Configuring BLAS and LAPACK on the Beaglebone Black/Green

Last revision: 1.7.2018 – SDB.

Introduction

Many of the numerical algorithms you will implement on the Beaglebone Black (BBB) or the Beaglebone Green (BBG) require fast vector and matrix operations. Those operations are provided by the BLAS and LAPACK libraries. For example, BLAS supplies basic vector-vector products (dot), vector-matrix products (gemv), and matrix-matrix multiply (gemm) amongst many other operations. LAPACK supplies higher-level matrix operations such as solvers, matrix decompositions, SVD, etc. More information about these libraries is available at:

- **BLAS**: http://www.netlib.org/blas/
- **LAPACK**: http://www.netlib.org/lapack/

(Note that several implementations of these libraries are available on the Web. The above Web pages provide the "canonical" distributions of these libraries.) The stock BBB does not come with a full installation of BLAS or LAPACK suitable for code development. Therefore, you must install both the header files and the libraries yourself in order to build and run programs needing BLAS or LAPACK functions. This document details one way to install these libraries.

Prerequisites

The following procedure worked on a Beaglebone running Debian 9.2 (specifically, the image "Debian 9.2 2017-10-10 4GB SD IoT" available at https://beagleboard.org/latest-images). You can find the version of Linux running on your BBB using the "uname" command, like this:

```
root@beaglebone:~/music_playpen# cat /etc/dogtag
BeagleBoard.org Debian Image 2017-10-10
```

Accessing BLAS and LAPACK from a C program

The original BLAS and LAPACK libraries were written in Fortran. The stock BBB comes with a C compiler, but no Fortran compiler. In this class we assume you will write your programs in C. Therefore, you need a way to access the libraries from a C program. This access is provided by "wrapper functions" which you call from C. The wrapper functions then turn around and invoke the Fortran functions for you. The C wrapper around BLAS is called CBLAS. The C wrapper around LAPACK is called LAPACKE. The wrapper code is provided on Netlib by the BLAS and LAPACK project teams.

In this document I will provide links to pre-built, binary versions of the libraries which you can simply download and install onto your BBB. If you wish to build your own versions of these libraries, the easiest path is to cross-compile the libraries using a Linux host. In Appendices A, B and C I provide the procedure I used build the libraries. This procedure is not easy, so unless you want to fool around with cross-compilation under Linux, I recommend you just install the binaries.

Installing the header files

The makefiles I supply with my programs assume that the BLAS and LAPACK header files have been

installed into /usr/include. Highlighted below are the six header files to install. (I provide a link to a place on the web where you may find the files. I don't distribute them myself due to copyright considerations.)

- **cblas_f77.h , cblas.h** : These files are available at in the tar bundle at http://www.netlib.org/blas/blast-forum/cblas.tgz. Download that file into a convenient directory and open it using the command "tar -zxvf cblas.tgz". The header files are under CBLAS/include.
- **lapacke.h, lapacke_config.h, lapacke_mangling.h, lapacke_utils.h**: These files are available in the tar bundle at http://www.netlib.org/lapack/#_lapack_version_3_7_0. Download tht file into a convenient director and open it using the command "tar -zxvf lapack-3.7.0.tgz". The header files are under lapack-3.7.0/LAPACKE/include.

Download all the above files onto your host computer and open them up using tar to extract the desired files.

You must install all six header files for your BLAS and LAPACK installation to work. The header files should live in the /usr/include directory on the Beaglebone so they may be found during compilation. Once you have downloaded and located the files on your host computer, you may copy them over to your BBB using remote copy commands like

```
scp cblas_f77.h root@192.168.7.2:/usr/include
scp cblas.h root@192.168.7.2:/usr/include
etc...
```

Note that this copies the files into the /usr/include directory on the BBB, as desired.

Installing the pre-built libraries

I have created static libraries for the BBB compatible with glibc 4.6, and placed them on GitHub in the MUSIC_Playpen project. The files you want are:

- blas_LINUX.a
- cblas_LINUX.a
- liblapack.a
- liblapacke.a

Copy them onto the Beaglebone into the directory /usr/lib using commands like this

```
scp blas_LINUX.a root@192.168.7.2:/usr/lib
scp cblas_LINUX.a root@192.168.7.2:/usr/lib
etc...
```

Testing your installation

Once you have completed installation of BLAS and LAPACK, I recommend you test your installation by attempting to build the code in the MUSIC_Playpen project.

