Get started with Arm Performance Libraries

Version 1.0

Non-Confidential

Issue 04
102574_0100_04_en

Get started with Arm Performance Libraries

Release information

Document history

Issue	Date	Confidentiality	Change
0100-04	30 May 2022	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Arm Performance Libraries is available in two versions: a version that is part of Arm Compiler for Linux, and a standalone version. This tutorial describes how to get started with the version that is part of Arm Compiler for Linux. To learn about how to get started with the standalone version, see the Get started with standalone ArmPL tutorial.

Arm Performance Libraries provide optimized standard core math libraries for high-performance computing applications on Arm processors. The library routines, which are available through both Fortran and C interfaces, include:

- BLAS - Basic Linear Algebra Subprograms (including XBLAS, the extended precision BLAS).
- LAPACK 3.10.0 - a comprehensive package of higher level linear algebra routines.
- FFT functions - a set of Fast Fourier Transform routines for real and complex data using the FFTW interface.
- Sparse linear algebra.
- libamath - a subset of libm, which is a set of optimized mathematical functions.
- libastring - a subset of libc, which is a set of optimized string functions.

Arm Performance Libraries are built with OpenMP across many BLAS, LAPACK, FFT, and sparse routines in order to maximize your performance in multi-processor environments.

2. Installation

Arm Performance Libraries is installed as part of Arm Compiler for Linux. Refer to Installing Arm Compiler for Linux for details on how to perform the installation.

Note

To use Arm Performance Libraries functions in your code, you must include the header file `<armpl.h>`. This header file is located in `/opt/arm/<armpl_dir>/include/`, or `<install_dir>/<armpl_dir>/include/` if you have installed to a different location than the default. If you use FFTs, you will also need to include the `fftw3.h` header file. If you include other legacy header files such as `blas.h` or `lapack.h`, they will also work.

---

3. Environment configuration

This section describes how to load the correct environment module for Arm Performance Libraries.

Before you begin

- Your administrator has installed Arm Performance Libraries as described in Installation.
- The environment module has been made available, as described in Configure.

Procedure

Use the following steps to configure your environment for Arm Performance Libraries:

1. Check which environment modules are available:

```
module avail
```

> **Note**
>
> If you do not see the Arm Compiler for Linux (`arm*` and GCC `gnu*` modulefiles, configure your `MODULEPATH` environment variable to include the installation directory:
>
> ```
> export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/
> ```

2. Load the appropriate module, or modules, for your toolchain.

   For Arm Compiler for Linux, load the compiler modulefile:

```
module load acfl/<version>
```

   For GCC load both the GCC and GCC Arm Performance Libraries modulefiles:

```
module load gnu/<version>
module load armpl/<version>
```

> **Tip**
>
> Consider adding the `module load` command to your `.profile` to run it automatically every time you log in.

3. Check your environment using the following commands, according to the compiler you are using.

> **Note**
>
> Ensure that the command contains the appropriate library directories from `/opt/arm`, which you installed during the installation procedure:

<table>
<tr><th>Compiler</th><th>Command</th></tr>
<tr><td>armclang|armclang++|armflang</td><td><code>which {armclang|armclang++|armflang}</code></td></tr>
<tr><td>gcc|gfortran</td><td><code>which {gcc|gfortran}</code></td></tr>
</table>

4. Compile and test the examples

Arm Performance Libraries include a number of example programs to compile and run. The examples are located in /opt/arm/<armpl_dir>/examples*/, or <install_dir>/<armpl_dir>/examples*/, if you have installed to a different location than the default.

Multiple examples directories are provided in the installation. The suffix of the directory name indicates whether the examples inside link to the 32-bit ('_lp64') or 64-bit ('_ilp64') integer variants, and sequential (no suffix indicator) or OpenMP ('_mp') multi-threaded variants, of Arm Performance Libraries.

For more information about the examples provided, see the Arm Performance Libraries Reference Guide

The default set of examples in the 'examples' directory link to the sequential, 32-bit integers variant of Arm Performance Libraries.

Each examples* directory contains the following:

- A Makefile to build and execute all of the example programs.
- A number of different C examples, *.c.
- A number of different Fortran examples, *.f90.
- Expected output for each example, *.expected.

The Makefile compiles and runs each example, and compares the generated output to the expected output. Any differences are flagged as errors.

