Google

IREE

MLIR-based End-to-End ML Tooling

benvanik@google.com / 2020-01-30

https://github.com/google/iree

iree-discuss@googlegroups.com

Intermediate **R**epresentation **E**xecution **E**nvironment

A MLIR-based compiler that lowers ML models to an IR that is optimized for real-time mobile/edge inference against heterogeneous hardware accelerators.

IREE Workflow

Source Model	IREE Compiler (MLIR-based)	IREE Runtime
Work in your frontend of choice: TensorFlow, JAX, PyTorch, Swift for TensorFlow, ONNX exportable forms, etc.	• Convert to XLA HLO Dialect • Identify **concurrency scopes** and dispatchable blocks of accelerable dense math • Convert dispatchable blocks to **executable binaries for target backends** • Emit code for **scheduling** dispatches • Wrap it all up in a **module**	• Dynamic module loader and **function invocation API** • **Hardware Abstraction Layer** (HAL) for easily plugging in hardware backends • (Optional) **interpreter** for the VM to allow flexible deployment

Towards Deep Pipelines

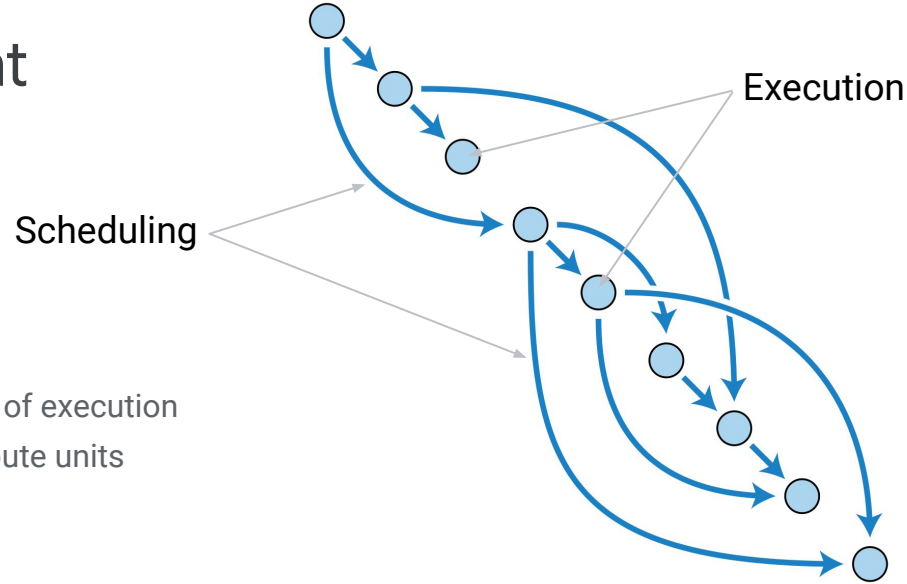
Goal is to allow dynamic power island/frequency adjustments to conserve power/thermals.

IREE is focused on **workloads that cannot rely on brute-force batch size increases** to achieve high utilization.

Example: What is ideal for a single-speaker voice-to-photon-optimized speech recognition system is not the same as what is ideal for a multi-TB multi-machine training workflow. Though it's possible to extend one to the other...

Scheduling 🤝 Execution

IREE's Key (not-so-novel) Insight



Scheduling: the ordering, dependency configuration, and timing of execution

Execution: the actual operations being executed on some compute units

Co-optimizing scheduling and execution enables greater concurrency and increased utilization.

Greedy Execution

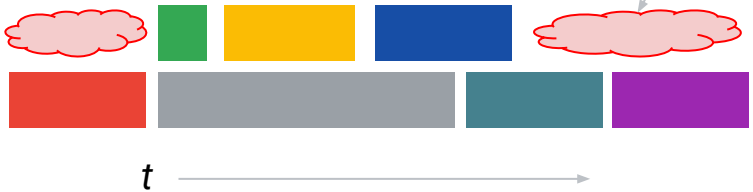
Conceptual Representation

Dataflow DAG

("scheduling")

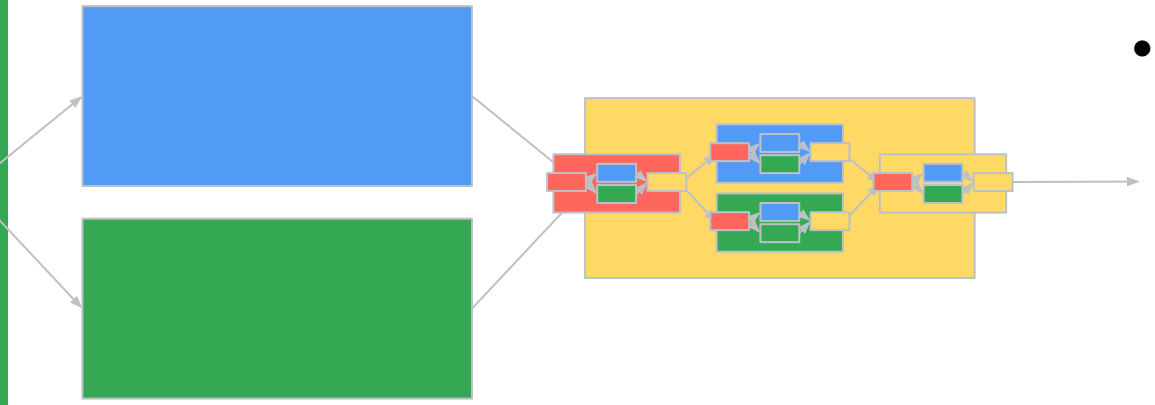
Bubble!

