(more info in speaker notes)

IREE Runtime Design

https://github.com/google/iree

Work by the IREE team and collaborators.

Presented by benvanik@google.com / Summer 2021.

IREE (IR Execution Environment) 👻

IREE is a deployment toolkit for ML programs.

ML is just code with certain characteristics: it has **a lot of dense math** operating on **a lot of data**. There’s no magic: **an ML model is just a program** and we’ve collectively been deploying programs for a long time.

IREE **compiles** ML programs into various **deployable artifacts** and has a small **runtime library** to support executing them.

The artifacts are designed to provide flexible deployment and optimal execution. **The compiler design supports this and is driven by the runtime needs.**

IREE Workflow

Compiler

- Accepts upstream MLIR dialects using tensors like std, math, linalg, and TOSA dialects
- Importer for TensorFlow SavedModels, TFLite Flatbuffers, XLA protos
- Partitions programs into host- and device-side logic and schedules execution
- Bakes out artifacts (IREE VM FlatBuffers, EmitC .c files, etc)

Artifacts

- Shared library-like modules
- Link against IREE built-in modules like the HAL or other user modules
- Contains the partitioned host program and embedded device programs (“executables”) in various formats, like a fat binary
- Logic for device selection, branching paths for different device types, etc - *it’s just code*

Runtime

- Dynamic linker for loading modules into isolated contexts
- Optional bytecode VM for running host programs
- HAL (Hardware Abstraction Layer) API and backends (CPU, Vulkan, CUDA, etc)
- Utility APIs for ease of use (session-style invocation)
- Low-level; designed to be wrapped in bindings for languages/environments

IREE Runtime Library

Small C11 library (~50-150KB), statically linked into applications.

Effectively just a plugin/extension system: loading, linkage, FFI, versioning.

Loads programs produced by IREE’s compiler and provides APIs for calling them.

Middleware-like design: intended to layer-into/compose-with hosting applications.

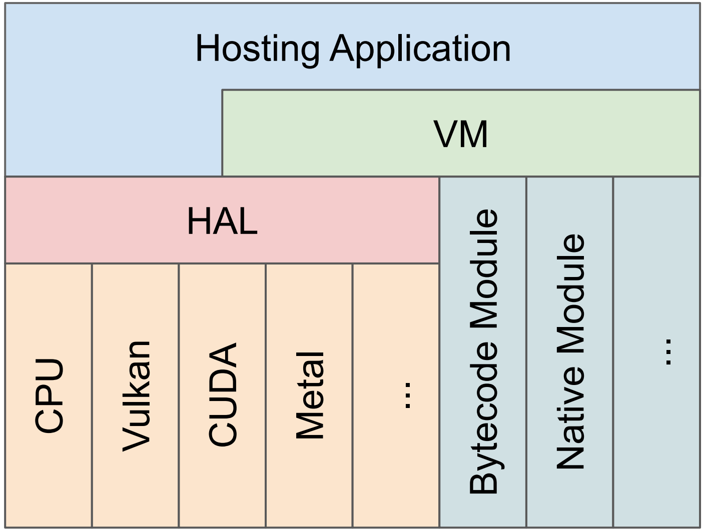
Optional built-in modules available for the loaded programs to use:

- Device management, hardware abstraction layer (HAL), interop
- String and data structure manipulation
- Application-provided custom modules (useful for I/O)

Simple Layering

Hosting applications use the VM API to load, dynamically link, and call programs. Optionally they can use the HAL API to manage devices, buffers, and perform their own execution (data conversion, DMA transfer, etc).

Only pay for what you use: only need to link in the VM modules and HAL backends desired for the target configuration.



Minimal dependencies:

- Core only requires a CRT (memcpy, etc)
- Bytecode VM requires flatcc (~2KB)
- HAL backends each have their own deps

IREE’s Scope at Runtime

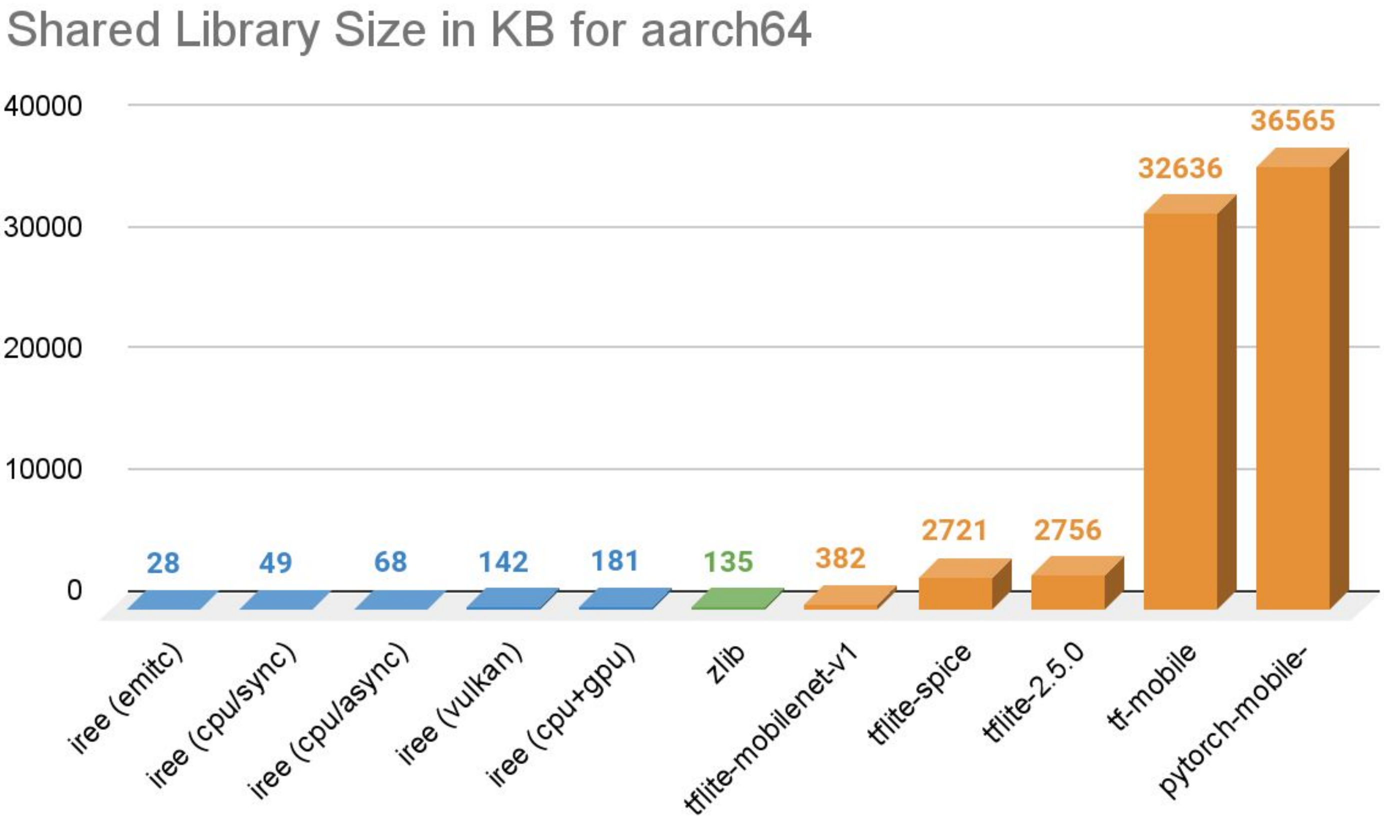
- ✅ Dynamic linking and program isolation
- ✅ Cross-platform/environment FFI
- ✅ Object lifetime and type system
- ✅ Sandboxed execution of host code
- ✅ Binary and environment versioning
- ✅ CPU/GPU/NPU management
- ✅ Last-mile JIT (WASM, SPIR-V)
- ✅ Efficient multithreaded scheduling
- ✅ Buffer allocation and access safety
- ✅ Execution remoting/enclaves
- ✅ Multi-tenant execution scheduling
- ✅ Invocation pipelining/overlap

🚫 Runtime model compilation
Just like C/Swift/Java/etc: you ship a compiled program, not program source + a compiler.

