



Arm Toolchain for Linux User Guide

Version 21.1.1

Non-Confidential

Copyright © 2025 Arm Limited (or its affiliates).
All rights reserved.

Issue 02

110477_211_02_en



Arm Toolchain for Linux User Guide

This document is Non-Confidential.

Copyright © 2025 Arm Limited (or its affiliates). All rights reserved.

This document is protected by copyright and other intellectual property rights.

Arm only permits use of this document if you have reviewed and accepted [Arm's Proprietary Notice](#) found at the end of this document.

This document (110477_211_02_en) was issued on 2025-11-20. There might be a later issue at <https://developer.arm.com/documentation/110477>

The product version is 21.1.1.

See also: [Proprietary notice](#) | [Product and document information](#) | [Useful resources](#)

Start reading

If you prefer, you can skip to [the start of the content](#).

Intended audience

Software developers

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.

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>.

Contents

1. Welcome to the Arm Toolchain for Linux.....	4
2. Installation.....	5
3. Getting started.....	8
4. How to use the Arm Performance Libraries.....	15
5. How to use BOLT with our toolchain.....	18
6. ACfL to ATfL porting guide.....	21
7. Standards support.....	24
8. OpenMP support.....	28
Proprietary notice.....	31
Product and document information.....	33
Product status.....	33
Revision history.....	33
Conventions.....	34
Useful resources.....	36

1. Welcome to the Arm Toolchain for Linux

The Arm Toolchain for Linux provides a complete environment for compiling and tuning your server and HPC applications.

2. Installation

This section describes how to download and install the Arm Toolchain for Linux.



The recommended installation method requires system administrator (`root`) assistance. Non-root installation is also possible, you can read about it further.

Recommended installation method

The first step is to configure your Linux package manager to be able to fetch packages from Arm. You do this step only once.

From the available options, select the packages repository that matches your installed Linux distribution. This adds the Arm toolchain repository to the package manager:

- Ubuntu 22.04

```
$ curl "https://developer.arm.com/packages/arm-toolchains:ubuntu-22/jammy/Release.key" | sudo gpg --dearmor -o /usr/share/keyrings/obs-oss-arm-com.gpg

$ echo "deb [signed-by=/usr/share/keyrings/obs-oss-arm-com.gpg] https://developer.arm.com/packages/arm-toolchains:ubuntu-22/jammy/ ." | sudo tee /etc/apt/sources.list.d/obs-oss-arm-com.list

$ sudo apt update
```

- Ubuntu 24.04

```
$ curl "https://developer.arm.com/packages/arm-toolchains:ubuntu-24/noble/Release.key" | sudo gpg --dearmor -o /usr/share/keyrings/obs-oss-arm-com.gpg

$ echo "deb [signed-by=/usr/share/keyrings/obs-oss-arm-com.gpg] https://developer.arm.com/packages/arm-toolchains:ubuntu-24/noble/ ." | sudo tee /etc/apt/sources.list.d/obs-oss-arm-com.list

$ sudo apt update
```

- Red Hat Enterprise Linux 8

```
$ sudo dnf install 'dnf-command(config-manager)'

$ sudo dnf config-manager -y --add-repo https://developer.arm.com/packages/arm-toolchains:rhel-8/el8/arm-toolchains:rhel-8.repo
```

- Red Hat Enterprise Linux 9

```
$ sudo dnf install 'dnf-command(config-manager)'

$ sudo dnf config-manager -y --add-repo https://developer.arm.com/packages/arm-toolchains:rhel-9/el9/arm-toolchains:rhel-9.repo
```

- Red Hat Enterprise Linux 10

```
$ sudo dnf install 'dnf-command(config-manager)'  
  
$ sudo dnf config-manager -y --add-repo https://developer.arm.com/packages/arm-toolchains:rhel-10/el10/arm-toolchains:rhel-10.repo
```

- Amazon Linux 2023

```
$ sudo dnf install 'dnf-command(config-manager)'  
  
$ sudo dnf config-manager -y --add-repo https://developer.arm.com/packages/arm-toolchains:amzn-2023/al2023/arm-toolchains:amzn-2023.repo
```

- SUSE Linux Enterprise Server 15

```
$ sudo zypper ar -f https://developer.arm.com/packages/arm-toolchains:sles-15/sl15/arm-toolchains:sles-15.repo
```

Select the installation command appropriate for your Linux distribution. This installs the Arm Toolchain for Linux, along with the Arm Performance Libraries, which is a required dependency:

- Ubuntu systems

```
$ sudo apt install arm-toolchain-for-linux
```

- Red Hat Enterprise Linux and Amazon Linux systems

```
$ sudo dnf install arm-toolchain-for-linux
```

- SUSE Linux Enterprise Server systems

```
$ sudo zypper install arm-toolchain-for-linux
```

