


Swift for TensorFlow: **Graph Program Extraction**

Mingsheng Hong <hongm@google.com>
Chris Lattner <clattner@google.com>

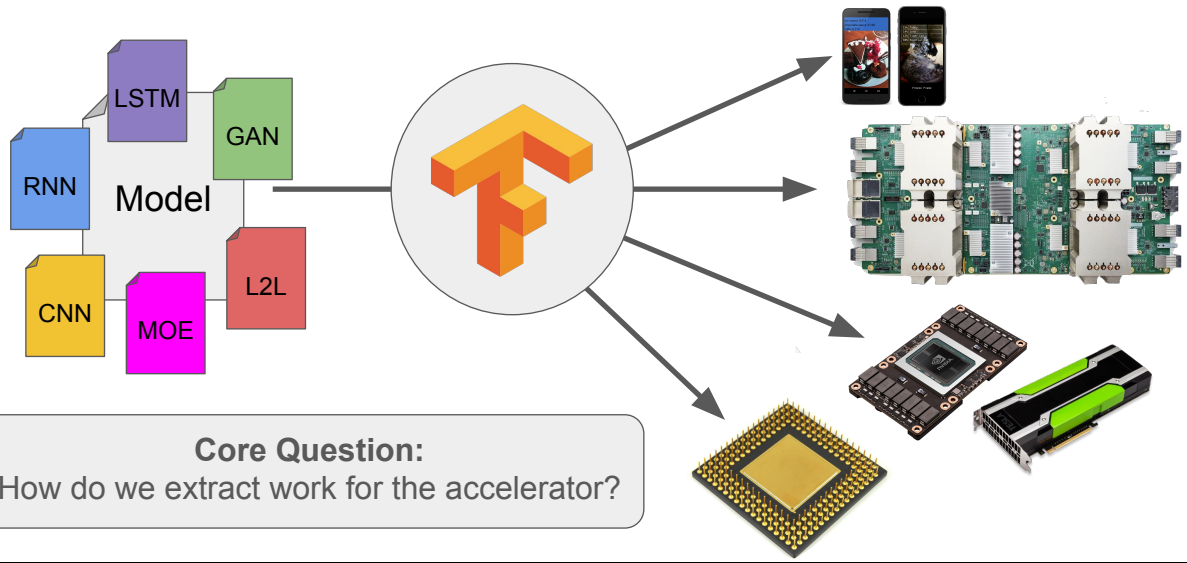
Presenting the work of many people!

Abstract (https://llvm.org/devmtg/2018-10/talk-abstracts.html#talk15)

Swift for Tensorflow (https://github.com/tensorflow/swift) is an Open Source project that provides a new way to develop machine learning models. It combines the usability/debuggability of imperative “define by run” programming models (like TensorFlow Eager and PyTorch) with the performance of TensorFlow session/XLA (graph compilation).

In this talk, we describe the design and implementation of deabstraction, Graph Program Extraction (GPE) and device partitioning used by Swift for TensorFlow. These algorithms rely on aggressive mid-level transformations that incorporate techniques including inlining, program slicing, interpretation, and advanced control flow analysis. While the initial application of these algorithms is to TensorFlow and machine learning, these algorithms may be applied to any domain that would benefit from an imperative definition of a computation graph, e.g. for high performance accelerators in other domains.

What are Machine Learning frameworks?



Through one reasonable lens, TensorFlow is a compiler. It processes machine learning models of various kinds, and supports targeting multiple kinds of high performance accelerators.

One major design question is how to represent the computation in the model, and how to extract it and execute it on an accelerator.

For the purposes of this talk, we'll explain things in terms of programming a single GPU, but our techniques generalize much more than that.

Approach #1: Eager Execution

```
x = ...
while tf.reduce_sum(x) < 100:
  a = tf.random_uniform(shape=[2, 2], maxval=100, dtype=tf.int32)
  b = tf.constant([[1, 2], [3, 4]], dtype=tf.int32)
  x = tf.nn.relu(x + a + b)
```

Usability: 👍

- Simple, easy, natural, flexible
- Error messages with sensible stack traces

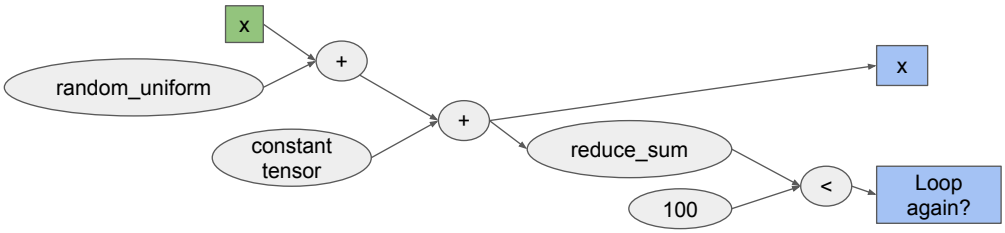
Performance: 👎

- Cross-op optimization (fusion, tiling, etc)
- Scalability to large accelerators

Eager execution is the simplest model, each call to a function kicks off a CUDA kernel that runs an accelerator as soon as each tensor method is executed. Python orchestrates those kernel launches, but the accelerator does the number crunching.

This is an obvious model that is easy for programmers to work with, but it turns out that you can get a lot of benefit from loop fusion and other standard compiler optimizations, and a simple eager execution mode makes this hard or impossible. This becomes a problem with very large accelerators, where you end up leaving them idle a lot.

Approach #2: Graph Building



Performance: 👍

- Graph level optimizations, scalability to large accelerators

Usability: 👎

- Awkward to stage control flow and side effects
- Dynamic models cannot be staged into a graph
- Error message quality (QoI)

APIs that build an explicit graph and execute it are another popular model. This has the benefit of supporting graph level optimizations (e.g. operation fusion), and can scale to support high performance accelerators. OTOH, these APIs are a lot more awkward to use (it is like using IRBuilder in your ML model) and can't represent general computation, and there is a trendline towards generality in the field.

Many other approaches

- Lightweight Modular Staging
- Tracing JITs
- Parse subsets of Python
- Hybrid approaches
- ...

How do we combine the usability of Eager mode with the performance and deployability of graphs?

The question though is how do we get the usability of eager with the performance of graphs? There are a bunch of other approaches people have been trying with lots of different tradeoffs, but they each provide different tradeoffs.

Swift for TensorFlow

http://github.com/tensorflow/swift

This is where Swift for TensorFlow comes in.