Actual Execution



("execution")

It’s dataflow graphs all the way down...



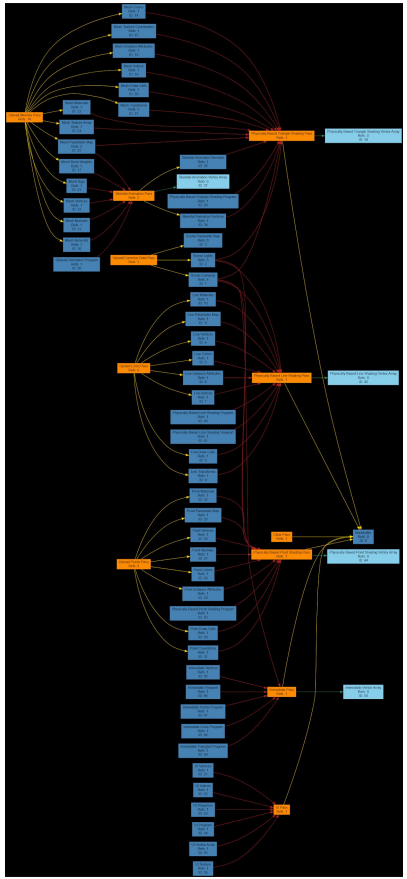
- Submissions
  - Command Buffers
    - Commands
      - Dispatches
        - Subgroups
          - Instructions

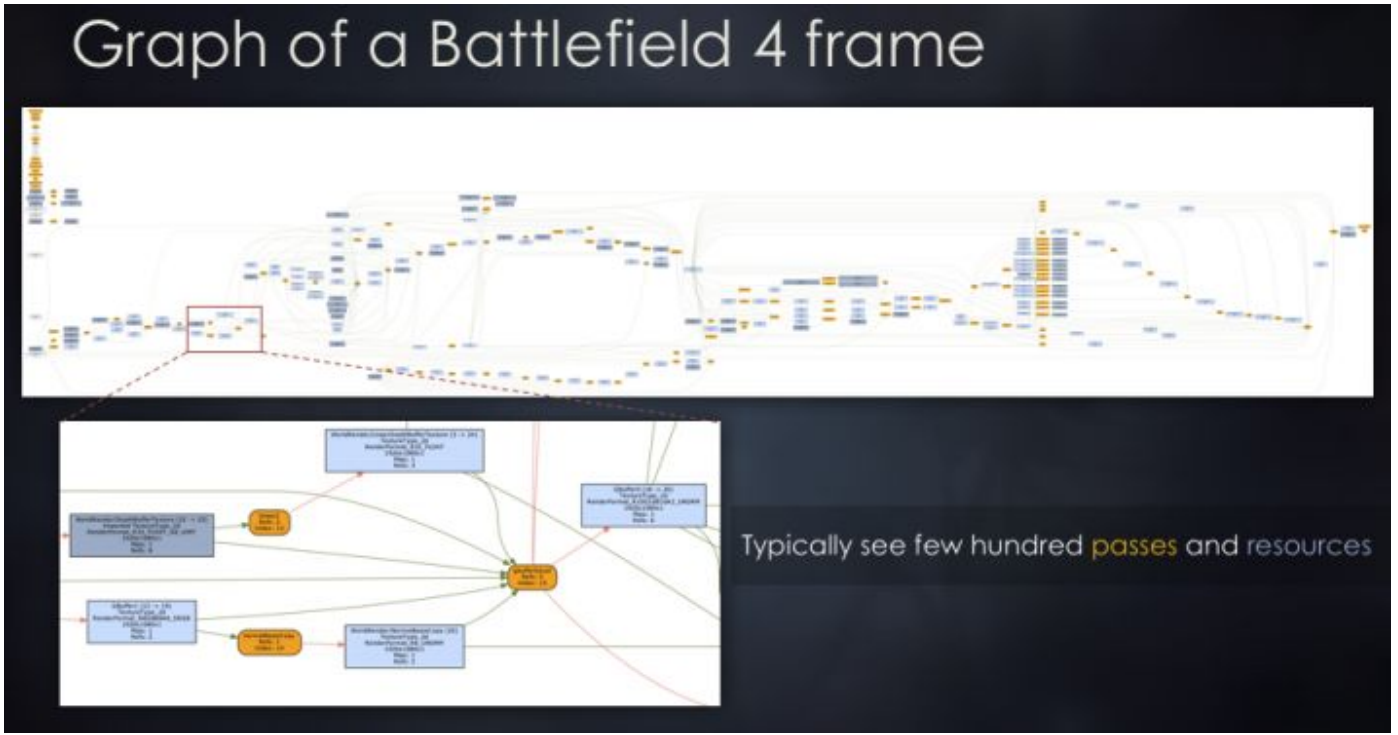
Scheduling hierarchy is just as critical to utilization as the memory hierarchy.
Latency hiding relies on effectively exploiting the scheduler.