Non-root installation method

You can install the Arm Toolchain for Linux, along with the Arm Performance Libraries in any directory that you wish to specify. For this, the auxiliary installation script can be used. This script does not require system administrator (`root`) assistance:

```
$ bash <(curl -L https://developer.arm.com/-/cdn-downloads/permalink/Arm-Toolchain-for-Linux/Package/user_install.sh) --yes <installation_directory>
```

The `<installation_directory>` path can be either absolute or relative. For example, you can invoke a local directory installation:

```
$ bash <(curl -L https://developer.arm.com/-/cdn-downloads/permalink/Arm-Toolchain-for-Linux/Package/user_install.sh) --yes .
```

After completion, you can see the `opt` directory tree unpacked into your local directory.

3. Getting started

This section describes how to get started with the Arm Toolchain for Linux. It describes how to use it to compile a source code into an executable binary.

What is in the toolchain

The default installation site of Arm Toolchain for Linux is the `/opt/arm/arm-toolchain-for-linux` directory. It contains a complete set of LLVM tooling, header files, compiler libraries, and runtime libraries, including the OpenMP runtime. The main tools are as follows:

- `armclang` - the C compiler
- `armclang++` - the C++ compiler
- `armflang` - the Fortran compiler



The LLVM equivalents of these commands are also available and functionally identical:

- `clang` - the C compiler
- `clang++` - the C++ compiler
- `flang` - the Fortran compiler

Configure and load environment modules

After installation, you can invoke any of those commands with their absolute paths, for example:

```
$ /opt/arm/arm-toolchain-for-linux/bin/armclang -print-resource-dir  
/opt/arm/arm-toolchain-for-linux/lib/clang/<version>
```

Although fully operable, this is not the best way to use the toolchain. Therefore, we recommend using the environment modules. You can install these on most of the existing Linux distributions:

- Ubuntu systems

```
$ sudo apt install environment-modules
```

- Red Hat Enterprise Linux and Amazon Linux systems

```
$ sudo dnf install environment-modules
```

- SUSE Linux Enterprise Server systems

```
$ sudo zypper install environment-modules
```

After installing the environment modules, you must execute a profile script which matches with your default shell:

- BASH

```
$ source /etc/profile.d/modules.sh
```

- csh or tcsh

```
$ source /etc/profile.d/modules.csh
```

Use the `module use` command to update your `MODULEPATH` environment variable to include the path to the Arm Toolchain for Linux module files directory:

```
$ module use /opt/arm/modulefiles
```

Use the `module avail` command to examine the list of available modules.

To load the module for Arm Toolchain for Linux, type:

```
$ module load atfl
```

This loads the default version of Arm Toolchain for Linux.

If multiple versions are available, you can load the necessary version by specifying `atfl/<version>`.

Now, the toolchain commands are accessible from the command line:

```
$ which armclang
/opt/arm/arm-toolchain-for-linux/bin/armclang

$ which armclang++
/opt/arm/arm-toolchain-for-linux/bin/armclang++

$ which armflang
/opt/arm/arm-toolchain-for-linux/bin/armflang
```

Using the compiler

Example 1: Compile and run an example C program

This example shows a simple program stored in a `.c` file, for example: `hello.c`:

```
#include <stdio.h>

int main()
{
    printf("Hello, World!");
    return 0;
}
```

To generate an executable binary file, compile your example C program with the `armclang` compiler. Specify the input file name, `hello.c`, and the binary file name (using `-o`), `hello`:

```
$ armclang -o hello hello.c
```

Run the executable binary file `hello`:

```
$ ./hello  
Hello, World!
```

Example 2: Compile and run an example Fortran program

This example shows a simple program stored in a `.f90` file, for example: `hello.f90`:

```
program hello  
  print *, 'hello world'  
end program
```

To generate an executable binary file, compile your example Fortran program with the `armflang` compiler. Specify the input file name, `hello.f90`, and the binary file name (using `-o`), `hello`:

```
$ armflang -o hello hello.f90
```

Run the executable binary file `hello`:

```
$ ./hello  
hello world
```

Compile and link C/C++ programs

To generate an executable binary, compile your source file, for example, `source.c`, with the `armclang` command:

```
$ armclang source.c
```

A binary with the filename `a.out` is the output.

Optionally, use the `-o` option to set the executable filename, for example, `binary`:

```
$ armclang -o binary source.c
```

You can specify multiple source files on a single line. This command compiles each source file individually and links them into a single executable binary file. For example, to compile the source files `source1.c`, and `source2.c`, use:

```
$ armclang -o binary source1.c source2.c
```

To compile each of your source files individually into the object files, specify the compile-only option, `-c`. Pass the resulting object files into another invocation of `armclang` to link them into an executable binary file:

```
$ armclang -c source1.c
$ armclang -c source2.c
$ armclang -o binary source1.o source2.o
```



For the C/C++ compiler's command-line arguments reference see: [Clang command-line argument reference](#).

