HAL Driver Features

Heterogeneity is one of IREE's core design principles. IREE aims to support various accelerators for compute, ranging from general purpose CPUs, GPUs, to other special purpose accelerators. IREE provides a Hardware Abstraction Layer (HAL) as a common interface to these accelerators. IREE exposes it via an C API for programmers and an MLIR dialect for compilers.

Heterogeneity inevitably means IREE needs to provide a solution for managing different features on different accelerators and their availability. This doc describes the designs and mechanisms.

General HAL Driver Features

IREE uses compilers to generate native code for each accelerator, serialize the native code, and embed the code in one flat byte code following FlatBuffer encoding format. The native code embedded in the final FlatBuffer file will indicate the target architecture and required feature sets. At runtime IREE selects a HAL driver meeting all the requirements to dispatch the workload to.

[TODO: describe the HAL functionality, C API, and dialect abstraction]

Vulkan HAL Driver Features

Vulkan has many mechanisms for supporting different hardware implementations: versions, extensions, features, limits. Vulkan uses SPIR-V to express the GPU program but Vulkan is just one client SPIR-V supports. So SPIR-V has its own mechanisms for supporting different clients: versions, capabilities, extensions. The mechanism in these two domains bear lots of similarity, but they are not exactly the same. We need to bridge these two worlds inside IREE.

IREE has its own Vulkan dialect, which defines the Vulkan target environment, including versions, extensions, features. These definitions leverage MLIR attribute for storage, parsing/printing, and validation. For example, we can have the following Vulkan target environment:

```
target_env = #vk.target_env<
  v1.1, r(120),
  [VK_KHR_spirv_1_4, VK_KHR_storage_buffer_storage_class],
  {
    maxComputeSharedMemorySize = 16384 : i32,
    maxComputeWorkGroupInvocations = 1024: i32,
    maxComputeWorkGroupSize = dense<[128, 8, 4]>: vector<3xi32>,
    subgroupFeatures = 7: i32,
    subgroupSize = 64 : i32
  }
>
```

The above describes a Vulkan implementation that supports specification version 1.1.120, supports `VK_KHR_spirv_1_4` and `VK_KHR_storage_buffer_storage_classs` extensions, has a max compute workgroup invocation of 1024, and so on.

The above bears lots of similarity with the output of the `vulkaninfo` utility. That's intended: `vulkaninfo` gives a detailed dump of a Vulkan implementation by following the structures of all the registered extensions to the specification. We pick relevant fields from it to compose the list in the above to drive code generation. These are just different formats for expressing the Vulkan implementation; one can image having a tool to directly dump the MLIR attribute form used by IREE from the `vulkaninfo`'s JSON dump.

When compiling ML models towards Vulkan, one specifies the target environment as a `#vk.target_env` attribute assembly via the `iree-vulkan-target-env` command-line option. At the moment only one target environment is supported; in the future this is expected to support multiple ones so that one can compile towards different Vulkan implementations at once and embed all of them in the final FlatBuffer and select at runtime. Another command-line option, `iree-vulkan-target-triple` is also available to allow specifying common triples and avoiding the lengthy target environment assembly string. `iree-vulkan-target-triple` will be overridden by `iree-vulkan-target-env` if both are given.

Under the hood, this Vulkan target environment is then converted to the SPIR-V target environment counterpart to drive code generation. The conversion happens in one of Vulkan dialect's utility function. The converted SPIR-V target environment is attached to the dispatch region's module for SPIR-V passes to use.

SPIR-V's target environment is very similar to the Vulkan target environment in the above; it lives in upstream MLIR repo and is documented here and implemented in SPIR-V dialect's `SPIRVAttribues.h` and `TargetAndABI.td`.

PR #3469, along with patch D89364, shows an example of the changes needed to add support for the VK_NV_cooperative_matrix extension for Vulkan/SPIR-V. The overall steps are as follows:

1. Add the enum corresponding to the extension to `VK_ExtensionAttr` in VulkanBase.td.
2. Add necessary capability bits to `VK_CapabilitiesAttr`.
3. Add a corresponding attribute to the `SPV_ResourceLimitsAttr` in TargetAndABI.td. (Note: The corresponding SPIR-V extension is likely already defined in `SPV_ExtensionAttr`)
4. Convert the capability bits specified in the attribute added to `VK_CapabilitiesAttr` to the attribute added to `SPV_ResourceLimitsAttr`.