Hello,World with MLIR (2)

25 Dec 2020

Continuing from the last article to create minimal Dialect to print tensor element with MLIR, I am going to illustrate the structure of the codebase of Dialect.

As noted previously, I put the whole repository on Lewuathe/mlir-hello. Please take a look into that if you need to know more.

Code Structure

The official site contains the general guide to create Dialect. Here is the illustration of the structure of the repository.

```
├── CMakeLists.txt
├── README.md
├── hello-opt
│   ├── CMakeLists.txt
│   └── hello-opt.cpp
├── hello-translate
│   ├── CMakeLists.txt
│   └── hello-translate.cpp
├── include
│   ├── CMakeLists.txt
│   └── Hello
│       ├── CMakeLists.txt
│       ├── HelloDialect.h
│       ├── HelloDialect.td
│       ├── HelloOps.h
│       ├── HelloOps.td
│       └── HelloPasses.h
├── lib
│   ├── CMakeLists.txt
│   └── Hello
│       ├── CMakeLists.txt
│       ├── HelloDialect.cpp
│       ├── HelloOps.cpp
```

```
│       ├── LowerToAffine.cpp
│       └── LowerToLLVM.cpp
├── test
│   ├── CMakeLists.txt
│   ├── Hello
│   │   ├── dummy.mlir
│   │   ├── print.mlir
│   │   ├── sample-opt.mlir
│   │   └── sample-translate.mlir
│   ├── lit.cfg.py
│   └── lit.site.cfg.py.in
```

ODS Declarations

`include` directory needs to include definitions of Dialect and Operations in Operation Definition Specification format (ODS). ODS is a framework to define the specification of Dialect and Operations declaratively. This framework is powered by the TableGen mechanism maintained in LLVM Core. MLIR generates the C++ code from the ODS declaration. We need to write the following code in CMakeFiles.

```
# Add the HelloOps for the dialect operations
add_mlir_dialect(HelloOps hello)

# Necessary to generate documentation
add_mlir_doc(HelloDialect -gen-dialect-doc HelloDialect Hello/)
add_mlir_doc(HelloOps -gen-op-doc HelloOps Hello/)
```


With this directive, CMake automatically generates the header files named `HelloOpsDialect.h.inc` and `HelloOps.h.inc` containing C++ code corresponding to the Dialect and operations you defined. We must include these files explicitly in the hand-written header files.

`HelloDialect.h`

```
#include "Hello/HelloOpsDialect.h.inc"
```

`HelloOps.h`

```
#define GET_OP_CLASSES
#include "Hello/HelloOps.h.inc"
```

It's worth noting that `HelloOps.h` uses preprocessor directive `#define GET_OP_CLASSES`. Interestingly `HelloOps.h.inc` contains several distinct sections in a file to fetch the only necessary information as desired by using the preprocessor directive. `GET_OP_CLASSES` will expand the declarations of operation classes.

Implementation Classes

The code implementing the operation, transformation, etc., should be put in the `lib/Hello` directory. `HelloDialect.cpp` needs to have an initializer at least.

```
#include "mlir/IR/Builders.h"
#include "mlir/IR/OpImplementation.h"

#include "Hello/HelloDialect.h"
#include "Hello/HelloOps.h"

using namespace mlir;
using namespace hello;

void HelloDialect::initialize() {
  addOperations<
#define GET_OP_LIST
#include "Hello/HelloOps.cpp.inc"
      >();
}
```

Note that we use `GET_OP_LIST` to render all the names of operations supported by Hello Dialect. Similarly, we can write the `HelloOps.cpp` file as follows.

```
#include "Hello/HelloOps.h"
#include "Hello/HelloDialect.h"
#include "mlir/IR/OpImplementation.h"

#define GET_OP_CLASSES
#include "Hello/HelloOps.cpp.inc"
```

This structure makes clear the separation between Dialect-related implementation and Operation-related implementation.