Compile and link Fortran programs

To generate an executable binary file, compile your source file, for example, `source.f90`, with the `armflang` command:

```
$ armflang source.f90
```

A binary with the filename `a.out` is the output.

You can specify multiple source files on a single line. This command compiles each source file individually and links them into a single executable binary file. For example, to compile the source files `source1.f90`, and `source2.f90`, use:

```
$ armflang -o binary source1.f90 source2.f90
```

To compile each of your source files individually into the object files, specify the compile-only option, `-c`. Pass the resulting object files into another invocation of `armflang` to link them into an executable binary file:

```
$ armflang -c source1.f90
$ armflang -c source2.f90
$ armflang -o binary source1.o source2.o
```

When mixing both C/C++ and Fortran codes in a single application, make sure that the Fortran runtime library is always linked in. You can ensure this by using the `armflang` command for linking:

```
$ armflang -c source1.f90
$ armclang -c source2.c
$ armflang -o binary source1.o source2.o
```

**Note**

For the Fortran compiler's command-line arguments reference visit this page: [Flang command-line argument reference](#).

Increase the optimization level

To control the optimization level, specify the `-o<level>` option on your compile line, and replace `<level>` with one of 0, 1, 2 or 3. The `-o0` option is the lowest, and the default, optimization level. `-o3` is the highest optimization level. Arm compilers perform auto-vectorization at level `-o2` and above.

For example, to compile the `source.c` file into an executable file named `binary`, and enable the `-o3` optimization level, use:

```
$ armclang -O3 -o binary source.c
```

To compile the `source.f90` file into an executable file named `binary`, and enable the `-o3` optimization level, use:

```
$ armflang -O3 -o binary source.f90
```

Similar to other compilers, the Arm Toolchain for Linux C and C++ compilers can also be supplied with the `-Ofast` command-line option. However this results in displaying the following deprecation warning:

```
warning: argument '-Ofast' is deprecated; use '-O3 -ffast-math' for
the same behavior, or '-O3' to enable only conforming optimizations [-Wdeprecated-ofast]
```

**Note**

You can achieve the effect of applying the `-Ofast` option when compiling the C/C++ programs by using the `-O3 -ffast-math` options instead.

For Fortran programs, using the `-Ofast` option triggers the following deprecation warning:

```
warning: argument '-Ofast' is deprecated; use '-O3 -ffast-math -fstack-
arrays' for the same behavior, or '-O3 -fstack-arrays' to enable only
conforming optimizations [-Wdeprecated-ofast]
```

You can achieve the effect of applying the `-Ofast` option when compiling the Fortran programs by using the `-O3 -ffast-math -fstack-arrays` options instead.

Compile and optimize using CPU auto-detection

If you tell the compiler what target CPU your application will run on, it can make target-specific optimization decisions. Target-specific optimization decisions help to ensure your application runs

as efficiently as possible. To tell the compiler to make target-specific compilation decisions, use the `-mcpu=<target>` option and replace `<target>` with your target processor, from a supported list of the targets.

The `-mcpu` option also supports the `native` argument. Using `-mcpu=native` enables the compiler to auto-detect the architecture and processor type of the CPU that you are running the compiler on.

For example, to auto-detect the target CPU and optimize your C application for this target, use:

```
$ armclang -O3 -mcpu=native -o binary source.c
```

To auto-detect the target CPU and optimize your Fortran application for this target, use:

```
$ armflang -O3 -mcpu=native -o binary source.f90
```

The `-mcpu` option supports a range of Armv8-A-based Systems-on-Chips (SoCs). When `-mcpu` is not specified, by default, `-mcpu=generic` is set. This setting generates a portable output suitable for any Armv8-A-based target.



Note

- The optimizations that are performed due to setting the `-mcpu` option, are independent of the optimizations that are performed due to setting the `-O<level>` option.
- If you run the compiler on one target, but plan to run the application you are compiling on a different target, do not use `-mcpu=native`. Instead, use `-mcpu=<target>` where `<target>` is the target processor that you run the application on.

Standards support

For information on C, C++, and Fortran language support in Arm Toolchain for Linux, see the [Standards support](#) section.

For details on OpenMP support in Arm Toolchain for Linux, see the [OpenMP support](#) section.

Fortran recommendations

When to use Arm Toolchain for Linux?

- To compile a code with the modern Fortran features (except coarrays/teams/collectives).
- To compile Applications that are standards compliant.
- To compile large scale applications like CP2K.
- To compile applications requiring quadruple precision real/complex type support.

When not to use Arm Toolchain for Linux?