Most current ML systems sidestep the scheduling hierarchy by making themselves compute or memory bound and ignoring latency hiding.

But games are examples of systems that take advantage of the hierarchy...

Job systems / task graphs / frame graphs...





[“Schedule early, complete late”]

Scheduling-Aware Allocation

Different allocators for different visibility scopes:

- Temporary memory used during execution is purely transient (stack/alloca)
- Temporary memory used across commands is transient within the command buffer (arena)
- Temporary memory used across command buffers is fenced (ringbuffer)
- Persistent memory can be whatever it needs to be:
  - External, long-lived
  - Ringbuffer if generational
  - Read-only/mapped
  - etc

Not aware, lots of duplication

Maximal reuse with no false dependencies

Modern Accelerator Scheduling

```
vkCmdPipelineBarrier(cmd, ...);
PopulateParamBuffer(input, output, params);
vkCmdPushConstants(cmd, … params);
size_x, size_y = texture_dimensions(input);
workgroup_xyz = {size_x / 16, size_y / 16, 1};
if (grayscale) {
  vkCmdBindPipeline(cmd, … pipeline_grayscale);
} else {
  vkCmdBindPipeline(cmd, … pipeline_color);
}
vkCmdDispatch(cmd, workgroup_xyz);
…
vkQueueSubmit(...);
```

Explicit barriers; no implicit synchronization

Parameters can be computed ahead of time

“Dynamic shapes” are naturally present

Conditional logic can happen during recording

Dispatch shape is independent from the pipeline

Nothing starts executing until after here

Vulkan/SPIR-V Principles Represented in IREE

- Clear split between executables and execution scheduling
  - SPIR-V (/MSL/DXIL/x86/etc) for **executables**
  - Vulkan (/Metal/D3D12/fibers/etc) for **scheduling**
- Favor forward job planning and cost amortization
  - Recoding command buffers; various pooling objects; etc
- Favor deep pipelines for async computation
  - Async and out-of-order execution in queues; fine-grained sync primitives
- Value hardware utilization, performance, and predictability
  - Multithreaded job preparation on CPU; no “magic” in drivers; runtime verification as extra layers

End-to-End Automation!

Interaction with other application logic?	Integrate with existing Vulkan context
Gain benefits from all existing accelerators?	Support multi-architecture modules
Split between work to dispatch on accelerators and scheduling logic?	Compiler magic! ✨
Better memory utilization?	Compiler magic! ✨
Better compute utilization?	Compiler magic! ✨
Better power management?	Compiler magic! ✨
Job scheduling & synchronization?	Vulkan-like hardware abstraction layer (HAL)
Various deployment environments?	HAL (⇒ VM) (⇒ Interpreter/C/LLVM)

IREE Compiler

IREE Dialects

- **Tensor value semantics** and basic math
- XLA HLO + std ops today, but could be any linear algebra-like dialect