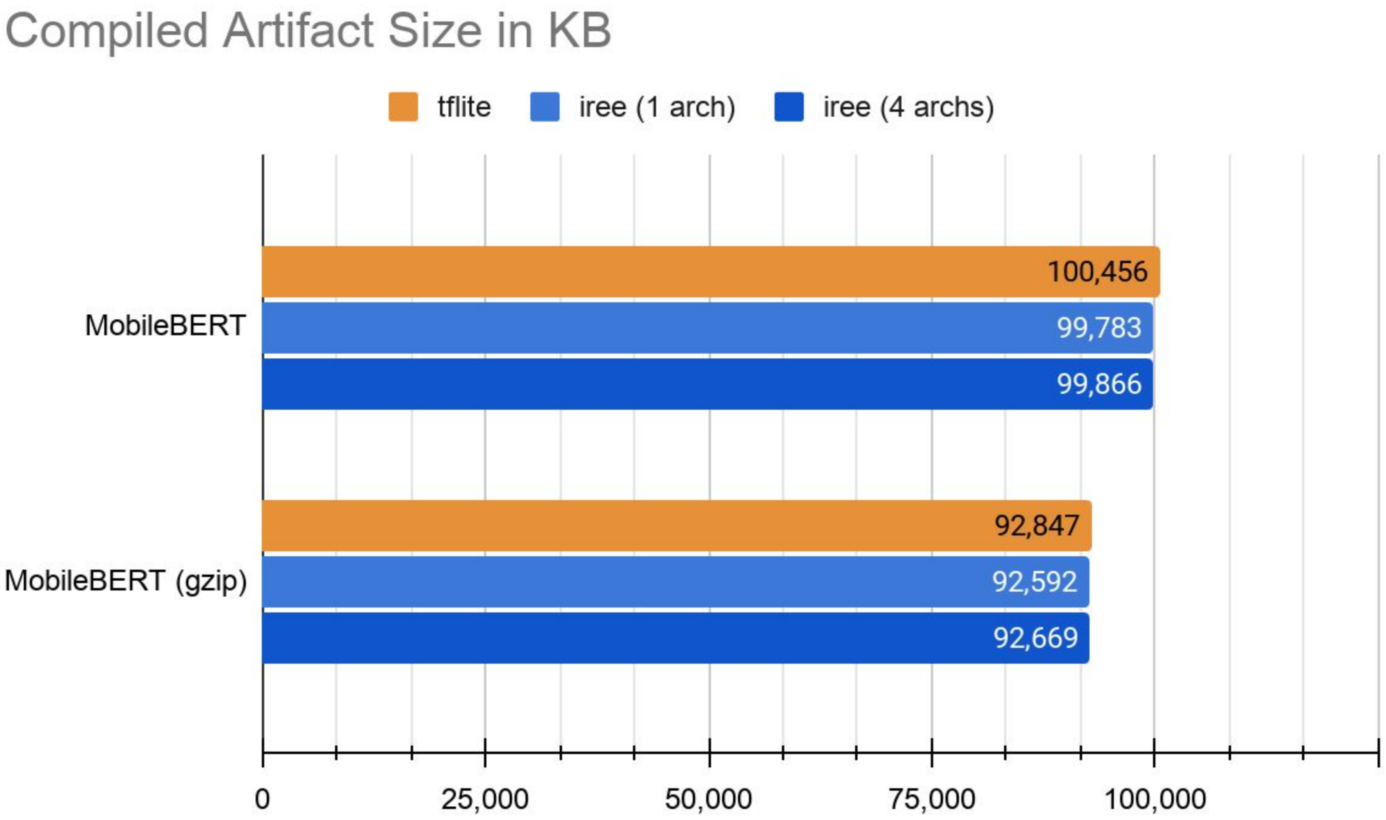
🚫 Multi-node work distribution
IREE is designed to be used as a low-level component within a higher-level work distribution system.

🚫 Monolithic legacy ML framework hosting
Explicit layering: IREE can’t wrap XLA/etc just as libjpeg can’t wrap ffmpeg.

🚫 Tiny static models for tiny embedded CPUs (*yet*)
At that scale (<16KB ROM) you don’t want any kind of framework/library regardless of size. A target in the future with further downscaling (emitc/llvm codegen of host and eliding the HAL).



Full-featured ML with the application size impact of zlib



Similar artifact sizes to classic solutions even when multi-targeting

Note: You Are Likely Already Multi-targeting 😱

```
demo.apk/lib/arm64-v8a/libtensorflowlite_jni.so      2,147,664 bytes
demo.apk/lib/armeabi-v7a/libtensorflowlite_jni.so    1,064,732 bytes
demo.apk/lib/x86/libtensorflowlite_jni.so            1,591,164 bytes
demo.apk/lib/x86_64/libtensorflowlite_jni.so         1,797,104 bytes


demo.apk/res/drawable-xxxhdpi/awesome-image.png
demo.apk/res/drawable-xxhdpi/awesome-image.png
demo.apk/res/drawable-xhdpi/awesome-image.png
demo.apk/res/drawable-hdpi/awesome-image.png
demo.apk/res/drawable-mdpi/awesome-image.png
```

Design

Key Tenet: N->1 is easy, 1->N is hard

The IREE design is **client/server**, **asynchronous**, and **modular** with careful consideration to use-case-driven **optionality** (synchronous, single-node, unity build, multi-tenant, enclave, etc). In our experience:

- Running properly-factored client/server architectures on a single node is easy
  - Example: the default IREE host-local CPU executor
- Using properly-factored asynchronous APIs as synchronous APIs is easy
  - Example: the IREE synchronous CPU device which uses the same HAL API as async
- Trimming properly-factored optional features is easy
  - Example: pay-for-what-you-use HAL backends, dynamic module linkage, and VM instruction extension sets
- Ignoring cheaply-obtained information when not needed is easy
  - Example: tight buffer lifetime tracking/management emitted by the compiler may be ignored at runtime by unified memory devices but is critical to discrete memory configurations

Terminology: “host” vs “device”

The **host schedules work** and **device executes it**.

See Compute Shader 101 for a fantastic overview.

