Tensorflow

Agenda

- Tensorflow Introduction
- Conferences
- XLA
  - just-in-time (JIT)
  - ahead-of-time (AOT)
- JIT and AOT examples

Tensorflow

TensorFlow™ is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

Tensors

A tensor consists of a set of primitive values shaped into an array of any number of dimensions.

```
3 # a rank 0 tensor; this is a scalar with shape []
[1., 2., 3.] # a rank 1 tensor; this is a vector with shape [3]
[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```

Rank and Shape

Rank	Math entity
0	Scalar (magnitude only)
1	Vector (magnitude and direction)
2	Matrix (table of numbers)
3	3-Tensor (cube of numbers)
n	n-Tensor (you get the idea)

Rank	Shape	Dimension number	Example
0	[]	0-D	A 0-D tensor. A scalar.
1	[D0]	1-D	A 1-D tensor with shape [5].
2	[D0, D1]	2-D	A 2-D tensor with shape [3, 4].
3	[D0, D1, D2]	3-D	A 3-D tensor with shape [1, 4, 3].
n	[D0, D1, ... Dn-1]	n-D	A tensor with shape [D0, D1, ... Dn-1].

Constants

Constants Variables have a value that does not change during runtime.

```
node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0) # also tf.float32 implicitly
print(node1, node2)
```

Placeholders

A graph can be parameterized to accept external inputs, known as placeholders. A placeholder is a promise to provide a value later.

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = a + b  # + provides a shortcut for tf.add(a, b)
```

```
print(sess.run(adder_node, {a: 3, b: 4.5}))
print(sess.run(adder_node, {a: [1, 3], b: [2, 4]}))
```

Variables

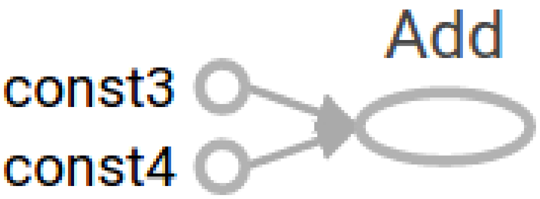
To make the model trainable, we need to be able to modify the graph to get new outputs with the same input. Variables allow us to add trainable parameters to a graph.

```
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-.3], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W * x + b
sess = tf.Session()
print(sess.run(linear_model, {x: [1, 2, 3, 4]}))
```

Computational Graph

A computational graph is a series of TensorFlow operations arranged into a graph of nodes

```
node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0) # also tf.float32 implicitly
node3 = tf.add(node1, node2)
sess  = tf.Session()
print("sess.run(node3):", sess.run(node3))

//Print result: sess.run(node3): 7.0
```



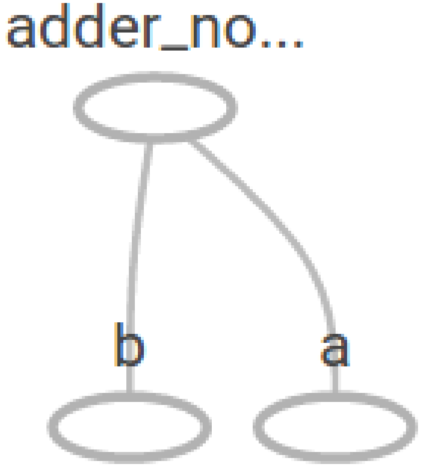
Session

A **session** encapsulates the control and state of the TensorFlow runtime.

```
# Create a default in-process session.
with tf.Session() as sess:
  # ...

# Create a remote session.
with tf.Session("grpc://example.org:2222"):
  # ...
```

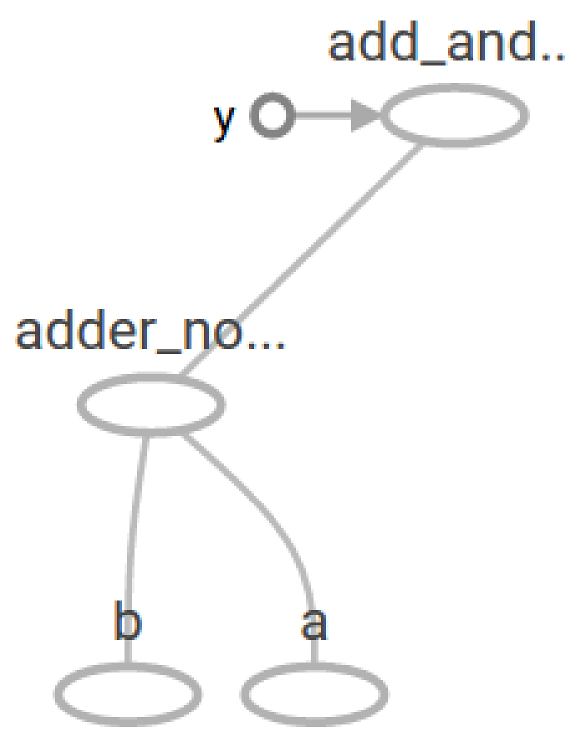
Computational Graph

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = a + b  # + provides a shortcut for tf.add(a, b)
sess = tf.Session()
print(sess.run(adder_node, {a: 3, b: 4.5}))
print(sess.run(adder_node, {a: [1, 3], b: [2, 4]}))

//print results:
7.5
[ 3.  7.]
```



Computational Graph

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = a + b  # + provides a shortcut for tf.add(a, b)
add_and_triple = adder_node * 3.
sess = tf.Session()
print(sess.run(add_and_triple, {a: 3, b: 4.5}))

//print result:
22.5
```



Device Placement

"/cpu:0": The CPU of your machine.

"/gpu:0": The GPU of your machine, if you have one.

"/gpu:1": The second GPU of your machine, etc.

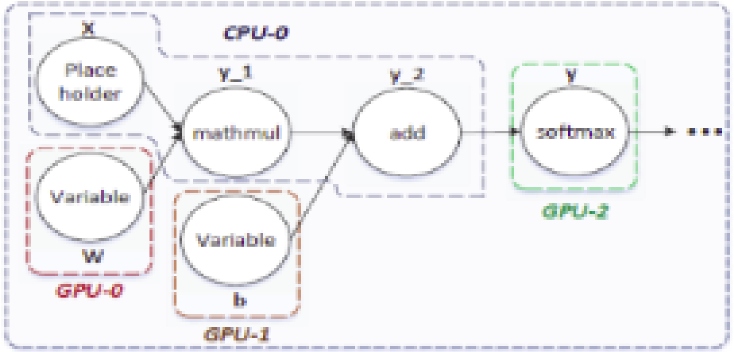
EXEMPLO 1: -- variable

```
with tf.device("/gpu:1"):
  v = tf.get_variable("v", [1])
```

EXEMPLO 2: -- computation

```
with tf.device("/device:CPU:0"):
  # Operations created in this context will be pinned to the CPU.
  img = tf.decode_jpeg(tf.read_file("img.jpg"))

with tf.device("/device:GPU:0"):
  # Operations created in this context will be pinned to the GPU.
  result = tf.matmul(weights, img)
```

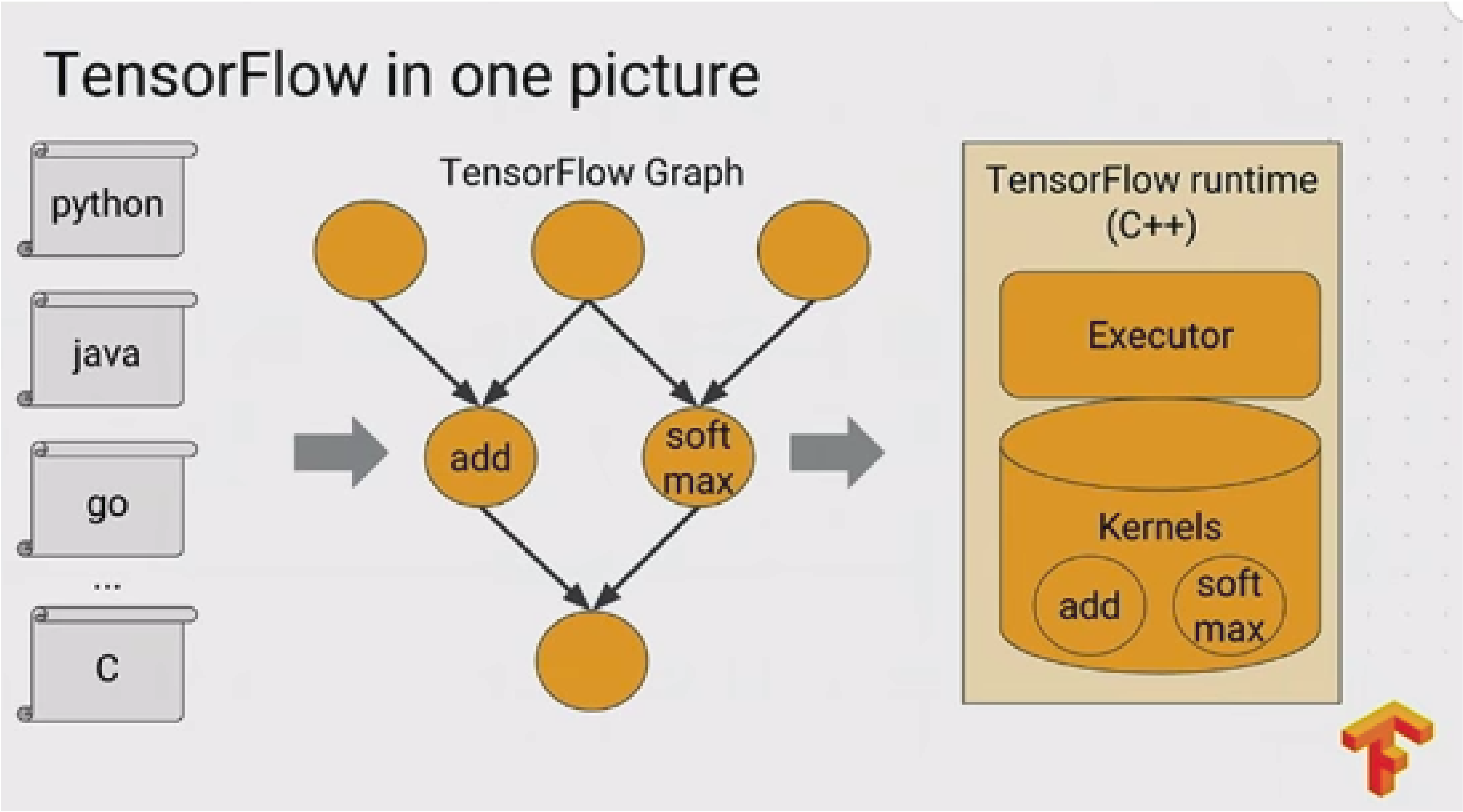
Device Placement



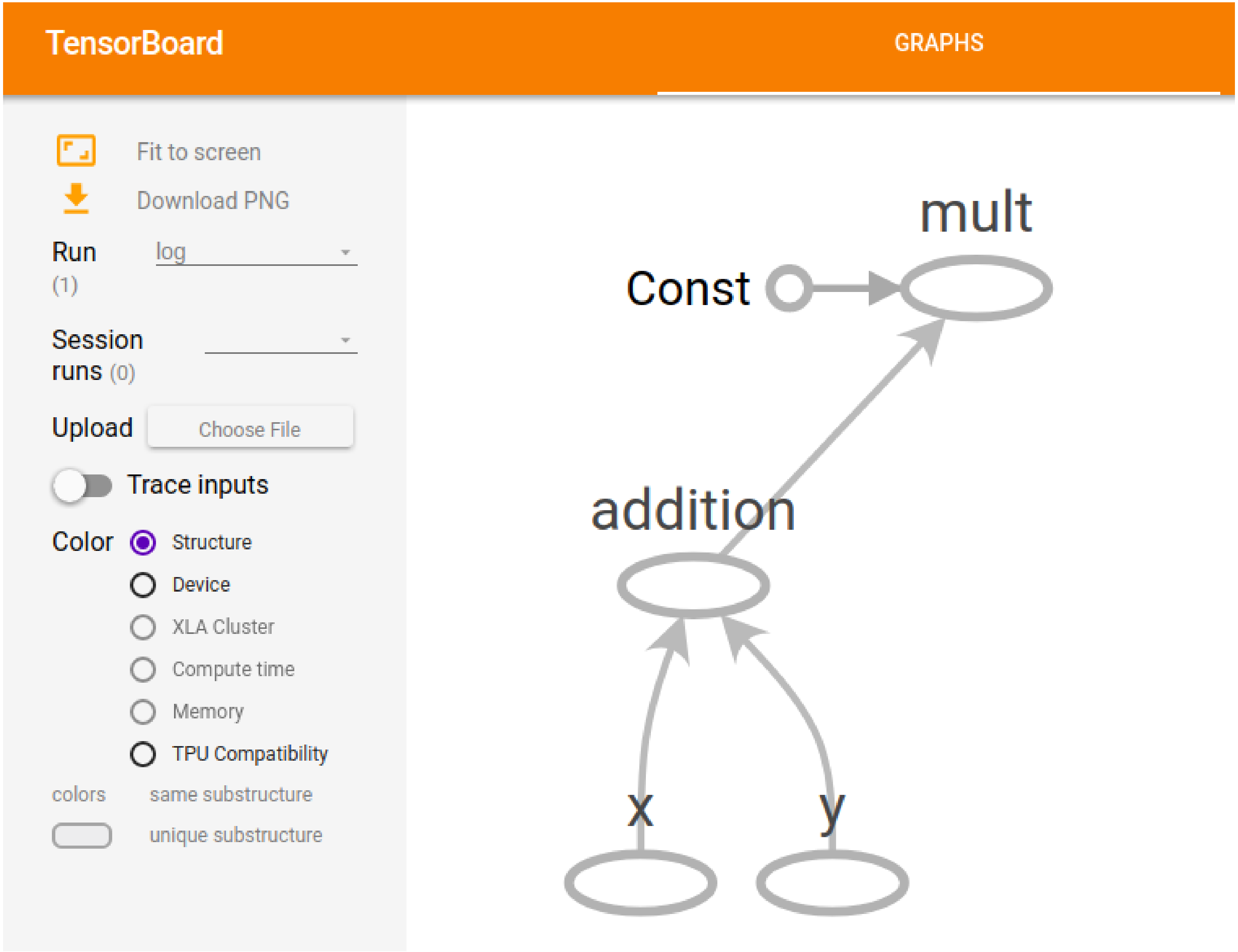
Code 3: Addition of tf.device

```
import tensorflow as tf

with tf.device('/cpu:0'):
   x = tf.placeholder(tf.float32, [None, 784])
with tf.device('/gpu:0'):
   W = tf.Variable(tf.zeros([784, 10]))
with tf.device('/gpu:1'):
   b = tf.Variable(tf.zeros([10]))
with tf.device('/cpu:0'):
   y_1 = tf.matmul(x, W)
with tf.device('/cpu:0'):
   y_2 = tf.add(y_1,b)
with tf.device('/gpu:2'):
   y = tf.nn.softmax(y_2)
```

