Lazy Tensor in S4TF

Status: Draft | In Review | Approved | Final | Obsolete
Authors: bgogul@google.com
Contributors (in alphabetical order): apassos@google.com, asuhan@google.com, burmako@google.com, saeta@google.com
Last Updated: 2019-06-20

The document outlines the design of a Swift For TensorFlow (S4TF) runtime mechanism called Lazy Tensor that provides an efficient way to stage machine learning computations to an accelerator device like GPU or TPU.

Background

There have been several different approaches to staging machine learning computations to accelerators such as GPUs and TPUs[1]. Traditionally, TensorFlow expected the programmer to build dataflow graphs that capture the semantics of their machine learning models. The dataflow graphs are then evaluated by a sophisticated runtime, which performs various optimization on the user-generated graph before launching the computations on an accelerator. This is known as the *graph mode* in TensorFlow.

Executing models in graph mode provided great performance as the runtime could perform various optimizations such as op fusion before launching the computations. However, they do not provide for a great user experience as graph mode is incompatible with the imperative programming model that programmers are familiar with: (1) performing line-by-line debugging is difficult as the computations don't execute until the whole graph is built, (2) control dependencies should be explicitly specified instead of using natural control flow constructs in a given language. Most recently, TensorFlow provides an eager execution mode, an imperative programming environment that evaluates operations immediately, without building graphs.

While eager mode addresses the usability issues in graph mode, kernel dispatch overheads become a significant performance tax when using TensorFlow today. Further, the TPU software stack (XLA) cannot perform advanced compilation optimizations such as operator fusion in eager mode. As a result, it's very valuable (for both GPUs and TPUs) to build up representations larger than a single kernel.

[1] This section provides just enough background and is not intended to be a survey.

Related work: TensorFlow provides a @tf.function decorator that can be added to a python function so that the computations in the function are staged into a computational graph before execution. TensorFlow uses a combination of tracing and AST transformations to compile the function into a tensorflow graph. While most of the discussion so far has focussed on TensorFlow, other frameworks like PyTorch have very similar programming models. For instance, PyTorch uses an imperative programming environment and provides mechanisms such as tracing and @torch.jit.script decorator to stage computations before launching them on accelerators.

In this document, we outline an approach called *Lazy Tensor* to extract and execute graph fragments from an unmodified Swift for TensorFlow program containing tensor computations.

Lazy Tensor Design

The idea behind lazy tensor is pretty simple and is inspired by lazy evaluation in programming languages, which delays the evaluation of an expression until its value is needed. Instead of evaluating a tensor computation right away, the runtime captures enough information about the operation so that it can be evaluated at a later time when it is needed. Even though this document discusses tensorflow-specific implementation, the ideas are very generic. In fact, lazy evaluation has already been shown to be effective for staging computations to xla in pytorch.

Virtualization of `TensorHandle`

First, we introduce an abstraction for the `TensorHandle` used with the `TensorFlow` library.

```
/// This protocol abstracts the underlying representation of a tensor. Any type
/// that conforms to this protocol can be used as a `TensorHandle` in the
/// `TensorFlow` library.
public protocol _AnyTensorHandle: class {
    var _tfeTensorHandle: TFETensorHandle { get }
}
```

The only requirement of an `_AnyTensorHandle` protocol is that there is a way to convert it into a `TFETensorHandle` that represents a concrete tensor handle in the TensorFlow Eager runtime.

