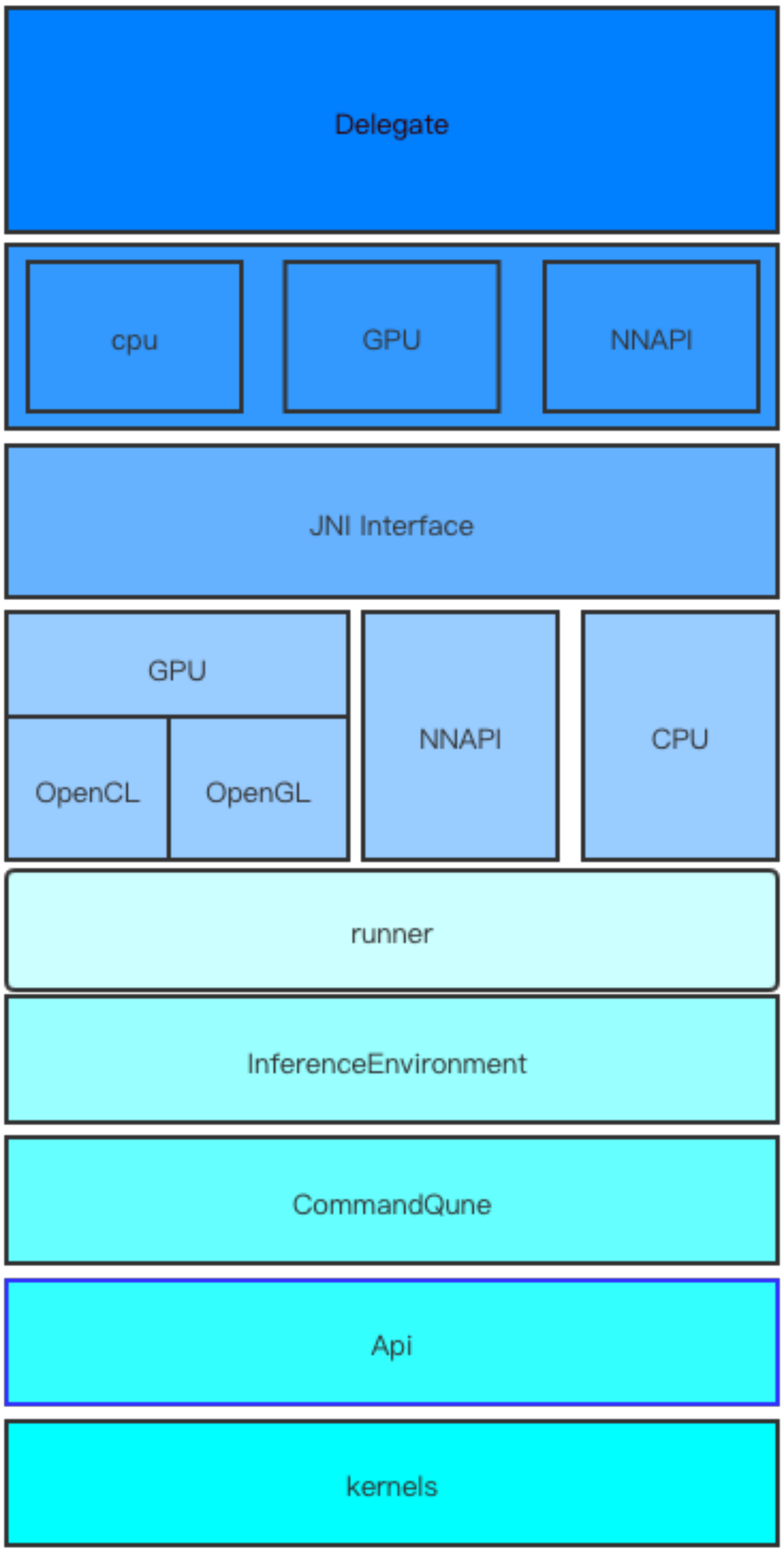
AI学习笔记--lite 源码库--delegates

- 简介

这里主要介绍的是运算方式的 delegates，以 GPU 为主。大致的结构如下：

-


Delegate
cpu
GPU
NNAPI
JNI Interface
GPU
OpenCL
OpenGL
NNAPI
CPU
runner
InferenceEnvironment
CommandQune
Api
kernels

