Bringing a new ML-Runtime backend in OpenXLA/IREE

Using LevelZero as Cast Study.

Stanley Winata (nod.ai)

Overview

- Motivation
- Enter SPIR-V
- Enters SHARK-Runtime
- Case Study with Implementing Level Zero HAL

Motivation: Common Existing ML stack challenges

- Monolithic frameworks containing tightly coupled components
- Overfitting to initial key use cases and target hardware architectures
  - E.g., assuming data center training; following NVIDA/CUDA closely for runtime
- Irregular support for different hardware; optionality often an afterthought
  - Extensively depending on vendor-specific solutions, at a high level
- Extensive manual (duplicated) work to support new workload
  - Regardless of existing supported hardware or new ones

Original slides by Lei Zhang

Motivation: Unscalable for various Stakeholders

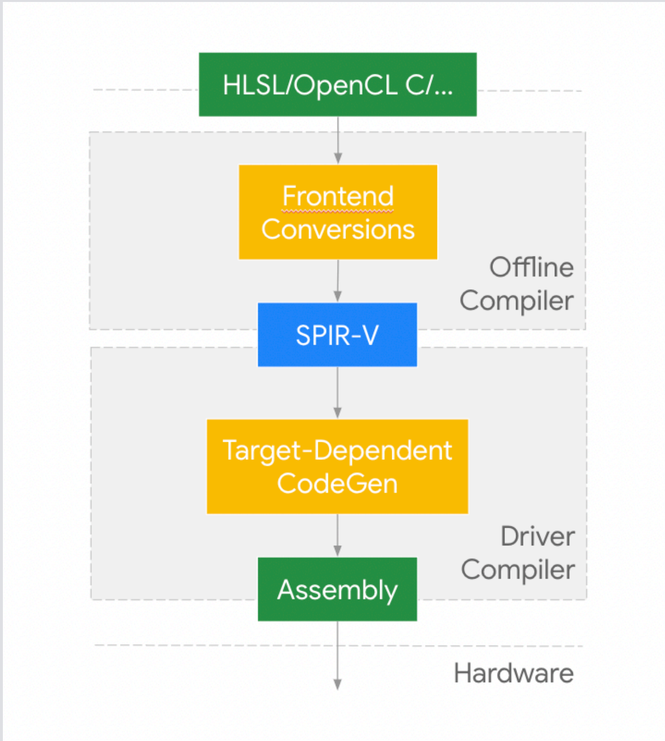
- With ML still exploding..
- Frameworks becoming more and more complex to maintain
- New use cases to support, new hardwares to enable, performance hacks to throw in, etc.
- Hardware vendors stretching resources to catch up
  - Everyone needs to develop a full stack equivalent to CUDA, cuBLAS, cuDNN, etc.
  - Fighting with {model/op x shape x precision x layout x architecture x …} explosion