Representation of a `LazyTensor`

Now the `_AnyTensorHandle` protocol allows us to define a `LazyTensor` type. A `LazyTensor` is either a concrete `TFE_TensorHandle` or the symbolic result of a deferred tensor operation (`LazyTensorOperation`).

```
class LazyTensor: _AnyTensorHandle {
    enum Handle {
        /// Bool indicates if this concrete TFETensorhandle was a result of
        /// materialization.
        case concrete(TFETensorHandle, materialized: Bool)
        /// Bool indicates whether this is a live tensor. This flag is used to
        /// heuristically determine whether this symbolic tensor should also be
        /// materialized whenever materialization of any other tensor is triggered.
        case symbolic(LazyTensorOperation, index: Int, isLive: Bool)
    }
}

class LazyTensorOperation: TFTensorOperation {
     enum Attribute {
         case boolValue(Bool)
         case intValue(Int)
         ...
     }

    let name: String
    var attrs: [String: Attribute]
    var inputs: [LazyTensor] // The actual implementation distinguishes
                             // single inputs and list inputs.
}
```

We will touch upon the `materialized` and `isLive` attributes later in the document.

Materialization of a `LazyTensor`

A symbolic tensor gets materialized into concrete tensor whenever the computed `_tfeTensorHandle` property of a `LazyTensor` is accessed:

```
class LazyTensor {
    // ...
    var _tfeTensorHandle: TFETensorHandle {
        switch handle {
        case .concrete(let h, _): return h
        case .symbolic(let op, let index, _):
            let h = materialize(op: op, index: index)
            // memoize the result as well.
            handle = .concrete(h, materialized: true)
            return h
        }
    }
}
```

Materialization of a symbolic tensor triggers the execution of all the deferred tensor computations on which the given symbolic tensor depends on. After materialization, the result is also memoized to avoid multiple executions of the same operation. The algorithm to convert a `LazyTensor` into a concrete `TFETensorHandle`, i.e., materialize a `LazyTensor` is as follows:

```
func materialize(op: LazyTensorOperation, index: Int) -> TFETensorHandle{
  // 1. Collect the set of reachable lazy tensor operations `ops`
  // by performing a dfs starting from op and following the inputs.

  // 2. Convert the set `ops` into a TensorFlow function and evaluate.

  // 3. Return the result of the evaluation at `index`.
}
```

Consider the following example:

```
let a = Tensor<Float>(10.0)
let b = Tensor<Float>(2.0)
let c = Tensor<Float>(3.0)
let w = a + b
let x = w - c
let y = x + x + w
let z = y + y
print(z)
```

Materialization is triggered when the runtime encounters the `print` statement. The extracted trace is as follows:

```
lazyTrace_8() -> (float) {
  %0 = Const[dtype: float, value: 10.0]()
  %1 = Const[dtype: float, value: 2.0]()
  %2 = Add[T: float](%0, %1)
  %3 = Const[dtype: float, value: 3.0]()
  %4 = Sub[T: float](%2, %3)
  %5 = Add[T: float](%4, %4)
  %6 = Add[T: float](%5, %2)
  %7 = Add[T: float](%6, %6)
  return %7
}
```

Practical Considerations

Lazy evaluation defer the execution of the tensor operations as until it is actually needed:

- Host code requires it -- for example printing tensor elements or making a control flow decision based on tensor elements.
- After each training step, since the graph is complete at that point and constructing graphs must be wrapped up at natural boundaries which maximize the chance to see the same graph for the entire duration of training and thus avoid recompilation.

Otherwise, the runtime costs associated with lazy evaluation may outweigh the benefits obtained by staging computations to a graph. In an ideal situation, all the tensor computations in the program would be staged to a single graph. However, that does not always happen in practice. In this section, we discuss the various aspects that have an effect on the frequency of materialization and associated runtime costs. We also discuss solutions to reduce the impact on the performance.