TensorFlow runtime



tensorboard



Conferences

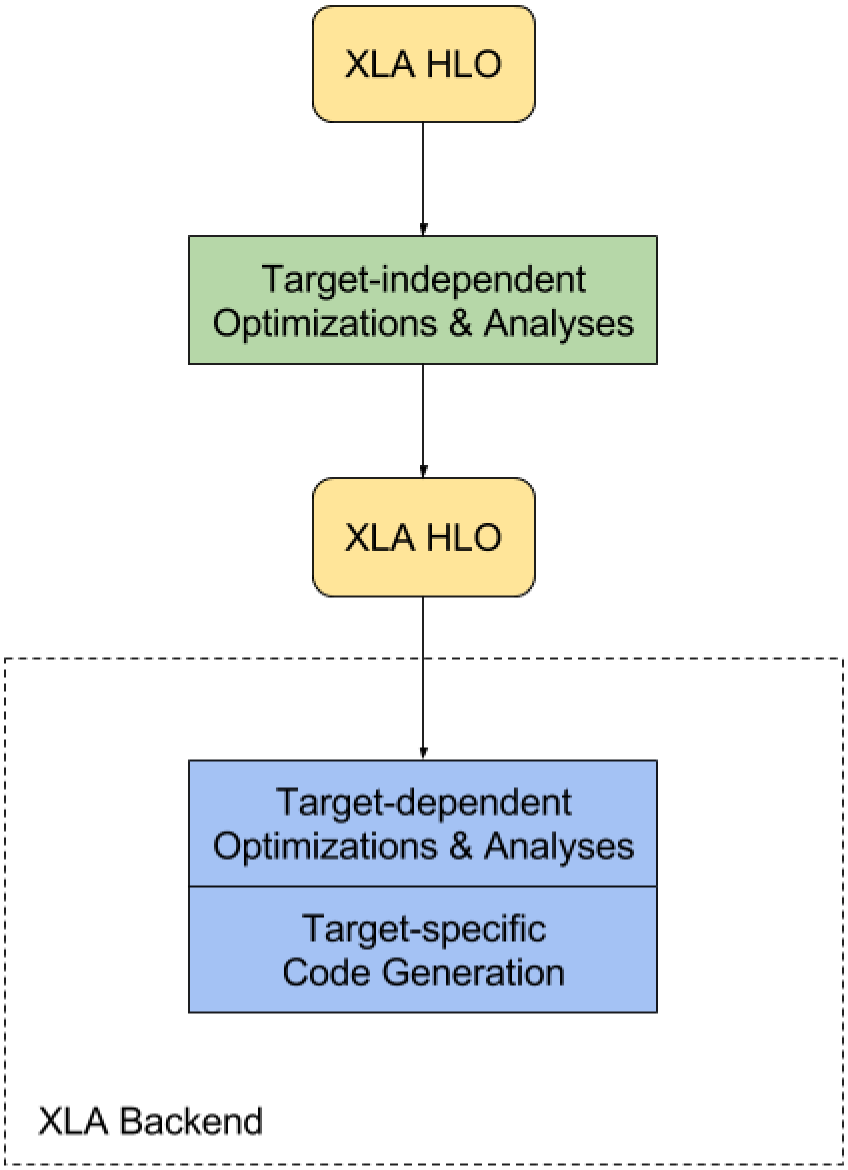
XLA

XLA (Accelerated Linear Algebra) is a domain-specific compiler for linear algebra that optimizes TensorFlow computations.

***OPTS**

- Improve execution speed
- Improve memory usage
- Reduce reliance on custom Ops
- Reduce mobile footprint.
- Improve portability

https://goo.gl/rAsyHB

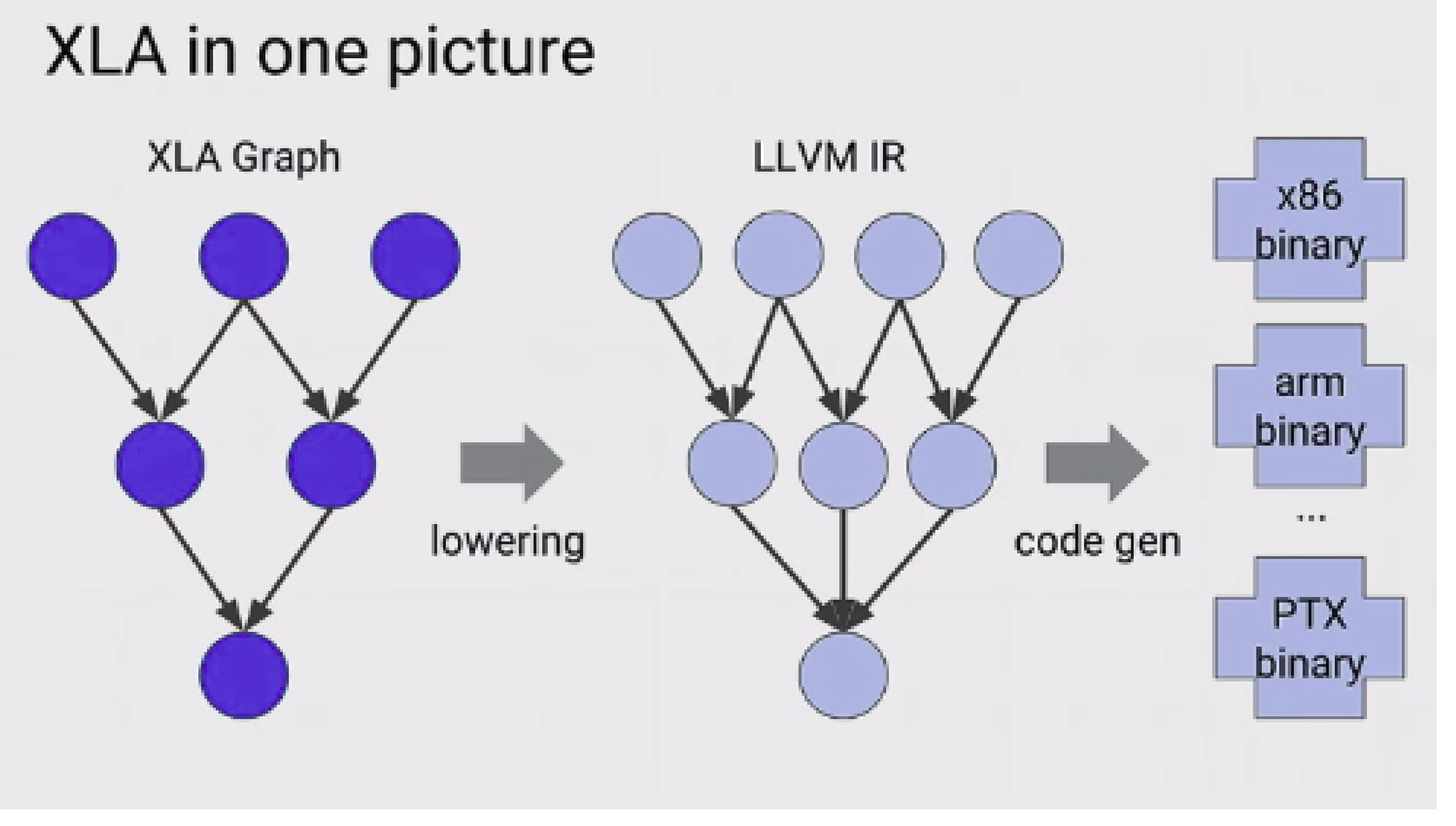


XLA

```
//tensorflow/compiler/xla

//tensorflow/compiler/tf2xla

//tensorflow/compiler/jit

//tensorflow/compiler/aot
```

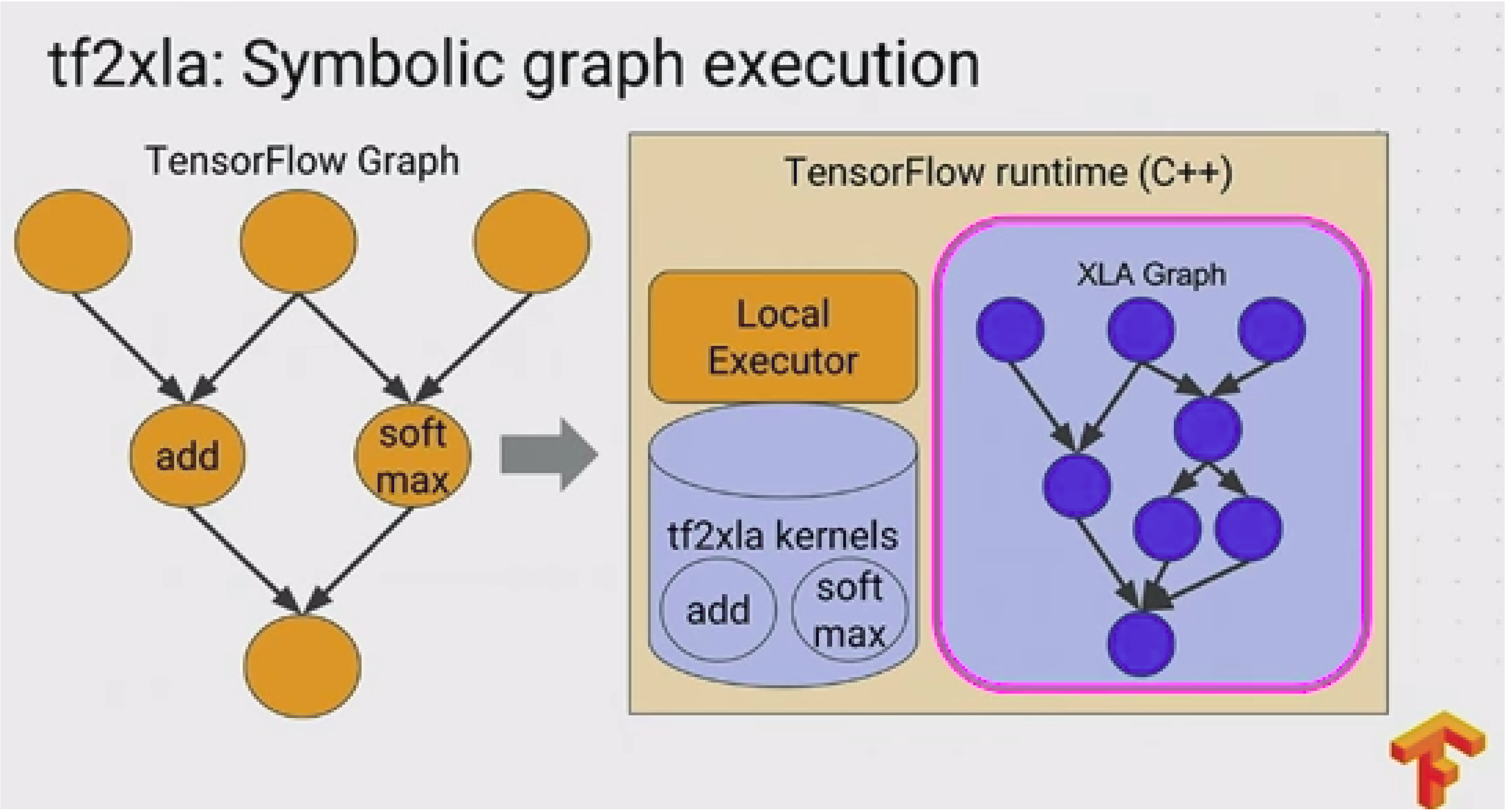
//tensorflow/compiler/xla

In general, XLA module is responsible to convert a given XLA Graph into binary by first lowering it into an LLVM IR, and after compiling the LLVM IR to given binary of a specific architecture (e.g. x86).