First-class language for machine learning

Designed for usability:

- Eager-style programming model
- Detect many errors without running code

Graph-based execution:

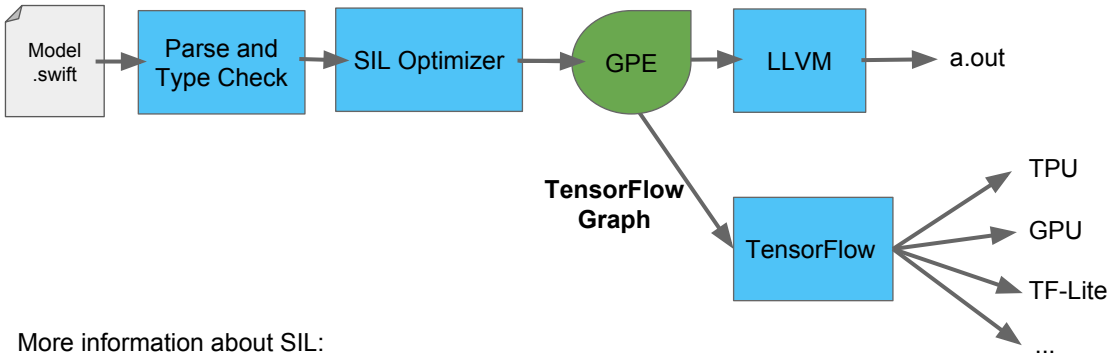
- Scalability and performance
- Deployment to mobile and servers

```
import TensorFlow
var x = Tensor<Float>([[1]])
for i in 1...5 {
    x += x • x
}
print(x)
```

http://github.com/tensorflow/swift

Swift for TensorFlow is a first-class language for machine learning. The entire idea of the project is to optimize for usability, even if it means making enhancements to the compiler and language. There are many aspects of this, but for this talk, we focus on this basic programming model. S4TF provides the usability of eager mode, combined with the performance of graphs.

How does this work?



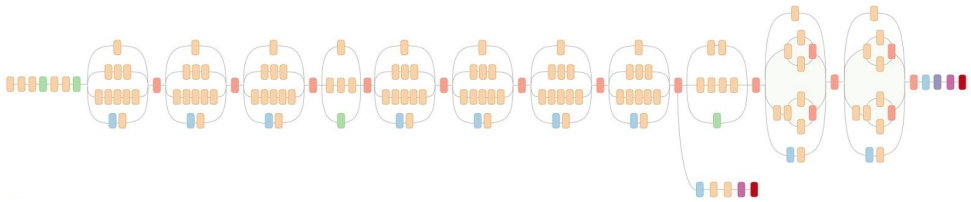
More information about SIL:
“Swift's High-Level IR”, LLVM Developer Meeting, Oct 2015

How does it work? This is a diagram of the Swift compiler, which includes a parser, typechecker and an optimizer for a high level IR called SIL. If you’d like to learn more about SIL, there was a talk a few years ago at the developer meeting.

One nice thing about this is that when you run code at -O0 mode, the ops are run one by one in tensorflow just like normal eager mode.

When the optimizer is turned on though, a technique called Graph Program Extraction extracts the tensor operations from the program and builds a tensorflow graph, fully automatically. Instead of hand waving about this, I’d like to invite my colleague Mingsheng up to talk about it now.

Graph Program Extraction



The exposition and examples are based on the GPE whitepaper https://github.com/tensorflow/swift/blob/master/docs/GraphProgramExtraction.md.

The technique has been implemented in the context of Swift as the host language, and tensorflow as the accelerator, but as you will see, the underlying design can be applied to other languages and accelerators as well.

An example program, and eager execution

```
func foo() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")  // invokes a TensorFlow operator

  if (...) { print("running") }  // arbitrary host computation

  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

(hand-off from Chris)
Thank you Chris. To show you how Graph Program Exaction works, let's take a look at this example program.
In this function, user first writes some tensor computation. Here we designate the magic tfop syntax to represent any operator that runs in TensorFlow.
User can write host logic, like printing some messages.
Control flows can also be used on tensor computation.

How do we run this program? One option is eager execution, as Chris introduced earlier. In that mode, we dispatch each tfop to tensorflow. When needed, we get the tensor results back to the host.

TensorFlow graph-based computation

```
func foo() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")
  if (...) { print("running") }
  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

<tensor computation in a graph>

Eager execution is simple and easy to work with. But for higher performance, we want to dispatch a larger chunk of tensor computation at once.

To do this, we need to extract a computational graph involving tensors, and dispatch the graph to tensorflow just like launching a GPU kernel. TensorFlow supports different device types. But for now, you can think of it as a GPU accelerator.

GPE: Clone Tensor ops into Graph Function

```
func foo() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")
  if (...) { print("running") }
  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

```
func foo_Graph() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")

  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

So let's look at what we'll get out of the graph program extraction.
We find those statements and control flows that can run in the graph, and move them over to the graph function we create. For high performance, we want to put control flow into the graph when possible.

GPE: Rewrite Host Function

```
func foo_Host() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")
  if (...) { print("running") }
  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

```
func foo_Graph() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")

  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

We then clean up the host code.

GPE: Launch and Rendezvous with Graph Function

```
func foo_Host() -> Tensor<Float> {
  let g = start_graph("foo_Graph")
  if (...) { print("running") }



  let w = wait_on_graph(g)
  return w
}
```

```
func foo_Graph() -> Tensor<Float> {
  var w = #tfop("RandomInitOp")

  for i in 0 ... 1000 {
    let x = #tfop("SomeOp", w)
    w = #tfop("AnotherOp", x)
  }

  return w
}
```

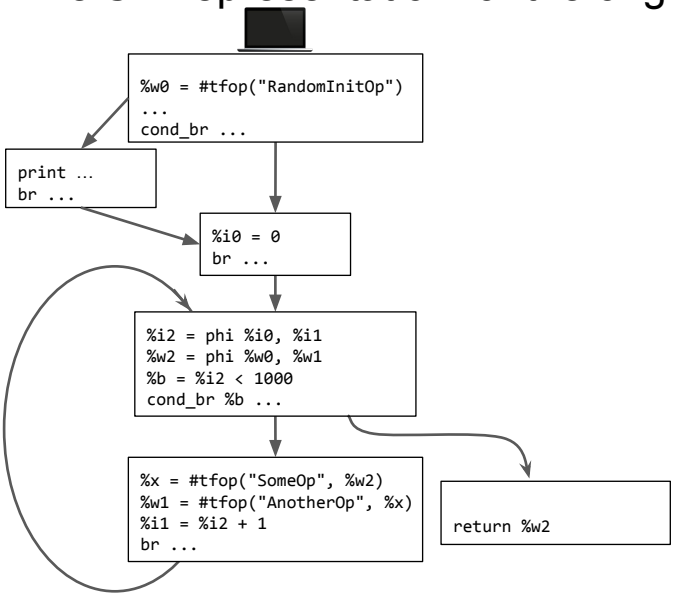
And we rewrite host code to asynchronously call into the graph function, as if we are launching a GPU kernel. The graph runs in TensorFlow, and that's how we accelerate tensor computation.