- Performance is not guaranteed. For the users seeking highest performance, Arm Toolchain for Linux is not recommended.
- OpenMP support is experimental.

- Code containing non-standard features/intrinsics might not work as expected.
- CMake versions older than 3.28 are not supported.

4. How to use the Arm Performance Libraries

You can get greater performance from your code if you enable linking to the optimized math libraries at compilation time.

How to link to the Arm Performance Libraries

To enable you to get the best performance on Arm-based systems, we recommend linking to the Arm Performance Libraries. Arm Performance Libraries provide optimized standard core math libraries for high-performance computing applications on Arm processors. Through a C interface, the following types of routines are available:

- BLAS: Basic Linear Algebra Subprograms, including XBLAS, the extended precision BLAS
- LAPACK: 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
- libastring: A subset of libc, which is a set of optimized string functions

The easiest way to make the Arm Performance Libraries visible to the compiler is to load the `arm-performance-libraries` environment module. After the `atf1` environment module has been loaded, the `arm-performance-libraries` module is available for loading:

```
$ module load arm-performance-libraries
```

This sets the `ARMPL_`-prefixed environment variables and two other critical environment variables: `LD_LIBRARY_PATH` and `PKG_CONFIG_PATH`.

To obtain a set of the command-line flags required for a specific variant of the Arm Performance Libraries we recommend invoking the `pkg-config` command.

The Arm Performance Libraries export several `pkg-config` modules. You must pick the one that you actually need, particularly when you plan to use OpenMP. Notice the `seq` and `omp` name parts:



Note

```
$ pkg-config --list-all | grep armpl
armpl ArmPL - Arm Performance Libraries
armpl-Fortran-dynamic-ilp64-omp ArmPL - Arm Performance Libraries
armpl-Fortran-dynamic-ilp64-seq ArmPL - Arm Performance Libraries
armpl-Fortran-dynamic-lp64-omp ArmPL - Arm Performance Libraries
armpl-Fortran-dynamic-lp64-seq ArmPL - Arm Performance Libraries
armpl-Fortran-static-ilp64-omp ArmPL - Arm Performance Libraries
armpl-Fortran-static-ilp64-seq ArmPL - Arm Performance Libraries
armpl-Fortran-static-lp64-omp ArmPL - Arm Performance Libraries
armpl-Fortran-static-lp64-seq ArmPL - Arm Performance Libraries
armpl-dynamic-ilp64-omp ArmPL - Arm Performance Libraries
armpl-dynamic-ilp64-seq ArmPL - Arm Performance Libraries
armpl-dynamic-lp64-omp ArmPL - Arm Performance Libraries
```

armpl-dynamic-lp64-seq	ArmPL - Arm Performance Libraries
armpl-static-ilp64-omp	ArmPL - Arm Performance Libraries
armpl-static-ilp64-seq	ArmPL - Arm Performance Libraries
armpl-static-lp64-omp	ArmPL - Arm Performance Libraries
armpl-static-lp64-seq	ArmPL - Arm Performance Libraries

C/C++ examples

To link to the OpenMP multi-threaded Arm Performance Libraries with a 64-bit integer interface, and include compiler and library optimizations for an Neoverse N1-based system, use:

```
$ armclang -fopenmp -o binary_code_with_math_routines.c -mcpu=neoverse-n1 `pkg-config armpl-dynamic-ilp64-omp --cflags --libs`
```

To link to the OpenMP multi-threaded Arm Performance Libraries with a 32-bit integer interface, and build portable output that is suitable for any Armv8-A-based system, use:

```
$ armclang -fopenmp -o binary_code_with_math_routines.c -mcpu=generic `pkg-config armpl-dynamic-lp64-omp --cflags --libs`
```

To link to the serial implementation of Arm Performance Libraries with a 32-bit integer interface, and include compiler and library optimizations for an Neoverse V2-based system, use:

```
$ armclang -o binary_code_with_math_routines.c -mcpu=neoverse-v2 `pkg-config armpl-dynamic-lp64-seq --cflags --libs`
```

Fortran examples

To link to the OpenMP multi-threaded Arm Performance Libraries with a 64-bit integer interface, and include compiler and library optimizations for an Neoverse N2-based system, use:

```
$ armflang -fopenmp -o binary_code_with_math_routines.f90 -mcpu=neoverse-n2 `pkg-config armpl-dynamic-ilp64-omp --cflags --libs`
```

To link to the OpenMP multi-threaded Arm Performance Libraries with a 32-bit integer interface, and build portable output that is suitable for any Armv8-A-based system, use:

```
$ armflang -fopenmp -o binary_code_with_math_routines.f90 -mcpu=generic `pkg-config armpl-dynamic-lp64-omp --cflags --libs`
```

To link to the serial implementation of Arm Performance Libraries with a 32-bit integer interface, and include compiler and library optimizations for an Neoverse V1-based system, use:

```
$ armflang -o binary_code_with_math_routines.f90 -mcpu=neoverse-v1 `pkg-config armpl-dynamic-lp64-seq --cflags --libs`
```

More information

For more information see: [Get started with Arm Performance Libraries \(stand-alone Linux version\)](#).