CPU:

- Host: `thread_pool.Enqueue(SomeWorkFn);`
- Device: `void SomeWorkFn() { … }`

CUDA:

- Host: `my_kernel<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);`
- Device: `__global__ void my_kernel(int n, float a, float *x, float *y) { … }`

Vulkan:

- Host: `vkCmdDispatch(command_buffer, workgroup_x, workgroup_y, workgroup_z);`
- Device: (in my_kernel.comp) `void main() { … }`

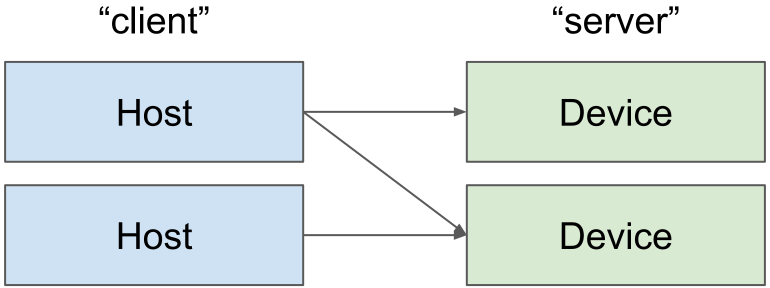
Client/Server-esque Split Design

Host “VM”: a lightweight pluggable virtual machine interface

Device “HAL”: a low-level Vulkan-like hardware abstraction layer

Enables isolation, independent execution frequencies, efficient batching/transactions, versioning, and multi-tenancy (both concurrent models and concurrent users).

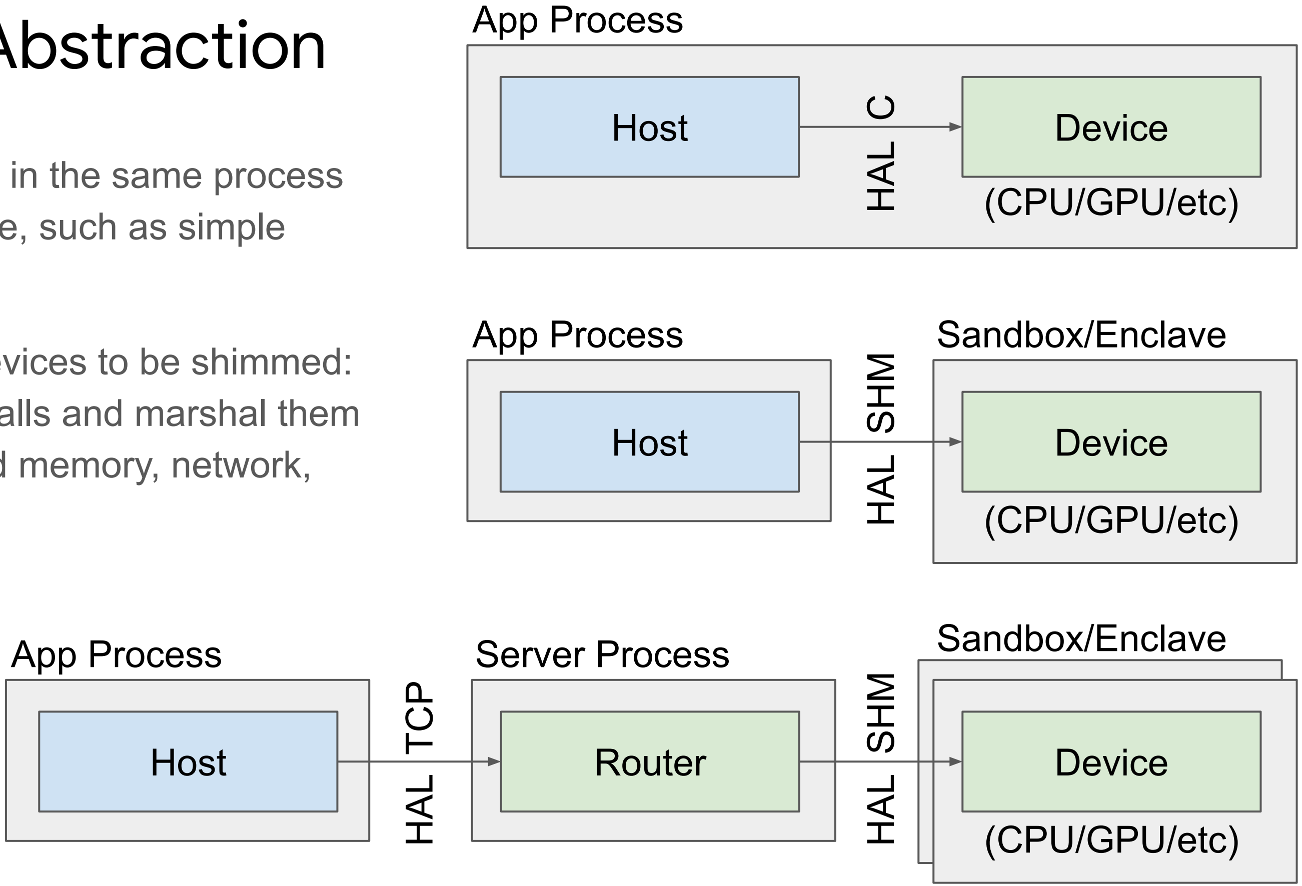
Almost all decent async systems follow this model: browsers, window managers, networking, databases, async IO, device drivers, containers, games, etc.



A Conceptual Abstraction

“Host” and “device” may be in the same process on the same physical device, such as simple local CPU execution.

HAL API design enables devices to be shimmed: take incoming device API calls and marshal them over RPC channels (shared memory, network, enclave APIs, etc).



A Clear Division of Responsibilities

Host

- Isolates host programs
- Enumerates and selects device(s)
- Interacts with hosting application
- Maintains shadow state of server (handles to device resources, etc)
- Builds and issues transactions ("command buffers") against the device
- Tracks device progress and synchronization ("semaphores")
- Validates host execution (proper HAL usage, cross-module FFI, local memory access); treats programs as untrusted

Device

- Owns low-level device handles (CUdevice, VkDevice, threads, etc)
- Loads and retains device-specific executable code (shaders/kernels/etc)
- Marshals requests for memory reservation and execution from hosts
- Maps HAL synchronization primitives to low-level primitives (fences, etc)
- Validates device execution (proper HAL usage, local memory access); treats host as untrusted

Security by Construction

Defense-in-depth: each layer assumes the other layers are untrusted.

Composable layering enables trade-offs based on usage. Less validation required when statically linking first-party generated code into application binaries vs. loading untrusted drive-by content from the web. Existing application-level signing and verification can be sufficient for loading native code when the signer is trusted.