Algorithm Overview

- Marking tensor ops for graph execution
- Partition the host function into a pair of <host, graph> functions
- Lower the graph function to a TensorFlow graph, and rewrite host function to call into TensorFlow graph

We just described what graph program extraction can do to accelerate tensor computation. Now let's look at how it works. The workflow has 3 major steps.
First, we mark all tensor ops that we can later move to the graph.
Second, we partition the host function, to create a graph function. That's where the marked instructions go.
Last, we prepare the graph function for TensorFlow execution, and rewrite host code to call into the graph.

The SIL representation for the original code

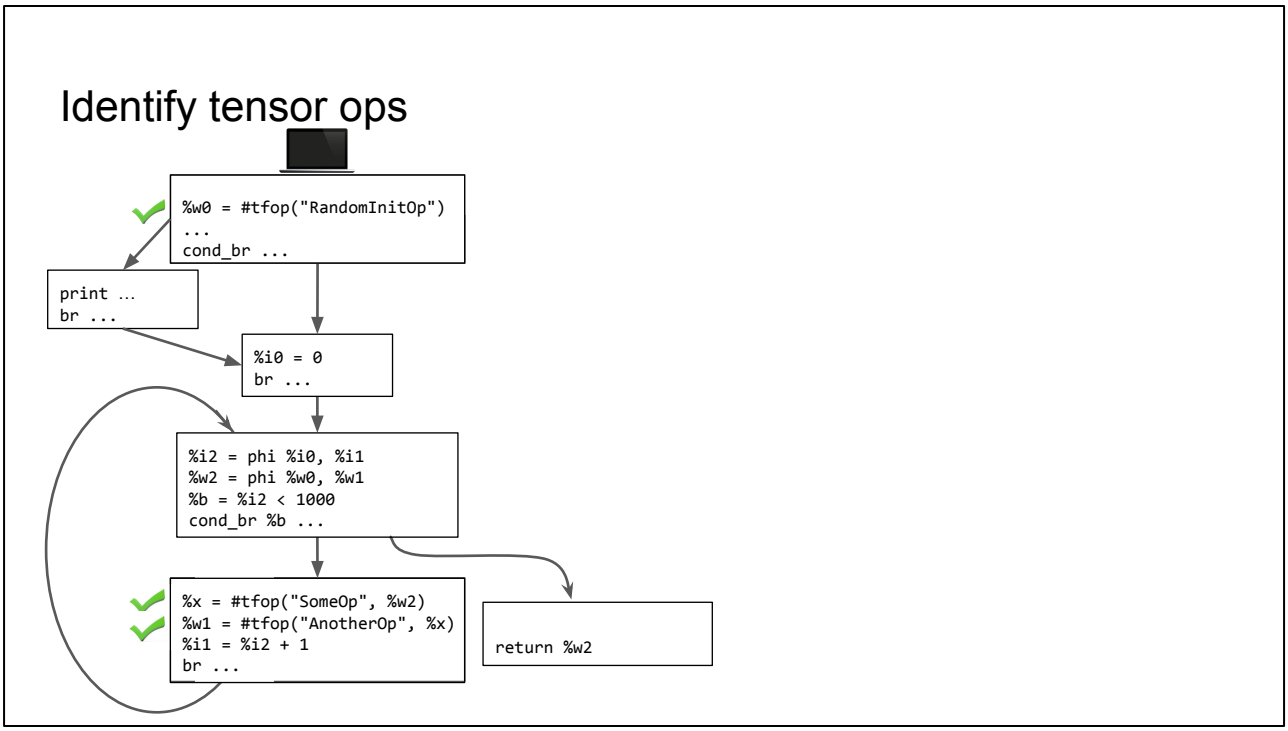


Our compiler analysis and transformation is done in the intermediate representation of Swift, called SIL. It's an SSA based IR.

Now I'll using the same example to walk you through the algorithms on the CFG representation.

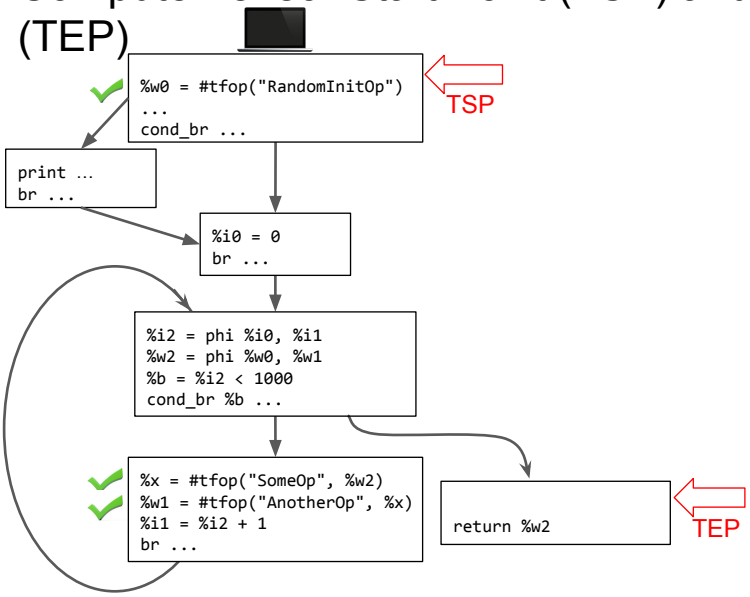
Here we listed the SIL code (with some simplifications) for the running example. It starts with some tensor code followed by an if condition to print message on the host. It then runs a loop, and returns the possibly updated tensor w.

(Note: for simplicity, we reuse #tfop to represent tensor ops, even though the correct SIL code would be graph_op. We want to avoid introducing that extra concept / complexity in this talk.)



In order to extract a graph of tensor computation, we first identify tensor ops. These ops must run on tensorflow. You can think of them as anchors in the graph function we are about to build.

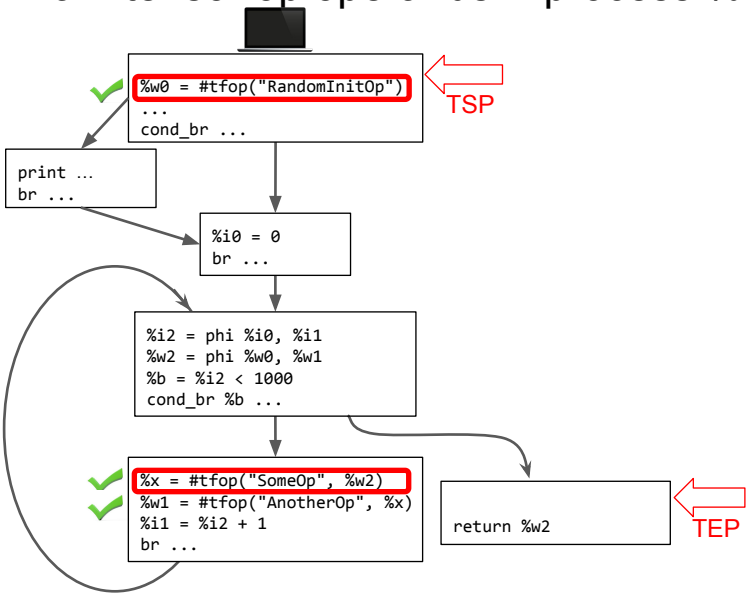
Compute Tensor Start Point (TSP) and Tensor End Point (TEP)



Next, we define TSP to be the first tfop instruction. We also define TEP. It's usually the last tfop instruction, but we also require that it post-dominate TSP. In this example, TEP is set to the first instruction in the return block.

