Arm® C/C++ Compiler
Developer and reference guide
Copyright © 2018–2020 Arm Limited or its affiliates. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1900-00</td>
<td>02 November 2018</td>
<td>Non-Confidential</td>
<td>Document release for Arm® C/C++ Compiler version 19.0</td>
</tr>
<tr>
<td>1910-00</td>
<td>08 March 2019</td>
<td>Non-Confidential</td>
<td>Update for Arm® C/C++ Compiler version 19.1</td>
</tr>
<tr>
<td>1920-00</td>
<td>07 June 2019</td>
<td>Non-Confidential</td>
<td>Update for Arm® C/C++ Compiler version 19.2</td>
</tr>
<tr>
<td>1930-00</td>
<td>30 August 2019</td>
<td>Non-Confidential</td>
<td>Update for Arm® C/C++ Compiler version 19.3</td>
</tr>
<tr>
<td>2000-00</td>
<td>29 November 2019</td>
<td>Non-Confidential</td>
<td>Update for Arm® C/C++ Compiler version 20.0</td>
</tr>
<tr>
<td>2010-00</td>
<td>23 April 2020</td>
<td>Non-Confidential</td>
<td>Update for Arm® C/C++ Compiler version 20.1</td>
</tr>
<tr>
<td>2010-01</td>
<td>23 April 2020</td>
<td>Non-Confidential</td>
<td>Documentation update 1 for Arm® C/C++ Compiler version 20.1</td>
</tr>
<tr>
<td>2020-00</td>
<td>25 June 2020</td>
<td>Non-Confidential</td>
<td>Update for Arm® C/C++ Compiler version 20.2</td>
</tr>
</tbody>
</table>

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

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.

Web Address

www.arm.com
Contents
Arm® C/C++ Compiler Developer and reference guide

Preface
About this book ........................................................................................................................................... 9

Chapter 1 Get started
1.1 Arm® C/C++ Compiler .......................................................................................................................... 1-12
1.2 Get started with Arm® C/C++ Compiler .............................................................................................. 1-14
1.3 Get support .............................................................................................................................................. 1-16

Chapter 2 Compile and Link
2.1 Using the compiler .................................................................................................................................. 2-18
2.2 Compile C/C++ code for Arm SVE and SVE2-enabled processors .................................................... 2-21
2.3 Generate annotated assembly code from C and C++ code ................................................................. 2-23
2.4 Writing inline SVE assembly .................................................................................................................. 2-25

Chapter 3 Optimize
3.1 Coding best practice for auto-vectorization .......................................................................................... 3-31
3.2 Control auto-vectorization with pragmas ............................................................................................... 3-32
3.3 Optimizing C/C++ code with Arm SIMD (Neon™) .............................................................................. 3-35
3.4 Optimizing C/C++ code with SVE and SVE2 ..................................................................................... 3-36
3.5 Arm Optimization Report ....................................................................................................................... 3-37
3.6 Optimization remarks ............................................................................................................................ 3-42
3.7 Prefetching with __builtin_prefetch ..................................................................................................... 3-44
Chapter 4

Compiler options

4.1 Arm C/C++ Compiler Options by Function ............................................................ 4-54
4.2 -### ...................................................................................................................... 4-58
4.3 -armpl= ................................................................................................................ 4-59
4.4 -c .......................................................................................................................... 4-61
4.5 -config ........................................................................................................ 4-62
4.6 -D ............................................................................................................................ 4-63
4.7 -E ............................................................................................................................ 4-64
4.8 -fassociative-math ............................................................................................... 4-65
4.9 -fcolor-diagnostics .............................................................................................. 4-66
4.10 -fcxx-exceptions ................................................................................................. 4-67
4.11 -fdenormal-fp-math= ........................................................................................ 4-68
4.12 -fexcepions ......................................................................................................... 4-69
4.13 -ffast-math .......................................................................................................... 4-70
4.14 -ffinite-math-only ............................................................................................... 4-71
4.15 -ffp-contract= ..................................................................................................... 4-72
4.16 -fhonor-infinities ............................................................................................... 4-73
4.17 -fhonor-nans ...................................................................................................... 4-74
4.18 -finline ................................................................................................................ 4-75
4.19 -finline-functions ............................................................................................... 4-76
4.20 -finline-hint-functions ....................................................................................... 4-77
4.21 -fiterative-reciprocal ........................................................................................ 4-78
4.22 -fito ...................................................................................................................... 4-79
4.23 -fmath-errno ....................................................................................................... 4-80
4.24 -fnocrash-diagnostics ....................................................................................... 4-81
4.25 -fopenmp ............................................................................................................ 4-82
4.26 -fopenmp-simd .................................................................................................. 4-83
4.27 -freciprocal-math ............................................................................................. 4-84
4.28 -fsave-optimization-record ............................................................................... 4-85
4.29 -fsign-char ......................................................................................................... 4-86
4.30 -fsign-zeros ....................................................................................................... 4-87
4.31 -fsimdmath ....................................................................................................... 4-88
4.32 -fsstrict-aliasing .............................................................................................. 4-89
4.33 -fsyntax-only .................................................................................................... 4-90
4.34 -ftrapping-math ............................................................................................... 4-91
4.35 -funsafe-math-optimizations .......................................................................... 4-92
4.36 -fvectorize ....................................................................................................... 4-93
4.37 -g ......................................................................................................................... 4-94
4.38 -g0 ....................................................................................................................... 4-95
4.39 -gcc-toolchain= .............................................................................................. 4-96
4.40 -gline-tables-only ........................................................................................... 4-97
4.41 -help .................................................................................................................. 4-98
4.42 -help-hidden ..................................................................................................... 4-99
4.43 -I ........................................................................................................................ 4-100
4.44 -include .......................................................................................................... 4-101
4.45 -iquote .............................................................................................................. 4-102
4.46 -isysroot ........................................................................................................... 4-103
4.47 -isystem ............................................................................................................ 4-104
4.48 -isystem-after .............................................................. 4-105
4.49 -I ........................................................................ 4-106
4.50 -march= .............................................................. 4-107
4.51 -mcpu= ..................................................................... 4-108
4.52 -O ........................................................................ 4-109
4.53 -o ........................................................................ 4-110
4.54 -print-search-dirs .................................................. 4-111
4.55 -Qunused-arguments .............................................. 4-112
4.56 -S ........................................................................ 4-113
4.57 -shared ............................................................... 4-114
4.58 -static ............................................................... 4-115
4.59 -std= ............................................................... 4-116
4.60 -U ........................................................................ 4-117
4.61 -v ........................................................................ 4-118
4.62 -version .................................................................. 4-119
4.63 -W ........................................................................ 4-120
4.64 -Wall ...................................................................... 4-121
4.65 -Warm-extensions .................................................. 4-122
4.66 -Wdeprecates ......................................................... 4-123
4.67 -Wl, ...................................................................... 4-124
4.68 -Wp, ...................................................................... 4-125
4.69 -w ........................................................................ 4-126
4.70 -working-directory ................................................ 4-127
4.71 -Xassembler .......................................................... 4-128
4.72 -Xlinker ............................................................... 4-129
4.73 -Xpreprocessor ..................................................... 4-130

