Get started with Arm Performance Libraries (stand-alone Linux version)

Version 24.10

Non-Confidential

Issue 00

102620_2410_00_en

Get started with Arm Performance Libraries (stand-alone Linux version)

Release information

Document history

Issue	Date	Confidentiality	Change
2410-00	1 October 2024	Non-Confidential	Updates coinciding with Arm PL 24.10
2404-00	4 April 2024	Non-Confidential	Updates coinciding with Arm PL 24.04
2310-00	30 September 2023	Non-Confidential	Updates coinciding with Arm PL 23.10
0100-04	30 May 2022	Non-Confidential	Initial release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

Arm Performance Libraries provides 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, cover the following functionality:

- BLAS - Basic Linear Algebra Subprograms.
- LAPACK 3.12.0 - 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.
- RNG functions for generating integer and floating point random numbers.
- libamath - an optimized collection of `math.h` mathematical functions.
- libastring - an optimized collection of `string.h` memory functions.

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

Arm Performance Libraries is available for Linux, macOS and Windows.

This tutorial describes how to get started with the stand-alone versions of Arm Performance Libraries for Linux, which are compatible with GCC and the NVIDIA HPC compilers (NVHPC). There is also a version of Arm Performance Libraries that is part of Arm Compiler for Linux. To learn about how to get started with the version in Arm Compiler for Linux, see the Get started with Arm Performance Libraries in Arm Compiler for Linux tutorial.

To learn about how to get started with the version of Arm Performance Libraries for macOS, see the Get started with Arm Performance Libraries for macOS tutorial. To learn about how to get started with the version of Arm Performance Libraries for Windows, see the Get started with Arm Performance Libraries for Windows tutorial.

2. Installation

The learn.arm.com install guide for Arm Performance Libraries covers the installation basics for all platforms.

Arm Performance Libraries can be downloaded from developer.arm.com.

Following installation you should have the environment variable `ARMPL_DIR` set to point to the directory in the Arm Performance Libraries installation which contains (amongst other things) the `include` and `lib` directories containing the header and library files.

3. Environment configuration

This section describes how to set up your environment before using Arm Performance Libraries with Linux.

Prerequisites

- You or your administrator has installed Arm Performance Libraries (see Installation).
- You have a compatible compiler installed on your system. The stand-alone versions of Arm Performance Libraries for Linux are compatible with one of:
    - GCC compilers (`gcc`, `g++` and `gfortran`), versions 7 to 14.
    - NVHPC compilers (`nvc`, `nvc++` and `nvfortran`), version 24.7.

Setup

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 Performance Libraries `armpl*` modulefiles, configure your `MODULEPATH` environment variable to include the installation directory:

```
export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/
```

2. Load the module for the compiler (GCC or NVHPC) that you are using.

For example:

```
module load armpl/24.10.0_gcc
```

4. Compile and test the examples

Arm Performance Libraries includes a number of example programs to compile and run.

The examples are located in `${ARMPL_DIR}/examples*`.

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.

Assuming you have first setup your environment to use Arm Performance Libraries (see Environment configuration), then to compile the examples and run the tests:

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

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

The examples Makefile for a `gcc` version of Arm Performance Libraries produces output similar to the following sample:

```
Compiling program armplinfo.f90:
gfortran -c -mcpu=native -I/opt/arm/armpl_24.10_gcc/include armplinfo.f90 -o
 armplinfo.o
Linking program armplinfo.exe:
gfortran -mcpu=native armplinfo.o -L/opt/arm/armpl_24.10_gcc/lib -larmpl -lamath -lm
 -o armplinfo.exe
Running program armplinfo.exe:
LD_LIBRARY_PATH=/opt/arm/armpl_24.10_gcc/lib:/opt/arm/gcc/lib64:/opt/arm/gcc-14.2.0/
lib:/opt/arm/armpl_24.10_gcc/lib  ./armplinfo.exe > armplinfo.res
ARMPL (ARM Performance Libraries)
Version 24.10.0-release
```

```
...

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 24.10 Copyright ARM 2024
 */

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

#include "round_eps_to_zero.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
        double y_real = round_eps_to_zero_d(creal(y[j]) / sqrt(n));
        double y_imag = round_eps_to_zero_d(cimag(y[j]) / sqrt(n));
        printf("%4d   (%7.4f%7.4f)\n", j + 1, y_real, y_imag);
    }

    // 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++) {
        double xx_j = round_eps_to_zero_d(xx[j]);
        // Scale factor of 1/n to output normalised data
        double x_j = round_eps_to_zero_d(x[j] / n);
        printf("%4d   %7.4f   %7.4f\n", j + 1, xx_j, x_j);
    }
    return 0;
}
```

To compile and run the example take a copy of the code from one of the examples directories and follow the steps below:

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

Compiler	Command
gcc	`gcc -c -I${ARMPL_DIR}/include fftw_dft_r2c_1d_c_example.c -o fftw_dft_r2c_1d_c_example.o`
nvc	`nvc -c -I${ARMPL_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
gcc	`gcc fftw_dft_r2c_1d_c_example.o -L${ARMPL_DIR}/lib -o fftw_dft_r2c_1d_c_example.exe -larmpl_lp64 -lm`
nvc	`nvc fftw_dft_r2c_1d_c_example.o -L${ARMPL_DIR}/lib -o fftw_dft_r2c_1d_c_example.exe -larmpl_lp64 -lm -fortranlibs`

The linker and compiler options are:

- `-I${ARMPL_DIR}/include` adds the Arm Performance Libraries location to the include directory search path.
- `-L${ARMPL_DIR}/lib` adds the Arm Performance Libraries location to the library search path.

