x10 deep dive

Target accelerators with Swift

Quick recap

x10 is an implementation of S4TF tensor operations:

- Same semantics
- Works well with accelerators (TPU, GPU)
- Uses XLA

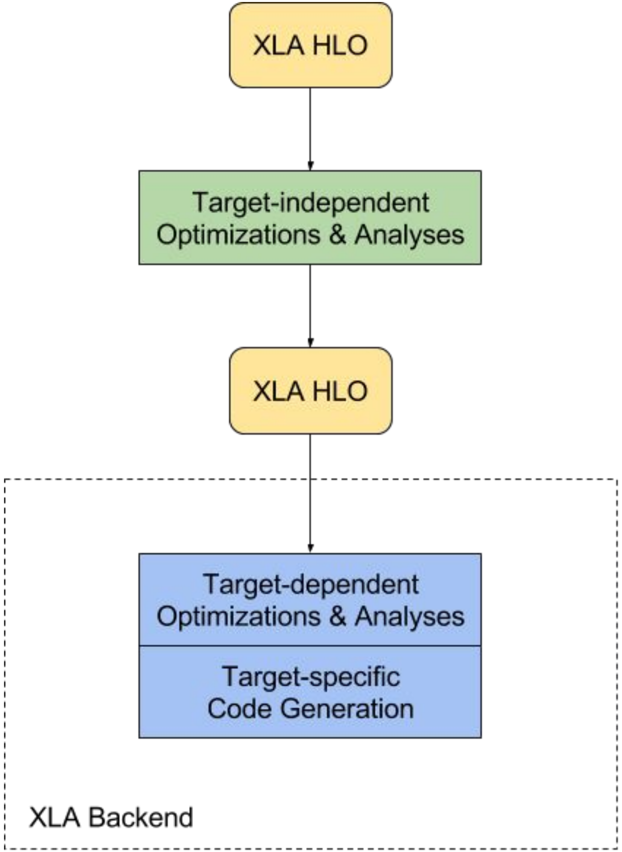
Intro

- Brief overview of XLA
- Why do we need the x10 approach?
- Multiple device support
- Mixed precision
- Results
- Future work

About XLA

What is XLA?

- **A**c**c**elerated **L**inear **A**lgebra: Domain-specific compiler for linear algebra
- XLA takes graphs specified in an IR called HLO (High Level Optimizer)
- Example operations: Dot, Add, Eq, Broadcast, Reshape, Conv (convolution)



XLA architecture diagram

XLA (HLO) is an IR

- Lower level than (Swift for) TensorFlow, PyTorch etc.
    - Framework operations are converted to XLA
    - Example: no batched matrix multiplication in XLA, can be expressed as elementary XLA operations
- Higher level than native code
    - CPU, GPU use LLVM to generate native code
    - TPU: custom code generation
- Target-independent optimization reused across frameworks and hardware types

Why x10

Target XLA directly

- Lower level than TensorFlow
- More direct control over generated code
- Control over device assignment, distributed training

x10: Just-in-time compiler

- Lazy tensor approach
- Operations are queued instead of eagerly evaluated
- A graph is constructed, translated to XLA & evaluated when:
    - Explicitly requested by a **LazyTensorBarrier()** call
    - `print(t)`
    - `if t.scalarized() > 42`