- 应用层API

在上层应用的封装中，我们首先找到了一个构造方法，并且在构造方法上面定义了两个从库函数中引用的两个 delegate 代理类。

```java
/** holds a gpu delegate */
GpuDelegate gpuDelegate = null;
/** holds an nnapi delegate */
NnApiDelegate nnapiDelegate = null;

/** Initializes an {@code ImageClassifier}. */
ImageClassifier(Activity activity) throws IOException {
  tfliteModel = loadModelFile(activity);
  tflite = new Interpreter(tfliteModel, tfliteOptions);
  labelList = loadLabelList(activity);
  imgData =
      ByteBuffer.allocateDirect(
          DIM_BATCH_SIZE
              * getImageSizeX()
              * getImageSizeY()
              * DIM_PIXEL_SIZE
              * getNumBytesPerChannel());
  imgData.order(ByteOrder.nativeOrder());
  filterLabelProbArray = new float[FILTER_STAGES][getNumLabels()];
  Log.d(TAG, "Created a Tensorflow Lite Image Classifier.");
}
```

在内部暴露的接口方法中，有以下几个值得注意的方法：

```java
/**
* 方法描述:
*  使用 GPU 方式计算 Tensorflow model
* 参数:
* 返回:
* @version
*/
public void useGpu() {
  if (gpuDelegate == null) {
    gpuDelegate = new GpuDelegate();
    tfliteOptions.addDelegate(gpuDelegate);
    recreateInterpreter();
  }
}

/**
* 方法描述:
*  使用 CPU 方式计算 Tensorflow model
* 参数:
* 返回:
* @version
*/
public void useCPU() {
  recreateInterpreter();
}

/**
* 方法描述:
*  使用 NNAPI 方式计算 Tensorflow model
* 参数:
* 返回:
*  @version
*/
public void useNNAPI() {
  nnapiDelegate = new NnApiDelegate();
  tfliteOptions.addDelegate(nnapiDelegate);
  recreateInterpreter();
}
```

这里我们有个叫计算容器代理的概念，delegate 这个module是执行方式不同的介质代理容器。在 tf 参数配置项目中，加入对应的代理，从而在对应的容器中执行相应的算法。这里看到上层的有 GPU/CPU/NNAPI 等方式。

以 GPU 为例，说明整个代理过程，首先看到类的继承关系：

```
public class GpuDelegate implements Delegate, Closeable
```

GpuDelegate 实现了接口 Delegate，这个接口是获取在 C 层的对象指针地址。

```
//接口定义
package org.tensorflow.lite;

public interface Delegate {
    long getNativeHandle();
}
//Gpu 部分实现
public long getNativeHandle() {
    return this.delegateHandle;
}
```

那么这个地址是如何获取的呢？需要看到构造方法：

```
//C 层对象地址指针
private long delegateHandle;

//参数类构造方法
public GpuDelegate(GpuDelegate.Options options) {
    this.delegateHandle = createDelegate(options.precisionLossAllowed,
options.inferencePreference);
}
//无参数构造方法
public GpuDelegate() {
    this(new GpuDelegate.Options());
}
```

createDelegate 方法是一个 native 方法，并且除了这个还提供了一个 destory 方法用于释放对象。

```
//创建 JNI 代理，并且返回对象指针地址
private static native long createDelegate(boolean var0, int var1);

//依据地址指针释放对象空间
private static native void deleteDelegate(long var0);

//引入对应的 so
static {
    System.loadLibrary("tensorflowlite_gpu_jni");
}
```

操作节点配置有两个参数，我们看源码：

```
public static final class Options {
    // 单一模型计算参数
    public static final int INFERENCE_PREFERENCE_FAST_SINGLE_ANSWER = 0;
    // 持续调用计算参数
    public static final int INFERENCE_PREFERENCE_SUSTAINED_SPEED = 1;
    boolean precisionLossAllowed = true;
    int inferencePreference = 0;

    public Options() {
    }

    public GpuDelegate.Options setPrecisionLossAllowed(boolean 
precisionLossAllowed) {
        this.precisionLossAllowed = precisionLossAllowed;
        return this;
    }

    public GpuDelegate.Options setInferencePreference(int preference) {
        this.inferencePreference = preference;
        return this;
    }
}
```