To compile the examples and run the tests:

1. Load the module for your compiler:
   - For Arm Compiler for Linux, use:

     ```
     module load acfl/<version>
     ```

   - For GCC, use:

     ```
     module load gnu/<version>
         module load armpl/<version>
     ```

2. Copy the 'examples*' directory somewhere writeable.
3. Change into the 'examples*' directory in the writeable location and run 'make':

```
cd path/to/examples*
    make
```

The Makefile that uses Arm Compiler for Linux produces output similar to the following sample:

```
Compiling program armplinfo.f90:
armflang -c -mcpu=native -armpl armplinfo.f90 -o armplinfo.o
Linking program armplinfo.exe:
armflang -mcpu=native armplinfo.o -armpl -o armplinfo.exe
Running program armplinfo.exe:
./armplinfo.exe > armplinfo.res
FORTRAN STOP
ARMPL (ARM Performance Libraries)

...

Testing: no example difference files were generated.
Test passed OK
```

Example: fftw_dft_r2c_1d_c_example.c

The `fftw_dft_r2c_1d_c_example.c` example does the following:

- Creates an FFT plan for a one-dimensional, real-to-Hermitian Fourier transform, and a plan for its inverse, Hermitian-to-real transform.
- Executes the first plan to output the transformed values in `y`.
- Destroys the first plan.
- Prints the components of the transform.
- Executes the second plan to get the original data, unscaled.
- Destroys the second plan.
- Outputs the original and restored values, scaled (they should be identical).

```
/*
 * fftw_dft_r2c_1d: FFT of a real sequence
 *
 * ARMPL version 22.0 Copyright Arm 2022
 */

#include <armpl.h>
#include <complex.h>
#include <fftw3.h>
#include <math.h>
#include <stdio.h>

int main(void) {
#define NMAX 20
        double xx[NMAX];
        double x[NMAX];
        // The output vector is of size (n/2)+1 as it is Hermitian
        fftw_complex y[NMAX / 2 + 1];

        printf(
            "ARMPL example: FFT of a real sequence using fftw_plan_dft_r2c_1d\n");
        printf(
            "----------------------------------------------------------------\n");
        printf("\n");

        /* The sequence of double data */
        int n = 7;
        x[0] = 0.34907;
        x[1] = 0.54890;
        x[2] = 0.74776;
        x[3] = 0.94459;
```

```
        x[4] = 1.13850;
        x[5] = 1.32850;
        x[6] = 1.51370;

        // Use dcopy to copy the values into another array (preserve input)
        cblas_dcopy(n, x, 1, xx, 1);

        // Initialise a plan for a real-to-complex 1d transform from x->y
        fftw_plan forward_plan = fftw_plan_dft_r2c_1d(n, x, y, FFTW_ESTIMATE);
        // Initialise a plan for a complex-to-real 1d transform from y->x (inverse)
        fftw_plan inverse_plan = fftw_plan_dft_c2r_1d(n, y, x, FFTW_ESTIMATE);

        // Execute the forward plan and then deallocate the plan
        /* NOTE: FFTW does NOT compute a normalised transform -
         * returned array will contain unscaled values */
        fftw_execute(forward_plan);
        fftw_destroy_plan(forward_plan);

        printf("Components of discrete Fourier transform:\n");
        printf("\n");
        int j;
        for (j = 0; j <= n / 2; j++)
                // Scale factor of 1/sqrt(n) to output normalised data
                printf("%4d   (%7.4f%7.4f)\n", j + 1, creal(y[j]) / sqrt(n),
                       cimag(y[j]) / sqrt(n));

        // Execute the reverse plan and then deallocate the plan
        /* NOTE: FFTW does NOT compute a normalised transform -
         * returned array will contain unscaled values */
        fftw_execute(inverse_plan);
        fftw_destroy_plan(inverse_plan);

        printf("\n");
        printf("Original sequence as restored by inverse transform:\n");
        printf("\n");
        printf("       Original  Restored\n");
        for (j = 0; j < n; j++)
                // Scale factor of 1/n to output normalised data
                printf("%4d   %7.4f   %7.4f\n", j + 1, xx[j], x[j] / n);
        return 0;
}
```

To compile and run the example take a copy of the code from <install-dir>/examples and follow the steps below:

1. To generate an object file, compile the source `fftw_dft_r2c_1d_c_example.c`:

Compiler	Command
armclang	`armclang -c -armpl fftw_dft_r2c_1d_c_example.c -o fftw_dft_r2c_1d_c_example.o`
gcc	`gcc -c -I<install_dir>/include fftw_dft_r2c_1d_c_example.c -o fftw_dft_r2c_1d_c_example.o`

2. Link the object code into an executable:

Compiler	Command
armclang	`armclang fftw_dft_r2c_1d_c_example.o -o fftw_dft_r2c_1d_c_example.exe -armpl -lm`
gcc	`gcc fftw_dft_r2c_1d_c_example.o -L<install_dir>/lib -o fftw_dft_r2c_1d_c_example.exe -larmpl_lp64 -lm`

The linker and compiler options are:

- `-armpl` provides a shorthand method to specify the required include, library, and link options to the Arm C/C++/Fortran Compiler. The available arguments it accepts are described in the Library selection section.
- `-L<install_dir>/lib` adds the Arm Performance Libraries location to the library search path.
- `-larmpl_lp64` links against Arm Performance Libraries.
- `-lm` links against the standard math libraries.