Chapter 5 Standards support

5.1 Supported C/C++ standards in Arm® C/C++ Compiler ........................................ 5-132
5.2 OpenMP 4.0 ............................................................ 5-133
5.3 OpenMP 4.5 ............................................................ 5-134

Chapter 6 Troubleshoot

6.1 Application segfaults at -Ofast optimization level ........................................... 6-136
6.2 Compiling with the -fpic option fails when using GCC compilers .................... 6-137
6.3 Error messages when installing Arm® Compiler for Linux ............................... 6-138
List of Tables

Arm® C/C++ Compiler Developer and reference guide

Table 5-1  Supported OpenMP 4.0 features ................................................................. 5-133
Table 5-2  Supported OpenMP 4.5 features ................................................................. 5-134
This preface introduces the *Arm® C/C++ Compiler Developer and reference guide*. It contains the following:

- *About this book on page 9.*
About this book


Using this book

This book is organized into the following chapters:

**Chapter 1 Get started**
This chapter introduces Arm C/C++ Compiler (part of Arm Compiler for Linux and Arm Allinea Studio), and describes how to get started with the compiler, and where to find further support.

**Chapter 2 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++`.

**Chapter 3 Optimize**
This chapter describes the optimization-specific features supported in Arm C/C++ Compiler.

**Chapter 4 Compiler options**
This chapter summarizes the supported options used with `armclang` and `armclang++`.

**Chapter 5 Standards support**
The support status of Arm C/C++ Compiler with the C/C++ language and OpenMP standards.

**Chapter 6 Troubleshoot**
Describes how to diagnose problems when compiling applications using Arm Fortran Compiler.

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.

Typographic conventions

*italic*
Introduces special terminology, denotes cross-references, and citations.

*bold*
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

*monospace*
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

*monospace* *italic*
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

*monospace* *bold*
Denotes language keywords when used outside example code.
<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**
Used in body text for a few terms that have specific technical meanings, that are defined in the Arm® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

**Feedback**

**Feedback on this product**
If you have any comments or suggestions about this product, contact your supplier and give:
• The product name.
• The product revision or version.
• An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

**Feedback on content**
If you have comments on content then send an e-mail to errata@arm.com. Give:
• The title *Arm C/C++ Compiler Developer and reference guide*.
• The number 101458_2020_00_en.
• If applicable, the page number(s) to which your comments refer.
• A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

**Note**
Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

**Other information**

• *Arm® Developer*.
• *Arm® Information Center*.
• *Arm® Technical Support Knowledge Articles*.
• Technical Support.
• *Arm® Glossary*. 
Chapter 1
Get started

This chapter introduces Arm C/C++ Compiler (part of Arm Compiler for Linux and Arm Allinea Studio), and describes how to get started with the compiler, and where to find further support.

It contains the following sections:

• 1.1 Arm® C/C++ Compiler on page 1-12.
• 1.2 Get started with Arm® C/C++ Compiler on page 1-14.
• 1.3 Get support on page 1-16.
1.1 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. It is built on the open-source Clang front-end and the LLVM-based optimization and code generation back-end.

Arm C/C++ Compiler supports modern C/C++ and OpenMP standards, has a built-in autovectorizer, and is tuned for the 64-bit Armv8-A architecture. It also supports compiling for Scalable Vector Extension (SVE) and SVE2-enabled target platforms.

Arm C/C++ Compiler is packaged with Arm Fortran Compiler and Arm Performance Libraries in a single package called Arm Compiler for Linux. Arm Compiler for Linux is available as part of Arm Allinea Studio.

Arm Allinea Studio is an end-to-end commercial suite for compiling, debugging, and optimizing Linux applications on Arm, and is comprised of Arm Compiler for Linux and Arm Forge.

The Arm Allinea Studio tools require a valid license to use them.

Arm® C/C++ Compiler resources

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

Arm Allinea Studio
- Arm Allinea Studio
- 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 Explore 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 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 the Troubleshoot on page 6-135
• Contact Arm Support
• Get software

——— Note ———

An HTML version of this guide is available in the <install_location>/<package_name>/share directory of your product installation.
1.2 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.

Prerequisites

Download and install Arm Compiler for Linux. You can download Arm Compiler for Linux from the Arm Allinea Studio Downloads 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, and you know the installation location of the tools, set the MODULEPATH environment variable to include the installation directory:

      ```
      export MODULEPATH=$MODULEPATH:<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/`.

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

      ```
      module load <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>
      ```

      For example:

      ```
      module load Generic-AArch64/SUSE/12/arm-linux-compiler/20.2
      ```

   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-20.2_Generic-AArch64_SUSE-12_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 (-o) the output binary file, `<binary>`, and the input source filename, `<source>.c/cpp`:

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

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

   To run your binary, use:

   ```
   ./<binary>
   ```
Example 1-1 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 <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>

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

   ```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 (-o) the input file, hello.c, and the binary name, 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 on page 2-17.
1.3 Get support

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

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

or

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

A description of each supported command-line option is available in *Compiler options on page 4-52*.

If you encounter a problem when developing your application and compiling with the Arm Compiler for Linux, see the *Troubleshoot on page 6-135* topic.

If you encounter a problem when using Arm Compiler for Linux, contact the Arm Support team:

*Contact Arm Support*
Chapter 2
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++.

It contains the following sections:
• 2.1 Using the compiler on page 2-18.
• 2.2 Compile C/C++ code for Arm SVE and SVE2-enabled processors on page 2-21.
• 2.3 Generate annotated assembly code from C and C++ code on page 2-23.
• 2.4 Writing inline SVE assembly on page 2-25.
2.1 Using the compiler

Describes how to generate executable binaries, compile and link object files, and enable optimization options.

Compile and link

To generate an executable binary, for example example1, compile the source file example1.c using:

```
armclang -o example1 example1.c
```

You can also specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary. For example:

```
armclang -o example1 example1a.c example1b.c
```

To compile each of your source files individually into an object file, specify the -c (compile-only) option, and then pass the resulting object files into another invocation of armclang to link them into an executable binary.