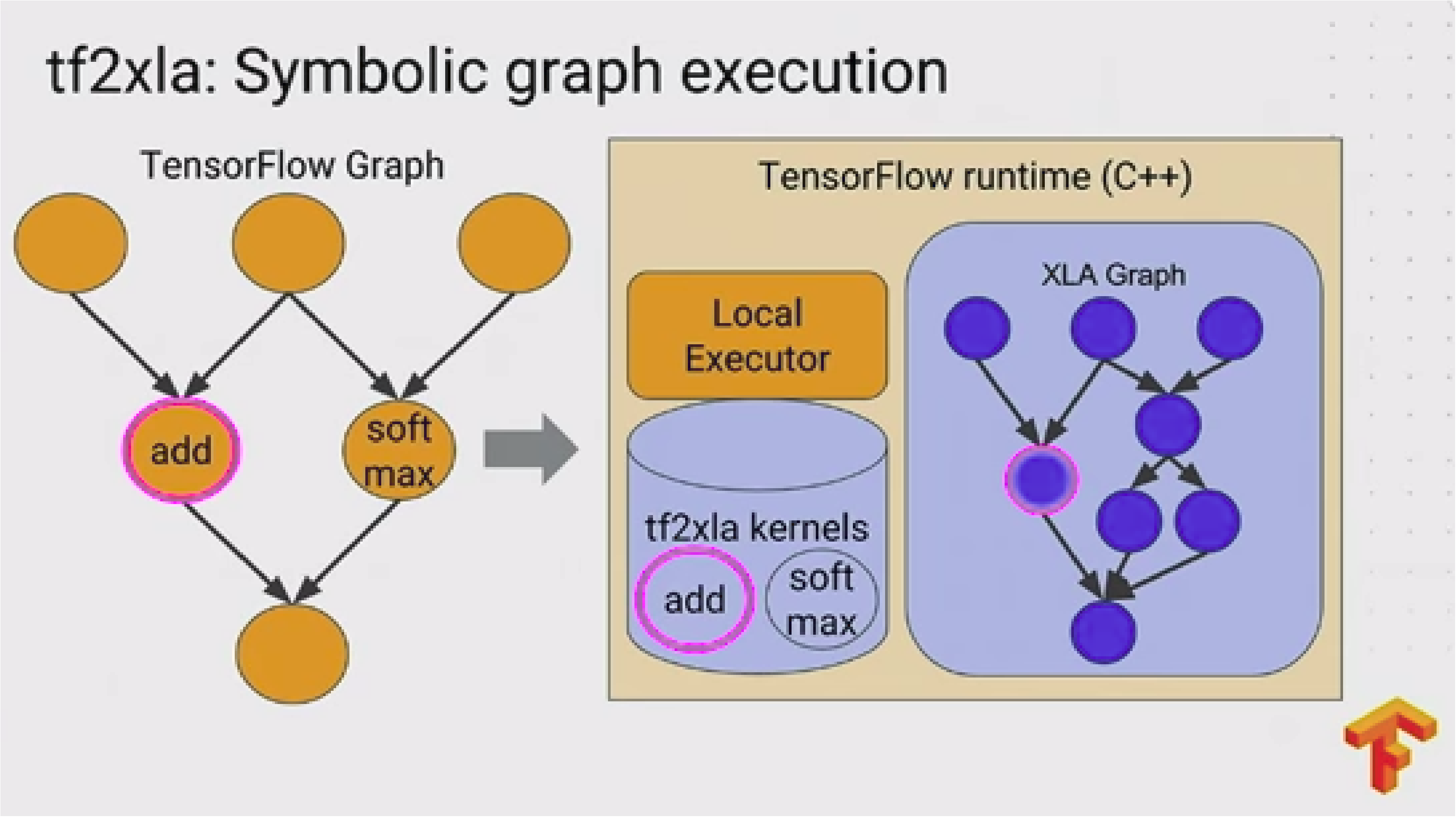
//tensorflow/compiler/tfa2xla

The tfa2xla module is responsible to convert an TFG into an XLA Graph. The example below convert the entire TFG into a XLA Graph (a common case when using AOT).



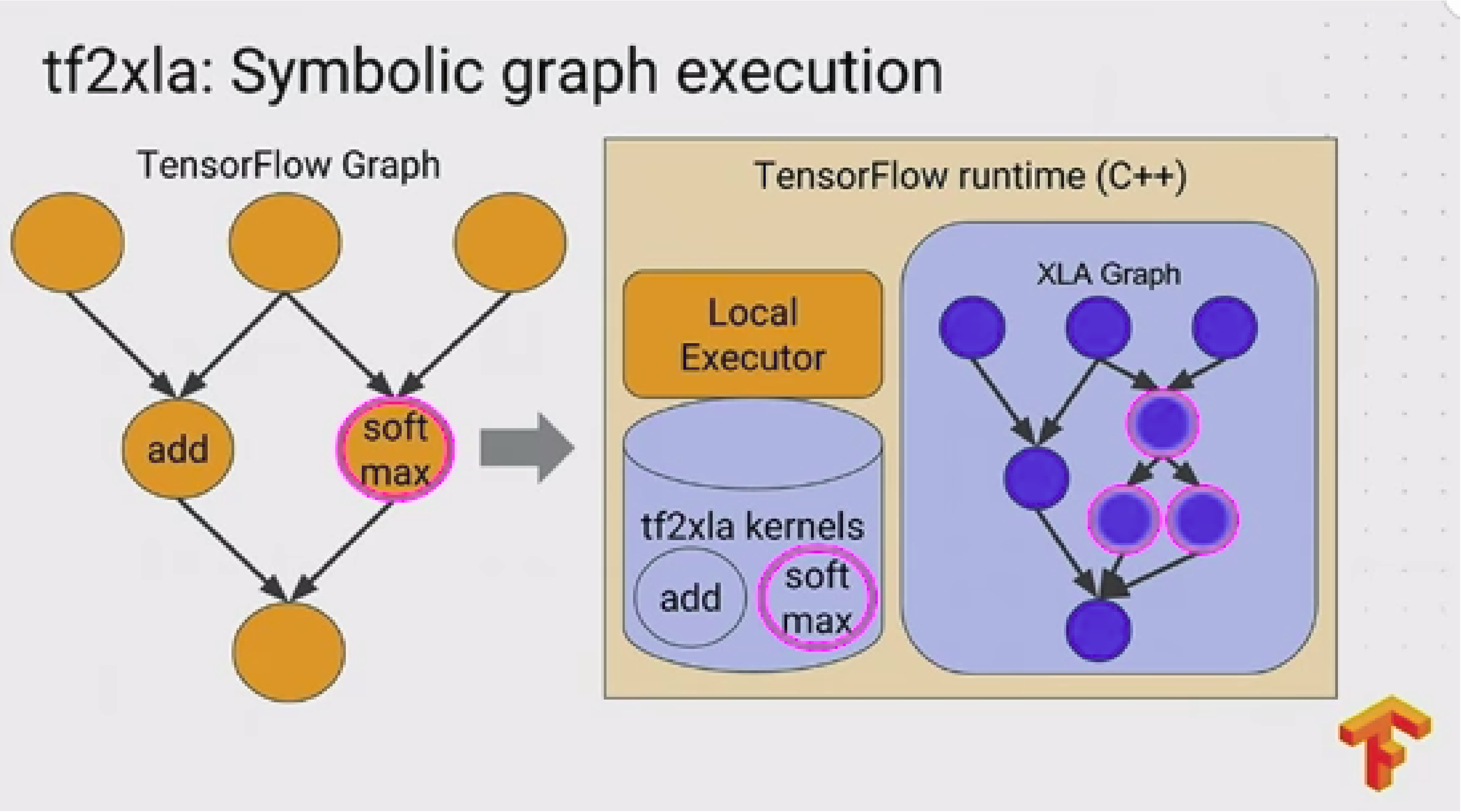
//tensorflow/compiler/tfa2xla

Each node of the TFG is converted into an XLA Graph's node. Observe that the TFG add node is converted into an XLA add node.

