3.TensorRT的C++接口解析

本章说明 C++ API 的基本用法，假设您从 ONNX 模型开始。 sampleOnnxMNIST更详细地说明了这个用例。

C++ API 可以通过头文件 `NvInfer.h` 访问，并且位于 `nvinfer1` 命名空间中。例如，一个简单的应用程序可能以：

```
#include “NvInfer.h”

using namespace nvinfer1;
```

TensorRT C++ API 中的接口类以前缀 `I` 开头，例如 `ILogger` 、 `IBuilder` 等。

CUDA 上下文会在 TensorRT 第一次调用 CUDA 时自动创建，如果在该点之前不存在。通常最好在第一次调用 TensoRT 之前自己创建和配置 CUDA 上下文。
为了说明对象的生命周期，本章中的代码不使用智能指针；但是，建议将它们与 TensorRT 接口一起使用。

3.1. The Build Phase

要创建构建器，首先需要实例化ILogger接口。此示例捕获所有警告消息，但忽略信息性消息：

```
class Logger : public ILogger
{
    void log(Severity severity, const char* msg) noexcept override
    {
        // suppress info-level messages
        if (severity <= Severity::kWARNING)
            std::cout << msg << std::endl;
    }
} logger;
```

然后，您可以创建构建器的实例：

```
IBuilder* builder = createInferBuilder(logger);
```

3.1.1. Creating a Network Definition

创建构建器后，优化模型的第一步是创建网络定义：

```
uint32_t flag = 1U <<static_cast<uint32_t>
    (NetworkDefinitionCreationFlag::kEXPLICIT_BATCH);

INetworkDefinition* network = builder->createNetworkV2(flag);
```

为了使用 ONNX 解析器导入模型，需要 `kEXPLICIT_BATCH` 标志。有关详细信息，请参阅显式与隐式批处理部分。

3.1.2. Importing a Model using the ONNX Parser

现在，需要从 ONNX 表示中填充网络定义。 ONNX 解析器 API 位于文件NvOnnxParser.h中，解析器位于`nvonnxparser` C++ 命名空间中。

```
#include “NvOnnxParser.h”

using namespace nvonnxparser;
```

您可以创建一个 ONNX 解析器来填充网络，如下所示：

```
IParser*  parser = createParser(*network, logger);
```

然后，读取模型文件并处理任何错误。

```
parser->parseFromFile(modelFile,
    static_cast<int32_t>(ILogger::Severity::kWARNING));
for (int32_t i = 0; i < parser.getNbErrors(); ++i)
{
std::cout << parser->getError(i)->desc() << std::endl;
}
```

TensorRT 网络定义的一个重要方面是它包含指向模型权重的指针，这些指针由构建器复制到优化的引擎中。由于网络是通过解析器创建的，解析器拥有权重占用的内存，因此在构建器运行之前不应删除解析器对象。

3.1.3. Building an Engine

下一步是创建一个构建配置，指定 TensorRT 应该如何优化模型。

```
IBuilderConfig* config = builder->createBuilderConfig();
```

这个接口有很多属性，你可以设置这些属性来控制 TensorRT 如何优化网络。一个重要的属性是最大工作空间大小。层实现通常需要一个临时工作空间，并且此参数限制了网络中任何层可以使用的最大大小。如果提供的工作空间不足，TensorRT 可能无法找到层的实现。默认情况下，工作区设置为给定设备的总全局内存大小；必要时限制它，例如，在单个设备上构建多个引擎时。

```
config->setMemoryPoolLimit(MemoryPoolType::kWORKSPACE, 1U << 20);
```

一旦指定了配置，就可以构建引擎。

```
IHostMemory*  serializedModel = builder->buildSerializedNetwork(*network,
*config);
```

由于序列化引擎包含权重的必要拷贝，因此不再需要解析器、网络定义、构建器配置和构建器，可以安全地删除：

```
delete parser;
delete network;
delete config;
delete builder;
```

然后可以将引擎保存到磁盘，并且可以删除它被序列化到的缓冲区。

```
delete serializedModel
```

注意：序列化引擎不能跨平台或 TensorRT 版本移植。引擎特定于它们构建的确切 GPU 模型（除了平台和 TensorRT 版本）。

3.2. Deserializing a Plan

假设您之前已经序列化了一个优化模型并希望执行推理，您将需要创建一个运行时接口的实例。与构建器一样，运行时需要一个记录器实例：

```
IRuntime* runtime = createInferRuntime(logger);
```

假设您已将模型从缓冲区中读取，然后可以对其进行反序列化以获得引擎：

```
ICudaEngine* engine =
  runtime->deserializeCudaEngine(modelData, modelSize);
```

3.3. Performing Inference

引擎拥有优化的模型，但要执行推理，我们需要管理中间激活的额外状态。这是通过 `ExecutionContext` 接口完成的：

```
IExecutionContext *context = engine->createExecutionContext();
```

一个引擎可以有多个执行上下文，允许一组权重用于多个重叠的推理任务。（当前的一个例外是使用动态形状时，每个优化配置文件只能有一个执行上下文。）

要执行推理，您必须为输入和输出传递 TensorRT 缓冲区，TensorRT 要求您在指针数组中指定。您可以使用为输入和输出张量提供的名称查询引擎，以在数组中找到正确的位置：

```
int32_t inputIndex = engine->getBindingIndex(INPUT_NAME);
int32_t outputIndex = engine->getBindingIndex(OUTPUT_NAME);
```

使用这些索引，设置一个缓冲区数组，指向 GPU 上的输入和输出缓冲区：

```
void* buffers[2];
buffers[inputIndex] = inputBuffer;
buffers[outputIndex] = outputBuffer;
```

然后，您可以调用 TensorRT 的 enqueue 方法以使用CUDA 流异步启动推理：

```
context->enqueueV2(buffers, stream, nullptr);
```

通常在内核之前和之后将`cudaMemcpyAsync()`排入队列以从 GPU 中移动数据（如果数据尚不存在）。`enqueueV2()`的最后一个参数是一个可选的 CUDA 事件，当输入缓冲区被消耗时发出信号，并且可以安全地重用它们的内存。

要确定内核（可能还有`memcpy()`）何时完成，请使用标准 CUDA 同步机制，例如事件或等待流。

如果您更喜欢同步推理，请使用`executeV2`方法而不是`enqueueV2`。