```
armclang -c -o example1a.o example1a.c
armclang -c -o example1b.o example1b.c
armclang -o example1 example1a.o example1b.o
```

Increase the optimization level

To increase the optimization level, use the -O<level> option. The -O0 option is the lowest optimization level, while -O3 is the highest. Arm C/C++ Compiler only performs auto-vectorization at -O2 and higher, and uses -O0 as the default setting. The optimization option can be specified when generating a binary, such as:

```
armclang -O3 -o example1 example1.c
```

The optimization option can also be specified when generating an object file:

```
armclang -O3 -c -o example1a.o example1a.c
armclang -O3 -c -o example1b.o example1b.c
```

or when linking object files:

```
armclang -O3 -o example1 example1a.o example1b.o
```

Compile and optimize using CPU auto-detection

Arm C/C++ Compiler supports the use of the -mcpu=native option, for example:

```
armclang -O3 -mcpu=native -o example1 example1.c
```

This option enables the compiler to automatically detect the architecture and processor type of the CPU you are running the compiler on, and optimize accordingly.

This option supports a range of Armv8-A-based SoCs, including ThunderX2, Neoverse N1, and A64FX.

--- Note ---

The optimization performed according to the auto-detected architecture and processor is independent of the optimization level that is denoted by the -O<level> option.

Common compiler options

See man armclang, armclang --help, or Compiler options on page 4-52, for more information about all the supported 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 .o file. 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 your target microarchitecture, Arm recommends using the -mcpu option instead of -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 architecture profile for that CPU. If you use -mcpu, you do not need to use the -march option.

mcpu supports a range of Armv8-A-based System-on-Chips (SoCs), including ThunderX2, Neoverse N1, and A64FX.

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

-Olevel
Specifies the level of optimization to use when compiling source files. The default is -O0.

--config /path/to/<config-file>.cfg
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 on page 4-52.

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

man {armclang|armclang++}
Related tasks
2.2 Compile C/C++ code for Arm SVE and SVE2-enabled processors on page 2-21

Related references
Chapter 4 Compiler options on page 4-52
1.3 Get support on page 1-16
2.2 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.

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 tutorial shows you how to compile code to take advantage of SVE (or SVE2) functionality. The executable that is generated during the tutorial can only be run on SVE-enabled (or SVE2-enabled) hardware, or with Arm Instruction Emulator.

Prerequisites

• 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 <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>

Procedure

1. Compile your SVE or SVE2 source and specify an SVE-enabled (or SVE2-enabled) architecture:

   - To compile without linking to Arm Performance Libraries, set -march to the architecture and feature set you want to target:
     For SVE:
     ```bash
     armclang -O<level> -march=armv8-a+sve -o <binary-filename> <source-filename>.c
     ```
     For SVE2:
     ```bash
     armclang -O<level> -march=armv8-a+sve2 -o <binary-filename> <source-filename>.c
     ```

   - To compile and link to the SVE version of Arm Performance Libraries, set -march to the architecture and feature set you want to target and add the -armpl=sve option to your command line:
     For SVE:
     ```bash
     armclang -O<level> -march=armv8-a+sve -armpl=sve -o <binary-filename> <source-filename>.c
     ```
     For SVE2:
     ```bash
     armclang -O<level> -march=armv8-a+sve2 -armpl=sve -o <binary-filename> <source-filename>.c
     ```

     For more information about the supported options for -armpl, see the -armpl description in Arm C/C++ Compiler Options by Function on page 4-54.

     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 Arm C/C++ Compiler Options by Function on page 4-54.

     ______ Note ______
     sve2 also enables sve.

2. Run the executable:

   ./<binary-filename>
Example 2-1 Example

This example compiles an example application source into assembly with auto-vectorization enabled.

The following C program 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
// example1.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 `example1.c` and specify the output file to be assembly (-S):

   ```
   armclang -O3 -S -march=armv8-a+sve -o example1.s example1.c
   ```

   The output assembly code is saved as `example1.s`.

2. (Optional) Inspect the output assembly code.

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

   ```assembly
   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
   stlw    {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, that is independent of the width of SVE vector registers.

3. Run the executable:

   ```
   ./example1
   ```

Related information

Porting and Optimizing HPC Applications for Arm SVE
2.3 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.

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 on page 2-21.

Prerequisites

- 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 <architecture>/<linux_variant>/<linux_version>/arm-linux-compiler/<version>
```

Procedure

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

   ```bash
   armclang -O<level> -S -o <assembly-filename>.s <source-filename>.c
   ```

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

   The `-O<level>` option specifies the optimization level. The `-O0` option is the lowest optimization level, while `-O3` is the highest. Arm C/C++ Compiler only performs auto-vectorization at `-O2` and higher.

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

3. Run the executable:

   ```bash
   ./<binary-filename>
   ```

Example 2-2 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
// example1.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 example1.s example1.c
```

The output assembly code is saved as `example1.s`. The section of the generated assembly language file that contains the compiled `subtract_arrays` function is as follows:

```
subtract_arrays:                        // @subtract_arrays
   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 example1.s example1.c
```

The output assembly code is saved as `example1.s`. The section of the generated assembly language file that contains the compiled `subtract_arrays` function is as follows:

```
subtract_arrays:                        // @subtract_arrays
   mov     x8, xzr
.LBB0_1:                                // =>This Inner Loop Header: Depth=1
   add     x9, x0, #16                  // =16
   add     x10, x1, x8             // =>This Inner Loop Header: Depth=1
   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.
2.4 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)
```

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.

```c
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:

```assembly
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).

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.

```c
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:

```c
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:

```c
void f(double *restrict x, double *restrict a, double *restrict b, double *restrict c, unsigned long n)
```
```c
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:

```c
void f(double *x, double *a, double *b, double *c, unsigned long n)
{
    unsigned long i;
    asm (
        "whilelo p0.d, %[i], %[n]               
        
        ld1d z0.d, p0/2, [%[a], %[i], lsl #3]  
        ld1d z1.d, p0/2, [%[b], %[i], lsl #3]  
        ld1d z2.d, p0/2, [%[c], %[i], lsl #3]  
        fmla z2.d, p0/m, z0.d, z1.d            
        st1d z2.d, p0, [%[x], %[i], lsl #3]    
        uqincd %[i]                            
        whilelo p0.d, %[i], %[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.
Chapter 3
Optimize

This chapter describes the optimization-specific features supported in Arm C/C++ Compiler.

It contains the following sections:

- 3.2 Control auto-vectorization with pragmas on page 3-32.
- 3.3 Optimizing C/C++ code with Arm SIMD (Neon™) on page 3-35.
- 3.4 Optimizing C/C++ code with SVE and SVE2 on page 3-36.
- 3.6 Optimization remarks on page 3-42.
- 3.7 Prefetching with __builtin_prefetch on page 3-44.
- 3.8 Vector routines support on page 3-46.
3.1 Coding best practice for auto-vectorization

Describes some best practices to follow to optimize your code for auto-vectorization.