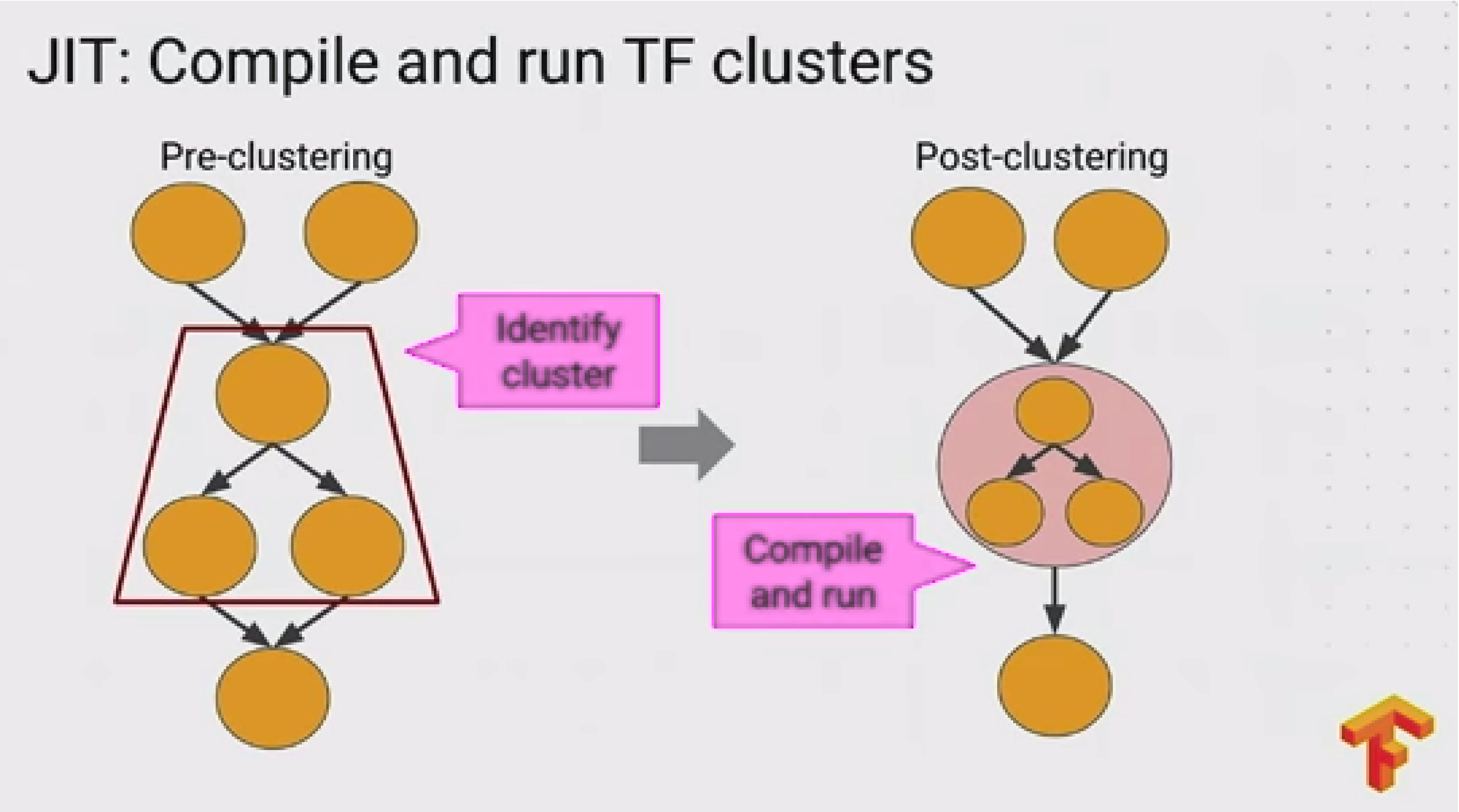


//tensorflow/compiler/tfa2xla

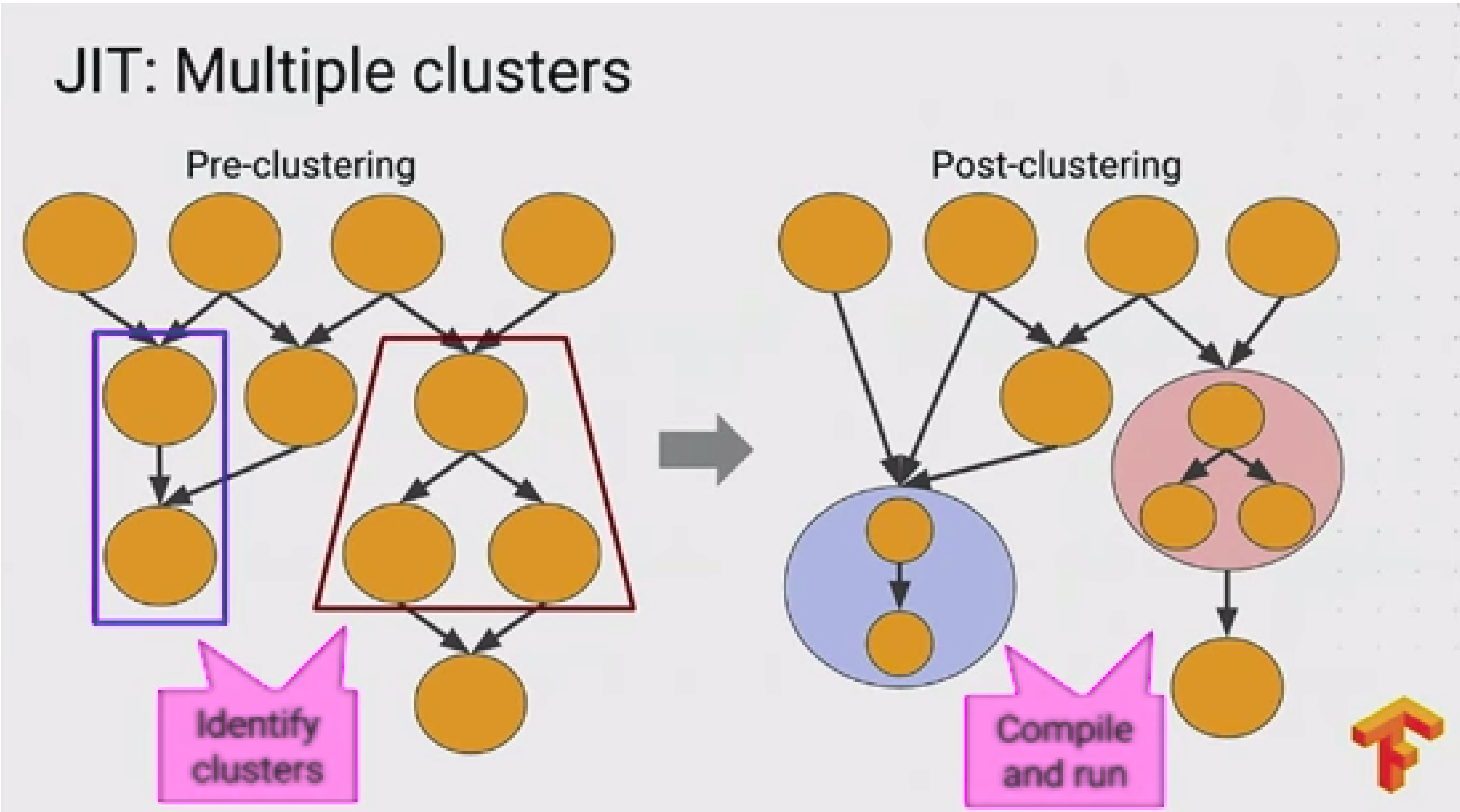
The TFG soft max node is converted into three XLA node.



//tensorflow/compiler/jit



//tensorflow/compiler/jit



//tensorflow/compiler/jit

Turning on JIT compilation

https://www.tensorflow.org/versions/master/experimental/xla/jit

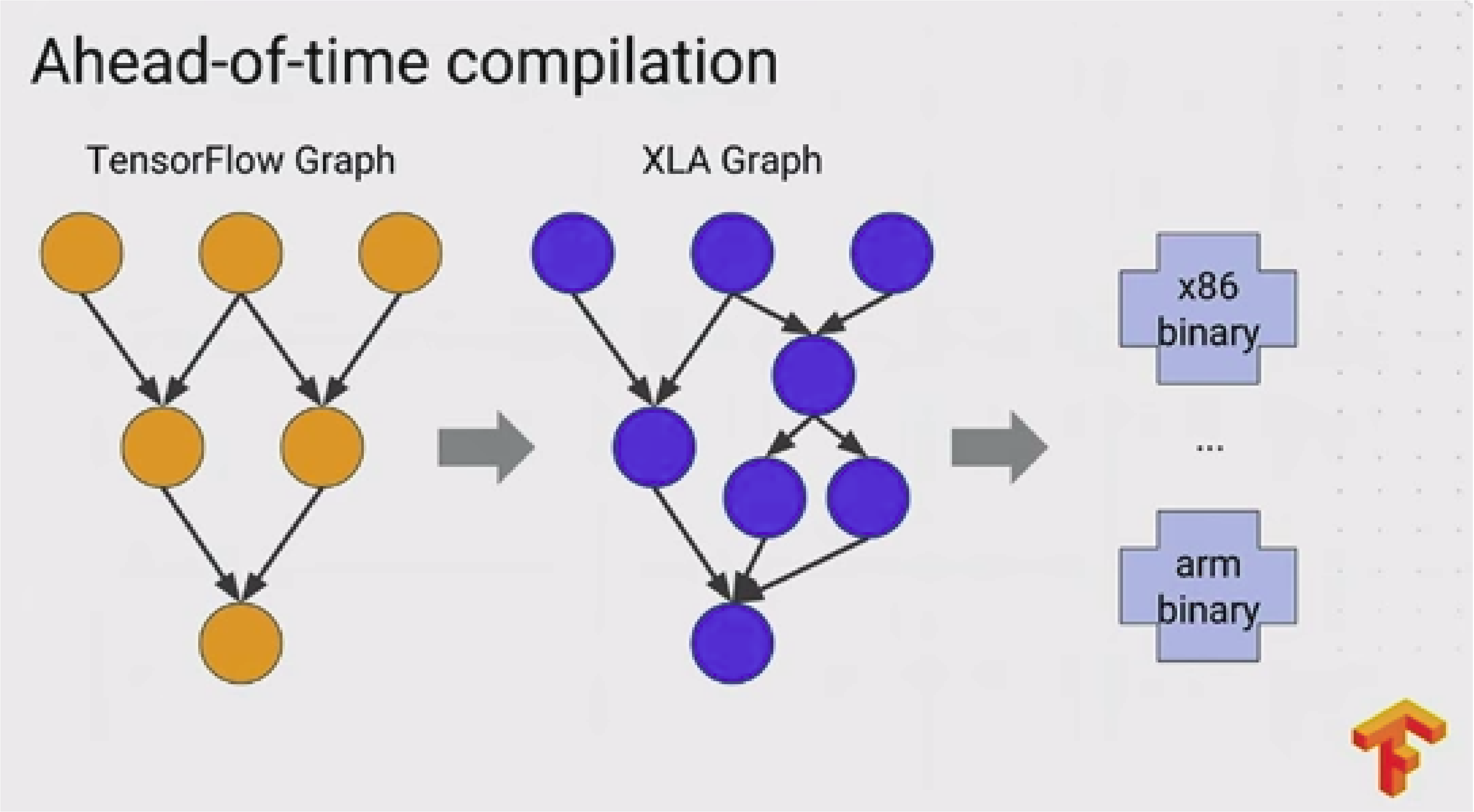
Whole session

```
config = tf.ConfigProto()
config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1
sess = tf.Session(config=config)  # All supported ops compiled with XLA.
```

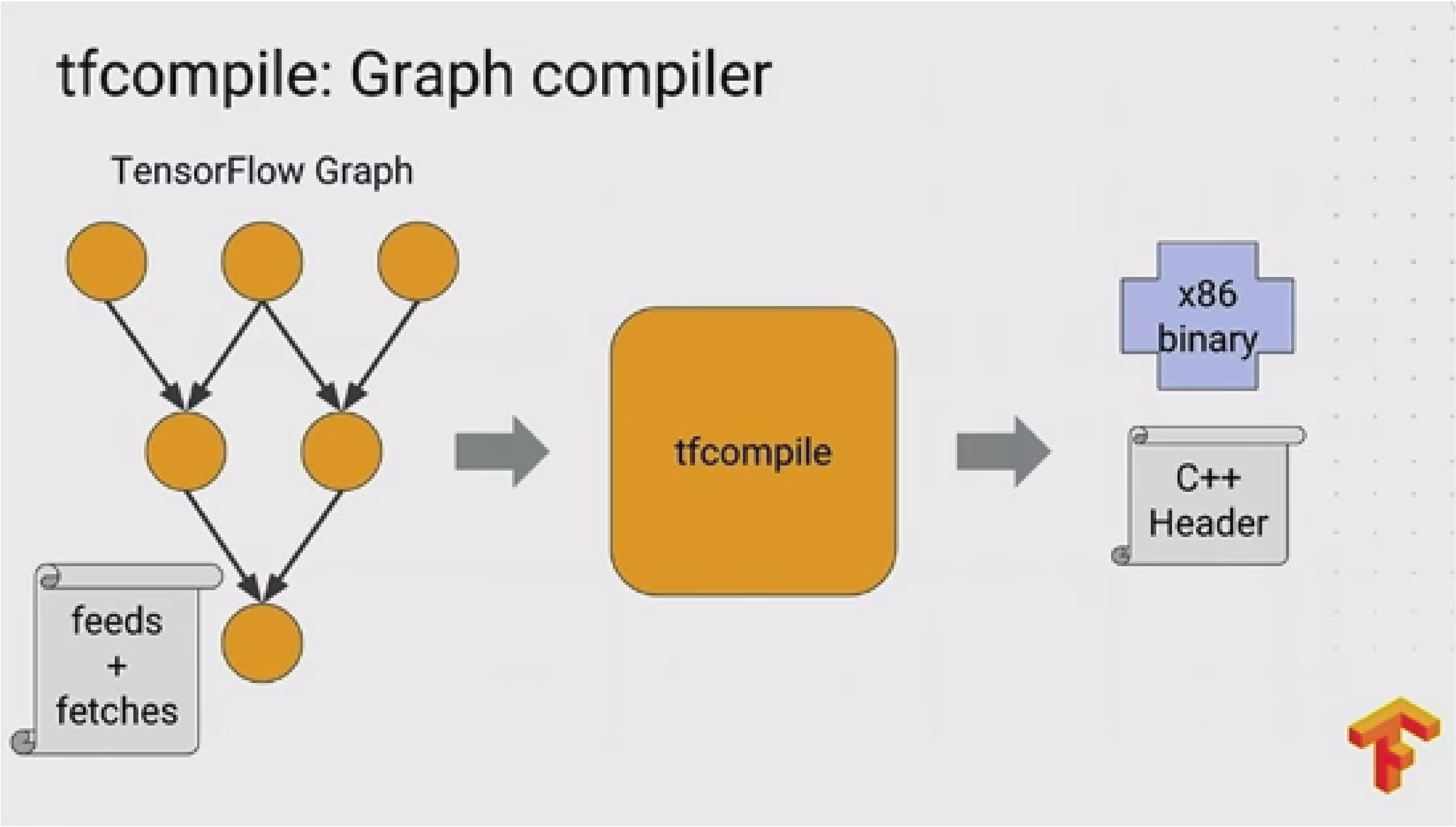
Manual scoped

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

//tensorflow/compiler/aot



//tensorflow/compiler/aot



//tensorflow/compiler/aot

tfcompile

Write code to call the computation:

```
#include "myproject/tests/test_matmul.h" // generated header

int main(int argc, char** argv) {
  foo::TestMatMul matmul;

  // 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 results
  CHECK_EQ(matmul.result0(0, 0), 58);
  return 0;
}
```

//tensorflow/compiler/aot

Smaller binaries on mobile

Binary size reduction on android-arm (stacked LSTM, 3 deep, 60 wide)

Original: 2.6MB (1MB runtime + 1.6MB graph)

