TensorRT如何工作

本章提供了有关 TensorRT 工作原理的更多详细信息。

5.1. Object Lifetimes

TensorRT 的 API 是基于类的，其中一些类充当其他类的工厂。对于用户拥有的对象，工厂对象的生命周期必须跨越它创建的对象的生命周期。例如，`NetworkDefinition`和`BuilderConfig`类是从构建器类创建的，这些类的对象应该在构建器工厂对象之前销毁。

此规则的一个重要例外是从构建器创建引擎。创建引擎后，您可以销毁构建器、网络、解析器和构建配置并继续使用引擎。

5.2. Error Handling and Logging

创建 TensorRT 顶级接口（`builder`、`runtime`或`refitter`）时，您必须提供`Logger`（C++、Python）接口的实现。记录器用于诊断和信息性消息；它的详细程度是可配置的。由于记录器可用于在 TensorRT 生命周期的任何时间点传回信息，因此它的生命周期必须跨越应用程序中对该接口的任何使用。实现也必须是线程安全的，因为 TensorRT 可以在内部使用工作线程。

对对象的 API 调用将使用与相应顶级接口关联的记录器。例如，在对`ExecutionContext::enqueue()`的调用中，执行上下文是从引擎创建的，该引擎是从运行时创建的，因此 TensorRT 将使用与该运行时关联的记录器。

错误处理的主要方法是`ErrorRecorde`（C++, Python）接口。您可以实现此接口，并将其附加到 API 对象以接收与该对象关联的错误。对象的记录器也将传递给它创建的任何其他记录器 - 例如，如果您将错误记录器附加到引擎，并从该引擎创建执行上下文，它将使用相同的记录器。如果您随后将新的错误记录器附加到执行上下文，它将仅接收来自该上下文的错误。如果生成错误但没有找到错误记录器，它将通过关联的记录器发出。

请注意，CUDA 错误通常是**异步**的 - 因此，当执行多个推理或其他 CUDA 流在单个 CUDA 上下文中异步工作时，可能会在与生成它的执行上下文不同的执行上下文中观察到异步 GPU 错误。

5.3 Memory

TensorRT 使用大量设备内存，即 GPU 可直接访问的内存，而不是连接到 CPU 的主机内存。由于设备内存通常是一种受限资源，因此了解 TensorRT 如何使用它很重要。

5.3.1. The Build Phase

在构建期间，TensorRT 为时序层实现分配设备内存。一些实现可能会消耗大量临时内存，尤其是在使用大张量的情况下。您可以通过构建器的`maxWorkspace`属性控制最大临时内存量。这默认为设备全局内存的完整大小，但可以在必要时进行限制。如果构建器发现由于工作空间不足而无法运行的适用内核，它将发出一条日志消息来指示这一点。

然而，即使工作空间相对较小，计时也需要为输入、输出和权重创建缓冲区。 TensorRT 对操作系统因此类分配而返回内存不足是稳健的，但在某些平台上，操作系统可能会成功提供内存，随后内存不足killer进程观察到系统内存不足，并终止 TensorRT .如果发生这种情况，请在重试之前尽可能多地释放系统内存。

在构建阶段，通常在主机内存中至少有两个权重拷贝：来自原始网络的权重拷贝，以及在构建引擎时作为引擎一部分包含的权重拷贝。此外，当 TensorRT 组合权重（例如卷积与批量归一化）时，将创建额外的临时权重张量。

5.3.2. The Runtime Phase

在运行时，TensorRT 使用相对较少的主机内存，但可以使用大量的设备内存。

引擎在反序列化时分配设备内存来存储模型权重。由于序列化引擎几乎都是权重，因此它的大小非常接近权重所需的设备内存量。

`ExecutionContext` 使用两种设备内存：

- 一些层实现所需的持久内存——例如，一些卷积实现使用边缘掩码，并且这种状态不能像权重那样在上下文之间共享，因为它的大小取决于层输入形状，这可能因上下文而异。该内存在创建执行上下文时分配，并在其生命周期内持续。
- 暂存内存，用于在处理网络时保存中间结果。该内存用于中间激活张量。它还用于层实现所需的临时存储，其边界由 `IBuilderConfig::setMaxWorkspaceSize()` 控制。

您可以选择通过 `ICudaEngine::createExecutionContextWithoutDeviceMemory()` 创建一个没有暂存内存的执行上下文，并在网络执行期间自行提供该内存。这允许您在未同时运行的多个上下文之间共享它，或者在推理未运行时用于其他用途。 `ICudaEngine::getDeviceMemorySize()` 返回所需的暂存内存量。

构建器在构建网络时发出有关执行上下文使用的持久内存和暂存内存量的信息，严重性为 `kINFO` 。检查日志，消息类似于以下内容：

```
[08/12/2021-17:39:11] [I] [TRT] Total Host Persistent Memory: 106528
[08/12/2021-17:39:11] [I] [TRT] Total Device Persistent Memory: 29785600
[08/12/2021-17:39:11] [I] [TRT] Total Scratch Memory: 9970688
```

默认情况下，TensorRT 直接从 CUDA 分配设备内存。但是，您可以将 TensorRT 的 `IGpuAllocator` （ C++ 、 Python ）接口的实现附加到构建器或运行时，并自行管理设备内存。如果您的应用程序希望控制所有 GPU 内存并子分配给 TensorRT，而不是让 TensorRT 直接从 CUDA 分配，这将非常有用。

TensorRT 的依赖项（ cuDNN和cuBLAS ）会占用大量设备内存。 TensorRT 允许您通过构建器配置中的 `TacticSources` （ C++ 、 Python ）属性控制这些库是否用于推理。请注意，某些层实现需要这些库，因此当它们被排除时，网络可能无法编译。

CUDA 基础设施和 TensorRT 的设备代码也会消耗设备内存。内存量因平台、设备和 TensorRT 版本而异。您可以使用 `cudaGetMemInfo` 来确定正在使用的设备内存总量。

注意：由于 CUDA 无法控制统一内存设备上的内存，因此 `cudaGetMemInfo` 返回的结果在这些平台上可能不准确。

5.4. Threading

一般来说，TensorRT 对象不是线程安全的。预期的运行时并发模型是不同的线程将在不同的执行上下文上操作。上下文包含执行期间的网络状态（激活值等），因此在不同线程中同时使用上下文会导致未定义的行为。
为了支持这个模型，以下操作是线程安全的：

- 运行时或引擎上的非修改操作。
- 从 TensorRT 运行时反序列化引擎。
- 从引擎创建执行上下文。

- 注册和注销插件。

在不同线程中使用多个构建器没有线程安全问题；但是，构建器使用时序来确定所提供参数的最快内核，并且使用具有相同 GPU 的多个构建器将扰乱时序和 TensorRT 构建最佳引擎的能力。使用多线程使用不同的 GPU 构建不存在此类问题。

5.5. Determinism

TensorRT `builder` 使用时间来找到最快的内核来实现给定的运算符。时序内核会受到噪声的影响——GPU 上运行的其他工作、GPU 时钟速度的波动等。时序噪声意味着在构建器的连续运行中，可能不会选择相同的实现。

`AlgorithmSelector`（C++，Python）接口允许您强制构建器为给定层选择特定实现。您可以使用它来确保构建器从运行到运行选择相同的内核。有关更多信息，请参阅算法选择和可重现构建部分。

一旦构建了引擎，它就是确定性的：在相同的运行时环境中提供相同的输入将产生相同的输出。