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

**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* on page 3-32.

**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.
3.2 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 optimization level `-O2` or higher. 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 pragma above 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.

```c
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:

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

## Suppress 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:

```c
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;
    }
}
```

You can also suppress SVE instructions while allowing Arm Neon instructions by adding a `vectorize_style` hint:

### vectorize_style(fixed_width)

Prefer fixed-width vectorization, resulting in Arm Neon instructions. For a loop with `vectorize_style(fixed_width)`, 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_style(scaled_width) (default)

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

For example:

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

## Unrolling and interleaving with pragmas

To better use processor resources, duplicate loops to reduce the loop iteration count and increase the Instruction-Level Parallelism (ILP). For scalar loops, the method is called **unrolling**. For vectorizable loops, it is **interleaving** that is performed.

### Unrolling

Unrolling a scalar loop, for example:

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

by a factor of two, gives:

```c
for (int i = 0; i < 32; i +=2) {
    data[i] = input[i] * other[i];
}
```
data[i+1] = input[i+1] * other[i+1];
}

For the example above, the unrolling factor (UF) is two. To unroll to the internal limit, the unroll pragma is inserted before the loop:

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

To unroll to a user-defined UF, instead insert:

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

**Interleaving**

To interleave, an Interleaving Factor (IF) is used instead of a UF. To accurately generate interleaved code, the loop vectorizer models the cost on the register pressure and the generated code size. When a loop is vectorized, the interleaved code can be more optimal than unrolled code.

Like the UF, the IF can be the internal limit or a user-defined integer. To interleave to the internal limit, the interleave pragma is inserted before the loop:

```c
#pragma clang loop interleave(enable)
```

To interleave to a user-defined IF, instead insert:

```c
#pragma clang loop interleave_count(_value_)
```

--- **Note** ---

Interleaving performed on a scalar loop does not unroll the loop correctly.
3.3 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. There are SIMD instruction sets for both AArch32 (equivalent to the Armv7 instructions) and for AArch64. Both can be used 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 for you.

  Arm C/C++ Compiler automatically vectorizes your code at higher optimization levels (-O2 and higher). The compiler identifies appropriate vectorization opportunities in your code and uses SIMD 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.

• 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 allowing the compiler to handle instruction scheduling and register allocation. The available intrinsics are defined in the language extensions document.

• Write SIMD assembly code.

  Although it is technically possible to optimize SIMD assembly by hand, it can be difficult because the pipeline and memory access timings have complex inter-dependencies. Instead of hand-writing assembly, Arm recommends the use of intrinsics.
3.4 Optimizing C/C++ code with SVE and SVE2

The Scalable Vector Extension (SVE and SVE2) to the Armv8-A architecture (AArch64) can be used to accelerate repetitive operations on the large data sets commonly encountered with High Performance Computing (HPC) applications.

SVE (and SVE2) instructions pack multiple lanes of data into large registers then perform the same operation across all data lanes, with predication to control which lanes are active. For example, consider the following SVE instruction:

```
ADD Z0.D, P0/M, Z1.D, Z2.D
```

The instruction specifies that an addition (ADD) operation is performed on a SVE vector register, split into 64-bit data lanes. D specifies the width of the data lane (doubleword, or 64 bits). The width of each vector register is some multiple of 128 bits, between 128 and 2048, but is not specified by the architecture. The predicate register P0 specifies which lanes must be active. Each active lane in Z1 is added to the corresponding lane in Z2 and the result is stored in Z0. Each lane is added separately. There are no carries between the lanes. The merge flag /M on the predicate specifies that inactive lanes retain their prior value.

Optimize your code for SVE

To optimize your code using SVE, you can either:

- Let the compiler auto-vectorize your code for you.
  
  Arm Compiler for Linux automatically vectorizes your code at optimization levels -O2 and higher. 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 on page 4-54 for more information on setting these options.

- Write SVE assembly code.

  For more information, see Writing inline SVE assembly on page 2-25.

  For more information about porting and optimizing existing applications to Arm SVE, see the Porting and Tuning HPC Applications for Arm SVE guide.

Related information

- Scalable Vector Extension (SVE, and SVE2) information
- Explore 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 64-bit Architecture (AArch64) with SVE support
- 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
3.5 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 = \frac{\text{true SVE vector width}}{128}

Loops vectorized using scalable vectors are annotated with \text{VS}<F,I>. For more information, see arm-opt-report reference on page 3-39.

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> \times <vectorization factor> \times <interleave count> \times <SVE scaling factor>

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 the Arm Optimization Report reference.

This section contains the following subsections:

- 3.5.1 How to use Arm Optimization Report on page 3-38.
- 3.5.2 arm-opt-report reference on page 3-39.
3.5.1 How to use Arm Optimization Report

This topic describes how to use Arm Optimization Report.

Prerequisites

Download and install Arm Compiler for Linux version 20.0+. 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.

An <filename>.opt.yaml report is generated by Arm 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 3-1  Example

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

   ```c
   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:

   ```sh
   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:

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

   Annotated source code is displayed in the terminal:
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 was unrolled 16 times. This means it is completely unrolled, with no remaining loops.
- All three instances of `foo()` are inlined

### Related references

3.5.2 arm-opt-report reference on page 3-39

### Related information

Arm Compiler for Linux and Arm Allinea Studio

**Take a trial**

**Help and tutorials**

#### 3.5.2 arm-opt-report reference

Arm Optimization Report (arm-opt-report) is a tool to generate an optimization report from YAML optimization record files.

`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.

---

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I</strong></td>
<td>A function was inlined.</td>
</tr>
<tr>
<td><strong>U&lt;N&gt;</strong></td>
<td>A loop was unrolled <code>&lt;N&gt;</code> times.</td>
</tr>
<tr>
<td><strong>V&lt;F, I&gt;</strong></td>
<td>A loop has been vectorized.</td>
</tr>
<tr>
<td></td>
<td>Each vector iteration that is performed has the equivalent of <code>F*I</code> scalar iterations.</td>
</tr>
<tr>
<td></td>
<td>Vectorization Factor, <code>F</code>, is the number of scalar elements that are processed in parallel.</td>
</tr>
<tr>
<td></td>
<td>Interleave count, <code>I</code>, is the number of times the vector loop was unrolled.</td>
</tr>
<tr>
<td><strong>VS&lt;F,I&gt;</strong></td>
<td>A loop has been vectorized using scalable vectors.</td>
</tr>
<tr>
<td></td>
<td>Each vector iteration performed has the equivalent of <code>N*F*I</code> scalar iterations, where <code>N</code> is the number of vector granules, which can vary according to the machine the program is run on.</td>
</tr>
<tr>
<td></td>
<td>For example, LLVM assumes a granule size of 128 bits when targeting SVE.</td>
</tr>
<tr>
<td></td>
<td><code>F</code> (Vectorization Factor) and <code>I</code> (Interleave count) are as described for <strong>V&lt;F, I&gt;</strong>.</td>
</tr>
</tbody>
</table>
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 tasks
3.5.1 How to use Arm Optimization Report on page 3-38

Related references
3.5.2 arm-opt-report reference on page 3-39
3.6 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 C/C++ Compiler prints compilation information to stderr. Optimization remarks prints this optimization information to the terminal, or you can choose to pipe them to an output file.

To enable optimization remarks, choose from following Rpass options:

- `-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 an expression for the type of remarks you wish to view.

Recommended `<regexp>` queries are:

- `-Rpass=(loop-vectorize|inline|loop-unroll)`
- `-Rpass-missed=(loop-vectorize|inline|loop-unroll)`
- `-Rpass-analysis=(loop-vectorize|inline|loop-unroll)`

where `loop-vectorize` filters remarks regarding vectorized loops, `inline` for remarks regarding inlining, and `loop-unroll` for remarks about unrolled loops.

Note

To search for all remarks, use the expression `.*`. Use this expression with caution; depending on the size of code, and the level of optimization, a lot of information can print.

To compile with optimization remarks enabled and pipe the information to an output file, pass the selected above options and debug information to `armclang`, and use `> <output_filename>.txt`. For example:

```
armclang -O<level> -Rpass[-<option>]=<remark> <filename>.c 2> <output_filename>.txt
```

This section contains the following subsection:

- 3.6.1 Enable Optimization remarks on page 3-42.

3.6.1 Enable Optimization remarks

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

Procedure

1. Compile your code. Use the `-Rpass=<regex>`, `-Rpass-missed=<regex>`, or `Rpass-analysis=<regex>` options:

   For example, for an input file `example.c`:

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

   Result:

   ```
   example.c:8:18: remark: hoisting `zext' [-Rpass=licm]
   ^
   for (int i=0;i<K; i++)
   ```

   ```
   example.c:8:4: remark: vectorized loop (vectorization width: 4, interleaved count: 2) [-Rpass=loop-vectorize]
   ^
   for (int i=0;i<K; i++)
   ```

   ```
   example.c:7:1: remark: 28 instructions in function [-Rpass-analysis=asm-printer]
   ^
   void foo(int K) {
   ```
2. Pipe the loop vectorization optimization remarks to a file. For example, to pipe to a file called vecreport.txt, use:

```
armclang -O3 -Rpass=loop-vectorize -Rpass-analysis=loop-vectorize
-Rpass-missed=loop-vectorize example.c 2> vecreport.txt
```

Alternatively, to enable optimization remarks and pipe the output information to a file, use:

```
armclang -O<level> -Rpass[-<option>]=<remark> example.c 2> <output_filename>.txt
```

A vecreport.txt file is output with the optimization remarks in it.

**Related information**

*Arm C/C++ Compiler*

**Related tasks**

3.6.1 Enable Optimization remarks on page 3-42
3.7 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 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 in 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
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) {        // L3 load prefetch (locality)
    __builtin_prefetch(foo, 0, 1);
}
void l2_load(void *foo) {        // L2 load prefetch (locality)
    __builtin_prefetch(foo, 0, 2);
}
```
```c
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 the `-c -march=armv8-a -O3` compiler options, generates the following assembly:

```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
```

Related information

Explore the Scalable Vector Extension (SVE)

SVE Vector Length Agnostic programming
3.8 Vector routines support

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.

This section contains the following subsections:

• 3.8.1 How to vectorize math routines in Arm® C/C++ Compiler on page 3-46.
• 3.8.2 How to declare custom vector routines in Arm® C/C++ Compiler on page 3-47.

3.8.1 How to vectorize math routines in Arm® C/C++ Compiler

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

Any C loop-using functions from <math.h> (or from <cmath> for C++) can be vectorized by invoking the compiler with the option -fsimdmath, together with the options that are needed to activate the auto-vectorizer (optimization level -O2 and above).

Examples

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

```
armclang -fsimdmath -c -O2 source.c
```

C example with loop invoking sin:

```c
/* C code example: source.c */
#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 */
    }
}
```

C++ example with loop invoking std::pow:

```cpp
// C++ code example: source.cpp
#include <cmath>
void do_something(float * a, float * b, unsigned N) {
    for (unsigned i = 0; i < N; ++i) {
        // some computation
        a[i] = std::pow(a[i], 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.

*Contact Arm Support*

*Related information*

SLEEF

Arm Optimized Routines

PGMath

Vector function ABI specification for AArch64

### 3.8.2 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 example shows the basic functionality for Advanced SIMD vectorization:

```c
#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);
```

To compile the code, invoke armclang with either the `-fopenmp` or the `-fopenmp-simd` options (automatic loop vectorization is activated starting from optimization level -O2):

```
$> armclang -fopenmp -O2 -c user_code.c -o objfile.o
```

You must link the output object file against an object file or library that provides the symbol `neon_foo`.

The following example shows the basic functionality for SVE vectorization:

```c
#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);
```

To compile the code, invoke armclang with either the `-fopenmp` or the `-fopenmp-simd` options (automatic loop vectorization is activated starting from optimization level -O2):

```
armclang -march=armv8-a+sve -fopenmp -O2 -c user_code.c -o objfile.o
```

You must link the output object file against an object file or library that provides the symbol `sve_foo`.
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 $WDS(\phi) \times N$ is either 8 or 16.

  $\phi$ is the name of the scalar function the directive applies to. For a definition of $WDS(\phi)$, 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 "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 $T_y^n$ maps to the correspondent Neon ACLE type $<T_y^n>xN_t$, where $N$ is the value that is specified in the $\text{simdlen}(N)$ clause. Values of $N$ that do not correspond to Neon ACLE types are unsupported.

2. If you specify $\text{inbranch}$, an extra mask parameter is added as the last parameter of the vector signature. The type of the parameter is the Neon ACLE type $\text{uint}<\text{BITS}xN>_t$, where:
   a. $N$ is the value that is specified in the $\text{simdlen}(N)$ clause.
   b. $\text{BITS}$ 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 $\text{uint}<\text{BITS}>_t$ integer in the mask vector.

Mapping rules for SVE
1. Each parameter type $T_y^n$ is mapped to the correspondent SVE ACLE type $\text{sve}<T_y^n>_t$.
2. An extra mask parameter of type $\text{svbool}_t$ is always added to the signature of the vector function, whether $\text{inbranch}$ or $\text{notinbranch}$ 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 $\text{svfloat64}_t \ F(\text{svfloat32}_t \ vx, \ \text{svbool}_t)$, 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 $00000000$.

Examples
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 to allow the loop auto-vectorization process.
- `-fopenmp` to enable the parsing of the OpenMP directives.

```
#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]);
}
```

Examples: Advanced SIMD
Simple:

```
#include <arm_neon.h>  
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo(float64x2_t a);
```

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:
To produce a vector loop that invokes user_vector_foo, compile the example code with armclang -fopenmp -O2 -c -S -o - example01.c -mllvm -force-vector-interleave=1:

```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:

```c
// filename: 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, 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 this code with armclang -fopenmp -O2 -c -S -o - example02.c -mllvm -force-vector-interleave=1:

```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]
  subs   x24, x24, #2            // =2
  add    v1.2d, v1.2d, v0.2d
b.ne    .LBB0_4
```

### Examples: SVE

**Simple:**

```c
// filename: 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 this code with armclang example03.c -march=armv8-a+sve -O2 -o -S -fopenmp:

```c
.LBB0_2:                                // %vector.body
  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
```
With linear:

```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 `example04.c -march=armv8-a+sve -O2 -o -S -fopenmp`:

```
.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, x21, x22
    b.mi .LBB0_2
```

Related concepts

3.8.1 How to vectorize math routines in Arm® C/C++ Compiler on page 3-46
3.8.2 How to declare custom vector routines in Arm® C/C++ Compiler on page 3-47
Chapter 4
Compiler options

This chapter summarizes the supported options used with 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. 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.