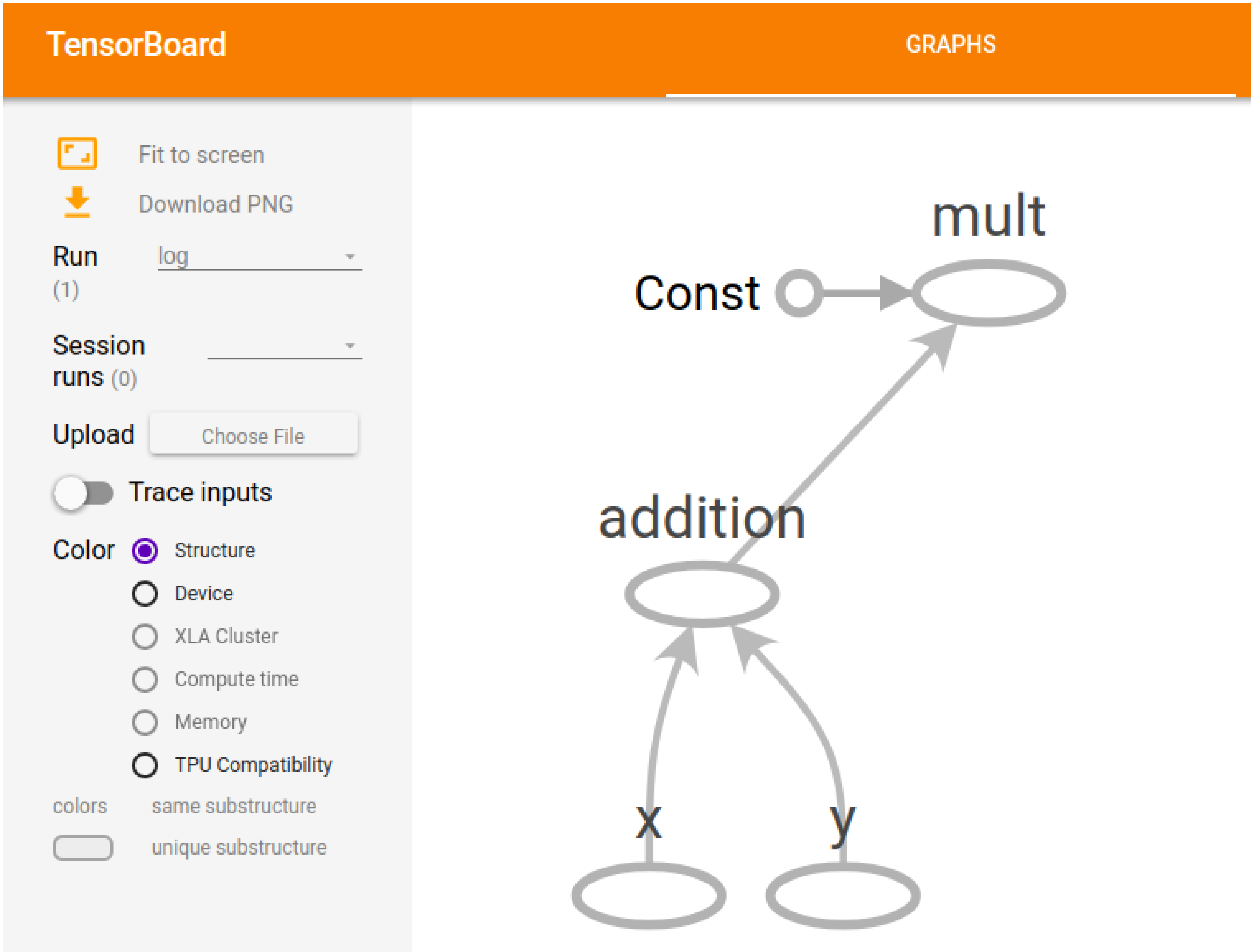
Compiled: **600KB** (**272KB code** + 330KB weights)

Example JIT

```python
import tensorflow as tf

jit_scope = tf.contrib.compiler.jit.experimental_jit_scope

X = tf.placeholder(tf.float32, name="x")
Y = tf.placeholder(tf.float32, name="y")
value = tf.constant(5, tf.float32)

addition = tf.add(X, Y, name="addition")

with jit_scope():
        mult = tf.multiply(addition, value, name ="mult")

session = tf.Session()
result = session.run(mult, feed_dict={X: [5,2,1], Y: [10,6,1]})

# writting the computational graph for tensorboard
writer = tf.summary.FileWriter("/home/rafael/Samsung/tests/log/", session.graph)

print(result)
```

Print --> [75. 40. 10.]

tensorboard

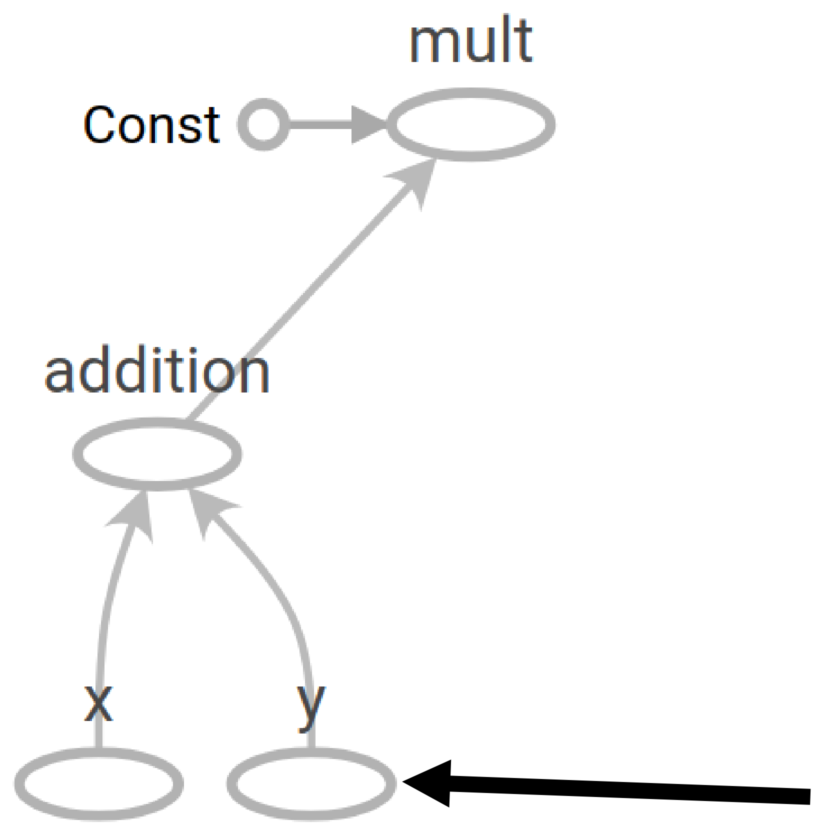


Example of Protobuf

```
node {
  name: "addition"
  op: "Add"
  input: "X"
  input: "Y"
 device: “CPU:0”
  attr {
    key: "T"
    value {
      type: DT_FLOAT
    }
  }
}
```

Execution -- *python example.py*

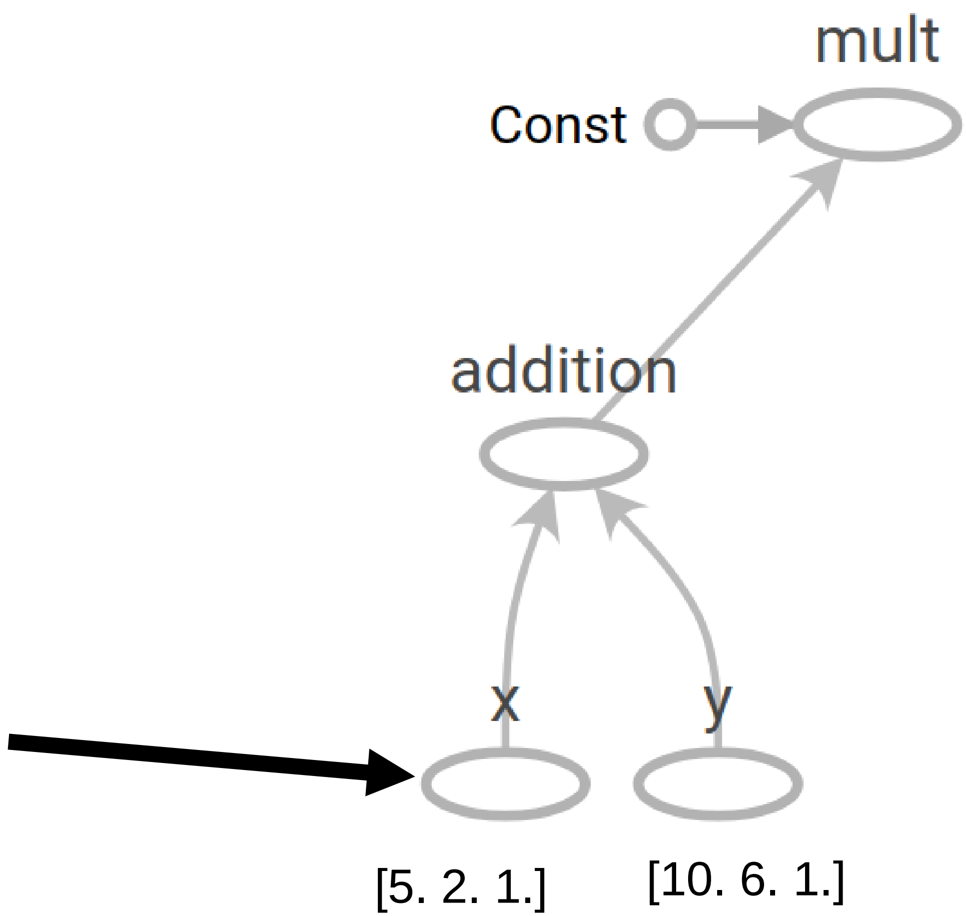
- Y = _Arg[T=DT_FLOAT, index=1, CPU:0"]() is dead: 0



[10. 6. 1.]

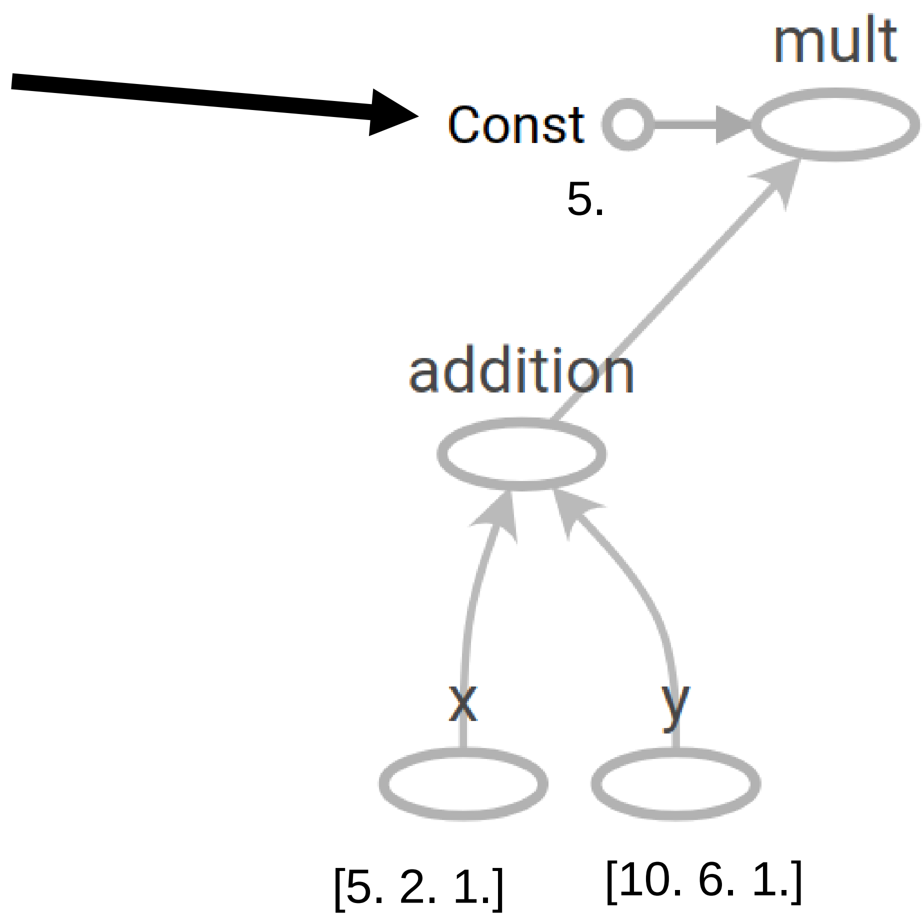
Execution -- *python example.py*

- X = _Arg[T=DT_FLOAT, index=0, CPU:0"]() is dead: 0



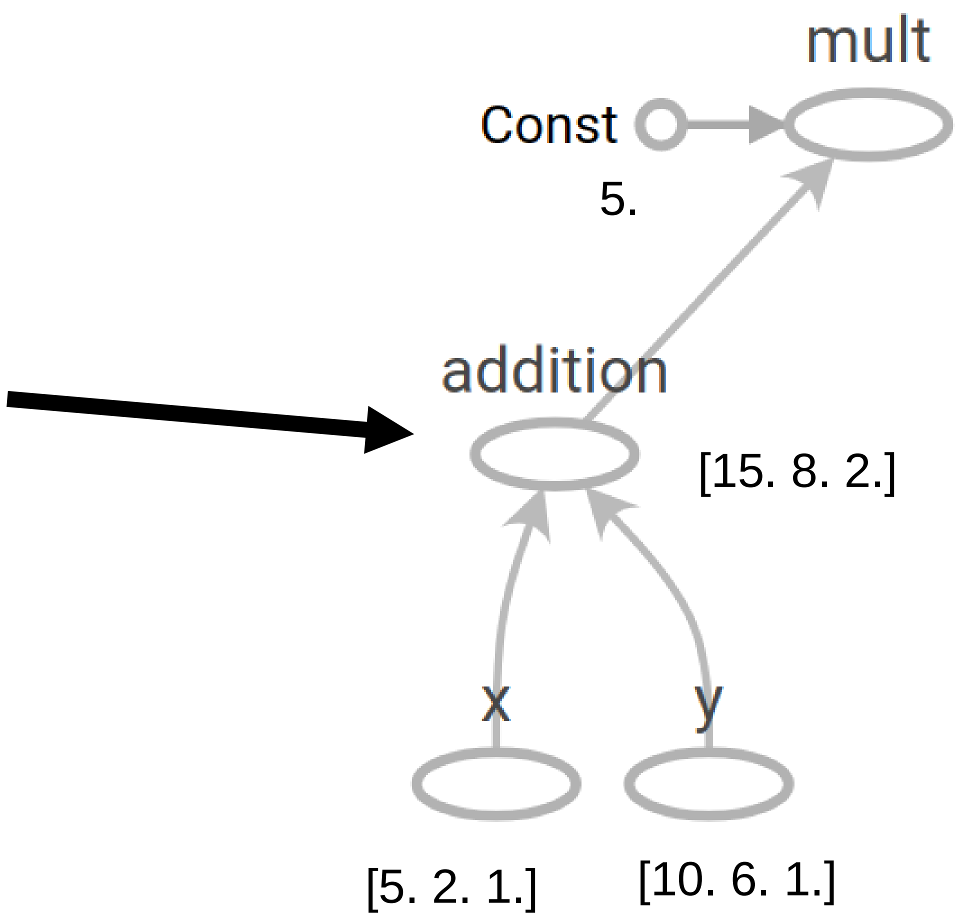
Execution -- *python example.py*

- Const = Const[dtype=DT_FLOAT, value=Tensor<type: float shape: [] values: 5>, CPU:0"]() is dead: 0



