来分析一下XLA是如何将tensorflow.GraphDef编译成可执行代码的。

目前XLA提供了AOT(提前编译)和JIT(即时编译)两种方式。

AOT

在编译技术里，AOT(提前编译)方式就是在代码执行阶段之前全部编译成目标指令，进入执行阶段后，不再有编译过程发生。

tensorflow的官网已经介绍了一个AOT的使用例子，这里引用一下这个例子，代码位于tensorflow/compiler/aot/tests/make_test_graphs.py，函数tfmatmul构建了一个简单的网络如下：

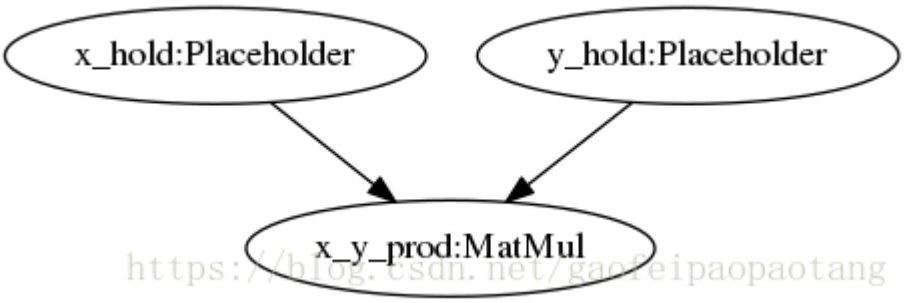

图1：matmul网络

例子中，我们将使用XLA的AOT方式将这计算图编译成可执行文件，需要四步：

步骤1：编写配置

配置网络的输入和输出节点，对应生成函数的输入输出参数。

```
/* tensorflow/compiler/aot/tests/test_graph_tfmatmul.config.pbtxt */
# Each feed is a positional input argument for the generated function.  The order
```

```
# of each entry matches the order of each input argument.  Here “x_hold” and “y_hold”
# refer to the names of placeholder nodes defined in the graph.
feed {
  id { node_name: "x_hold" }
  shape {
    dim { size: 2 }
    dim { size: 3 }
  }
}
feed {
  id { node_name: "y_hold" }
  shape {
    dim { size: 3 }
    dim { size: 2 }
  }
}

# Each fetch is a positional output argument for the generated function.  The order
# of each entry matches the order of each output argument.  Here “x_y_prod”
# refers to the name of a matmul node defined in the graph.
fetch {
  id { node_name: "x_y_prod" }
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25

步骤2：使用tf_library构建宏来编译子图为静态链接库

```
load("//third_party/tensorflow/compiler/aot:tfcompile.bzl", "tf_library")

# Use the tf_library macro to compile your graph into executable code.
tf_library(
    # name is used to generate the following underlying build rules:
    # <name>           : cc_library packaging the generated header and object files
    # <name>_test      : cc_test containing a simple test and benchmark
    # <name>_benchmark : cc_binary containing a stand-alone benchmark with minimal deps;
    #                    can be run on a mobile device
    name = "test_graph_tfmatmul",
    # cpp_class specifies the name of the generated C++ class, with namespaces allowed.
    # The class will be generated in the given namespace(s), or if no namespaces are
    # given, within the global namespace.
```

```
    cpp_class = "foo::bar::MatMulComp",
    # graph is the input GraphDef proto, by default expected in binary format.  To
    # use the text format instead, just use the ‘.pbtxt’ suffix.  A subgraph will be
    # created from this input graph, with feeds as inputs and fetches as outputs.
    # No Placeholder or Variable ops may exist in this subgraph.
    graph = "test_graph_tfmatmul.pb",
    # config is the input Config proto, by default expected in binary format.  To
    # use the text format instead, use the ‘.pbtxt’ suffix.  This is where the
    # feeds and fetches were specified above, in the previous step.
    config = "test_graph_tfmatmul.config.pbtxt",
)
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