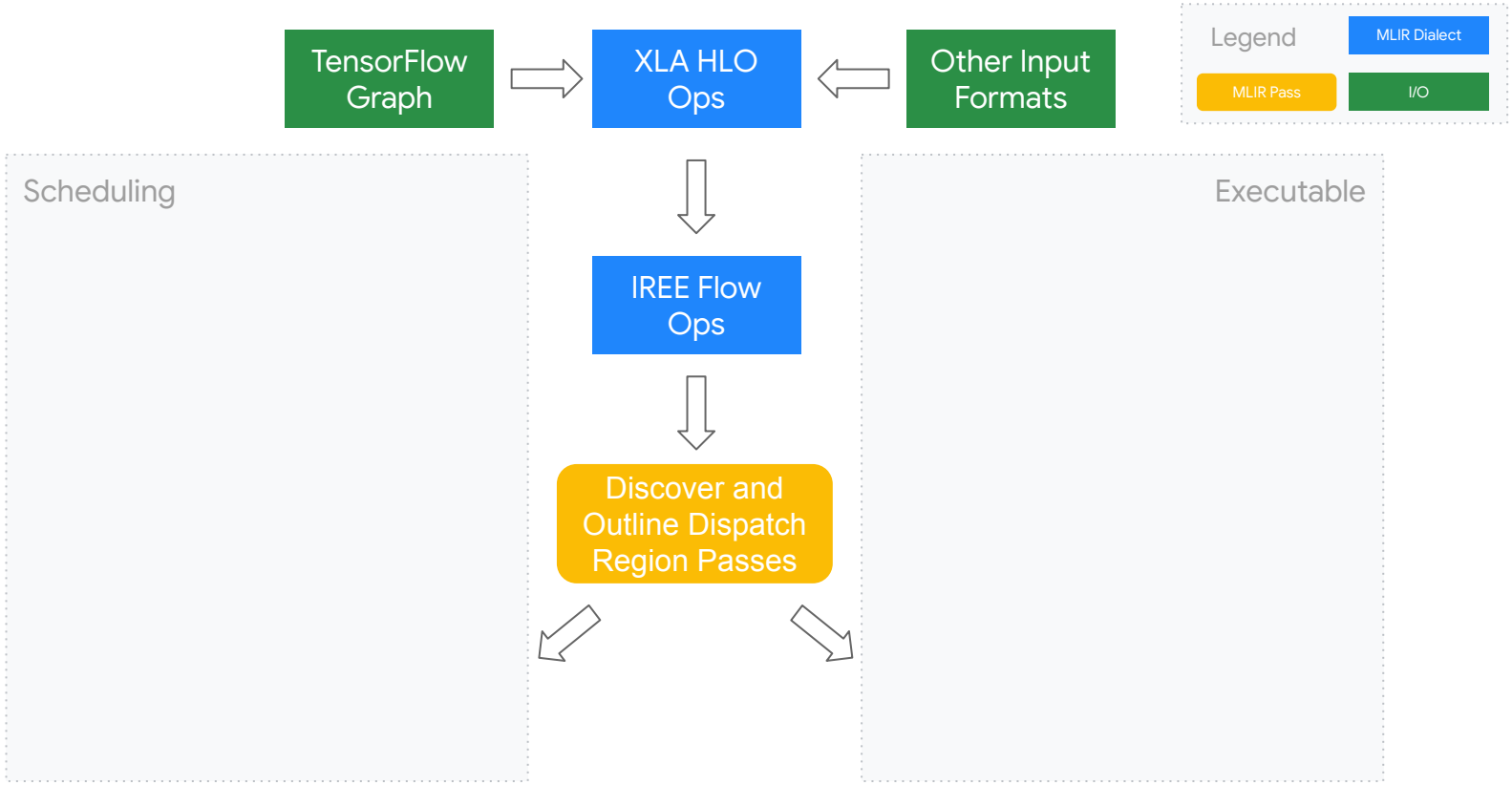
- Data and execution **flow** modeling
- **Splits scheduling from executable operations**

- Vulkan-like allocation and execution model encoding
- Executable compilation via **architecture specific backend compiler plugins**

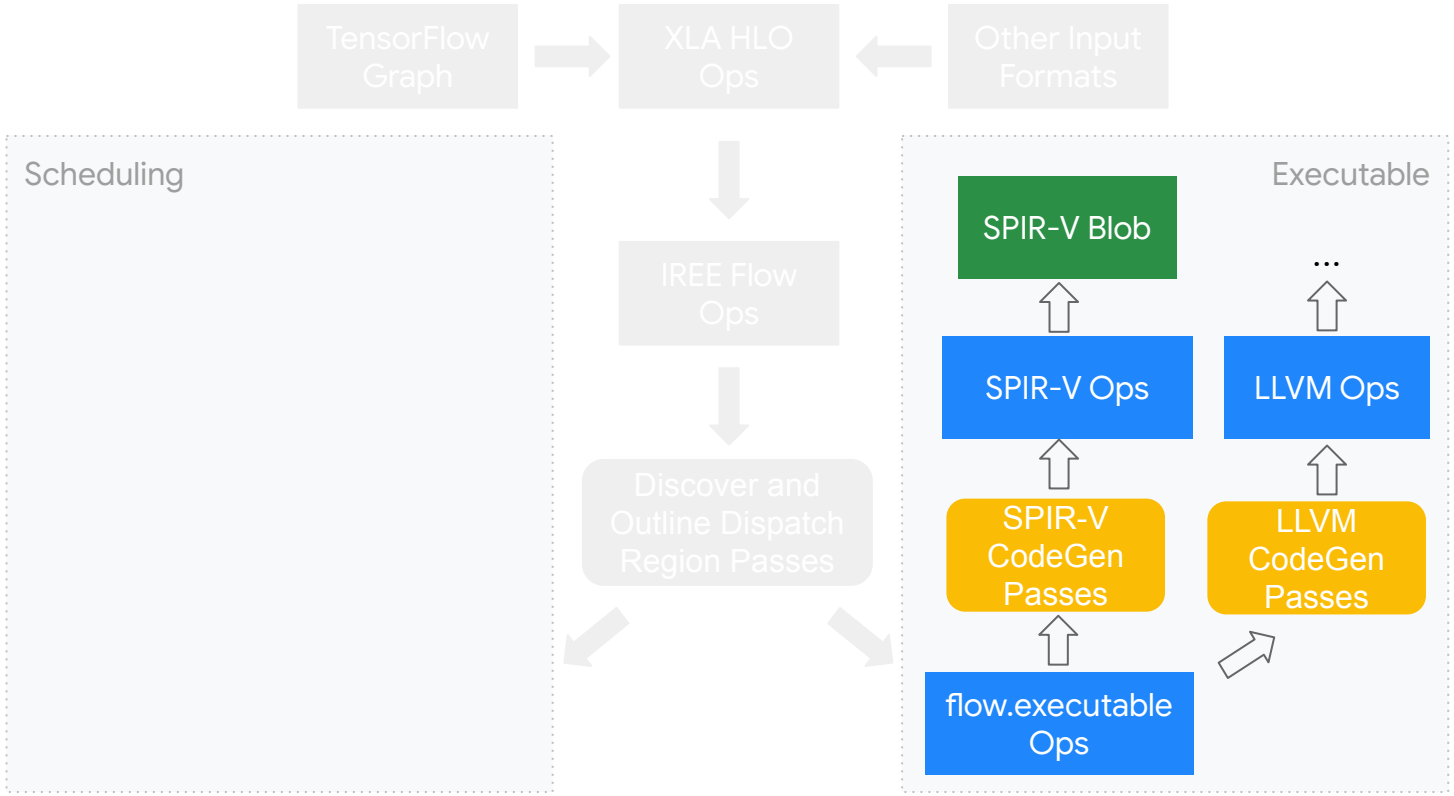
- Restricted virtual machine (no raw pointers, limited type support, etc)
- Dynamic module linkage definitions (imports, exports, globals, etc)
- **“It’s just code”** flexibility, ~2x **scheduling** perf cost