3. Run the executable on your Arm system:

```
./fftw_dft_r2c_1d_c_example.exe
```

The executable produces output as follows:

```
ARMPL example: FFT of a real sequence using fftw_plan_dft_r2c_1d
----------------------------------------------------------------

Components of discrete Fourier transform:

   1   ( 2.4836 0.0000)
   2   (-0.2660 0.5309)
   3   (-0.2577 0.2030)
   4   (-0.2564 0.0581)

Original sequence as restored by inverse transform:

       Original  Restored
   1    0.3491    0.3491
   2    0.5489    0.5489
   3    0.7478    0.7478
   4    0.9446    0.9446
   5    1.1385    1.1385
   6    1.3285    1.3285
   7    1.5137    1.5137
```

5. Optimized math routines - libamath

libamath (in the /opt/arm/<armpl_dir>/lib directory) contains AArch64-optimized versions of the following scalar functions, in both single and double precision: exponential (exp, ;exp2), logarithm (log, ;log2, ;log10), and error functions (erf, ;erfc). In addition, optimized single precision sine and cosine functions are included (sinf, ;cosf, ;sincosf). Linking to libamath ahead of libm will ensure use of these optimized functions.

libamath also contains vectorized versions (Neon and SVE) of all of the common ;math.h ;functions in libm. To provide an enhanced performance using optimized functions, these functions are used by Arm C/C++ Compiler whenever possible. The compiler automatically links to the libamath library. You do not have to supply any specific compiler options to initiate this behavior.

When using libamath with the GCC compiler, you must explicitly link to the libamath library before linking to libm. For example:

```
gcc code_with_math_routines.c -lamath -lm
```

```
gfortran code_with_math_routines.f -lamath -lm
```

6. Optimized string routines - libastring

libastring (in the `/opt/arm/<armpl_dir>/lib` directory) provides a set of replacement `string.h` functions which are optimized for AArch64: `bcmp`, `memchr`, `memcpy`, `memmove`, `memset`, `strchr`, `strchrnul`, `strcmpstrcpy`, `strlen`, `strncmp`, `strnlen`. Linking to libastring ahead of libc ensures use of these optimized functions.

As with the libamath library, to provide an enhanced performance by default, Arm C/C++ Compiler automatically links to the libastring library before it links to libc. You do not have to supply any specific compiler options to initiate this behavior.

When using libastring with the GCC compiler, you must explicitly link to the libastring library to benefit from the performance increase. For example:

```
gcc code_with_string_routines.c -lastring
```

```
gfortran code_with_string_routines.f -lastring
```

7. Library selection

To instruct your compiler to use the optimum version of Arm Performance Libraries for your target system, you can use the `-armpl` (Arm C/C++/Fortran Compiler) or `-larmpl` (GCC) option. These options also enable the optimized versions of the C mathematical functions declared in the `math.h` library, tuned scalar and vector implementations of Fortran math intrinsics, and auto-vectorization of mathematical functions (disable this using `-fno-simdmath`).

Supported options and arguments are:

Arm Compiler	GCC	Description
`-armpl`	`-larmpl`	Link to Arm Performance Libraries with the default settings.
`-armpl=lp64` (Default)	`-DINTEGER32` (Compile) `-larmpl_lp64` (Link)	Use 32-bit integers.
`-armpl=ilp64` (Default if using `-i8`)	`-DINTEGER64` (Compile) `-larmpl_ilp64` (Link)	Use 32-bit integers.
`-armpl=sequential` (Default)	`-larmpl_lp64`	Use the single-threaded library.
`-armpl=parallel` (Default if using `-fopenmp`)	`-larmpl_lp64_mp`	Use the OpenMP multi-threaded library.

Separate multiple arguments using a comma, for example: `-armpl=<arg1>,<arg2>`.

Default option and argument behavior

For information about the default behavior of the `-armpl` option and its arguments in Arm Compiler for Linux, see the Arm C/C++ Compiler or Arm Fortran Compiler reference guide.

Linking against static libraries

The Arm Performance Libraries are supplied in both static and shareable versions, `libarmpl_lp64.a` and `libarmpl_lp64.so`. By default, the commands given above link to the shareable version of the library, `libarmpl_lp64.so`, if that version exists in the specified directory.

To force linking with the static library, either:

- Use the compiler flag `-static`, for example:

```
{armclang|armclang++|armflang} driver.{c|cpp|f90} -L${ARMPL_LIBRARIES} -static -
larmpl_lp64
```

- Insert the name of the static library in the command line, for example:

```
{armclang|armclang++|armflang} driver.{c|cpp|f90} ${ARMPL_LIBRARIES}/
libarmpl_lp64.a
```

8. Documentation

The Arm Performance Libraries Reference Guide is available on the Arm Developer website.

9. Get support

To help identify the issue that you are experiencing, find information about the version of Arm Performance Libraries that you are using and the system that you are running on. You can obtain the necessary system and library information by running the `libarmpl.so` file. You can find the `libarmpl.so` file in the '`/lib/`' directory of your installation.

You must load the Arm Performance Libraries environment module for your system before running `libarmpl.so`. For example:

```
./<install_location>/lib/libarmpl.so
```

10. Related information

Here are some resources related to material in this guide:

- Arm Performance Libraries Reference Guide
- For further information about to the standard BLAS Fortran interfaces, refer to the BLAS FAQ
- For further information about the LAPACK and BLAS routines, refer to the LAPACK documentation