To learn more about integrating Arm Performance Libraries with Arm Toolchain For Linux see: [Using Arm Performance Libraries \(ArmPL\) with ATfL](#).



The Arm Performance Libraries suite is also the provider of the vectorized math routines library, `libamath`. This is a subset of the `libm` functions, which makes it possible to vectorize the loops containing calls to those functions. The Arm Toolchain for Linux default configuration instructs the C/C++ and Fortran compilers to make use of this library during vectorization automatically. No further command-line options are needed. You can disable this by specifying the `-fveclib=none` option.

5. How to use BOLT with our toolchain

BOLT is a post-link optimizer available in ATfL. To benefit from it you can use the following tools:

- `llvm-bolt`
- `llvm-bolt-heatmap`
- `perf2bolt`

Example optimization of a pathological case

To try this example, download and adapt our Telemetry repository. You can do that with the following commands:

```
$ git clone https://git.gitlab.arm.com/telemetry-solution/telemetry-solution.git
$ cd telemetry-solution
$ git checkout ffaf62bd11d42213fe175fbe3d313a246c85535f
$ cd tools/ustress
```

As BOLT utilizes relocations, you must ensure their emission. In our example, this requires the following modification with the `sed` command:

```
$ sed -i 's/^LINKER_FLAGS = -lm$/LINKER_FLAGS = -lm -Wl,--emit-relocs/' Makefile
```

Also, you must deactivate the assertions:

```
$ sed -i 's/^\(s*assert.*\)$/\n\n1/' lli_cache_workload.c
```

Compile the example for a specific CPU. Available alternatives are:

- Neoverse N1 (CPU=NEOVERSE-N1)
- Neoverse N2 (CPU=NEOVERSE-N2)
- Neoverse V1 (CPU=NEOVERSE-V1)

The following example command compiles the example for Neoverse V1:

```
$ make CC=armclang CPU=NEOVERSE-V1 USE_C=1
```

You can verify, that the binary file contains relocations:

```
$ llvm-readelf --sections lli_cache_workload | grep .rela.text
[14] .rela.text          RELA               0000000000000000 0f34a0 0004e0 18    I
45  13    8
```

Record a performance profile from an example execution of this binary:

```
$ perf record -e cycles:u -- ./l1i_cache_workload 5
```

Convert this performance profile to the BOLT format using `perf2bolt`:

```
$ perf2bolt -p perf.data -o perf.boltdata --nl l1i_cache_workload
```

Optimize this binary with BOLT:

```
$ llvm-bolt \
  l1i_cache_workload \
  -o l1i_cache_workload.bolt \
  --data perf.boltdata \
  --dyno-stats \
  --print-profile-stats \
  -reorder-blocks=ext-tsp \
  -reorder-functions=hfsort \
  -split-functions \
  -split-all-cold \
  -split-eh
```

The name of the new optimized binary is `l1i_cache_workload.bolt`. You can use the `repeat` command of the `csh` shell to obtain a set of comparable statistics:

```
$ csh
% repeat 3 perf stat -e L1I_CACHE_REFILL,instructions -- ./l1i_cache_workload 5
% repeat 3 perf stat -e L1I_CACHE_REFILL,instructions -- ./l1i_cache_workload.bolt 5
% exit
```

Notice how much faster the second binary executes.

Using the `llvm-bolt-heatmap` tool, you can visualize the improvements with heatmaps:

```
$ perf record -e cycles:u -- ./l1i_cache_workload 5
$ llvm-bolt-heatmap -p perf.data --nl l1i_cache_workload -o heatmap-orig.txt
$ perf record -o perf-bolt.data -e cycles:u -- ./l1i_cache_workload.bolt 5
$ llvm-bolt-heatmap -p perf-bolt.data --nl l1i_cache_workload.bolt -o heatmap-bolt.txt
```

Inspect both heatmaps and observe the difference:

```
$ less heatmap-orig.txt
$ less heatmap-bolt.txt
```

As these heatmaps are usually big, we suggest using the `aha` utility to convert them into HTML pages and view them in a web browser:

```
$ aha -b -f heatmap-orig.txt heatmap-orig.htm  
$ aha -b -f heatmap-bolt.txt heatmap-bolt.html
```

6. ACfL to ATfL porting guide

This section describes the key differences between using the Arm Toolchain for Linux (ATfL) and the Arm Compiler for Linux (ACfL). It also lists command-line options and macros that you can use with ATfL as alternatives to those used in ACfL.