TSP and TEP together set the boundaries of the tensor program region. From that region, we will extract the graph function.

Mark tensor op operands -- process %w0 and %x

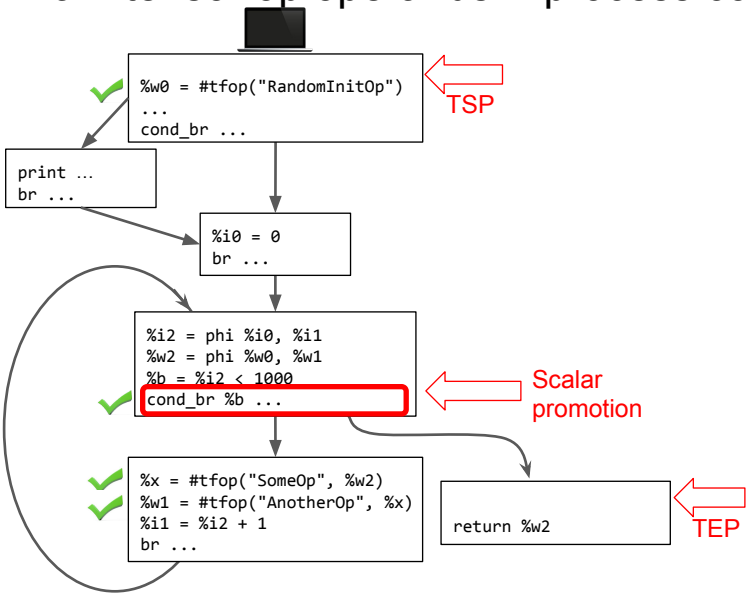


Now we start processing these tf ops. The goal is to mark them and their relevant operands in a transitive way, for them to run in the graph.

When we mark an instruction, we also marked its parent basic block. This way we can establish the CFG structure of the graph function.

Here we first mark the tfop w0 and x, along with their associated blocks.

Mark tensor op operands -- process cond_br %b ...

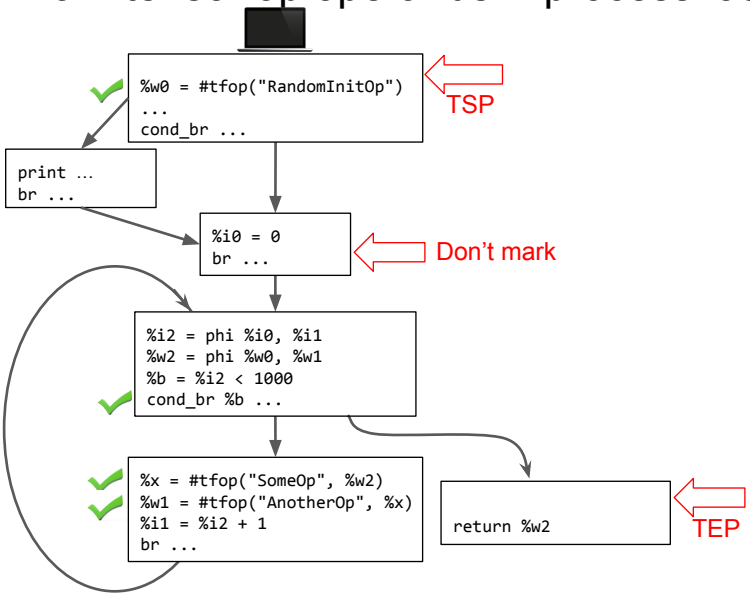


When a block is marked, we process all its predecessors transitively, and assess what other blocks we want to mark for the graph function.

Here we first look at the loop header block, as a predecessor of the loop body. Since loop body is control dependent on the header, in order to run the loop body in the graph, we have two options in terms of how to handle conditional branch of the header block.

One option is to run it on the host, and send the output to the graph. To minimize communication between host and graph, we go with another option, where we run the conditional branch itself in the graph. Since the graph only runs tensor computation, this option involves promoting the scalar value b into a tensor. We call this scalar promotion.

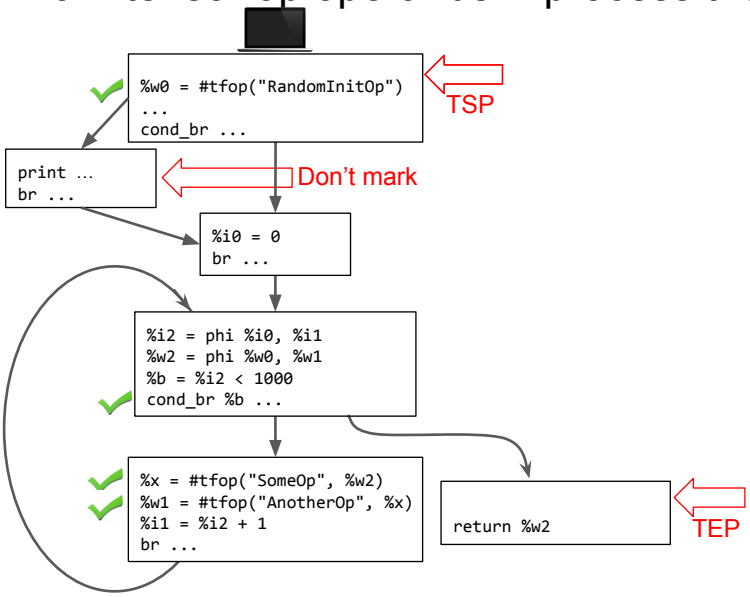
Mark tensor op operands -- process loop preheader



We then recursively process the predecessor of the loop header block, also known as the preheader. In this case, loop header post dominates the preheader, so it is not control dependent on preheader. As such, we don't mark the preheader for graph execution.