Passes for Lowering

In addition to these files, the Hello dialect has two files for lowering the Hello code to LLVM. `LowerToAffine.cpp` and `LowerToLLVM.cpp`. These passes define the way to convert one Dialect to another dialect. In our case, Hello Dialect must be compiled into the executable format to run it. Since the code is transformed into LLVM IR format, we can execute it. Therefore the goal of these passes is lowering Hello Dialect to LLVM while passing Affine, Standard dialects. In `hello-op` CLI, we register these passes as follows.

```
// Register passes to be applied in this compile process
mlir::PassManager passManager(&context);
mlir::OpPassManager &optPm = passManager.nest<mlir::FuncOp>();
optPm.addPass(hello::createLowerToAffinePass());
passManager.addPass(hello::createLowerToLLVMPass());
```

We will look into the detail for the transformation and pass the infrastructure itself another time.

The following directive in CMake is required to compile the project properly. You can add additional libraries as you like here if necessary.

```
add_mlir_dialect_library(MLIRHello
    HelloDialect.cpp
    HelloOps.cpp
    LowerToAffine.cpp
    LowerToLLVM.cpp

    ADDITIONAL_HEADER_DIRS
    ${PROJECT_SOURCE_DIR}/include/Hello

    DEPENDS
    MLIRHelloOpsIncGen

    LINK_LIBS PUBLIC
```

```
    MLIRIR
  )
```

Run hello-opt

`hello-opt` is a tool to convert Hello dialect code to LLVM IR quickly. It loads necessary dialects from the registry. The MLIR module is loaded and transformed into the `mlir::OwningModuleRef` class.

```
int main(int argc, char **argv) {
  mlir::registerPassManagerCLOptions();
  cl::ParseCommandLineOptions(argc, argv, "Hello compiler\n");

  mlir::registerAllPasses();
  mlir::MLIRContext context;
  context.getOrLoadDialect<hello::HelloDialect>();
  context.getOrLoadDialect<mlir::StandardOpsDialect>();
  context.getOrLoadDialect<mlir::LLVM::LLVMDialect>();

  mlir::OwningModuleRef module;
  if (int error = loadAndProcessMLIR(context, module)) {
    return error;
  }

  dumpLLVMIR(*module);

  return 0;
}
```

Let's say we have the following Hello dialect code.

```
func @main() {
    %0 = "hello.constant"() {value = dense<1.0> : tensor<2x3xf64>} :
 () -> tensor<2x3xf64>
    "hello.print"(%0) : (tensor<2x3xf64>) -> ()
    return
}
```

It defines a constant tensor whose all elements are 1.0 with the shape <2x3>. And print each element according to its tensor shape. Let's execute it.

Build the project as follows.

```
mkdir build && cd build

# Path to the LLVM artifacts we build previously
LLVM_DIR=/path/to/llvm-project/build/lib/cmake/llvm \
  MLIR_DIR=/path/to/llvm-project/build/lib/cmake/mlir \
  cmake -G Ninja ..

cmake --build . --target hello-opt
```

`hello-op` will dump the LLVM IR into the `print.ll` file.

```
# Lower MLIR to LLVM IR
./build/bin/hello-opt ./test/Hello/print.mlir > /path/to/print.ll
```

You can use `lli` to execute the LLVM bitcode format interactively.

```
lli /path/to/print.ll

1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
```

It works finally!

Besides that, MLIR has many exciting topics to be discussed, such as Interfaces, DRR for rewriting. Please visit the great official website for more about MLIR. I'll extend the Hello dialect more if I get a chance to do so.

Enjoy!

Hello,World with MLIR (1)

24 Dec 2020

Machine learning is one of the hottest fields these days. No one can stop this emerging field's progress, and no one can know the true potential of machine learning applications. The new concept or new design of the model is appearing day by day. That's true.

But it's also true that every machine learning (ML) application, artificial intelligence (AI), is still running the ordinary von-Neumann computer we are all familiar with. Hence, all ML applications and AI need to be compiled into the format our favorite can execute. The shadow hero of this challenge is **compilers**.

The intensive research in the field of a compiler for ML has been done this year. You can find vast amount of papers regarding this topic. In addition to the research, there is much software implementing the cutting-edge concept of the compiler for ML. Here is a tiny part of the whole compiler for ML.

- MLIR
- Apache TVM
- ProGraML

In this article, I tried to create a minimal example to show a “Hello, World”-like a message with MLIR. (Reality, the title should have been *“Hello, Tensor in MLIR”*) You can find the excellent tutorial documentation in the official site of MLIR. But the toy language is a little bit larger than I expected. All I want to do was printing some data in standard output with the code generated by MLIR.

