Application Note 002

Pass Manager Getting Started Guide

This application note describes the pass manager software architecture and basics to implement a pass in ONNC. This note also provides a concrete pass example in ONNC which counts neural network operators in an ONNX model for reference. This document is compliant with the ONNC Community Docker image v1.0. You may download the Docker image from Docker Hub[1]

[1] https://hub.docker.com/r/onnc/onnc-community

TABLE OF CONTENT

1. Introduction

ONNC inherits the concept of pass management from the LLVM infrastructure and the pass manager is one of the most important features in ONNC as well. Any analysis or transformation on a target program can be implemented as a pass in the ONNC framework. The design philosophy behind the ONNC pass manager is not only to support the concept of pass management from LLVM but also to enable automatic iterative compilation in ONNC. The LLVM pass manager stops immediately when something fails at any pass. It relies on users to adjust parameters and retry. However, compilation failures occur much more frequently in the neural network model compilation than in the traditional (e.g. C/C++) compilation. Therefore, it is an important feature for ONNC to support iterative compilation and embed the design in the pass manager.

The ONNC framework takes care of most functionalities of the pass manager including automatic pass scheduling and inter-pass dependencies. In this application note, we focus on the pass implementation more than the pass manager internals since most ONNC users spend their efforts in designing a new pass other than modifying the pass manager.

2. Pass

Pass is an abstraction of each execution in ONNC framework. It is designed for manipulating an ONNC IR graph to achieve a specific goal. Users may define customized pass types, register a pass into pass manager, and let pass manager administrate the executions.

2.1 Inheriting from the *CustomPass<T>* abstract class

The ***CustomPass<T>*** abstract class defines several virtual functions. These member functions are invoked by the pass manager on each execution.

Prototype
virtual ReturnType doInitialization(Module&);
virtual ReturnType runOnModule(Module&) = 0;
virtual ReturnType doFinalization(Module&);

Method	Description
doInitialization	The first-invoked method in a pass. Acquire resources such as files, network and etc.
runOnModule	Implement module manipulations in this method.
doFinalization	The last-called method in a pass. Release resources and prepare next run.

The above three methods are invoked exactly once per run. Users can assemble meaningful values and return informative result to pass manager. ONNC use an enumeration type ***PassResult*** to distinguish execution results. ***PassResult*** is usually encoded as bit mask and the following table lists all possible values.

Value	Description
`kModuleNoChanged`	No update to the module content or do nothing.
`kModuleChanged`	There are some modifications on module or invoke successfully.
`kPassRetry`	Cannot finish invocation due to some reason. Need to retry.
`kPassFailure`	Failed to action.

2.2 Overriding *runOnModule()*

CustomPass<T> has a default ***doInitialization()*** funtion doing nothing and ***doFinalization()*** returning ***kModuleNoChanged***. Users can write their own passes by simply deriving from ***CustomPass<T>*** and override the ***runOnModule()*** virtual function.

```
class MyPass : public CustomPass<MyPass> {
public:
    ReturnType runOnModule(Module& module) override {
        // do something here
        return kModuleChanged;
    }
};
```

The type argument in ***CustomPass<T>*** has to be the same as the derived class name.

2.3 Defining Pass Dependency

If a customized pass needs output generated by other pass, users have to override the method ***getAnalysisUsage()*** to pass the pass dependency information to the pass manager.

Prototype
virtual void getAnalysisUsage(AnalysisUsage&) const;

The ***AnalysisUsage*** object contains the analysis usage information of a pass. Users need to call its ***addRequired()*** function to define pass dependency. The following code snippet shows an example where ***MyPass*** depends on the other two passes, ***Foo*** and ***Bar***. Once the pass manager is aware of the dependency, it will execute ***Foo*** and ***Bar*** before ***MyPass***.

```
class Foo: public CustomPass<Foo> { /* implementation goes here */ };
class Bar: public CustomPass<Bar> { /* implementation goes here */ };

class MyPass : public CustomPass<MyPass> {
public:
    /* other code here */
    void getAnalysisUsage(AnalysisUsage& usage) const override {
        usage.addRequired<Foo>();
        usage.addRequired<Bar>();
    }
};
```

3. Pass Manager

Pass manager is designed to manage pass instances and pass executions. ONNC provides simple APIs for users to register passes, their dependencies, and administrate pass execution. From users' point of view, the implementation details of the pass manager are handled by the ONNC framework. In most cases, users only need to focus on how to register a pass to the pass manager.

3.1 Registering a Pass

Users have to register a pass object via the method ***add()*** in the pass manager before they can be executed. There is only one registered pass object running at same time. The ***add()*** prototype is shown as below.

Prototype
void add(Pass*);

The following example shows how a pass object is registered to the pass manager.

```
PassManager manager;
manager.add(new MyPass);
```

Pass manager gets pass dependency via the ***getAnalysisUsage()*** method, create and run unregistered pass objects if users declare such dependency in their customized pass type. Note that conditional dependency is not supported in the ONNC framework

because the output of the ***getAnalysisUsage()*** method has to remain the same throughout the compilation process.