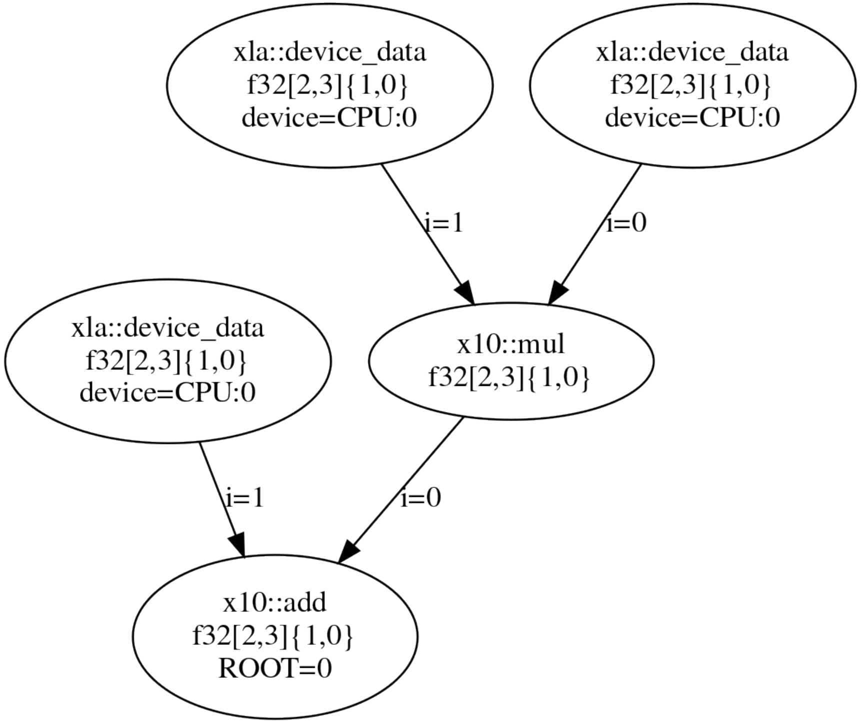
```
let x = Tensor<Float>(
  shape: [2, 3],
  scalars: [1, 2, 3, 1, 2, 3])

let y = Tensor<Float>(
  shape: [2, 3],
  scalars: [4, 5, 6, 4, 5, 6])

let z = Tensor<Float>(
  shape: [2, 3],
  scalars: [7, 8, 9, 7, 8, 9])

print(x * y + z)

// Prints:
// [[11.0, 18.0, 27.0],
//  [11.0, 18.0, 27.0]]
```

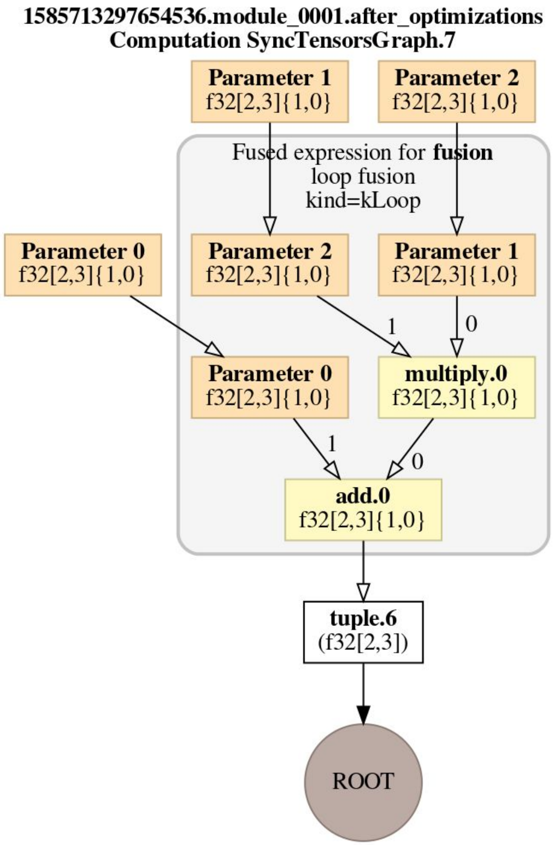


x10 computation graphs

```
let x = Tensor<Float>(shape: [2, 3], scalars: [1, 2, 3, 1, 2, 3])
let y = Tensor<Float>(shape: [2, 3], scalars: [4, 5, 6, 4, 5, 6])
let z = Tensor<Float>(shape: [2, 3], scalars: [7, 8, 9, 7, 8, 9])
print((x * y + z).irText)
```

The snippet above prints:

```
IR {
  %0 = f32[2,3]{1,0} xla::device_data(), device=CPU:0
  %1 = f32[2,3]{1,0} xla::device_data(), device=CPU:0
  %2 = f32[2,3]{1,0} xla::device_data(), device=CPU:0
  %3 = f32[2,3]{1,0} x10::mul(%2, %1)
  %4 = f32[2,3]{1,0} x10::add(%3, %0), ROOT=0
}
```



XLA optimized graph

Why not compile ahead of time?

```
if runtime_flag_1_or_2 {
    middle = layer1(input)
} else {
    middle = layer2(input)
}
output = anotherLayer(middle)
```

Why not compile ahead of time?