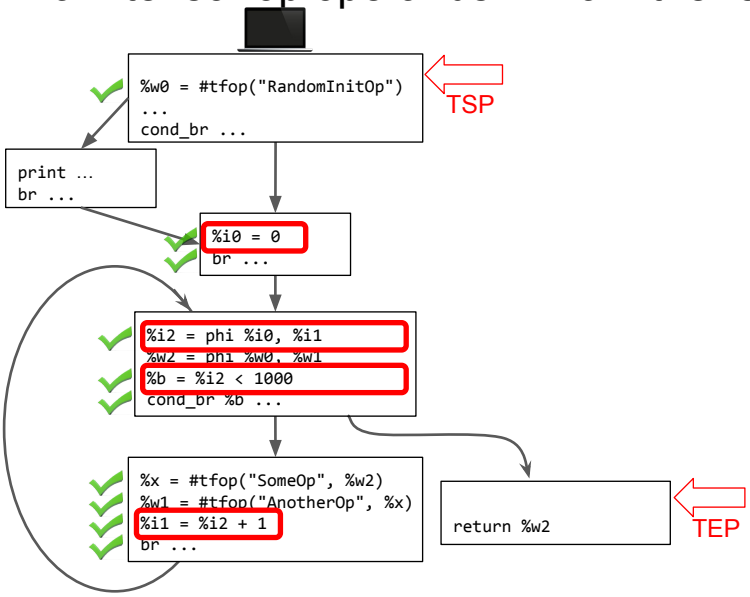
But we need to recursively process the predecessors of preheader, in case the preheader is itself control dependent on some other block.

Mark tensor op operands -- process the print block



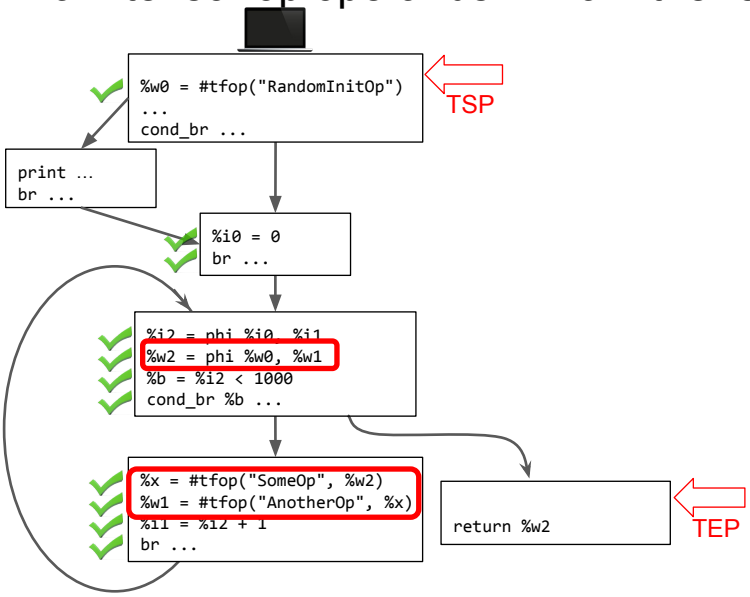
This leads us to the print block that only runs host code. We find the loop preheader is not control dependent on that print block, so we don't need to mark the print block.

Mark tensor op operands -- mark the remaining instructions



We continue to process the tf ops and their operands recursively.
We find that the instruction producing b can be marked to run the graph. Recursively, we can mark i2, i0 and i1.

Mark tensor op operands -- mark the remaining instructions

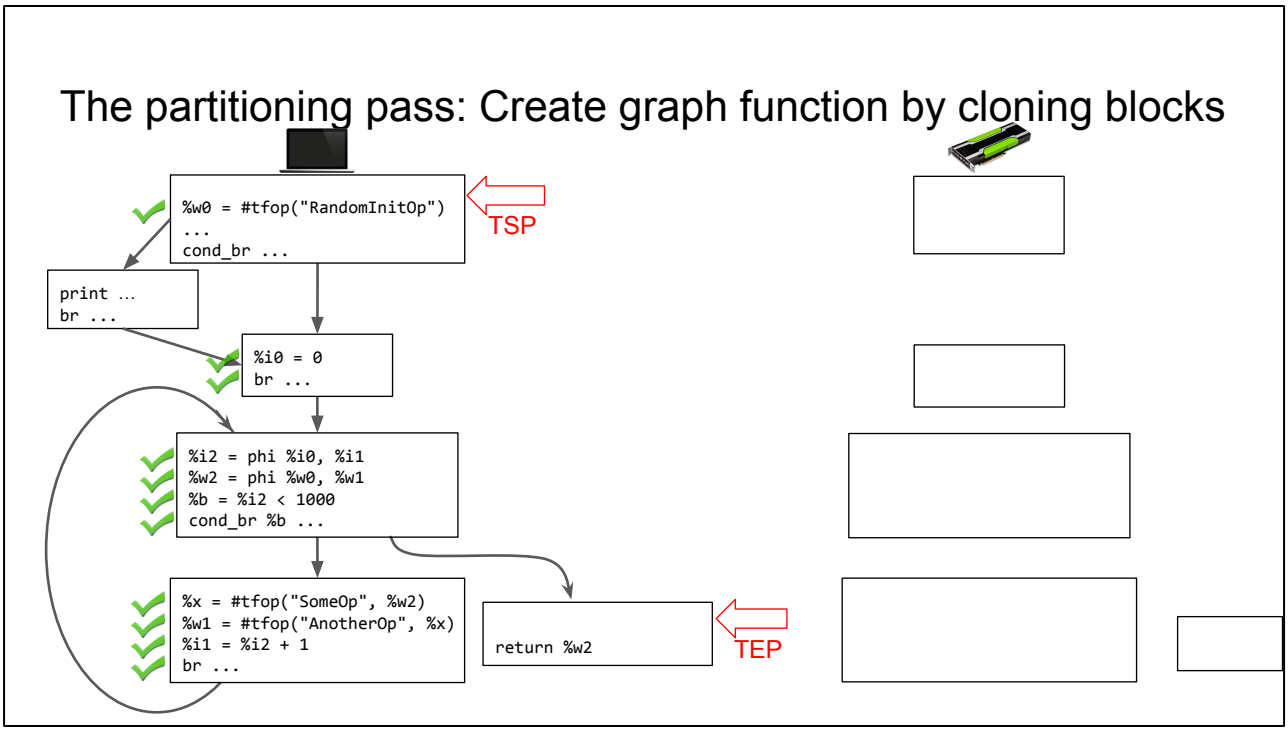


When processing the tfop x, we can recursively mark w2 and w1 in the graph as well.