Original slides by Lei Zhang

Enters SPIR-V

Introduction

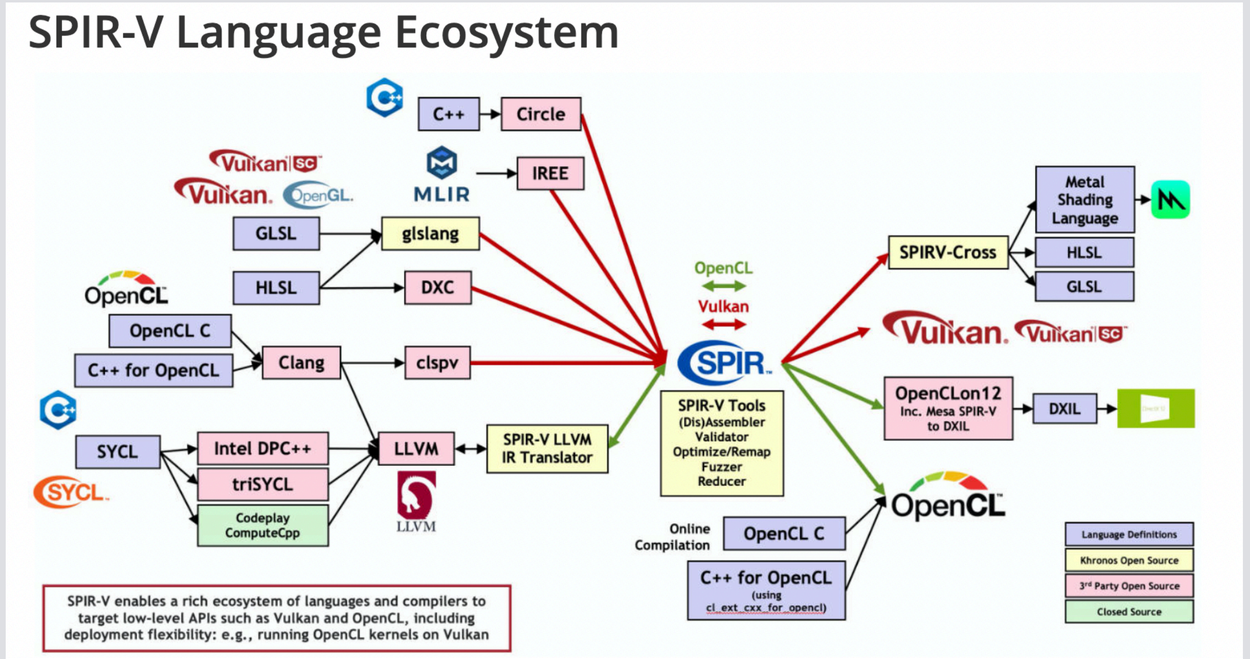
- SPIR-V is a low-level IR that represent GPU computation.
- Hardware vendor can write implementation of SPIR-V spec for their hardware to encourage **stability** and **compatibility**.
- SPIR-V can also take in hardware specific extensions for optimizations (i.e NV_COOPERATIVE_MATRIX)



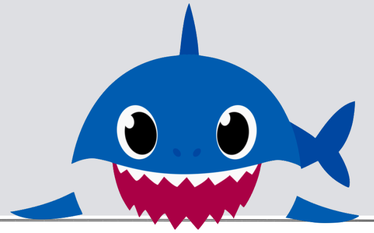
Enters SPIR-V

Portability