- We don't know until runtime: `anotherLayer`∘`layer1` or `anotherLayer`∘`layer2`
- Solution 1: compute and materialize `middle`, apply `anotherLayer` on it
  - Can increase memory usage dramatically: model won't fit
  - Can decrease performance dramatically: memory bandwidth is finite
  - XLA can reuse buffers based on lifetimes, recompute expressions instead of storing them etc.
- Solution 2: compile all combinations?
  - Might not even know the domain, not all flags are boolean
  - We might have 10 independent flags

Multiple device support

API

- No scopes, only constructors.
- All Tensor constructors are augmented with on: `Tensor([1,2], on: Device.default)`
- `LazyTensorBarrier(on: device)` to call a barrier on the given device.
- `Tensor` and `KeyPathIterable` support `.init(copying: Self, to: Device)` for device transfers.
- `Device.defaultTFEager` for CPU TensorFlow.

Caveats

- A trace can only contain one device. Create a worker thread for each device!
- `crossReplicaSum(scale)` operates over devices chosen by `LazyTensorBarrier(on: device, devices: crossReplicaSumDevices)`
- The devices in a cross-replica sum must evaluate the same computation.

Mixed precision

Why?

- Faster training
- Reduced memory usage
- Same training accuracy

API

- Same idea as device transfers:
  - `t.toReducedPrecision` and `t.toFullPrecision` convert `t` to bfloat16 / full precision.
  - Conversions don't change logical type, still `Tensor<Float>`.
  - `KeyPathIterable` to convert entire models.
- High-level API:
  - Training loop takes a `useAutomaticMixedPrecision` flag.
  - Weights are kept in full precision, inputs and activations are reduced precision.

Results

Great speed

- ResNet-50, TPU 8x8 slice, mixed precision: 76000 images / second
    - In the ballpark of state of art performance
- CIFAR-10 using ResNet-50 on one GPU
    - Around 2.5x faster training on one GTX 1060
    - Memory usage reduction from 4.5 GB to 3 GB
- Tracing overhead hidden behind step time on accelerator

Great usability

- For ResNet and MNIST examples, code changes are very minimal
  - Completely unmodified model
  - Add one line: `LazyTensorBarrier()` per training step
  - Use the training loop API for distributed (model still unmodified)
- Fully implements S4TF, imperative experience
  - Print tensors
  - Set breakpoints
  - Step through the code

Future work

Dynamic shapes

- Step n: `f(x, y)` with `x` and `y` of shape `[2, 3]`
- Step n + 1: `f(x, y)` with `x` and `y` of shape `[7, 5]`
- We detect and compile the `[7, 5]` version automatically
- Not a problem! Right?

Dynamic shapes

- Recompilation works in moderation
- “Megamorphic” shapes: almost all time spent (re-)compiling
- Example: Mask R-CNN

Dynamic shapes - Mitigation

- Just don’t do it, (re-)write your model to not have them
- `PrintMetrics()`, look for `UncachedCompile` counter to confirm

Dynamic shapes - Future

- Support for dynamic shapes in XLA
- On S4TF side
  - Runtime profiling: observe shapes
  - Erase highly dynamic dimension sizes, leave the rest alone for XLA to use in optimizations
  - Use an interpreter(-ish) for highly dynamic portions, compile longest traces possible around them
  - Have a mode which throws when introducing shape instability (example: slice with variable upper bound)
- Not a pure software problem, difficulty depends on hardware too

Accidental forced evaluation

- Accidental use of tensor values in host code: `if norm(t) < 0.01`
- Usually quite easy to rework with replacing(with:where:)
- Accidental use of operations which don’t have an XLA lowering (very few!)
- Current mitigation: use a debugger and set a breakpoint on `XLATensor_materialize`
- Future: collect backtraces and expose them as counters
- Future: counters for operations without XLA lowering (easy!)

Future: multiple tiers

- Best case: fuse the entire computation and compile it as a whole
- Degrade gracefully and automatically when noticing problems
  - Shorten compiled traces until they hit the compiled code cache reliably
  - Go all the way back to fully eager execution for highly dynamic portions
  - Shortening the traces + eviction can also solve the accelerator OOM
- Continuous profiling infrastructure, shared for all types of hardware

Infinitely hackable S4TF

- Most S4TF operator translations to XLA are in C++
- Idea: use Swift
  - Auto-generate tensor methods for all XLA operators
  - Write all translations in Swift, as regular code using tensors
  - C++ surface: just the auto-generated code!
- Already done for some interesting operators: stridedSlice and stridedSliceGrad

Additional resources, Q&A

Additional resources

- API guide
- Troubleshooting
- Tests
- Curated examples coming soon

Q&A