Tips for Using CMake and GNU Autotools on Cray Heterogeneous Systems

Introduction

CMake and the GNU Autotools are tools that help manage the application build process. They are designed to increase the portability of source code packages by generating configurations for native makefiles based on input from users and system introspection. These build tools work well on simple, homogenous systems, but require additional configuration for the heterogeneous mix of processors, accelerators, compilers, and libraries that are present on modern HPC systems. This document describes conventions and methods that can be used to avoid or address some of the more frequently encountered issues when building on a heterogeneous Cray system. It assumes basic familiarity with CMake and Autotools.

General Processor Targeting Conventions

Cray's programming environment includes processor-targeting modules, compiler drivers, and libraries for building applications on login or Cray Development and Login (CDL) nodes that will be subsequently executed on compute nodes. To compile code optimized for the compute node on a Cray system, it is important to first load the desired processor-targeting module (`craype-haswell`, `craype-mic-knl`, etc.) and the desired `PrgEnv-*` compiling environment (Cray, Intel, GNU, etc.). These targeting modules configure the compiler driver scripts (`cc`, `CC`, `ftn`) to compile code optimized for the processor.

Default Processor Targets

A default set of CPU- network-, and accelerator-type modules are typically loaded on a system to target building for the compute nodes. For example, a Cray XC30 system with Sandybridge compute nodes should have a default environment that loads the `craype-network-aries` and `craype-sandybridge` modules. It is always important to check which modules are loaded prior to building an application, especially on a system with heterogeneous compute nodes. The list of loaded modules can be obtained by running the '`module list`' command.

Portable Builds

If no default targeting modules are loaded in a user's environment, the compiler driver scripts will generate un-optimized, but portable binaries. Because users most often want high-performance code produced for execution on compute nodes, a warning will be emitted. This warning can be suppressed by taking the following actions to explicitly indicate that portable code is intended.

```
$ export CRAY_CPU_TARGET=x86_64
$ module swap craype-network-aries craype-network-none
```

To build an application that runs on a login or CDL node, make sure the `craype-network-none` module is loaded. `craype-network-none` assures that no network libraries are loaded and that all network library dependencies are ignored.

CMake

Beginning with version 3.5.0, CMake has been enhanced to work with the Cray Programming Environment.

Cross Compiling

Cross compiling can be enabled for compute nodes on a Cray system by setting CMAKE_SYSTEM_NAME to CrayLinuxEnvironment as in the following example:

```
$ cmake -DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment ...
```

This will ensure that the appropriate build settings and search paths are configured. The platform will pull its configuration from the current environment variables and will configure a project to use the compiler wrappers from the Cray Programming Environment's `PrgEnv-*` modules if loaded.

CMake however is not able to confirm that it is running in a Cray environment on systems running CLE 6 due to a change in the Cray Programming Environment. When this occurs, the following error is issued:

```
Neither the CRAYOS_VERSION or XTOS_VERSION environment variables are
defined. This platform file should be used inside the Cray Linux
Environment for targeting compute nodes (NIDs)
```

Cray is working with Kitware to update CMake to address the issue. Until this issue is resolved, setting the CRAYOS_VERSION environment variable as shown below prior to running CMake can be used to work around the problem.

```
$ export CRAYOS_VERSION=6
```

Multi-component Builds

When building a multi-component application intended to run on a Cray system with heterogeneous login and compute nodes, the application should be built in stages, changing the processor target appropriately to build for the different processor types. For example, if you want to build Python, and use it to build part of an application, build Python for the login node first. Then point CMake to this version of Python, reset your processor and network targeting modules to direct the next stage of the build for the compute node, and continue the build. Remember to pass `-DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment` to CMake to enable cross-compiling.

Using CMake's `try-run` command

The `try_run` command is used to compile and run a test to determine system attributes or software capability. Using `try_run` when cross-compiling will generate an error similar to the one in the following example:

```
CMake Error: TRY_RUN() invoked in cross-compiling mode, please set the
following cache variables appropriately: ...
For details see /path/to/build/TryRunResults.cmake
```

This error requires manual intervention by the user. Edit the file, `TryRunResults.cmake`, that is created by CMake and replace the placeholder values:

```
set( run_result
      "PLEASE_FILL_OUT-FAILED_TO_RUN"
      CACHE STRING "Result from TRY_RUN" FORCE)

set( run_result__TRYRUN_OUTPUT
      "PLEASE_FILL_OUT-NOTFOUND"
      CACHE STRING "Output from TRY_RUN" FORCE)
```

Copy the `TryRunResults.cmake` file out of your build directory so it won't be deleted when you rebuild.