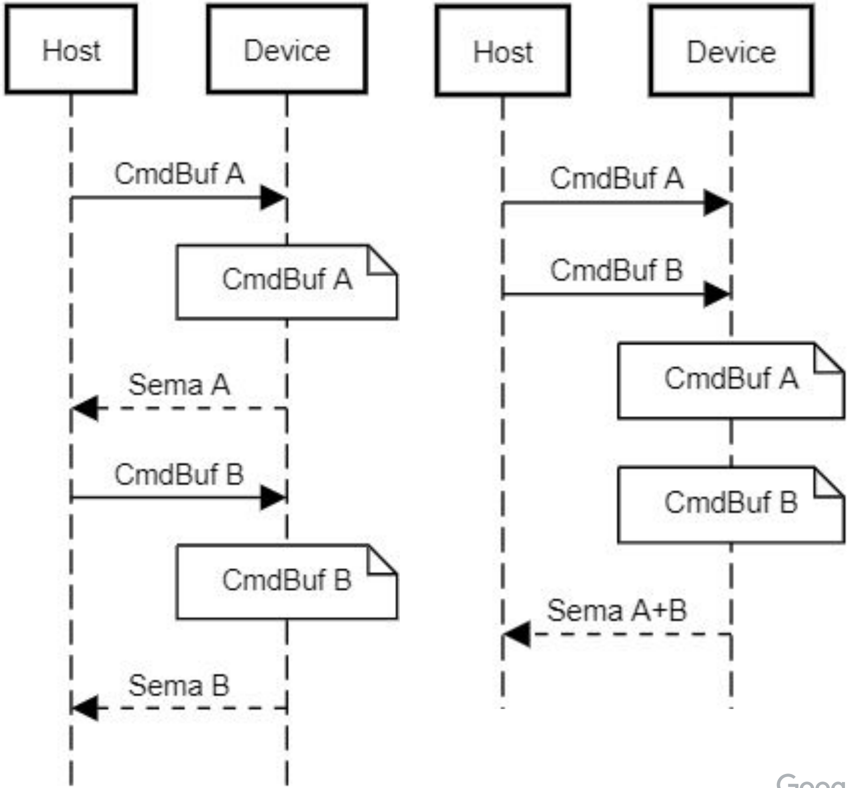
All the same techniques and approaches that are used in normal software engineering. No ML-special behavior.

Host: uses IREE bytecode VM to execute programs (by default). Multiple programs are safe to execute within the application process (assuming no bugs in the VM - can also be sandboxed if desired).

Device: explicit HAL API ensures buffer ranges are valid and accessed correctly. Executable security depends on target: intermediate forms like WebAssembly and SPIR-V can provide memory safety on-demand, while ELFs containing native code assume they execute within a sandbox or are already verified.

Fundamentals of Asynchronous Execution

- **Handles** for resources, not pointers
    - Allows forward referencing of remote objects
    - Avoids implicit memory coherency/access issues - **no void*s**!
- **Transactions** for grouping related operations into batches
    - Avoids round-trips between operations
    - Enables remote execution planning/optimization
- **Sequencing** for chaining transactions
    - Avoids round-trips between transactions
    - Enables remote redistribution/balancing
    - Enables pipelining and batching



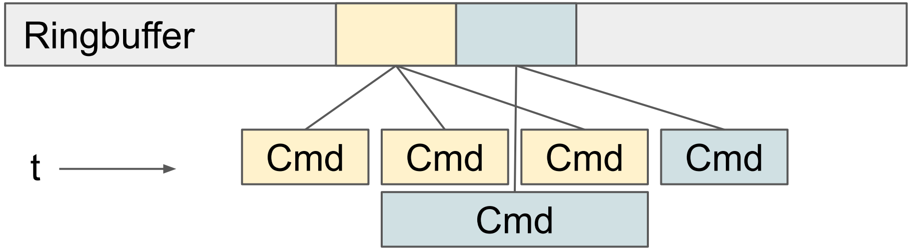
Allocation in Pipelined Systems

Naive: synchronously execute and commit enough for only 1 transaction at a time.

Easy: enable pipelining by committing memory for all in-flight operations.

Optimal: overcommit memory only for operations that may overlap. See FrameGraph.

Ringbuffers/streaming allocation required for maximum concurrency:



Error Handling in Pipelined Systems

Can’t use normal patterns (“if(!ok) return;”) as pipelined errors are:

- Non-deterministic:
  - Overlapping/out-of-order work causes failures to happen out of order too
  - Any work-arounds prevent pipelining; only useful in debug builds
- Unpredictable:
  - Hardware watchdogs or the OS may reset underlying devices
- Expensive:
  - Need to propagate errors back to the host... *somehow*
  - Querying requires a round-trip (see glGetError) - need to pipeline that just like the work
- Not useful for control flow:
  - Need to enqueue work without waiting to see if prior work has finished

Modularity via Orthogonality

API is designed to avoid link-the-world situations; naturally include-what-you-use:

- VM instruction extensions (i64/f32/f64) via configuration
- HAL backends (CUDA, Vulkan, etc) by way of device driver factories
- Vulkan/CUDA/etc dynamically link against drivers; cheap to always include and conditionally use when available
  - No need for a “CUDA build”
  - Cross-compile without SDKs / avoid SDK version-locking
  - Wrap existing application-owned device handles
- CPU asynchronous threaded task system or synchronous in-order executor
  - Can co-exist! Compiler emits code to decide when to use either based on the work being executed and/or the state of the system
- No op/kernel libraries: no byte shipped that is (known) not needed

Flexible Artifact Deployment

- Target and deploy just as you would an Android APK/iOS IPA, web app, etc
  - IREE artifacts are multiarchtecture/fat binaries
  - Bundle targets together and let the runtime choose, or slice and deploy only what's needed
- Multiple device types (GPU, CPU, etc) and execution formats
  - IR (SPIR-V, WebAssembly), native machine code (x64), vendor-specific (PTX)
- Multiple CPU architectures (aarch64, x64, etc)
  - Runtime subarchitecture specialization (arm v8.3 | v8.4, etc)
- Dynamic linker enables polyfills and dependency versioning
  - Extend backward/forward compatibility range based on needs
- Fits into existing build/content pipelines
  - Source model is like a PSD/RAW/PNG, compiled artifact is like a JPEG
  - Keep source around and regenerate as needed
  - Include durable targets (WebAssembly, etc) for decade+ longevity, if you care

Host-side: VM

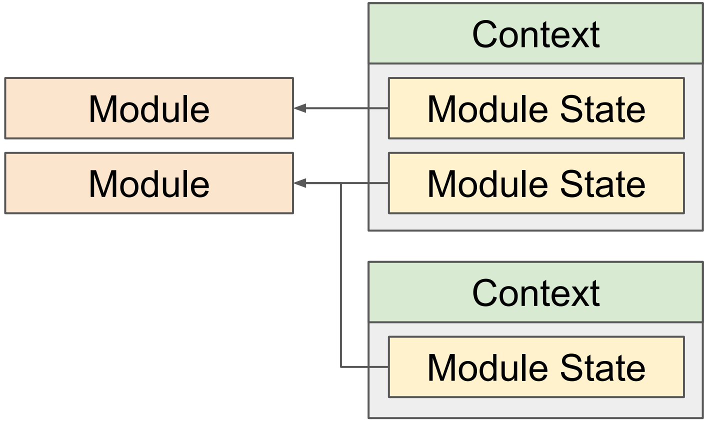
(Virtual Machine)

VM: Lightweight User-mode Processes and an FFI

- Responsible for “host” programs
- Workload is mostly calls into the HAL and some simple buffer offset arithmetic
- Isolates loaded module instances and their state (ala processes)
- Dynamic runtime linking of modules implemented in C or compiler-generated
    - Used for versioning/compatibility shimming
    - Enables host code sharing (similar model architectures/etc)
- Reference-counted type-safe handles: no void*, no need for an MMU
- Bytecode module format runs anywhere
    - Minimal implementation is i32-only, ~10KB
    - Extensions for i64, f32, and f64
- Coroutines: cellular batching, wider parallelism, reduced thrashing

VM module:context :: shared object:process

- **Contexts** perform dynamic linking and instantiate **module state**
- **Modules** import and export **functions**
  - Virtual interface with a defined FFI, part of the public API
  - Single unified interface for built-in, compiler-generated, and user-defined modules
  - Reflection queries for plumbing higher-level calling convention info
  - Type system and import signature verification
  - Supports coroutine-style continuation
- **Abstract call stack**
  - Cross-environment backtraces
  - Reference-counted object lifetime tracking
  - Storage for coroutine state



VM: Types

Host code is untrusted: no void*, need to check types and prevent use-after-free:

`!vm.ref/iree_vm_ref_t`: intrusive atomic ref count with type ID

Retain/release-style (ala CoreFoundation/ObjC/etc).