Both toolchains provide frontends for compiling C, C++, and Fortran code. The names of these frontends are consistent across both: `armclang` for C, `armclang++` for C++, and `armflang` for Fortran.

ATfL supports quadruple precision real and complex types. ACfL does not.

Reference version

Compiler	Version
Arm Compiler for Linux (ACfL)	24.10
Arm Toolchain for Linux (ATfL)	21.1

ArmPL integration

In ACfL, Arm Performance Libraries (ArmPL) are bundled with the compiler. To simplify usage, the `-armpl` flag is provided to automatically include the necessary headers and libraries for BLAS, LAPACK, FFT, and other numerical routines.

In contrast, ATfL does not include ArmPL directly, but it depends on the ArmPL package, which is installed alongside the toolchain. To use ArmPL with ATfL, you must manually specify the locations of the headers and libraries using `pkg-config`. See the [Getting started](#) guide for detailed instructions.



Vector math functions from libamath are accessible without manually specifying the libraries.

C/C++ frontend

The C/C++ frontend in ATfL is identical to the one used in ACfL. Both are built on the Clang frontend from the upstream LLVM project.

Fortran frontend

The Fortran frontend in ATfL is based on the new LLVM Flang project. This frontend is a modern, from-scratch implementation. In contrast, ACfL used the older Classic Flang frontend, available at <https://github.com/flang-compiler/flang>. Because ATfL introduces a new Fortran frontend, this section describes the differences in Fortran support between the two toolchains.

Major difference

	Compatibility
Binary	No
Module file	No
Array descriptor	No

Difference in Fortran features

Feature	ACfL	ATfL
Base Fortran standard	Fortran 2008	Fortran 2018
Base OpenMP Specification	OpenMP 4.0	OpenMP 3.1
Parameterized Derived Type	Supported	PDT Kind - Supported PDT Length - Not supported
Preprocessor	Use <code>-cpp</code> to switch ON	Always ON Use <code>-cpp</code> to switch ON processing of predefined macros and macros from the command line
Directives	<code>ivdep</code> , <code>prefetch</code> , <code>unroll</code> , <code>nounroll</code> , <code>vector always</code> , <code>vector vectorlength</code> , <code>novector</code>	<code>unroll</code> , <code>unroll_and_jam</code> , <code>nounroll</code> , <code>nounroll_and_jam</code> , <code>vector always</code> , <code>novector</code> , <code>fixed</code> , <code>free</code> , <code>ignore_tkr</code> , <code>assume_aligned</code> , <code>inline</code> , <code>forceinline</code> , <code>noinline</code>
Line length	2100	Unlimited
Recursive functions	Use <code>-frecursive</code>	Default
Main function	Is a library linked at link-time	Is generated in the object file containing the program statement

Difference in command-line flags

The following table summarises some of the most commonly used compiler flags in gfortran and gives their equivalent in the Arm Fortran compiler:

ACfL	ATfL	Description
<code>-module <path></code>	<code>-J <path></code>	Specifies a directory to place module files
	<code>-I <path></code>	Specifies a directory to search for module files
<code>-Mallocatable=03</code>	<code>-frealloc-lhs</code>	Use Fortran 2003 standard semantics for assignments to allocatables
<code>-Mallocatable=95</code>	<code>-fno-realloc-lhs</code>	Use pre-Fortran 2003 standard semantics for assignments to allocatables
<code>-r8</code>	<code>-fdefault-real-8</code>	Sets the default KIND for REAL and COMPLEX declarations, constants, functions, and intrinsics
<code>-i8</code>	<code>-fdefault-integer-8</code>	Set the default KIND for INTEGER and LOGICAL to 64bit (for example, <code>KIND = 8</code>)

Difference in predefined macros

The following table summarises the macros predefined in the `armflang` compiler:

ACfL	ATfL	Value	Description
__FLANG	__flang__	1	Selection of compiler dependent source at compile time
__arch64	__arch64	1	Selection of architecture dependent source at compile time
__aarch64__	__arch64__	1	Selection of architecture dependent source at compile time
__ARM_ARCH	N/A	8	Selection of architecture dependent source at compile time
__ARM_ARCH__	N/A	8	Selection of architecture dependent source at compile time
__armflang_major__	__flang_major__	24/21	Underlying LLVM version details
__armflang_minor__	__flang_minor__	10/1	Underlying LLVM version details
__armflang_version__	__flang_patchlevel__	24.10.1/0	Underlying LLVM version details
__linux__	__linux__	1	Targeted Operating System
__linux	__linux__	1	Targeted Operating System
linux	__linux__	1	Targeted Operating System

7. Standards support

C standard support

See the [Clang C status page](#) for the details on C standard support.