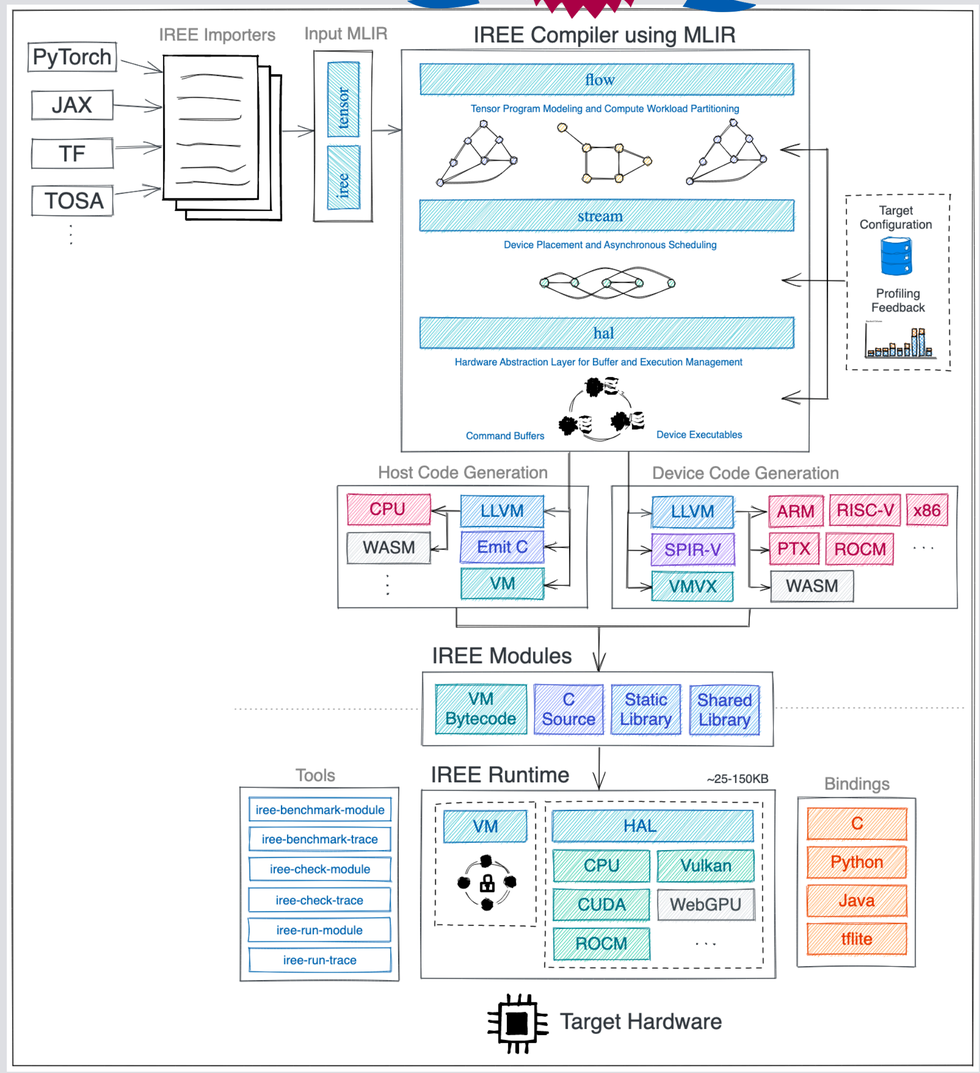
- A **common lowering point** for targeting more hardware and platform-specific APIs:
  - Vulkan/GLSL → AMD GPU, NVIDIA GPU, Mobile device
  - OpenCL/Kernel → Intel GPU, NVIDIA GPU
  - Vulkan/MoltenVK → MacOS and IOS devices.
  - SPIRV-Cross/Tint → WGSL, for WebGPU on web
  - WGSL: designed to be trivially translatable from/to SPIR-V
  - SPIRV-Cross/Tint → MSL, for Metal on Apple platforms
  - SPIRV-Cross/Tint → HLSL, for DirectX on Microsoft platform