Explicit type registration; custom types for custom modules allowed. Represented in the compiler as MLIR types and targetable via standard compiler workflows.

Built-in types (so far):

- `!vm.buffer/iree_vm_buffer_t`: fixed-size byte buffer
- `!vm.list/iree_vm_list_t`: std::vector<T>, supporting variants/ref types

VM: Modules

Modules are like an ELF shared library/DLL. The IREE runtime provides a dynamic linker.

VM modules are just a small C interface: a module can be defined as a FlatBuffer with bytecode, native C/C++ (handwritten or generated with emitc), or anything you can reach from the C ABI (python, javascript, etc).

Built-in IREE functionality is also done via modules, like the HAL, allowing multi-versioning and compatibility shims in the same way one would do it for a normal application.

The most general and well-supported scenario has the compiler produce FlatBuffers containing the IREE VM bytecode (.vmfb). Provides a dynamically deployable mmap-able binary with near zero load-time overhead.

VM: Low-level Native Modules

Low-level C interface; good for small code and interop (exporting to python/etc):

```
// vm.import @semaphore.create(
//   %device : !vm.ref<!hal.device>,
//   %initial_value : i32
// ) -> !vm.ref<!hal.semaphore>

IREE_VM_ABI_EXPORT(iree_hal_module_semaphore_create,
                   iree_hal_module_state_t,
                   ri, r) {
  iree_hal_device_t* device = NULL;
  IREE_RETURN_IF_ERROR(iree_hal_device_check_deref(args->r0, &device));
  uint32_t initial_value = (uint32_t)args->i1;
  ..
  rets->r0 = iree_hal_semaphore_move_ref(semaphore);
  return iree_ok_status();
}
```

VM: High-level Native Modules

High-level C++ interface; much easier to use with larger binary size:

```
// vm.import @semaphore.create(
//   %device : !vm.ref<!hal.device>,
//   %initial_value : i32
// ) -> !vm.ref<!hal.semaphore>

StatusOr<vm::ref<iree_hal_semaphore_t>> SemaphoreCreate(
    vm::ref<iree_hal_device_t> device, int32_t initial_value) {
  ...
  return semaphore;
}
```

VM: Bytecode Modules

The most general and well-supported scenario uses FlatBuffers to provide a mmap-able binary with near zero load-time overhead containing the IREE VM bytecode (.vmfb).

Just one implementation of the module interface; WebAssembly, Lua, Python, etc can be implemented with equal fidelity.

Goals:

- Portability (runs anywhere C can run)
- Manageable implementation (~15KB)
- Near 1:1 mapping with MLIR dialect
- Rich type and FFI verification
- Safe isolation even without OS and hardware support

BytecodeModuleDef
Header for runtime versioning
Import functions from other modules
Export functions to other modules
Reflection metadata for FFI
Read-only data (.rodata) for constants and compiled HAL device executables
Initialized **read-write data** (.data) for variables
VM bytecode for the host program (.text)

VM: Bytecode ISA and Execution

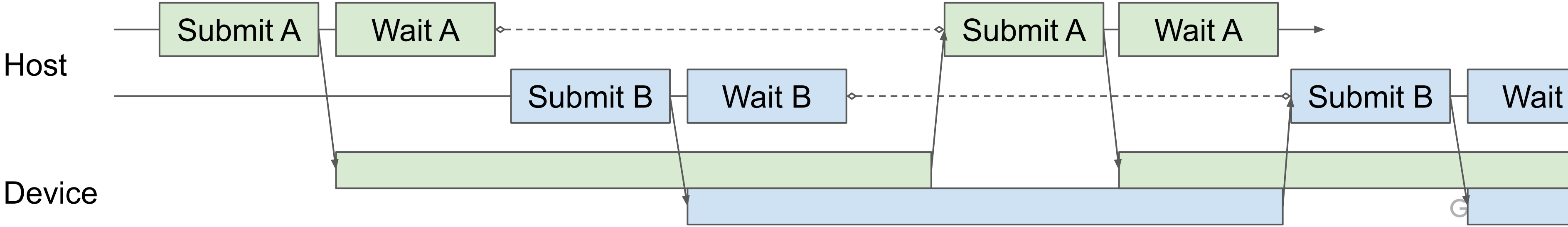
- Virtual register-based instructions
- Direct execution (no load-time processing)
- Efficiently dispatchable with computed gotos
- Support for external ref counted types
  - VM tracks references to avoid use-after-free
  - Opaque type round-tripping
  - Built-in types for lists and byte buffers
- Memory access only though checked buffers
- Instruction set extension mechanism
  - Optional 64-bit integer type
  - Optional 32/64-bit floating-point types
- Stack frames and stack walking support
- MLIR is easily lowered to C, LLVM-IR, etc

```
vm.export @loop_sum
vm.func @loop_sum(%count : i32) -> i32 {
  %c1 = vm.const.i32 1 : i32
  %i0 = vm.const.i32.zero : i32
  vm.br ^loop(%i0 : i32)
^loop(%i : i32):
  %in = vm.add.i32 %i, %c1 : i32
  %cmp = vm.cmp.lt.i32.s %in, %count : i32
  vm.cond_br %cmp, ^loop(%in : i32),
                   ^loop_exit(%in : i32)
^loop_exit(%ie : i32):
  vm.return %ie : i32
}
```

VM: Coroutines

ABI has built-in coroutine support; works across languages/runtimes.

- Enables multiple programs - or invocations of the same - to overlap
- VM execution yields on wait handles; multi-wait for epoll-style waiting
- Compiler determines what work may overlap based on data flow
- Enables dispatch coalescing in the HAL



Device-side: HAL

(Hardware Abstraction Layer)

HAL: A Compute-only Subset of Vulkan + Some Tweaks

Modern GPU APIs like Vulkan, Metal, Direct3D 12, and (lower-level) CUDA all effectively do the same thing with different spelling:

- **Enumerate and query devices** and their capabilities
- **Define executable code** that runs on the device
- **Allocate** unified or discrete memory and provide cache control
- **Organize work into sequences** for deferred submission
- Provide **explicit synchronization primitives** for ordering submissions

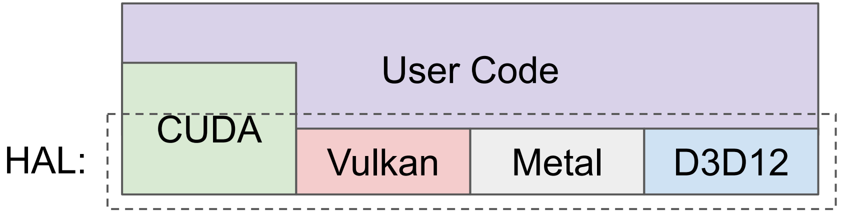
The IREE HAL wraps these up in Vulkan-themed functions (as it’s the only open and cross-platform/vendor solution) - but it’s just style.

