Getting Started with LLVM

January 11, 2018

Contents

1 Introduction

This document is intended to help users quickly set up LLVM. For a good introduction to LLVM refer to `http://www.cs.cornell.edu/~asampson/blog/llvm.html`

2 Prerequisites

Before you begin installing everything, ensure that these things are done first. For reference, all of this has been developed on Ubuntu 16.04 LTS.

- You must have access to the local LLVM git repository. Dr. Goeders can give you access.
- Git must be installed.
- To use Eclipse to edit LLVM projects, you need to install Eclipse and the CDT. Eclipse 4.6.0 is known to work.

3 Building LLVM

- Create a folder to store all the files in. `~/llvm` is recommended.
- Check out the project with `git clone https://[yourname]@bitbucket.org/jgoeders/llvm.git ~/llvm`. You may need to be in your home directory for everything to run properly.
- Set up the build environment: run `cmake ../llvm` from `~/llvm/build`.
- Run `make`. This may take quite a while, 30 minutes or more. You can try to parallelize the build across n processors with `-j`n to speed up the compilation significantly. The higher the number the faster the builds will take, but the more RAM will be used. Parallelizing across 7 cores can take over 16 GB of RAM. In case you run out of RAM the compilation can fail. In this case simply re-run `make` without any parallelization flags to finish the compilation.
- Add the following line at the end `.bashrc` file in your home directory.

  ```
  export PATH="/home/$USER/llvm/build/bin:$PATH"
  ```

  This will make it possible to use the `lli` command without typing the full path to the binary file every time. Note the path will be slightly different if `llvm` is not in the home directory.

4 Folder Structure

There are a number of folders in `~/llvm`. Here is a brief overview of what each of them contain.

1. `build` – LLVM places the generated files in this folder.
2. `docs` – This folder contains documentation.
3. `fiji` – Contains the Fault Injector JavaScript Interface to test passes on the MSP430. For more details see the appropriate user manual.
4. `llfi` – The LLFI fault injector simulator. This is no longer active.
5. `llvm` – The LLVM source code.
6. `projects` – Each LLVM pass we develop is placed in this folder.
7. `tests` – Tests for the passes are placed in this folder. This includes the CHStone self-checking benchmarks.

5 Compiling a Pass

This section contains instructions on how to compile and use your pass after you have written it. For help with that process, please look at `http://llvm.org/docs/WritingAnLLVMPass.html`.

1. To begin, create a subdirectory in the `projects` directory for your pass.
2. Add your *.cpp files here.
3. Copy a `CMakeLists.txt` file from one of the other project subfolders to your current one and modify a few lines. The top option, where it says `add_llvm_loadable_module`, should have the name of your new pass that you registered in the source code. The source code files should be listed underneath.
4. Open `~/llvm/projects/CMakeLists.txt` and look towards the bottom. You will see `add_subdirectory (dataflowProtection)`. Add an identical line to the end of the file, changing the "dataflowProtection" to match the folder name that you just created.
5. Look at line 8, where it says `set(LLVM_DIR ...` If the top most LLVM folder is not stored in the home directory, change the path to match your LLVM installation directory.
6. Navigate to the `~/llvm/projects/build` directory and run `cmake ..` (note the two periods). This command will create the appropriate makefiles for each subdirectory.
7. Run `make` in the `build` subdirectory. You should now see a new folder in the build directory with the same name as the subdirctory you made in the projects folder. Inside this folder you should have some cmake files and a .so file with your project name, which means it worked!

6 Test a Pass

Once you have your .so file compiled, you are ready to test your pass!

6.1 Using Your Own Test

You will most likely want to create your own tests

1. Navigate to the `tests` directory under your main LLVM folder.
2. If you didn't install in `~/llvm`, you need to change the `Makefile.common` to match your installation folder.
3. Create a subdirectory with your test files in it.
4. Copy a `Makefile` from another test directory and modify it as needed. The "schedule2" makefile is a good place to begin.
   - `LEVEL` – the location of `Makefile.common` relative to your test directory. Typically, `LEVEL = ..`
   - `TARGET` – your test file name without the .c extension
   - `OPTS` – all of the LLVM optimizations you want to apply. One of the options should be the pass you want to test on it.
   - `OPT_LIBS` – should point to the .so file you created when you compiled your pass
5. Run `make` on your folder.