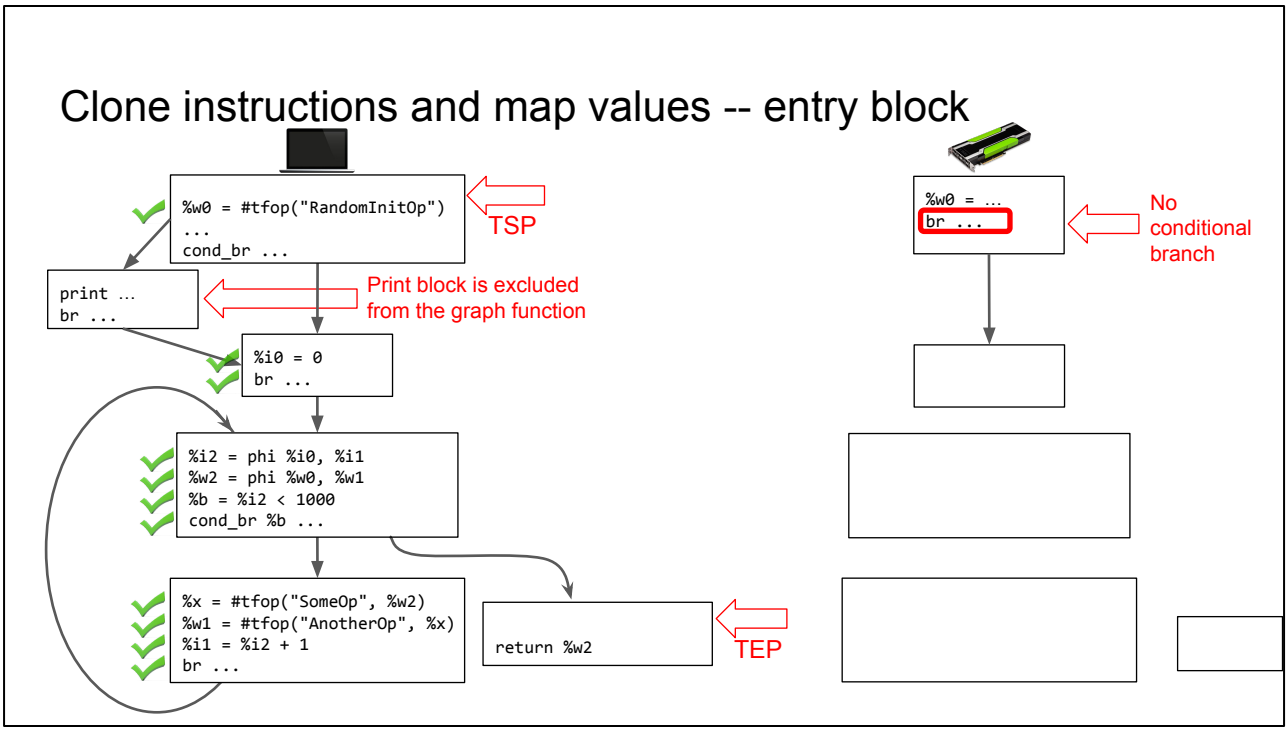
By now, we have completed the marking of instructions that we want to run in the graph.

We note a few things:

- First, the cond_br in the entry block is not marked. This is because we did not mark the print block that only runs on host.
- Second, the entire loop control flow has been marked for graph execution.

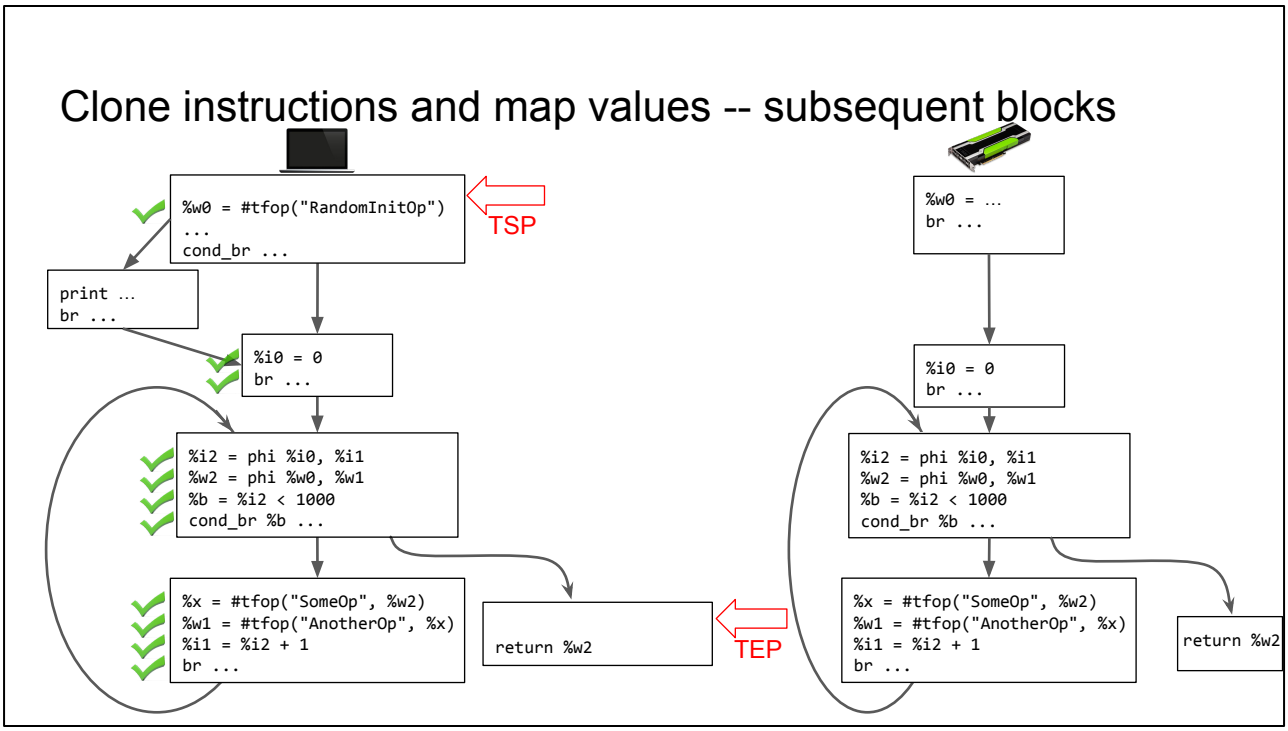


Now that we are done with marking, the partitioning is relatively straightforward. First, we create the CFG structure of the graph function, by cloning those marked blocks in the host function.