Enters SHARK-Runtime

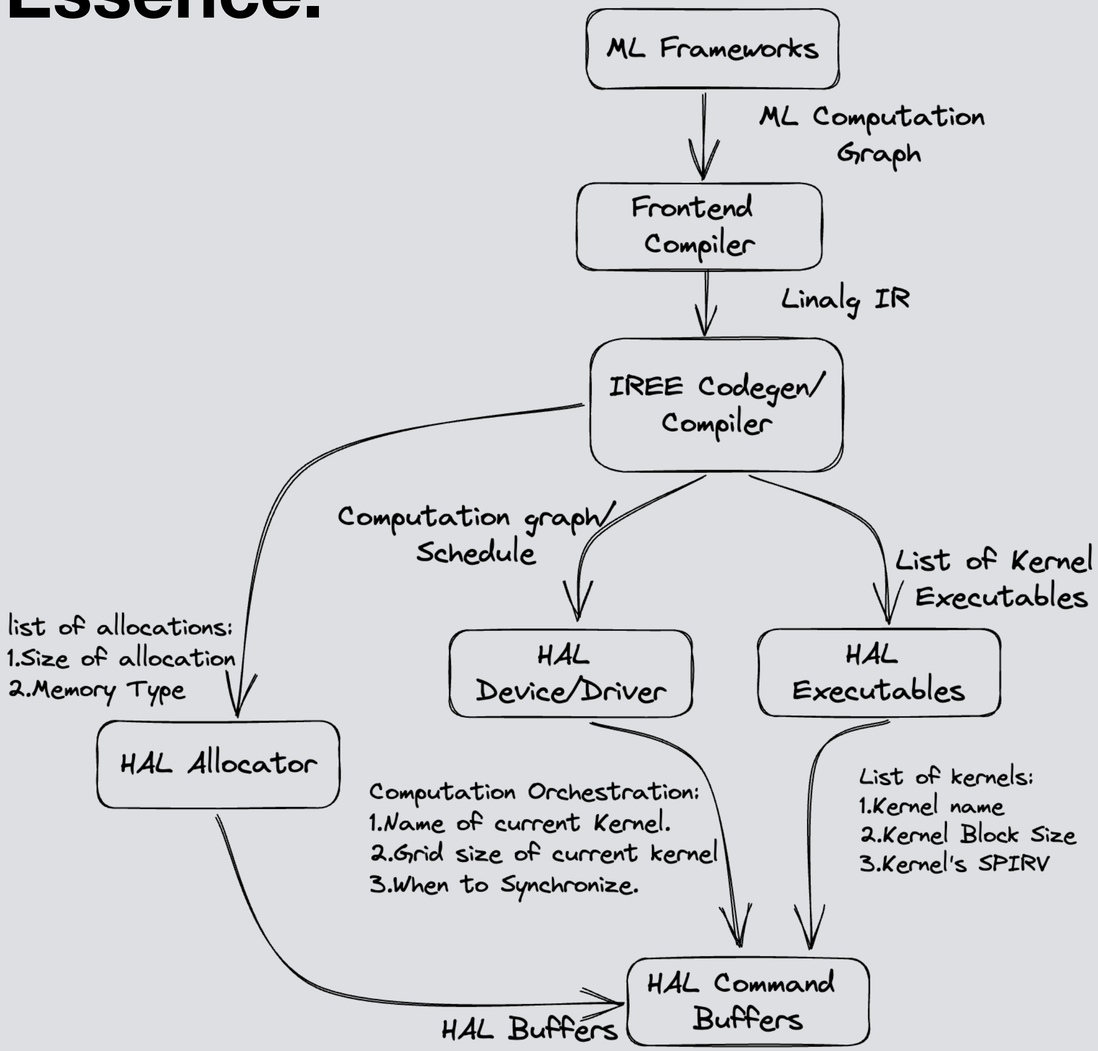


- IREE is an open-source MLIR-based end-to-end compiler and runtime that lowers ML models for datacenter and edge workload.
- SHARK-Runtime is a Nod's fork of OpenXLA/IREE with experimental performance features and hardware support.
- Battle tested layering and organization structure; very modular design.
- Supports X86, NVIDIA, AMD, RISC-V, SPIR-V and ARM
- Supports Tensorflow, JAX, TFLite, PyTorch
- Active ecosystem with tooling, knowledge sharing, etc.
- SHARK-SPIR-V supports various platforms of different form factors. (i.e Qualcomm Adreno, Apple M family chips, Intel/ Nvidia/AMD datacenter GPUs)



Enters SHARK-Runtime

Runtime in an Essence.



HAL: Hardware Abstraction Layer

- HAL is similar to a runtime abstraction. If we implement the abstraction with target backend APIs, we will be able to use the target backend in IREE right away. It contain these components:
- Driver
- Devices
- Allocator
- Buffers
- Command Buffers
- Executable
- Semaphores
- All HAL backends validated through standard tests by CTS.