LLVM IR /
VM Bytecode /
WebAssembly /
etc

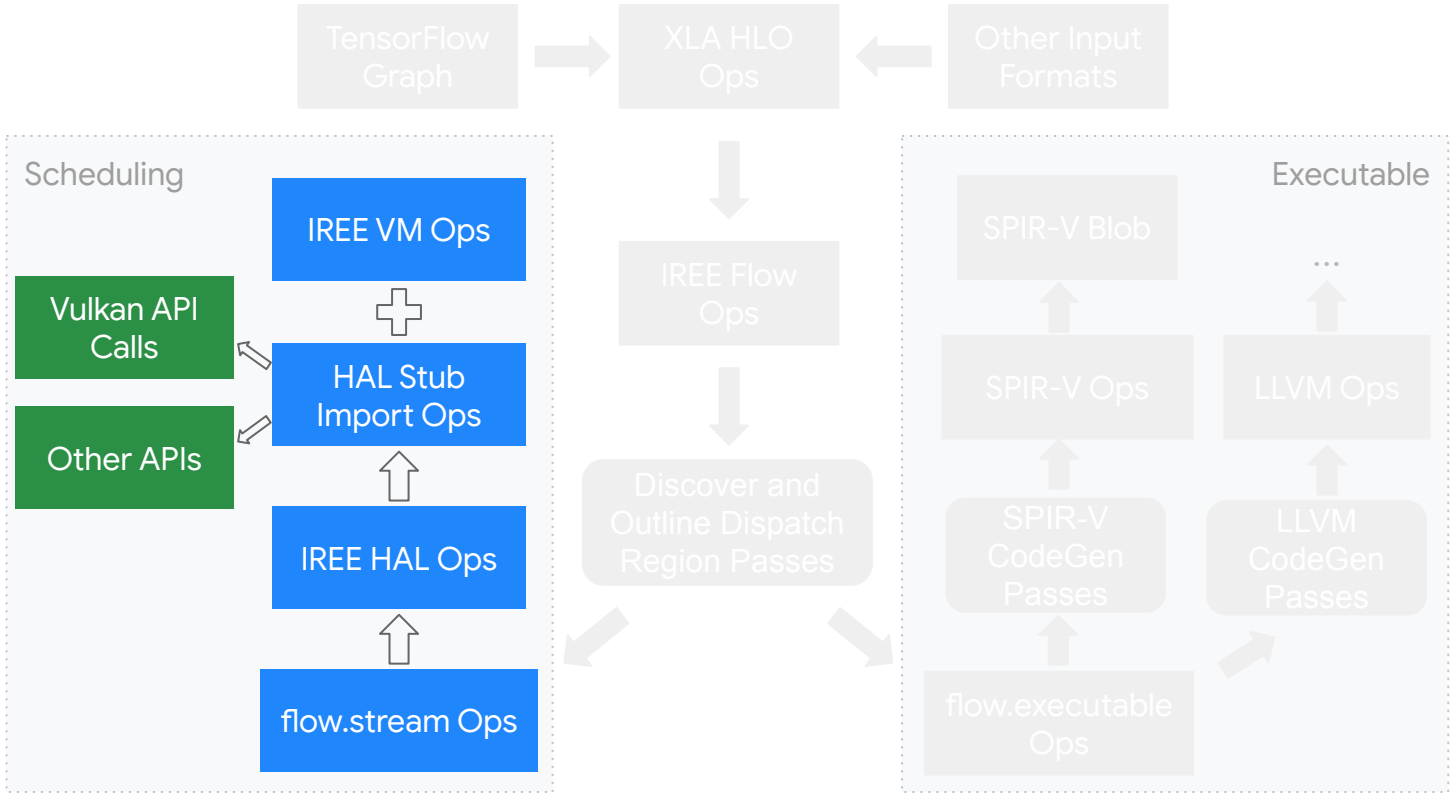
Compilation inside IREE



Compilation inside IREE



Compilation inside IREE



Source Models

```
class NormalModel(tf.Module):
  @tf.function(input_signature=[tf.TensorSpec([4], tf.float32)])
  def elementwise(self, a):
    return ((a + a) - a) * a
```