HAL: A Low-Level Abstraction

Predictable, portable, efficient, and small.

- No implicit state tracking: users own only the state they need
- No implicit cross-device memory transfer: DMA is a first-class concept
- No unmanaged void* access: explicit buffer and binding APIs
- No implicit synchronization: barriers are opt-in

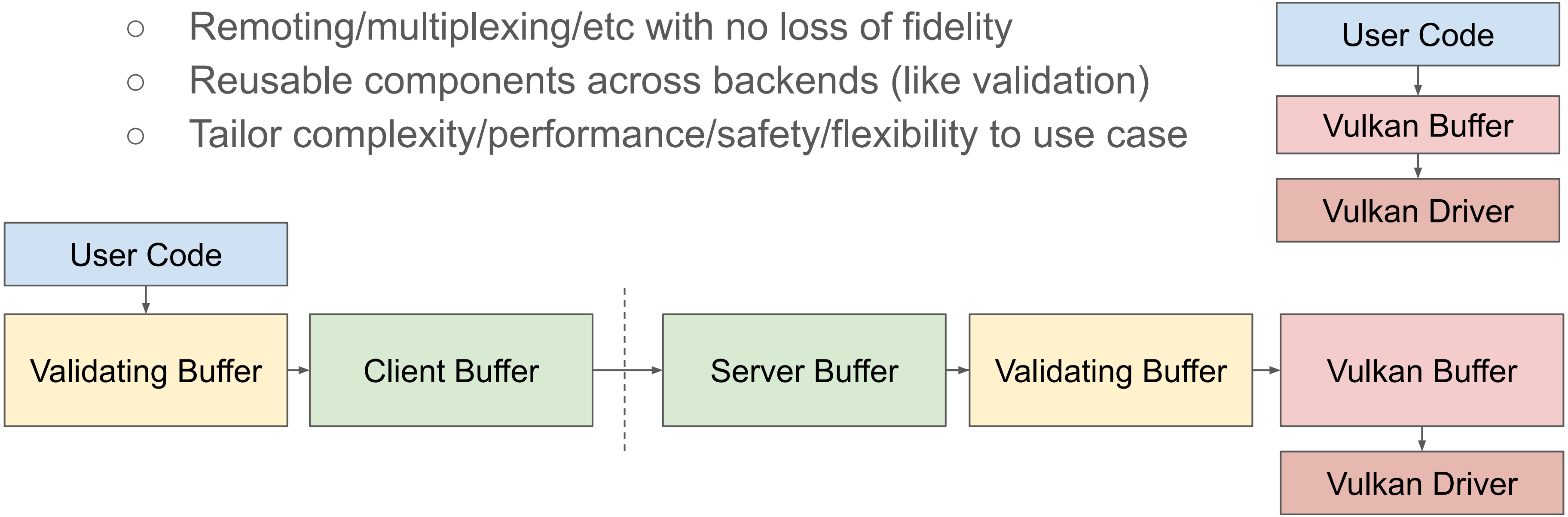
For larger full-stack solutions like CUDA only the lower-level primitives are used.



HAL: Virtual Interface

The HAL API is (nearly) a header-only interface:

- All resources (executables, buffers, semaphores, etc) are just handles
- Composition instead of inheritance:
  - Remoting/multiplexing/etc with no loss of fidelity
  - Reusable components across backends (like validation)
  - Tailor complexity/performance/safety/flexibility to use case



HAL: Drivers and Devices

- 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)

HAL: Allocators

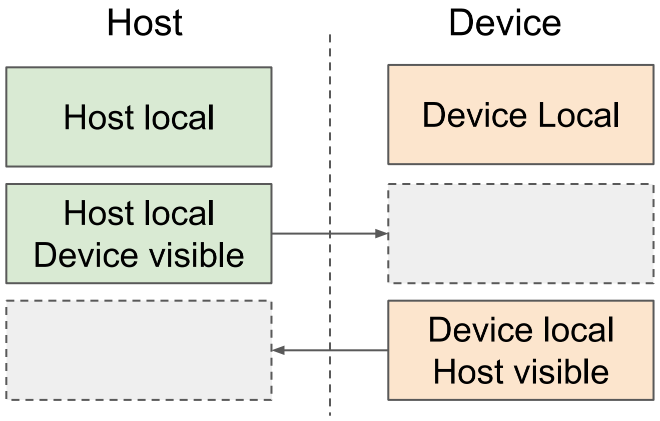
All buffer handles obtained via a device-provided allocator handle:

- 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.

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

HAL: Command Buffers

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.

Commands:

- Synchronization: `ExecutionBarrier`, `SetEvent/WaitEvent`
- DMA: `CopyBuffer`, `FillBuffer`, `UpdateBuffer`
- Execution: `Dispatch`, `DispatchIndirect`, `Execute` (command buffer)
- Recording-only state: `PushConstants`, `BindDescriptorSet`

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)

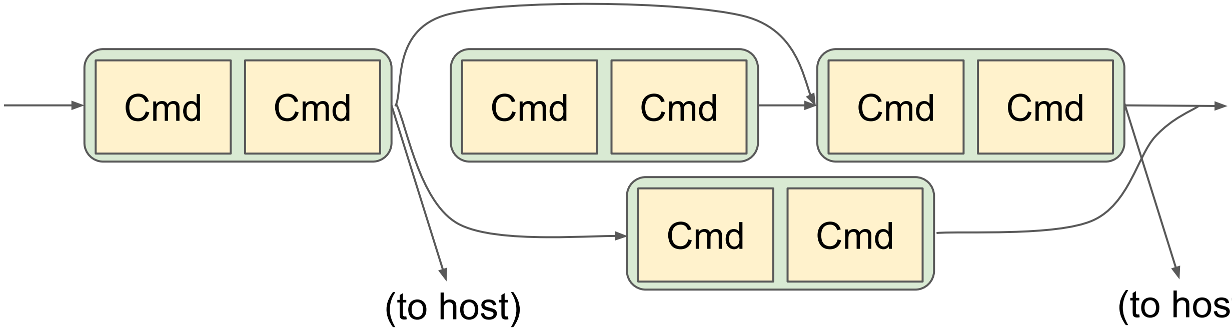
HAL: Semaphores (aka Fences)

Each semaphore is a monotonically increasing timeline, submissions can specify:

- [Wait for a semaphore to reach >=t]
- [Signal a semaphore to t=t’]