It contains the following sections:
• 4.1 Arm C/C++ Compiler Options by Function on page 4-54.
• 4.2 -### on page 4-58.
• 4.3 -armpl= on page 4-59.
• 4.4 -c on page 4-61.
• 4.5 -config on page 4-62.
• 4.6 -D on page 4-63.
• 4.7 -E on page 4-64.
• 4.8 -fassociative-math on page 4-65.
• 4.9 -fcolor-diagnostics on page 4-66.
• 4.10 -fcxx-exceptions on page 4-67.
• 4.11 -fdenormal-fp-math= on page 4-68.
• 4.12 -fexceptions on page 4-69.
• 4.13 -ffast-math on page 4-70.
• 4.14 -ffinite-math-only on page 4-71.
• 4.15 -ffp-contract= on page 4-72.
• 4.16 -fhoonor-infinities on page 4-73.
• 4.17 -fhoonor-nans on page 4-74.
• 4.18 -ffinline on page 4-75.
• 4.19 -ffinline-functions on page 4-76.
4.20 -finline-hint-functions on page 4-77.
4.21 -fiterative-reciprocal on page 4-78.
4.22 -f1to on page 4-79.
4.23 -fmath-errno on page 4-80.
4.24 -fno-crash-diagnostics on page 4-81.
4.25 -fopenmp on page 4-82.
4.26 -fopenmp-simd on page 4-83.
4.27 -freciprocal-math on page 4-84.
4.28 -fsave-optimization-record on page 4-85.
4.29 -fsigend-char on page 4-86.
4.30 -fsigend-zeros on page 4-87.
4.31 -fismath on page 4-88.
4.32 -fstrict-aliasing on page 4-89.
4.33 -fsyntax-only on page 4-90.
4.34 -ftrapping-math on page 4-91.
4.35 -funsafe-math-optimizations on page 4-92.
4.36 -fvectorize on page 4-93.
4.37 -g on page 4-94.
4.38 -g0 on page 4-95.
4.39 -gcc-toolchain= on page 4-96.
4.40 -gline-tables-only on page 4-97.
4.41 -help on page 4-98.
4.43 -l on page 4-100.
4.44 -include on page 4-101.
4.45 -iquote on page 4-102.
4.46 -isysroot on page 4-103.
4.47 -isystem on page 4-104.
4.48 -isystem-after on page 4-105.
4.49 -l on page 4-106.
4.50 -march= on page 4-107.
4.51 -mcpu= on page 4-108.
4.52 -O on page 4-109.
4.53 -o on page 4-110.
4.54 -print-search-dirs on page 4-111.
4.55 -Qunused-arguments on page 4-112.
4.56 -S on page 4-113.
4.57 -shared on page 4-114.
4.58 -static on page 4-115.
4.59 -std= on page 4-116.
4.60 -U on page 4-117.
4.61 -v on page 4-118.
4.62 -version on page 4-119.
4.63 -W on page 4-120.
4.64 -Wall on page 4-121.
4.65 -Warm-extensions on page 4-122.
4.66 -Wdeprecated on page 4-123.
4.67 -Wl, on page 4-124.
4.68 -Wp, on page 4-125.
4.69 -w on page 4-126.
4.70 -working-directory on page 4-127.
4.71 -Xassembler on page 4-128.
4.72 -Xlinker on page 4-129.
4.73 -Xpreprocessor on page 4-130.
4.1 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.7 -E on page 4-64</td>
<td>Only run the preprocessor.</td>
</tr>
<tr>
<td>4.56 -S on page 4-113</td>
<td>Only run preprocess and compilation steps.</td>
</tr>
<tr>
<td>4.4 -c on page 4-61</td>
<td>Only run preprocess, compile, and assemble steps.</td>
</tr>
<tr>
<td>4.10 -fcxx-exceptions on page 4-67</td>
<td>Enable C++ exceptions.</td>
</tr>
<tr>
<td>4.12 -fexceptions on page 4-69</td>
<td>Enable support for exception handling.</td>
</tr>
<tr>
<td>4.19 -finline-functions on page 4-76</td>
<td>Inline suitable functions.</td>
</tr>
<tr>
<td>4.20 -finline-hint-functions on page 4-77</td>
<td>Inline functions which are (explicitly or implicitly) marked inline.</td>
</tr>
<tr>
<td>4.25 -fopenmp on page 4-82</td>
<td>Enable OpenMP and link in the OpenMP library, libomp.</td>
</tr>
<tr>
<td>4.26 -fopenmp-simd on page 4-83</td>
<td>Enable processing of ‘simd’ and the ‘declare simd’ pragma, without enabling OpenMP or linking in the OpenMP library, libomp. Enabled by default.</td>
</tr>
<tr>
<td>4.33 -fsyntax-only on page 4-90</td>
<td>Show syntax errors but do not perform any compilation.</td>
</tr>
</tbody>
</table>

**File options**

Options that specify input or output files.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.43 -I on page 4-100</td>
<td>Add directory to include search path. Directories specified with the -I option apply to both the quote form of the include directive and the system header form.</td>
</tr>
<tr>
<td>4.5 -config on page 4-62</td>
<td>Passes the location of a configuration file to the compile command.</td>
</tr>
<tr>
<td>4.44 -include on page 4-101</td>
<td>Include file before parsing.</td>
</tr>
<tr>
<td>4.45 -iquote on page 4-102</td>
<td>Add directory to the include search path.</td>
</tr>
<tr>
<td>4.46 -isysroot on page 4-103</td>
<td>For header files, set the system root directory (usually /).</td>
</tr>
<tr>
<td>4.47 -isystem on page 4-104</td>
<td>Override the system include directory.</td>
</tr>
<tr>
<td>4.48 -isystem-after on page 4-105</td>
<td>Add directory to end of the SYSTEM include search path.</td>
</tr>
<tr>
<td>4.53 -o on page 4-110</td>
<td>Write output to &lt;file&gt;.</td>
</tr>
<tr>
<td>4.70 -working-directory on page 4-127</td>
<td>Resolve file paths relative to the specified directory.</td>
</tr>
</tbody>
</table>

