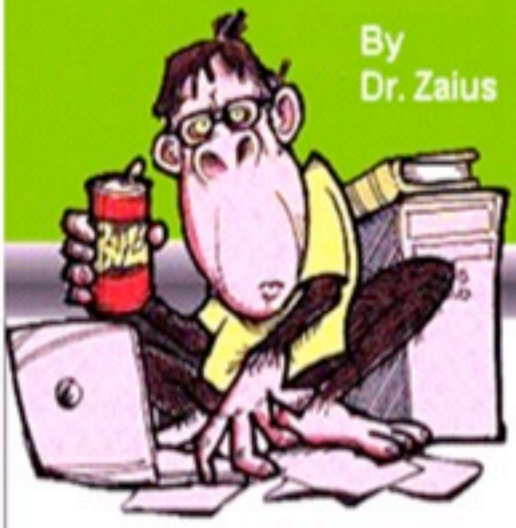
ClusterChimps.org

CUDA

Autotools

By
Dr. Zaius



The ClusterChimps Guide to integrating

CUDA and GNU Autotools

This guide is intended to help developers who are familiar with the GNU build system to incorporate applications written in Nvidia's "C for CUDA" into their Autotools builds. We will briefly describe the changes necessary to each file and provide you with an example tarball that illustrates building stand alone CUDA based applications as well as CUDA shared and static libraries. This guide is NOT intended to be useful to people completely unfamiliar with the GNU build system.

About ClusterChimps.org

ClusterChimps is dedicated to helping bring inexpensive supercomputing to the masses by leveraging emerging technologies coupled with bright ideas and open source software. We do this because we believe it will help advance computation intensive research areas including basic research, engineering, earth science, biology, materials science, and alternative energy research just to name a few.

About the Author

Dr. Zaius is a well renowned orangutan in the field of cluster computing and GPGPU programming. He has spent most of his career in the financial industry working at exchanges, investment banks, and hedge funds. He is currently the driving force behind the site ClusterChimps.org. Originally from the island of Borneo, Dr. Zaius now resides in New York City with his wife and 3 children. He can be reached at zaius@clusterchimps.org

Table of Contents

Chapter

1

Introduction

According to Wikipedia, "The **GNU build system**, also known as the **Autotools**, is a suite of programming tools designed to assist in making source-code packages portable to many Unix-like systems. It can be difficult to make a software program portable: the C compiler differs from system to system; certain library functions are missing on some systems; header files may have different names. One way to handle this is write conditional code, with code blocks selected by means of preprocessor directives (#ifdef); but because of the wide variety of build environments this approach quickly becomes unmanageable. The GNU build system is designed to address this problem more manageably.

The GNU build system is part of the GNU toolchain and is widely used in many free-software and open-source packages. The tools comprising the GNU build system are free-software-licensed under the GNU General Public License with special license exceptions permitting use of the GNU build system with proprietary software."

If you are building Unix / Linux based software and you want your software to be easily ported to different flavors of Unix / Linux then Autotools *may* be for you. I say "may" be for you because Autotools only provides native support for C, C++, Objective C, and Fortran. If you are using one of these languages and the portability of your software is important to you then Autotools *is* for you. If you don't care about portability then Autotools might be overkill but you never know when you are going to need to care about portability. How many times have you been asked to build something for a single platform and six months later been told that it has to run on three different platforms? Wouldn't you rather be well prepared for the future than have to scramble at the last minute?

Autotools

Autotools is comprised of several different tools: Autoconf, Automake, and Libtool. The Autoconf tool is actually a suite of tools (Autoheader, Autom4te, Autoreconf, Autoscan, Autoupdate, ifnames) whose primary task is to generate a Bourne shell configuration script. Autoconf is designed in a manner that the configuration script it generates should be able to run on any Unix-like platform even if Autotools is not installed on that platform. This makes it particularly useful as a build system for source code packages that are intended for widespread distribution.

Automake

Writing a makefile is not that complicated, however, makefile users tend to expect a certain level of functionality from a makefile. For a makefile to be considered complete developers expect certain make targets to be defined (all, clean, install, uninstall, etc...) as well as certain features to be configurable (multi-threaded | single-threaded, shared-libs | static-libs, etc...). Building all of this by hand in a portable manner is tedious and error prone. That's where Automake comes in. Automake generates standard makefile templates (Makefile.in) from high-level specification files (Makefile.am). While Automake expects the Makefile.am to be written in a higher-level abstraction you can put regular makefile syntax in the Makefile.am as well. This feature allows you to extend the functionality of the Automake program. We will make use of this feature to add support for CUDA.