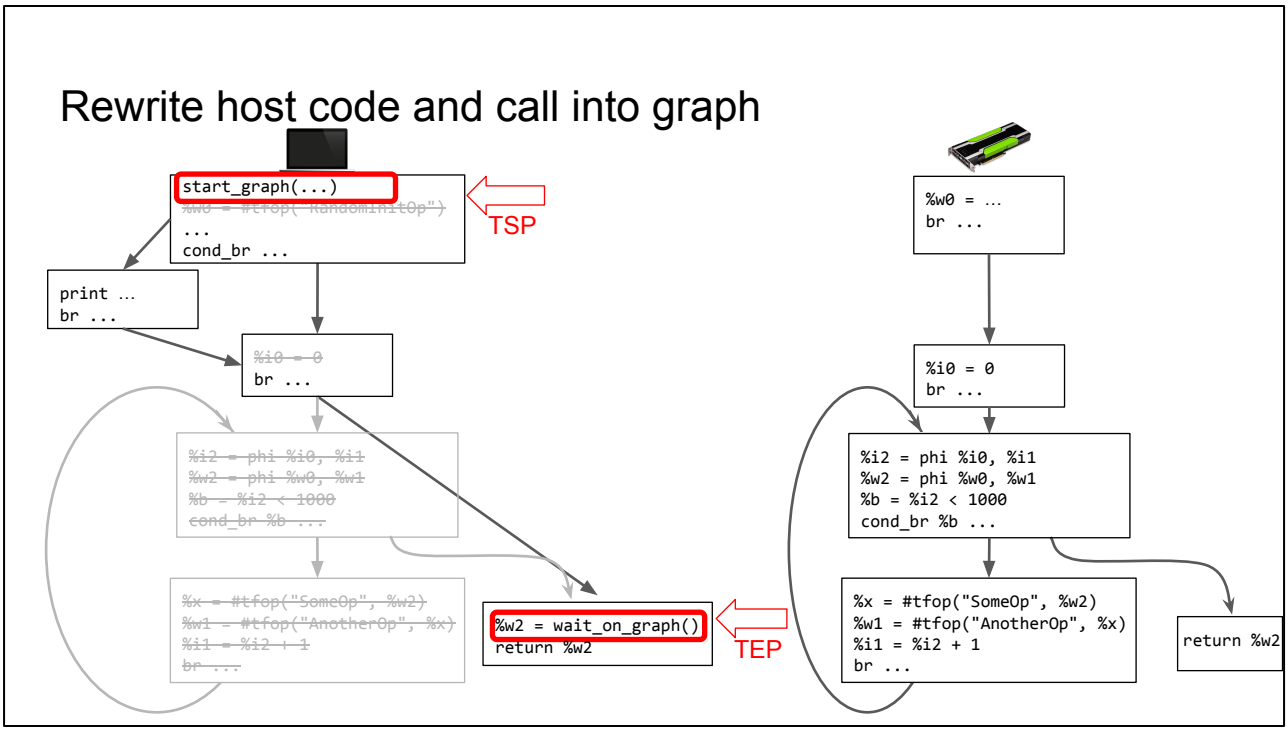


We then clone those marked instructions over. For the entry block, since the conditional branch is NOT marked, we synthesize an unconditional branch, going to the node that's the immediate postdominator of the entry block. It's the loop preheader block in this case.

This achieves the goal of excluding the print block from the graph function. This is the payoff of the earlier control dependence analysis when we mark the instructions, and it showcases the program slicing technique in our design, where the print block is sliced out of the graph computation.

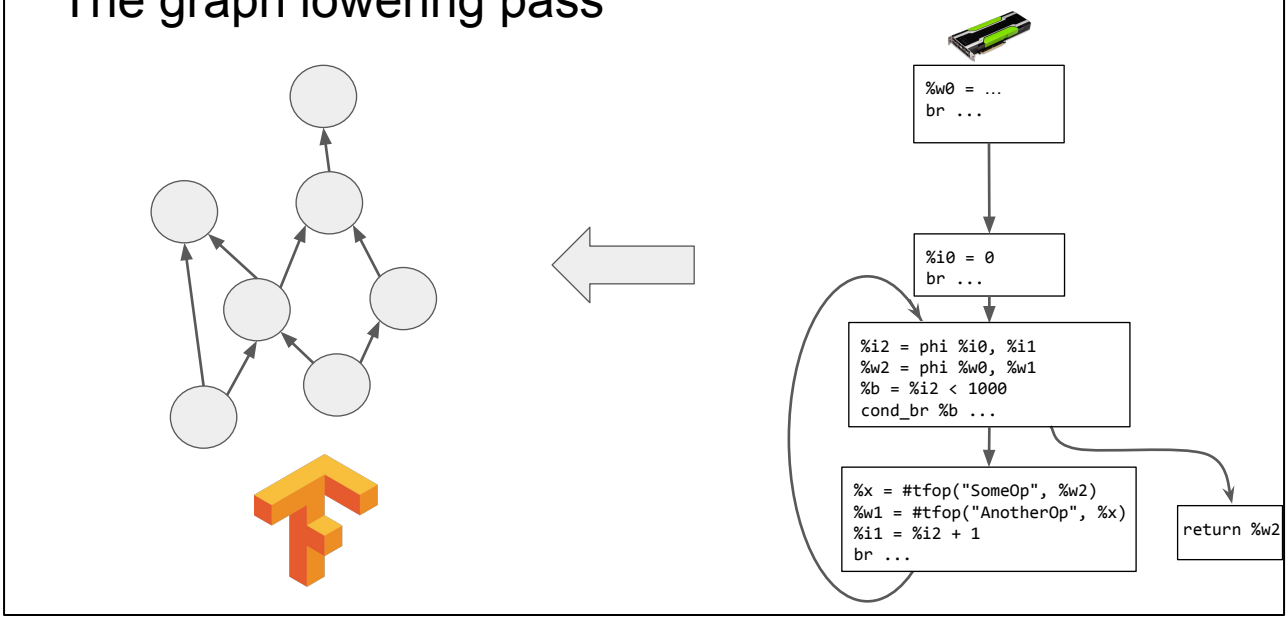


We clone over the other marked instructions. Nothing surprising.



The final step of the partitioning is to rewrite the host code. At TSP, we insert a runtime call to start the graph computation asynchronously. At TEP, we insert a call to wait for the graph computation.
We also delete those marked tensor instructions from the host. The deletion of the loop related basic blocks causes us to rewire the loop preheader to go directly to the return block. Again, this is another way of saying we have moved the loop to the graph function.

The graph lowering pass



Now that we have a SIL representation of the graph function, we lower it to a graph representation supported by TensorFlow. The lowering is mostly mechanical.

The lowered graph is then stored as a serialized string in the generated binary. When we start the graph computation at runtime, the string gets deserialized into a TF graph.

