arm

Arm® C/C++ Compiler

Version 22.0.2

Developer and Reference Guide

Arm® C/C++ Compiler

Developer and Reference Guide

Copyright © 2018–2022 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
1900-00	2 November 2018	Non-Confidential	Document release for Arm C/C++ Compiler version 19.0
1910-00	8 March 2019	Non-Confidential	Update for Arm C/C++ Compiler version 19.1
1920-00	7 June 2019	Non-Confidential	Update for Arm C/C++ Compiler version 19.2
1930-00	30 August 2019	Non-Confidential	Update for Arm C/C++ Compiler version 19.3
2000-00	29 November 2019	Non-Confidential	Update for Arm C/C++ Compiler version 20.0
2010-00	23 April 2020	Non-Confidential	Update for Arm C/C++ Compiler version 20.1
2010-01	23 April 2020	Non-Confidential	Documentation update 1 for Arm C/C++ Compiler version 20.1
2020-00	25 June 2020	Non-Confidential	Update for Arm C/C++ Compiler version 20.2
2030-00	4 September 2020	Non-Confidential	Update for Arm C/C++ Compiler version 20.3
2030-01	16 October 2020	Non-Confidential	Documentation update 1 for Arm C/C++ Compiler version 20.3
2100-00	30 March 2021	Non-Confidential	Update for Arm C/C++ Compiler version 21.0
2110-00	24 August 2021	Non-Confidential	Update for Arm C/C++ Compiler version 21.1
2110-01	26 August 2021	Non-Confidential	Documentation update 1 for Arm C/C++ Compiler version 21.1

Issue	Date	Confidentiality	Change
2200-00	4 March 2022	Non-Confidential	Update for Arm C/C++ Compiler version 22.0
2201-00	1 April 2022	Non-Confidential	Update for Arm C/C++ Compiler version 22.0.1
2202-00	25 May 2022	Non-Confidential	Update for Arm C/C++ Compiler version 22.0.2

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.

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

Contents

List of Tables

1 Introduction

Provides information to help you use the Arm® C/C++ Compiler component of Arm® Compiler for Linux. Arm® C/C++ Compiler is an auto-vectorizing, Linux-space C and C++ compiler, tailored for Server and High Performance Computing (HPC) workloads. Arm® C/C++ Compiler supports Standard C and C++ source code and is tuned for Arm®v8-A based processors.

1.1 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. Signal 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 bold`	Language keywords when used outside example 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: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.
⚠ Caution	Recommendations. Not following these recommendations might lead to system failure or damage.
⚠ Warning	Requirements for the system. Not following these requirements might result in system failure or damage.

Convention	Use
⚠ Danger	Requirements for the system. Not following these requirements will result in system failure or damage.
📝 Note	An important piece of information that needs your attention.
💡 Tip	A useful tip that might make it easier, better or faster to perform a task.
📌 Remember	A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2 Get started

This chapter introduces Arm® C/C++ Compiler (part of Arm Compiler for Linux), and describes how to get started with the compiler.

2.1 Arm C/C++ Compiler

Arm® C/C++ Compiler is a Linux user space C/C++ compiler for server and High Performance Computing (HPC) Arm-based platforms. Arm C/C++ Compiler is built on the open-source Clang front-end and the LLVM 13.0.0-based optimization and code generation back-end.

Arm C/C++ Compiler supports modern C/C++ (see Supported C/C++ standards in Arm C/C++ Compiler), OpenMP 4.0, and OpenMP 4.5 standards, has a built-in autovectorizer, and is tuned for the 64-bit Arm architecture. Arm C/C++ Compiler also supports compiling for Scalable Vector Extension- (SVE-) and SVE2-enabled target platforms.

- A Linux user space C/C++ compiler for server and High-Performance Computing (HPC) Arm-based platforms.
- Built on the open-source Clang front-end and the LLVM-based optimization and code generation back-end.
- Tuned for the 64-bit Arm architecture, and includes a built-in autovectorizer.
- Packaged with Arm Fortran Compiler and Arm Performance Libraries in a single package called Arm Compiler for Linux.

Resources

To learn more about Arm C/C++ Compiler (part of Arm Compiler for Linux) and other Arm server and HPC tools, refer to the following information:

- **Arm Compiler for Linux**:
    - Arm C/C++ Compiler web page
    - Installation instructions
    - Release history
    - Supported platforms
- **Porting guidance**:
    - Porting and tuning resources
    - Arm GitLab Packages wiki
    - Arm HPC Ecosystem
- **SVE and SVE2 information**:
    - Scalable Vector Extension (SVE, and SVE2) information
    - For an overview of SVE and why it is useful for HPC, see Learn about the Scalable Vector Extension (SVE).

- ◦ For a list of SVE and SVE2 instructions, see the Arm A64 Instruction Set Architecture.

- ◦ White Paper: A sneak peek into SVE and VLA programming. An overview of SVE with information on the new registers, the new instructions, and the Vector Length Agnostic (VLA) programming technique, with some examples.

- ◦ White Paper: Arm Scalable Vector Extension and application to Machine Learning. In this white paper, code examples are presented that show how to vectorize some of the core computational kernels that are part of a machine learning system. These examples are written with the Vector Length Agnostic (VLA) approach introduced by the Scalable Vector Extension (SVE).

- ◦ Arm C Language Extensions (ACLE) for SVE. The SVE ACLE defines a set of C and C++ types and accessors for SVE vectors and predicates.

- ◦ DWARF for the ARM® 64-bit Architecture (AArch64) with SVE support. This document describes the use of the DWARF debug table format in the Application Binary Interface (ABI) for the Arm 64-bit architecture.

- ◦ Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support. This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the Arm 64-bit architecture.

- ◦ Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A. This supplement describes the Scalable Vector Extension to the Armv8-A architecture profile.

- **Support and sales**:

  - ◦ If you encounter a problem when developing your application and compiling with the Arm C/C++ Compiler, see Troubleshoot

  - ◦ Get software

> **Note**
>
> An HTML version of this guide is available in the `<install_location>/`
> `<package_name>/share` directory of your product installation.

2.2 Get started with Arm C/C++ Compiler

Describes how to get started with Arm® C/C++ Compiler. In this topic, you will find out where to download and find installation instructions for Arm Compiler for Linux and how to use Arm C/C++ Compiler to compile C/C++ source into an executable binary.

Before you begin
Download and install Arm Compiler for Linux. You can download Arm Compiler for Linux from the download page. Learn how to install and configure Arm Compiler for Linux, using the Arm Compiler for Linux installation instructions on the Arm Developer website.

Procedure

1. Load the environment module for Arm Compiler for Linux:
   a) As part of the installation, Arm recommends that your system administrator makes the Arm Compiler for Linux environment modules available to all users of the tool suite.
      To see which environment modules are available on your system, run:

   ```
   module avail
   ```

   > **Note**
   >
   > If you cannot see the Arm Compiler for Linux environment module, but you know the installation location, use `module use` to update your `MODULEPATH` environment variable to include that location:
   >
   > ```
   > module use <path/to/installation>/modulefiles/
   > ```
   >
   > replacing `<path/to/installation>` with the path to your installation of Arm Compiler for Linux. The default installation location is `/opt/arm/`.
   >
   > `module use` sets your `MODULEPATH` environment variable to include the installation directory:

   b) To load the module for Arm Compiler for Linux, run:

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

   where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

   For example:

   ```
   module load acfl/22.0.2
   ```

   c) Check your environment. Examine the `PATH` variable. `PATH` must contain the appropriate `bin` directory from `<path/to/installation>`:

   ```
   echo $PATH
   /opt/arm/arm-linux-compiler-22.0.2_Generic-AArch64_SUSE-15_aarch64-linux/
   bin:...
   ```

   > **Note**
   >
   > To automatically load the Arm Compiler for Linux every time you log into your Linux terminal, add the `module load` command for your system and product version to your `.profile` file.

2. To generate an executable binary, compile your program with Arm C/C++ Compiler.
   Specify the input source filename, `<source>.c|cpp`, and use `-o` to specify the output binary file, `<binary>`:

   ```
   {armclang|armclang++} -o <binary> <source>.{c|cpp}
   ```

Results

Arm C/C++ Compiler builds your binary `<binary>`.

To run your binary, use:

```
./<binary>
```

Example 2-1: Example: Compile and run a Hello World program

This example describes how to write, compile, and run a simple "Hello World" C program.

1. Load the environment module for your system:

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

    where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

    For example:

    ```
    module load acfl/22.0.2
    ```

2. Create a "Hello World" program and save it in a `.c` file, for example: `hello.c`:

    ```
    #include <stdio.h>
    int main() {
        printf("Hello, World!");
        return 0;
    }
    ```

3. To generate an executable binary, compile your Hello World program with Arm C/C++ Compiler.

    Specify the input file, `hello.c`, and the binary name (using `-o`), `hello`:

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

4. Run the generated binary `hello`:

    ```
    ./hello
    ```

Next steps

For more information about compiling and linking as separate steps, and how optimization levels effect auto-vectorization, see Compile and Link.

2.3 Use Arm Compiler for Linux securely in shared environments

Arm® Compiler for Linux provides features and language support in common with other toolchains. Misuse of these common features and language support can provide access to arbitrary files, execute system commands, and reveal the contents of environment variables.

If you deploy Arm Compiler for Linux into environments where security is a concern, then Arm strongly recommends that you do all of the following:

- To limit tool access to only the necessary files, sandbox the tools.
- Remove all non-essential environment variables.
- Prevent execution of other binaries.
- Segregate different users from one another.
- Limit execution time.

2.4 Get support

To see a list of all the supported compiler options in your terminal, use:

```
{armclang|armclang++} --help
```

or

```
man {armclang|armclang++}
```

A description of each supported command-line option is available in Compiler options.

If you encounter a problem when developing your application and compiling with the Arm® Compiler for Linux, see the Troubleshoot topic.

3 Compile and Link

This chapter describes the basic functionality of Arm® C/C++ Compiler, and describes how to compile your C/C++ source with `armclang` or `armclang++`.

3.1 Using the compiler

Describes how to generate executable binaries, compile and link object files, and enable optimization options, with Arm® C/C++ Compiler.

Compile and link

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

```
armclang -o source.c
```

A binary with the filename `source` is output.

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

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

You can specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary. 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 an object file, specify the compile-only option, `-c`, and then pass the resulting object files into another invocation of `armclang` to link them into an executable binary.

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

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`, `3`, or `fast`. `-O0` option is the lowest, and the default, optimization level. `-Ofast` is the highest optimization level. Arm C/C++ Compiler performs auto-vectorization at levels `-O2`, `O3`, and `-Ofast`.

For example, to compile `source.c` into a binary called `binary`, and use the `-O3` optimization level, use:

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

Compile and optimize using CPU auto-detection

If you tell Arm C/C++ Compiler what target CPU your application will run on, the compiler can make target-specific optimization decisions. Target-specific optimization decisions help 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 targets). To see what processors are supported by the `-mcpu` option, see -mcpu=.

In addition, the `-mcpu` option also supports a `native` argument. `-mcpu=native` enables Arm C/C++ 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 the application for this target, use:

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

The `-mcpu` option supports a range of Armv8-A-based Systems-on-Chips (SoCs), including: ThunderX2, Neoverse N1, Neoverse N2, Neoverse V1, and A64FX. When `-mcpu` is not specified, by default, `-mcpu=generic` is set, which generates portable output suitable for any Armv8-A-based target.

- The optimizations that are performed from setting the `-mcpu` option (also known as hardware, or CPU, tuning) are independent of the optimizations that are performed from setting the `-O<level>` option.

- If you run the compiler on one target, but will 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 will run the application on.

Note

Link to a math library

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

To enable you to get the best performance on Arm-based systems, Arm recommends linking to 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. To find out what the latest version of LAPACK that is supported in Arm Performance Libraries is, see Arm Performance Libraries.

- 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.

To instruct Arm C/C++ Compiler to use the optimum version of Arm Performance Libraries for your target architecture and implementation, you can use the `-armpl=` compiler option. `-armpl=` enables the optimized versions of the C mathematical functions that are declared in the `math.h` library and auto-vectorization of mathematical functions (which can be disabled using `-fno-simdmath`). `-armpl=` supports arguments which enable you to use 32- or 64-bit integers, and either the serial library or the OpenMP multi-threaded library.

For example:

- To link to the OpenMP multi-threaded Arm Performance Libraries with a 64-bit integer interface, and include compiler and library optimizations for an A64FX-based system, use:

```
armclang code_with_math_routines.c -armpl=ilp64,parallel -mcpu=a64fx
```

- 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 code_with_math_routines.c -armpl -fopenmp -mcpu=generic
```

- To link to the serial implementation of Arm Performance Libraries with a 32-bit integer interface, and include compiler and library optimizations for a Neoverse N1-based system, use:

```
armclang code_with_math_routines.c -armpl=lp64,sequential -mcpu=neoverse-n1
```

For a full list of supported arguments for `-armpl`, see -armpl=.

If you want to link to another custom library, you can specify the library to `armclang` using the `-l<library>` compiler option. For more information, see -l.

Common compiler options

This section describes some common options to use with Arm C/C++ Compiler.

> **Note**
>
> For more information about all the supported compiler options, run `man armclang`, `armclang --help`, or see Compiler options.

`-S`

Outputs assembly code, rather than object code. Produces a text `.s` file containing annotated assembly code.

-c

Performs the compilation step, but does not perform the link step. Produces an ELF object file (`<file>.o`). To later link object files into an executable binary, run `armclang` again, passing in the object files.

-o <file>

Specifies the name of the output file.

-march=name[+[no]feature]

Targets an architecture profile, generating generic code that runs on any processor of that architecture. For example `-march=armv8-a`, `-march=armv8-a+sve`, or `-march=armv8-a+sve2`.

> **Note**
>
> If you know what your target CPU is, Arm recommends using the `-mcpu` option instead of `-march`. For a complete list of supported targets, see -march=.

-mcpu=native

Enables the compiler to automatically detect the CPU you are running the compiler on, and optimize accordingly. The compiler selects a suitable target profile for that CPU. If you use `-mcpu`, you do not need to use the `-march` option.

`-mcpu` supports a number of Armv8-A-based Systems-on-Chip (SoCs), including: ThunderX2, Neoverse N1, Neoverse N2, Neoverse V1, and A64FX.

> **Note**
>
> When `-mcpu` is not specified, it defaults to `-mcpu=generic` which generates portable output suitable for any Armv8-A-based target.

For more information, see -mcpu=.

-O<level>

Specifies the level of optimization to use when compiling source files. The supported options are: `-O0`, `-O1`, `-O2`, `-O3`, and `-Ofast`. The default is `-O0`. Auto-vectorization is enabled at `-O2`, `-O3`, and `-Ofast`.

> **Warning**
>
> `-Ofast` performs aggressive optimizations that might violate strict compliance with language standards.

For more information, see -O.

--config /path/to/<config-file>

Passes the location of a configuration file to the compile command. Use a configuration file to specify a set of compile options to be run at compile time. The configuration file can be passed at compile time, or an environment variable can be set for it to be used for every

invocation of the compiler. For more information about creating and using a configuration file, see Configure Arm Compiler for Linux.

--help

Describes the most common options that are supported by Arm C/C++ Compiler. To see more detailed descriptions of all the options, use `man armclang`.

--version

Displays version information.

For a detailed description of all the supported compiler options, see Compiler options.

To view the supported options on the command-line, use the `man` pages:

```
man {armclang|armclang++}
```

Alternatively, if you use a bash terminal and have the 'bash-completion' package installed, you can use 'command line completion' (also known as 'tab completion'). To complete the command or option that you are typing in your terminal, press the **Tab** button on your keyboard. If there are multiple options available to complete the command or option with, the terminal presents these as a list. If an option is specified in full, and you press **Tab**, Arm Compiler for Linux returns the supported arguments to that option.

For more information about how command line completion is enabled for bash terminal users of Arm Compiler for Linux, see the installation instructions.

Related information

3.2 Compile C/C++ code for Arm SVE and SVE2-enabled processors

Arm® C/C++ Compiler supports compiling for Scalable Vector Extension (SVE) and Scalable Vector Extension version two (SVE2)-enabled target processors.

Before you begin

- Ensure you have installed Arm Compiler for Linux.

  For information about installing Arm Compiler for Linux, see Install Arm Compiler for Linux.

- Ensure you have loaded the environment module for Arm Compiler for Linux. To load the environment module, run:

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

where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

For example:

```
module load acfl/22.0.2
```

- Your target must be SVE- or SVE2-enabled hardware, or you must download, install, and load the correct environment module for Arm Instruction Emulator.

  For more information about installing and setting up your environment for Arm Instruction Emulator, see Install Arm Instruction Emulator.

About this task
SVE and SVE2 support enables you to:

- Assemble source code containing SVE and SVE2 instructions.

- Disassemble ELF object files containing SVE and SVE2 instructions.

- Compile C and C++ code for SVE and SVE2-enabled targets, with an advanced auto-vectorizer that is capable of taking advantage of the SVE and SVE2 features.

This topic shows you how to compile code to take advantage of SVE (or SVE2) functionality. The generated executable can be run on SVE-enabled (or SVE2-enabled) hardware, or emulated using Arm Instruction Emulator.

Procedure
1. Compile your SVE or SVE2 source:

   - If you are both compiling and running on SVE-enabled (or SVE2-enabled) hardware, enable compiler optimizations using `-mcpu=native`.

     To compile SVE or SVE2 code without linking to Arm Performance Libraries, use:

     ```
     armclang -O<level> -mcpu=native -o <binary> <source.c>
     ```

     To compile SVE or SVE2 code and link to Arm Performance Libraries, use:

     ```
     armclang -O<level> -mcpu=native -armpl -o <binary> <source.c>
     ```

   - To compile SVE (or SVE2) code on hardware that is not SVE-enabled, but that will be run on SVE-enabled (or SVE2-enabled) hardware, specify your SVE-enabled (or SVE2-enabled) processor using `-mcpu=<target>`.

     To compile SVE or SVE2 code without linking to Arm Performance Libraries, use:

     ```
     armclang -O<level> -mcpu=<target> -o <binary> <source.c>
     ```

To compile SVE or SVE2 code and link to Arm Performance Libraries, use:

```
armclang -O<level> -mcpu=<target> -armpl -o <binary> <source.c>
```

> **Note**
>
> If you do not know the target processor, specify an SVE-enabled target architecture using `-march=armv8-a+sve` (or an SVE2-enabled target using `-march=armv8-a+sve2`), instead of using `-mcpu=<target>`.

- To compile SVE (or SVE2) code to emulate with Arm Instruction Emulator, compile the code and specify an SVE-enabled (or SVE2-enabled) architecture using `-march=`.

  To compile SVE code without linking to Arm Performance Libraries, use:

  ```
  armclang -O<level> -march=armv8-a+sve -o <binary> <source.c>
  ```

  To compile SVE code and link to Arm Performance Libraries, use:

  ```
  armclang -O<level> -march=armv8-a+sve -armpl -o <binary> <source.c>
  ```

  To compile SVE code for an Armv8.2-A-based target, and link to Arm Performance Libraries, use:

  ```
  armclang -O<level> -march=armv8.2-a+sve -armpl -o <binary> <source.c>
  ```

> **Note**
>
> To compile SVE2 code, replace `+sve` with `+sve2` in the `-march` option argument.

For more information about the supported options for `-armpl`, for example to control using 32-bit or 64-bit integers, or to use the single or OpenMP multi-threaded library, see the `-armpl` description in -armpl=.

> **Note**
>
> - To enable optimal vectorization, set `-O<level>` to be `-O2`, or higher.
> - There are several SVE2 Cryptographic Extensions available: `sve2-aes`, `sve2-bitperm`, `sve2-sha3`, and `sve2-sm4`. Each extension is enabled using the `march` compiler option. For a full list of supported `-march` options, see -march=.
> - `sve2` also enables `sve`.

2. Run the executable:

- To run the executable on SVE-enabled (or SVE2-enabled) hardware, use:

```
./<binary>
```

- To run and emulate the instructions using Arm Instruction Emulator, use:

```
armie -msve-vector-bits=<value> ./<binary>
```

Replace `<value>` with the vector length to use (which must be a multiple of 128 bits up to 2048 bits).

Note

For more information about using Arm Instruction Emulator, see the Arm Instruction Emulator documentation.

Example 3-1: Example: Compile SVE code for any SVE-enabled architecture

This example compiles some application source (`source.c`) for any SVE-enabled target architecture, analyzes the vectorized SVE assembly, and runs the binary using Arm Instruction Emulator.

One benefit of SVE is the support for an automatic vector-length agnostic (VLA) programming model, which allows code to be compiled, and when run, the application adapts and uses the available vector length on the target. This means you can compile or hand-code your application for SVE once, and you do not need to rewrite or recompile it if you want to run it on another SVE-enabled target with a different vector length.

The following C code subtracts corresponding elements in two arrays and writes the result to a third array. The three arrays are declared using the `restrict` keyword, telling the compiler that they do not overlap in memory.