Libtool

A library is a software module that performs some service to users through an API. Libraries come in two flavors: shared and static. Shared libraries differ from static libraries in that the object code in a shared library is not linked into the executable. When a binary is linked against a shared library all that is linked into the binary is the place holder to the shared library. Another difference between shared and static libraries is that at runtime multiple binaries can share the same code pages that a shared library occupies in RAM (hence the name "shared"). Shared libraries provide three main benefits. They reduce the amount of RAM consumed on a running system, they reduce the amount of disk space consumed by binaries, and they provide a more flexible software update process.

Building shared libraries is one area where Autotools really comes through for its users. While most systems have somewhat standardized on shared library implementation details, there is not much standardization around the building, naming, and management. Libtool provides a

set of Autoconf macros that hide library naming differences in makefiles and provides an optional dynamic loader for your shared libraries.

Building + Installing

Building an Autotools managed source code package is very simple and straight forward. Your users download your compressed tarball, uncompress it, run configure, and type make install. That's it! When you run configure you have the ability to fine tune your build and the configuration script takes care of platform-specific configurations for you. For example:

```
zaius> gunzip cuda-example-1.0.tar.gz
zaius> tar –xvf cuda-example-1.0.tar
zaius> cd cuda-example-1.0
zaius> configure
zaius> make all
zaius> make install
```

The commands above are all you need to do to build and install our Autotools CUDA example source code package described below.

A Simple Example

As part of this guide we have put together a simple Autotools based example to illustrate how to enable CUDA support for your Autotools builds. The example contains CUDA based targets and is named cuda-example-0.0.0.tar.gz. Our example contains a stand alone CUDA binary (named binaryCuda), a binary that links in a shared library that contains CUDA code (named binaryCudaShLib), and a binary that links in a static library that contains CUDA code (named binaryCudaStLib). Each binary is logically identical. They all generate test data and then loop through calling a CUDA kernel. The difference in the binaries is in their structure. The stand alone CUDA binary has the source for the CUDA kernel imbedded the same compilation unit as the programs main. The binary that links in a shared library is built from two separate targets. The first target is a shared library that contains the CUDA kernel and the second target contains the main program that generates the test data and makes calls to the shared library. The binary that links in a static library is identical in structure to the shared library example with the only difference being that the library is static.

The kernel that we are using for this example is a simple binomial options pricing model that was borrowed from Nvidia's CUDA examples. We're not going to get into the source for the

example code because it's really irrelevant to our discussion, however, we will walk through the changes that we needed to make to a standard Autotools build to enable CUDA support.

Chapter

2

Adding CUDA Support

The example tarball can be downloaded from http://www.clusterchimps.org/autotools.php. Once you have downloaded the tarball uncompress it, untar it and take a look at the contents.

```
zaius> gunzip cuda-example-0.0.0.tar.gz
zaius> tar -xvf cuda-example-0.0.0.tar
zaius> cd cuda-example-0.0.0
zaius> ls

AUTHORS      Makefile.in  config.guess  cudalt.py   src
COPYING      NEWS         config.sub    depcomp
ChangeLog    README       configure     install-sh
INSTALL      aclocal.m4   configure.ac  ltmain.sh
Makefile.am  compile      cuda.mk       missing
```

The contents of the directory should look familiar to you. There are two files that are not part of a standard Autotools build: cuda.mk and cudalt.py. These are new files that we added to enable CUDA support. We also made changes to the configure.ac file and to the target Makefile.am files that are in the src directory. In this chapter we will walk you through what changes we made and why.

configure.ac

The configure.ac file is what Autoconf uses to generate your projects configure script. One of the functions that the configure script provides is the ability for you to fine tune the dependencies of the project. If we are building CUDA based applications our project is going to have a dependency on the CUDA libraries isn't it? By making use of the AC_ARG_WITH macro we can provide a default location for our project to find CUDA and allow our users to override the default location if they chose to install CUDA to a non-standard location. Below we take a look at the contents of the configure.ac file.