HAL: Simple Add Example

```
func.func @add(%arg0: !hal.buffer_view, %arg1: !hal.buffer_view) -> !hal.buffer_view attributes {iree.abi.stub} {
  %c-1_i32 = arith.constant -1 : i32
  %c1_i64 = arith.constant 1 : i64
  %c1 = arith.constant 1 : index
  %c-1_i64 = arith.constant -1 : i64
  %c0 = arith.constant 0 : index
  %c32 = arith.constant 32 : index
  %c2 = arith.constant 2 : index
  %c4 = arith.constant 4 : index
  %c553648160_i32 = arith.constant 553648160 : i32
  %c1_i32 = arith.constant 1 : i32
  %_timeline_value = util.global.load @_timeline_value : i64
  %_timeline_semaphore = util.global.load @_timeline_semaphore : !hal.semaphore
  %_device_query_0 = util.global.load @_device_query_0 : i1
  %_pipeline_layout_0 = util.global.load @_pipeline_layout_0 : !hal.pipeline_layout
  %_executable_add_dispatch_0 = util.global.load @_executable_add_dispatch_0 : !hal.executable
  hal.buffer_view.assert<%arg0 : !hal.buffer_view> message("tensor") shape([%c2, %c4]) type(%c553648160_i32) encoding(%c1_i32)
  %buffer = hal.buffer_view.buffer<%arg0 : !hal.buffer_view> : !hal.buffer
  %device = hal.ex.shared_device : !hal.device
  %allocator = hal.device.allocator<%device : !hal.device> : !hal.allocator
  hal.buffer.assert<%buffer : !hal.buffer> message("tensor") allocator(%allocator : !hal.allocator) minimum_length(%c32) type(DeviceVisible) usage("TransferSource|
  hal.buffer_view.assert<%arg1 : !hal.buffer_view> message("tensor") shape([%c2, %c4]) type(%c553648160_i32) encoding(%c1_i32)
  %buffer_0 = hal.buffer_view.buffer<%arg1 : !hal.buffer_view> : !hal.buffer
  hal.buffer.assert<%buffer_0 : !hal.buffer> message("tensor") allocator(%allocator : !hal.allocator) minimum_length(%c32) type(DeviceVisible) usage("TransferSourc
  %buffer_1 = hal.allocator.allocate<%allocator : !hal.allocator> type("HostVisible|DeviceVisible|DeviceLocal") usage("TransferSource|TransferTarget|Transfer|Dispa
  %cmd = hal.command_buffer.create device(%device : !hal.device) mode("OneShot|AllowInlineExecution") categories("Transfer|Dispatch") : !hal.command_buffer
  cf.cond_br %_device_query_0, ^bb1, ^bb2
^bb1:  // pred: ^bb0
  hal.command_buffer.push_descriptor_set<%cmd : !hal.command_buffer> layout(%_pipeline_layout_0 : !hal.pipeline_layout)[%c0] bindings([
    %c0 = (%buffer : !hal.buffer)[%c0, %c32],
    %c1 = (%buffer_0 : !hal.buffer)[%c0, %c32],
    %c2 = (%buffer_1 : !hal.buffer)[%c0, %c32]
  ])
  hal.command_buffer.dispatch<%cmd : !hal.command_buffer> target(%_executable_add_dispatch_0 : !hal.executable)[0] workgroups([%c1, %c2, %c1])
  hal.command_buffer.execution_barrier<%cmd : !hal.command_buffer> source("Dispatch|Transfer|CommandRetire") target("CommandIssue|Dispatch|Transfer") flags("None")
  hal.command_buffer.finalize<%cmd : !hal.command_buffer>
  %0 = util.null : !hal.fence
  %1 = arith.addi %_timeline_value, %c1_i64 : i64
  %fence = hal.fence.create at<%_timeline_semaphore : !hal.semaphore>(%1) -> !hal.fence
  hal.device.queue.execute<%device : !hal.device> affinity(%c-1_i64) wait(%0) signal(%fence) commands([%cmd])
  util.global.store %1, @_timeline_value : i64
  %status = hal.fence.await until([%fence]) timeout_millis(%c-1_i32) : i32
  util.status.check_ok %status, "failed to wait on timepoint"
  %view = hal.buffer_view.create buffer(%buffer_1 : !hal.buffer)[%c0, %c32] shape([%c2, %c4]) type(%c553648160_i32) encoding(%c1_i32) : !hal.buffer_view
  return %view : !hal.buffer_view
```

HAL: Device and Driver

- Drivers provide device enumeration
  - Driver examples: CUDA, Vulkan, local CPU
  - Available drivers change over time: external nvidia GPU plugged in, remote host available
- Devices model logical resource/execution scopes
  - Device examples: GPU 0, GPU 1, GPU 0+1 (mirrored), CPU package 0
  - Available devices change over time: discrete GPU powered on, pooled device unused
- Devices are primarily factories for resources like buffers and executables
- Capability query interface to allow compiled modules to select/adapt
- Hosting applications can wrap existing device handles (thread pools, VkDevice, CUdevice, etc) for interoperability
- Note: a CPU is a device! (more later)

Original slides by Ben Vanik

HAL: Device and Driver

MLIR:

```
%device = hal.ex.shared_device : !hal.device
%allocator = hal.device.allocator<%device : !hal.device> : !hal.allocator
hal.buffer.assert<%buffer : !hal.buffer> message("tensor") allocator(%allocator : !hal.allocator) minimum_length(%c32) type(DeviceVisible) usage("TransferS
hal.buffer_view.assert<%arg1 : !hal.buffer_view> message("tensor") shape([%c2, %c4]) type(%c553648160_i32) encoding(%c1_i32)
%buffer_0 = hal.buffer_view.buffer<%arg1 : !hal.buffer_view> : !hal.buffer
hal.buffer.assert<%buffer_0 : !hal.buffer> message("tensor") allocator(%allocator : !hal.allocator) minimum_length(%c32) type(DeviceVisible) usage("Transfe
%buffer_1 = hal.allocator.allocate<%allocator : !hal.allocator> type("HostVisible|DeviceVisible|DeviceLocal") usage("TransferSource|TransferTarget|Transfer
%cmd = hal.command_buffer.create device(%device : !hal.device) mode("OneShot|AllowInlineExecution") categories("Transfer|Dispatch") : !hal.command_buffer
```