```c
// source.c
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
    for (int i = 0; i < ARRAYSIZE; i++)
    {
        a[i] = b[i] - c[i];
    }
}
int main()
{
    subtract_arrays(a, b, c);
}
```

1. Compile `source.c` and specify the output file to be assembly (`-s`):

```
armclang -O3 -S -march=armv8-a+sve source.c
```

> **Note**
>
> If you know your target processor, you can enable microarchitecture-level optimizations by using `-mcpu=<target>` instead of `-march=armv8-a+sve`.

The output assembly code is saved as `source.s`.

2. Inspect the output assembly code.

   The section of the generated assembly language file containing the compiled `subtract_arrays` function appears as follows:

   ```
   subtract_arrays:                             // @subtract_arrays
   // BB#0:
           orr     w9, wzr, #0x400
           mov     x8, xzr
           whilelo p0.s, xzr, x9
   .LBB0_1:                                     // =>This Inner Loop Header: Depth=1
           ld1w    {z0.s}, p0/z, [x1, x8, lsl #2]
           ld1w    {z1.s}, p0/z, [x2, x8, lsl #2]
           sub     z0.s, z0.s, z1.s
           st1w    {z0.s}, p0, [x0, x8, lsl #2]
           incw    x8
           whilelo p0.s, x8, x9
           b.mi    .LBB0_1
   // BB#2:
           ret
   ```

   SVE instructions operate on the `z` and `p` register banks. In this example, the inner loop is almost entirely composed of SVE instructions. The auto-vectorizer has converted the scalar loop from the original C source code into a vector loop.

3. Re-compile `source.c`, and this time build an executable:

   ```
   armclang -O3 -march=armv8-a+sve -o sve-binary source.c
   ```

   The output binary is saved as `sve-binary`.

4. Run the binary. Either:
   - Run the binary on SVE-enabled hardware:

     ```
     ./sve-binary
     ```

   - Run and emulate the binary using Arm Instruction Emulator, specifying a vector length of 512 bits:

     ```
     armie -msve-vector-bits=512 ./sve-binary
     ```

Related information

Learn about the Scalable Vector Extension (SVE)
Arm A64 Instruction Set Architecture
White Paper: A sneak peek into SVE and VLA programming
White Paper: Arm Scalable Vector Extension and application to Machine Learning
Arm C Language Extensions (ACLE) for SVE
DWARF for the ARM
Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support
Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A
Porting and Optimizing HPC Applications for Arm SVE

3.3 Generate annotated assembly code from C and C++ code

Arm® C/C++ Compiler can produce annotated assembly code. Generating annotated assembly code is a good first step to see how the compiler vectorizes loops.

Before you begin

- Install Arm Compiler for Linux. For information about installing Arm Compiler for Linux, see Install Arm Compiler for Linux.

- Load the module for Arm Compiler for Linux, run:

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

where `<package-version>` is `<major-version>.<minor-version>{.<patch-version>}`.

For example:

```
module load acfl/22.0.2
```

About this task

> **Note**
> To use SVE functionality, you need to use a different set of compiler options. For more information, refer to Compile C/C++ code for Arm SVE and SVE2-enabled processors.

Procedure

1. Compile your source and specify an assembly code output:

```
armclang -S <source>.c
```

The option `-s` is used to output assembly code.

The compiler outputs a `<source>.s` file.

2. Inspect the `<source>.s` file to see the annotated assembly code that was created.

Example 3-2: Example

This example compiles an example application source into assembly code without auto-vectorization, then re-compiles it with auto-vectorization enabled. You can compare the assembly code to see the effect the auto-vectorization has.

The following C application subtracts corresponding elements in two arrays, writing the result to a third array. The three arrays are declared using the `restrict` keyword, indicating to the compiler that they do not overlap in memory.

```c
// source.c
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
    for (int i = 0; i < ARRAYSIZE; i++)
    {
        a[i] = b[i] - c[i];
    }
}
int main()
{
    subtract_arrays(a, b, c);
}
```

1. Compile the example source without auto-vectorization (`-O1`) and specify an assembly code output (`-s`):

```
armclang -O1 -S -o source_O1.s source.c
```

The output assembly code is saved as `source_O1.s`. The section of the generated assembly language file that contains the compiled `subtract_arrays` function is as follows:

```
subtract_arrays:                          // @subtract_arrays
// BB#0:
        mov     x8, xzr
.LBB0_1:                                  // =>This Inner Loop Header: Depth=1
        ldr     w9, [x1, x8]
        ldr     w10, [x2, x8]
        sub     w9, w9, w10
        str     w9, [x0, x8]
        add     x8, x8, #4                // =4
        cmp     x8, #1, lsl #12           // =4096
        b.ne    .LBB0_1
// BB#2:
        ret
```

This code shows that the compiler has not performed any vectorization, because we specified the -o1 (low optimization) option. Array elements are iterated over one at a time. Each array element is a 32-bit or 4-byte integer, so the loop increments by 4 each time. The loop stops when it reaches the end of the array (1024 iterations * 4 bytes later).

2. Recompile the application with auto-vectorization enabled (-o2):

```
armclang -O2 -S -o source_O2.s source.c
```

The output assembly code is saved as `source_O2.s`. The section of the generated assembly language file that contains the compiled `subtract_arrays` function is as follows:

```
subtract_arrays:                        // @subtract_arrays
// BB#0:
        mov     x8, xzr
        add     x9, x0, #16             // =16
.LBB0_1:                                // =>This Inner Loop Header: Depth=1
        add     x10, x1, x8
        add     x11, x2, x8
        ldp     q0, q1, [x10]
        ldp     q2, q3, [x11]
        add     x10, x9, x8
        add     x8, x8, #32             // =32
        cmp     x8, #1, lsl #12         // =4096
        sub     v0.4s, v0.4s, v2.4s
        sub     v1.4s, v1.4s, v3.4s
        stp     q0, q1, [x10, #-16]
        b.ne    .LBB0_1
// BB#2:
        ret
```

This time, we can see that Arm C/C++ Compiler has done something different. SIMD (Single Instruction Multiple Data) instructions and registers have been used to vectorize the code. Notice that the LDP instruction is used to load array values into the 128-bit wide Q registers. Each vector instruction is operating on four array elements at a time, and the code is using two sets of Q registers to double up and operate on eight array elements in each iteration. Therefore, each loop iteration moves through the array by 32 bytes (2 sets * 4 elements * 4 bytes) at a time.

3.4 Writing inline SVE assembly

Inline assembly (or inline asm) provides a mechanism for inserting user-written assembly instructions into C and C++ code. This allows you to manually vectorize parts of a function without having to write the entire function in assembly code.

Note

This information assumes that you are familiar with details of the SVE architecture, including vector-length agnostic registers, predication, and WHILE operations.

Using inline assembly instead of writing a separate `.s` file has the following advantages:

- Inline assembly code shifts the burden of handling the procedure call standard (PCS) from the programmer to the compiler. This includes allocating the stack frame and preserving all necessary callee-saved registers.

- Inline assembly code gives the compiler more information about what the assembly code does.

- The compiler can inline the function that contains the assembly code into its callers.

- Inline assembly code can take immediate operands that depend on C-level constructs, such as the size of a structure or the byte offset of a particular structure field.

Structure of an inline assembly statement

The compiler supports the GNU form of inline assembly. It does not support the Microsoft form of inline assembly.

More detailed documentation of the `asm` construct is available at the GCC website.

Inline assembly statements have the following form:

```
asm ("instructions" : outputs : inputs : side-effects);
```

Where:

instructions

is a text string that contains AArch64 assembly instructions, with at least one newline sequence n between consecutive instructions.

outputs

is a comma-separated list of outputs from the assembly instructions.

inputs

is a comma-separated list of inputs to the assembly instructions.

side-effects

is a comma-separated list of effects that the assembly instructions have, besides reading from inputs and writing to outputs.

Also, the `asm` keyword might need to be followed by the `volatile` keyword.

Outputs

Each entry in outputs has one of the following forms:

```
[name] "=&register-class" (destination)
[name] "=register-class" (destination)
```

The first form has the register class preceded by =&. This specifies that the assembly instructions might read from one of the inputs (specified in the `asm` statement's inputs section) after writing to the output.

The second form has the register class preceded by =. This specifies that the assembly instructions never read from inputs in this way. Using the second form is an optimization. It allows the compiler to allocate the same register to the output as it allocates to one of the inputs.

Both forms specify that the assembly instructions produce an output that the compiler can store in the C object specified by `destination`. This can be any scalar value that is valid for the left side of a C assignment. The register-class field specifies the type of register that the assembly instructions require. It can be one of:

r

> if the register for this output when used within the assembly instructions is a general-purpose register (x0-x30)

w

> if the register for this output when used within the assembly instructions is a SIMD and floating-point register (v0-v31).

It is not possible for outputs to contain an SVE vector or predicate value. All uses of SVE registers must be internal to the inline assembly block.

It is the responsibility of the compiler to allocate a suitable output register and to copy that register into the `destination` after the `asm` statement is executed. The assembly instructions within the instructions section of the `asm` statement can use one of the following forms to refer to the output value:

%[name]

> to refer to an r-class output as `xN` or a w-class output as `vN`

%w[name]

> to refer to an r-class output as `wN`

%s[name]

> to refer to a w-class output as `sN`

%d[name]

> to refer to a w-class output as `dN`

In all cases `N` represents the number of the register that the compiler has allocated to the output. The use of these forms means that it is not necessary for the programmer to anticipate precisely which register is selected by the compiler. The following example creates a function that returns the value 10. It shows how the programmer is able to use the `%w[res]` form to describe the movement of a constant into the output register without knowing which register is used.

```
int f()
{
int result;
asm("movz %w[res], #10" : [res] "=r" (result));
return result;
}
```

In optimized output the compiler picks the return register (0) for `res`, resulting in the following assembly code:

```
movz w0, #10
ret
```

Inputs

Within an `asm` statement, each entry in the inputs section has the form:

```
[name] "operand-type" (value)
```

This construct specifies that the `asm` statement uses the scalar C expression value as an input, referred to within the assembly instructions as name. The operand-type field specifies how the input value is handled within the assembly instructions. It can be one of the following:

r

if the input is to be placed in a general-purpose register (`x0-x30`)

w

if the input is to be placed in a SIMD and floating-point register (`v0-v31`).

[output-name]

if the input is to be placed in the same register as output output-name. In this case the `[name]` part of the input specification is redundant and can be omitted. The assembly instructions can use the forms described in the **Outputs** section above (`%[name]`, `%w[name]`, `%s [name]`, `%d[name]`) to refer to both the input and the output.

i

if the input is an integer constant and is used as an immediate operand. The assembly instructions use `%[name]` in place of immediate operand `#N`, where `N` is the numerical value of value.

In the first two cases, it is the responsibility of the compiler to allocate a suitable register and to ensure that it contains value on entry to the assembly instructions. The assembly instructions must refer to these registers using the same syntax as for the outputs (`%[name]`, `%w[name]`, `%s [name]`, `%d[name]`).

It is not possible for inputs to contain an SVE vector or predicate value. All uses of SVE registers must be internal to instructions.

This example shows an `asm` directive with the same effect as the previous example, except that an i-form input is used to specify the constant to be assigned to the result.

```
int f()
{
int result;
asm("movz %w[res], %[value]" : [res] "=r" (result) : [value] "i" (10));
return result;
}
```

Side effects

Many `asm` statements have effects other than reading from inputs and writing to outputs. This is true of `asm` statements that implement vectorized loops, since most such loops read from or write to memory. The `side-effects` section of an `asm` statement tells the compiler what these additional effects are. Each entry must be one of the following:

`"memory"`

> if the `asm` statement reads from or writes to memory. This is necessary even if inputs contain pointers to the affected memory.

`"cc"`

> if the `asm` statement modifies the condition-code flags.

`"xN"`

> if the `asm` statement modifies general-purpose register N.

`"vN"`

> if the `asm` statement modifies SIMD and floating-point register N.

`"zN"`

> if the `asm` statement modifies SVE vector register N. Since SVE vector registers extend the SIMD and floating-point registers, this is equivalent to writing "vN".

`"pN"`

> if the `asm` statement modifies SVE predicate register N.

Use of volatile

Sometimes an `asm` statement might have dependencies and side effects that cannot be captured by the `asm` statement syntax. For example, if there are three separate `asm` statements (not three lines within a single `asm` statement), that do the following:

- The first sets the floating-point rounding mode.

- The second executes on the assumption that the rounding mode set by the first statement is in effect.

- The third statement restores the original floating-point rounding mode.

It is important that these statements are executed in order, but the `asm` statement syntax provides no direct method for representing the dependency between them. Instead, each statement must add the keyword `volatile` after `asm`. This prevents the compiler from removing the `asm` statement as dead code, even if the `asm` statement does not modify memory and if its results appear to be unused. The compiler always executes asm volatile statements in their original order.

For example:

```
asm volatile ("msr fpcr, %[flags]" :: [flags] "r" (new_fpcr_value));
```

> **Note**
> An `asm volatile` statement must still have a valid side effects list. For example, an asm volatile statement that modifies memory must still include `"memory"` in the side-effects section.

Labels

The compiler might output a given `asm` statement more than once, either as a result of optimizing the function that contains the `asm` statement or as a result of inlining that function into some of its callers. Therefore, `asm` statements must not define named labels like `.loop`, since if the `asm` statement is written more than once, the output contains more than one definition of label `.loop`. Instead, the assembler provides a concept of relative labels. Each relative label is simply a number and is defined in the same way as a normal label. For example, relative label 1 is defined by:

```
1:
```

The assembly code can contain many definitions of the same relative label. Code that refers to a relative label must add the letter `f` to refer to the next definition (`f` is for forward) or the letter `b` (backward) to refer to the previous definition. A typical assembly loop with a pre-loop test would therefore have the following structure. This allows the compiler output to contain many copies of this code without creating any ambiguity.

```
    ...pre-loop test...
    b.none      2f
1:
    ...loop...
    b.any       1b
2:
```

Example

The following example shows a simple function that performs a fused multiply-add operation (x=a·b +c) across four passed-in arrays of a size that is specified by `n`:

```
void f(double *restrict x, double *restrict a, double *restrict b, double *restrict
 c,
    unsigned long n)
{
for (unsigned long i = 0; i < n; ++i)
{
    x[i] = fma(a[i], b[i], c[i]);
}
}
```

An `asm` statement that exploited SVE instructions to achieve equivalent behavior might look like the following:

```
void f(double *x, double *a, double *b, double *c, unsigned long n)
{
unsigned long i;
asm ("whilelo p0.d, %[i], %[n]                    \n\
1:                                                \n\
        ld1d z0.d, p0/z, [%[a], %[i], lsl #3]  \n\
```

```
        ld1d z1.d, p0/z, [%[b], %[i], lsl #3]   \n\
        ld1d z2.d, p0/z, [%[c], %[i], lsl #3]   \n\
        fmla z2.d, p0/m, z0.d, z1.d             \n\
        st1d z2.d, p0, [%[x], %[i], lsl #3]     \n\
        uqincd %[i]                             \n\
        whilelo p0.d, %[i], %[n]                \n\
        b.any 1b"
: [i] "=&r" (i)
: "[i]" (0),
    [x] "r" (x),
    [a] "r" (a),
    [b] "r" (b),
    [c] "r" (c),
    [n] "r" (n)
: "memory", "cc", "p0", "z0", "z1", "z2");
}
```

> **Note**
>
> Keeping the `restrict` qualifiers would be valid but would have no effect.

The input specifier `"[i]" (0)` indicates that the assembly statements take an input 0 in the same register as output `[i]`. In other words, the initial value of `[i]` must be zero. The use of =& in the specification of `[i]` indicates that `[i]` cannot be allocated to the same register as `[x]`, `[a]`, `[b]`, `[c]`, or `[n]` (because the assembly instructions use those inputs after writing to `[i]`).

In this example, the C variable `i` is not used after the `asm` statement. The `asm` statement reserves a register that it can use as scratch space. Including `"memory"` in the side effects list indicates that the `asm` statement reads from and writes to memory. Therefore, the compiler must keep the `asm` statement even though `i` is not used.

3.5 Support for Vector Length Specific (VLS) programming

Describes the support for Vector Length Specific (VLS) programming in Arm® Compiler for Linux.

To compile VLS code with Arm Compiler for Linux 22.0.2, you must write code that uses the Arm C Language Extensions (ACLE) for SVE. Arm Compiler for Linux 22.0.2 does not support autovectorization with VLS programming.

In your code, the SVE vector length can be controlled using the `arm_sve_vector_bits` ACLE attribute. At compile time, you must specify the SVE vector length using the `-msve-vector-bits=<arg>` compiler option.

VLS code must only be executed on hardware which offers an SVE vector length that matches the SVE vector length that the code was designed and compiled for.

> **Note**
>
> You can check the SVE vector length of a target at compile-time using the `__ARM_FEATURE_SVE_BITS` define.

To learn more about VLS programming, and view some VLS code examples, see SVE Vector Length Specific (VLS) programming.

Example: A simple VLS code example

The following example (`c-acle-sve-vector-bits-simple.c`) shows how to protect Arm C Language Extension (ACLE) code at compile time, for an intended SVE vector length of 512 bits:

```
#include <arm_sve.h>
#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t vls_vec_t __attribute__((arm_sve_vector_bits(512)));
#else
#error Only -msve-vector-bits=512 is supported
#endif
// Function to add 2 input vectors.
vls_vec_t vls_add(vls_vec_t a, vls_vec_t b) {
    return svadd_s32_x(svptrue_b32(), a, b);
}
```

To specify the SVE vector length, the vector types are declared with the `arm_sve_vector_bits` attribute.

To compile `c-acle-sve-vector-bits-simple.c`, on the compile line, include the `-msve-vector-bits=512` option. For example:

```
 armclang -march=armv8-a+sve -c -O2 -msve-vector-bits=512 c-acle-sve-vector-bits-
simple.c
```

> **Note**
>
> If you do not specify the `-msve-vector-bits=<arg>` option, a compile-time error occurs, through the `#error` directive.

Arm Compiler for Linux 22.0.2 generates:

```
vls_add:
    add     z0.s, z0.s, z1.s
    ret
```

which uses the SVE instruction set.

Related information

SVE Vector Length Specific (VLS) programming
Arm C Language Extensions (ACLE) for SVE specification

4 Optimize

This chapter provides information about how to optimize your code for server and High Performance Computing (HPC) Arm-based platforms, and describes the optimization-specific features that Arm® C/C++ Compiler support you to optimize your code.

4.1 Optimizing C/C++ code with Arm SIMD (Neon)

Describes how to optimize with Advanced SIMD (Neon®) using Arm® C/C++ Compiler.

The Arm SIMD (or Advanced SIMD) architecture, its associated implementations, and supporting software, are commonly referred to as Neon technology. Arm Compiler for Linux generates SIMD instructions to accelerate repetitive operations on the large data sets commonly encountered with High Performance Computing (HPC) applications.

Arm SIMD instructions perform "Packed SIMD" processing; the SIMD instructions pack multiple lanes of data into large registers, then perform the same operation across all data lanes.

For example, consider the following SIMD instruction:

```
ADD V0.2D, V1.2D, V2.2D
```

The instruction specifies that an addition (`ADD`) operation is performed on two 64-bit data lanes (2D). `D` specifies the width of the data lane (doubleword, or 64 bits) and `2` specifies that two lanes are used (that is the full 128-bit register). Each lane in `v1` is added to the corresponding lane in `v2` and the result is stored in `v0`. Each lane is added separately. There are no carries between the lanes.

Coding with SIMD

To take advantage of SIMD instructions in your code:

- Let the compiler auto-vectorize your code.

  Arm C/C++ Compiler automatically vectorizes your code at the `-O2`, `-O3`, and `-Ofast` higher optimization levels. The compiler identifies suitable vectorization opportunities in your code and uses SIMD instructions where appropriate.

  At the `-O1` optimization level, you can use the `-fvectorize` option to enable auto-vectorization.

  At the lowest optimization level, `-O0`, auto-vectorization is never performed, even if you specify `-fvectorize`.

  For more information about auto-vectorization best practice, see Coding best practice for auto-vectorization.

- Use intrinsics directly in your C code.

Intrinsics are C or C++ pseudo-function calls that the compiler replaces with the appropriate SIMD instructions. Intrinsics let you use the data types and operations available in the SIMD implementation, while also allowing the compiler to handle instruction scheduling and register allocation. The available intrinsics are defined in the ARM C Language Extensions Architecture (ACLE) specification.

- Write SIMD assembly code.

> **Note**
>
> Optimizing SIMD assembly manually can be difficult because the pipeline and memory access timings have complex inter-dependencies. Instead of manually changing assembly code, Arm recommends the use of intrinsics.

Related information

-fvectorize on page 95
Overview of Neon technology
Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile
Coding for Neon
Arm Neon Programmer's Guide
Arm C Language Extensions

4.2 Optimizing C/C++ code with SVE and SVE2

The Arm®v8-A Scalable Vector Extension (SVE) and Scalable Vector Extension version two (SVE2) can be used to accelerate repetitive operations on the large data sets commonly encountered with High Performance Computing (HPC) applications.

To optimize your code using SVE, you can:

- Let the compiler auto-vectorize your code for you.

  Arm Compiler for Linux automatically vectorizes your code at optimization levels `-O2`, `-O3`, and `-Ofast`. The compiler identifies appropriate vectorization opportunities in your code and uses SVE instructions where appropriate.

  At optimization level `-O1` you can use the `-fvectorize` option to enable auto-vectorization.

  At the lowest optimization level, `-O0`, auto-vectorization is never performed, even if you specify `-fvectorize`. See Arm C/C++ Compiler Options by Function for more information on setting these options.

  For more information about auto-vectorization best practice, see Coding best practice for auto-vectorization.

- Use SVE intrinsics.

SVE intrinsics are function calls that the compiler replaces with an appropriate SVE instruction or sequence of SVE instructions. The SVE intrinsics provide almost as much control as writing SVE assembly code, but leave the allocation of registers to the compiler.

The SVE instrinsics are defined in the Arm C Language Extensions for SVE specification.

- Write SVE assembly code.

  For more information, see Writing inline SVE assembly.

Related information

-fvectorize on page 95

Porting and Optimizing HPC Applications for Arm SVE guide

Scalable Vector Extension (SVE, and SVE2) information

Learn about the Scalable Vector Extension (SVE)

Arm A64 Instruction Set Architecture

White Paper: A sneak peek into SVE and VLA programming

White Paper: Arm Scalable Vector Extension and application to Machine Learning

Arm C Language Extensions (ACLE) for SVE

DWARF for the ARM

Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support

Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A

4.3 Coding best practice for auto-vectorization

To produce optimal and auto-vectorized output, structure your code to provide hints to the compiler. Well-structured application code, that has hints, enables the compiler to detect code behaviors that it would otherwise not be able to detect. The more behaviors the compiler detects, the better vectorized your output code is.

Each of the following sections describe a different method that that can help the compiler to better detect code features.

Use `restrict`

If appropriate, use the `restrict` keyword when using C/C++ code. The C99 `restrict` keyword (or the non-standard C/C++ `__restrict__` keyword) indicates to the compiler that a specified pointer does not alias with any other pointers, for the lifetime of that pointer. `restrict` allows the compiler to vectorize loops more aggressively because it becomes possible to prove that loop iterations are independent and can be executed in parallel.

> **Note**
>
> C code might use either the `restrict` or `__restrict__` keywords. C++ code must use the `__restrict__` keyword.

If the restrict keywords are used incorrectly (that is, if another pointer is used to access the same memory) then the behavior is undefined. It is possible that the results of optimized code will differ from that of its unoptimized equivalent.

Use pragmas

The compiler supports pragmas. Use pragmas to explicitly indicate that loop iterations are independent of each other.

For more information, see Control auto-vectorization with pragmas.

Use < to construct loops

Where possible, use < conditions, rather than <= or != conditions, when constructing loops. < conditions help the compiler to prove that a loop terminates before the index variable wraps.

If signed integers are used, the compiler might be able to perform more loop optimizations because the C standard allows for undefined behavior in signed integer overflow. However, the C standard does not allow for undefined behavior in unsigned integers.

Use the `-ffast-math` option

The `-ffast-math` option can significantly improve the performance of generated code. However, it breaks compliance with IEEE and ISO standards for mathematical operations.

> **Warning**
>
> Ensure that your algorithms are tolerant of potential inaccuracies that could be introduced by the use of this option.

For more information, see -ffast-math.

4.4 Control auto-vectorization with pragmas

Arm® C/C++ Compiler supports pragmas to both encourage and suppress auto-vectorization. These pragmas use, and extend, the `pragma clang loop` directives.

For more information about the `pragma clang loop` directives, see Auto-Vectorization in LLVM, at llvm.org.

Note
In each of the following examples, the pragma only affects the loop statement immediately following it. If your code contains multiple nested loops, you must insert a pragma before each one to affect all the loops in the nest.

Enable auto-vectorization with pragmas

Auto-vectorization is enabled at the `-O2`, `-O3`, and `-Ofast` optimization levels. When enabled, auto-vectorization examines all loops.

If static analysis of a loop indicates that it might contain dependencies that hinder parallelism, auto-vectorization might not be performed. If you know that these dependencies do not hinder vectorization, use the `vectorize` pragma to inform the compiler.

To use the `vectorize` pragma, insert the following line immediately before the loop:

```
#pragma clang loop vectorize(assume_safety)
```

The preceding pragma indicates to the compiler that the following loop contains no data dependencies between loop iterations that would prevent vectorization. The compiler might be able to use this information to vectorize a loop, where it would not typically be possible.

Note
The `vectorize` pragma does not guarantee auto-vectorization. There might be other reasons why auto-vectorization is not possible or worthwhile for a particular loop.

Warning
Ensure that you only use this pragma when it is safe to do so. Using the `vectorize` pragma when there are data dependencies between loop iterations might result in incorrect behavior.

For example, consider the following loop, that processes an array `indices`. Each element in `indices` specifies the index into a larger `histogram` array. The referenced element in the `histogram` array is incremented.

```
void update(int *restrict histogram, int *restrict indices, int count)
{
  for (int i = 0; i < count; i++)
  {
    histogram[ indices[i] ]++;
  }
}
```

The compiler is unable to vectorize this loop, because the same index could appear more than once in the `indices` array. Therefore, a vectorized version of the algorithm would lose some of the increment operations if two identical indices are processed in the same vector load/increment/store sequence.

However, if you know that the `indices` array only ever contains unique elements, then it is useful to be able to force the compiler to vectorize this loop. This is accomplished by placing the `vectorize` pragma before the loop:

```
{
    #pragma clang loop vectorize(assume_safety)
    for (int i = 0; i < count; i++)
    {
        histogram[ indices[i] ]++;
    }
}
```

You can compile the file `c-histogram-assume-safety.c` with:

```
armclang -c -O2 c-histogram-assume-safety.c
```

Control auto-vectorization with pragmas

If auto-vectorization is not required for a specific loop, you can disable it or restrict it to only use Arm SIMD (Neon®) instructions.

To suppress auto-vectorization on a specific loop, add `#pragma clang loop vectorize(disable)` immediately before the loop.

In this example, a loop that would be trivially vectorized by the compiler is ignored:

```
void combine_arrays(int *restrict a, int *restrict b, int count)
{
    #pragma clang loop vectorize(disable)
    for ( int i = 0; i < count; i++ )
    {
        a[i] = b[i] + 1;
    }
}
```

Which, to compile, use:

```
armclang -c -O2 c-vectorize-disable.c
```

You can also hint to the compiler to use fixed-width (`fixed`) or scalable (`scalable`) vectorization using the `#pragma clang loop vectorize_width` hint. The `fixed` and `scalable` arguments are optional. By default, `fixed` is set.

`vectorize_width(fixed)` (default)

    Prefer fixed-width vectorization, resulting in Arm Neon instructions. For a loop with `vectorize_width(fixed)`, the compiler prefers to generate Arm Neon instructions, though SVE instructions might still be used with a fixed-width predicate (such as gather loads or scatter stores).

`vectorize_width(scalable)`

Prefer scaled-width vectorization, resulting in SVE instructions. For a loop with `vectorize_width(scalable)`, the compiler prefers SVE instructions but can choose to generate Arm Neon instructions or not vectorize at all.

For example:

```
void combine_arrays(int *restrict a, int *restrict b, int count)
{
    #pragma clang loop vectorize(enable) vectorize_width(fixed)
    for ( int i = 0; i < count; i++ )
    {
        a[i] = b[i] + 1;
    }
}
```

Which, to compile, use:

```
armclang -c -O2 -march=armv8.2-a+sve c-fixed-width.c
```

Unrolling and interleaving with pragmas

Unrolling and *Interleaving* are two, related, optimization methods which increase the amount of work executed in each iteration of the loop. This can increase Instruction-Level Parallelism (ILP), which can improve performance.

The following sections describe how to use pragmas to control the unrolling and interleaving behavior of the compiler.

Unrolling

Unrolling involves re-writing a scalar loop as a sequence of instructions so that loop overhead is reduced.

Take the following example:

```
void fn(int *data, int *input, int *other) {
    #pragma clang loop unroll_count(2)
    for (int i = 0; i < 64; i++) {
        data[i] = input[i] * other[i];
    }
}
```

Instead of making 64 iterations, you can re-write the loop to make only 32 iterations:

```
for (int i = 0; i < 64; i +=2) {
  data[i] = input[i] * other[i];
  data[i+1] = input[i+1] * other[i+1];
}
```

The second version of the code reduces the number of iterations by a factor of two. Reducing the number of iterations reduces the loop administration overhead, but can also increase the number of live variables and the register pressure.

For the preceding example, the factor by which a loop has been unrolled, is called the *Unrolling Factor* (UF), in this case, UF=2.

While the compiler can automate the task of unrolling, the manual method can provide more flexibility. An alternative manual approach is to use the "unroll" pragma to force the compiler to unroll a loop without applying cost benefit analysis. The `unroll` pragma allows you to tell the compiler to unroll the following loop using maximum UF (the internal limit), or to a user-defined `_value_`.

For example, to use the `unroll` pragma and unroll to the internal limit, use:

```
#pragma clang loop unroll(enable)
```

Or, to unroll to a user-defined UF of `_value_`, use:

```
#pragma clang loop unroll_count(_value_)
```

Interleaving

Interleaving is a specific instance of unrolling that is applied during vectorization. The result is as if the vectorized loop is then unrolled. Interleaving is applied where compiler heuristics show that vectorization would be beneficial. The decision to interleave is mainly dependent on register utilization.

Note

Interleaving is enabled by default in Arm Compiler for Linux, except for SVE-enabled targets.

Similar to unrolling, you can use use an `interleave` pragma to manually force the compiler to interleave a loop, without applying a cost benefit analysis. Interleaving is controlled by an *Interleaving Factor* (IF), which when set for the `interleave` pragma, can be the internal limit (the maximum IF) or a user-defined integer:

- To interleave to the internal limit, insert `#pragma clang loop interleave(enable)` before your loop.

- To interleave to a user-defined IF, insert `#pragma clang loop interleave_count(_value_)` before your loop, and replace `_value_` with the IF value.

  For example, to set an IF of eight, use:

```
void fn(int *data, int *input, int *other) {
    #pragma clang loop interleave_count(8) vectorize_width(4)
    for (int i = 0; i < 64; i++) {
```

```
        data[i] = input[i] * other[i];
    }
}
```

> **Note**
>
> Interleaving performed on a scalar loop does not unroll the loop correctly.

4.5 Predefined macro support

Discusses predefined macro support in Arm® C/C++ Compiler.

Predefined macros are object-like macros that are assigned a value automatically by the compiler.

Predefined macros are available to use in preprocessor statements in your code and allow you to check properties at compile time, and if required, change the code in response to those properties. The properties could be about the compiler, the compilation options, or the system being targeted.

This topic describes how you can generate an exhaustive list of the predefined macros that are supported by Arm C/C++ Compiler, then provides descriptions for some of the most useful (including ACLE-provided) predefined macros.

How to generate an exhaustive list of supported predefined macros

To generate an exhaustive list of the supported predefined C/C++ macros in Arm C/C++ Compiler, run:

```
armclang -x {c|c++} /dev/null -dM -E
```

Use one of `c` or `c++` to generate the predefined macros for `c` or `c++` code, respectively.

Where:

-x c|c++

Tells the compiler to override the file extension and instead use `c|c++`, as specified.

/dev/null

Provides an (empty) input file for the compiler. If you do not provide an input file, the compiler will produce an error: 'error: no input files'.

-dM

Prints the supported common and system-specific predefined macros that are available to the compiler during the preprocessing phase.

-E

Tells the compiler to stop after the pre-processing phase and print the output to stdout (unless redirected).

When you run one of the example compile lines, a list of supported predefined macros, with their values for your system, is printed to your terminal window.

Useful predefined macros for C/C++ code

The following table describes some of the most useful (but non-ACLE) predefined macros that are supported in Arm C/C++ Compiler:

Table 4-1: Supported predefined macros for C/C++ code

Macro	Value	Purpose
`__ARM_LINUX_COMPILER__`	1	Defined as an integer value and expands to 1 to indicate Arm Compiler for Linux.
`__ARM_LINUX_COMPILER_BUILD__`	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux build number.
`__armclang_major__`	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux major version number.
`__armclang_minor__`	INTEGER	Defined as an integer value and expands to the Arm Compiler for Linux minor version number.
`__armclang_version__`	STRING	Defined as a string value and expands to the full Arm Compiler for Linux version number.
`__DATE__`	STRING	Defined as a string value (format `mmm dd yyyy`) and expands to the current date.
`__FILE__`	STRING	Defined as a string value and expands to the filename of the current file. Where `__FILE__` reports a filepath in addition to the filename, the filepath is relative to the search path used by the preprocessor to locate the file. `__FILE__` is useful to use with `__LINE__` to identify both a file and line of code.
`__LINE__`	INTEGER	Defined as an integer value and expands to the number of the line of code that contains this macro. `__LINE__` is useful to use with `__FILE__` to identify both a file and line of code.
`_OPENMP`	INTEGER	Defined as a decimal integer literal value and expands to the year and month (`yyyymm`) of the OpenMP standard that is implemented.
`__TIME__`	STRING	Defined as a string value (format `hh:mm:ss`) and expands to the current time.

> **Note**
>
> The preceding list is not exhaustive. To see an exhaustive list, follow the instructions in the previous section.

Useful Arm C Language Extensions (ACLE) (for SVE) predefined macros

Arm C/C++ Compiler also supports the predefined macros that are provided by the ACLE and the ACLE for the Scalar Vector Extension (SVE).

All ACLE macros are specific to Arm-based systems, have the prefix `__ARM_`, and expand to integral constant expressions that are suitable for use in an `#if` directive, unless otherwise specified. For example, if you use `__ARM_FEATURE_FMA` to interpret whether the hardware floating-point architecture supports fused floating-point multiply-accumulate, the macro expands to `1` if the hardware does. An `#if` directive can be used to only run some code if the macro returns the expected value, for example:

```
#if __ARM_FEATURE_FMA = 1
  ...
```

To determine the version of the ACLE, or ACLE for SVE, specification that is implemented on your target, use:

- For ACLE: the `__ARM_ACLE` predefined macro. The version of ACLE is calculated using:

```
__ARM_ACLE = (100 \* major_version) + minor_version
```

For example, an implementation that implements version 2.1 of the ACLE specification defines `__ARM_ACLE` as `201`.

- `1` if the ACLE for SVE implemented on your hardware.

Some useful ACLE predefined macros include:

Table 4-2: ACLE predefined macros

Macro	Value	Purpose
`__ARM_ACLE`	INTEGER	Defined as an integer value and expands to the value that represents the ACLE version implementation.
`__ARM_FEATURE_COMPLEX`	1	Defined as an integer value and expands to 1 if the system supports complex addition and complex multiply-accumulate vector instructions.
`__ARM_FEATURE_DOTPROD`	1	Defined as an integer value and expands to 1 if the system: • Supports dot product data manipulation instructions • Has vector intrinsics available
`__ARM_FEATURE_FMA`	1	Defined as an integer value and expands to 1 if the system supports floating-point fused multiply-accumulate.
`__FP_FAST_FMA`	1	Defined as an integer value and expands to 1 if the supported `fma()` function evaluates faster than executing the expression `(x * y) + z`.

Note

A full list of the supported ACLE predefined macros is available in the ACLE specification.

Some useful ACLE for SVE (and SVE2) predefined macros include:

Table 4-3: ACLE for SVE (and SVE2) predefined macros

Macro	Value	Purpose
`__ARM_FEATURE_SVE`	1	Defined as an integer value and expands to 1 if the implementation generates code for an SVE target and that all the base SVE functions are available.
`__ARM_FEATURE_SVE_BF16`	1	Defined as an integer value and expands to 1 if all the BFloat 16 extension function are available.
`__ARM_FEATURE_SVE_BITS`	INTEGER	Defined as an integer value and expands to a non-zero value, `N`, if: • The implementation generates code for an SVE target • The `arm_sve_vector_bits(N)` attribute is available
`__ARM_FEATURE_SVE_MATMUL_FP32`	1	Defined as an integer value and expands to 1 if all the FP32 matrix multiply extension functions are available.
`__ARM_FEATURE_SVE_MATMUL_FP64`	1	Defined as an integer value and expands to 1 if all the FP64 matrix multiply extension functions are available.
`__ARM_FEATURE_SVE_MATMUL_INT8`	1	Defined as an integer value and expands to 1 if all theINT8 matrix multiple extension functions are available.
`__ARM_FEATURE_SVE_NONMEMBER_OPERATORS`	1	Defined as an integer value and expands to 1 if C++ code can define non-member operator functions for SVE types.
`__ARM_FEATURE_SVE_PREDICATE_OPERATORS`	1	Defined as an integer value and expands to 1 if, when you apply the `arm_sve_vector_bits` attribute to `svbool_t`, the attribute creates a type that supports basic built-in vector operations.
`__ARM_FEATURE_SVE_VECTOR_OPERATORS`	1	Defined as an integer value and expands to 1 if, when you apply the `arm_sve_vector_bits` attribute to an SVE vector type, the attribute creates a type that supports the GNU vector extensions.
`__ARM_FEATURE_SVE2`	1	Defined as an integer value and expands to 1 if the implementation generates code for an SVE2 target, and that all the base SVE2 functions are available.
`__ARM_FEATURE_SVE2_AES`	1	Defined as an integer value and expands to 1 if all the AES-128 functions are available.

Macro	Value	Purpose
`__ARM_FEATURE_SVE2_BITPERM`	1	Defined as an integer value and expands to 1 if all the bit permutation functions are available.
`__ARM_FEATURE_SVE2_SHA3`	1	Defined as an integer value and expands to 1 if all the SHA-3 functions are available.
`__ARM_FEATURE_SVE2_SM4`	1	Defined as an integer value and expands to 1 if all the SM4 functions are available.

> **Note**
>
> A full list of the supported ACLE for SVE (and SVE2) predefined macros, that includes more detailed descriptions of the macros and their dependencies, is available in the ACLE for SVE specification.

4.6 Vector routines support

This section describes how to vectorize loops in C and C++ workloads that invoke the math routines from `libm`, and how to interface custom vector functions with serial code.

4.6.1 How to vectorize math routines in Arm C/C++ Compiler

Arm® C/C++ Compiler supports the vectorization of loops within C workloads that invoke the math routines from `libm`.

Any C loop-using functions from `<math.h>` can be vectorized by invoking the compiler with the option `-fsimdmath` with one of the optimization level options that activate the auto-vectorizer: `-O2`, `-O3`, or `-Ofast`.

Examples

The following examples show loops with math function calls that can be vectorized by invoking the compiler with:

```
armclang -c -fsimdmath -O2 c-vectorize-math-sin.c
```

C example with loop invoking `sin`:

```
#include <math.h>
void do_something(double * a, double * b, unsigned N) {
    for (unsigned i = 0; i < N; ++i) {
        // some computation
        a[i] = sin(b[i]);
        // some computation
    }
}
```

How it works

Arm C/C++ Compiler contains `libamath`, a library with SIMD implementations of the routines that are provided by `libm`, along with a `math.h` file that declares the availability of these SIMD functions to the compiler.

During loop vectorization, the compiler is aware of these vectorized routines, and can replace a call to a scalar function (for example, a double-precision call to `sin`) with a call to a `libamath` function that takes a vector of double-precision arguments, and returns a result vector of doubles.

The `libamath` library is built using the fastest implementations of scalar and vector functions from the following Open Source projects:

- Arm Optimized Routines
- SLEEF
- PGMath

Limitations

This is an experimental feature which can sometimes lead to performance degradations. Arm encourages users to test the applicability of this feature on their non-production code, and will address any possible inefficiency in a future release.

Related information

SLEEF
Arm Optimized Routines
PGMath
Vector function ABI specification for AArch64

4.6.2 How to declare custom vector routines in Arm C/C++ Compiler

To vectorize loops that invoke serial functions, `armclang` can interface with user-provided vector functions.

To expose the vector functions available to the compiler, use the `#pragma omp declare variant` directive on the scalar function declaration or definition.

The following examples show the basic functionality. In the examples, the `omp declare variant` pragma declares a vector variant of `foo()`, called `myneon_foo` (Advanced SIMD) or `mysve_foo()` (SVE). In each example, `foo()` is called by `do_something()`, which, if vectorized by the compiler, allows the call to be replaced by calls to `myneon_foo()` or `mysve_foo()`.

For Advanced SIMD vectorization, the example is:

```
// declarations or definitions visible at compile time in myvecroutines.h
#include <arm_neon.h>
int32x2_t neon_foo(float64x2_t);
#pragma omp declare variant(neon_foo) \
        match(construct = {simd(simdlen(2), notinbranch)}, \
                device = {isa("simd")})
```

```
int foo(double);
// loop in the user code, in user_code.c
#include "path/to/myvecroutines.h"void do_something(int * a, double * b, unsigned N)
 {
     for (unsigned i = 0; i < N; ++i)
         a[i] = foo(b[i]);
}
```

To compile the example code (`c-custom-simd-src.c`) with automatic loop optimization enabled, invoke `armclang` with the `-fopenmp` (or `-fopenmp-simd`) and `-O2` (or higher) optimization options:

> **Note**
>
> Automatic loop vectorization is enabled at the `-O2`, `O3`, and `-Ofast` optimization levels.

```
armclang -c -fopenmp -O2 c-custom-simd-testing.c
```

For SVE vectorization, the example (`c-custom-sve-src.c`) is:

```
// declarations or definitions visible at compile time in myvecroutines.h
#include <arm_sve.h>
svint32_t sve_foo(svfloat64_t, svbool_t);
#pragma omp declare variant(sve_foo) \
        match(construct = {simd(notinbranch)}, \
                device = {isa("sve")}, \
                implementation = {extension("scalable")})
int foo(double);
// loop in the user code, in user_code.c
#include "path/to/myvecroutines.h"void do_something(int * a, double * b, unsigned N)
 {
     for (unsigned i = 0; i < N; ++i)
     a[i] = foo(b[i]);
}
```

To compile the SVE code, again invoke `armclang` with the `-fopenmp` and `-O2` (or higher) optimization options:

```
armclang -c -fopenmp -O2 -march=armv8.2-a+sve c-custom-sve-testing.c
```

> **Note**
>
> `-march=armv8-a+sve` tells the compiler to generate code for an SVE-enabled Arm®v8-A -based system:
>
> - If you are compiling and running on an SVE-enabled system, you can replace `-march=armv8-a+sve` with `-mcpu=native`.
>
> - If you are compiling and running on a system that is not SVE-enabled, SVE binaries that are compiled with `-march=armv8-a+sve` can be emulated with Arm Instruction Emulator. For more information, see Arm Instruction Emulator.

> **Note**
> If you are trying either of these examples on your system, you must link the output object file against an object file or library that provides the `myneon_foo` or `mysve_foo` symbols.

The vector function that is associated to the scalar function must have a signature that obeys to the rules of the chapter on **USER DEFINED VECTOR FUNCTIONS** of the Vector Function Application Binary Interface (VFABI) Specification for AArch64. The rules are summarized in section **Mapping rules**.

`declare variant` support

For a complete description of '*declare variant*', refer to the OpenMP 5.0 specifications.

The current level of support covers the following features:

- OpenMP 5.0 `declare variant`, for the `simd` trait of the `construct` trait set.

> **Note**
> There is no support for the following clauses in the `simd` trait of the `construct` set:
>
> ◦   `uniform`
>
> ◦   `aligned`
>
> The `linear` clause in the `simd` trait is only supported for pointers with a linear step of 1. There is no support for linear modifiers.

For VFABI specifications, there is support for the following features:

- `simdlen(N)` is supported when targeting Advanced SIMD vectorization. Its value must be a power of 2 so that the `WDS(f) x N` is either 8 or 16.

  `f` is the name of the scalar function the directive applies to. For a definition of `WDS(f)`, refer to the VFABI.

> **Note**
> To ensure the vector `w` function obeys the AAVPCS defined in the VFABI, you must explicitly mark the function with `__attribute__((aarch64_vector_pcs))`.

- To allow scalable vectorization when targeting SVE, you must omit the `simdlen` clause, and you must specify the implementation trait extension `extension("scalable")`.

- The supported scalar function signature in C and C++ are in the forms:

  1. `void (Ty1, Ty2,..., TyN)`

  2. `Ty1 (Ty2, Ty3,..., TyN)`

  where `Ty#n` are:

  1. Any of the integral type values of size 1, 2, 4, or 8 (in bytes), signed and unsigned.

2. Floating-point type values of half, single or double-precision.

3. Pointers to any of the previous types.

There is no support for variadic functions or C++ templates.

Mapping rules

Common mapping rules

1. Each parameter and the return value of the scalar function, maps to a correspondent parameter and return value in the vector signature, in the same order.

2. A parameter that is marked with `linear` is left unchanged in the vector signature.

3. The `void` return type is left unchanged in the vector signature.

Mapping rules for Advanced SIMD

1. Each parameter type `Ty#n` maps to the correspondent Neon® ACLE type `<Ty#n>x<N>_t`, where `N` is the value that is specified in the `simdlen(N)` clause. Values of `N` that do not correspond to NEON ACLE types are unsupported.

2. If you specify `inbranch`, an extra `mask` parameter is added as the last parameter of the vector signature. The type of the parameter is the NEON ACLE type `uint<BITS>x<N>_t`, where:

   a. `N` is the value that is specified in the `simdlen(N)` clause.

   b. `BITS` is the size (in bits) of the *Narrowest Data Size (NDS)* associated to the scalar function, as defined in the VFABI.

   c. To select active or inactive lanes, set all bits to 1 (active) or 0 (inactive) in the corresponding `uint<BITS>_t` integer in the mask vector.

Mapping rules for SVE

1. Each parameter type `Ty#n` is mapped to the correspondent SVE ACLE type `sv<Ty#n>_t`.

2. An extra *mask* parameter of type `svbool_t` is always added to the signature of the vector function, whether `inbranch` or `notinbranch` is used. Active and inactive lanes of the mask are set as described in the section **SVE Masking** of the VFABI:

   "The logical lane subdivision of the predicate corresponds to the lane subdivision of the vector data type generated for the *Widest Data Type (WDS)*, with one bit in the predicate lane for each byte of the data lane. Active logical lanes of the predicate have the least significant bit set to 1, and the rest set to zero. The bits of the inactive logical lanes of the predicate are set to zero."

   For example, in the function `svfloat64_t F(svfloat32_t vx, svbool_t)`, the `WDS` is 8, therefore the lane subdivision of the mask is 8-bit. Active lanes are set by the bit sequence `00000001`, inactive lanes are set with `00000000`.

Example: Vectorizing with the custom user vector function

The following examples show you how to vectorize with the custom user vector function. The examples use:

- `-O2` to enable the minimal level of optimizations required for the loop auto-vectorization process.

- `-fopenmp` to enable the parsing of the OpenMP directives.

> **Note**
> - The same functionality for `declare variant` can also be achieved with `-fopenmp-simd`.
> - `-mllvm -force-vector-interleave=1` simplifies the output and can be omitted for regular compiler invocations.

The code in these examples has been produced by Arm Compiler for Linux 20.0.

For both Advanced SIMD and SVE, the `linear` clause can improve the vectorization of functions accessing memory through contiguous pointers. For example, in the function `double sincos(double, double *, double *)`, the memory pointed to by the pointer parameters is contiguous across loop iterations. To improve the vectorization of this function, use the `linear` clause:

```
#include <arm_sve.h>
void CustomSinCos(svfloat64_t, double *, double *);
#pragma omp declare variant(CustomSinCos) \
       match(construct = {simd(notinbranch, linear(sinp), linear(cosp))}, \
              device = {isa("sve")}, \
              implementation = {extension("scalable")})
double sincos(double in, double *sinp, double *cosp);
void f(double *in, double *sin, double *cos, unsigned N) {
    for (unsigned i = 0; i < N; ++i)
        sincos(in[i], &sin[i], &cos[i]);
}
```

Example: Advanced SIMD

Simple:

```
// filename: c-custom-vec-routines-example01.c
#include <arm_neon.h>
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo(float64x2_t a);
#pragma omp declare variant(user_vector_foo) \
       match(construct = {simd(simdlen(2), notinbranch)}, \
              device = {isa("simd")})
double foo(double);
void do_something(double * restrict a, double * b, unsigned N) {
    for (unsigned i = 0; i < N; ++i)
        a[i] = foo(b[i]);
}
```

To produce a vector loop that invokes `user_vector_foo`, compile the example code with:

```
 armclang -c -fopenmp -O2 -mllvm -force-vector-interleave=1 -S c-custom-vec-
routines-example01.c
```

```
//...
.LBB0_4:                                      // =>This Inner Loop Header: Depth=1
    ldr       q0, [x25], #16
    bl        user_vector_foo
    subs      x23, x23, #2              // =2
```

```
    str      q0, [x24], #16
    b.ne     .LBB0_4
```

With `linear`:

```
// filename: c-custom-vec-routines-example02.c
#include <arm_neon.h>
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo_linear(float64x2_t,
  float *);
#pragma omp declare variant(user_vector_foo_linear) \
    match(construct = {simd(simdlen(2), notinbranch, linear(b))}, \
          device = {isa("simd")})
double foo_linear(double a, float* b);
void do_something_linear(double * restrict a, double * b, float * x, unsigned N) {
    for (unsigned i = 0; i < N; ++i)
    a[i] = foo_linear(b[i], &x[i]);
}
```

To produce a vector loop that invokes `user_vector_foo_linear`, compile the code with:

```
 armclang -c -fopenmp -O2 -mllvm -force-vector-interleave=1 -S c-custom-vec-
routines-example02.c
```

```
.LBB0_4:                                 // =>This Inner Loop Header: Depth=1
    str    q1, [sp, #32]           // 16-byte Folded Spill
    ldr    q0, [x26], #16
    ldp    q2, q1, [sp, #16]       // 32-byte Folded Reload
    shl    v1.2d, v1.2d, #2
    add    v1.2d, v2.2d, v1.2d
    fmov   x0, d1
    bl     user_vector_foo_linear
    ldr    q1, [sp, #32]           // 16-byte Folded Reload
    str    q0, [x25], #16
    ldr    q0, [sp]                // 16-byte Folded Reload
    subs   x24, x24, #2            // =2
    add    v1.2d, v1.2d, v0.2d
    b.ne   .LBB0_4
```

Example: SVE

Simple:

```
// filename: c-custom-vec-routines-example03.c
#include <arm_sve.h>
svfloat16_t user_vector_foo_sve(svfloat64_t a, svbool_t mask);
#pragma omp declare variant(user_vector_foo_sve) \
      match(construct = {simd(notinbranch)}, \
      device = {isa("sve")}, \
      implementation = {extension("scalable")})
float16_t foo(double);
void do_something(float16_t * restrict a, double * b, unsigned N) {
    for (unsigned i = 0; i < N; ++i)
        a[i] = foo(b[i]);
}
```

Compile the code with:

```
armclang -c -fopenmp -O2 -march=armv8.2-a+sve -S c-custom-vec-routines-example03.c
```

```
.LBB0_2:                                    // %vector.body
                                            // =>This Inner Loop Header: Depth=1
    ld1d    { z0.d }, p4/z, [x19, x21, lsl #3]
    mov     p0.b, p4.b
    bl      user_vector_foo_sve
    st1h    { z0.d }, p4, [x20, x21, lsl #1]
    incd    x21
    whilelo p4.d, x21, x22
    b.mi    .LBB0_2
```

With `linear`:

```
// filename: c-custom-vec-routines-example04.c
#include <arm_sve.h>
svfloat64_t user_vector_foo_linear_sve(svfloat64_t, float *, svbool_t);
#pragma omp declare variant(user_vector_foo_linear_sve) \
        match(construct = {simd(notinbranch, linear(b))}, \
              device = {isa("sve")}, \
              implementation = {extension("scalable")})
double foo_linear(double a, float* b);
void do_something_linear(double * restrict a, double * b, float * x, unsigned N) {
    for (unsigned i = 0; i < N; ++i)
        a[i] = foo_linear(b[i], &x[i]);
}
```

To generate an invocation to the user vector function `user_vector_foo_linear` in the vector loop, compile the code with:

```
armclang -c -fopenmp -O2 -march=armv8.2-a+sve -S c-custom-vec-routines-example04.c
```

```
.LBB0_2:                                    // %vector.body
                                            // =>This Inner Loop Header: Depth=1
    ld1d    { z0.d }, p4/z, [x20, x22, lsl #3]
    add     x0, x19, x22, lsl #2
    mov     p0.b, p4.b
    bl      user_vector_foo_linear_sve
    st1d    { z0.d }, p4, [x21, x22, lsl #3]
    incd    x22
    whilelo p4.d, x22, x23
    b.mi    .LBB0_2
```

4.7 Link Time Optimization (LTO)

This section describes what Link Time Optimization (LTO) is, when LTO is useful, and how to compile with LTO. The section also provides reference information about the `llvm-ar` and `llvm-ranlib` LLVM utilities that are required to compile static libraries with LTO.

4.7.1 What is Link Time Optimization (LTO)

Link Time Optimization is a form of interprocedural optimization that is performed at the time of linking application code. Without LTO, Arm® Compiler for Linux compiles and optimizes each source file independently of one another, then links them to form the executable. With LTO, Arm Compiler for Linux can process, consume, and use inter-module dependency information from across all the source files to enable further optimizations at link time. LTO is particularly useful when source files that have already been compiled separately.

The following describes the workflow that Arm Compiler for Linux takes with and without LTO enabled, in more detail:

- Without LTO:

  1. Source files are translated into separate ELF object files (`.o`) and passed to the linker.

  2. The linker processes the separate ELF object files, together with library code, to create the ELF executable.

- With LTO:

  1. Source files are translated into a bitcode object files (`.o`), and passed to the linker. LLVM Bitcode is an intermediate form of code that is understood by the optimizer.

  2. To extract the module dependency information, the linker processes the bitcode and object files together and passes them to the LLVM optimizer utility, libLTO.

  3. The LLVM optimizer utility, libLTO, uses the module dependency information to filter out unused modules, and create a single highly optimized ELF object file. Additional optimizations are possible by knowing the module dependency information. The new ELF object file is returned to the linker.

  4. The linker links the new ELF object file with the remaining ELF object files and library code, to generate an ELF executable.

Limitations

LTO in Arm Compiler for Linux has some limitations:

- To compile static libraries, you must create a library archive file that libLTO can use at link time. `armllvm-ar`, as well as some open-source utility tools can create this archive file. For more information about `armllvm-ar`, see armllvm-ar and reference.

- Partial linking is not supported with LTO because partial linking only works with ELF objects, rather than bitcode files.

- If your library code calls a function that was defined in the source code, but is removed by libLTO, you might get linking errors.

- Bitcode objects are not guaranteed to be compatible across Arm Compiler for Linux versions. When linking with LTO, ensure that all your bitcode files are built using the same version of the compiler.

- You can not analyze LTO-optimized code using Arm Optimization Reports. Arm Optimization Reports analyzes object files that are generated by Arm Compiler for Linux before they are passed to the linker. Therefore, you can not use Arm Optimization Reports to investigate the vectorization decisions that LTO enables the linker to make.

4.7.2 Compile with Link Time Optimization (LTO)

This topic describes how to compile your C/C++ source code with Link Time Optimization (LTO), using Arm® C/C++ Compiler.

Before you begin

- Download and install Arm Compiler for Linux. You can download Arm Compiler for Linux from the download page. Learn how to install and configure Arm Compiler for Linux, using the installation instructions on the Arm Developer website.

- Load the environment module for Arm Compiler for Linux for your system.

- To compile your code with static libraries, you must create an archive of your libraries using an archive utility tool. Arm Compiler for Linux version 20.3+ includes variants of the LLVM archive utility tools `llvm-ar` (`armllvm-ar`) and `llvm-ranlib` (`armllvmran-lib`).

  If you use a Makefile to create the library archive and compile your application, open your Makefile and update any references of `llvm-ar` to `armllvm-ar`, and `llvm-ranlib` to `armllvm-ranlib`.

  Note If you use `ar` to create your archives, you must also use the LLVM Gold Plugin to enable `ar` to use LLVM bitcode object files. For more information, see the LLVM gold plugin documentation.

  For more information about `armllvm-ar`, see armllvm-ar and reference. For more information about `armllvm-ranlib`, see armllvm-ranlib reference.

Procedure

1. To generate an executable binary with LTO enabled, compile and link your code with `armclang|armclang++`, and pass the `-flto` option:

   - For dynamic library compilation, use:

     ```
     {armclang|armclang++} -O<level> -flto -o <binary> <sources>
     ```

   - For static library compilation:

     a. Compile, but do not link, your code with LTO:

     ```
     {armclang|armclang++} -O<level> -flto -c <sources>
     ```

     The result is one or more `.o` files, one per source file that was passed to `armclang|armclang++`.

     b. Create the archive file for your static library object files:

     ```
     armllvm-ar [config-options] [operation{modifiers)}] <archive> [<files>]
     armllvm-ranlib <archive>
     ```

For example:

```
armllvm-ar rc example-archive.a source1.o source2.o
armllvm-ranlib example-archive.a
```

`armllvm-ar` builds a single archive file from one or more `.o` files. `r` is an operation that instructs `armllvm-ar` to replace existing archive files or, if they are new files, add the files to the end of the archive. `c` is a modifier to `r` that disables the warning which informs you that an archive has been created.

`armllvm-ranlib` builds an index for the `<archive>` file.

For a more detailed description of `armllvm-ar`, see armllvm-ar and reference. For a more detailed description of `armllvm-ar`, see armllvm-ranlib reference.

c. Link your remaining object files together with your archive file:

```
{armclang|armclang++} -O<level> -flto -o <binary> <sources>.o <archive>
```

> **Note**
> The `<archive>` file is used in place of the object files that where combined into the `<archive>` file by `armllvm-ar`.

2. (Optional) Use a tool like objdump to analyze the binary and view how the compiler optimized your code:

```
objdump -d <binary>
```

Results

Arm C/C++ Compiler builds your LTO-optimized binary `<binary>`.

To run your binary, use:

```
./<binary>
```

Example 4-1: Example: Compare code compiled with and without LTO

The following example application code is composed of two source files. `c-lto-main.c` contains the `main` function which calls and a second function, `foo`, contained in `c-lto-foo.c`. Compiling and analyzing example code without LTO enabled, then with LTO enabled, allows us to see the effect that LTO has on the application compilation.

1. Create the example source code files:

    a. Write and save the following code as a `c-lto-main.c` source file:

    ```
    #include <stdio.h>
    #include <stdlib.h>
    extern double foo(double);
    ```

```
int main(int argc, char *argv[]) {
    // Expected command line:
    if (argc != 3) {
        fprintf(stderr, "Incorrect arguments.");
        fprintf(stderr, " Usage: %s <filename> <size>", argv[0]);
        exit(1);
    }
    char *filename = argv[1];
    int numelts = atoi(argv[2]);
    FILE *file = fopen(filename, "rw");
    // Read in some binary data
    double *data = (double*)malloc(numelts * sizeof(double));
    fread(data, sizeof(double), numelts, file);
    // Do 'something' to the data
    for (int i = 0; i < numelts; i++)
        data[i] = foo(data[i]);
    // Overwrite the file.
    rewind(file);
    fwrite(data, sizeof(double), numelts, file);
    fclose(file);
    free(data);
    return EXIT_SUCCESS;
}
```

b. Write and save the following code as a `c-lto-foo.c` source file:

```
double foo(double val) {
    return val * 2.0;
}
```

2. Use `armclang` to compile the code both without and with LTO enabled:

a. To compile without LTO, into a binary called `binary-no-lto`, use:

```
armclang -O3 -o binary-no-lto c-lto-main.c c-lto-foo.c
```

b. To compile with LTO, into a binary called `binary-lto`, use:

```
armclang -O3 -flto -o binary-no-lto c-lto-main.c c-lto-foo.c
```

3. To analyze the files to see the effect that LTO has on the generated code, use `objdump` to investigate the `main` function in the binary:

```
objdump -d binary-no-lto
```

In the following pseudo code:

- `{addr*}` represents an address. `{addr_main}`, `{addr_foo}`, and `{addr_loop_start}` are addresses that are given specific pseudo address names for the purpose of this example.

- {enc} represents the encoding.

For `binary-no-lto`, you can see separate functions `main` and `foo` in the following pseudo code:

```
...
{addr_main} <main>:
    ...
    {addr*}:        {enc}       ldr     d0, [x23]
```

```
    {addr*}:          {enc}      bl      {addr_foo} <foo>
    {addr*}:          {enc}      subs    x22, x22, #0x1
    {addr*}:          {enc}      str     d0, [x23], #8
    {addr*}:          {enc}      b.ne    {addr_main}
    ...
...
{addr_foo} <foo>:
    {addr*}:          {enc}      fadd    d0, d0, d0
    {addr*}:          {enc}      ret
```

`main` has a scalar loop with a branch to `foo` in it:

```
{addr*}:          {enc}          bl        {addr_foo} <foo>
```

Whereas in `binary-lto`, you see one `main` function:

```
...
{addr} <main>:
    ...
    {addr_loop_start}:    {enc}      ldr     q0, [x12], #16
    {addr*}:              {enc}      subs    x11, x11, #0x2
    {addr*}:              {enc}      fadd    v0.2d, v0.2d, v0.2d
    {addr*}:              {enc}      str     q0, [x13]
    {addr*}:              {enc}      mov     x13, x12
    {addr*}:              {enc}      b.ne    {addr_loop_start}
    ...
...
```

In `main` in `binary-lto`, the simple `foo` function has been inlined and transformed into a vectorized loop: `fadd v0.2d, v0.2d, v0.2d`.

Related information

4.7.3 armllvm-ar and reference

This topic describes `armllvm-ar`. `armllvm-ar` is a utility tool provided in the Arm® Compiler for Linux package, and is a variant of the LLVM `llvm-ar` utility tool.

`armllvm-ar` is an archiving tool that is similar to the Unix utility `ar`. However, unlike `ar`, `armllvm-ar` is able to understand the LLVM bitcode files that LLVM-based compilers produce when Link Time Optimization (LTO) is enabled.

`armllvm-ar` can archive several `.o` object (or bitcode object) files into a single archive library. As `armllvm-ar` archives the files, the tool creates a symbol table of the files. At link time, you can pass the archive to the compiler to link it into your application. When an archive is used by the compiler at link time, the symbol table enables linking to be performed faster than it would take the linker to link each file separately.

> **Note**
>
> For information about how `llvm-ar` differs from `ar`, see the llvm-ar LLVM command documentation.

Syntax

`armllvm-ar` can be run on the command line or through a Machine Readable Instruction (MRI) script. The following syntax is the command line syntax

```
armllvm-ar [config-options] [operation{modifiers)}] <archive> [<files>]
```

> **Note**
>
> `armllvm-ar` inherits the same syntax as `llvm-ar`.

Options for `armllvm-ar` are separated into *Configuration options*, *Operations*, and *Modifiers*:

- Configuration options are options that either configure how `llvm-ar` runs (for example how to set the default archive format), or are options to display help or version information.
- Operations are actions that are performed on an archive. You can only pass one operation to `armllvm-ar`.
- Modifiers control how the operation completes the action. You can specify multiple modifiers to an operation, however, each operation supports different modifiers.

Options, Operations, and Modifiers

`armllvm-ar` supports the same options, operations, and modifiers that are supported by LLVM's `llvm-ar` tool. To see the options, operations, and modifiers that are supported by both utility tools, see the LLVM llvm-ar reference documentation.

Outputs

A successful run of `armllvm-ar` returns `0` and creates an archive called <archive>, which normally has a `.a` suffix. A nonzero return value indicates an error.

Related information

armllvm-ranlib reference on page 63

4.7.4 armllvm-ranlib reference

This topic describes `armllvm-ranlib`. `armllvm-ranlib` is a utility tool provided in the Arm® Compiler for Linux package, and is a variant of the LLVM `llvm-ranlib` utility tool.

Like, `llvm-ranlib` is a synonym to the LLVM archiver tool `llvm-ar -s`, `armllvm-ranlib` is a synonym for running `armllvm-ar -s`.

> **Note**
> For a full description of `llvm-ranlib` see the llvm-ranlib LLVM command documentation.

4.8 Profile Guided Optimization (PGO)

Learn about Profile Guided Optimization (PGO) and how to use `llvm-profdata`. `llvm-profdata` is LLVM's utility tool for profiling data and displaying profile counter and function information. `llvm-profdata` is included in Arm® Compiler for Linux.

Profile Guided Optimization (PGO) is a technique where you use profiling information to improve application run-time performance. To use PGO, you must generate profile information from an application, then recompile the application code while passing profile information to the compiler. The compiler can interpret and use the profile information to make informed optimization decisions. For example, when the compiler knows the frequency of a function call in an applications code, it can help the compiler make inlining decisions.

To enable the compiler to make the best optimization decisions for your applications code, you must pass profiling data that is representative of the applications typical workload. To generate profiling information that is representative of a typical workload, compile your application with your typical compiler options and run the application as you typically would.

The profile information can be generated from either:

- A sampling profiler
- An instrumented version of the code.

LLVM's documentation describes both methods. In this section, we only describe how to:

- Generate profile information from an instrumented version of the application code.
- Use `llvm-profdata` to combine and convert profile information from instrumented code into a format that the compiler can read as an input.

4.8.1 How to compile with Profile Guided Optimization (PGO)

Learn how to use Profile Guided Optimization (PGO) with Arm® C/C++ Compiler.

Before you begin

- Download and install Arm Compiler for Linux.
- Load the Arm Compiler for Linux environment module for your system.
- Add the `llvm-bin` directory to your `PATH`. For example:

```
PATH=$PATH:<install-dir>/../llvm-bin
```

Where `<install-dir>` is the Arm Compiler for Linux install location.

> **Note**
> To obtain `<install-dir>` for your system, load the Arm Compiler for Linux environment module and run `which armclang`. The returned path is your `<install-dir>`.

About this task

> **Note**
> The following procedure describes how to generate, and use, profile data using Arm Compiler for Linux. Profile data files generated by GCC compilers cannot be used by Arm Compiler for Linux.

Procedure

1. Build an instrumented version of your application code. Compile your application with the `-fprofile-instr-generate` option:

```
armclang -O<level> [options] -fprofile-instr-generate=<profdata_file>.profraw
  <source>.c -o <binary>
```

> - For good optimization, use `-O2` optimization level or higher.
> - To ensure that the instrumented executable represents the real executable, compile your application code with the same compiler options.
> - By default, if you do not specify a `<profdata_file>.profraw`, when you compile the application with `-fprofile-instr-generate`, the profile data is written to `default.profraw`.
>
> **Note**
> To change this behavior, either specify `<profdata_file>.profraw` on the compile line, or set the `LLVM_PROFILE_FILE="<profdata_file>.profraw"` when you run your application (see next step). Both `-fprofile-instr-generate` and `LLVM_PROFILE_FILE` can use the following modifiers to uniquely set profile data filename:
> - `%p` to state the process ID
> - `%h` to state the hostname
> - `%m` to state the unique profile name.
>
> For example, `LLVM_PROFILE_FILE="example-{%p|%h|%m}.profraw"`.
>
> If both `-fprofile-instr-generate` and `LLVM_PROFILE_FILE` are set, `LLVM_PROFILE_FILE` takes priority.

2. Run your application code with a typical workload. Either:

- Run it with default behavior:

```
./<binary>
```

  The profile data is written to the profile data file specified in the previous step, or if no file was specified, to `default.profraw`.

- Run the application and specify a new filename for the `.profraw` file using the `LLVM_PROFILE_FILE` environment variable:

```
LLVM_PROFILE_FILE=<profdata_file>.profraw ./<binary>
```

  The profile data is written to `<profdata_file>.profraw`.

3. Combine and convert your `.profraw` files into a single processed `.profdata` file using the `llvm-profdata` tool `merge` command:

- If you have a single `.profraw` file, use:

```
llvm-profdata merge -output=<profdata_file>.profdata <file>.profraw
```

> **Note**
>
> Where you only have one `.profraw` file, no files are combined, however, you must still run the `merge` command to convert the file format.

- If you have multiple `.profraw` files, you can combine and convert them into a single profile data file, `.profdata`, by either:
    - Passing each `.profraw` file in separately:

    ```
    llvm-profdata merge -output=<outfile>.profdata <filename1>
      [<filename2> ...]
    ```

    - Passing in all the `.profraw` files in a directory:

    ```
    llvm-profdata merge -output=<outfile>.profdata *.profraw
    ```

4. Recompile your application code and pass the profile data file, `<outfile>.profdata`, to `armclang` using the `-fprofile-instr-use=<outfile>.profdata` option:

```
armclang -O<level> -fprofile-instr-use=<outfile>.profdata <source>.c -o <binary>
```

  This step can be repeated without having to regenerate a new profile data file. However, as compilation decisions change and change the output application code, `armclang` might get to a point where the profile data can no longer be used. At this point, `armclang` will output a warning.

Example 4-2: Example: Compiling code with PGO

This example uses 'foo.c' as the source code file and 'foo-binary'.

1. Build an instrumented version of the `foo-binary` application code:

```
armclang -O2 -fprofile-instr-generate foo.c -o foo-binary
```

2. Run `foo-binary` with a typical workload twice, creating separate `.profraw` files using their process ID to distinguish them:

```
LLVM_PROFILE_FILE="foorun-%p.profraw"
./foo-binary
./foo-binary
```

3. Combine and convert the `.profraw` files into a single processed `.profdata` file:

```
llvm-profdata merge -output=foorun.profdata foorun-*.profraw
```

4. Recompile the `foo-binary` application code passing the `foorun.profdata` profile data file to `armclang`:

```
armclang -O2 -fprofile-instr-use=foorun.profdata foo.c -o foo-binary
```

Related information

LLVM's documentation
LLVM Command Guide

4.8.2 llvm-profdata reference

This topic describes the commands and lists the options for the `llvm-profdata` tool, for instrumentation-built profile data.

Note

Full documentation for the `llvm-profdata` is available online in the LLVM Command Guide.

In Arm® Compiler for Linux, the `llvm-profdata` tool is located in `<install_dir>/arm-linux-compiler-*/llvm-bin`. To enable the `llvm-profdata` tool, add the `llvm-bin` directory to your `PATH`.

`llvm-profdata` accepts three commands: `merge`, `show`, and `overlap`. The following table describes each.

Table 4-4: Describes the commands for `llvm-profdata`

Command	Syntax	Description	Common options
merge	`llvm-profdata merge -instr [options] [filename1] {[filename2] ...}`	merge combines multiple, instrumentation-built, profile data files into a single, indexed, profile data file.	• `-weighted-files=<weight>,<filename>` • `-input-files=<path>` • `-sparse=true|false` • `-num-threads=<value>` • `-prof-sym-list=<path>` • `-compress-all-sections=true|false`
show	`llvm-profdata show -instr [options] [filename]`	show displays profile counter and (optional) function information for a profile data file.	• `-all-functions` • `-counts` • `-function=<string>` • `-text` • `-topn=<value>` • `-memop-sizes` • `-list-below-cutoff` • `-showcs`
overlap	`llvm-profdata overlap [options] [base profile] [test profile]`	overlap displays the overlap of profile counter information for two profile data files or, optionally, for any functions that match a given string (`<string>`).	• `-function=<string>` • `-value-cutoff=<value>` • `-cs`

Global options that all of the commands accept include:

- `-help`
- `-output=<filename>`

Related information
LLVM Command Guide

4.9 Arm Optimization Report

Arm Optimization Report builds on the llvm-opt-report tool available in open source LLVM. Arm Optimization Report shows you the optimization decisions that the compiler is making, in-line with your source code, enabling you to better understand the unrolling, vectorization, and interleaving behavior.

Unrolling

Example questions: Was a loop unrolled? If so, what was the unroll factor?

Unrolling is when a scalar loop is transformed to perform multiple iterations at once, but still as scalar instructions.

The unroll factor is the number of iterations of the original loop that are performed at once. Sometimes, loops with known small iteration counts are completely unrolled, such that no loop structure remains. In completely unrolled cases, the unroll factor is the total scalar iteration count.

Vectorization

Example questions: Was a loop vectorized? If so, what was the vectorization factor?

Vectorization is when multiple iterations of a scalar loop are replaced by a single iteration of vector instructions.

The vectorization factor is the number of lanes in the vector unit, and corresponds to the number of scalar iterations that are performed by each vector instruction.

Note

The true vectorization factor is unknown at compile time for SVE, because SVE supports scalable vectors.

When SVE is enabled, Arm Optimization Report reports a vectorization factor that corresponds to a 128-bit SVE implementation.

If you are working with an SVE implementation with a larger vector width (for example, 256 bits or 512 bits), the number of scalar iterations that are performed by each vector instruction increases proportionally.

```
SVE scaling factor = <true SVE vector width> / 128
```

Loops vectorized using scalable vectors are annotated with vs<F,I>. For more information, see arm-opt-report reference.

Interleaving

Example question: What was the interleave count?

Interleaving is a combination of vectorization followed by unrolling; multiple streams of vector instructions are performed in each iteration of the loop.

The combination of vectorization and unrolling information tells you how many iterations of the original scalar loop are performed in each iteration of the generated code.

```
Number of scalar iterations = <unroll factor> x <vectorization factor> x <interleave
  count> x <SVE scaling factor>
```

Note

The number of scalar iterations is not an exact figure. For SVE code, the compiler can use the predication capabilities of SVE. For example, a 10-iteration scalar operation on 64-bit values takes 3 iterations on a 256-bit SVE-enabled target.

Reference

The annotations Arm Optimization Report uses to annotate the source code, and the options that can be passed to `arm-opt-report` are described in **arm-opt-report reference**.

4.9.1 How to use Arm Optimization Report

This topic describes how to use Arm Optimization Report.

Before you begin

Download and install Arm® Compiler for Linux. For more information, see Download Arm Compiler for Linux and Installation.

Procedure

1. To generate a machine-readable `.opt.yaml` report, at compile time add `-fsave-optimization-record` to your command line.
    A `<filename>.opt.yaml` report is generated by Arm C/C++/Fortran Compiler, where `<filename>` is the name of the binary.

2. To inspect the `<filename>.opt.yaml` report, as augmented source code, use `arm-opt-report`:

    ```
    arm-opt-report <filename>.opt.yaml
    ```

    Annotated source code appears in the terminal.

Example 4-3: Example

1. Create an example file called `example.c` containing the following code:

    ```
    void bar();
    void foo() { bar(); }
    void Test(int *res, int *c, int *d, int *p, int n) {
    int i;
    #pragma clang loop vectorize(assume_safety)
    for (i = 0; i < 1600; i++) {
        res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
    }
    for (i = 0; i < 16; i++) {
        res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
    }
    foo();
    foo(); bar(); foo();
    }
    ```

2. Compile the file, for example to a shared object `example.o`:

    ```
    armclang -O3 -fsave-optimization-record -c -o example.o example.c
    ```

    This generates a file, `example.opt.yaml`, in the same directory as the built object.

    For compilations that create multiple object files, there is a report for each build object.

> **Note**
>
> This example compiles to a shared object, however, you could also compile to a static object or to a binary.

3. View the `example.opt.yaml` file using `arm-opt-report`:

```
arm-opt-report example.opt.yaml
```

Annotated source code is displayed in the terminal:

```
< example.c
  1          | void bar();
  2          | void foo() { bar(); }
  3          |
  4          | void Test(int *res, int *c, int *d, int *p, int n) {
  5          |    int i;
  6          |
  7          |    #pragma clang loop vectorize(assume_safety)
  8     V4,1 |    for (i = 0; i < 1600; i++) {
  9          |       res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
 10          |    }
 11          |
 12  U16     |    for (i = 0; i < 16; i++) {
 13          |       res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
 14          |    }
 15          |
 16 I        |    foo();
 17          |
 18          |    foo(); bar(); foo();
    I        |    ^
    I        |                     ^
 19          | }
```

The example Arm Optimization Report output can be interpreted as follows:

- The `for` loop on line 8:
  - Is vectorized
  - Has a vectorization factor of four (there are four 32-bit integer lanes)
  - Has an interleave factor of one (so there is no interleaving)
- The `for` loop on line 12 wis unrolled 16 times. This means it is completely unrolled, with no remaining loops.
- All three instances of `foo()` are inlined

Related information

Arm Compiler for Linux
Help and tutorials

4.9.2 arm-opt-report reference

This reference topic describes the options that are available for `arm-opt-report`. The topic also describes the annotations that `arm-opt-report` can use to annotate source code.

`arm-opt-report` uses a YAML optimization record, as produced by the `-fsave-optimization-record` option of LLVM, to output annotated source code that shows the various optimization decisions taken by the compiler.

Note

`-fsave-optimization-record` is not set by default by Arm® Compiler for Linux.

Possible annotations are:

Annotation	Description
`I`	A function was inlined.
`U<N>`	A loop was unrolled `<N>` times.
`V<F, I>`	A loop has been vectorized. Each vector iteration that is performed has the equivalent of `F*I` scalar iterations. Vectorization Factor, `F`, is the number of scalar elements that are processed in parallel. Interleave count, `I`, is the number of times the vector loop was unrolled.
`VS<F,I>`	A loop has been vectorized using scalable vectors. Each vector iteration performed has the equivalent of `N*F*I` scalar iterations, where `N` is the number of vector granules, which can vary according to the machine the program is run on. **Note:** LLVM assumes a granule size of 128 bits when targeting SVE. `F` (Vectorization Factor) and `I` (Interleave count) are as described for `V<F,I>`.

Syntax

