FreeSandal

樹莓派, 樹莓派之學習, 樹莓派之教育

【鼎革．革鼎】：RASPBIAN STRETCH《六之 J.3下》

2017-12-13 | 懸鉤子 | 發表迴響

查詢所謂

· LLVM ERROR: Program used external function '__aeabi_unwind_cpp_pr0' which could not be resolved!

錯誤訊息，知道恐和系統之 gcc 版本與或編譯旗號有關哩？

後讀

import librosa gives LLVM ERROR

	Goutham Kamath

I am trying to install librosa inside a docker on armv7 Raspberry pi. Here are the steps I followed:

```
apt-get update
apt-get install cython build-essential libedit-dev
apt-get install llvm-4.0 llvm-4.0-dev llvm-dev


LLVM_CONFIG=/usr/lib/llvm-4.0/bin/llvm-config pip install llvmlite==0.19.0

LLVM_CONFIG=/usr/lib/llvm-4.0/bin/llvm-config pip install numba==0.32

pip install librosa==0.5.1
```

```
11
12
13 I am able to successfully install without errors and also can import llvmlite and numba
14
15 >> import librosa
16 LLVM ERROR: Program used external function '__aeabi_unwind_cpp_pr0' which could not be
17
18
19 I am running Python 2.7.12 with GCC 5.4.0 inside docker with ubuntu-14.04 base.
```

	Brian McFee

Yikes! It looks like an abi incompatibility issue that you might want to raise with the numba or llvmlite folks. We're not doing anything fancy with numba, so it's almost certainly deeper in the stack than librosa.

A couple of things to try:

- Upgrade numba and llvmlite to more recent versions (if availably on rasp)
- If nothing works, you can back off resampy to the older 0.1.x series. It's functionally and API-equivalent, but built on cython instead of numba. It's not a long-term fix, but it should unblock you until the numba issue gets resolved upstream.

也曾多方嘗試，終究無功而返！

除非出海者，亦想自己造船？！

The **LLVM** Compiler Infrastructure

LLVM Overview

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-Style license.

The primary sub-projects of LLVM are:

1. The **LLVM Core** libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.
2. **Clang** is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles (e.g. about 3x faster than GCC when compiling Objective-C code in a debug configuration), extremely useful error and warning messages and to provide a platform for building great source level tools. The Clang Static Analyzer is a tool that automatically finds bugs in your code, and is a great example of the sort of tool that can be built using the Clang frontend as a library to parse C/C++ code.
3. The **LLDB** project builds on libraries provided by LLVM and Clang to provide a great native debugger. It uses the Clang ASTs and expression parser, LLVM JIT, LLVM disassembler, etc so that it provides an experience that "just works". It is also blazing fast and much more memory efficient than GDB at loading symbols.
4. The **libc++** and **libc++ ABI** projects provide a standard conformant and high-performance implementation of the C++ Standard Library, including full support for C++11.
5. The **compiler-rt** project provides highly tuned implementations of the low-level code generator support routines like "`__fixunsdfdi`" and other calls generated when a target doesn't have a short sequence of native instructions to implement a core IR operation. It also provides implementations of run-time libraries for dynamic testing tools such as AddressSanitizer, ThreadSanitizer, MemorySanitizer, and DataFlowSanitizer.
6. The **OpenMP** subproject provides an OpenMP runtime for use with the OpenMP implementation in Clang.
7. The **polly** project implements a suite of cache-locality optimizations as well as

auto-parallelism and vectorization using a polyhedral model.
8. The **libclc** project aims to implement the OpenCL standard library.
9. The **klee** project implements a “symbolic virtual machine” which uses a theorem prover to try to evaluate all dynamic paths through a program in an effort to find bugs and to prove properties of functions. A major feature of klee is that it can produce a testcase in the event that it detects a bug.
10. The **SAFECode** project is a memory safety compiler for C/C++ programs. It instruments code with run-time checks to detect memory safety errors (e.g., buffer overflows) at run-time. It can be used to protect software from security attacks and can also be used as a memory safety error debugging tool like Valgrind.
11. The **lld** project aims to be the built-in linker for clang/llvm. Currently, clang must invoke the system linker to produce executables.

In addition to official subprojects of LLVM, there are a broad variety of other projects that use components of LLVM for various tasks. Through these external projects you can use LLVM to compile Ruby, Python, Haskell, Java, D, PHP, Pure, Lua, and a number of other languages. A major strength of LLVM is its versatility, flexibility, and reusability, which is why it is being used for such a wide variety of different tasks: everything from doing light-weight JIT compiles of embedded languages like Lua to compiling Fortran code for massive super computers.

As much as everything else, LLVM has a broad and friendly community of people who are interested in building great low-level tools. If you are interested in getting involved, a good first place is to skim the LLVM Blog and to sign up for the LLVM Developer mailing list. For information on how to send in a patch, get commit access, and copyright and license topics, please see the LLVM Developer Policy.

眼下恐無善法的呦★

llvmlite

A lightweight LLVM python binding for writing JIT compilers

The old llvmpy binding exposes a lot of LLVM APIs but the mapping of C++-style memory management to Python is error prone. Numba and many JIT compilers do not need a full LLVM

API. Only the IR builder, optimizer, and JIT compiler APIs are necessary.

llvmlite is a project originally tailored for Numba's needs, using the following approach:

- A small C wrapper around the parts of the LLVM C++ API we need that are not already exposed by the LLVM C API.
- A ctypes Python wrapper around the C API.
- A pure Python implementation of the subset of the LLVM IR builder that we need for Numba.

Numba

A compiler for Python array and numerical functions

Numba is an Open Source NumPy-aware optimizing compiler for Python sponsored by Anaconda, Inc. It uses the remarkable LLVM compiler infrastructure to compile Python syntax to machine code.

It is aware of NumPy arrays as typed memory regions and so can speed-up code using NumPy arrays. Other, less well-typed code will be translated to Python C-API calls effectively removing the "interpreter" but not removing the dynamic indirection.

Numba is also not a tracing JIT. It *compiles* your code before it gets run either using run-time type information or type information you provide in the decorator.

Numba is a mechanism for producing machine code from Python syntax and typed data structures such as those that exist in NumPy.

Dependencies

- llvmlite
- numpy (version 1.9 or higher)
- funcsigs (for Python 2)