C++ standard support

See the [Clang C++ status page](#) for the details on C++ standard support.

Fortran Standards Support

This section summarizes Fortran standards support in Flang. The information is only provided as a guideline. The `TODOs` and the `Not Yet Implemented` messages emitted by the compiler for unimplemented features should be treated as authoritative.



No distinction is made between the support in the Parser/Semantics and MLIR or Lowering support.

Fortran 2023

See [A first take on Fortran 202X features for LLVM Flang](#) document describing the new features in Fortran 2023. The following table summarizes the status of all important Fortran 2023 features.

The Status column uses the letters P, Y, N for the implementation status:

- P : Partial. When the implementation is incomplete for a few cases
- Y : Yes. When the implementation is complete
- N : No. When the implementation is absent

Feature	Status	Comments
Allow longer statement lines and overall statement length	Y	•
Automatic allocation of lengths of character variables	N	•
The specifiers <code>typeof</code> and <code>classof</code>	N	•
Conditional expressions and arguments	N	•
More use of <code>boz</code> constants	P	All usages other than <code>enum</code> are supported
Intrinsics for extracting tokens from a string	N	•
Intrinsics for Trig functions that work in degrees	N	•
Intrinsics for Trig functions that work in half revolutions	N	•
Changes to <code>system_clock</code>	N	•
Changes for conformance with the new IEEE standard	Y	•
Additional named constants to specify kinds	Y	•

Feature	Status	Comments
Extensions for c_f_pointer intrinsic	N	•
Procedures for converting between fortran and c strings	N	•
The at edit descriptor	N	•
Control over leading zeros in output of real values	N	•
Extensions for Namelist	N	•
Allow an object of a type with a coarray ultimate component to be an array or allocatable	N	•
Put with Notify	N	•
Error conditions in collectives	N	•
Simple procedures	N	•
Using integer arrays to specify subscripts	N	•
Using integer arrays to specify rank and bound of an array	N	•
Using an integer constant to specify rank	N	•
Reduction specifier for do concurrent	P	Syntax is accepted
Enumerations	N	•

Fortran 2018

The following table summarizes the status of all important Fortran 2018 features.

The Status column uses the letters P, Y, N for the implementation status:

- P : Partial. When the implementation is incomplete for a few cases
- Y : Yes. When the implementation is complete
- N : No. When the implementation is absent

Feature	Status	Comments
Asynchronous communication	P	Syntax is accepted
Teams	N	•
Image failure	P	stat_failed_image is added
Form team statement	N	•
Change team construct	N	•
Coarrays allocated in teams	N	•
Critical construct	N	•
Lock and unlock statements	N	•
Events	N	•
Sync team construct	N	•
Image selectors	N	•
Intrinsic functions get_team and team_number	N	•
Intrinsic function image_index	N	•
Intrinsic function num_images	N	•
Intrinsic function this_image	N	•
Intrinsic move_alloc extensions	P	•

Feature	Status	Comments
Detecting failed and stopped images	N	•
Collective subroutines	N	•
New and enhanced atomic subroutines	N	•
Failed images and stat= specifiers	N	•
Intrinsic function coshape	N	•

Fortran 2008

The following table summarizes the status of all important Fortran 2008 features.

The Status column uses the letters P, Y, N for the implementation status:

- P : Partial. When the implementation is incomplete for a few cases
- Y : Yes. When the implementation is complete
- N : No. When the implementation is absent

Feature	Status	Comments
Coarrays	N	Lowering and runtime support is not implemented
do concurrent	P	Sequential execution works. Parallel support in progress
Internal procedure as an actual argument or pointer target	Y	Current implementation requires stack to be executable. See Proposal

Fortran 2003

The following table summarizes the status of all important Fortran 2003 features.

The Status column uses the letters P, Y, N for the implementation status:

- P : Partial. When the implementation is incomplete for a few cases
- Y : Yes. When the implementation is complete
- N : No. When the implementation is absent

Feature	Status	Comments
Parameterized Derived Types	P	PDT with length type parameters is not supported. See Proposal
Assignment to allocatable	P	Assignment to whole allocatable in FORALL is not implemented
Pointer Assignment	P	Polymorphic assignment in FORALL is not implemented
The VOLATILE attribute	P	VOLATILE in procedure interfaces is not implemented
Asynchronous input/output	P	IO will happen synchronously
MIN/MAX extensions for CHARACTER	P	Some variants are not supported

Fortran 95

All features are supported.

Fortran 90

All features are supported.

FORTTRAN 77

All features are supported.

8. OpenMP support

Clang OpenMP support

See the [status page](#) for the details on Clang OpenMP support.