The whole code has been put in Lewuathe/mlir-hello.

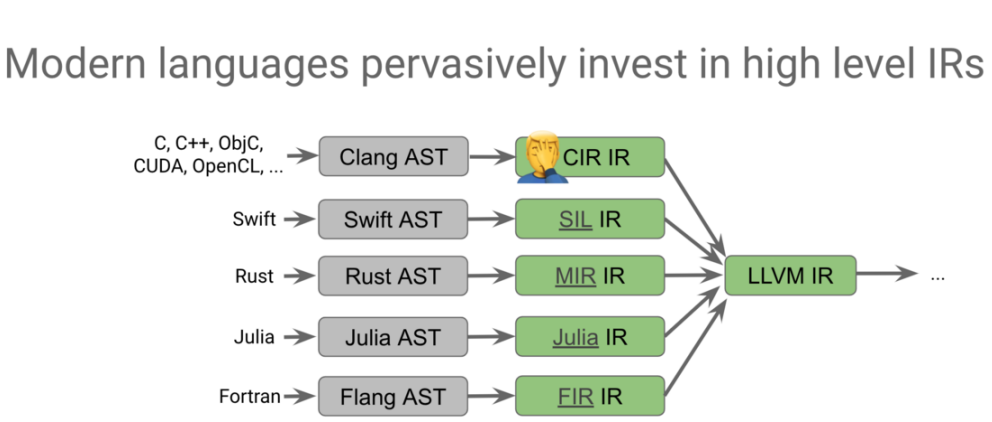
With this Dialect, you can run the following program to print the elements of a tensor.

```
# Lower MLIR to LLVM IR and execute it
$ ./build/bin/hello-opt ./test/Hello/print.mlir | lli
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
```

We are going to walk through the series of articles working as practical tutorials to develop MLIR Dialect.

What is MLIR

Let’s briefly describe MLIR first. MLIR is a compiler infrastructure supporting higher-level intermediate representation than LLVM. More accurately, MLIR is a framework to extend the compiler infrastructure itself. In the past, we had a bunch of front-end languages, optimization passes, and backend hardware. They are isolated, and the code which otherwise should be reused is reinvented again and again. LLVM successfully achieves to remove the code duplication and share the knowledge among developers. But fine-grained optimization process may require a higher-level understanding of the front-end language semantics and structure.



- Language specific optimizations
- Dataflow driven type checking - e.g. definitive initialization, borrow checker
- Progressive lowering from high level abstractions

That's why many languages develop their own IR (e.g., SIL, MIR) for the optimization and transformation along awarding the high-level language structures. The motivation of MLIR is to remove this redundant work by introducing the comprehensive framework for the high-level transformation of IR.

The ML in MLIR does not stand for Machine Learning. It's an abbreviation of **Multi-Layer Intermediate Representation** whose goal is to lower the high-level language and keep the multi-layered language structures defined by **Dialect** to hardware code so that the MLIR framework is capable of recognizing the semantics of the higher-level language. This lowering process is the core idea of MLIR. Please see lowering part in the documentation for more.

MLIR supports plugin-architecture called Dialect. You can extend almost anything of MLIR functionality by using that. That's so powerful that you may find yourself an inventor of a programming language! We'll see how we implement a new dialect by looking into the Hello Dialect I have written.

Install LLVM and MLIR

At first, we need to compile the MLIR from the latest branch of the LLVM project.

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project

mkdir build && cd build
cmake -G Ninja ../llvm \
  -DLLVM_ENABLE_PROJECTS=mlir \
  -DLLVM_BUILD_EXAMPLES=ON \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_RTTI=ON \
  -DLLVM_TARGETS_TO_BUILD="host"

cmake --build . --target check-mlir
```

Please make sure to use `-DCMAKE_BUILD_TYPE=Release`. We need to use the release version of the tools and binaries for Dialect development. You can find the official build instruction here.

Since it will take 10~20 minutes or more, let's deep dive into the Dialect codebase next time.

Thanks!

Reference

- The Deep Learning Compiler: A Comprehensive Survey
- zwang4/awesome-machine-learning-in-compilers
- CGO 2020 Talk
- Lewuathe/mlir-hello

No rule to make target for MacOSX SDK

22 Dec 2020

I am usually conservative for upgrading the libraries my machine depends on. I do not click the update button even the OS installer urges me to do so because it is likely to temporarily slow down the productivity to deal with the problem that occurred just after the upgrade. That's not fun.

But this time, I was careless. I accidentally approved upgrading the Xcode in my mac. The target version is 12.3. I believed there should have been no problem in just upgrading one build toolchain. But I was wrong.

No rule to make libcurses.

After I upgraded Xcode and install the toolchain alongside, my C++ project with CMake throws the following error.

