XLA Runtime + Cuda Graphs

RFC: Cuda Graphs support in XLA

XLA Runtime Supports Multiple Exported Functions

- XLA today always exports a single entry point function from the HLO module
- XLA runtime modules do not have this limitation, and can export multiple functions
- Exported functions are callable from the C++ API: `Executable::Execute(int ordinal, Arguments arg)`

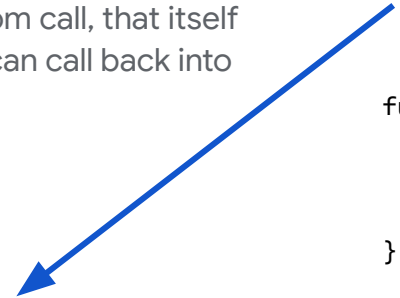
```
rt.export @add ordinal 0
rt.export @mul ordinal 1

func @add(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)
    -> tensor<?xf32> {
  %0 = mhlo.add %arg0, %arg1 : tensor<?xf32>
  return %0 : tensor<?f32>
}

func @mul(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)
    -> tensor<?xf32> {
  %0 = mhlo.mul %arg0, %arg1 : tensor<?xf32>
  return %0 : tensor<?f32>
}
```

XLA Custom Calls Support Function References

- XLA runtime can pass a reference to an exported function to custom calls
- Executable calls into a custom call, that itself using a function reference can call back into the executable function



```
CustomCall::Bind("call_ref")
  .Arg<Memref>()
  .Arg<Memref>()
  .Attr<FunctionRef>("func")
  .To([](Memref a, Memref b, FunctionRef func) {
    return func(a, b);
  });
```

```
rt.export @add ordinal 0
rt.export @mul ordinal 1

func @call_ref(%arg0: tensor<?xf32>,
               %arg1: tensor<?xf32>)
  -> tensor<?xf32> attributes { rt.custom_call = "call_ref" }

func @impl(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)
  -> tensor<?xf32> {
  %0 = mhlo.add %arg0, %arg1 : tensor<?xf32>
  return %0 : tensor<?f32>
}

func @main(%arg0: tensor<?xf32>, %arg1: tensor<?xf32>)
   -> tensor<?xf32> {
  %0 = call @call_ref(%arg0, %arg1) { func = @impl }
     : (tensor<?xf32>, tensor<?xf32>) -> tensor<?xf32>
  return %0 : tensor<?f32>
}
```

XLA Exports Graph Capture Functions

- XLA runtime implements a custom call that takes a reference to “graph capture” function
- Before calling graph capture function runtime sets up CUDA graph building “context”.
- Instead of a graph capture XLA can provide a set of custom calls that build graph using explicit API

```
CustomCall::Bind("xla.graph.launch")
  .Arg<RemainingArgs>()
  .Attr<FunctionRef>("capture")
  .To([](RemainingArgs args, FunctionRef func) {
    if (pointers_changed(args)) {
      cudaStreamBeginCapture();
      Status captured = func(a, b);
      cudaStreamEndCapture();
      UpdateGraph();
    }
    return ExecuteCachedGraph(args, fun);
  });
```

```
rt.export @main ordinal 0
rt.export @graph.capture ordinal 1

// This function does not execute "add". Once it's lowered
// to runtime HAL it adds an "add" command to the "command
// buffer" (cuda graph).
func @graph.capture(%arg0: tensor<?xf32>,
                    %arg1: tensor<?xf32>) -> ... {
  %0 = mhlo.add %arg0, %arg1 : tensor<?xf32>
  return %0 : tensor<?f32>
}

func @main(%arg0: tensor<?xf32>,
           %arg1: tensor<?xf32>) -> ... {
  // Xla runtime is responsible for capturing graphs,
  // caching them, and submitting to the compute stream.
  %0 = call @xla.graph.launch(%arg0, %arg1)
    { capture = @graph.capture }
    : (tensor<?xf32>, tensor<?xf32>) -> tensor<?xf32>
  return %0 : tensor<?f32>
}
```

Basic implementation is on GitHub