```
arm-opt-report [options] <input>
```

Options

Generic Options:

`--help`

    Displays the available options (use `--help-hidden` for more).

--help-list

> Displays a list of available options (`--help-list-hidden` for more).

--version

> Displays the version of this program.

llvm-opt-report options:

--hide-detrimental-vectorization-info

> Hides remarks about vectorization being forced despite the cost-model indicating that it is not beneficial.

--hide-inline-hints

> Hides suggestions to inline function calls which are preventing vectorization.

--hide-lib-call-remark

> Hides remarks about the calls to library functions that are preventing vectorization.

--hide-vectorization-cost-info

> Hides remarks about the cost of loops that are not beneficial for vectorization.

--no-demangle

> Does not demangle function names.

-o=<string>

> Specifies an output file to write the report to.

-r=<string>

> Specifies the root for relative input paths.

-s

> Omits vectorization factors and associated information.

--strip-comments

> Removes comments for brevity

--strip-comments=<arg>

> Removes comments for brevity. Arguments are:
>
> - `none`: Do not strip comments.
> - `c`: Strip C-style comments.
> - `c++`: Strip C++-style comments.
> - `fortran`: Strip Fortran-style comments.

Outputs

Annotated source code.

Related information

4.10 Optimization remarks

Optimization remarks provide you with information about the choices that are made by the compiler. You can use them to see which code has been inlined or they can help you understand why a loop has not been vectorized.

By default, Arm® Compiler for Linux prints optimization remark information to `stderr`. If this is your terminal output, you might want to redirect the terminal output to a separate file to store and search the remark information more easily.

To enable optimization remarks, pass one or more of the following `-Rpass` options (in any order) to `armclang|armclang++` at compile time:

- `-Rpass=<regex>`: Information about what the compiler has optimized.

- `-Rpass-analysis=<regex>`: Information about what the compiler has analyzed.

- `-Rpass-missed=<regex>`: Information about what the compiler failed to optimize.

For each option, replace `<regex>` with a remark expression that you want see. The supported remark types are:

- `loop-vectorize`: Provides remarks about vectorized loops.

- `inline`: Provides remarks about inlining.

- `loop-unroll`: Provides remarks about unrolled loops.

`<regex>` can be one or more of the preceding remark types. If you filter for multiple remark types, separate each type with a pipe (`|`) character.

For example, to request information about loops that were successfully vectorized, loops that the compiler failed to vectorize, and information about why a loop failed to vectorize, use:

```
armclang -O<level> -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-
analysis=loop-vectorize <source>.c
```

Alternatively, you can choose to print all optimization remark information by specifying `.*` for `<regex>`, such as:

```

```

It can also be useful to redirect the optimization remarks to a separate file. The general syntax to compile with optimization remarks enabled (`-Rpass[-<option>]`) and redirect the information to an output file (such as, `<remarks-file.txt>`), is:

```
armclang -O<level> -Rpass[-<option>]=<regex> <source>.c 2> <remarks-file.txt>
```

> **Note**
>
> `2> <remarks-file.txt>` assumes a Bourne-shell syntax. You need to replace this with the appropriate syntax to redirect output in your shell type.

4.10.1 Enable Optimization remarks

Describes how to enable optimization remarks and redirect the information they provide to an output file.

Before you begin

Download and install Arm® Compiler for Linux. You can download Arm Compiler for Linux from the download page. Learn how to install and configure Arm Compiler for Linux, using the installation instructions on the Arm Developer website.

Procedure

1. Compile your code with optimization remarks. To enable optimization remarks, pass one or more of `-Rpass=<regex>`, `-Rpass-missed=<regex>`, or `Rpass-analysis=<regex>` on your compile line.
   For example, to report all the information about what the compiler has optimized (`-Rpass`), and what the compiler has analyzed (`-Rpass-analysis`) when compiling an input file called `source.c`, use:

   ```
   armclang -O3 -Rpass=.* -Rpass-analysis=.* source.c
   ```

   Result:

   ```
   source.c:8:18: remark: hoisting zext [-Rpass=licm]
           for (int i=0;i<K; i++)
           ^
   source.c:8:4: remark: vectorized loop (vectorization width: 4, interleaved count:
    2) [-Rpass=loop-vectorize]
           for (int i=0;i<K; i++)
           ^
   source.c:7:1: remark: 28 instructions in function [-Rpass-analysis=asm-printer]
           void foo(int K) {
           ^
   ```

2. Or, to print the optimization remark information to a separate file, instead of `stderr`, run:

   ```
   armclang -O<level> -Rpass[-<option>]=<remark(s)> <source>.c 2> <remarks-file.txt>
   ```

   Replacing `2>` with the appropriate redirection syntax for the shell type you are using.

Results

A `<remarks-file.txt>` file is output with the optimization remarks in it.

Related information

Arm C/C++ Compiler

4.11 Prefetching with __builtin_prefetch

This topic describes how you can enable prefetching in your C/C++ code with Arm® Compiler for Linux.

To reduce the cache-miss latency of memory accesses, you can prefetch data. When you know the addresses of data in memory that are going to be accessed soon, you can inform the target, through instructions in the code, to fetch the data and place them in the cache before they are required for processing.

Note that the prefetching instruction is a hardware hint, which means that your target processor might, or might not, actually prefetch the data.

`__builtin_prefetch` syntax

In Arm Compiler for Linux the target can be instructed to prefetch data using the `__builtin_prefetch` C/C++ function, which takes the syntax:

```
__builtin_prefetch (const void *addr[, rw[, locality]])
```

where:

`addr` (required)

Represents the address of the memory.

`rw` (optional)

A compile-time constant which can take the values:

- `0` (default): prepare the prefetch for a read

- `1` : prepare the prefetch for a write to the memory

`locality` (optional)

A compile-time constant integer which can take the following temporal locality (L) values:

- `0`: None, the data can be removed from the cache after the access.

- `1`: Low, L3 cache, leave the data in the L3 cache level after the access.

- `2`: Moderate, L2 cache, leave the data in L2 and L3 cache levels after the access.

- `3` (default): High, L1 cache, leave the data in the L1, L2, and L3 cache levels after the access.

Note

`addr` must be expressed correctly or Arm C/C++ Compiler will generate an error.

> **Note** Take care when inserting prefetch instructions into the inner loops of code because these instructions will inhibit vectorization. Depending on the context of the code, it might be possible to include prefetch instructions outside of the inner loop of your source code, and not inhibit vectorization.

Example

To illustrate the different forms the `__builtin_prefetch` function can take, see the example functions in the following code, `c-prefetching.c`:

```c
void streaming_load(void *foo) {          // Streaming load
    __builtin_prefetch(foo + 1024,        // Address can be offset
                       0,                 // Read
                       0                  // No locality - streaming access
                       );
}
void l3_load(void *foo) {
    __builtin_prefetch(foo, 0, 1);        // L3 load prefetch (locality)
}
void l2_load(void *foo) {
    __builtin_prefetch(foo, 0, 2);        // L2 load prefetch (locality)
}
void l1_load(void *foo) {
    __builtin_prefetch(foo, 0, 3);        // L1 load prefetch (locality)
}
void streaming_store(void *foo) {
    __builtin_prefetch(foo + 1024, 1, 0); // Streaming store
}
void l3_store(void *foo) {
    __builtin_prefetch(foo, 1, 1);        // L3 store prefetch (locality)
}
void l2_store(void *foo) {
    __builtin_prefetch(foo, 1, 2);        // L2 store prefetch (locality)
}
void l1_store(void *foo) {
    __builtin_prefetch(foo, 1, 3);        // L1 store prefetch (locality)
}
```

Which, when compiled using:

```
armclang -c -O3 c-prefetching.c
```

generates the following assembly:

```
streaming_load:
      prfm    PLDL1STRM, [x0, 1024]      ; Streaming load
      ret
l3_load:
      prfm    PLDL3KEEP, [x0]            ; L3 load prefetch (locality)
      ret
l2_load:
      prfm    PLDL2KEEP, [x0]            ; L2 load prefetch (locality)
      ret
l1_load:
      prfm    PLDL1KEEP, [x0]            ; L1 load prefetch (locality)
      ret
streaming_store:
      prfm    PSTL1STRM, [x0, 1024]      ; Streaming store
      ret
```

```
l3_store:
        prfm    PSTL3KEEP, [x0]             ; L3 store prefetch (locality)
        ret
l2_store:
        prfm    PSTL2KEEP, [x0]             ; L2 store prefetch (locality)
        ret
l1_store:
        prfm    PSTL1KEEP, [x0]             ; L1 store prefetch (locality)
        ret
```

5 Compiler options

This chapter describes the options supported by `armclang` and `armclang++`.

`armclang` and `armclang++` provide many command-line options, including most Clang command-line options in addition to a number of Arm-specific options. Many common options, together with the Arm-specific options, are described in this chapter. The same options are also described in the tool through the `--help` option (run `armclang|armclang++ --help`), and in the `man` pages (run `man armclang|armclang++`).

Additional information about community feature command-line options is available in the Clang and LLVM documentation on the LLVM Compiler Infrastructure Project web site, http://llvm.org.

To see a list of arguments that Arm® C/C++ Compiler supports for a specific option, bash terminal users can also use command line completion (also known as tab completion). For example, to list the supported arguments for `-ffp-contract=` with `armclang` type the following command line into your terminal (but do not run it):

```
armclang -ffp-contract=
```

Press the **Tab** button on your keyboard. The arguments supported by `-ffp-contract=` return:

```
fast  off  on
```

Note

For more information about enabling this for other terminal types, see the installation instructions.

5.1 Arm C/C++ Compiler Options by Function

This provides a summary of the `armclang` and `armclang++` command-line options that Arm® C/C++ Compiler supports.

Actions
Options that control what action to perform on the input.

Option	Description
-E	Stop after pre-processing. Output the pre-processed source.
-S	Stop after compiling the source and emit assembler files.
-c	Stop after compiling or assembling sources and do not link. This outputs object files.

Option	Description
-fopenmp	Enable ('-fopenmp') or disable ('-fno-openmp' [default]) OpenMP and link in the OpenMP library, libomp.
-fopenmp-simd	Enable processing of 'simd' and the 'declare simd' pragma, without enabling OpenMP or linking in the OpenMP library, libomp. Enabled by default.
-fsyntax-only	Show syntax errors but do not perform any compilation.

File options

Options that specify input or output files.

Option	Description
-I	Add a directory to include search path and Fortran module search path.
-config	Passes the location of a configuration file to the compile command.
-idirafter	Add directory to include search path after system header file directories.
-include	Include file before parsing.
-iquote	Add directory to include search path. Directories specified with the '-iquote' option apply only to the quote form of the include directive.
-isysroot	For header files, set the system root directory (usually /).
-isystem	Add a directory to the include search path, before system header file directories.
-o	Write the output to '<file>'.
-working-directory	Resolve file paths relative to the specified directory.

Basic driver options

Options that affect basic functionality of the armclang or armflang driver.

Option	Description
-###	Print (but do not run) the commands to run for this compilation.
-gcc-toolchain=	Search for GCC installation in the specified directory on targets which commonly use GCC. The directory usually contains 'lib{,32,64}/gcc{,-cross}/$triple' and 'include'. If specified, sysroot is skipped for GCC detection. Note: executables (for example, 'ld') used by the compiler are not overridden by the selected GCC installation.
-help	Display available options.
-help-hidden	Display hidden options. Only use these options if advised to do so by your Arm representative.
-print-search-dirs	Print the paths that are used for finding libraries and programs.
-v	Show commands to run and use verbose output.
-version	Show the version number and some other basic information about the compiler.

Optimization options

Options that control what optimizations should be performed.

Option	Description
-O	Specifies the level of optimization to use when compiling source files.
-armpl=	Enable Arm Performance Libraries (ArmPL).
-fassociative-math	Allow ('-fassociative-math') or do not allow ('-fno-associative-math' [default]) the re-association of operands in a series of floating-point operations.
-fdenormal-fp-math=	Specify the denormal numbers the code is allowed to require.
-ffast-math	Enable ('-ffast-math') or disable ('-fno-fast-math' [default, except with '-Ofast']) aggressive, lossy floating-point optimizations.
-ffinite-math-only	Enable ('-ffinite-math-only') or disable ('-fno-finite-math-only' [default, except with '-Ofast']) optimizations that ignore the possibility of NaN and +/-Inf.
-ffp-contract=	Controls when the compiler is permitted to generate fused floating-point operations (for example, Fused Multiply-Add (FMA) operations).
-fhonor-infinities	Allow ('-fno-honor-infinites') or do not allow ('-fhonor-infinites' [default, except with '-Ofast']) optimizations that assume the arguments and results of floating point arithmetic are not +/-Inf.
-fhonor-nans	Allow ('-fno-honor-nans') or do not allow ('-fhonor-nans' [default, except with '-Ofast']) optimizations that assume the arguments and results of floating point arithmetic are not NaN.
-finline-functions	Inline ('-finline-functions') or do not inline ('-fno-inline-functions') suitable functions.
-finline-hint-functions	Inline functions which are (explicitly or implicitly) marked 'inline'.
-flto	Enable ('-flto') or disable ('-fno-lto' [default]) Link Time Optimizations (LTO).
-fmath-errno	Require ('-fmath-errno' [default, except with '-Ofast']) or do not require ('-fno-math-errno') math functions to indicate errors.
-freciprocal-math	Enable ('-freciprocal-math') or disable ('-fno-reciprocal-math' [default, except with '-Ofast']) division operations to be reassociated.
-fsave-optimization-record	Enable ('-fsave-optimization-record') or disable ('-fno-save-optimization-record' [default]) the generation of a YAML optimization record file.
-fsigned-zeros	Allow ('-fno-signed-zeros') or do not allow ('-fsigned-zeros' [default, except with '-Ofast']) optimizations that ignore the sign of floating point zeros.
-fsimdmath	Enable ('-fsimdmath' [default for 'armflang']) or disable ('-fno-simdmath' [default for 'armclang|armclang++']) the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h
-fstrict-aliasing	Tells the compiler to adhere ('-fstrict-aliasing'), or not ('-fno-strict-aliasing'), to the aliasing rules defined in the source language.

Option	Description
-ftrapping-math	Tell the compiler to assume ('-ftrapping-math'), or not to assume ('-fno-trapping-math'), that floating point operations can trap. For example, divide by zero.
-funsafe-math-optimizations	Enable ('-funsafe-math-optimizations') or disable ('-fno-unsafe-math-optimizations' [default, except with '-Ofast']) reassociation and reciprocal math optimizations.
-fvectorize	Enable ('-fvectorize' [default]) or disable ('-fno-vectorize') loop vectorization.
-march=	Specifies the base architecture and extensions available on the target.
-mcpu=	Select which CPU architecture to optimize for.
-mrecip	Enable optimizations that replace division by reciprocal estimation and refinement.
-msve-vector-bits=	Specifies the length of SVE vector register, in bits, for Vector Length Specific (VLS) programming. Defaults to the Vector Length Agnostic (VLA) value of 'scalable'. (AArch64 only)

C/C++ Options

Options that affect the way C workloads are compiled.

Option	Description
-fcommon	Place uninitialized global variables in a common block
-fsigned-char	Set the type of 'char' to be signed ('fsigned-char') or unsigned ('fno-signed-char' [default]).
-std=	Language standard to compile for.

Development options

Options that facilitate code development.

Option	Description
-fcolor-diagnostics	Enable ('-fcolor-diagnostics') or disable ('-fno-color-diagnostics' [default]) using colors in diagnostics.
-g	Generate source-level debug information with DWARF version 4.
-g0	Disable the generation of source-level debug information.
-gline-tables-only	Emit debug line number tables only.

Warning options

Options that control the behavior of warnings.

Option	Description
-Qunused-arguments	Do not emit a warning for unused driver arguments.
-W	Enable ('-W<warning>') or disable ('-Wno-<warning>') a specified warning, '<warning>'.
-Wall	Enable all warnings.

Option	Description
-Warm-extensions	Enable warnings about the use of non-standard language features supported by armclang
-Wdeprecated	Enable warnings for deprecated constructs and define __DEPRECATED.
-fno-crash-diagnostics	Disable the auto-generation of preprocessed source files and a script for reproduction during a clang crash.
-w	Suppress all warnings.

Preprocessor options

Options that control the behavior of the preprocessor.

Option	Description
-D	Define a macro name to a value, '-D<macro>=<value>'. If a value is omitted, the macro is defined as 1.
-U	Undefine a macro, '-U<macro>'.

Linker options

Options that are passed on to the linker or affect linking.

Option	Description
-L	Add a directory to the list of paths that the linker searches for user libraries.
-Wl,	Pass comma-separated arguments to the linker, '-Wl,<arg>,<arg>,...'.
-Xlinker	Pass an argument to the linker, '-Xlinker <arg>'.
-l	Search for a library when linking, '-l<library>'.
-shared	Create a shared object that can be linked against.
-static	Link against static libraries.

5.2 -###

Print (but do not run) the commands to run for this compilation.

Syntax

```
armclang -###
```

5.3 -armpl=

Enable Arm Performance Libraries (ArmPL).

Instructs the compiler to link with ArmPL. ArmPL provides functions optimized for a range of supported CPUs. The most suitable implementation is detected at runtime, not at compilation

time, and it does not require any other compilation flags. This option also enables optimized versions of the C mathematical functions declared in the math.h library, tuned scalar and vector implementations of Fortran math intrinsics. This option implies `-fsimdmath`.

The -armpl option also enables:

*   Optimized versions of the C mathematical functions declared in math.h.

*   Optimized versions of Fortran math intrinsics.

*   Auto-vectorization of C mathematical functions (disable this with `-fno-simdmath`).

*   Auto-vectorization of Fortran math intrinsics (disable this with `-fno-simdmath`).

Default

By default, `-armpl` is not set (in other words, `OFF`)

Default argument behavior

If `-armpl` is set with no arguments, the default behavior of the option is `armpl=lp64,sequential`.

However, the default behavior of the arguments is also determined by the specification (or not) of the `-i8` (when using armflang) and `-fopenmp` options:

*   If the `-i8` option is not specified, `lp64` is enabled by default. If `-i8` is specified, `ilp64` is enabled by default.

*   If the `-fopenmp` option is not specified, `sequential` is enabled by default. If `-fopenmp` is specified, `parallel` is enabled by default.

In other words:

*   Specifying `-armpl` sets `-armpl=lp64,sequential`.

*   Specifying `-armpl` and `-i8` sets `-armpl=ilp64,sequential`.

*   Specifying `-armpl` and `-fopenmp` sets `-armpl=lp64,parallel`.

*   Specifying `-armpl`, `-i8`, and `-fopenmp` sets `-armpl=ilp64,parallel`.

Syntax

```
armclang -armpl=<arg1>,<arg2>...
```

Arguments

lp64

>   Use 32-bit integers. (default)

ilp64

>   Use 64-bit integers. Inverse of lp64. (default if using `-i8` with armflang).

sequential

>   Use the single-threaded implementation of Arm Performance Libraries. (default)

`parallel`

Use the OpenMP multi-threaded implementation of Arm Performance Libraries. Inverse of sequential. (default if using `-fopenmp`)

5.4 -c

Stop after compiling or assembling sources and do not link. This outputs object files.

Syntax