configure.ac

```
AC_PREREQ([2.59])
AC_INIT([cuda-example], [0.0.0], [zaius@clusterchimps.org])
AM_INIT_AUTOMAKE

AC_PROG_CC
AM_PROG_CC_C_O
AC_PROG_LIBTOOL

AC_CONFIG_FILES([Makefile
                 src/Makefile
                 src/shared-library/Makefile
                 src/shared-library/lib/Makefile
                 src/shared-library/main/Makefile
                 src/static-library/Makefile
                 src/static-library/lib/Makefile
                 src/static-library/main/Makefile
                 src/binary/Makefile
                ])

#find out what architecture we are
ARCH=`uname -m`
if [[ $ARCH == "x86_64" ]];
then
  SUFFIX="64"
else
  SUFFIX=""
fi
```

```
# Setup CUDA paths
AC_ARG_WITH([cuda],
   [  --with-cuda=PATH    prefix where cuda is installed
                          [default=/usr/local/cuda]])


if test -n "$with_cuda"
then
   CUDA_CFLAGS="-I$with_cuda/include"
   CUDA_LIBS="-lcudart"
   CUDA_LDFLAGS="-L$with_cuda/lib$SUFFIX"
   NVCC="$with_cuda/bin/nvcc"
else
   CUDA_CFLAGS="-I/usr/local/cuda/include"
   CUDA_LIBS="-lcudart"
   CUDA_LDFLAGS="-L/usr/local/cuda/lib$SUFFIX"
   NVCC="nvcc"
fi

AC_SUBST(CUDA_CFLAGS)
AC_SUBST(CUDA_LIBS)
AC_SUBST(CUDA_LDFLAGS)
AC_SUBST(NVCC)

AC_OUTPUT
```

There is nothing special with regards to what we have put in our configure.ac file. We set up a default location for CUDA as well as default values for libraries, linker flags, and the CUDA compiler. We also allow the user to override the location of CUDA.

cuda.mk

As we mentioned earlier Autotools provides native support for C, C++, Objective C, and Fortran. You will notice that CUDA is not in that list. Somehow we need to tell Autotools (more specifically Automake) how to deal with .cu files. We mentioned earlier that Automake not only supports a high-level abstraction language but it also supports standard make syntax. We can make use of that fact and teach Automake how to handle files with a .cu extension. Below we take a look at the contents of cuda.mk.

cuda.mk

```
.cu.o:
    $(NVCC) -gencode=arch=compute_13,code=sm_13 -o $@ -c $<

.cu.lo:
    $(top_srcdir)/cudalt.py $@ $(NVCC) \
    -gencode=arch=compute_13,code=sm_13 \
    --compiler-options=\"$(CFLAGS) \
    $(DEFAULT_INCLUDES) $(INCLUDES) \
    $(AM_CPPFLAGS) $(CPPFLAGS) \" -c $<
```

The first rule in this file tells Automake how to build a stand alone CUDA binary (how to generate a .o out of a .cu file). The second rule in the file tells Automake how to generate a .lo out of a .cu file. A .lo file is what is generated by libtool to support shared libraries. The contents of the .cu.o rule should look familiar to you. It is basically just the flags that you would pass to nvcc to compile a .cu file (recall that we defined $(NVCC) in our configure.ac file). The contents of the .cu.lo rule are again just what we would pass to nvcc to create a shared library with the exception of cudalt.py at the beginning of the rule. The cudalt.py file is a python file that acts as a libtool script for CUDA which we will cover shortly.

Why are we placing these rules in a file at the top-level of our project? Since we want all of our CUDA based targets in the entire project to use the same rules we just put them in a file at the top-level and we include that file in all of our Makefile.am files that are building CUDA targets. This way we can change the nvcc flags for all of our targets just by updating this one file. What if you don't want the same nvcc flags on all of the targets in your project? In that case just don't include this file in your targets Makefile.am and put the .cu.o or .cu.lo rules directly in your targets Makefile.am.

cudalt.py

When libtool is used to generate a shared library it does a bit more than just compile your source. When you type make libtool also produces a .lo file. This file is used by the Automake system when linking project libraries with binaries. Since libtool doesn't know anything about files with a .cu extension we need to have some way of generating our .lo file and compiling our source. That is were the second rule in our cuda.mk file comes into play. It will call cudalt.py to compile our .cu file and to generate the necessary .lo file.

