Get started with Arm Performance Libraries (Windows version)

Version 24.04

Non-Confidential

Issue 00
109361_2404_00_en

Get started with Arm Performance Libraries (Windows version)

Release information

Document history

Issue	Date	Confidentiality	Change
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

Proprietary Notice

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 (including XBLAS, the Extra Precise BLAS).
- LAPACK 3.11.0 - 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.

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

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 Windows.

Prerequisites

- You or your administrator has installed Arm Performance Libraries (see Installation).
- You have installed on your system either:
  - Microsoft Visual Studio, or
  - LLVM for Windows with `clang` C/C++ compiler and, for those with Fortran code to compile, `flang-new`.

See the Release Notes included in your installation for compiler version requirements associated with your release.

Setup

The Arm Performance Libraries installer takes care of setting the environment variable `ARMPL_DIR` and appending `%ARMPL_DIR%\bin` to your `PATH`. You should not need to do anything extra to configure your Windows system to use the libraries.

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.

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 C examples and run the tests:

1. Using the Windows File Explorer locate your installation of Arm Performance Libraries (`%ARMPL_DIR%`).
2. Copy one of the `examples*` folders somewhere writable.
3. Open a command prompt (search: `cmd`), then change into the `examples*` directory in the writeable location and run `nmake`:

```
cd path\to\examples*
nmake
```

The Makefile produces output similar to the following sample:

```
Compiling program armpl_dgemm_interleave_batch_c_example.c:

        cl.exe -c /MD /nologo /I"%ARMPL_DIR%\include"
 armpl_dgemm_interleave_batch_c_example.c /
Foarmpl_dgemm_interleave_batch_c_example.obj
armpl_dgemm_interleave_batch_c_example.c
Linking program armpl_dgemm_interleave_batch_c_example.exe:

        cl.exe /MD /Fearmpl_dgemm_interleave_batch_c_example.exe
 armpl_dgemm_interleave_batch_c_example.obj armpl_lp64_mp.dll.lib /link /
libpath:"%ARMPL_DIR%\lib"
Microsoft (R) C/C++ Optimizing Compiler Version 19.36.32532 for ARM64
```

```
Copyright (C) Microsoft Corporation.  All rights reserved.

Microsoft (R) Incremental Linker Version 14.36.32532.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:armpl_dgemm_interleave_batch_c_example.exe
"/libpath:C:\Program Files\Arm Performance Libraries\armpl_24.04\lib"
armpl_dgemm_interleave_batch_c_example.obj
armpl_lp64_mp.dll.lib

Running program armpl_dgemm_interleave_batch_c_example.exe with 4 threads:

        armpl_dgemm_interleave_batch_c_example.exe >
 armpl_dgemm_interleave_batch_c_example.res ;

ARMPL example: interleave-batch matrix multiplication
-----------------------------------------------------

blas_sum = 3.955600e+06
  ib_sum = 3.955600e+06

------------------------------------- Example
 armpl_dgemm_interleave_batch_c_example.exe completed ---------------------------

Compiling program dgesdd_c_example.c:
...

All tests completed
___________________

Success: All examples completed successfully
```

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 24.04 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
cl (MD)	`cl /MD /c /I%ARMPL_DIR%\include /Fofftw_dft_r2c_1d_c_example.obj fftw_dft_r2c_1d_c_example.c`
cl (MT)	`cl /MT /c /I%ARMPL_DIR%\include /Fofftw_dft_r2c_1d_c_example.obj fftw_dft_r2c_1d_c_example.c`
clang (MD)	`clang -fms-runtime-lib=dll -c -I%ARMPL_DIR%\include -o fftw_dft_r2c_1d_c_example.obj fftw_dft_r2c_1d_c_example.c`
clang (MT)	`clang -fms-runtime-lib=static -c -I%ARMPL_DIR%\include -o fftw_dft_r2c_1d_c_example.obj fftw_dft_r2c_1d_c_example.c`

Note

`clang` compiler flags `-fms-runtime-lib=dll` and `-fms-runtime-lib=static` correspond to the `cl /MD` and `/MT` flags, respectively.

