Using AOT compilation

What is tfcompile?

tfcompile is a standalone tool that ahead-of-time (AOT) compiles TensorFlow graphs into executable code. It can reduce total binary size, and also avoid some runtime overheads. A typical use-case of **tfcompile** is to compile an inference graph into executable code for mobile devices.

The TensorFlow graph is normally executed by the TensorFlow runtime. This incurs some runtime overhead for execution of each node in the graph. This also leads to a larger total binary size, since the code for the TensorFlow runtime needs to be available, in addition to the graph itself. The executable code produced by **tfcompile** does not use the TensorFlow runtime, and only has dependencies on kernels that are actually used in the computation.

The compiler is built on top of the XLA framework. The code bridging TensorFlow to the XLA framework resides under tensorflow/compiler (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/), which also includes support for just-in-time (JIT) compilation (https://www.tensorflow.org/performance/xla/jit) of TensorFlow graphs.

What does tfcompile do?

tfcompile takes a subgraph, identified by the TensorFlow concepts of feeds and fetches, and generates a function that implements that subgraph. The **feeds** are the input arguments for the function, and the **fetches** are the output arguments for the function. All inputs must be fully specified by the feeds; the resulting pruned subgraph cannot contain Placeholder or Variable nodes. It is common to specify all Placeholders and Variables as feeds, which ensures the resulting subgraph no longer contains these nodes. The generated function is packaged as a **cc_library**, with a header file exporting the function signature, and an object file containing the implementation. The user writes code to invoke the generated function as appropriate.

Using tfcompile

This section details high level steps for generating an executable binary with **tfcompile** from a TensorFlow subgraph. The steps are:

- Step 1: Configure the subgraph to compile
- Step 2: Use the **tf_library** build macro to compile the subgraph
- Step 3: Write code to invoke the subgraph
- Step 4: Create the final binary

Step 1: Configure the subgraph to compile

Identify the feeds and fetches that correspond to the input and output arguments for the generated function. Then configure the **feeds** and **fetches** in a **tensorflow.tfcompile.Config** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/aot/tfcompile.proto) proto.

```
# 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" }
}
```

Step 2: Use tf_library build macro to compile the subgraph

This step converts the graph into a **`cc_library`** using the **`tf_library`** build macro. The **`cc_library`** consists of an object file containing the code generated from the graph, along with a header file that gives access to the generated code. **`tf_library`** utilizes **`tfcompile`** to compile the TensorFlow graph into executable code.

```
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",
)
```

To generate the GraphDef proto (test_graph_tfmatmul.pb) for this example, run make_test_graphs.py (https://www.tensorflow.org) and specify the output location with the –out_dir flag.

Typical graphs contain **`Variables`** (https://www.tensorflow.org/api_guides/python/state_ops) representing the weights that are learned via training, but **`tfcompile`** cannot compile a subgraph that contain **`Variables`**. The freeze_graph.py (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/python/tools/freeze_graph.py) tool converts variables into constants, using values stored in a checkpoint file. As a convenience, the **`tf_library`** macro supports the **`freeze_checkpoint`** argument, which runs the tool. For more examples see tensorflow/compiler/aot/tests/BUILD (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/aot/tests/BUILD).

Constants that show up in the compiled subgraph are compiled directly into the generated code. To pass the constants into the generated function, rather than having them compiled-in, simply pass them in as feeds.

For details on the **`tf_library`** build macro, see tfcompile.bzl (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/aot/tfcompile.bzl).

For details on the underlying **`tfcompile`** tool, see tfcompile_main.cc (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/aot/tfcompile_main.cc).

Step 3: Write code to invoke the subgraph

This step uses the header file (**`test_graph_tfmatmul.h`**) generated by the **`tf_library`** build macro in the previous step to invoke the generated code. The header file is located in the **`bazel-genfiles`** directory corresponding to the build package, and is named based on the name attribute set on the **tf_library** build macro. For example, the header generated for **test_graph_tfmatmul** would be **test_graph_tfmatmul.h**. Below is an abbreviated version of what is generated. The generated file, in **bazel-genfiles**, contains additional useful comments.

```
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
```

The generated C++ class is called **MatMulComp** in the **foo::bar** namespace, because that was the **cpp_class** specified in the **tf_library** macro. All generated classes have a similar API, with the only difference being the methods to handle arg and result buffers. Those methods differ based on the number and types of the buffers, which were specified by the **feed** and **fetch** arguments to the **tf_library** macro.

There are three types of buffers managed within the generated class: **args** representing the inputs, **results** representing the outputs, and **temps** representing temporary buffers used internally to perform the computation. By default, each instance of the generated class allocates and manages all of these buffers for you. The **AllocMode** constructor argument may be used to change this behavior. A convenience library is provided in **tensorflow/compiler/aot/runtime.h** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/aot/runtime.h) to help with manual buffer allocation; usage of this library is optional. All buffers should be aligned to 32-byte boundaries.

The generated C++ class is just a wrapper around the low-level code generated by XLA.

Example of invoking the generated function based on **tfcompile_test.cc** (https://www.github.com/tensorflow/tensorflow/blob/r1.3/tensorflow/compiler/aot/tests/tfcompile_test.cc):

```
#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;
}
```

Step 4: Create the final binary

This step combines the library generated by `tf_library` in step 2 and the code written in step 3 to create a final binary. Below is an example `bazel` BUILD file.

```
# 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",
    ]
)
```