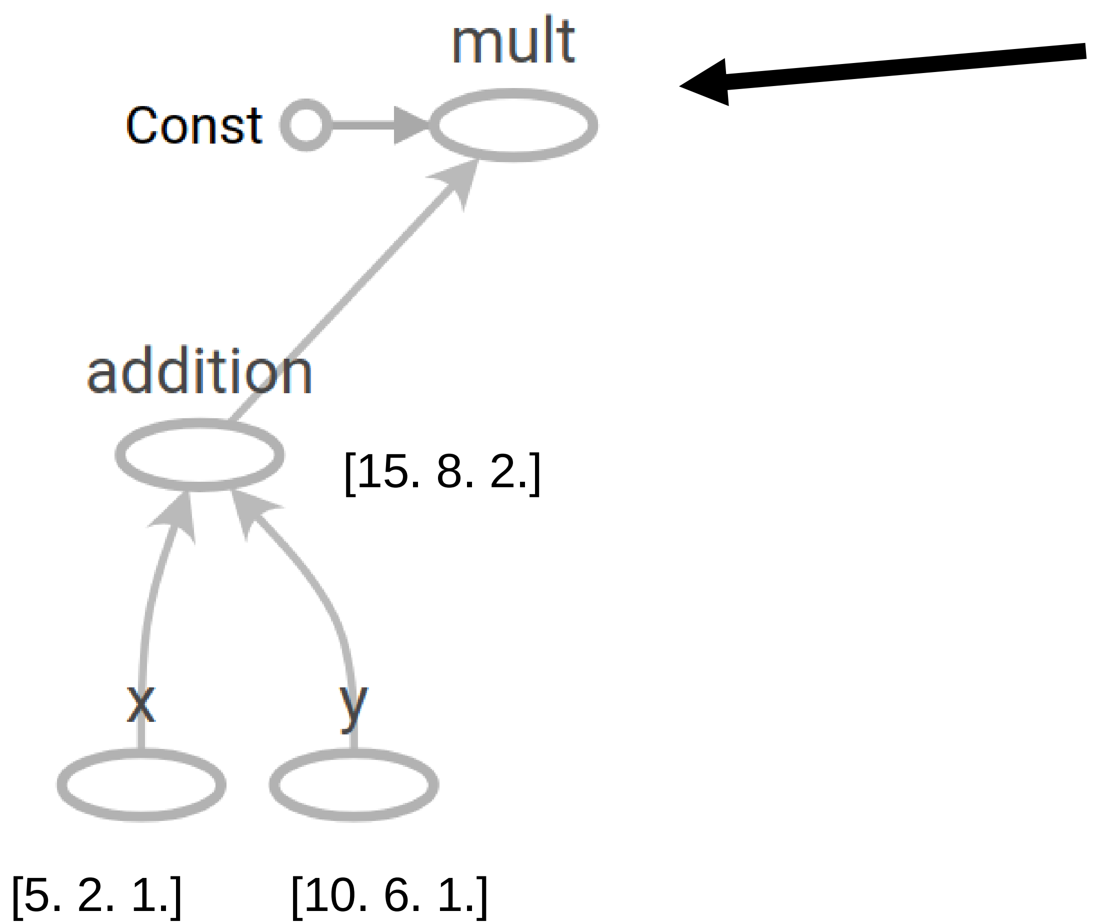
Execution -- *python example.py*

- addition = Add[T=DT_FLOAT, CPU:0"](X,Y) is dead: 0



Execution -- *python example.py*

- **[XLA]** mult = multiply[Targs=[DT_FLOAT, DT_FLOAT], Tresults=[DT_FLOAT], function=funcMult, _device="CPU:0"](addition, Const) is dead: 0

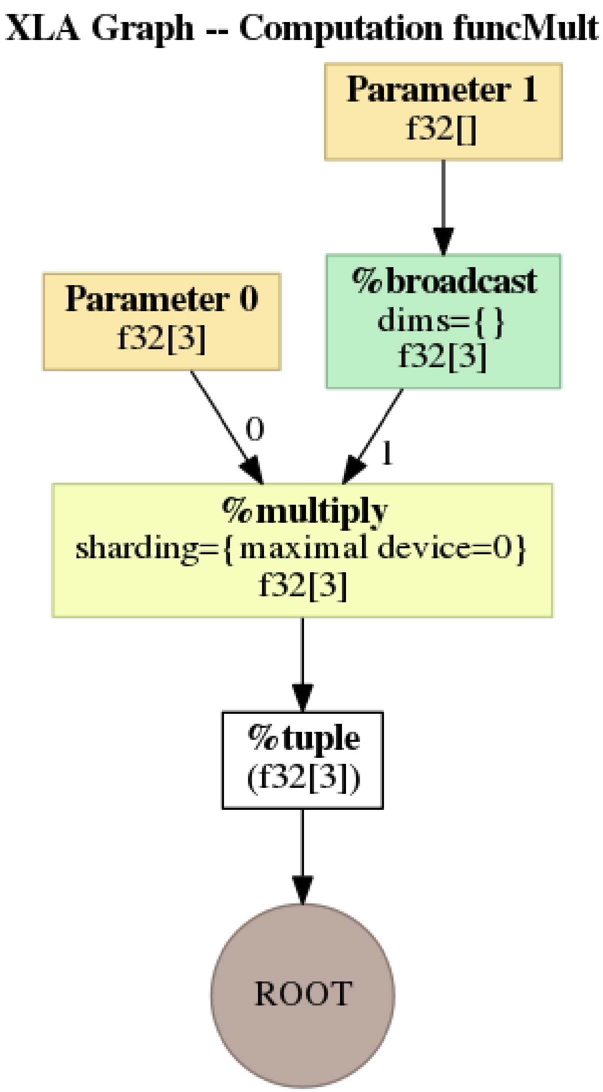


XLA compilation

- **XlaCompilationCache::Compile XLA JIT compilation cache**
  - Signature: funcMult ,float[3],float[]
  - num_inputs = 2
    - 0: dtype=float present=1 shape=[3]
    - 1: dtype=float present=1 shape=[]
  - num_outputs = 1
    - 0: dtype=1

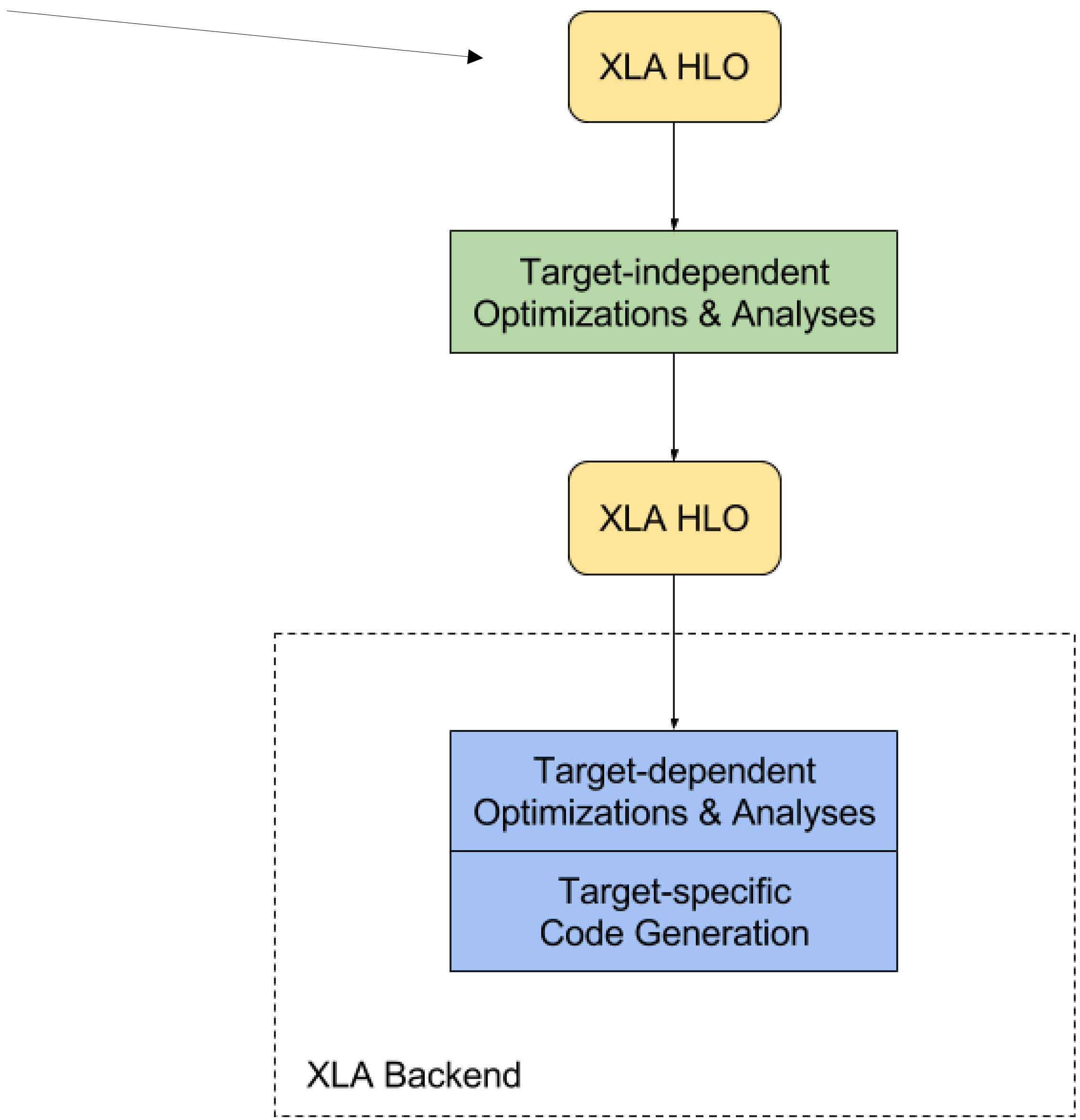
Compilation cache miss for the signature above!!