C Impl:

```
typedef struct iree_hal_level_zero_driver_t {
  iree_hal_resource_t resource;
  iree_allocator_t host_allocator;
  // Identifier used for the driver in the IREE driver registry.
  // We allow overriding so that multiple LevelZero versions can be exposed in
  // the same process.
  iree_string_view_t identifier;
  int default_device_index;

  // Level Zero Driver Handle.
  ze_driver_handle_t driver_handle;
  ze_context_handle_t context;
  // LevelZero symbols.
  iree_hal_level_zero_dynamic_symbols_t syms;
} iree_hal_level_zero_driver_t;
```

```
typedef struct iree_hal_level_zero_device_t {
  iree_hal_resource_t resource;
  iree_string_view_t identifier;

  // Block pool used for command buffers with a larger block size (as command
  // buffers can contain inlined data uploads).
  iree_arena_block_pool_t block_pool;

  // Optional driver that owns the Level Zero symbols. We retain it for our
  // lifetime to ensure the symbols remains valid.
  iree_hal_driver_t* driver;

  // Level Zero APIs.
  ze_device_handle_t device;
  uint32_t command_queue_ordinal;
  ze_command_queue_handle_t command_queue;
  ze_event_pool_handle_t event_pool;

  iree_hal_level_zero_context_wrapper_t context_wrapper;
  iree_hal_allocator_t* device_allocator;

} iree_hal_level_zero_device_t;
```

HAL: Allocator

- All buffer handles obtained via a device-provided allocator handle:
- Explicit host/device placement and visibility control
- Wrap host buffers (void*) if able
- Import/export for interoperability (import VkBuffer, etc)
- Buffer constraint queries (min alignment, max size, etc)
- Implementation can be simple (malloc/free), perform pooling (VMA for Vulkan), or defer to external allocators reusing application memory or memory shared across devices.
- Compiler emits VM code to perform tensor-level packing and allocation.

Original slides by Ben Vanik

HAL: Allocator

MLIR:
```
%buffer_1 = hal.allocator.allocate<%allocator : !hal.allocator>
```

C Impl:

LevelZero Allocating Buffer

```
if (iree_all_bits_set(params->type, IREE_HAL_MEMORY_TYPE_DEVICE_LOCAL)) {
  // Device local case.
  if (iree_all_bits_set(params->type, IREE_HAL_MEMORY_TYPE_HOST_VISIBLE)) {
    status = LEVEL_ZERO_RESULT_TO_STATUS(
        allocator->context->syms,
        zeMemAllocShared(allocator->context->level_zero_context,
                         &memAllocDesc, &hostDesc, allocation_size,
                         alloc_alignment, allocator->level_zero_device,
                         (void**)&device_ptr));
    host_ptr = (void*)device_ptr;
  } else {
    // Device only.
    status = LEVEL_ZERO_RESULT_TO_STATUS(
        allocator->context->syms,
        zeMemAllocDevice(allocator->context->level_zero_context,
                         &memAllocDesc, allocation_size, alloc_alignment,
                         allocator->level_zero_device, (void**)&device_ptr));
  }
} else {
  // Since in Level Zero host memory is visible to device, we can simply
  // allocate on host and set device_ptr to point to same data.
  status = LEVEL_ZERO_RESULT_TO_STATUS(
      allocator->context->syms,
      zeMemAllocHost(allocator->context->level_zero_context, &hostDesc,
                     allocation_size, 64, &host_ptr));
  device_ptr = (iree_hal_level_zero_device_ptr_t)host_ptr;
}
```

SHARK-Runtime/experimental/level_zero/level_zero_allocator.c

LevelZero Freeing buffer

```
static void iree_hal_level_zero_buffer_free(
    iree_hal_level_zero_context_wrapper_t* context,
    iree_hal_memory_type_t memory_type,
    iree_hal_level_zero_device_ptr_t device_ptr, void* host_ptr) {
  if (iree_all_bits_set(memory_type, IREE_HAL_MEMORY_TYPE_DEVICE_LOCAL)) {
    // Device local.
    LEVEL_ZERO_IGNORE_ERROR(context->syms,
                            zeMemFree(context->level_zero_context, device_ptr));
  } else {
    // Host local.
    LEVEL_ZERO_IGNORE_ERROR(context->syms,
                            zeMemFree(context->level_zero_context, host_ptr));
  }
}
```