步骤3：编写代码以调用子图

第二步会生成一个头文件和Object文件，头文件test_graph_tfmatmul.h的内容如下：

```
/* test_graph_tfmatmul.h */
namespace foo {
namespace bar {

// MatMulComp represents a computation previously specified in a
// TensorFlow graph, now compiled into executable code.
class MatMulComp {
 public:
  // AllocMode controls the buffer allocation mode.
  enum class AllocMode {
    ARGS_RESULTS_AND_TEMPS,  // Allocate arg, result and temp buffers
    RESULTS_AND_TEMPS_ONLY,  // Only allocate result and temp buffers
  };

  MatMulComp(AllocMode mode = AllocMode::ARGS_RESULTS_AND_TEMPS);
  ~MatMulComp();

  // Runs the computation, with inputs read from arg buffers, and outputs
  // written to result buffers. Returns true on success and false on failure.
  bool Run();

  // Arg methods for managing input buffers. Buffers are in row-major order.
  // There is a set of methods for each positional argument.
  void** args();
```

```
  void set_arg0_data(float* data);
  float* arg0_data();
  float& arg0(size_t dim0, size_t dim1);

  void set_arg1_data(float* data);
  float* arg1_data();
  float& arg1(size_t dim0, size_t dim1);

  // Result methods for managing output buffers. Buffers are in row-major order.
  // Must only be called after a successful Run call. There is a set of methods
  // for each positional result.
  void** results();

  float* result0_data();
  float& result0(size_t dim0, size_t dim1);
};

}  // end namespace bar
}  // end namespace foo
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44

引用头文件，编写使用端代码：

```
#define EIGEN_USE_THREADS
#define EIGEN_USE_CUSTOM_THREAD_POOL

#include <iostream>
#include "third_party/eigen3/unsupported/Eigen/CXX11/Tensor"
#include "tensorflow/compiler/aot/tests/test_graph_tfmatmul.h" // generated

int main(int argc, char** argv) {
  Eigen::ThreadPool tp(2);  // Size the thread pool as appropriate.
  Eigen::ThreadPoolDevice device(&tp, tp.NumThreads());

  foo::bar::MatMulComp matmul;
  matmul.set_thread_pool(&device);

  // Set up args and run the computation.
  const float args[12] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  std::copy(args + 0, args + 6, matmul.arg0_data());
  std::copy(args + 6, args + 12, matmul.arg1_data());
  matmul.Run();

  // Check result
  if (matmul.result0(0, 0) == 58) {
    std::cout << "Success" << std::endl;
  } else {
    std::cout << "Failed. Expected value 58 at 0,0. Got:"
              << matmul.result0(0, 0) << std::endl;
  }

  return 0;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30

步骤4：使用cc_binary创建最终的可执行二进制文件

```
# Example of linking your binary
# Also see //third_party/tensorflow/compiler/aot/tests/BUILD
load("//third_party/tensorflow/compiler/aot:tfcompile.bzl", "tf_library")

# The same tf_library call from step 2 above.
tf_library(
    name = "test_graph_tfmatmul",
    ...
)

# The executable code generated by tf_library can then be linked into your code.
cc_binary(
    name = "my_binary",
    srcs = [
        "my_code.cc",  # include test_graph_tfmatmul.h to access the generated header
    ],
    deps = [
        ":test_graph_tfmatmul",  # link in the generated object file
        "//third_party/eigen3",
    ],
    linkopts = [
          "-lpthread",
    ]
)
```

四步编译出了可执行的文件，但是其实第二步中，tf_library宏的输出就是计算图对应的可执行文件了，包含一个头文件和Object文件。 所以计算图的编译工作主要在tf_library完成的，我们来分析一下tf_library的实现, tf_library定义在文件tensorflow/compiler/aot/tfcompile.bzl中：

```
/* tensorflow/compiler/aot/tfcompile.bzl */
...
def tf_library(name, graph, config,
               freeze_checkpoint=None, freeze_saver=None,
               cpp_class=None, gen_test=True, gen_benchmark=True,
               visibility=None, testonly=None,
               tfcompile_flags=None,
               tfcompile_tool="//tensorflow/compiler/aot:tfcompile",
               deps=None, tags=None):
```

```
...
# Rule that runs tfcompile to produce the header and object file.
  header_file = name + ".h"
  object_file = name + ".o"
  ep = ("__" + PACKAGE_NAME + "__" + name).replace("/", "_")
  native.genrule(
      name=("gen_" + name),
      srcs=[
          tfcompile_graph,
          config,
      ],
      outs=[
          header_file,
          object_file,
      ],
      cmd=("$(location " + tfcompile_tool + ")" +
           " --graph=$(location " + tfcompile_graph + ")" +
           " --config=$(location " + config + ")" +
           " --entry_point=" + ep +
           " --cpp_class=" + cpp_class +
           " --target_triple=" + target_llvm_triple() +
           " --out_header=$(@D)/" + header_file +
           " --out_object=$(@D)/" + object_file +
           " " + (tfcompile_flags or "")),
      tools=[tfcompile_tool],
      visibility=visibility,
      testonly=testonly,
      # Run tfcompile on the build host since it's typically faster on the local
      # machine.
      #
      # Note that setting the local=1 attribute on a *test target* causes the
      # test infrastructure to skip that test.  However this is a genrule, not a
      # test target, and runs with --genrule_strategy=forced_forge, meaning the
      # local=1 attribute is ignored, and the genrule is still run.
      #
      # https://www.bazel.io/versions/master/docs/be/general.html#genrule
      local=1,
      tags=tags,
  )
...
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50

上面我节选了tf_library代码中关键的一步，这步调用tfcompile_tool命令行工具，生成头文件和二进制问题。可以看到调用tfcompile_tool的命令行包括–graph，–config等等。

tfcompile_tool的入口main函数定义在tensorflow/compiler/aot/tfcompile_main.cc中，编译过程主要分为四步：

1、由GraphDef构建tensorflow.Graph。

2、调用xla.XlaCompiler.CompileGraph，将tensorflow.Graph编译为xla.Computation。

3、调用xla.CompileOnlyClient.CompileAheadOfTime函数，将xla.Computation编译为可执行代码。

4、保存编译结果到头文件和object文件

TensorFlow目前支持的AOT编译的平台有x86-64和ARM.

JIT

JIT全称Just In Time（即时）.在即时编译中，计算图在不会在运行阶段前被编译成可执行代码，而是在进入运行阶段后的适当的时机才会被编译成可执行代码，并且可以被直接调用了。

关于JIT编译与AOT编译优缺点的对比，不是本章的主题，限于篇幅这里不做过多的分析了。我们直接来看TensorFlow中JIT的实现。

Python API中打开JIT支持的方式有一下几种：

方式一、通过Session设置：

这种方式的影响是Session范围的，内核会编译尽可能多的节点。