Flexible host / graph communication

```
let x = #tfop("RandomInitOp")
let y = #tfop("SomeOtherOp")
let z = atariSimulator(x, y)  // runs on the host
let u = #tfop("Add", z, ...)
```

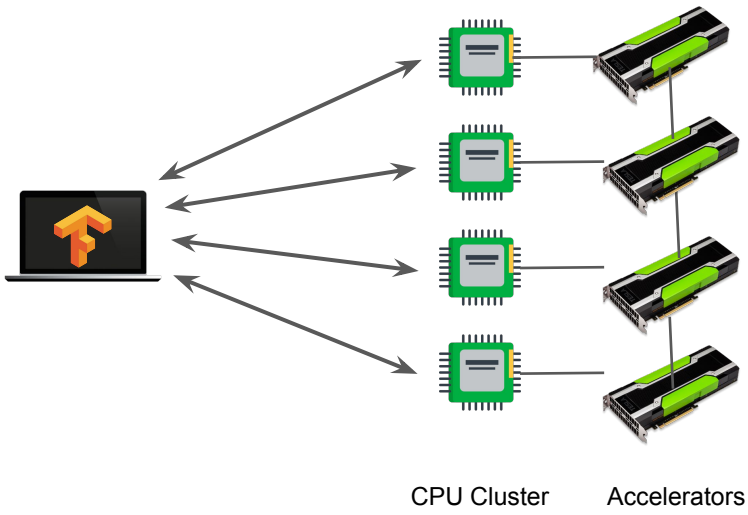
```
let x = RecvFromTF()
let y = RecvFromTF()
let z = atariSimulator(x, y)
SendToTF(z)
```

```
let x = #tfop("RandomInitOp")
let y = #tfop("SomeOtherOp")
#tfop("SendToHost", x)
#tfop("SendToHost", y)
let z = #tfop("RecvFromHost")
let u = #tfop("Add", z, ...)
```

One of the nice things about this is that we can support flexible communication between host and graph functions.
In addition to being able to synchronize at the end of the graph function execution, the host and graph functions can also communicate through sending and receiving tensors in the middle of their execution.
In this example, user feeds tensors x and y produced by the graph into a host function that runs the atari game simulator.

When we generate the graph function, we insert a send op to send a tensor like x to the host. We also add a recv function call in the host to receive that tensor. Similarly, we reverse the direction of sends/recvs when we want to feed a tensor produced on the host into the graph function, like tensor z here.

Device partitioning support



So far we've been assuming that the TensorFlow execution of the graph is on a single device, like GPU. In practice, TensorFlow itself is a distributed system, supporting different types of devices such as CPU, GPU and TPU. We are often feeding clusters of machines <click> which may each have one or more attached accelerators, and the different kinds of accelerators are usually running different programs.

To do this, our compiler partitions the graph function into a set of per-device graph functions, and inserts sends/recvs for cross device communication. The algorithms are very similar to the host-graph partitioning that we covered earlier.

Building a Programming Model

Image credit: wikimedia

Based on the compiler transformations, let's talk about what programming model we can build.

We want high level APIs, not tensor assembly!

```
let result: Builtin = #tfop("MatMul", images, weights)
```

```
let result: Tensor<Float> = images • weights
```

We want to build a programming model with nice abstractions but without runtime overhead. This way, users need not limit themselves to the lower level constructs based on the tfop magic.

Introduce Tensor struct

```
struct Tensor<T> {
  var value: Builtin  // the underlying, untyped tensor value
  init(value: Builtin) { self.value = value }
```

We first introduce a Tensor struct to wrap tensor values produced by tfops.

--

(Note: for simplicity, we glossed over Tensor vs TensorHandle in this talk.)

Introduce Tensor struct

```
struct Tensor<T> {
  var value: Builtin  // the underlying, untyped tensor value
  init(value: Builtin) { self.value = value }

  static func +(_ a: Tensor<T>, _ b: Tensor<T>) -> Tensor<T> {
    return Tensor<T>(#tfop("Add", a.value, b.value))
  }

  func matmul(_ b: Tensor<T>) -> Tensor<T> {
    return Tensor<T>(#tfop("MatMul", self.value, b.value))
  }
}
```

We can then define static and member methods on this struct. Here are some examples.

Add structs, tuples and functions

```
// x, a, b have type Tensor<Float>
let result = x.matmul(a) + b
```

```
// After inlining the calls
let tmp = Tensor(#tfop("MatMul", x.value, a.value))
let result = Tensor(#tfop("Add", tmp.value, b.value))
```

```
// After decomposing structs (same for tuples)
let tmp = #tfop("MatMul", x_value, a_valueb)
let result = #tfop("Add", tmp_value, b_value)
```

User can write nice code like this, and functions and structs can get inlined and scalarized away by the compiler.

We can also inline higher order functions, as long as the closures do not escape.

Add generics, using specialization

```
struct DenseLayer<T : Numeric> {
  var weights: Tensor<T>
  var bias: Tensor<T>
  init(inputSize: Int, outputSize: Int) {
...
}
fcl = DenseLayer<Float>(inputSize: 28 * 28, outputSize: 10)
```


```
struct DenseLayer_Float {
  var weights: Tensor_Float
  var bias: Tensor_Float
  init(inputSize: Int, outputSize: Int) {
...
}
fcl = DenseLayer_Float(inputSize: 28 * 28, outputSize: 10)
```

We can support generics as well, and we specialize them when we do graph extraction.

Swifty API for ML

```
let imageBatch = Dataset(elements: images)
let labelBatch = Dataset(elements: labels)

for (image, label) in zip(imageBatch, labelBatch) {
  let y = image • w + b
  let loss = (y - label).squared().mean()
  print(loss)
}
```

The end result of supporting all these abstractions is that we can provide beautiful APIs for end users. In this example, the data sets are used to read and feed training data into the ML model running in the loop, on a per-batch basis.

Summary of Graph Program Extraction

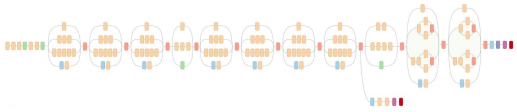
GPE can be applied to:

- Other programming languages
- New (non-ML) domains and their accelerators

So this concludes our deep dive into graph program extraction and the associated programming model. GPE is a set of algorithms that can be implemented on other SSA-based IRs.
The types of computational graphs that we extract also need not be limited to TensorFlow or machine learning. The techniques we presented should be generally applicable to domains where you want to slice out specific types of computation in the forms of dataflow graphs, and run them onto accelerator devices.

Now, back to Chris.

Swift for TensorFlow



Graph Program Extraction

TensorFlow Library

Python Interoperability

$$\nabla f$$

Automatic Differentiation

This talk has been about one of the major components of the Swift for TensorFlow project. It also includes work to integrate TensorFlow itself and build APIs for it, support for directly integrating Python and using Python APIs in Swift code, and perhaps most interesting for compiler folk: first class support for automatic differentiation.

```
func f(x: Tensor<Float>) -> Tensor<Float> {
   return tanh(x • x)
}

let ∇f = #gradient(f)
∇f(y)
```

Swift First-Class Automatic Differentiation Manifesto

Automatic differentiation allows computing gradients and derivatives of functions in vector-space, and an example of the power you get with language integration. If you're interested, search for the Swift automatic differentiation manifesto.

Get involved!

http://github.com/tensorflow/swift

Swift for TensorFlow is all open source, available now on Github. If you're interested in finding out more, we have a bunch of technical whitepapers available and a public mailing list.

Thank you