Use the `TryRunResults.cmake` file as an additional cache file when rebuilding:

```
$ cmake -C /path/to/TryRunResults.cmake ...
```

Using an Emulator

The `CMAKE_CROSSCOMPILING_EMULATOR` property is available starting with CMake 3.6. This property can be used instead of manually modifying the CMake's `TryRunResults.cmake` file when trying to run a binary targeted for a Cray compute node that differs from the processor on the login node. If this property is set, CMake will try to run certain commands by prefixing an emulator. An emulator can be anything from a specialized hardware emulator to a simple shell script encapsulating the use of a workload manager to execute a binary on a compute node.

The following is an example script using SLURM:

```
#!/usr/bin/env bash
srun --ntasks 1 --partition workq $@
exit $?
```

To use the above script, point CMAKE_CROSSCOMPILING_EMULATOR to the script:

```
$ cmake -DCMAKE_SYSTEM_NAME=CrayLinuxEnvironment \
-DCMAKE_CROSSCOMPILING_EMULATOR=/path/to/script...
```

Starting with CMake 3.6, several commands have the ability to use an emulator when cross-compiling.

```
try_run
add_test
add_custom_command
add_custom_target
```

Getting Help

For more information on cross-compiling with CMake on a Cray system, please visit:
https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html#cross-compiling-for-the-cray-linux-environment

For information on CMake's try_run behavior when cross-compiling, please visit:
https://cmake.org/cmake/help/latest/command/try_run.html#behavior-when-cross-compiling

CMake 3.5 release notes:
https://cmake.org/cmake/help/v3.5/release/3.5.html

CMake 3.6 Release Notes:
https://cmake.org/cmake/help/v3.6/release/3.6.html

How to Build CMake

Cray does not distribute CMake. The following demonstrates how to build CMake 3.5.1 that was downloaded from https://cmake.org/. Note that the CC and CXX environment variables were intentionally not set, so the build process defaults to directly calling the `gcc` and `g++` compilers.

```
$ tar xzvf cmake-3.5.1.tar.gz
$ cd cmake-3.5.1
$ module load gcc # load a current version of GCC
$ export PE_INSTALL=<Installation directory for PE tools>
$ mkdir -p $PE_INSTALL/cmake/3.5.1
$ ./configure --prefix=$PE_INSTALL/cmake/3.5.1
$ gmake install
```

A modulefile can be created for CMake by using the `craypkg-gen` command as in the following example.

```
$ module load craypkg-gen
$ craypkg-gen -m $PE_INSTALL/cmake/3.5.1
```

The resulting modulefile is placed in the file `$PE_INSTALL/modulefiles/cmake/3.5.1`. It is recommended to edit the modulefile and change the line "`append-path PATH ....`" to "`prepend-path PATH .....`". This will ensure that if `/usr/bin/cmake` exists, it will not be used when the `cmake` modulefile is loaded. Here is an example of using the `cmake` modulefile:

```
$ module use $PE_INSTALL/modulefiles
$ module load cmake
$ cmake –version

cmake version 3.5.1

CMake suite maintained and supported by
Kitware(kitware.com/cmake).
```

To create an RPM of the resulting CMake package, use the `craypkg-gen` utility as demonstrated in the following example.

```
$ export CLE_PE_INSTALL=<System PE tools directory>
$ craypkg-gen -r $PE_INSTALL/cmake/3.5.1 \
     --prefix=$CLE_PE_INSTALL
```

The resulting RPM that is created will be called `craypkg-cmake-3.5.1-0.x86_64.rpm`.

GNU Autotools / configure

The GNU Autotools-generated script, `configure`, determines system attributes and then sets up a source package for compilation. There are some general conventions that can be followed to decrease the likelihood of encountering a configuration error, to relocate the package's binaries, or to easily compile for different Cray targets.