Proactively materialize `LazyTensor` instances

Note that a trace extracted for concretizing a symbolic tensors also contains computations necessary to concretize other symbolic tensors in the program. For example, in the extracted trace `lazyTrace_8`, values `%2`, `%4`, and `%6` correspond to the tensor variables `w`, `x`, and `y`, respectively. Consequently, `w`, `x`, and `y` can also be materialized along with `z`, if we mark these values as outputs as follows:

```
lazyTrace_8() -> (float, float, float, float) {
  %0 = Const[dtype: float, value: 10.0]()
  %1 = Const[dtype: float, value: 2.0]()
  %2 = Add[T: float](%0, %1)
  %3 = Const[dtype: float, value: 3.0]()
  %4 = Sub[T: float](%2, %3)
  %5 = Add[T: float](%4, %4)
  %6 = Add[T: float](%5, %2)
  %7 = Add[T: float](%6, %6)
  return (%2, %4, %6, %7)
}
```

However, increasing the number of outputs can have a negative impact on performance. For example, if you have ResNet-50 and don't fuse the forward and the backward graphs, there will hundreds of tensors which need to be inputs to the backward graph and need to be materialized as outputs of the forward graph, which leads to OOM at high batch sizes. Therefore, we should be conservative when marking a value in the extracted trace as an output. One heuristic that seems to work well is to only mark those values that will potentially be used outside of the extracted function. When this heuristic is used, note that `%5` is not marked as an output as it corresponds to `x + x`, the result of which is not used outside.

We keep track of useful operations by observing the `LazyTensorOperation` instances in the initialization and deinitialization of `LazyTensor` instances. Effectively, a `LazyTensorOperation` is considered useful if any symbolic `LazyTensor` refers to it. The only exception to this rule is that symbolic `LazyTensor` instances that are only used as inputs of a `LazyTensorOperation` are ignored. Typically these correspond to the operations such as `x + x` in `lazyTrace_8`. The `isLive` attribute of a symbolic `LazyTensor` is used to distinguish these cases.

Our heuristic to track usefulness works quite well in practice. For example, in the trace extracted for a model with several layers, only the operations in the last layer of the network are marked as outputs.

We can improve the logic for marking outputs further if we have liveness information from the compiler. For example, `w`, `x`, and `y` are not considered live at `print(z)`. Therefore, if we have liveness information, we will only mark `z` as the output, which is the most desirable outcome for this example.

Function Caching

In a typical model, materialization happens at the end of every iteration of a training loop and the extracted trace should be identical at each materialization step. To avoid the repeated cost of creating the tensorflow function and running all the ensuing optimizations at every iteration, it is important to cache them. The TensorFlow and XLA runtimes already have caching mechanisms based on structural similarity of functions. In the case of XLA, the shapes are also used as the part of the cache key. To leverage the caching mechanism in XLA and TensorFlow (as well as to cache extracted traces in the S4TF runtime), we need to preserve the structural similarity of the extracted traces at materializations points. In this section, we will discuss a few techniques for this purpose.

Deterministic order for the operations in the trace

This is easily achieved as we collect the necessary operations by performing a DFS starting from the symbolic tensor under consideration. We can have a deterministic order by adding the operations to the trace in the order in which they were visited during DFS.

Promotion of constant tensors to arguments

Note that the constants in the program are baked into the extracted program. This is not always desirable. Consider the following example:

```
var sum = 0.0
for i in 1...10 {
   sum = sum + Float(i)
   print ("\(sum)")
}
```

Let us unroll the execution of few iterations of the loop:

```
sum0 = 0
i1 = 1.0
sum1 = sum0 + Float(i1)
print ("\(sum1)")
sum2 = sum1 + Float(i2)
print ("\(sum2)")
sum3 = sum2 + Float(i3)
print ("\(sum3)")
// ...
```

If we always bake the constants into the extracted trace, we will get a different function each time:

```
lazyTrace_sum1() -> (float) {
   %0 = Const[dtype: float, value: 0.0]() // sum0
   %1 = Const[dtype: float, value: 1.0]() // i1
   %2 = Add[T: float](%0, %1)
   return %2
}
```

```
lazyTrace_sum2() -> (float) {
   %0 = Const[dtype: float, value: 1.0]() // sum1
   %1 = Const[dtype: float, value: 2.0]() // i2
   %2 = Add[T: float](%0, %1)
   return %2
}
```

```
lazyTrace_sum3() -> (float) {
   %0 = Const[dtype: float, value: 3.0]() // sum2
   %1 = Const[dtype: float, value: 3.0]() // i3
   %2 = Add[T: float](%0, %1)
   return %2
}
```