综上所看到的的，默认 TensorFlow-Lite 是使用了单次推理计算的属性，也可以使用持续的，以适应不同的场景。

大致的理解了 JAVA 调用层 API，接下来我们看一下在 JNI 乃至 C 层的接口。

- ## 中间接 JNI 部分

这里需要看的代码在 /tensorflow/tensorflow/lite/delegates这个目录下。首先看外层，可以看到支持的代理模式有 flex、gpu、nnapi、xnnpack 这四个模块。还是以 GPU 为例，JNI 调用的接口部分在 Java 这个目录下，可能熟悉 bazel 编译工具的同学知道，在每个 build 文件中，可以引用大部分工程需要的文件，这样使用起来对某类大型项目管理会有很好的扩展性。在 java/src目录下面，拆分成为两个部分，一部分是 Java 接口，一部分是 JNI 的实现部分。JNI 的实现部分看gpu_delegate_jni.cc。

```
#include <jni.h>

#include "tensorflow/lite/delegates/gpu/delegate.h"

#ifdef __cplusplus
extern "C" {
#endif  // __cplusplus

JNIEXPORT jlong JNICALL
Java_org_tensorflow_lite_gpu_GpuDelegate_createDelegate(
    JNIEnv* env, jclass clazz, jboolean precision_loss_allowed,
    jint inference_preference) {
  TfLiteGpuDelegateOptionsV2 options = TfLiteGpuDelegateOptionsV2Default();
  if (precision_loss_allowed == JNI_TRUE) {
    options.inference_priority1 =
TFLITE_GPU_INFERENCE_PRIORITY_MIN_LATENCY;
    options.inference_priority2 =
        TFLITE_GPU_INFERENCE_PRIORITY_MIN_MEMORY_USAGE;
    options.inference_priority3 =
TFLITE_GPU_INFERENCE_PRIORITY_MAX_PRECISION;
  }
  options.inference_preference = static_cast<int32_t>
(inference_preference);
  return reinterpret_cast<jlong>(TfLiteGpuDelegateV2Create(&options));
}

JNIEXPORT void JNICALL
Java_org_tensorflow_lite_gpu_GpuDelegate_deleteDelegate(
    JNIEnv* env, jclass clazz, jlong delegate) {
  TfLiteGpuDelegateV2Delete(reinterpret_cast<TfLiteDelegate*>(delegate));
}

#ifdef __cplusplus
}  // extern "C"
#endif  // __cplusplus
```

从上面代码可以看到，代理的核心入口类是 delegates/xx/delegate.h 这里定义的，接口类实现的功能也就是创建一个对象并且返回对象的指针地址。调用的方法是 delegate 中定义的`TfLiteGpuDelegateV2Create`方法。跳转到头文件，只有 3 个方法：

```
// Populates TfLiteGpuDelegateOptionsV2 as follows:
//   is_precision_loss_allowed = false
//   inference_preference =
TFLITE_GPU_INFERENCE_PREFERENCE_FAST_SINGLE_ANSWER
//   priority1 = TFLITE_GPU_INFERENCE_PRIORITY_MAX_PRECISION
//   priority2 = TFLITE_GPU_INFERENCE_PRIORITY_AUTO
//   priority3 = TFLITE_GPU_INFERENCE_PRIORITY_AUTO
TFL_CAPI_EXPORT TfLiteGpuDelegateOptionsV2
TfLiteGpuDelegateOptionsV2Default();
```

```
// Creates a new delegate instance that need to be destroyed with
// TfLiteGpuDelegateV2Delete when delegate is no longer used by TFLite.
//
// This delegate encapsulates multiple GPU-acceleration APIs under the hood
to
// make use of the fastest available on a device.
//
// When `options` is set to `nullptr`, then default options are used.
TFL_CAPI_EXPORT TfLiteDelegate* TfLiteGpuDelegateV2Create(
    const TfLiteGpuDelegateOptionsV2* options);

// Destroys a delegate created with `TfLiteGpuDelegateV2Create` call.
TFL_CAPI_EXPORT void TfLiteGpuDelegateV2Delete(TfLiteDelegate* delegate);
```

函数的实现如下：

```
TfLiteDelegate* TfLiteGpuDelegateV2Create(
    const TfLiteGpuDelegateOptionsV2* options) {
  auto* gpu_delegate = new tflite::gpu::Delegate(options);
  TFLITE_LOG_PROD_ONCE(tflite::TFLITE_LOG_INFO,
                       "Created TensorFlow Lite delegate for GPU.");
  return gpu_delegate ? gpu_delegate->tflite_delegate() : nullptr;
}

void TfLiteGpuDelegateV2Delete(TfLiteDelegate* delegate) {
  delete tflite::gpu::GetDelegate(delegate);
}
```