**Basic driver options**

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 -### on page 4-58</td>
<td>Print (but do not run) the commands to run for this compilation.</td>
</tr>
<tr>
<td>4.39 -gcc-toolchain= on page 4-96</td>
<td>Use the gcc toolchain at the given directory.</td>
</tr>
<tr>
<td>4.41 -help on page 4-98</td>
<td>Display available options.</td>
</tr>
</tbody>
</table>
### Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4.42 -help-hidden on page 4-99</strong></td>
<td>Display hidden options. Only use these options if advised to do so by your Arm representative.</td>
</tr>
<tr>
<td><strong>4.54 -print-search-dirs on page 4-111</strong></td>
<td>Print the paths that are used for finding libraries and programs.</td>
</tr>
<tr>
<td><strong>4.61 -v on page 4-118</strong></td>
<td>Show commands to run and use verbose output.</td>
</tr>
<tr>
<td><strong>4.62 -version on page 4-119</strong></td>
<td>Show the version number and some other basic information about the compiler.</td>
</tr>
</tbody>
</table>

### Optimization options

Options that control what optimizations should be performed.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4.52 -O on page 4-109</strong></td>
<td>Specifies the level of optimization to use when compiling source files.</td>
</tr>
<tr>
<td><strong>4.3 -armpl= on page 4-59</strong></td>
<td>Enable Arm Performance Libraries (ArmPL) (disabled by default).</td>
</tr>
<tr>
<td><strong>4.8 -fassociative-math on page 4-65</strong></td>
<td>Allow (-fassociative-math) or prevent (-fno-associative-math) the re-association of operands in a series of floating-point operations. Default is -fno-associative-math.</td>
</tr>
<tr>
<td><strong>4.11 -fdenormal-fp-math= on page 4-68</strong></td>
<td>Specify the denormal numbers the code is allowed to require.</td>
</tr>
<tr>
<td><strong>4.13 -ffast-math on page 4-70</strong></td>
<td>Allow aggressive, lossy floating-point optimizations (disabled by default).</td>
</tr>
<tr>
<td><strong>4.14 -ffinite-math-only on page 4-71</strong></td>
<td>Enable optimizations that ignore the possibility of NaN and +/-Inf (disabled by default).</td>
</tr>
<tr>
<td><strong>4.15 -ffp-contract= on page 4-72</strong></td>
<td>Controls when the compiler is permitted to form fused floating-point operations (for example, FMAs).</td>
</tr>
<tr>
<td><strong>4.16 -fhlont-infinities on page 4-73</strong></td>
<td>Do not allow optimizations that assume the arguments and results of floating point arithmetic are not +/-Inf.</td>
</tr>
<tr>
<td><strong>4.17 -fhlont-nans on page 4-74</strong></td>
<td>Do not allow optimizations that assume the arguments and results of floating point arithmetic are not NaN.</td>
</tr>
<tr>
<td><strong>4.18 -finline on page 4-75</strong></td>
<td>Enable/disable inlining (enabled by default).</td>
</tr>
<tr>
<td><strong>4.21 -fiterative-reciprocal on page 4-78</strong></td>
<td>Allow optimizations that replace division by reciprocal estimation and refinement.</td>
</tr>
<tr>
<td><strong>4.22 -flio on page 4-79</strong></td>
<td>Enable (-flio) or disable (-fno-lto) Link Time Optimizations (LTO).</td>
</tr>
<tr>
<td><strong>4.23 -fnmath-errno on page 4-80</strong></td>
<td>Require math functions to indicate errors.</td>
</tr>
<tr>
<td><strong>4.27 -freciprocional-math on page 4-84</strong></td>
<td>Allow division operations to be reassigned.</td>
</tr>
<tr>
<td><strong>4.28 -fsave-optimization-record on page 4-85</strong></td>
<td>Enable (-fsave-optimization-record) or disable (-fno-save-optimization-record) the generation of a YAML optimization record file. Default is -fno-save-optimization-record.</td>
</tr>
<tr>
<td><strong>4.29 -fsangned-char on page 4-86</strong></td>
<td>Set the type of ‘char’ to be signed. Disabled by default.</td>
</tr>
<tr>
<td><strong>4.30 -fsigned-zeros on page 4-87</strong></td>
<td>Do not allow optimizations that ignore the sign of floating point zeros. Enabled by default.</td>
</tr>
<tr>
<td><strong>4.31 -fsimdmath on page 4-88</strong></td>
<td>Enables the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h</td>
</tr>
<tr>
<td><strong>4.32 -fstrict-aliasing on page 4-89</strong></td>
<td>Tells the compiler to adhere to the aliasing rules defined in the source language (enabled by default when using '-Ofast').</td>
</tr>
<tr>
<td><strong>4.34 -ftrapping-math on page 4-91</strong></td>
<td>Tell the compiler to assume (-ftrapping-math), or not to assume (-fno-trapping-math), that floating point operations can trap.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>4.35 -funsafe-math-optimizations on page 4-92</td>
<td>Enable a subset of the optimizations allowed by -ffast-math.</td>
</tr>
<tr>
<td>4.36 -fvectorize on page 4-93</td>
<td>Enable/disable loop vectorization (enabled by default).</td>
</tr>
<tr>
<td>4.50 -march= on page 4-107</td>
<td>Specifies the base architecture and extensions available on the target.</td>
</tr>
<tr>
<td>4.51 -mcpu= on page 4-108</td>
<td>Select which CPU architecture to optimize for.</td>
</tr>
</tbody>
</table>

**C/C++ Options**

Options that affect the way C workloads are compiled.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.59 -std= on page 4-116</td>
<td>Language standard to compile for.</td>
</tr>
</tbody>
</table>

**Development options**

Options that facilitate code development.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9 -fcolor-diagnostics on page 4-66</td>
<td>Use colors in diagnostics.</td>
</tr>
<tr>
<td>4.37 -g on page 4-94</td>
<td>Generate source-level debug information.</td>
</tr>
<tr>
<td>4.38 -g0 on page 4-95</td>
<td>Disable generation of source-level debug information (default).</td>
</tr>
<tr>
<td>4.40 -gline-tables-only on page 4-97</td>
<td>Emit debug line number tables only.</td>
</tr>
</tbody>
</table>

**Warning options**

Options that control the behavior of warnings.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.55 -Qunused-arguments on page 4-112</td>
<td>Do not emit a warning for unused driver arguments.</td>
</tr>
<tr>
<td>4.63 -W on page 4-120</td>
<td>Enable the specified warning.</td>
</tr>
<tr>
<td>4.64 -Wall on page 4-121</td>
<td>Enable all warnings.</td>
</tr>
<tr>
<td>4.65 -Warm-extensions on page 4-122</td>
<td>Enable warnings about the use of non-standard language features supported by armclang or armflang.</td>
</tr>
<tr>
<td>4.66 -Wdeprecated on page 4-123</td>
<td>Enable warnings for deprecated constructs and define __DEPRECATED.</td>
</tr>
<tr>
<td>4.24 -fno-crash-diagnostics on page 4-81</td>
<td>Disable the auto-generation of preprocessed source files and a script for reproduction during a clang crash.</td>
</tr>
<tr>
<td>4.69 -w on page 4-126</td>
<td>Suppress all warnings.</td>
</tr>
</tbody>
</table>