Caution

The FFTW interface, used in this example, defines its own complex types `fftw_complex` (double precision) and `fftwf_complex` (single precision). If you are using the Microsoft compiler `cl` these types are defined using the Microsoft complex types `_Dcomplex` and `_Fcomplex`, respectively. See C complex math support for more details about the Microsoft complex types. When using clang on Windows with Arm Performance Libraries, the FFTW complex types are instead defined using simple 2-element arrays to represenset the complex types: `double [2]` and `float [2]`. The FFTW examples provided are set up to work with `cl` and use manipulation functions such as `creal` and `cimag` to extract the real and imaginary parts of a number. In order to get the Arm Performance Libraries FFTW examples to work with `clang` instead, you should replace calls with array manipulation instead. E.g. `creal(y[j])` in `fftw_dft_r2c_1d_c_example.c` becomes `y[j][0]` and `cimag(y[j])` becomes `y[j][1]`. For more information, see the Arm Performance Libraries Reference Guide where we discuss complex types on Windows.

2. Link the object code into an executable:

Compiler	Command
cl (MD)	`cl /MD fftw_dft_r2c_1d_c_example.obj /Fefftw_dft_r2c_1d_c_example.exe %ARMPL_DIR%\lib\armpl_lp64.lib`
cl (MT)	`cl /MT fftw_dft_r2c_1d_c_example.obj /Fefftw_dft_r2c_1d_c_example.exe %ARMPL_DIR%\lib\armpl_lp64.lib`
clang (MD)	`clang -fms-runtime-lib=dll fftw_dft_r2c_1d_c_example.obj -o fftw_dft_r2c_1d_c_example.exe  %ARMPL_DIR%\lib\armpl_lp64.lib`
clang (MT)	`clang -fms-runtime-lib=static fftw_dft_r2c_1d_c_example.obj -o fftw_dft_r2c_1d_c_example.exe  %ARMPL_DIR%\lib\libarmpl_lp64.lib`

The linker and compiler options are:

- `/MD` for `cl` and `-fms-runtime-lib=dll` for `clang`: compile and link code that will use DLL versions of the Microsoft UCRT.
- `/MT` for `cl` and `-fms-runtime-lib=static` for `clang`: compile and link code that will use static versions of the Microsoft UCRT.
- `-I%ARMPL_DIR%\include` adds the Arm Performance Libraries location to the include directory search path.
- `%ARMPL_DIR%\lib\armpl_lp64.lib` links against Arm Performance Libraries (serial, 32-bit integer interfaces, `/MD` linkage).
- `%ARMPL_DIR%\lib\libarmpl_lp64.lib` links against Arm Performance Libraries (serial, 32-bit integer interfaces, `/MT` linkage).

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 aheead of the versions available in the Microsoft C runtime library.

libamath also contains Neon vectorized versions of all of the common `math.h` functions. It is provided as a static library, `libamath.lib`, and as a dynamic library, `libamath.dll`.

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 showing how to call vector functions in `%ARMPL_DIR%\examples_lp64\amath.c`.

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

6. 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`	`%ARMPL_DIR\lib\armpl_lp64.dll.lib`	Use 32-bit integers, single-threaded library.
`/DINTEGER64 /I%ARMPL_DIR \include`	`%ARMPL_DIR%\lib\armpl_ilp64. dll.lib`	Use 64-bit integers, single-threaded library.
`/I%ARMPL_DIR\include`	`%ARMPL_DIR\lib\armpl_lp64_mp. dll.lib`	Use 32-bit integers, multi-threaded (OpenMP) library.
`/DINTEGER64 /I%ARMPL_DIR \include`	`%ARMPL_DIR%\lib\armpl_ilp64_mp. dll.lib`	Use 64-bit integers, multi-threaded (OpenMP) library.

Linking against static libraries

The libraries are supplied in both static and dynamic (DLL) versions. The commands given above link to the dynamic versions of the libraries. To force linking with static versions of the libraries, prefix the library names with `lib` and remove `.dll`. For example, `libarmpl_lp64_mp.lib` is the static 32-bit integer multi-threaded library. For more information about linkage on Windows, see the Windows UCRT linkage section of the Arm Performance Libraries Reference Guide.

7. 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.exe` program. You can find the `armpl-info.exe` file in the `%ARMPL_DIR%\bin` directory of your installation. Execute the program to see the information printed to the screen:

```
%ARMPL_DIR%\bin\armpl-info.exe
```

Other releases of Arm Performance Libraries

Arm Performance Libraries is also available:

- As part of Arm Compiler for Linux.
- As stand-alone Linux releases, compatible with GCC and NVHPC.
- For macOS, compatible with LLVM.