cudalt.py

```
#!/usr/bin/python
# libtoolish hack: compile a .cu file like libtool does
import sys
import os

lo_filepath = sys.argv[1]
o_filepath = lo_filepath.replace(".lo", ".o")

try:
   i = o_filepath.rindex("/")
   lo_dir = o_filepath[0:i+1]
   o_filename = o_filepath[i+1:]

except ValueError:
   lo_dir = ""
   o_filename = o_filepath


local_pic_dir = ".libs/"
local_npic_dir = ""
pic_dir = lo_dir + local_pic_dir
npic_dir = lo_dir + local_npic_dir

pic_filepath = pic_dir + o_filename
npic_filepath = npic_dir + o_filename
local_pic_filepath = local_pic_dir + o_filename
local_npic_filepath = local_npic_dir + o_filename
```

```
# Make lib dir
try:
   os.mkdir(pic_dir)
except OSError:
   pass

# generate the command to compile the .cu for shared library
args = sys.argv[2:]
args.extend(["-Xcompiler","-fPIC"]) # position indep code
args.append("-o")
args.append(pic_filepath)
command = " ".join(args)
print command

# compile the .cu
rv = os.system(command)
if rv != 0:
    sys.exit(1)

# generate the command to compile the .cu for static library
args = sys.argv[2:]
args.append("-o")
args.append(npic_filepath)
command = " ".join(args)
print command

# compile the .cu
rv = os.system(command)
if rv != 0:
    sys.exit(1)

# get libtool version
fd = os.popen("libtool --version")
libtool_version = fd.readline()
# this loop supresses the broken pipe errors
# you get by not reading all the data
for dog in fd.readlines():
    noop = 1;
fd.close()
```

```
# generate the .lo file
f = open(lo_filepath, "w")
f.write("# " +  lo_filepath + " - a libtool object file\n")
f.write("# Generated by " + libtool_version + "\n")
f.write("#\n")
f.write("# Please DO NOT delete this file!\n")
f.write("# It is necessary for linking the library.\n\n")

f.write("# Name of the PIC object.\n")
f.write("pic_object='" + local_pic_filepath + "'\n\n")

f.write("# Name of the non-PIC object.\n")
f.write("non_pic_object='" + local_npic_filepath + "'\n")
f.close()

sys.exit(0)
```

The cudalt.py file has four main sections. The first section sets up the file names and creates the necessary sub-directories. The second section compiles the source so that it can be used in a shared library. The third section compiles the source so that it can be used in a static library and the last section creates the .lo file that the Automake system needs. Now that we have all of the plumbing in place let's take a look at how we make use of the plumbing in our target's Makefile.ams.

src/binary/Makefile.am

This target is a stand alone CUDA application. By stand alone I mean that the main and kernel source are in the same target. In this example we will make use of the first make rule that we defined in the top-level cuda.mk file (the .cu.o rule). Just in case you didn't memorize the rule here it is again:

```
.cu.o:
    $(NVCC) -gencode=arch=compute_13,code=sm_13 -o $@ -c $<
```

If you take a look at the files in the src/binary directory you will find the following:

```
Makefile.am  Makefile.in  binomialOptions_kernel.cu  main.cu
```

The `main.cu` file actually includes the `binomialOptions_kernel.cu` file so when we list the sources in the Makefile.am file we need only list `main.cu`. The contents of the Makefile.am are:

Makefile.am

```
include $(top_srcdir)/cuda.mk

binaryCuda_LINK = $(CC) $(binaryCuda_CFLAGS) $(CFLAGS) \
                  $(binaryCuda_LDFLAGS) $(LDFLAGS) -o $@

bin_PROGRAMS       = binaryCuda
binaryCuda_SOURCES = main.cu
EXTRA_DIST         = binomialOptions_kernel.cu

binaryCuda_CFLAGS  = $(CUDA_CFLAGS)
binaryCuda_LDADD   = $(CUDA_LIBS)
binaryCuda_LDFLAGS = $(CUDA_LDFLAGS)
```

The include at the top of the file pulls in our .cu.o make rule so that Automake knows how to compile a .cu file into a .o file. Since we are dealing with .cu files we need to tell Automake how to link our application. We do that with the binary_LINK variable. The rest of this file is pretty normal but make note of the EXTRA_DIST variable. This is necessary so that this file gets included in a make dist build. Since it is not listed as a SOURCE Automake doesn't know that it is important and would not distribute it. Also note the CUDA_ variables were defined earlier in our configure.ac file.

src/static-library/lib/Makefile.am

This target is a static library that encapsulates a CUDA kernel. This example will make use of the first rule that we defined in our top-level cuda.mk file (the .cu.o rule). Again for those of you who do not possess a photographic memory:

```
.cu.o:
    $(NVCC) -gencode=arch=compute_13,code=sm_13 -o $@ -c $<
```

If you look in the src/static-library/lib directory you will find the following:

```
Makefile.am              Makefile.in      binomialOptions.cu
binomialOptions_kernel.cu
```

The `binomialOptions.cu` file actually includes the `binomialOptions_kernel.cu` file so when we list the sources in the Makefile.am file we need only list `binomialOptions.cu`. The contents of the Makefile.am are:

Makefile.am

```
include $(top_srcdir)/cuda.mk

LINK = $(CC) -o  $@ $(CUDA_LDFLAGS) $(CUDA_LIBS)
EXTRA_DIST = binomialOptions_kernel.cu

lib_LIBRARIES = libbopm.a

libbopm_a_SOURCES = \
    binomialOptions.cu
```

The include at the top pulls in our .cu.o make rule and since we are dealing with .cu files we need to tell Autotools how to link with the LINK variable. The EXTRA_DIST is necessary so the kernel source gets distributed with a make dist build. The rest is just standard Automake usage.

src/static-library/main/Makefile.am

There is actually nothing at all special about this Makefile.am I am only including it here for completeness.

Makefile.am

```
bin_PROGRAMS = binaryCudaStLib

binaryCudaStLib_SOURCES = \
    main.c

binaryCudaStLib_LDADD = \
    $(top_builddir)/src/static-library/lib/libbopm.a

binaryCudaStLib_CPPFLAGS = \
    -I$(top_srcdir)

binaryCudaStLib_LDFLAGS = \
    -lm $(CUDA_LDFLAGS) $(CUDA_LIBS)
```

src/shared-library/lib/Makefile.am

This target is a shared library that encapsulates a CUDA kernel. This example will make use of the second rule that we defined in our top-level cuda.mk file (the .cu.lo rule). Again for those of you with the attention span of a fruit fly:

```
.cu.lo:
    $(top_srcdir)/cudalt.py $@ $(NVCC) \
    -gencode=arch=compute_13,code=sm_13 \
    --compiler-options=\" $(CFLAGS) $(DEFAULT_INCLUDES)\
    $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS) \" -c $<
```

If you look in the src/shared-library/lib directory you will find the following:

```
Makefile.am              Makefile.in      binomialOptions.cu
binomialOptions_kernel.cu
```

The `binomialOptions.cu` file actually includes the `binomialOptions_kernel.cu` file so when we list the sources in the Makefile.am file we need only list `binomialOptions.cu`. The contents of the Makefile.am are:

Makefile.am

```
include $(top_srcdir)/cuda.mk

LINK = $(LIBTOOL) --mode=link $(CC) -o $@ $(CUDA_LDFLAGS)\
       $(CUDA_LIBS)
EXTRA_DIST = binomialOptions_kernel.cu

lib_LTLIBRARIES = libbopm.la

libbopm_la_SOURCES = \
    binomialOptions.cu
```

As before the include at the top pulls in our .cu.lo make rule and since we are dealing with .cu files we need to tell Autotools how to link with the LINK variable. The EXTRA_DIST is necessary

so the kernel source gets distributed with a make dist build. The rest is just standard libtool usage.

src/shared-library/main/Makefile.am

There is actually nothing at all special about this Makefile.am I am only including it here for completeness.

Makefile.am

```
bin_PROGRAMS = binaryCudaShLib

binaryCudaShLib_SOURCES = \
    main.c

binaryCudaShLib_LDADD = \
    $(top_builddir)/src/shared-library/lib/libbopm.la

binaryCudaShLib_CPPFLAGS = \
    -I$(top_srcdir)

binaryCudaShLib_LDFLAGS = \
    -lm
```

Like I said... nothing to see here. The last piece to the puzzle is in the top-level Makefile.am. We have added a cudalt.py file to enable libtool like functionality for CUDA libraries but unless we make a change to the top-level Makefile.am that python file will not be distributed in a make dist build.

Top-level Makefile.am

```
SUBDIRS=src
EXTRA_DIST=cudalt.py
```

That's it! If you follow these simple guidelines you can distribute your CUDA code using GNU Autotools.

Epilog

I hope you enjoyed the preceding pages and found them informative. If you think you have found any inaccuracies please drop me a note at zaius@clusterchimps.org. I may not get back to you immediately (I do have a day job to pay the bills) but I appreciate any and all feedback.

If you would like to be notified of new publications just like us on Facebook. Any updates to all ClusterChimps tools and publications will posted to our Facebook page. We will be releasing a publication on how to build and program a Virtual Supercomputer soon so, if vast amounts of inexpensive computational capacity is something that interests you, be on the lookout for it.

ClusterChimps is dedicated to helping bring inexpensive supercomputing to the masses by leveraging emerging technologies coupled with bright ideas and open source software. We do this because we believe it will help advance computation intensive research areas including basic research, engineering, earth science, biology, materials science, and alternative energy research just to name a few.

- Dr Zaius