创建完毕代理后，会对代理容器做一个参数配置初始化工作：

```
//初始化，设置 TFLiteContext 并且设置delegate参数
Status Prepare(TfLiteContext* context,
                 const TfLiteDelegateParams* delegate_params) {
    // Extract TFLite delegate execution plan from the context and convert
it
    // into FlowGraph32.
    GraphFloat32 graph;
    //创建容器
    RETURN_IF_ERROR(BuildFinalModel(context, delegate_params, &graph));

    //设置输入容器
    std::vector<uint32_t> input_refs;
    {
      const auto& inputs = graph.inputs();
      input_refs.reserve(inputs.size());
      for (auto input : inputs) {
        input_refs.push_back(input->tensor.ref);
      }
    }
    //设置输出容器
    std::vector<uint32_t> output_refs;
    {
      const auto& outputs = graph.outputs();
      output_refs.reserve(outputs.size());
      for (auto output : outputs) {
        output_refs.push_back(output->tensor.ref);
      }
    }

    std::unique_ptr<InferenceBuilder> builder;
    bool graph_is_destroyed;
    //初始化 OpenCL 接口 API，如果 OpenCL不支持，使用 OpenGL
    Status status = InitializeOpenClApi(&graph, &builder,
&graph_is_destroyed);
    if (!status.ok()) {
      context->ReportError(context, "%s", status.error_message().c_str());
      context->ReportError(context, "Falling back to OpenGL");
      // 一旦创建初始化失败，就会重新创建一个容器，并且初始化 OpenGL
      // Graph need to be re-created because it is moved above.
      GraphFloat32 graph2;
      if (graph_is_destroyed) {
        RETURN_IF_ERROR(BuildFinalModel(context, delegate_params,
&graph2));
      }
      RETURN_IF_ERROR(
          InitializeOpenGlApi(graph_is_destroyed ? &graph2 : &graph,
&builder));
    }

    // At this point tflite didn't allocate tensors yet, therefore, collect
    // indices and set all input and output tensors from tflite later.
    // TF在初期就创建好输入和输出的容器，之后都在对应的缓存中处理。
    input_indices_.reserve(input_refs.size());
    for (uint32_t tensor_index : input_refs) {
      const int64_t object_index = input_indices_.size();
      input_indices_.push_back(tensor_index);
      RETURN_IF_ERROR(
          builder->SetInputObjectDef(object_index,
GetObjectDef(tensor_index)));
    }
    output_indices_.reserve(output_refs.size());
    for (uint32_t tensor_index : output_refs) {
      const int64_t object_index = output_indices_.size();
      output_indices_.push_back(tensor_index);
      RETURN_IF_ERROR(builder->SetOutputObjectDef(object_index,
                                                  GetObjectDef(tensor_index
)));
    }

    return builder->Build(&runner_);
  }
```

代理的 API 在 cc的局部变量中有定义：

```
TfLiteDelegate delegate_ = {
      reinterpret_cast<void*>(this),  // .data_
      DelegatePrepare,                // .Prepare
      nullptr,                        // .CopyFromBufferHandle
      nullptr,                        // .CopyToBufferHandle
      nullptr,                        // .FreeBufferHandle
      kTfLiteDelegateFlagsNone,       // .flags
  };
//定义了参数、CL 命令容器、GL 命令容器、输入容器和输出容器等
  TfLiteGpuDelegateOptionsV2 options_;
  std::unique_ptr<cl::InferenceEnvironment> cl_environment_;
  std::unique_ptr<gl::InferenceEnvironment> gl_environment_;
  std::unique_ptr<InferenceRunner> runner_;
  std::vector<int64_t> input_indices_;
  std::vector<int64_t> output_indices_;
};
```