XLA Graph



XLA Graph -- Computation funcMult
Parameter 1
f32[]
Parameter 0
f32[3]
%broadcast
dims={}
f32[3]
0
1
%multiply
sharding={maximal device=0}
f32[3]
%tuple
(f32[3])
ROOT

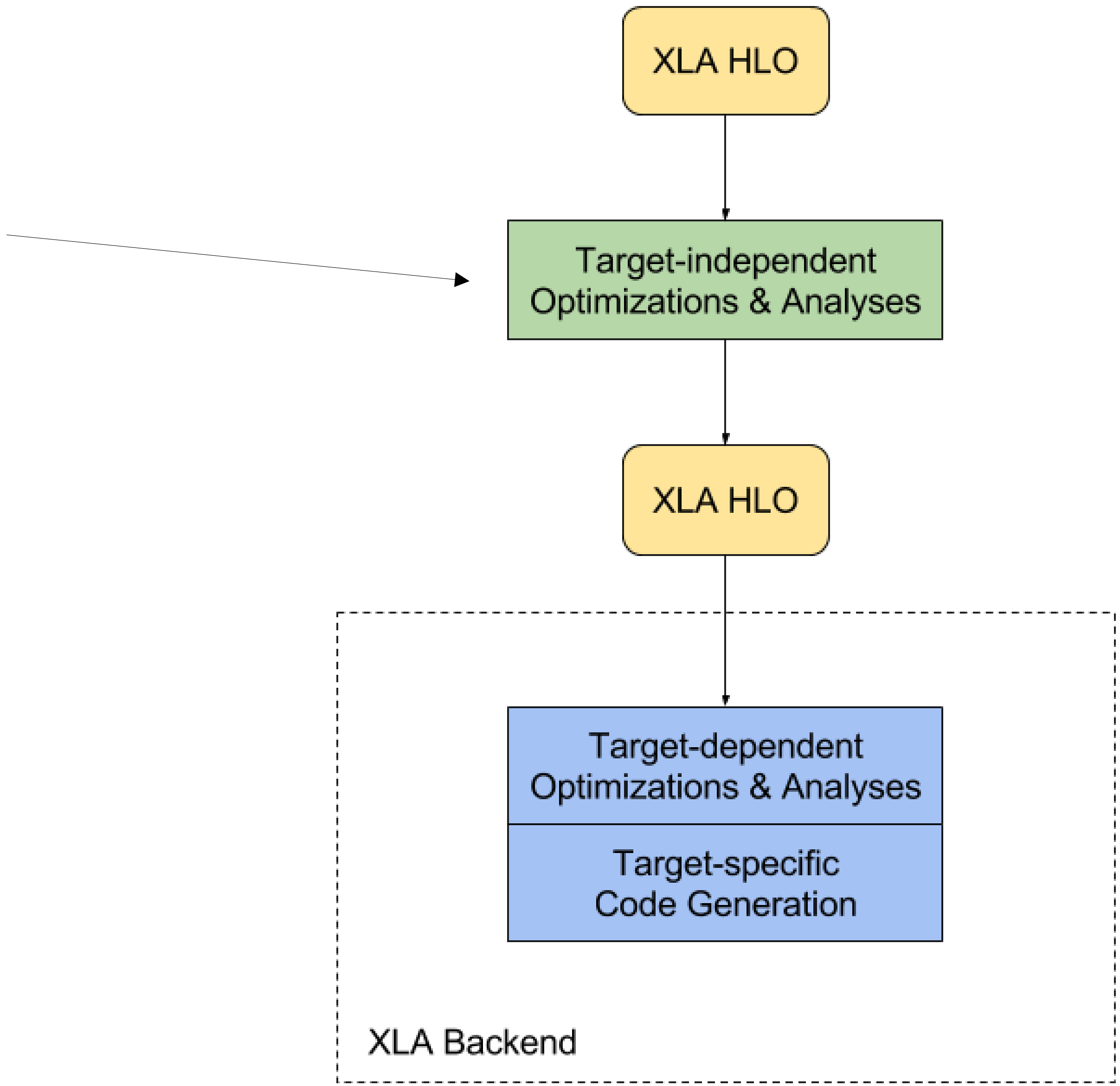
XLA compilation



HLO IR

```
HloModule funcMult:

ENTRY
%cluster_0[_XlaCompiledKernel=true,_XlaNumConstantArgs=0,_XlaNumResourceArgs=0].v4
(arg0: f32[3], arg1: f32[]) -> (f32[3]) {
  %arg0 = f32[3]{0} parameter(0)
  %arg1 = f32[] parameter(1)
  %broadcast = f32[3]{0} broadcast(f32[] %arg1), dimensions={}
  %multiply = f32[3]{0} multiply(f32[3]{0} %arg0, f32[3]{0} %broadcast), sharding={maximal
device=0} # metadata=op_type: "Mul" op_name: "mult"
  ROOT %tuple = (f32[3]{0}) tuple(f32[3]{0} %multiply)
}
```

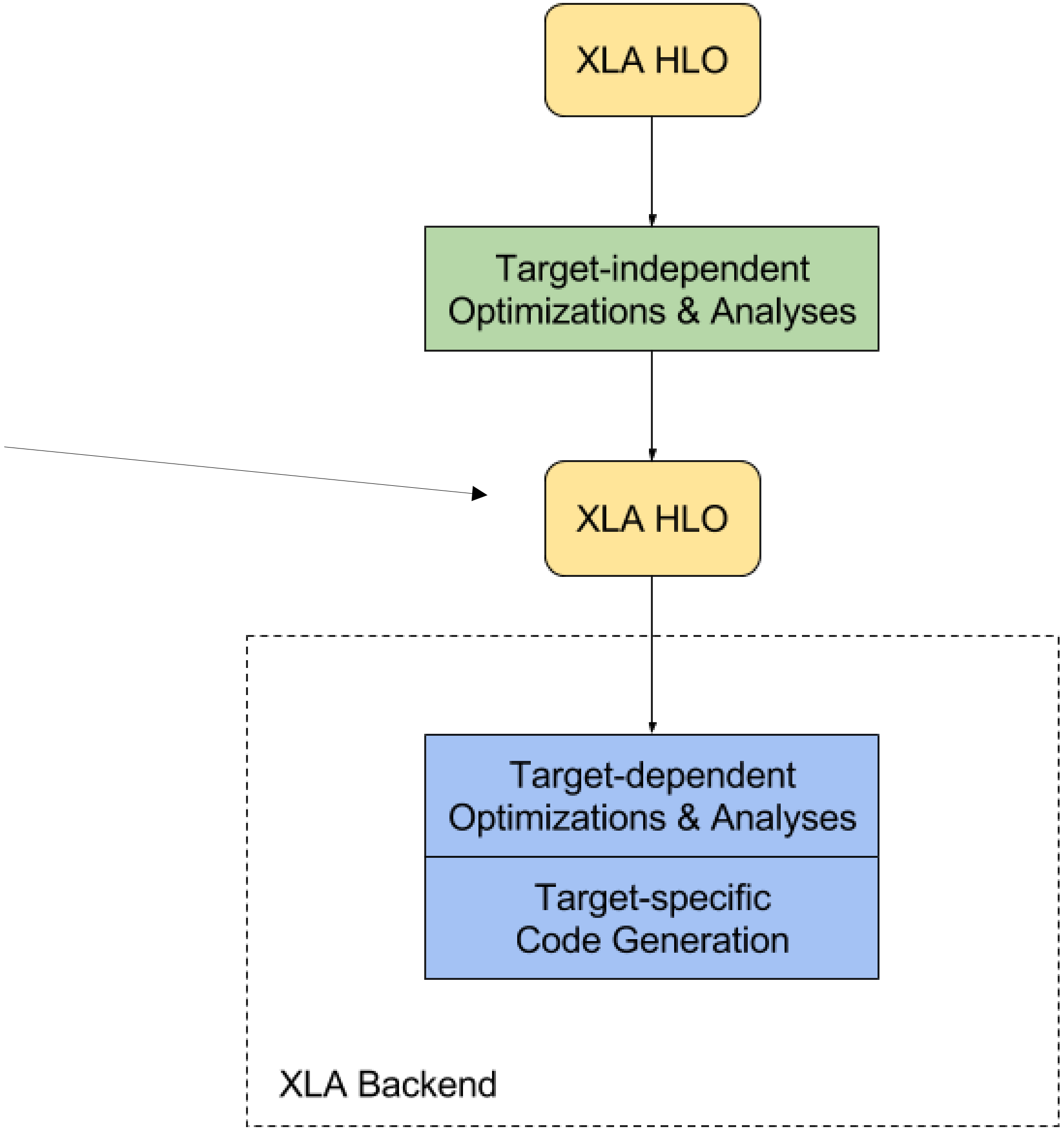
XLA compilation



XLA compilation -- OPT

- HLO pass pipeline CPU
- HLO pass CallInliner
- HLO pass convolution-canonicalization
- HLO pass simplification
- HLO pass batchnorm_rewriter
- HLO pass algsimp
- HLO pass tuple-simplifier
- HLO pass dce
- HLO pass reshape-mover
- HLO pass constant_folding
- HLO pass transpose-folding
- HLO pass cse
- HLO pass fusion
- HLO pass layout-assignment
- HLO pass algsimp
- HLO pass cse
- HLO pass cpu-parallel-task-assigner
- HLO pass dce
- HLO pass copy-insertion
- HLO pass dce
- HLO pass flatten-call-graph

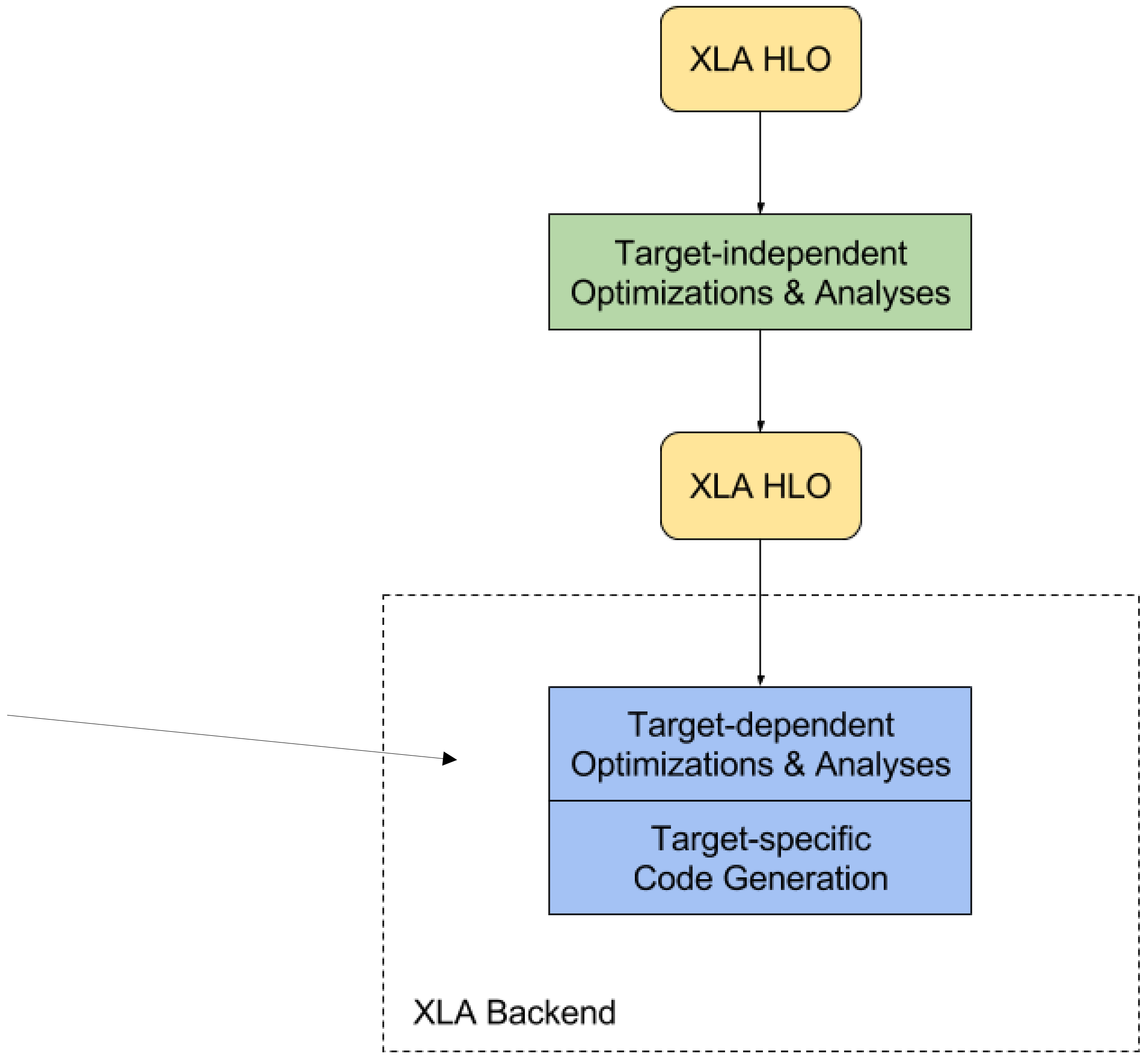
XLA compilation



HLO IR

```
HloModule funcMult:

ENTRY
%cluster_0[_XlaCompiledKernel=true,_XlaNumConstantArgs=0,_XlaNumResourceArgs=
0].v4 (arg0: f32[3], arg1: f32[]) -> (f32[3]) {
  %arg0 = f32[3]{0} parameter(0)
  %arg1 = f32[] parameter(1)
  %fusion = f32[3]{0} fusion:kLoop(f32[3]{0} %arg0, f32[] %arg1), calls=
%fused_computation # metadata=op_type: "Mul" op_name: "mult"
    %fused_computation (arg0.param_0: f32[3], arg1.param_1: f32[]) -> f32[3] {
      %arg0.param_0 = f32[3]{0} parameter(0)
      %arg1.param_1 = f32[] parameter(1)
      %broadcast.1 = f32[3]{no layout} broadcast(f32[] %arg1.param_1), dimensions={}
      ROOT %multiply.1 = f32[3]{0} multiply(f32[3]{0} %arg0.param_0, f32[3]{no layout}
%broadcast.1), sharding={maximal device=0} # metadata=op_type: "Mul" op_name:
"mult"
    }
  ROOT %tuple = (f32[3]{0}) tuple(f32[3]{0} %fusion)
}
```