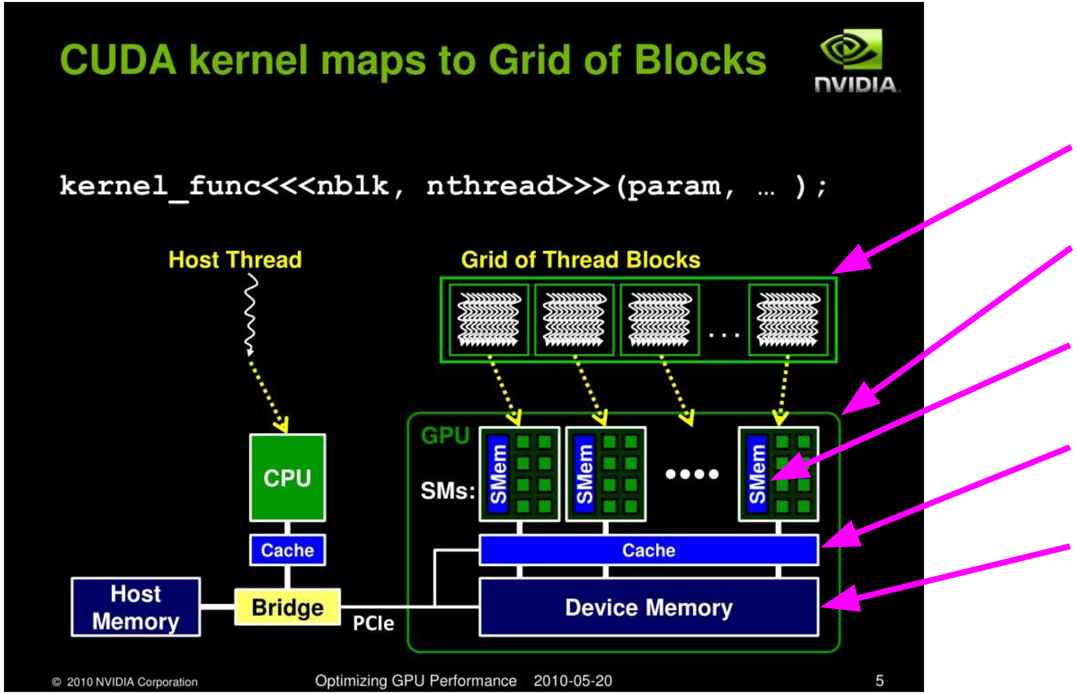
Defines a DAG. Both host and device can signal and wait. No round-trips!

Lazy updating: fine for latency between signal and availability.



CPUs as HAL Devices

CPU: Modeled as a GPU



CUDA kernel maps to Grid of Blocks

```
kernel_func<<<nblk, nthread>>>(param, … );
```

© 2010 NVIDIA Corporation Optimizing GPU Performance 2010-05-20 5

Tiled tasks to execute

Package with multiple cores

L1 cache (local to core)

L2+ cache (shared among cores)

DRAM

Example hand-authored executable showing what the compiler generates:

```
static int dispatch_tile_a(
    const iree_hal_executable_dispatch_state_v0_t* dispatch_state,
    const iree_hal_vec3_t* workgroup_id) {
  int workgroup_size_x = dispatch_state->workgroup_size.x;
  const float* src = ((const float*)dispatch_state->binding_ptrs[0]);
  float* dst = ((float*)dispatch_state->binding_ptrs[1]);
  for (int i = 0; i < workgroup_size_x; ++i) {
    float v = src[workgroup_id->x * workgroup_size_x + i];
    v = (v * 2.0f + 1.0f) / log(v);
    dst[workgroup_id->x * workgroup_size_x + i] = v;
  }
  return 0;
}
```

Simplified example of how the executables are invoked:

```
for (uint32_t z = 0; z < dispatch_state.workgroup_count.z; ++z) {
  for (uint32_t y = 0; y < dispatch_state.workgroup_count.y; ++y) {
    for (uint32_t x = 0; x < dispatch_state.workgroup_count.x; ++x) {
      iree_hal_vec3_t workgroup_id = {{x, y, z}};
      int ret = entry_fn_ptr(&dispatch_state, &workgroup_id);
    }
  }
}
```

That’s the whole story: a fixed function signature and 3 loops around a call :)

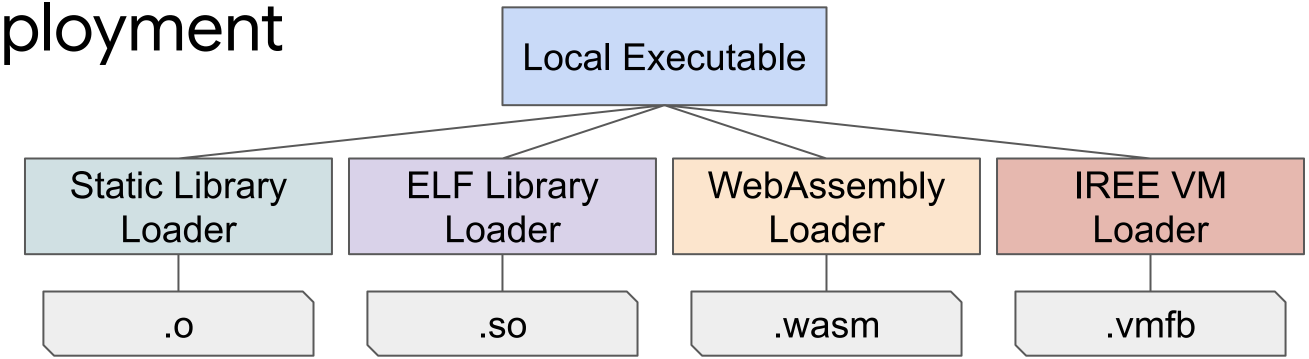
CPU: Constrained Execution of Compute Workloads

HAL Executables: just some functions (aka kernels aka shaders)

- Executables export functions and may have internal functions
- Exports can be chosen at load time (subarchitecture switching)
- Functions are stateless and only use the buffers provided for I/O
- Functions are called via a fixed ABI (see executable_library.h)
- Functions may only call imports via a provided import table
- No syscalls allowed
- ABI is versioned for backward/forward compatibility

Enables safe lockless multi-threaded execution, platform/runtime version independence, bounds checking (both internal and external), and isolation.

CPU: Flexible Deployment



Static deployment:

- Static libraries linked into the hosting application

Dynamic deployment:

- Multi-arch shared libraries
- WebAssembly (or other JIT’ed runtimes)
- IREE VM (VMVX)

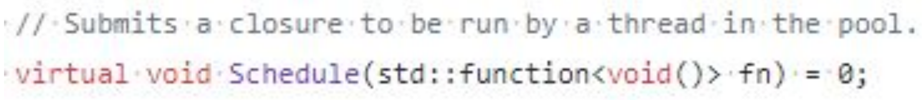
Bring-your-own: pluggable system

CPU: Platform-agnostic Shared Libraries

- IREE compiler targets the Linux ABI
  - Single LLVM target triple per architecture - x64 variant will run on any x64 OS
  - Common calling convention across all platforms except Windows; on Windows we adapt
  - Imports are explicitly declared and can be shimmed (for versioning/ABI compatibility)
  - Exports all have the same signature and are only called by the IREE HAL code
  - No syscalls allowed so no need to worry about compatibility
- ELF used for storage
  - Can dlopen on Linux/Android, readelf/etc to dump, can contain DWARF symbols, etc
  - Stripped ELF is produced via LLD and is just ~1.5KB overhead
  - Embedded ELF loader for cross-platform loading (Windows/bare-metal/etc)
    - Loader code size is 3KB and performs only one allocation per load
    - There is no more efficient way to load machine code :)
- Fallback path for platform-native libraries (DLL/MachO/etc)

CPU: Multi-threading

This is **not** a state-of-the-art way to schedule work:



```
// Submits a closure to be run by a thread in the pool.
virtual void Schedule(std::function<void()> fn) = 0;
```

Though cheaper to go core->core than CPU->GPU, **avoiding round-trips and synchronization is just as critical to performance on CPUs**.