```
armclang -c
```

5.5 -config

Passes the location of a configuration file to the compile command.

Use a configuration file to specify a set of compile options to be run at compile time. The configuration file can be passed at compile time, or an environment variable can be set for it to be used for every invocation of the compiler. For more information about creating and using a configuration file, see the installation instructions.

Syntax

```
armclang --config <arg>
```

5.6 -D

Define a macro name to a value, '-D<macro>=<value>'. If a value is omitted, the macro is defined as 1.

Syntax

```
armclang -D<macro>=<value>
```

5.7 -E

Stop after pre-processing. Output the pre-processed source.

Syntax

```
armclang -E
```

5.8 -fassociative-math

Allow ('-fassociative-math') or do not allow ('-fno-associative-math' [default]) the re-association of operands in a series of floating-point operations.

For example, (a * b) + (a * c) => a * (b + c). Note: Using `-fassociative-math` violates the ISO C and C++ language standard.

Default

Default is `-fno-associative-math`.

Syntax

```
armclang -fassociative-math, -fno-associative-math
```

5.9 -fcolor-diagnostics

Enable ('-fcolor-diagnostics') or disable ('-fno-color-diagnostics' [default]) using colors in diagnostics.

The output uses ANSI escape sequences to determine the color.

Default

Default is `-fno-color-diagnostics`.

Syntax

```
armclang -fcolor-diagnostics, -fno-color-diagnostics
```

5.10 -fcommon

Place uninitialized global variables in a common block

Default

Default is `-fno-common`.

Syntax

```
armclang -fcommon, -fno-common
```

5.11 -fdenormal-fp-math=

Specify the denormal numbers the code is allowed to require.

Syntax

```
armclang -fdenormal-fp-math=<arg>
```

Arguments

ieee

IEEE 754 denormal numbers.

preserve-sign

Flushed-to-zero number signs are preserved in the sign of 0.

positive-zero

Flush denormal numbers to positive zero.

5.12 -ffast-math

Enable ('-ffast-math') or disable ('-fno-fast-math' [default, except with '-Ofast']) aggressive, lossy floating-point optimizations.

Using `-ffast-math` is equivalent to specifying the following options individually:

- `-fassociative-math`
- `-ffinite-math-only`
- `-ffp-contract=fast`
- `-fno-math-errno`
- `-fno-signed-zeros`
- `-fno-trapping-math`
- `-freciprocal-math`

Default

Default is `-fno-fast-math`, except where `-Ofast` is used. Using `-Ofast` enables `-ffast-math`.

Syntax

```
armclang -ffast-math, -fno-fast-math
```

5.13 -ffinite-math-only

Enable ('-ffinite-math-only') or disable ('-fno-finite-math-only' [default, except with '-Ofast']) optimizations that ignore the possibility of NaN and +/-Inf.

Default

Default is `-fno-finite-math-only`, except where `-Ofast` is used. Using `-Ofast` enables `-ffinite-math-only`.

Syntax

```
armclang -ffinite-math-only, -fno-finite-math-only
```

5.14 -ffp-contract=

Controls when the compiler is permitted to generate fused floating-point operations (for example, Fused Multiply-Add (FMA) operations).

On the compile line, `-ffp-contract` supports three arguments to control the generation of fused floating-point operations: `OFF`, `ON`, and `FAST`. However, at the source level, you can also use the `STDC FP_CONTRACT={OFF|ON}` pragma to control the fused floating-point operation generation for C/C++ code:

- When `-ffp-contract` is set to `{off|on}`, `STDC FP_CONTRACT={OFF|ON}` is honored where it is specified, and can switch the generation.

- When `-ffp-contract` is set to `fast`, generation is always set to `FAST` and the `STDC FP_CONTRACT` pragma is ignored.

To produce better optimized code, allow the compiler to generate fused floating-point operations.

> **Note**
> The fused floating-point instructions typically operate to a higher degree of accuracy than individual multiply and add instructions.

Default

For Fortran code, the default is `-ffp-contract=fast`. For C/C++ code, the default is `-ffp-contract=off`.

Syntax

```
armclang -ffp-contract={fast\|on\|off}
```

Arguments

`fast`

Generate fused floating-point operations whenever possible, even if the operations are not permitted by the language standard. Note: Some fused floating-point contractions are not permitted by the C/C++ standard because they can lead to deviations from the expected results.

`on`

Generate fused floating-point operations only when the language permits it. For example, for C/C++ code, floating-point contractions are permitted in a single C/C++ statement, however, for Fortran code, floating-point contractions are always permitted.

`off`

Do not generate fused floating-point operations.

5.15 -fhonor-infinities

Allow ('-fno-honor-infinites') or do not allow ('-fhonor-infinites' [default, except with '-Ofast']) optimizations that assume the arguments and results of floating point arithmetic are not +/-Inf.

Default

Default is `-fhonor-infinites`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-honor-infinites`.

Syntax

```
armclang –fhonor-infinities, –fno-honor-infinities
```

5.16 -fhonor-nans

Allow ('-fno-honor-nans') or do not allow ('-fhonor-nans' [default, except with '-Ofast']) optimizations that assume the arguments and results of floating point arithmetic are not NaN.

Default

Default is `-fhonor-nans`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-honor-nans`.

Syntax

```
armclang –fhonor-nans, –fno-honor-nans
```

5.17 -finline-functions

Inline ('-finline-functions') or do not inline ('-fno-inline-functions') suitable functions.

Note: For all `-finline-*` and `-fno-inline-*` options, the compiler ignores all but the last option that is passed to the compiler command.

Default

For `armclang|armclang++`, the default at `-O0` and `-O1` is `-fno-inline-functions`, and the default at `-O2` and higher is `-finline-functions`. For `armflang`, the default at all optimization levels is `-finline-functions`.

Syntax

```
armclang -finline-functions, -fno-inline-functions
```

5.18 -finline-hint-functions

Inline functions which are (explicitly or implicitly) marked 'inline'.

Note: For all `-finline-*` and `-fno-inline-*` options, the compiler ignores all but the last option that is passed to the compiler command.

Default

Disabled by default at `-O0` and `-O1`. Enabled by default at `-O2` and higher.

Syntax

```
armclang -finline-hint-functions
```

5.19 -flto

Enable ('-flto') or disable ('-fno-Ito' [default]) Link Time Optimizations (LTO).

You must pass the option to both the link and compile commands. When LTO is enabled, compiler object files contain an intermediate representation of the original code. When linking the objects together into a binary at link time, the compiler performs optimizations. It can allow the compiler to inline functions from different files, for example.

Default

Default is `-fno-lto`.

Syntax

```
armclang -flto, -fno-lto
```

5.20 -fmath-errno

Require ('-fmath-errno' [default, except with '-Ofast']) or do not require ('-fno-math-errno') math functions to indicate errors.

Use `-fmath-errno` if your source code uses errno to check the status of math function calls. If your code never uses errno, you can use `-fno-math-errno` to unlock optimizations such as:

1. In C/C++ it allows sin() and cos() calls that take the same input to be combined into a more efficient sincos() call.

2. In C/C++ it allows certain pow(x, y) function calls to be eliminated completely when y is a small integral value.

Default

Default is `-fmath-errno`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-math-errno`.

Syntax

```
armclang -fmath-errno, -fno-math-errno
```

5.21 -fno-crash-diagnostics

Disable the auto-generation of preprocessed source files and a script for reproduction during a clang crash.

Default

By default, `-fno-crash-diagnostics` is disabled. The default behavior of the compiler enables crash diagnostics.

Syntax

```
armclang -fno-crash-diagnostics
```

5.22 -fopenmp

Enable ('-fopenmp') or disable ('-fno-openmp' [default]) OpenMP and link in the OpenMP library, libomp.

Default

Default is `-fno-openmp`.

Syntax

```
armclang -fopenmp, -fno-openmp
```

5.23 -fopenmp-simd

Enable processing of 'simd' and the 'declare simd' pragma, without enabling OpenMP or linking in the OpenMP library, libomp. Enabled by default.

Syntax

```
armclang -fopenmp-simd, -fno-openmp-simd
```

5.24 -freciprocal-math

Enable ('-freciprocal-math') or disable ('-fno-reciprocal-math' [default, except with '-Ofast']) division operations to be reassociated.

Default

Default is `-fno-reciprocal-math`, except where `-Ofast` is used. Using `-Ofast` enables `-freciprocal-math`.

Syntax

```
armclang -freciprocal-math, -fno-reciprocal-math
```

5.25 -fsave-optimization-record

Enable ('-fsave-optimization-record') or disable ('-fno-save-optimization-record' [default]) the generation of a YAML optimization record file.

Optimization records are files named <output name>.opt.yaml, which can be parsed by arm-opt-report to show what optimization decisions the compiler is making, in-line with your source code. For more information, see the 'Optimize' chapter in the compiler developer and reference guide.

Default

Default is `fno-save-optimization-record`.

Syntax

```
armclang –fsave-optimization-record, -fno-save-optimization-record
```

5.26 -fsigned-char

Set the type of 'char' to be signed ('fsigned-char') or unsigned ('fno-signed-char' [default]).

Default

Default is for the type of 'char' to be unsigned, `fno-signed-char`.

Syntax

```
armclang –fsigned-char, -fno-signed-char
```

5.27 -fsigned-zeros

Allow ('-fno-signed-zeros') or do not allow ('-fsigned-zeros' [default, except with '-Ofast']) optimizations that ignore the sign of floating point zeros.

Default

Default is `-fsigned-zeros`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-signed-zeros`.

Syntax

```
armclang –fsigned-zeros, -fno-signed-zeros
```

5.28 -fsimdmath

Enable ('-fsimdmath' [default for 'armflang']) or disable ('-fno-simdmath' [default for 'armclang| armclang++']) the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h

When vectorizing, `-fsimdmath` allows the compiler to generate calls to various vectorized library routines. These routines might use different algorithms to the scalar routine algorithms and their bit-reproducibility is not guaranteed. If you require your code to be bit reproducible, compile your code using the `-fno-simdmath` option.

Default

For `armclang|armclang++`, the default is `-fno-simdmath`. For `armflang`, the default is `-fsimdmath`.

Syntax

```
armclang –fsimdmath, –fno-simdmath
```

5.29 -fstrict-aliasing

Tells the compiler to adhere ('-fstrict-aliasing'), or not ('-fno-strict-aliasing'), to the aliasing rules defined in the source language.

In some circumstances, this flag allows the compiler to assume that pointers to different types do not alias.

Default

Default is `-fno-strict-aliasing`, except where `-Ofast` is used. Using `-Ofast` enables `-fstrict-aliasing`.

Syntax

```
armclang –fstrict-aliasing
```

5.30 -fsyntax-only

Show syntax errors but do not perform any compilation.

Syntax

```
armclang –fsyntax-only
```

5.31 -ftrapping-math

Tell the compiler to assume ('-ftrapping-math'), or not to assume ('-fno-trapping-math'), that floating point operations can trap. For example, divide by zero.

Possible traps include:

- Division by zero
- Underflow
- Overflow
- Inexact result
- Invalid operation.

Default

Default is `-ftrapping-math`, except where `-Ofast` is used. Using `-Ofast` enables `-fno-trapping-math`.

Syntax

```
armclang -ftrapping-math, -fno-trapping-math
```

5.32 -funsafe-math-optimizations

Enable ('-funsafe-math-optimizations') or disable ('-fno-unsafe-math-optimizations' [default, except with '-Ofast']) reassociation and reciprocal math optimizations.

Using `--funsafe-math-optimizations` is equivalent to specifying the following flags individually:

- `-fassociative-math`

- `-freciprocal-math`

- `-fno-signed-zeros`

- `-fno-trapping-math`

Default

Default is `-fno-unsafe-math-optimizations`, except where `-Ofast` is used. Using `-Ofast` enables `-funsafe-math-optimizations`.

Syntax

```
armclang -funsafe-math-optimizations, -fno-unsafe-math-optimizations
```

5.33 -fvectorize

Enable ('-fvectorize' [default]) or disable ('-fno-vectorize') loop vectorization.

Default

Default is `-fno-vectorize`, except where `-O2`, `-O3`, or `-Ofast` are used. Using `-O2`, `-O3`, or `-Ofast` enables `-fvectorize`.

Syntax

```
armclang -fvectorize, -fno-vectorize
```

5.34 -g

Generate source-level debug information with DWARF version 4.

Default

Disabled by default.

Syntax

```
armclang -g
```

5.35 -g0

Disable the generation of source-level debug information.

Default

Enabled by default.

Syntax

```
armclang -g0
```

5.36 -gcc-toolchain=

Search for GCC installation in the specified directory on targets which commonly use GCC. The directory usually contains 'lib{,32,64}/gcc{,-cross}/$triple' and 'include'. If specified, sysroot is skipped for GCC detection. Note: executables (for example, 'ld') used by the compiler are not overridden by the selected GCC installation.

Syntax

```
armclang --gcc-toolchain=<arg>
```

5.37 -gline-tables-only

Emit debug line number tables only.

Syntax

```
armclang -gline-tables-only
```

5.38 -help

Display available options.

Syntax

```
armclang -help, --help
```

5.39 -help-hidden

Display hidden options. Only use these options if advised to do so by your Arm representative.

Syntax

```
armclang --help-hidden
```

5.40 -I

Add a directory to include search path and Fortran module search path.

Directories specified with the `-I` option apply to both the quote form of the include directive and the system header form. For example, #include "file" (quote form), and #include <file> (system header form). Directories specified with `-I` are searched before system include directories and, in `armclang|armclang++` only, after directories specified with `-iquote` (for the quoted form). If any directory is specified with both `-I` and `-isystem` then the directory is searched for as if it were only specified with `-isystem`.

For `armflang`, search for module-files in the directories that are specified with the `-I` option. Directories that are specified with `-I` are searched after the current working directory and before standard system module locations.

Syntax

```
armclang -I<dir>
```

5.41 -idirafter

Add directory to include search path after system header file directories.

Directories specified with the `-idirafter` option apply to both the quote form of the include directive and the system header form. For example, #include "file" (quote form), and #include <file> (system header form). Directories specified with the `-idirafter` option are searched after system header file directories. Directories specified with `-idirafter` are treated as system directories.

Syntax

```
armclang -idirafter<arg>
```

5.42 -include

Include file before parsing.

Syntax

```
armclang -include<file>, --include<file>
```

5.43 -iquote

Add directory to include search path. Directories specified with the '-iquote' option apply only to the quote form of the include directive.

Directories specified with the `-iquote` option only apply to the quote form of the include directive, such as #include "file". For such directives, directories specified with `-iquote` are searched first, before directories specified by `-I`.

Syntax

```
armclang -iquote<directory>
```

5.44 -isysroot

For header files, set the system root directory (usually /).

Syntax

```
armclang -isysroot<dir>
```

5.45 -isystem

Add a directory to the include search path, before system header file directories.

Directories specified with the `-isystem` option apply to both the quote form of the include directive and the system header form. For example, #include "file" (quote form), and #include <file> (system header form). Directories specified with the `-isystem` option are searched after directories specified with `-I` and before system header file directories. Directories specified with `-isystem`

are treated as system directories. If any directory is specified with both `-I` and `-isystem` then the directory is searched for as if it were only specified with `-isystem`.

Syntax

```
armclang -isystem<directory>
```

5.46 -L

Add a directory to the list of paths that the linker searches for user libraries.

Syntax

```
armclang -L<dir>
```

5.47 -l

Search for a library when linking, '-l<library>'.

Note: 'lib' is prepended to the supplied library name. For example, to search for 'libm', use `-lm`.

Syntax

```
armclang -l<library>
```

5.48 -march=

Specifies the base architecture and extensions available on the target.

Usage: `-march=<arg>` where `<arg>` is constructed as *name[+[no]feature+...]*:

name

> `armv8-a` : Armv8 application architecture profile.
>
> `armv8.1-a` : Armv8.1 application architecture profile.
>
> `armv8.2-a` : Armv8.2 application architecture profile.
>
> `armv8.3-a` : Armv8.3 application architecture profile.
>
> `armv8.4-a` : Armv8.4 application architecture profile.
>
> `armv8.5-a` : Armv8.5 application architecture profile.
>
> `armv8.6-a` : Armv8.6 application architecture profile.

feature

Is the name of an optional architectural feature that can be explicitly enabled with +feature and disabled with +nofeature.

For AArch64, the following features can be specified:

- `crc` - Enable CRC extension. On by default for `-march=armv8.1-a` or higher.

- `crypto` - Enable Cryptographic extension.

- `fullfp16` - Enable FP16 extension.

- `lse` - Enable Large System Extension instructions. On by default for `-march=armv8.1-a` or higher.

- `sve` - Scalable Vector Extension (SVE). This feature also enables `fullfp16`. See Scalable Vector Extension for more information.

- `sve2`- Scalable Vector Extension version two (SVE2). This feature also enables `sve`. See Arm A64 Instruction Set Architecture for SVE and SVE2 instructions.

- `sve2-aes` - SVE2 Cryptographic extension. This feature also enables `sve2`.

- `sve2-bitperm` - SVE2 Cryptographic Extension. This feature also enables `sve2`.

- `sve2-sha3` - SVE2 Cryptographic Extension. This feature also enables `sve2`.

- `sve2-sm4` - SVE2 Cryptographic Extension. This feature also enables `sve2`.

Syntax

```
armclang -march=<arg>
```

5.49 -mcpu=

Select which CPU architecture to optimize for.

Syntax

```
armclang -mcpu=<arg>
```

Arguments

native

Auto-detect the CPU architecture from the build computer.

thunderx2t99

Optimize for Marvell ThunderX2 based computers.

neoverse-n1

Optimize for Neoverse N1 based computers.

neoverse-n2

Optimize for Neoverse N2 based computers.

neoverse-v1

> Optimize for Neoverse V1 based computers.

a64fx

> Optimize for Fujitsu A64FX based computers.

generic

> Generate portable code suitable for any Armv8-A based computer.

5.50 -mrecip

Enable optimizations that replace division by reciprocal estimation and refinement.

Default

Disabled by default, except where `-ofast` is used. Using `-ofast` enables `-mrecip`.

Syntax

```
armclang -mrecip
```

5.51 -msve-vector-bits=

Specifies the length of SVE vector register, in bits, for Vector Length Specific (VLS) programming. Defaults to the Vector Length Agnostic (VLA) value of 'scalable'. (AArch64 only)

Default

Default is `scalable`.

Syntax

```
armclang -msve-vector-bits=<arg>
```

Arguments

128

> Compile for an SVE vector register length of 128-bits.

256

> Compile for an SVE vector register length of 256-bits.

512

> Compile foran SVE vector register length of 512-bits.

1024

> Compile for an SVE vector register length of 1024-bits.

2048

Compile for an SVE vector register length of 2048-bits.

scalable

Do not compile for any specific SVE vector register length. Compile for Vector Length Agnostic (VLA) programming. (Default)

5.52 -O

Specifies the level of optimization to use when compiling source files.

Note: If you use `-O2`, `-O3`, or `-Ofast` with the `-fsimdmath` option, the compiler might vectorize loops using calls to vectorized math routines, affecting the bit reproducibility. For more information, see the `-fsimdmath` option description.

Default

The default is `-O0`. However, for the best balance between ease of debugging, code size, and performance, it is important to choose an optimization level that is appropriate for your goals.

Syntax

```
armclang -O<level>
```

Arguments

0

Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.

1

Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.

2

High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.

3

Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

fast

Enables all the optimizations from level 3 including those performed with the `-ffp-mode=fast` option. This level also performs other aggressive optimizations that might violate strict compliance with language standards. `-Ofast` implies `-ffast-math`.

5.53 -o

Write the output to '<file>'.

Default

If a user-defined filename is not provided, the compiler uses the input filename as the output filename (replacing the extension, as appropriate). If a user-defined filename is provided, the compiler writes the output to the provided filename.

Syntax

```
armclang -o<file>
```

5.54 -print-search-dirs

Print the paths that are used for finding libraries and programs.

Syntax

