IREE CUDA backend

This document is intended to provide an overview of the design choices made to support CUDA within IREE. It describes both the HAL runtime and the NVVM codegen side.

CUDA HAL Driver

The CUDA HAL driver is in `iree/hal/drivers/cuda/` directory. It is written in C following the standards of the rest of the HAL module.

CUDA library dependency

IREE calls directly into `CUDA driver API`. CUDA library is loaded dynamically and cuda.h header from CUDA SDK is part of IREE third_party project. Therefore IREE doesn't require CUDA SDK to be installed when building iree tools. At runtime HAL CUDA driver will load libcuda.so/nvcuda.dll library and load a subset of the cuda driver API used in HAL. The list of functions being used are in the file `iree/hal/drivers/cuda/dynamic_symbols_tables.h`

Driver

There is no direct equivalent in CUDA to the HAL driver abstraction. We use it to hold the symbols loaded for all the devices.

Device

The equivalent to HAL device in CUDA is the `CUcontext`, it holds all the state related to memory allocations.

Command buffer

We implement command buffers using `CUDA Graph API`. Using the Graph API allows to easily encode fine grain dependencies between dispatch without having to create multiple streams. Note that Graph API is meant to be used for command buffers that can be recorded once and used several times and there may be a performance penalty to using Graph API for direct command buffer. It is likely that we will also have a pure stream implementation in the future if we see performance problems with direct command buffer usages.

Event and Barrier

In HAL Event and Barrier are used for GPU<->GPU synchronization either within a command buffer (Event and Barrier) or between command buffers.

The current implementation ignores events and barriers and serializes all the nodes of the graph in order to have a conservative but correct solution.

The design we plan for the future is to map dependencies within a command buffer to graph dependencies in the CUDA Graph API. When an event is signaled all the leaf nodes of the graph will be saved in HAL data

structure and when the same command buffer waits on the signal we will add all the nodes as dependency to the future nodes added to the graph.

For simplicity we always serialize command buffers sent to the same command queue.

Allocator

The allocator will forward allocation requests to `cuMemHostAlloc` for host accessible memory and `cuMemAlloc` for device only memory.

Buffer

CUDA buffers are represented either as a host pointer or a device pointer of type `CUdeviceptr`.

Executable

HAL executable maps naturally to a PTX module. The compiler will generate a flat buffer containing a PTX text module as well as a list of entry point function names and the workgroup size associated with those entry points.

Semaphore

Timeline semaphore is used in IREE to handle coarse grain synchronization for CPU<->GPU, GPU<->GPU and CPU<->CPU. The interface follows closely `Vulkan timeline semaphore spec`. There is currently no simple way to implement this on CUDA. There are several solutions discussed on this `IREE issue` but no obvious solution. For now we force CPU and GPU to be synchronized after every submit to ensure correctness and ignore the semaphore.

NVVM Codegen

NVVM and PTX

NVVM is a CUDA specific IR composed of LLVM IR and NVVM specific intrinsics. It can be compiled to PTX text using LLVM PTX backend. NVVM has an associated dialect in MLIR that translates 1:1 to NVVM intrinsics. This is what we are using to generate the PTX kernel code.

IREE flow

IREE's `target independent codegen` converts the compiler input to Linalg on Tensors. Afterward IREE will call the LinalgToLLVMGPU codegen passes.

Once we get into LinalgToLLVMGPU passes we first do bufferize to generate Linalg on Buffers. Then we apply MLIR generic passes to convert linalg to SCF dialect and then SCF to Standard dialect. After that we convert Standard dialect to LLVM+NVVM dialect.

Example

Save the following mlir in /tmp/add.mlir

```
func.func @add(%arg0: tensor<4xf32>, %arg1: tensor<4xf32>) -> tensor<4xf32>
{
  %0 = tensor.empty() : tensor<4xf32>
  %1 = linalg.generic {
    indexing_maps = [
      affine_map<(d0) -> (d0)>, affine_map<(d0) -> (d0)>, affine_map<(d0) -
> (d0)>], iterator_types = ["parallel"]}
      ins(%arg0, %arg1 : tensor<4xf32>, tensor<4xf32>)
      outs(%0 : tensor<4xf32>) {
  ^bb0(%in: f32, %in_0: f32, %out: f32):
    %2 = arith.addf %in, %in_0 : f32
    linalg.yield %2 : f32
  } -> tensor<4xf32>
  return %1 : tensor<4xf32>
}
```

```
# First compile into a VM bytecode module.
$ ../iree-build/tools/iree-compile \
  --iree-hal-target-backends=cuda \
  /tmp/add.mlir \
  -o /tmp/add.vmfb

# Run the module through CUDA HAL backend.
$ ../iree-build/tools/iree-run-module \
  --device=cuda \
  --module=/tmp/add.vmfb \
  --function=add \
  --input="4xf32=[1 2 3 4]" \
  --input="4xf32=[2 2 2 2]"

EXEC @add
4xf32=3 4 5 6
```