To promote constants to arguments of functions, we use the following heuristics:

Promote materialized tensors. If the constant tensor was the result of materializing a lazy tensor, promote it to an argument. The `materialized` field of a concrete `LazyTensor` is used to track such `materialized` tensors. In our example, `sum1` and `sum2` will be promoted to a function argument:

```
lazyTrace_sum2(%0: float) -> (float) {
   %1 = Const[dtype: float, value: 2.0]() // i2
   %2 = Add[T: float](%0, %1)
   return %2
}
```

```
lazyTrace_sum3(%0: float) -> (float) {
   %1 = Const[dtype: float, value: 3.0]() // i3
   %2 = Add[T: float](%0, %1)
   return %2
}
```

Note that this is still not ideal as `sum0`, `i1`, and `i2` will not be promoted as a function argument because they were originally a constant tensor in the program and is not the result of materialization. T

Promote based on history. If the currently extracted function has a similar signature to a previously extracted function, we compare the operations in the traces by doing a linear scan. If the functions differ only due to a subset of the constant tensors, we promote all these constants to function arguments. This strategy will promote `sum0`, `i1`, `i2`, `i3` to be function arguments as well. Note that this heuristic similar to the widening heuristic in formal program analysis.

Using both these heuristics, we will get the following trace for every iteration of the loop:

```
lazyTrace_sum3(%0: float, %1: float) -> (float) {
   %2 = Add[T: float](%0, %1)
   return %2
}
```

Shape Computations + Lazy Tensor

We do not track the shapes of tensors on the swift side. Consequently, anytime a shape is required (e.g., Flatten, DropOut), the corresponding lazy tensor gets materialized. Such materialization has the unfortunate consequence of increasing the frequency of materialization and making the extracted traces shorter.

In some cases, it is possible to rewrite these layers to use a shape operation that is stageable, such as tf.Shape. However, it is not always a viable solution. For example, the implementation of matmul in tensorflow/swift-apis has control-flow that depends on the result of the shape computation. One idea that is promising is to use the shape inference function associated with a registered TensorFlow op to keep track of shapes on the swift side. Note that the shape functions won't trigger materialization.

Control Flow

Control-flow constructs like `if` and `for` get unrolled in the extracted trace. Staging them in lazy evaluation requires further support at the library level (functional `if` and `while`) and possibly the compiler as well. One of the key issues that need to be addressed is how to deal with the non-tensor code either in the conditional or the loop body. For instance, do the results of executing non-tensor code during staging get baked into the staged function? Alternatively, do they get executed every time the staged function is executed? Staging control-flow constructs is a topic in itself and we plan to address it in a separate design review.

Composability with AutoDiff

LazyTensor composes nicely with AutoDiff. Note that AutoDiff generates the code for gradient computations at compile time. Lazy Tensor simply stages and evaluates the gradient computations at runtime.

Diagnostics

We don't have a great way to diagnose errors encountered during the execution of extracted traces. Ideally, we would like to report the source line that is relevant for failing operation along with a runtime stack trace. However, deferred evaluation prevents us from getting a useful stack trace to diagnose the failure.

When we start doing shape inference while building the traces, we should be able to raise most of these errors sooner and get a reasonable stack trace. This is also the approach used in pytorch/xla. Once we sort these issues out, runtime crashes should only occur in rare instances (e.g., division by zero, square root of negative, etc.). One could also turn off lazy evaluation and go back to op-by-op dispatch to debug the issue if needed.