Currently we support an end-to-end flow for TensorFlow 2.0 **saved models**(**tf.Modules**). Other front-ends that can lower to XLA HLO in MLIR are easy to layer on top (contributions welcome!). This aspires to be a (pragmatically) complete modeling of core TensorFlow features, including statefulness, control flow, lists, structs, etc.:

```
class StatefulModel(tf.Module):
  def __init__(self):
    super(Stateful, self).__init__()
    self.counter = tf.Variable(0.0)

  @tf.function(input_signature=[tf.TensorSpec([], tf.float32)])
  def increment(self, x):
    self.counter.assign(self.counter + x)
```

IREE Input Dialects

```
module @NormalModule {
 func @hloElementwiseOps(%arg0: tensor<4xf32>) -> tensor<4xf32> attributes {iree.module.export} {
   %0 = xla_hlo.add %arg0, %arg0 : tensor<4xf32>
   %1 = xla_hlo.sub %0, %arg0 : tensor<4xf32>
   %2 = xla_hlo.mul %1, %arg0 : tensor<4xf32>
   return %2 : tensor<4xf32>
```

IREE **mostly passes ops through to backend compiler plugins** but has a few requirements:

- Control flow, shapes, etc are standardized
- Partitioning logic needs to know where to put ops (via dialect/op interfaces, etc)

The input can be a mix of dialects that match these constraints.

XLA HLO models a small stable set of math ops that we can reuse from TensorFlow and PyTorch frontends. Really excited about a future MLIR core dialect that targets a similar level of abstraction.

We're also investigating adding support for the Linalg dialect as an input to allow linalg to perform much of the partitioning work instead of doing it ourselves. **Hooray composition!**

IREE Flow Dialect

```
%0 = flow.ex.stream.fragment(%workload = %... : vector<3xi32>, %params = %... : tensor<4xf32>) -> tensor<4xf32> {
  %1 = flow.dispatch @model::@entry_point[%workload : vector<3xi32>](%params) : (tensor<4xf32>) -> tensor<4xf32>
  // (other dispatches, tensor subregion updates, slices, etc)
  flow.return %1 : tensor<4xf32>
}
```

A representation that **models the dependencies between streams and within streams** to define **concurrency domains**:

- Across streams (persistent buffers)
- Within streams (transient buffers)
- Within dispatch executables (workgroup memory)

Major compilation phases:

- Isolate dispatchable IR (that can be compiled by executable backends) from scheduling and data flow IR
- Outline the dispatchable IR into executables
- Form streams representing data flow and execution flow boundaries
- Perform stream placement based on target configuration

```
func @main(%arg0: tensor<1x5xf32>, %arg1: tensor<1x5x3x1xf32>) -> tensor<5x1x5xf32> {
 %0 = flow.dispatch.region[%workload_0 : vector<3xi32>](
                                %arg2 = %... : tensor<5x3xf32>, %arg3 = %... : tensor<3x5xf32>) -> tensor<5x5xf32> {
  %dot = "xla_hlo.dot"(%arg2, %arg3) {...} : (tensor<5x3xf32>, tensor<3x5xf32>) -> tensor<5x5xf32>
  flow.return %dot : tensor<5x5xf32>                                                    Executable to be outlined
 }
 %3 = flow.dispatch.region[%workload_1 : vector<3xi32>](
                                %arg2 = %... : tensor<5x5xf32>, %arg3 = %... : tensor<5x1xf32>) -> tensor<5x5xf32> {
  %cst = constant dense<1.0> : tensor<f32>
  %8 = "xla_hlo.broadcast_in_dim"(%arg3) {broadcast_dimensions = ...} : (tensor<5x1xf32>) -> tensor<5x1x5xf32>
  %10 = xla_hlo.mul %8, %... : tensor<5x1x5xf32>
  %11 = "xla_hlo.broadcast_in_dim"(%...)  : (tensor<f32>) -> tensor<5x1x5xf32>
  %12 = "xla_hlo.compare"(%10, %11) {comparison_direction = "GT"} : (...) -> tensor<5x1x5xi1>
  %15 = "xla_hlo.broadcast_in_dim"(%...) {broadcast_dimensions = ...} : (tensor<5xf32>) -> tensor<5x5xf32>
  %16 = xla_hlo.add %arg2, %15 : tensor<5x5xf32>
  %17 = xla_hlo.max %..., %16 : tensor<5x5xf32>
  %18 = "xla_hlo.reshape"(%17) : (tensor<5x5xf32>) -> tensor<5x1x5xf32>
  %19 = "xla_hlo.select"(%12, %..., %18) : (tensor<5x1x5xi1>, tensor<5x1x5xf32>, tensor<5x1x5xf32>) -> tensor<5x1x5xf32>
  %20 = "xla_hlo.copy"(%19) : (tensor<5x1x5xf32>) -> tensor<5x1x5xf32>
  %21 = "xla_hlo.reshape"(%20) : (tensor<5x1x5xf32>) -> tensor<5x5xf32>
  flow.return %21 : tensor<5x5xf32>                                                     Executable to be outlined
 }
 ...
 return %... : tensor<5x1x5xf32>
}
```