这里的 inferenceEnvironment是控制操作符，里面的定义在 api.h 中如下：

```
// CL 操作符
// Manages all resources that need to stay around as long as any inference
is
// running using the OpenGL backend.
class InferenceEnvironment {
public:
  virtual ~InferenceEnvironment() = default;

  virtual Status NewInferenceBuilder(
      GraphFloat32&& model, const InferenceOptions& options,
      std::unique_ptr<InferenceBuilder>* builder) = 0;
};

struct InferenceEnvironmentOptions {
  CommandQueue* queue = nullptr;
};

// Creates a new OpenGL environment that needs to stay around until all
// inference runners are destroyed.
Status NewInferenceEnvironment(
    const InferenceEnvironmentOptions& options,
    std::unique_ptr<InferenceEnvironment>* environment,
    InferenceEnvironmentProperties* properties /* optional */);

}  // namespace gl
}  // namespace gpu
}  // namespace tflite

// GL 操作符
// Manages all resources that need to stay around as long as any inference
is
// running using the OpenGL backend.
class InferenceEnvironment {
public:
  virtual ~InferenceEnvironment() = default;

  virtual Status NewInferenceBuilder(
      GraphFloat32&& model, const InferenceOptions& options,
      std::unique_ptr<InferenceBuilder>* builder) = 0;
};

struct InferenceEnvironmentOptions {
  CommandQueue* queue = nullptr;
};

//操作符命令：

// Encapsulated compilation/runtime tradeoffs.
enum class InferenceUsage {
  UNKNOWN,

  // InferenceRunner will be used only once. Therefore, it is important to
  // minimize bootstrap time as well.
  FAST_SINGLE_ANSWER,

  // Prefer maximizing the throughput. Same inference runner will be used
  // repeatedly on different inputs.
  SUSTAINED_SPEED,
};

// Defines aspects to control while instantiating a runner.
enum class InferencePriority {
  UNKNOWN,

  AUTO,

  MIN_LATENCY,

  MAX_PRECISION,

  MIN_MEMORY_USAGE,
};
```

里面一个重要的变量是CommandQueue，所有控制命令都存于此队列。这里分开分析。

- OpenGL

在入口方法，详细代码在 `command_queue.cc` 文件中，查看方法如下：

```
std::unique_ptr<CommandQueue> NewCommandQueue(const GpuInfo& gpu_info) {
  if (gpu_info.type == GpuType::ADRENO) {
    int flush_every_n = 1;
    // On Adreno 630 and Adreno 505 there is up to 2x performance boost
when
    // glFlush happens not so often.
    if (gpu_info.gpu_model == GpuModel::ADRENO630 ||
        gpu_info.gpu_model == GpuModel::ADRENO505) {
      flush_every_n = 10;
    }
    return absl::make_unique<AdrenoCommandQueue>(flush_every_n);
  }
  return absl::make_unique<DefaultCommandQueue>();
}
```

这里的代码可以看到，GPU 中 Adreno 505和 630 型号的片子支持并行操作，其余走的是 default 的形式。

- Adreno GPU

```
// On Adreno do flush periodically as this affects performance. Command
queue
// needs to be manually managed to ensure that accumulated work goes to GPU
as
// fast as it can.
//
// Also, on older Adreno devices glFlush is required after every memory
barrier
// to avoid hitting GPU driver bug.
class AdrenoCommandQueue : public DefaultCommandQueue {
public:
  explicit AdrenoCommandQueue(int flush_every_n)
      : flush_every_n_(flush_every_n) {}

  Status Dispatch(const GlProgram& program, const uint3& workgroups) final
{
    RETURN_IF_ERROR(DefaultCommandQueue::Dispatch(program, workgroups));
    if ((++program_counter_ % flush_every_n_) == 0) {
      glFlush();
    }
    return OkStatus();
  }

  Status WaitForCompletion() override {
    program_counter_ = 0;
    return DefaultCommandQueue::WaitForCompletion();
  }

  Status Flush() final {
    // Flush exactly once after the last dispatch.
    if (program_counter_ != 0) {
      program_counter_ = 0;
      glFlush();
    }
    return OkStatus();
  }

private:
  const int flush_every_n_;
  int program_counter_ = 0;
};

}  // namespace
```

- Default GPU

```
class DefaultCommandQueue : public CommandQueue {
public:
  Status Dispatch(const GlProgram& program, const uint3& workgroups)
override {
    RETURN_IF_ERROR(program.Dispatch(workgroups));
    return TFLITE_GPU_CALL_GL(glMemoryBarrier, GL_ALL_BARRIER_BITS);
  }

  Status WaitForCompletion() override {
    // TODO(akulik): Maybe let the user choose which wait method to use.
    return GlActiveSyncWait();
  }

  Status Flush() override { return OkStatus(); }
};
```