```
No rule to make target `/Applications/Xcode.app/Contents/Developer/Pla
tforms/MacOSX.platform/Developer/SDKs/MacOSX11.0.sdk/usr/lib/libcurse
s.tbd'
```

Yes, `MacOSX11.0.sdk` is no more available, but my project still refers to the old SDKs unexpectedly. Some of the dependencies are correctly found under `MacOSX11.1.sdk`. Only the process to find libcurses failed.

I struggled to find the solution for hours and finally reached an unsophisticated answer.

Create Symlink!

Since some tools keep referring to the old SDK, let's create a link to the new one with the old name!

```
sudo ln -s \
  /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platfor
m/Developer/SDKs/MacOSX.sdk  \
  /Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platfor
m/Developer/SDKs/MacOSX11.0.sdk
```

It works. But that's a quite dirty way. I'm not sure how much impact it has on other software. If you have any ideas to resolve the situation better, please let me know.

Too Many Open Files with fluent-logger

11 Nov 2020

We tend to use fluentd (td-agent) to keep the access log of the service persistently. Fluentd is one of the most reliable and flexible middleware to collect various kinds of application logging. We can quickly broaden the target of logging by using many types of plugins maintained by the community.

As is often the case with the middleware libraries, we have found unexpected behavior of fluentd due to our lack of full knowledge of the library. Our Rails application often loses the application log!

This article will illustrate why that problem happens and caveats when we use fluent-logger in Rails application.

Problem

Our Rails application has a controller that collects its access log.

```
require 'fluent-logger'

class OurController < ApplicationController
```

```
  before_action :setup_fluent_logger
  before_action :access_log

  def setup_fluent_logger
    @log = Fluent::Logger::FluentLogger.new(
      nil,
      host: 'localhost',
      port: 24_224
    )
  end

  def access_log
    record = ...
    @log.post("test.access", record)
  end
end
```

Initially, I thought recreating the fluent-logger instance whenever it gets access is safe to dump the log. But that's not true at the end of the day. This controller fails to post logs when it gets many accesses from clients.

Cause

To detect the cause, I have tried to write a script to reproduce the issue.

```
require 'singleton'
require 'fluent-logger'

class OurController
  include Singleton

  def initialize
    @counter = 0
    setup_logger
  end

  def setup_logger
    @logger = Fluent::Logger::FluentLogger.new(
      nil,
      host: 'localhost',
      port: 24_224
    )
  end

  def post(v)
```

```
      @logger.post('debug.test', {
          a: @counter,
          v: v
      })
      @counter += 1
    end
  end


  puts "Start..."

  def post(i)
    controller = OurController.instance
    controller.setup_logger
    controller.post(i)
  end

  thread_num = 256

  # Multi Thread
  threads = (0..thread_num).map do |i|
    Thread.new {
      post(i)
    }
  end

  threads.each do |t|
    puts "Waiting for #{t}"
    t.join
  end
```

When I ran the program with `thread_num = 256`, it fails to dump the log due to connection failure to the local td-agent, but it did not with `thread_num = 200`. There is an implicit threshold between 200 and 256, I thought. Here is what I have found.

```
➜ launchctl limit
  cpu         unlimited      unlimited
  filesize    unlimited      unlimited
  data        unlimited      unlimited
  stack       8388608        67104768
  core        0              unlimited
  rss         unlimited      unlimited
  memlock     unlimited      unlimited
  maxproc     2784           4176
  maxfiles    256            unlimited
```

It hit the max number of file descriptors in the system. The `Too many open files` error was thrown by td-agent around the time.