Appendix A: Creating a cross-compilation environment

In this appendix I detail how to create a cross-compilation environment for the BBB on your Linux PC – i.e. your laptop or your desktop computer. Ordinarily, if you compile a program on a particular computer, you intend to run that program on that computer. However, to build executables for the

BBB, we will use an Intel architecture, Linux PC to perform the compilation, but create binaries which will run on the BBB's ARM architecture. The process is called “cross-compilation”. The computer where the build is performed (i.e. your PC) is called the “host”, and the computer where the binaries will run (i.e. the Beaglebone) is called the “target”.

Here are the steps required to create a cross-compilation environment.

1. Download a compiler and utilities which you can use for cross-compilation. The preferred installation is provided by the Linaro organization. Binary packages with all the tools needed are available under https://releases.linaro.org/components/toolchain/binaries/.

   It is important to make sure the build environment you create on the host matches the version of gcc present on your BB. The best way to check this is to log into your BB and issue the command “gcc --version”. Then select the Linaro package closest to your gcc version. For example, on my Beaglebone Black, I get

   ```
   root@beaglebone:~# gcc --version
   gcc (Debian 6.3.0-18) 6.3.0 20170516
   ```

   Since the glibc on my BB is version 6.3.0, the package package want will be named something like “gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf.tar.xz”. This name means something like:

   - **gcc-linaro-6.3.1-2017.05**: GCC version 6.3.1, released in May 2017.
   - **x86_64**: Compiler runs on x86_64 (64 bit Intel) platform.
   - **arm-linux-gnueabihf:** The build target is an ARM processor running Linux. The ABI (binary interface to the libraries) assumes GNU/GCC. The ending “hf” means floating point is implemented in hardware.

   If the version of gcc on your BBB is too old (i.e. doesn't match an available gcc version), consider reflashing the BBB with a newer Linux distro compatible with one of the Linaro distributions.

2. Download the tar file into a convenient place on your host Linux machine. I placed mine in my home directory.
3. Linaro distributes binaries compressed using the .xz format. To open them up, use the command “tar --xz -xvf filename.xz”.
4. Now you will have a directory with all build tools under your home directory on the host. My build environment was installed under /home/sbrorson/Beaglebone/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf. Yours should be in a similar place.
5. The compilers you will need live under the bin subdirectory. Mine are under /home/sbrorson/Beaglebone/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-gnueabihf/bin.
6. To build a program, set your environment and invoke GCC on the host as follows:

   ```
   export PATH=$PATH:/home/sbrorson/Beaglebone/gcc-linaro-6.3.1-2017.05-
   x86_64_arm-linux-gnueabihf/bin

   export CC=arm-linux-gnueabihf-gcc

   ${CC} myprog.c -o myprog
   ```

   This command will build the code in the file “myprog.c” using the ARM GCC compiler, and will put the executable into the file called “myprog”.

7. Once you have created an executable on the host, you may copy it to the target BBB using the command “scp myprog root@192.168.7.2:/root/myprog”.
8. Then log into your BBB, and run the program by typing “./myprog”.

Once you have installed your cross-compilation environment, I suggest you test it out by writing a simple “hello_world.c” program, and following the above procedure to verify everything is working.

Appendix B: Building BLAS from source

In this appendix I present a procedure for building BLAS from source. Much of the procedure outlined in this section was originally found at this web page:

http://stackoverflow.com/questions/21263427/cross-compiling-armadillo-linear-algebra-library

Here is a step-by-step procedure. Do the following steps on your host computer (i.e. these are instructions for cross-compiling):

1. Download BLAS-3.7.0 from location linked here: http://www.netlib.org/blas/#_reference_blas_version_3_7_0. The file you want is blas-3.7.0.tgz
2. Open the file using “tar -zxvf blas-3.7.0.tgz”.
3. cd BLAS-3.7.0
4. Edit the file “make.inc” to look like this:

```
####################################################################
#  BLAS make include file.                                         #
#  March 2007                                                      #
####################################################################
#
SHELL = /bin/sh
#
#  The machine (platform) identifier to append to the library names
#
PLAT = _LINUX
#
#  Modify the FORTRAN and OPTS definitions to refer to the
#  compiler and desired compiler options for your machine.  NOOPT
#  refers to the compiler options desired when NO OPTIMIZATION is
#  selected.  Define LOADER and LOADOPTS to refer to the loader and
#  desired load options for your machine.
#
BASE = /home/sbrorson/Beaglebone/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-
gnueabihf/bin/

FORTRAN  = $(BASE)arm-linux-gnueabihf-gfortran
OPTS     = -O3 -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC
DRVOPTS  = $(OPTS)
NOOPT    = -O3 -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC
LOADER   = $(BASE)arm-linux-gnueabihf-gfortran
LOADOPTS = -O3 -mfpu=vfpv3 -mfloat-abi=hard -march=armv7
#
#  The archiver and the flag(s) to use when building archive (library)
#  If you system has no ranlib, set RANLIB = echo.
#
ARCH     = $(BASE)arm-linux-gnueabihf-gcc-ar
ARCHFLAGS= cr
RANLIB   = $(BASE)arm-linux-gnueabihf-ranlib
#
```

```
#  The location and name of the Reference BLAS library.
#
BLASLIB      = blas$(PLAT).a
```