Even small amounts of synchronization can be 10000x+ slower.

Many layers of caches; exploiting cache locality is critical to performance.

Only benefit of the naive scheduling above is that it's easier for a *human* to author. IREE is a *machine* 🤖

CPU: Multi-threading

Systems like taskflow and TBB demonstrate how to do this better:

- DAGs of tasks enabling out-of-order execution/pipelining
- Tasks are subdivided for locality-aware execution across compute units
- No pipeline-aware transient allocation :(
- High-level and mostly focused around hand-authored programs :(

Looks exactly like a GPU compute API (CUDA, Vulkan, etc):

- Command buffers and semaphores for DAGs
- Grid dispatch for locality-aware distribution
- Explicit memory management to enable pipelined allocation
- Perfect for targeting by machine: low overhead, predictable performance

CPU: IREE's GPU-like Task Scheduler

See iree/task/executor.h for details and paper links (this is a good overview).

- Designed for executing compute workloads coming from command buffers
  - 3D grid topology
  - DAG construction is forward-only single-pass with barriers/fences
- Stall/interlock-aware scheduling data structures
  - Minimizes cross-core locking and cache invalidation when posting/consuming work
  - 8-16KB+~1KB/worker overhead, 0 allocations in steady-state scheduling/execution
- Cache-aware work stealing
  - Minimizes latency in the event of worker preemption or non-uniform tile work
  - Source workers are chosen based on topology: steal from adjacent workers first
- Look-ahead for descheduling/wake latency reduction
  - Knows what work is pending, only spins if needed
  - Futex-based signaling for low-overhead wakes, multi-wait for external handles

CPU: Task System Tracing

Full Tracy instrumentation.

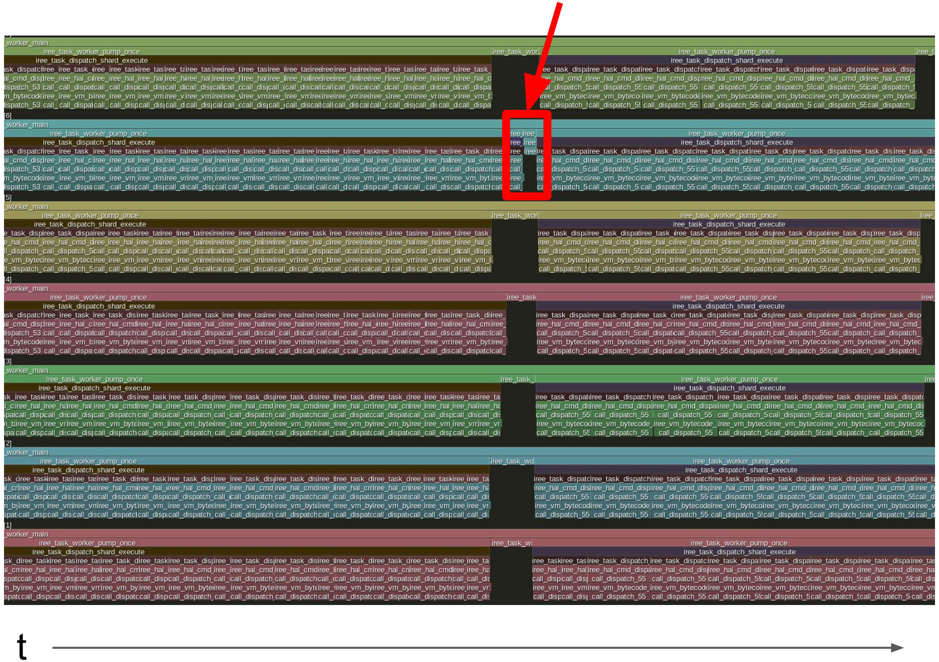
Like Nsight for CUDA: see the shape of work as it executes.

Cross-platform and easy to setup.

A good proxy for GPUs/accelerators: sequential work and over/under distribution are easy to spot and fix.

You can't fix what you can't see.

Sequential work w/ data dependency 😞



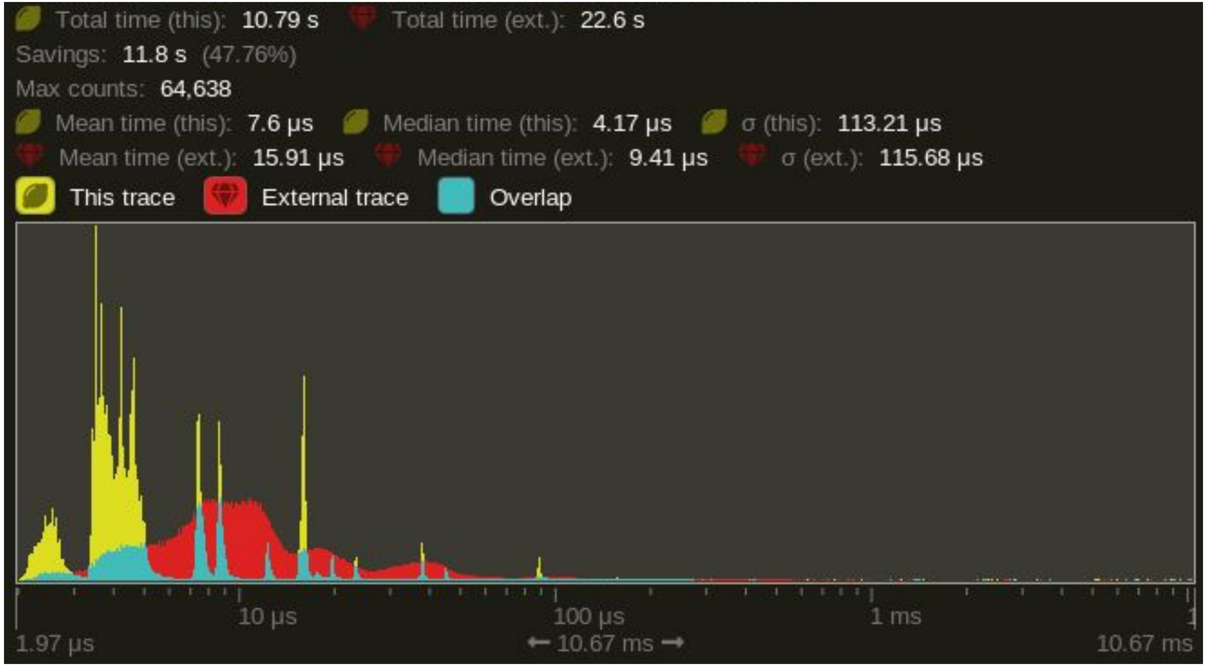
CPU: Example of Work Distribution Cache Effects

Same system, same work, just *ever so slightly* less naive scheduling: **2x faster**

Red: greedy tile execution

Yellow: 32 tile slices/core

Locality matters :)



Total time (this): 10.79 s   Total time (ext.): 22.6 s

Savings: 11.8 s (47.76%)

Max counts: 64,638

Mean time (this): 7.6 μs   Median time (this): 4.17 μs   σ (this): 113.21 μs

Mean time (ext.): 15.91 μs   Median time (ext.): 9.41 μs   σ (ext.): 115.68 μs

End-to-End Example

(coming soon)