```
# Config to turn on JIT compilation
config = tf.ConfigProto()
config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1

sess = tf.Session(config=config)
```

- 1
- 2
- 3
- 4
- 5

方式二、通过tf.contrib.compiler.jit.experimental_jit_scope():

这种方式影响scope内的所有节点，这种方式会对Scope内的所有节点添加一个属性并设置为true: _XlaCompile=true.

```
jit_scope = tf.contrib.compiler.jit.experimental_jit_scope

x = tf.placeholder(np.float32)
with jit_scope():
   y = tf.add(x, x)  # The "add" will be compiled with XLA.
```

- 1
- 2
- 3
- 4
- 5
- 6

方式三、通过设置device:

通过设置运行的Device来启动JIT支持。

```
with tf.device("/job:localhost/replica:0/task:0/device:XLA_GPU:0"):
  output = tf.add(input1, input2)
```

- 1
- 2

接下来我们来分析一下这个问题：上面的这些接口层的设置，最终是如何影响内核中计算图的计算的呢?

首先来回忆一下 TensorFlow技术内幕（五）：核心概念的实现分析 的图4，session的本地执行这一节：graph在运行前，需要经过一系列优化和重构（包括前一章中分析的grappler模块的优化）。其中一步涉及到类：tensorflow.OptimizationPassRegistry，通过此类我们可以运行其中注册的tensorflow.GraphOptimizationPass的子类，每一个子类都是实现了一种graph的优化和重构的逻辑。XLA JIT 相关的Graph优化和重构，也是通过这个入口来执行的。

JIT相关的tensorflow.GraphOptimizationPass注册代码在：

```
/* tensorflow/compiler/jit/jit_compilation_pass_registration.cc */
...
namespace tensorflow {

REGISTER_OPTIMIZATION(OptimizationPassRegistry::POST_REWRITE_FOR_EXEC, 10,
                      MarkForCompilationPass);

// The EncapsulateSubgraphs pass must run after the MarkForCompilationPass. We
```

```
// also need to run it after the graph been rewritten to have _Send nodes added
// for fetches. Before the _Send nodes are added, fetch nodes are identified by
// name, and encapsulation might remove that node from the graph.
REGISTER_OPTIMIZATION(OptimizationPassRegistry::POST_REWRITE_FOR_EXEC, 20,
                      EncapsulateSubgraphsPass);

// Must run after EncapsulateSubgraphsPass.
REGISTER_OPTIMIZATION(OptimizationPassRegistry::POST_REWRITE_FOR_EXEC, 30,
                      BuildXlaLaunchOpsPass);