Example IREE Flow Dialect Ops

See FlowOps.td

- Scheduling ops
  - flow.dispatch
  - flow.stream.fragment
  - flow.stream.begin, flow.stream.append, flow.stream.end
- Region ops for inlined execution
  - flow.dispatch.region, flow.reduction.region
- Executable ops for outlined execution
  - flow.executable, flow.dispatch.entry, flow.reduction.entry
- Variable ops
  - flow.variable, flow.variable.load, flow.variable.store
- Small set of Tensor ops (which tend to be schedule-impacting)
  - flow.tensor.load, flow.tensor.store
  - flow.tensor.slice, flow.tensor.update

IREE HAL Dialect

```
hal.command_buffer.bind_descriptor_set %arg0, %0, set=0, %1, offsets=[%2]

hal.command_buffer.dispatch %arg0, %0, entry_point=0, workgroup_xyz=[%1, %2, %3]

%off, %len = hal.buffer_view.compute_range %arg0, shape=[%0#0, %0#1], indices=[%1#0, %1#1],
lengths=[%2#0, %2#1], element_size=4
```

Effectively a compute-only subset of the Vulkan API matching 1:1 **the IREE runtime HAL API represented in MLIR**. This allows us to perform whole-program optimization that would otherwise be done at runtime. This removes the requirement for complex (and unpredictable) guesswork at runtime related to caching, inferring usage semantics, and allocation. Note that runtime adaptation is still possible, but no longer a load-bearing part of the whole system.

Major compilation phases:

- Convert tensor values to buffers
- Convert Flow streams to command buffers
- Materialize synchronization primitives based on data flow analysis
- Compile executables to required target backends
- **Timeline semaphores/fences**

```
module {
  hal.executable @executable_module {
    hal.executable.entry_point @entry_point {ordinal=0, workgroup_size = dense<[32, 1, 1]> : vector<3xi32>}
    hal.executable.binary {data = dense<[...]> : vector<1620xi8>, format = "SPIR-V"} {
      spv.module "Logical" "GLSL450" ...
    }
  }
```

Executable: SPIR-V

```
  func @main(%arg0: !iree.ref<!hal.buffer>, %arg1: !iree.ref<!hal.buffer>) -> !iree.ref<!hal.buffer> {
    %dev = hal.ex.shared_device : !iree.ref<!hal.device>
    %allocator = hal.device.allocator %dev : !iree.ref<!hal.allocator>
    %buffer = hal.allocator.allocate.shaped %allocator, "HostVisible|DeviceVisible|DeviceLocal",
              "Constant|Transfer|Mapping|Dispatch", shape=[...], element_size=4 : !iree.ref<!hal.buffer>
    hal.ex.defer_release %buffer : !iree.ref<!hal.buffer>
    %cmd = hal.command_buffer.create %dev, "OneShot", "Transfer|Dispatch" : !iree.ref<!hal.command_buffer>
    hal.command_buffer.begin %cmd
    %exe = hal.ex.cache_executable %dev, @executable_module : !iree.ref<!hal.executable>
    hal.ex.push_binding %cmd, 0, %arg1, shape=[%c1_i32, %c5_i32, %c3_i32, %c1_i32], element_size=4
    hal.ex.defer_release %arg1 : !iree.ref<!hal.buffer>
    hal.command_buffer.dispatch %cmd, %exe, entry_point=0, workgroup_xyz=[%c1_i32, %c5_i32, %c1_i32]
    %memory_barrier = hal.make_memory_barrier "DispatchWrite", "DispatchRead" : tuple<i32, i32>
    hal.command_buffer.execution_barrier %cmd, "CommandRetire", "CommandIssue",
                                                    memory_barriers=[%memory_barrier]
    %exe_1 = hal.ex.cache_executable ...
    ...
    hal.command_buffer.end %cmd
    hal.ex.submit_and_wait %dev, %cmd
    return %buffer : !iree.ref<!hal.buffer>
  }
}
```

Scheduling: Vulkan

Example IREE HAL Dialect Ops

See HALOps.td

- Command-buffer ops
  - hal.command_buffer.{create|begin|end|...}
  - hal.command_buffer.dispatch.*
- Memory management ops
  - hal.allocator.allocate.*
  - hal.buffer.*, hal.buffer_view.*
- Synchronization ops
  - hal.make_memory_barrier, hal.make_buffer_barrier
  - hal.command_buffer.execution_barrier
  - Barriers, fences, semaphores
- Hardware object ops
  - Devices, executables, descriptors, caches, etc.