SHARK-Runtime/experimental/level_zero/level_zero_allocator.c

HAL: Buffers

- Memory type:
  - HOST|DEVICE_LOCAL: where the buffer lives
  - HOST|DEVICE_VISIBLE: who can access it
  - TRANSIENT: queue-local pooled memory
- Allowed Usage: CONSTANT|TRANSFER|MAPPING|DISPATCH
- Allowed Access: READ|WRITE
- Mapping: explicit checked map/unmap to get host pointer access
  - Random void* access is a non-optimal convenience for humans
  - Invalidation and flushing critical for non-coherent memory types

Original slides by Ben Vanik

HAL: Buffers

- Purpose: To wrap a struct around device pointer for abstraction, to be usable with rest of HAL.

```
typedef struct iree_hal_level_zero_buffer_t {
  iree_hal_buffer_t base;
  void* host_ptr;
  iree_hal_level_zero_device_ptr_t device_ptr;
} iree_hal_level_zero_buffer_t;
```

SHARK-Runtime/experimental/level_zero/level_zero_buffer.c

HAL: Command Buffer

- Reusable recordings of a sequence of commands to execute. Like a CUstream that can be replayed.
- Execution order and concurrency defined by synchronization commands.
- Primary/secondary nesting: primary is dynamic, secondary is static and reused.
- Runtime can be totally asynchronous (dependent on graph and fence insertion)
- Commands:
- Synchronization: ExecutionBarrier, SetEvent/WaitEvent
- DMA: CopyBuffer, FillBuffer, UpdateBuffer
- Execution: Dispatch, DispatchIndirect, Execute (command buffer)
- Recording-only state: PushConstants, BindDescriptorSet

Original slides by Ben Vanik

HAL: Command Buffer

MLIR:

```
%cmd = hal.command_buffer.create device(%device : !hal.device) mode("OneShot|AllowInlineExecution") categories("Transfer|Dispatch") : !hal.command_buffer
```

C Impl:

```
typedef struct {
  iree_hal_command_buffer_t base;
  iree_hal_level_zero_context_wrapper_t* context;
  iree_arena_block_pool_t* block_pool;
  ze_command_list_handle_t command_list;

  // Keep track of the current set of kernel arguments.
  int32_t push_constant[IREE_HAL_LEVEL_ZERO_MAX_PUSH_CONSTANT_COUNT];
  void* current_descriptor[];
} iree_hal_level_zero_direct_command_buffer_t;
```

SHARK-Runtime/experimental/level_zero/direct_command_buffer.c

HAL: Command Buffer

MLIR:

```
#hal.device.match.executable.format<"opencl-spirv-fb"> {
  %pipeline_layout = hal.pipeline_layout.lookup device(%device : !hal.device) layout(#pipeline_layout) :
  hal.command_buffer.push_descriptor_set<%cmd : !hal.command_buffer> layout(%pipeline_layout : !hal.pipel
    %c0 = (%buffer : !hal.buffer)[%c0, %c32],
    %c1 = (%buffer_0 : !hal.buffer)[%c0, %c32],
    %c2 = (%buffer_1 : !hal.buffer)[%c0, %c32]
  ])
  hal.command_buffer.dispatch.symbol<%cmd : !hal.command_buffer> target(@add_dispatch_0::@opencl_spirv_fb
  hal.return
}
hal.command_buffer.execution_barrier<%cmd : !hal.command_buffer> source("Dispatch|Transfer|CommandRetire"
hal.command_buffer.finalize<%cmd : !hal.command_buffer>
```

C Impl:

```
// access proper stream from command buffer
LEVEL_ZERO_RETURN_IF_ERROR(
    command_buffer->context->syms,
    zeKernelSetGroupSize(func, block_size_x, block_size_y, block_size_z),
    "zeKernelSetGroupSize");

// Patch the push constants in the kernel arguments.
for (iree_host_size_t i = 0; i < num_constants; i++) {
  *((uint32_t*)command_buffer->current_descriptor[i + constant_base_index]) =
      command_buffer->push_constant[i];
}
iree_host_size_t num_kernel_args = constant_base_index + num_constants;
for (iree_host_size_t i = 0; i < num_kernel_args; i++) {
  LEVEL_ZERO_RETURN_IF_ERROR(
      command_buffer->context->syms,
      zeKernelSetArgumentValue(func, i,
                               sizeof(command_buffer->current_descriptor[i]),
                               command_buffer->current_descriptor[i]),
      "zeKernelSetArgumentValue");
}

// Kernel thread-dispatch
ze_group_count_t dispatch;
dispatch.groupCountX = workgroup_x;
dispatch.groupCountY = workgroup_y;
dispatch.groupCountZ = workgroup_z;

// Launch kernel on the GPU
LEVEL_ZERO_RETURN_IF_ERROR(
    command_buffer->context->syms,
    zeCommandListAppendLaunchKernel(command_buffer->command_list, func,
                                    &dispatch, NULL, 0, NULL),
    "zeCommandListAppendLaunchKernel");
return iree_ok_status();
```