}  // namespace tensorflow
...
```

可以看到JIT编译相关的tensorflow.GraphOptimizationPass有三个：

1、tensorflow.MarkForCompilationPass：
上面提到的开启JIT的三种设置方式，就是在此类中进行检查的。通过检查这些设置，此类首先会挑选出所有开启JIT并且目前版本支持JIT编译的节点，并且运行聚类分析，将这些等待JIT编译的节点分到若干个Cluster中，看一下下面的例子：

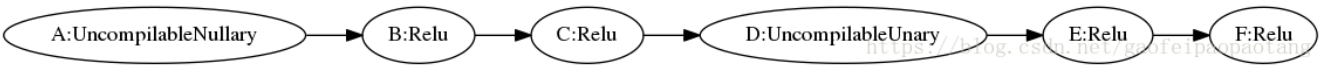

图2：MarkForCompilationPass优化前

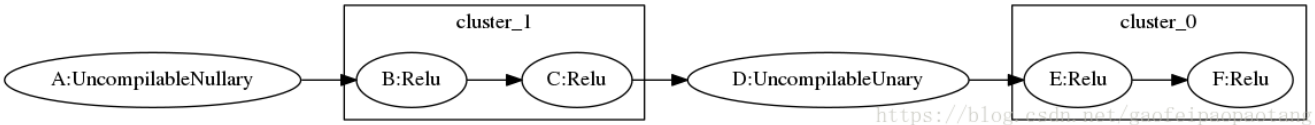

图3：MarkForCompilationPass优化后

B,C节点被标记到cluster 1，E，F节点被标记到cluster 0. A，E应为不支持编译所以没有被分配cluster.

2、tensorflow.EncapsulateSubgraphsPass：

这一步优化分三步，

第一步：为上一个优化类MarkForCompilationPass mark形成的cluster分别创建对应的SubGraph对象。

第二步：为每个SubGraph对象创建对应的FunctionDef，并将创建的FunctionDef添加到FunctionLibrary中。

这里补充一下TensorFlow中Funtion的概念，FucntionDef的定义如下：

```
/* tensorflow/core/framework/function.proto */

message FunctionDef {
  // The definition of the function's name, arguments, return values,
  // attrs etc.
  OpDef signature = 1;

  map<string, AttrValue> attr = 5;

  repeated NodeDef node_def = 3;