The specific edits to make involve:

- Set BASE to point to your cross-compiler bin directory.
- Configure the names of the build programs (FORTRAN, ARCH, etc) to invoke the Linaro versions of the programs.
- Set the compile options to turn on hardware floating point.
- Set the compile options to include the flag -fPIC.

5. Do the build by issuing the command “make”. If everything built successfully, you will have a static library called “blas_LINUX.a” placed into the BLAS-3.7.0 directory.
6. Next download CBLAS from the location linked here: http://www.netlib.org/blas/#_cblas. The file you want is called “cblas.tgz”.
7. Open the file using the command “tar -zxvf cblas.tgz”.
8. cd CBLAS
9. Edit the file “Makefile.in” to look like this:

```
#
# Makefile.LINUX
#
#
# If you compile, change the name to Makefile.in.
#
#

#----------------------------------------------------------------------------
# Shell
#----------------------------------------------------------------------------

SHELL = /bin/sh

#----------------------------------------------------------------------------
# Platform
#----------------------------------------------------------------------------

PLAT = LINUX

#----------------------------------------------------------------------------
# Libraries and includes
#----------------------------------------------------------------------------

BASE = /home/sbrorson/Beaglebone/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-
gnueabihf/bin
BLLIB = /home/sbrorson/Beaglebone/BLAS-3.7.0/blas_LINUX.a
CBLIB = ../lib/cblas_$(PLAT).a

#----------------------------------------------------------------------------
# Compilers
#----------------------------------------------------------------------------

CC = $(BASE)arm-linux-gnueabihf-gcc
FC = $(BASE)arm-linux-gnueabihf-gfortran
LOADER = $(FC)
```

```
#--------------------------------------------------------------------------
# Flags for Compilers
#--------------------------------------------------------------------------

CFLAGS = -O3 -DADD_ -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC
FFLAGS = -O3 -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC

#--------------------------------------------------------------------------
# Archive programs and flags
#--------------------------------------------------------------------------

ARCH = $(BASE)arm-linux-gnueabihf-gcc-ar
ARCHFLAGS = cr
RANLIB = $(BASE)arm-linux-gnueabihf-ranlib
```

The specific things to change include:

- Set the BASE directory to point to your cross-compiler's bin directory.
- Set BLIB to point to the place where the BLAS libray was dumped when built.
- Set the build tool names names to invoke the Linaro tools.
- Set the build flags to turn on hardware floating point.
- Set the compile options to include the flag -fPIC.

10. Now build everything using the command “make”.
11. If everything builds correctly, you should have a static library called “cblas_LINUX.a” placed into the CBLAS/lib directory. You may copy them over to your BB, into the /usr/include directory.

Appendix C: Building LAPACK from source

In this appendix I present a procedure for building LAPACK from source. Some of the procedure outlined in this section was originally found at this web page:

http://stackoverflow.com/questions/21263427/cross-compiling-armadillo-linear-algebra-library

However, unlike building BLAS, I had to make a number of changes to the above procedure in order to get LAPACK working on my BBB.

Here is a step-by-step procedure. Do the following steps on your host computer (i.e. these are instructions for cross-compiling).