```
2020-11-11 12:06:17 +0900 [warn]: #0 [forward_input] thread exited by
 unexpected error plugin=Fluent::Plugin::ForwardInput title=:event_loo
p error_class=Errno::EMFILE error="Too many open files - accept(2)"
#<Thread:0x00007fea449aa020@event_loop@/Users/sasaki/.rbenv/versions/
2.5.4/lib/ruby/gems/2.5.0/gems/fluentd-1.11.5/lib/fluent/plugin_helpe
r/thread.rb:70 run> terminated with exception (report_on_exception is
 true):
...
```

```
/Users/sasaki/.rbenv/versions/2.5.4/lib/ruby/2.5.0/socket.rb:1313:in `
__accept_nonblock': Too many open files - accept(2) (Errno::EMFILE)
```

Initializing the fluent-logger object opens a new file descriptor, which caused the `Too many open files` error. It looks like an event library Cool.io creates a file descriptor when it starts the event loop.

```
def start
  super
  # event loop does not run here, so mutex lock is not required
  thread_create :event_loop do
    begin
      default_watcher = DefaultWatcher.new
      event_loop_attach(default_watcher)
      @_event_loop_running = true
      @_event_loop.run(@_event_loop_run_timeout) # this method blocks
    ensure
      @_event_loop_running = false
    end
  end
end
```

Since we cannot have any privilege to modify the Cool.io codebase quickly, what we can do to the maximum is avoiding the recreation of fluent-logger instance. In this case, the `setup_fluent_logger` method looks as follows. Note that we use `||=` to initialize `@log` instead of `=` assignment.

```
class OurController < ApplicationController
  def setup_fluent_logger
    @log ||= Fluent::Logger::FluentLogger.new(
      nil,
      host: 'localhost',
      port: 24_224
    )
  end
end
```

In general, we cannot guarantee the order of the call of the method of `before_action`. We should have made `@log` thread-safe. I would stress that that

problem does not devote to fluentd or Rails itself. Our practice of programming causes that. I have gotten another lesson here.

Return std::make_unique from function?

29 Oct 2020

If you are an expert in modern C++, you must be familiar with the move semantics of C++. Move semantics provides us a chance to improve the performance by eliminating the unnecessary copy of the object, introduced in C++11. If your code contains a large object and sees the time when it's copied often, it's worth considering it.

Additionally, you may also be familiar with the smart pointer of C++. `std::unique_ptr` is a type of smart pointer. It prohibits the programmer from copying the pointer so that we can keep the ownership semantics clear. If we change the ownership of the object, move semantics will come into your scope. Generally, you can write the following code.

```
class A {}

std::unique_ptr<A> a1 = std::make_unique<A>();

std::unique_ptr<A> a2 = a1; // Compilation error

std::unique_ptr<A> a3 = std::move(a1); // OK
```

It clarifies **we are never able to copy the `std::unique_ptr`** .

Returning unique pointer from a function?

But I have found the following code passed the compile.

```
class A {}

std::unique_ptr<A> f() {
  return std::make_unique<A>();
}

std::unique_ptr<A> a = f();
```

Hmm, if I remember correctly, a function in C++ returns a copy of the returned object. A function `f` returns a copy of `std::unique_ptr` constructed by `std::make_unique` . How is it possible?

Copy Elision

This stackoverflow answers my question.

C++ seems to have a specification to define the case where we can omit the copy operation. According to that,

> When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object [...] This elision of copy/move operations, called copy elision, is permitted [...] in a return statement in a function with a class return type, when the expression is the name of a non-volatile automatic object with the same cv-unqualified type as the function return type [...]
>
> When the criteria for elision of a copy operation are met and the object to be copied is designated by an lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue.

So the compiler is allowed to omit the copy operation for this particular case. In short, **if the returned value is non-volatile (which is expected to have no side-effect) and local variable**, a compiler can safely omit the copy operation. This rule seems to be applied to the case returning `std::unique_ptr` .

To fully understand the detail of the rule, we need to dig deeper into the specification of C++. But I'll stop around here. The lesson I've got so far is **there is an official rule to allow us to return `std::unique_ptr` from function**. That's enough for a typical developer like me :)

Reference

- Copy Elision
- Returning unique_ptr from functions
- How does returning std::make_unique<SubClass> work?

Older | Newer