Using the Compiler Drivers

Using the Cray compiler drivers allows the packager to swap different CPU, PrgEnv, or compiler modulefiles to easily target their respective environments. To tell `configure` to use the compiler drivers, set the 'CC', 'FTN', and 'CXX' variables as in the following two examples.

```
$ module load PrgEnv-cray
$ ./configure CC=cc FTN=ftn CXX=CC ...

$ module swap PrgEnv-cray PrgEnv-intel
$ ./configure CC=cc FTN=ftn CXX=CC ...
```

Building Multiple Packages for Multiple Platforms at Once

`configure` will create all files that it needs in a sub-directory. This can be harnessed to do multiple

builds easily in a single source directory as shown in the following example.

```
for CPU_TARGET in "ivybridge" "mic-knl" ; do
       module load cray-${CPU_TARGET}
       mkdir ${CPU_TARGET}
       pushd ${CPU_TARGET}
       ../configure CC=cc FTN=ftn CXX=CC ...
       make
       popd
       module unload cray-${CPU_TARGET}
done
```

Relocating Binaries

If the final libraries or binaries will be relocated, the `--prefix` can be set to a privileged final location, and DESTDIR can be passed to Make with the user-writable location. Any hard-coded paths will use the prefix but the libraries can be installed into DESTDIR. The following example demonstrates how to use `--prefix` and `DESTDIR`

```
$ ./configure --prefix=/opt/product
$ ./make install DESTDIR=/tmp/build_area
```

Dynamic Linking

Some packager-written `configure` tests assume dynamic linking, and some pitfalls can be avoided if builders also use dynamic linking by setting the environment variable, `CRAYPE_LINK_TYPE,` to dynamic:

```
$ export CRAYPE_LINK_TYPE=dynamic
```

Cross-compiling with Autotools

`configure` checks which build options will work on a system by performing test compiles and executions. This can be problematic if the login or CDL node is unable to execute the tests, because the builder is targeting a different Instruction Set Architecture (ISA) than is supported on the build machine. To address this, `configure` can be put into cross-compile by setting `configure's --host` or `cross-compiling` flags as in the following examples.

```
$ ./configure cross_compiling=yes
```

or

```
$ ./configure --host= x86_64-unknown-linux-gnu  #cpu-vendor-os
```

However, some Autotools projects do not support cross-compiling. If the compute node and login node processors are compatible (ie. Sandybridge and Ivybridge), another option is to try running configure

while targeting the login node processor, allowing configure's execution tests to pass, and then switch to targeting the compute node processor before compiling. This avoids putting configure into cross-compiling mode when it isn't supported.

```
$ module load craype-sandybridge
$ ./configure CC=cc FTN=ftn CXX=CC ...
$ module swap craype-sandybridge craype-mic-knl
$ make
```

Running configure with CCE

Since CCE is designed and optimized for HPC systems, it gets little exposure in more general Linux environments where GNU Autotools projects are typically tested. Interoperability between autoconf and libtool and the Cray Compiling Environment (CCE) has been recently improved, and Cray has submitted a patch to libtool that is waiting inclusion. The patch fixes shared library issues such as:

- Failure to detect OpenMP flag (`--homp`)
- Failure to detect pass to linker option (`-Wl`)
- Failure to detect PIC option (`--hpic`)
- Failure to detect link-type opts (`--dynamic/--static`)
- Failure to generate archive creation command

Due to these problems it is recommended that users compiling with CCE download and build the latest `autoconf`, then download and patch the latest `libtool`. With the updated tools the user should replace the packages `ltmain.sh` and rerun `autoreconf -i`. Some projects erroneously detect CCE as a GCC compiler. If the `libtool` patch is not present, users can override this faulty test by giving the correct answer, `ac_cv_c_compiler_gnu=no`, directly to `configure`.

Cray has made the patched upstream `libtool git` repository available at the following location:

https://github.com/Cray/libtool

The URL references a mirror of the current upstream `libtool` repository, with the CCE patch applied. To use, follow the same steps you normally would for building upstream `libtool` from a `git` clone.

```
$ git clone https://github.com/Cray/libtool.git
$ cd libtool
$ ./bootstrap && ./configure && make && make install
```