1. Download lapack-3.7.0 from the location linked here: http://www.netlib.org/lapack/#_lapack_version_3_7_0. Download the file lapack-3.7.0.tgz.
2. Open the file using “tar -zxvf lapack-3.7.0.tgz”.
3. cd lapack-3.7.0
4. Edit the file “make.inc” to look like this:

```
####################################################################
#  LAPACK make include file.                                       #
#  LAPACK, Version 3.7.0                                           #
#  December 2016                                                   #
####################################################################
#
```

```
SHELL = /bin/sh
#
#  Modify the FORTRAN and OPTS definitions to refer to the
#  compiler and desired compiler options for your machine.  NOOPT
#  refers to the compiler options desired when NO OPTIMIZATION is
#  selected.  Define LOADER and LOADOPTS to refer to the loader and
#  desired load options for your machine.
#
#  Note: During a regular execution, LAPACK might create NaN and Inf
#  and handle these quantities appropriately. As a consequence, one
#  should not compile LAPACK with flags such as -ffpe-trap=overflow.
#
BASE = /home/sbrorson/Beaglebone/gcc-linaro-6.3.1-2017.05-x86_64_arm-linux-
gnueabihf/bin/


FORTRAN  = $(BASE)arm-linux-gnueabihf-gfortran
OPTS     = -O2 -frecursive -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC
DRVOPTS  = $(OPTS)
NOOPT    = -O0 -frecursive -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC
LOADER   = $(BASE)arm-linux-gnueabihf-gfortran
LOADOPTS = -mfpu=vfpv3 -mfloat-abi=hard -march=armv7
#
#  Comment out the following line to include deprecated routines to the
#  LAPACK library.
#
#BUILD_DEPRECATED = Yes
#
# Timer for the SECOND and DSECND routines
#
# Default : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME
# TIMER    = EXT_ETIME
# For RS6K : SECOND and DSECND will use a call to the EXTERNAL FUNCTION ETIME_
# TIMER    = EXT_ETIME_
# For gfortran compiler: SECOND and DSECND will use a call to the INTERNAL
FUNCTION ETIME
TIMER    = INT_ETIME
# If your Fortran compiler does not provide etime (like Nag Fortran Compiler,
etc...)
# SECOND and DSECND will use a call to the INTERNAL FUNCTION CPU_TIME
# TIMER    = INT_CPU_TIME
# If neither of this works...you can use the NONE value... In that case,
SECOND and DSECND will always return 0
# TIMER     = NONE
#
#  Configuration LAPACKE: Native C interface to LAPACK
#  To generate LAPACKE library: type 'make lapackelib'
#  Configuration file: turned off (default)
#  Complex types: C99 (default)
#  Name pattern: mixed case (default)
#  (64-bit) Data model: LP64 (default)
#
# CC is the C compiler, normally invoked with options CFLAGS.
#
CC = $(BASE)arm-linux-gnueabihf-gcc
CFLAGS = -O3 -DADD_ -mfpu=vfpv3 -mfloat-abi=hard -march=armv7 -fPIC
#
#  The archiver and the flag(s) to use when building archive (library)
#  If you system has no ranlib, set RANLIB = echo.
#
ARCH     = $(BASE)arm-linux-gnueabihf-gcc-ar
ARCHFLAGS= cr
```

```
RANLIB   = $(BASE)arm-linux-gnueabihf-ranlib
#
#  Location of the extended-precision BLAS (XBLAS) Fortran library
#  used for building and testing extended-precision routines.  The
#  relevant routines will be compiled and XBLAS will be linked only if
#  USEXBLAS is defined.
#
# USEXBLAS    = Yes
XBLASLIB     =
# XBLASLIB    = -lxblas
#
#  The location of the libraries to which you will link.  (The
#  machine-specific, optimized BLAS library should be used whenever
#  possible.)
#
BLASLIB      = /home/sbrorson/Beaglebone/BLAS/BLAS-3.7.0/blas_LINUX.a
CBLASLIB     = /home/sbrorson/Beaglebone/BLAS/CBLAS/lib/cblas_LINUX.a
LAPACKLIB    = liblapack.a
TMGLIB       = libtmglib.a
LAPACKELIB   = liblapacke.a
```

The specific things to change include:

- Set the BASE directory to point to your cross-compiler's bin directory.
- Set the build tool names names to invoke the Linaro tools.
- Set the build flags to turn on hardware floating point.
- Set the compile options to include the flag -fPIC.
- Set BLASLIB and CBLASLIB to point to the place where these libraries were placed when you built BLAS (Appendix B).

5. cd SRC && make
6. cd ../LAPACKE && make
7. cd .. && make. Make will build any remaining parts of the LAPACK library, and then will try to test it. The testing phase will fail; you will get error messages like this:

```
/bin/sh: ./testlsame: cannot execute binary file
/bin/sh: ./testslamch: cannot execute binary file
/bin/sh: ./testdlamch: cannot execute binary file
/bin/sh: line 1: ./testsecond: cannot execute binary file
/bin/sh: line 1: ./testdsecnd: cannot execute binary file
/bin/sh: line 1: ./testieee: cannot execute binary file
/bin/sh: line 1: ./testversion: cannot execute binary file
```

   These error messages are normal and expected – you can ignore them. They occur because make attempts to run some test programs using the libraries it just built. However, because the libraries were built for an ARM target system, they cannot be run on your Intel host.

8. At this point, you should have static libraries called “liblapack.a” and “liblapacke.a” placed into the lapack-3.7.0 directory. You may copy them over to your BB, into the /usr/lib directory.