Compiler Support

While we don't need compiler support for the implementation of Lazy Tensor, the compiler can help in many ways. Clearly, adding compiler support is a huge topic and requires a separate design review on its own. We just present some half-baked ideas here to give a taste of the possibilities:

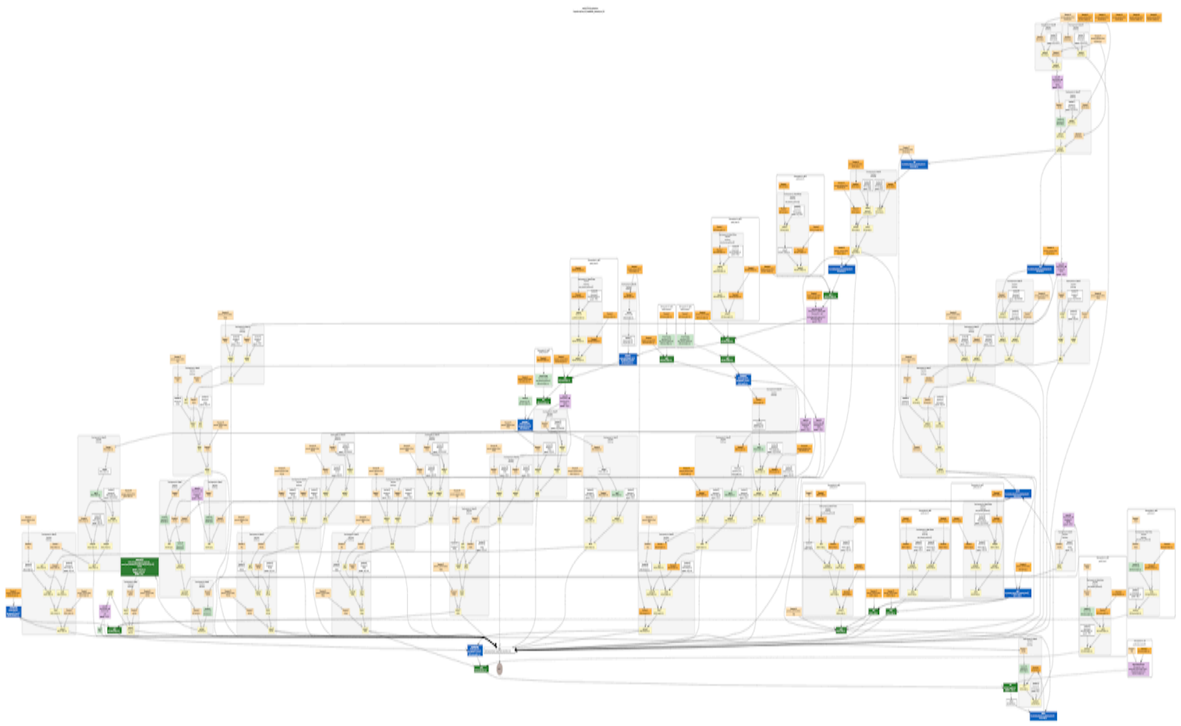
- Functionalization of control-flow constructs like `if` and `for`.
- Transform functions such that frequency of materialization is reduced (e.g, by moving code that triggers materialization as late as possible).
- Optimizations like Common subexpression elimination (CSE) can help in reducing the size of the extracted functions when we have a large number of tensors that depend on mostly-overlapping computations.
- Transformations to aid runtime failure diagnostics.

- ...

Preliminary Results

We have some preliminary results with using lazy evaluation. We are able to fuse the entire training loop for models like MNIST and ResNet18 into a single trace. We show the extracted traces for MNIST. We haven't done an exhaustive performance evaluation yet, but running MNIST on GPU with lazy tensor is about 10% faster than eager (op-by-op dispatch) execution. We will be doing more evaluations and applying the ideas on more models in the coming days and keep the community posted.

Extracted training loop with XLA optimizations like op fusion ([svg]()):



Extracted training loop before any optimizations (svg):