- `-larmpl_lp64` links against Arm Performance Libraries (serial, 32-bit integer interfaces).
- `-lm` links against the standard math libraries.
- `-fortranlibs` links against the NVHPC Fortran runtime library.

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 contains AArch64-optimized versions of the following scalar `math.h` functions:

- `cosf`, `sinf`, `sincosf`, `tanf`, `acos(f)`, `asin(f)`, `atan(f)`, `atan2(f)`,
- `exp(f)`, `exp2(f)`, `expm1(f)`, `log(f)`, `log2(f)`, `log10(f)`, `log1p(f)`,
- `cosh(f)`, `sinh(f)`, `tanh(f)`, `acosh(f)`, `asinh(f)`, `atanh(f)`,
- `pow(f)`, `erf(f)`, `erfc(f)`, and `cbrt(f)`.

Suffix `f` indicates a single precision implementation, while no suffix indicates double precision and suffix `(f)` indicates that both precisions are available.

Linking to libamath will ensure use of the optimized functions ahead of the versions available in the system math library.

libamath also contains vectorized versions (Neon and SVE) of all of the common `math.h` functions in libm. It is provided as a static library, `libamath.a`, and as a dynamic library, `libamath.so`.

libamath is located in `${ARMPL_DIR}/lib` and function prototypes are given in the header file `$ARMPL_DIR/include/amath.h`. There is also an example in `$ARMPL_DIR/examples_lp64/amath.c`.

To benefit from the performance increase given by libamath, you must explicitly link to the libamath library before linking to libm. For example, with GCC compilers:

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

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

For more information about using the vectorized functions in libamath, see this community.arm.com blog.

6. Optimized string routines – libastring

libastring 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 system string libraries ensures use of these optimized functions.

libastring is located in `${ARMPL_DIR}/lib`. It is provided as a static library, `libstring.a`, and as a dynamic library, `libstring.so`.

libastring is located in `${ARMPL_DIR}/lib`. To benefit from the performance increase given by libastring, you must explicitly link to the libastring library before linking to libc. For example, with GCC compilers:

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

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

7. Library selection

Arm Performance Libraries contains multiple different types of library. Your installation contains both dynamic and static libraries, and, in each case, there are serial and multi-threaded libraries. Furthermore, for each of those combinations there are also libraries which take 32-bit integer arguments in function interfaces, and also libraries which take 64-bit integer arguments.

Here we show the options needed to use the different types of library.

Compile	Link	Description
`-I${ARMPL_DIR}/include`	`-larmpl_lp64`	Use 32-bit integers, single-threaded library.
`-DINTEGER64 -I${ARMPL_DIR}/include`	`-larmpl_ilp64`	Use 64-bit integers, single-threaded library.
`-I${ARMPL_DIR}/include`	`-larmpl_lp64_mp`	Use 32-bit integers, multi-threaded (OpenMP) library.
`-DINTEGER64 -I${ARMPL_DIR}/include`	`-larmpl_ilp64_mp`	Use 64-bit integers, multi-threaded (OpenMP) library.

Linking against static libraries

The libraries are supplied in both static and dynamic versions, `libarmpl_lp64.a` and `libarmpl_lp64.so`. By default, the commands given above link to the dynamic 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:

```
gcc driver.c -L${ARMPL_DIR}/lib -static -larmpl_lp64 -lm
```

```
nvc driver.c -L${ARMPL_DIR}/lib -static -larmpl_lp64 -lm -fortranlibs
```

- Insert the name of the static library in the command line:

```
gcc driver.c ${ARMPL_DIR}/lib/libarmpl_lp64.a -lm
```

8. Further information

The following links contain detailed documentation about different aspects of using Arm Performance Libraries:

- The learn.arm.com install guide shows how to install Arm Performance Libraries on all supported platforms.
- See the developer.arm.com downloads page for stand-alone versions of Arm Performance Libraries for the full list of supported platforms.
- Arm Compiler for Linux, which includes Arm Performance Libraries, can also be downloaded from developer.arm.com.
- Arm Performance Libraries Reference Guide provides comprehensive documentation for all functions.
- If you have any questions or queries about using Arm Performance libraries please post a message on the Compilers and Libraries support forum. See below for guidance on how to do this effectively.

Reporting issues

To get help with any issue that you are experiencing, it helps to report 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 `armpl-info` executable in the `bin` directory of your Arm Performance Libraries installation.

Tip

We also recommend using perf-libs-tools, which is an Open Source project that can be used to profile your usage of Arm Performance Libraries, and also includes some scripts to visualize the data it produces. Providing the output reports from a perf-libs-tools run of your application when posting on the forum is incredibly useful, especially when reporting performance-related issues.

Other releases of Arm Performance Libraries

Arm Performance Libraries is also available:

- As part of Arm Compiler for Linux.
- For macOS, compatible with LLVM.
- For Windows, compatible with MSVC and LLVM.