XLA compilation



XLA compilation --LLVM IR

```
; ModuleID = '__compute_module'
source_filename = "__compute_module"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux_gnu"

define void @multFunc(i8* align 8 dereferenceable(8) %retval, i8* noalias %run_options, i8** noalias %params, i8** noalias %temps, i64* noalias
%prof_counters) #0 {
entry:
  %fusion.invar_address.dim.0 = alloca i64
  %0 = getelementptr inbounds i8*, i8** %params, i64 0
  %arg0.untyped = load i8*, i8** %0, !invariant.load !0, !dereferenceable !1, !align !2
  %1 = bitcast i8* %arg0.untyped to [3 x float]*
  %2 = getelementptr inbounds i8*, i8** %params, i64 1
  %arg1.untyped = load i8*, i8** %2, !invariant.load !0, !dereferenceable !3, !align !2
  %3 = bitcast i8* %arg1.untyped to float*
  %4 = getelementptr inbounds i8*, i8** %temps, i64 0
  %5 = load i8*, i8** %4, !invariant.load !0, !dereferenceable !1, !align !2
  %fusion = bitcast i8* %5 to [3 x float]*
  store i64 0, i64* %fusion.invar_address.dim.0
  br label %fusion.loop_header.dim.0

fusion.loop_header.dim.0:                    ; preds = %fusion.loop_body.dim.0, %entry
  %fusion.indvar.dim.0 = load i64, i64* %fusion.invar_address.dim.0
  %6 = icmp uge i64 %fusion.indvar.dim.0, 3
  br i1 %6, label %fusion.loop_exit.dim.0, label %fusion.loop_body.dim.0

fusion.loop_body.dim.0:                      ; preds = %fusion.loop_header.dim.0
  %7 = getelementptr inbounds [3 x float], [3 x float]* %1, i64 0, i64 %fusion.indvar.dim.0
  %8 = load float, float* %7, !invariant.load !0, !noalias !4
  %9 = load float, float* %3, !invariant.load !0, !noalias !4
  %10 = fmul fast float %8, %9
  %11 = getelementptr inbounds [3 x float], [3 x float]* %fusion, i64 0, i64 %fusion.indvar.dim.0
  store float %10, float* %11, !alias.scope !4, !noalias !7
  %invar.inc = add nuw nsw i64 %fusion.indvar.dim.0, 1
  store i64 %invar.inc, i64* %fusion.invar_address.dim.0
  br label %fusion.loop_header.dim.0

fusion.loop_exit.dim.0:                      ; preds = %fusion.loop_header.dim.0
  %tuple = bitcast i8* %retval to [1 x i8*]*
  %12 = getelementptr inbounds [1 x i8*], [1 x i8*]* %tuple, i64 0, i64 0
  %13 = bitcast [3 x float]* %fusion to i8*
  store i8* %13, i8** %12, !alias.scope !7, !noalias !4
  %prof_counter.computation = getelementptr i64, i64* %prof_counters, i64 0
  ret void
}
```

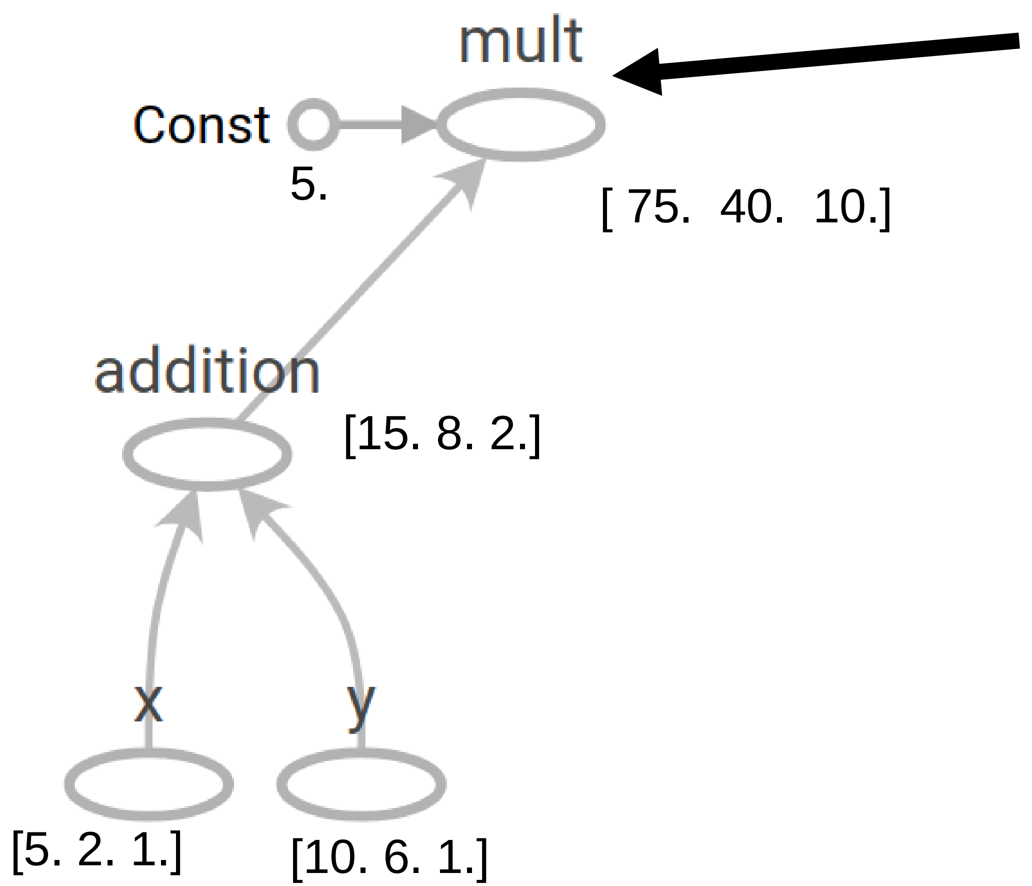
XLA compilation -- x86_64

```
0x00000000    mov     rax, qword ptr [rdx]
0x00000003    mov     rdx, qword ptr [rdx + 8]
0x00000007    mov     rcx, qword ptr [rcx]
0x0000000a    vmovss  xmm0, dword ptr [rdx]
0x0000000e    vmulss  xmm1, xmm0, dword ptr [rax]
0x00000012    vmovss  dword ptr [rcx], xmm1
0x00000016    vmulss  xmm1, xmm0, dword ptr [rax + 4]
0x0000001b    vmovss  dword ptr [rcx + 4], xmm1
0x00000020    vmulss  xmm0, xmm0, dword ptr [rax + 8]
0x00000025    vmovss  dword ptr [rcx + 8], xmm0
0x0000002a    mov     qword ptr [rdi], rcx
0x0000002d    ret
```

Execution -- *python example.py*

- **[XLA]** mult = multiply[Targs=[DT_FLOAT, DT_FLOAT], Tresults=[DT_FLOAT], function=funcMult, _device="CPU:0"](addition, Const) is dead: 0

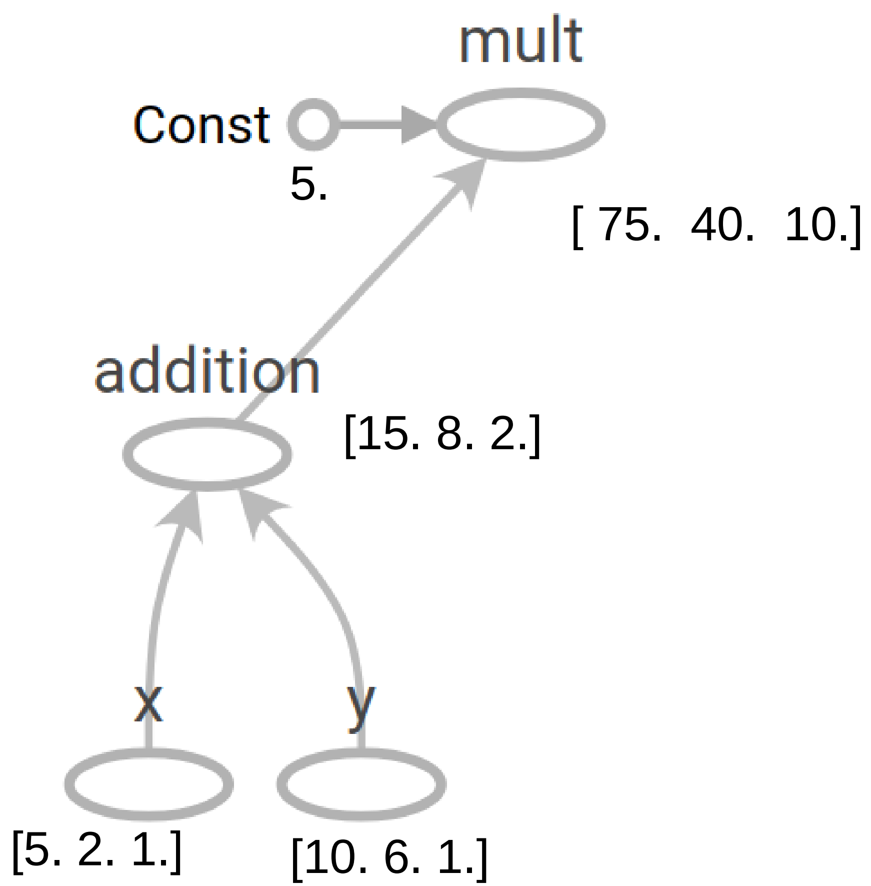
funcMult found



Execution -- *python example.py*

- retval(mult)

[75. 40. 10.]



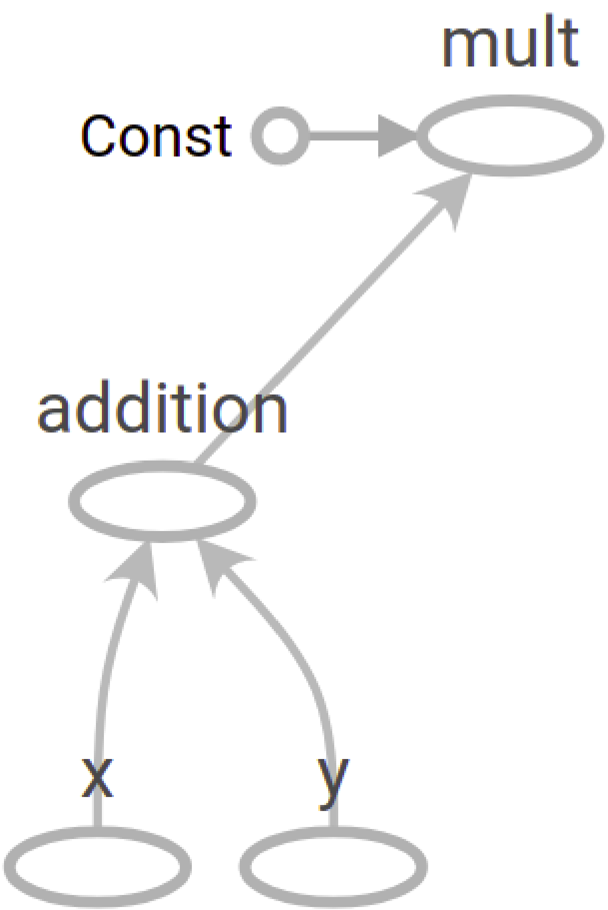
Example AOT

```
import tensorflow as tf

x_hold = tf.placeholder(tf.float32, name="x_hold")
y_hold = tf.placeholder(tf.float32, name="y_hold")
x_y_prod = tf.multiply(x_hold, y_hold, name ="x_y_prod")

session = tf.Session()
```

TensorFlow Graph

It is necessary to create the protobuf from the TensorFlow Graph

tfmatmul.pb



Feeds and Fetches

tfmatmul.config.pbtxt

```
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 }
  }
}

fetch {
  id { node_name: "x_y_prod" }
}
```

Invoke tfcompile

```
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 = "tfmatmul",
    cpp_class = "foo::bar::MatMulComp",
    graph = "tfmatmul.pb",
    config = "tfmatmul.config.pbtxt",
)
```

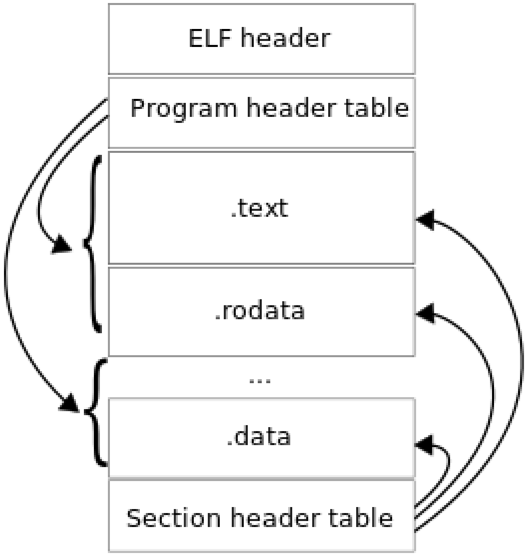
Invoke the tfcompile passing this script as parameter

tfcompile OPT

tfcompile makes use of XLA optimizations by converting the protobuf into HLO IR.

tfcompile results (header and binary)

```
namespace foo {
namespace bar {

class MatMulComp {
 public:
  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();

  bool Run();
  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);
  void** results();
  float* result0_data();
  float& result0(size_t dim0, size_t dim1);
};

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



tfcompile has as result one header file and one binary (e.g. x86).

Using the header and binary files

```
#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;
}
```

Compile the code linking the binary generated by tfcompile and execute.