SHARK-Runtime/experimental/level_zero/direct_command_buffer.c

HAL Executables

- Executables provide one or more entry points that have well-defined signatures:
- Loaded via an optionally-stateful device-defined cache handle
  - Enables asynchronous/batched preparation (JITing/translation)
  - Enables persistent caching (VkPipelineCache) to reduce cold-start time
- Explicit I/O layout enables validation during calls
  - Only buffers specified in the layout may be accessed during execution
  - Compiler determines the layout and can trade off performance/size/tracking overhead
- Implementation can multi-version/specialize (CPU uarch, GPU shared memory size/extension availability)

Original slides by Ben Vanik

HAL Executables

- Purpose: Storing information from schema into C++ friendly form.
- Input: flatbuffer specified by schema
- Output:

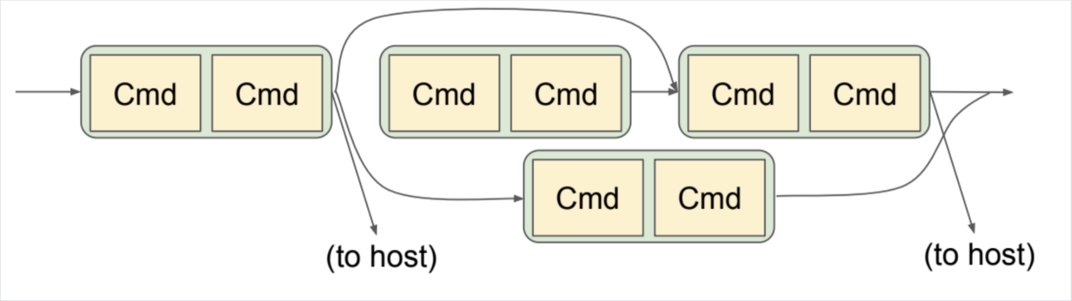
```
typedef struct iree_hal_level_zero_native_executable_function_t {
  ze_kernel_handle_t level_zero_function;
  uint32_t block_size_x;
  uint32_t block_size_y;
  uint32_t block_size_z;
} iree_hal_level_zero_native_executable_function_t;

typedef struct iree_hal_level_zero_native_executable_t {
  iree_hal_resource_t resource;
  iree_hal_level_zero_context_wrapper_t* context;
  iree_hal_pipeline_layout_t** pipeline_layouts;
  iree_host_size_t entry_count;
  ze_module_handle_t module;
  iree_hal_level_zero_native_executable_function_t entry_functions[];
} iree_hal_level_zero_native_executable_t;
```

SHARK-Runtime/experimental/level_zero/native_executables.c

nod

HAL: Semaphore / Synchronization

- Each semaphore is a monotonically increasing timeline, submissions can specify:
- [Wait for a semaphore to reach >=t]
- [Signal a semaphore to t=t']
- Defines a DAG. Both host and device can signal and wait. No round-trips!
- Lazy updating: fine for latency between signal and availability.




```
%fence = hal.fence.create at<%_timeline_semaphore : !hal.semaphore>(%1) -> !hal.fence
hal.device.queue.execute<%device : !hal.device> affinity(%c-1_i64) wait(%0) signal(%fence) commands([%cmd])
util.global.store %1, @_timeline_value : i64
%status = hal.fence.await until([%fence]) timeout_millis(%c-1_i32) : i32
util.status.check_ok %status, "failed to wait on timepoint"
%view = hal.buffer_view.create buffer(%buffer_1 : !hal.buffer)[%c0, %c32] shape([%c2, %c4]) type(%c553648160_i32)
return %view : !hal.buffer_view
```