每一个 CommandQueue 都被赋予了三个接口方法：

```
class CommandQueue {
public:
  virtual ~CommandQueue() = default;

  // Dispatches a program. It may or may not call glFlush.
  virtual Status Dispatch(const GlProgram& program,
                          const uint3& workgroups) = 0;

  // Called at the end of dispatching of all programs.
  virtual Status Flush() = 0;

  // Waits until all programs dispatched prior this call are completed.
  virtual Status WaitForCompletion() = 0;
};
```

```
// By default memory barrier is inserted after every dispatch.
std::unique_ptr<CommandQueue> NewCommandQueue(const GpuInfo& gpu_info);

}  // namespace gl
}  // namespace gpu
}  // namespace tflite
```

GPU 部分还有两个重要的东西，一个是 API 的实现部分，还有一个 runner 部分。以 Android 平台为例，测试整个 GPU_Delegate 部分，可以 build 一个 so 出来，并且写一段测试代码。

- build 模块

TFLite 编译需要使用的是 Bazel 2.0,如果版本不对可能导致 build 异常。现在看整个 GPU 的 build 文件，如下：

```
# build -c opt --config android_arm64 --copt -Os --copt -
DTFLITE_GPU_BINARY_RELEASE --copt --linkopt -s --strip always
:libtensorflowlite_gpu_delegate.so
cc_binary(
    name = "libtensorflowlite_gpu_delegate.so",
    linkopts = [
        "-Wl,-soname=libtensorflowlite_gpu_delegate.so",
    ] + select({
        "//tensorflow:android": [
            "-lEGL",
            "-lGLESv3",
            "-fvisibility=hidden",
        ],
        "//tensorflow:windows": [],
        "//conditions:default": [
            "-fvisibility=hidden",
        ],
    }),
    linkshared = 1,
    linkstatic = 1,
    tags = [
        "nobuilder",
        "notap",
    ],
    deps = [":delegate"],
)

# bazel build -c opt --cpu ios_arm64 --copt -Os --copt -
DTFLITE_GPU_BINARY_RELEASE --copt -fvisibility=hidden --linkopt -s --strip
always --cxxopt=-std=c++14 :libtensorflowlite_gpu_metal --
apple_platform_type=ios
ios_static_framework(
    name = "tensorflow_lite_gpu_framework",
    hdrs = [
        "metal_delegate.h",
        "metal_delegate_internal.h",
    ],
    minimum_os_version = "10.0",
    deps = [":metal_delegate"],
)

# Note: Support for MacOS is best-effort at the moment.
# bazel build -c opt --copt -Os --copt -DTFLITE_GPU_BINARY_RELEASE --copt -
fvisibility=hidden --linkopt -s --strip always --cxxopt=-std=c++14
:tensorflow_lite_gpu_dylib --apple_platform_type=macos
macos_dylib(
    name = "tensorflow_lite_gpu_dylib",
    minimum_os_version = "10.13",
    tags = [
        "manual",
        "nobuilder",
        "notap",
    ],
    deps = [
        ":metal_delegate",
        ":metal_delegate_internal",
    ],
)
```

这里可以 build Mac OS 的 GPUDelegate 也可以 build Android 可用的 so。暴露的头文件包括了 delegate.h 等，使用 build 命令对整个模块进行 build。

```
➜  tensorflow git:(master) ✗ bazel build --config android_arm64
//tensorflow/lite/delegates/gpu:libtensorflowlite_gpu_delegate.so
```

在按照官方文档介绍，编辑代码:

```
////////
// Set up interpreter.
auto model = FlatBufferModel::BuildFromFile(model_path);
ops::builtin::BuiltinOpResolver op_resolver;
std::unique_ptr<Interpreter> interpreter;
InterpreterBuilder(*model, op_resolver)(&interpreter);

////////
// NEW: Prepare GPU delegate.
auto* delegate = TfLiteGpuDelegateV2Create(/*default options=*/nullptr);
if (interpreter->ModifyGraphWithDelegate(delegate) != kTfLiteOk) return;

////////
// Run inference.
WriteToInputTensor(interpreter->typed_input_tensor<float>(0));
if (interpreter->Invoke() != kTfLiteOk) return;
ReadFromOutputTensor(interpreter->typed_output_tensor<float>(0));
```

```
////////
// Clean up.
TfLiteGpuDelegateV2Delete(delegate);
```

如此，可以 build 一个模块用于测试 delegate 部分功能。