You will now have two .ll files in your folder. The *.clang.ll is the default IR before the optimizations, and the *.opt.ll is the IR after your optimizations have been run. If you need to look at any of the intermediate files you can find the commands and options in the `Makefile.common` file in the `tests` root directory.

6.2 Using the CHStone Tests

The 12 CHStone benchmarks are used for HLS testing. However, we use them because they contain a fair selection of use cases. Additionally, they are self checking. That allows you to test your pass and see if it provides functional equivalence. Here's how to use the automated test:

1. Navigate to `tests/chstone` in your LLVM folder.
2. Modify the `Makefile.common` to include your passes eg. `OPT_PASSES = -ExitMarker -TMR -CFCSS`. All user passes under `projects` are automatically loaded.
3. Run `make` to compile all programs, or `make run` to compile and run the programs.

The makefile will iterate through all programs in sequence and compile them. If there is an error it will halt. If `make run` is executed, it will then run all of the benchmarks. Each one will execute, then compare the result to the known answer. It will output the results of the comparison and if the program passed. The script halts if any test fails.

7 LLVM and Eclipse

You can write your code in a plain text editor, or you can use Eclipse to help you manage all of the classes and methods. This guide was written for Eclipse Neon using the CDT.

7.1 Setting Up the Project

1. Select `File` → `New` → `Makefile Project with Existing Code`.
2. Enter `llvm` as the project name.
3. For the existing code location field, browse to the LLVM root directory
4. Use the "Linux GCC" toolchain. It will take several minutes for the indexer to process everything into your project so you will get a large number of false errors in your source code.
5. Right click on your "llvm" project directory and select "Properties"
6. Navigate to "C/C++ Build" and change the build directory to your `llvm/build` folder using the "File system" button.
7. Change to the "Behavior" tab and enable parallel builds. We recommend using 3-4 parallel jobs.
8. Click "Apply" then "OK".
9. When you click on the "Build" button LLVM will be compiled.

7.2 Using Eclipse To Build Your Passes

1. Right click on the `projects/build` subdirectory, then `Make Targets` → `Create`.
2. Call the target name `all` and click OK.
3. To build your pass, right click on the build folder and click `Make Targets` → `Build` → `Build` (with the target `all` selected).
4. After the first time that you've done this, you can rebuild all your passes by pressing F9.

8 What to Push to the Git Repository

In order to keep the git repository clean, there are only a few files that you need to commit. The remainder can be easily generated from the committed files. Here are the files that you should commit.

8.1 New Passes

- `~/llvm/projects/CMakeLists.txt` if you modified it.
- The source (*.cpp, *.h) files in your project subdirectory.
- The `CMakeLists.txt` in your project subdirectory.

8.2 New Tests

Your new tests should likewise contain only your source code and Makefile in your subdirectory.

9 Adding Command Line Arguments to the Backend

In rare cases you might want to change the arguments that a specific backend takes. However, there isn't much documentation out there for this. This section is what worked for me, not a definitive guide. I'll use the MSP430 as my example, replace it with the backend target that you need.

1. Look in `MSP430.td`. There is a section labelled "Subtarget Features". Follow the convention there to create your command line argument. The first argument is what is enabled/disabled on the command line. The second is a variable that the backend will refer to internally. The third is most likely the default value. The last argument is what is displayed in the backend help message.
2. Go a few lines lower to the "Supported processors" section. In the square brackets you should choose what processor models should have your argument enabled by default.
3. Open `MSP430Subtarget.h`. In the MSP430Subtarget definition you should add in a boolean class member. The name of this variable should be the second argument from step 1.
4. You will probably need to create a getter function. Place it in the public section of the class definition.

You are now able to access the variable in any function related to a subtarget. To pass in arguments over the command line either use `-mcpu=$SUBTARGET` to enable or disable a standard set of optimizations, as specified in step 2. Otherwise, you can do `-mattr=+$OPTION`, where `$OPTION` is the first argument used in step 1.