  map<string, string> ret = 4;
}
```

Function可以看做一个独立的计算图，node_def就是这个子图包含的所有节点。Function可以被实例化和调用，方式是向调用方的计算图中插入一个Call节点，这类节点的运算核(OpKernel)是CallOp:

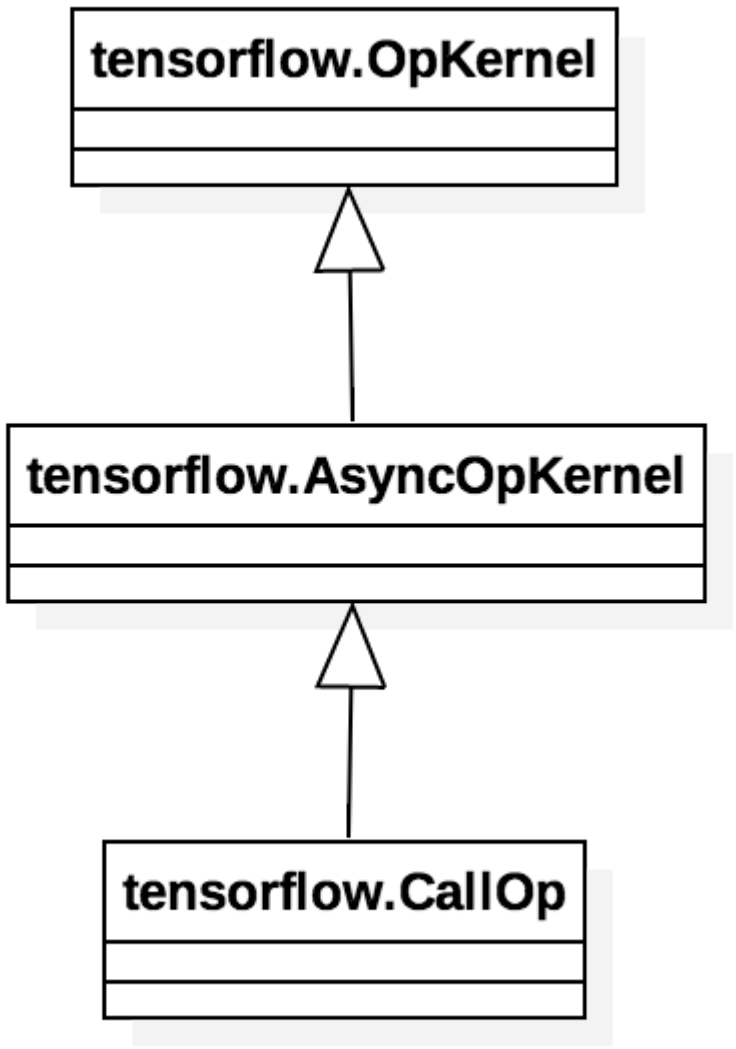

图4：类CallOp

我们知道计算图的计算最终是由Executor对象驱动的，CallOp是连接调用方计算图的Executor和Function内计算图的桥梁：CallOp对外响应Executor的调用，对内会为每次调用创建一个独立的Executor来驱动Function内部计算图的运算。

第三步：重新创建一张新的计算图，首先将原计算图中没有被mark的节点直接拷贝过来，然后为每个SubGraph对应的Function创建CallOp节点，最后创建计算图中数据和控制依赖关系。

下面的例子中，就将C和c节点一起，替换成了F1节点，调用了Function F1:

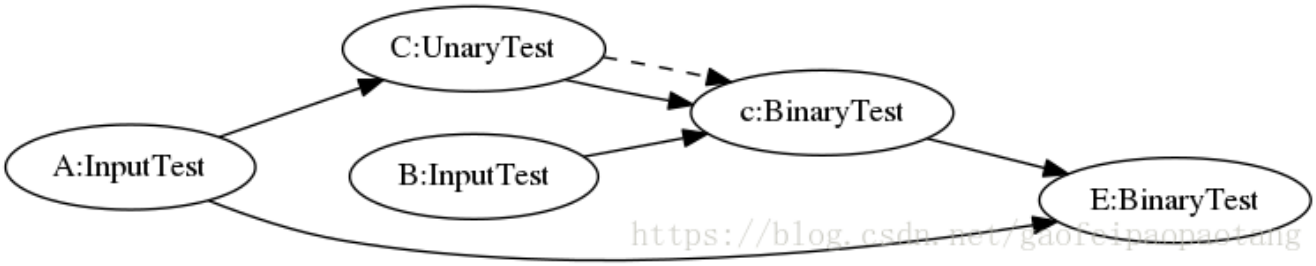

图5：EncapsulateSubgraphsPass优化前

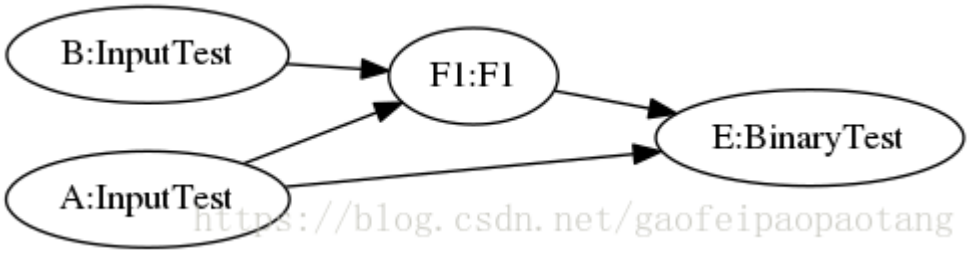

图6：EncapsulateSubgraphsPass优化后

3、tensorflow.BuildXlaLaunchOpsPass：
经过EncapsulateSubgraphsPass优化的计算图中的function call节点全部替换成xlalaunch节点。

JIT的关键就是这个xlalaunch节点。xlalaunch节点节点的运算名为”_XlaLaunch”,运算核是XlaLocalLaunchOp，按照运算核的要求它的父类也是OpKernel。

XlaLocalLaunchOp对外响应Executor的调用请求，对内调用JIT相关API类编译和执行FunctionDef。当然对编译结果会有缓存操作，没必要每次调用都走一次编译过程：

步骤一：调用XlaCompilationCache的将FunctionDef编译为xla.LocalExecutable。在cache没命中的情况下，会调用xla.LocalClient执行真正的编译

步骤二：调用xla.LocalExecutable.Run

JIT方式目前支持的平台有X86-64, NVIDIA GPU。

小结

以上分析的是XLA在TensorFlow中的调用方式：AOT方式和JIT方式。

两种方式下都会将整个计算图或则计算图的一部分直接编译成可执行代码。两则的区别也是比较明显的，除了编译时机不一样外，还有就是runtime（运行时）的参与程度。AOT中彻底不需要运行时的参与了，而JIT中还是需要运行时参与的，但是JIT会优化融合原计算图中的节点，加入XlaLaunch节点，来加速计算图的执行。

后面我们会详细分析一下XLA这个编译器的内部实现。