IREE VM Dialect

Code: VMOps.td

```
vm.func @cmp_lt_i32_s(%arg0 : i32, %arg1 : i32) -> i32 {
  %slt = vm.cmp.lt.i32.s %arg0, %arg1 : i32
  vm.return %slt : i32
}
```

A simple virtual machine designed to be **fast (enough) to interpret without further lowering**. This allows us to produce machine-independent modules that can be deployed out-of-band with application binaries, be verified in-depth (statically and dynamically), and run the scheduling logic on remote devices (sidecar MCUs, persistent kernels on GPUs, etc). **Co-routines natively supported for efficient cooperative scheduling (and cellular batching)**.

Easy to lower to other forms, if needed:

- LLVM IR
- Textual C (to export to other compilers/toolchains)
- WASM (no need to go through LLVM IR if binary size is a concern)
- SPIR-V (for persistent kernels)

(it's possible to use IREE's Flow and HAL dialects without lowering through it, but the IREE dynamic runtime uses it)

```
vm.module {
  vm.rodata @executable_module dense<[...]> : vector<1620xi8>
  vm.global.ref @executable_cached mutable : !iree.ref<!hal.executable>
  vm.rodata @embedded_rodata dense<[...]> : tensor<3x5xf32>
```

Constants and variables

```
  vm.func @main(%arg0: !iree.ref<!hal.buffer>, %arg1: !iree.ref<!hal.buffer>) -> !iree.ref<!hal.buffer> {
    %device = vm.call @hal.ex.shared_device() : () -> !iree.ref<!hal.device>
    %alloc = vm.call @hal.device.allocator(%device) : (!iree.ref<!hal.device>) -> !iree.ref<!hal.allocator>
    %buffer = vm.call.variadic @hal.allocator.allocate.shaped(%alloc, ...) : (...) -> !iree.ref<!hal.buffer>
    vm.call @hal.ex.defer_release(%buffer) : (!iree.ref<!hal.buffer>) -> ()
    %cmd = vm.call @hal.command_buffer.create(...) : (!iree.ref<!hal.device>, ...) -> !iree.ref<!hal.command_buffer>
    vm.call @hal.command_buffer.begin(%cmd) : (!iree.ref<!hal.command_buffer>) -> ()
    %exe = vm.call @find_cached_executable(...) : (!iree.ref<!hal.device>) -> !iree.ref<!hal.executable>
    vm.call.variadic @hal.ex.push_binding(...) : (...)
    vm.call @hal.ex.defer_release(...) : (!iree.ref<!hal.buffer>) -> ()
    vm.call @hal.command_buffer.dispatch(%cmd, %exe, ...) : (...) -> ()
    vm.call.variadic @hal.command_buffer.execution_barrier(%cmd, ...) : (...)
    vm.import @hal.command_buffer.end(%command_buffer : !iree.ref<!hal.command_buffer>)
    vm.call @hal.ex.submit_and_wait(%device, %cmd) : (!iree.ref<!hal.device>, !iree.ref<!hal.command_buffer>) -> ()
    vm.return %buffer : !iree.ref<!hal.buffer>
  }
```

VM scheduling

```
  vm.export @main
  vm.import @hal.ex.submit_and_wait(%device : !iree.ref<!hal.device>, %command_buffer : !iree.ref<!hal.command_buffer>)
  ...
}
```

Imports and exports

MLIR Experience Thus Far

For the first time we can reason about co-optimizing scheduling and execution.

Relatively small amount of code! Core improvements are reducing boilerplate.

Reusing core dialects (linalg, lowerings to LLVM IR, etc) has massive benefit and allows focus.

Tools built on MLIR become part of a toolkit, not an island.

Project Status

Tentative Milestones in 2020

Alpha; usable for some situations but API not yet stable and most optimizations still TBD

General Availability; usable for shipping apps, focus on refinement and additional support

Q1 Q2 Q3 Q4

Experimental; focused on functionality and covering core use cases

Beta; API stable and focus is on optimization and widening platform support

RFC

- Github: https://github.com/google/iree
  - Feature request issues welcome! Let us know what you want :)
  - Core dialects: https://github.com/google/iree/tree/master/iree/compiler/Dialect
  - HAL: https://github.com/google/iree/blob/master/iree/hal/api.h
  - Samples: https://github.com/google/iree/tree/master/iree/samples
  - Colab notebooks showing python->execution: https://github.com/google/iree/tree/master/colab (note: pip not installed on public colab servers yet -- requires custom kernel launch)
- Mailing list
  - iree-discuss@googlegroups.com
  - https://groups.google.com/forum/#!forum/iree-discuss