Flang OpenMP support

This section describes the OpenMP API features supported by Flang. It is intended as a general reference. For the most accurate information on unimplemented features, see the compiler's TODO OR Not Yet Implemented messages, which we consider authoritative. With the exception of a few cases, Flang offers full support for OpenMP 2.5, and partial support for OpenMP 3.1, and OpenMP 4.0 standards. The tables below describe the current status of OpenMP 4.0, 3.1, 3.0 feature support.

Work is ongoing to add support for OpenMP 4.5 and newer versions. A support statement for these will be shared in the future.



Note

No distinction is made between the support in Parser/Semantics, MLIR, Lowering, or the OpenMPIRBuilder.

OpenMP 4.0

The table below details the implementation status of the OpenMP 4.0 features.

The Status column uses the letters P, Y, N for the implementation status:

- P : Partial. When the implementation is incomplete for a few cases
- Y : Yes. When the implementation is complete
- N : No. When the implementation is absent

Feature	Status	Comments
proc_bind clause	Y	•
simd construct	P	Some clauses are not supported
declare simd construct	N	•
do simd construct	Y	•
target data construct	N	•
target construct	N	•
target update construct	N	•
declare target directive	N	•
teams construct	N	•
distribute construct	N	•
distribute simd construct	N	•

Feature	Status	Comments
distribute parallel loop construct	N	•
distribute parallel loop simd construct	N	•
depend clause	P	Depend clause with array sections are not supported
declare reduction construct	N	•
atomic construct extensions	Y	•
cancel construct	N	•
cancellation point construct	N	•
parallel do simd construct	Y	•
target teams construct	N	•
teams distribute construct	N	•
teams distribute simd construct	N	•
target teams distribute construct	N	•
teams distribute parallel loop construct	N	•
target teams distribute parallel loop construct	N	•
teams distribute parallel loop simd construct	N	•
target teams distribute parallel loop simd construct	N	•

OpenMP 3.1, OpenMP 3.0

The table below describes the implementation status of the OpenMP 3.x features.

The Status column uses the letters P, Y, N for the implementation status:

- P : Partial. When the implementation is incomplete for a few cases
- Y : Yes. When the implementation is complete
- N : No. When the implementation is absent

Feature	Status	Comments
intent(in) in firstprivate	Y	•
pointers in firstprivate and lastprivate	Y	•
final and mergeable clauses in task	Y	•
taskyield construct	Y	•
atomic construct extensions	Y	•
assumed-size arrays are shared	Y	•
allocatable arrays in private, firstprivate, lastprivate, reduction, copyin, copyprivate	Y	•
firstprivate in default	Y	•
collapse clause	Y	•
schedule kind auto	Y	•
task construct	P	delayed execution of tasks is not supported
taskwait construct	Y	•

OpenMP 2.5, OpenMP 1.1

All of the features except the following are supported:

- `atomic`: complex type, different but compatible types in lhs and rhs
- `threadprivate` (character type) constructs/clauses

Proprietary Notice

This document is protected by copyright and other related rights and the use 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 Limited ("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 the subject matter of this document infringes any third party patents.

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

This document may include technical inaccuracies or typographical errors. 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, any patents, copyrights, trade secrets, trademarks, or other rights.

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.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted 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 or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm’s trademark usage guidelines at <https://www.arm.com/company/policies/trademarks>. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Product and document information

Read the information in these sections to understand the release status of the product and documentation, and the conventions used in Arm documents.

Product status

All products and services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Revision history

These sections can help you understand how the document has changed over time.

Document release information

The Document history table gives the issue number and the released date for each released issue of this document.

Document history

Issue	Date	Confidentiality	Change
211-02	20 November 2025	Non-Confidential	Update the list of supported Fortran directives and OpenMP specification
211-01	21 October 2025	Non-Confidential	Release based on upstream LLVM 21.1.1
201-00	30 April 2025	Non-Confidential	Initial release

Change history

The Change history tables describe the technical changes between released issues of this document in reverse order. Issue numbers match the revision history in [Document release information](#) on page 33.

Table 2: Differences between issues

Change	Location
Updated with 201-00 release details	Release note document
Updated with 211-01 release details	Release note document
Updated with 211-02 release details	Release note document

Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
monospace	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
monospace <u>underline</u>	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: <pre>MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2></pre>
SMALL CAPITALS	Terms that have specific technical meanings as defined in the <i>Arm® Glossary</i> . For example, IMPLEMENTATION DEFINED , IMPLEMENTATION SPECIFIC , UNKNOWN , and UNPREDICTABLE .



Caution

We recommend the following. If you do not follow these recommendations your system might not work.



Your system requires the following. If you do not follow these requirements your system will not work.



You are at risk of causing permanent damage to your system or your equipment, or harming yourself.



This information is important and needs your attention.



A useful tip that might make it easier, better or faster to perform a task.



A reminder of something important that relates to the information you are reading.

Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.