```
armclang -print-search-dirs, --print-search-dirs
```

5.55 -Qunused-arguments

Do not emit a warning for unused driver arguments.

Syntax

```
armclang -Qunused-arguments
```

5.56 -S

Stop after compiling the source and emit assembler files.

Syntax

```
armclang -S
```

5.57 -shared

Create a shared object that can be linked against.

Syntax

```
armclang -shared, --shared
```

5.58 -static

Link against static libraries.

This option prevents runtime dependencies on shared libraries. This is likely to result in larger binaries.

Syntax

```
armclang -static, --static
```

5.59 -std=

Language standard to compile for.

The list of valid standards depends on the input language of your source file. To view a list of supported standards, add -std= to a compile line and run the command. The compiler generates an error message and lists the valid arguments for your source. Note: You must provide a source file that the compiler can locate to generate the list of supported standards for -std=.

Syntax

```
armclang -std=<arg>, --std=<arg>
```

5.60 -U

Undefine a macro, '-U<macro>'.

Syntax

```
armclang -U<macro>
```

5.61 -v

Show commands to run and use verbose output.

Syntax

```
armclang -v
```

5.62 -version

Show the version number and some other basic information about the compiler.

Syntax

```
armclang --version, --vsn
```

5.63 -W

Enable ('-W<warning>') or disable ('-Wno-<warning>') a specified warning, '<warning>'.

Syntax

```
armclang -W<warning>
```

5.64 -Wall

Enable all warnings.

Syntax

```
armclang -Wall
```

5.65 -Warm-extensions

Enable warnings about the use of non-standard language features supported by armclang

Syntax

```
armclang -Warm-extensions
```

5.66 -Wdeprecated

Enable warnings for deprecated constructs and define __DEPRECATED.

Syntax

```
armclang -Wdeprecated
```

5.67 -Wl,

Pass comma-separated arguments to the linker, '-Wl,<arg>,<arg>,...'.

Syntax

```
armclang -Wl,<arg>,<arg2>...
```

5.68 -w

Suppress all warnings.

Syntax

```
armclang -w
```

5.69 -working-directory

Resolve file paths relative to the specified directory.

Syntax

```
armclang -working-directory<arg>
```

5.70 -Xlinker

Pass an argument to the linker, '-Xlinker <arg>'.

Syntax

```
armclang -Xlinker <arg>
```

6 Standards support

This chapter describes the support status of Arm® C/C++ Compiler with the C/C++ language and OpenMP standards.

6.1 Supported C/C++ standards in Arm C/C++ Compiler

This topic describes the support for the C and C++ language standards in Arm® C/C++ Compiler.

C support

For C language compilation, Arm C/C++ Compiler fully supports the C17 standard (ISO/IEC 9899:2018), as well as the gnu17 extensions, and prior published standards (C89 and GNU89, GNUC99 and GNU99, and C11 and GNU11).

To select which language standard Arm C/C++ Compiler should use, use the -std= compiler option with the argument:

- `c89` or `c90` for the 'ISO C 1990' standard

- `gnu89` or `gnu90` for the 'ISO C 1990 with GNU extensions' standard

- `c99` for the 'ISO C 1999' standard

- `gnu99` for the 'ISO C 1999 with GNU extensions' standard

- `c11` for the 'ISO C 2011' standard

- `gnu11` for the 'ISO C 2011 with GNU extensions' standard

- `c17` or `c18`, or for the 'ISO C 2017' standard

- `gnu17` or `gnu18` for the 'ISO C 2017 with GNU extensions' standard

The default for C code compilation is `-std=gnu17`.

C++ support

For C++ language compilation, Arm C/C++ Compiler fully supports the C++17 standard (ISO/IEC 14882:2017), as well as the gnu++17 extensions, and prior published standards (C++98 and gnu++98, C++03 and gnu++03, C++11 and gnu++11, and C++14 and gnu++14).

Note

Exported templates, as included in the C++98 standard, were removed in the C++11 standard, and are not supported.

To select which language standard Arm C/C++ Compiler should use, use the -std= compiler option with the argument:

- `c++98` or `c++03` for the the 'ISO C++ 1998 with amendments' standard

- `gnu++98` or `gnu++03` for the the 'ISO C++ 1998 with amendments and GNU extensions' standard

- `c++11` for the the 'ISO C++ 2011 with amendments' standard

- `gnu++11` for the the 'ISO C++ 2011 with amendments and GNU extensions' standard

- `c++14` for the 'ISO C++ 2014 with amendments' standard

- `gnu++14` for the 'ISO C++ 2014 with amendments and GNU extensions' standard

- `c++17` for the 'ISO C++ 2017 with amendments' standard

- `gnu++17` for the 'ISO C++ 2017 with amendments and GNU extensions' standard

The default for C++ code compilation is `-std=gnu++14`.

Specific features that are, and are not, supported in the C++ standards are detailed on the LLVM C++ Support in Clang. Arm C/C++ Compiler version 22.0.2 is based on Clang version 13.0.0.

Related information

6.2 OpenMP 4.0

Describes which OpenMP 4.0 features are supported by Arm® C/C++ Compiler.

Table 6-1: Supported OpenMP 4.0 features

Open MP 4.0 Feature	Support
C/C++ Array Sections	Yes
Thread affinity policies	Yes
`simd` construct	Yes
`declare simd` construct	No
Device constructs	No
Task dependencies	Yes
`taskgroup` construct	Yes
User defined reductions	Yes
Atomic capture swap	Yes
Atomic `seq_cst`	Yes
Cancellation	Yes
`OMP_DISPLAY_ENV`	Yes

Related information
OpenMP thread mapping

6.3 OpenMP 4.5

Describes which OpenMP 4.5 features are supported by Arm® C/C++ Compiler.

Table 6-2: Supported OpenMP 4.5 features

Open MP 4.5 Feature	Support
`doacross` loop nests with ordered	Yes
`linear` clause on `loop` construct	Yes
`simdlen` clause on `simd` construct	Yes
Task priorities	Yes
`taskloop` construct	Yes
Extensions to device support	No
`if` clause for combined constructs	Yes
`hint` clause for `critical` construct	Yes
`source` and `sink` dependence types	Yes
C++ Reference types in data sharing attribute clauses	Yes
Reductions on C/C++ array sections	Yes
`ref`, `val`, and `uval` modifiers for `linear` clause.	Yes
Thread affinity query functions	Yes
Hints for lock API	Yes

Related information
OpenMP thread mapping

6.4 OpenMP 5.0

Describes which OpenMP 5.0 features are supported by Arm® C/C++ Compiler.

Table 6-3: Supported OpenMP 5.0 features

Open MP 5.0 Feature	Support
Support `!=` in the canonical loop form	Yes
`#pragma omp loop` directive	No
Collapse imperfect and non-rectangular nested loops	Yes
C++ range-base for loop	Yes
Clause: `if` for SIMD directives	Yes
Inclusive scan extension (matching C++17 PSTL)	Yes
Memory allocators	Yes
`allocate` directive and `allocate clause`	Yes
OMPD and OMPT interfaces	No
Thread affinity extension	Yes
Taskloop reduction	Yes

Open MP 5.0 Feature	Support
Task affinity	No
Clause: depend on the `taskwait` construct	No
Depend objects and detachable tasks	Yes
`mutexinoutset` dependence-type for tasks	Yes
Combined `taskloop` constructs	Yes
`master taskloop` and `parallel master taskloop`	Yes
`master taskloop simd` and `parallel master taskloop simd`	Yes
`atomic` and `simd` constructs inside SIMD code	Yes
SIMD nontemporal	Yes
Infer target functions and variables from initializers	No
`OMP_TARGET_OFFLOAD` environment variable	Yes
Support full 'defaultmap' functionality	Yes
Device-specific functions	Yes
Clause: `device_type`	Yes
Clause: `extended device`	Yes
Clause: `uses_allocators` clause	Yes
Clause: `in_reduction`	No
`omp_get_device_num()`	No
Structure mapping of references	No
Nested target `declare`	Yes
Implicitly map 'this' (this[:1])	Yes
Allow access to the reference count (`omp_target_is_present`)	No
`requires` directive	No
Clause: unified_shared_memory	Yes
Clause: unified_address	No
Clause: reverse_offload	No
Clause: atomic_default_mem_order	Yes
Clause: dynamic_allocators	No
User-defined mappers	No
Mapping lambda expression	Yes
Clause: use_device_addr for target data	Yes
Support close modifier on map clause	Yes
`teams` construct on the host device	No
Support non-contiguous array sections for target update	No
Pointer attachment	No
Hints for the `atomic` construct	Yes
C11 and C++11/14/17 support	Yes
Lambda support	Yes
Array shaping	Yes
Library shutdown (`omp_pause_resource[_all]`)	No

Open MP 5.0 Feature	Support
Metadirectives	No
Conditional modifier for `lastprivate` clause	Yes
Iterator and multidependences	Yes
`depobj` directive and `depobj` dependency `kind`	Yes
User-defined function variants	No
Pointer and reference to pointer-based array reductions	No
Prevent new type definitions in clauses	Yes
Memory model update (`seq_cst`, `acq_rel`, `release`, `acquire`, ...)	Yes

7 Supporting reference information

This chapter describes the compatibility with GCC and the relationship with the LLVM Clang compiler, on which Arm® C/C++ Compiler is based. The chapter also provides information on the environment variables available in Arm C/C++ Compiler.

7.1 Arm Compiler for Linux environment variables

Describes where to find reference information about the environment variables that are available in Arm® Compiler for Linux.

The environment variables that are available to use with both Arm C/C++/Fortran Compiler and Arm Performance Libraries are described on the Environment variables reference for Arm Server and High Performance Computing (HPC) tools Developer webpage.

7.2 GCC compatibility provided by Arm C/C++ Compiler

The compiler in Arm® C/C++ Compiler 22.0.2 is based on Clang and LLVM technology. As such, it provides a high degree of compatibility with GCC.

Arm C/C++ Compiler can build most of the C code that is written to be built with GCC. However, Arm C/C++ Compiler is not 100% source compatible in all cases. Specifically, Arm C/C++ Compiler does not aim to be bug-compatible with GCC. That is, Arm C/C++ Compiler does not replicate GCC bugs.

7.3 Clang and LLVM documentation

Arm® C/C++ Compiler 22.0.2 is based on Clang version 13.0.0, which is part of the LLVM Compiler Infrastructure open source project.

The Arm C/C++ Compiler documentation describes the features that are supported in Arm C/C++ Compiler. Where possible, the functionality of the open source technology is preserved. This means that there are additional features available in Arm Compiler for Linux that are not listed in the documentation. These additional features are known as community features. For support level definitions, see Support level definitions.

You can find the documentation about how to use the community features at:

- The Clang Compiler User Manual

- The Clang 13.0.0 Release Notes:

  https://releases.llvm.org/13.0.0/tools/clang/docs/ReleaseNotes.html

---

Note

Arm C/C++ Compiler is based on 13.0.0.

The `third_party_licenses.txt` file includes details of all the open source software projects which are relevant to Arm Compiler for Linux 22.0.2, including the git hashes of the open source projects that Arm C/C++ Compiler is based on.

7.4 Support level definitions

This describes the levels of support for various Arm® Compiler for Linux features.

Arm Compiler for Linux is built on open source technology. Therefore, it has more functionality than the set of product features described in the documentation. The following definitions clarify the levels of support and guarantees on functionality that are expected from these features.

Identification in the documentation

All features that are documented in the Arm Compiler for Linux documentation are product features, except where explicitly stated. The limitations of non-product features are explicitly stated.

Product features

Product features are suitable for use in a production environment. The functionality is well-tested, and is expected to be stable across feature and update releases.

Community features

Arm Compiler for Linux is built on LLVM technology and preserves the functionality of that technology where possible. This means that there are additional features available in Arm Compiler for Linux that are not listed in the documentation. These additional features are known as community features. For information on these community features, see the documentation for the Clang LLVM project and https://releases.llvm.org/13.0.0/tools/clang/docs/ReleaseNotes.html. For Fortran support, also see the Flang community GitHub web site.

Note

Arm Compiler for Linux 22.0.2 is based on Clang 13.0.0.

Where community features are referenced in the documentation, they are indicated with [COMMUNITY].

- Arm makes no claims about the quality level or the degree of functionality of these features, except when explicitly stated in this documentation.

- Functionality might change significantly between releases.

You are responsible for making sure that any generated code that uses unsupported or community features operates correctly.

Some community features might become product features in the future, but Arm provides no roadmap for this. Arm is interested in understanding your use of these features, and welcomes feedback on them.

Deprecated features

A deprecated feature is one that Arm plans to remove from a future release of Arm Compiler for Linux. Arm does not make any guarantee regarding the testing or maintenance of deprecated features. Therefore, Arm does not recommend using a feature after it is deprecated.

For information on replacing deprecated features with supported features, refer to the Arm Compiler for Linux documentation and Release Notes.

Unsupported features

With both the product and community feature categories, specific features and use-cases are known not to function correctly, or are not intended for use with Arm Compiler for Linux.

Limitations of product features are stated in the documentation. Arm cannot provide an exhaustive list of unsupported features, or unsupported use-cases, for community features.

8 Troubleshoot

This chapter describes how to diagnose problems when compiling applications using Arm® C/C++ Compiler.

8.1 Application segfaults at -Ofast optimization level

A program runs correctly when the binary is built using the `-O3` optimization level, but encounters a runtime crash or segfault with `-Ofast` optimization level.

Condition

The runtime segfault only occurs when `-Ofast` is used to compile the code. The segfault disappears when you add the `-fno-stack-arrays` option to the compile line. .

The `-fstack-arrays` option is enabled by default at `-Ofast`

When the `-fstack-arrays` option is enabled, either on its own or enabled with `-Ofast` by default, the compiler allocates arrays for all sizes using the local stack for local and temporary arrays. This helps to improve performance, because it avoids slower heap operations with `malloc()` and `free()`. However, applications that use large arrays might reach the Linux stack-size limit at runtime and produce program segfaults. On typical Linux systems, a default stack-size limit is set, such as 8192 kilobytes. You can adjust this default stack-size limit to a suitable value.

Solution

Use `-Ofast -fno-stack-arrays` instead. The combination of `-Ofast -fno-stack-arrays` disables automatic arrays on the local stack, and keeps all other `-Ofast` optimizations. Alternatively, to set the stack so that it is larger than the default size, call `ulimit -s unlimited` before running the program.

8.2 Compiling with the -fpic option fails when using GCC compilers

Describes the difference between the `-fpic` and `-fPIC` options when compiling for Arm with GCC and Arm® Compiler for Linux.

Condition

Failure can occur at the linking stage when building Position-Independent Code (PIC) on AArch64 using the lower-case `-fpic` compiler option with GCC compilers (gfortran, gcc, g++), in preference to using the upper-case `-fPIC` option.

Note

- This issue does not occur when using the `-fpic` option with Arm Compiler for Linux (`armflang`/`armclang`/`armclang++`), and it also does not occur on x86_64 because -fpic operates the same as `-fPIC`.

- PIC is code which is suitable for shared libraries.

Cause

Using the `-fpic` compiler option with GCC compilers on AArch64 causes the compiler to generate one less instruction per address computation in the code, and can provide code size and performance benefits. However, it also sets a limit of 32k for the Global Offset Table (GOT), and the build can fail at the executable linking stage because the GOT overflows.

Note

When building PIC with Arm Compiler for Linux on AArch64, or building PIC on x86_64, `-fpic` does not set a limit for the GOT, and this issue does not occur.

Solution

Consider using the `-fPIC` compiler option with GCC compilers on AArch64, because it ensures that the size of the GOT for a dynamically linked executable will be large enough to allow the entries to be resolved by the dynamic loader.

8.3 Error messages when installing Arm Compiler for Linux

If you experience a problem when installing Arm® Compiler for Linux, consider the following points.

- To perform a system-wide install, ensure that you have the correct permissions. If you do not have the correct permissions, the following errors are returned:
  - Systems using RPM Package Manager (RPM):

    ```
    error: can't create transaction lock on /var/lib/rpm/.rpm.lock (Permission
     denied)
    ```

  - Debian systems using dpkg:

    ```
    dpkg: error: requested operation requires superuser privilege
    ```

- If you install using the `--install-to <directory>` option, ensure that the system you are installing on has the required rpm or dpkg binaries installed. If it does not, the following errors are returned:
  - Systems using RPM Package Manager (RPM):

    ```
    Cannot find 'rpm' on your PATH. Unable to extract .rpm files.
    ```

◦ Debian systems using dpkg:

```
Cannot find 'dpkg' on your PATH. Unable to extract .deb files.
```

8.4 Error moving Arm Compiler for Linux modulefiles

Describes a workaround to use if you move Arm® Compiler for Linux environment modulefiles.

Moving installed Arm Compiler for Linux modulefiles causes them to stop working

By default, Arm Compiler for Linux modulefiles are configured to find the Arm Compiler for Linux binaries at a location that is relative to the modulefiles.

Moving or copying the modulefiles to a new location means that the installed binaries are no longer at the same relative location to the new modulefile location. When trying to locate binaries, the broken relative links between the new modulefile location and the location of the installed binaries causes the new modulefiles to fail.

Workaround

Move the dependency modulefile directories `/moduledeps` and `module_globals` with the modulefile or modulefile directory you are moving:

- If you move an individual modulefile, such as the `acfl/<package-version>` modulefile, move the `/moduledeps/` and `/module_globals/` modulefile directories to one directory level above the new location of the modulefile you moved.

- If you move the `/modulefiles/` directory, move the `/moduledeps/` and `/module_globals/` modulefile directories to the same new directory location as `/modulefiles/`.

Note
`<package-version>` is equivalent to `<major-version>.<minor-version>{.<patch-version>}`.

Related information
|armcompilersuite| installation instructions

8.5 Code is not bit-reproducible

Describes the compiler options to use to generate bit-reproducible code.

Condition

Code is being compiled with autovectorization enabled (using one of the `-O2`, `-O3`, or `-Ofast` optimization levels, or using `-fvectorize`), and compiled with the `-fsimdmath` option.

> **Note**
>
> For `armflang`, `-fsimdmath` is enabled by default.

Autovectorization with `-fsimdmath` is preventing bit-reproducibility

The `-fsimdmath` option allows the compiler to generate calls to vectorized library routines. The vectorized library routines might use different algorithms to the scaler routine algorithms, and bit-reproducibility between the two versions is not guaranteed. In other words, both the scalar and vector routines give the same result, but the scalar version might not give the exact same bits as the vector version.

Therefore, when `-fsimdmath` is used on your compile line alongside enabling autovectorization, the compiler might vectorize loops using calls to vectorized math routines, affecting the bit reproducibility.

Solution

If you require your code to be bit reproducible, compile your code using the `-fno-simdmath` option.

8.6 binutils does not automatically unload with module unload

Describes what to do if you have unloaded the Arm® Compiler for Linux modulefile, but still see 'binutils' loaded.

Conditions

The Arm Compiler for Linux modulefile has been loaded for a compiling session, and at the end of the compiling session, the Arm Compiler for Linux modulefile has been unloaded using the `module unload <modulefile>` command.

After unloading the Arm Compiler for Linux modulefile, the 'binutils' modulefile remains loaded. If you use any other utility tools that use 'binutils', such as ld or objdump, they will use the incorrect 'binutils' modulefile on your system.

> **Note**
>
> - If you use `module purge` to unload all loaded modulefiles, you will not experience this issue.
> - This unloading behavior applies to both Environment Modules-based systems and lmod environment modules-based systems.

Cause

'binutils' is required by the Arm Compiler for Linux and the binutils modulefile is automatically loaded when the Arm Compiler for Linux modulefile is loaded. However,the 'binutils' modulefile

is not automatically unloaded when the Arm Compiler for Linux modulefile is unloaded using the `module unload` command.

Workaround

At the end of your compiling session, you must explicitly unload both the Arm Compiler for Linux and 'binutils' modulefiles. Alternatively, you can unload all modulefiles at once, for example, using the `module purge` command.