**Preprocessor options**

Options controlling the behavior of the preprocessor.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6 -D on page 4-63</td>
<td>Define &lt;macro&gt; to &lt;value&gt; (or 1 if &lt;value&gt; omitted).</td>
</tr>
<tr>
<td>4.60 -U on page 4-117</td>
<td>Undefine macro &lt;macro&gt;.</td>
</tr>
</tbody>
</table>
### Linker options

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.67 -Wl, on page 4-124</td>
<td>Pass the comma separated arguments in &lt;arg&gt; to the linker.</td>
</tr>
<tr>
<td>4.71 -Xassembler on page 4-128</td>
<td>Pass &lt;arg&gt; to the assembler.</td>
</tr>
<tr>
<td>4.72 -Xlinker on page 4-129</td>
<td>Pass &lt;arg&gt; to the linker.</td>
</tr>
<tr>
<td>4.49 -l on page 4-106</td>
<td>Search for the library named &lt;library&gt; when linking.</td>
</tr>
<tr>
<td>4.57 -shared on page 4-114</td>
<td>Create a shared object that can be linked against.</td>
</tr>
<tr>
<td>4.58 -static on page 4-115</td>
<td>Link against static libraries.</td>
</tr>
</tbody>
</table>
4.2 -###

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

**Syntax**

```
armclang -###
```
4.3 -armpl=

Enable Arm Performance Libraries (ArmPL) (disabled by default).

Instructs the compiler to load the optimum version of Arm Performance Libraries for your target architecture and implementation. 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.

ArmPL provides libraries suitable for a range of supported CPUs. If you intend to use -armpl, you must also specify the required architecture using the -mcpu flag.

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)
Use the ‘Generic’ SVE library from Arm Performance Libraries.

Note:
• To enable SVE compilation and library usage on SVE-enabled targets, use -armpl -mcpu=native.
• To enable SVE(2) compilation and library usage on a target without native support for these features, use -armpl=sve -march=armv8-a+<Feature>, where <Feature> is one of sve, sve2, sve2-bitperm, sve2-aes, sve2-sha3, or sve2-sm4.
4.4 -c

Only run preprocess, compile, and assemble steps.

Syntax

```
armclang -c
```
4.5 **-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 https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-allinea-studio/installation/configure.

**Syntax**

```
armclang --config <arg>
```
4.6 -D

Define <macro> to <value> (or 1 if <value> omitted).

Syntax

armclang -Dmacro=<value>
4.7 -E

Only run the preprocessor.

Syntax

armclang -E
4.8 \texttt{-fassociative-math}

Allow \texttt{-fassociative-math} or prevent \texttt{-fno-associative-math} the re-association of operands in a series of floating-point operations. Default is \texttt{-fno-associative-math}.

For example, $(a \ast b) + (a \ast c) \Rightarrow a \ast (b + c)$. Note: Using \texttt{-fassociative-math} violates the ISO C and C++ language standard.

**Default**

Default is \texttt{-fno-associative-math}.

**Syntax**

\begin{verbatim}
armclang -fassociative-math, -fno-associative-math
\end{verbatim}
4.9 -fcolor-diagnostics

Use colors in diagnostics.

Syntax

armclang -fcolor-diagnostics, -fno-color-diagnostics
4.10 -fcxx-exceptions

Enable C++ exceptions.

Syntax

```
armclang -fcxx-exceptions
```
4.11 -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.
4.12 -fexceptions

Enable support for exception handling.

**Syntax**

```
armclang -fexceptions
```
4.13 -ffast-math

Allow aggressive, lossy floating-point optimizations (disabled by default).

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

-ffp-contract=fast, -fno-math-errno, -fno-signed-zeros, -fno-trapping-math, and -freciprocal-math

Syntax

armclang -ffast-math
4.14 -ffinite-math-only

Enable optimizations that ignore the possibility of NaN and +/-Inf (disabled by default).

**Syntax**

```
armclang -ffinite-math-only
```
4.15  -ffp-contract=

Controls when the compiler is permitted to form fused floating-point operations (for example, FMAs). These instructions typically operate to a higher degree of accuracy than individual multiply and add instructions.

Syntax

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

Arguments

- **fast**
  Always (default for Fortran workloads). Note: They are not strictly allowed according to the C/C++ standard because they can lead to deviations from the expected results.

- **on**
  Only in the presence of the FP_CONTRACT pragma (default for C/C++ workloads).

- **off**
  Never.
4.16  -fhonor-infinities

Do not allow optimizations that assume the arguments and results of floating point arithmetic are not +/- Inf.

**Syntax**

```
armclang -fhonor-infinities, -fno-honor-infinities
```
4.17 -fhonor-nans

Do not allow optimizations that assume the arguments and results of floating point arithmetic are not NaN.

Syntax

armclang -fhonor-nans, -fno-honor-nans
4.18 -finline

Enable/disable inlining (enabled by default).

Syntax

armclang -finline, -fno-inline
4.19 -finline-functions

Inline suitable functions.

**Syntax**

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

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

Syntax

```
armclang -finline-hint-functions
```
4.21 -fiterative-reciprocal

Allow optimizations that replace division by reciprocal estimation and refinement.

**Syntax**

armclang -fiterative-reciprocal, -fno-iterative-reciprocal
4.22 -flto

Enable (-flto) or disable (-fno-lto) 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
4.23 \texttt{-fmath-errno}

Require math functions to indicate errors.

Use \texttt{-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 \texttt{-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.

\textbf{Syntax}

\begin{verbatim}
armclang -fmath-errno, -fno-math-errno
\end{verbatim}
4.24 **-fno-crash-diagnostics**

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

**Syntax**

```
armclang -fno-crash-diagnostics
```
4.25 -fopenmp

Enable OpenMP and link in the OpenMP library, libomp.

Syntax

armclang -fopenmp
4.26 -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
4.27 -freciprocal-math

Allow division operations to be reassociated

Syntax

armclang -freciprocal-math
4.28 -fsave-optimization-record

Enable (-fsave-optimization-record) or disable (-fno-save-optimization-record) the generation of a YAML optimization record file. Default is -fno-save-optimization-record.

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
4.29 -fsigned-char

Set the type of ‘char’ to be signed. Disabled by default.

Syntax

armclang -fsigned-char, -fno-signed-char
4.30 -fsigned-zeros

Do not allow optimizations that ignore the sign of floating point zeros. Enabled by default.

Syntax

armclang -fsigned-zeros, -fno-signed-zeros
4.31  -fsimdpmath

Enables the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h

Syntax

armclang -fsimdpmath, -fno-simdpmath
4.32  -fstrict-aliasing

Tells the compiler to adhere to the aliasing rules defined in the source language (enabled by default when using ‘-Ofast’).

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

Syntax

```
armclang -fstrict-aliasing
```
4.33 -fsyntax-only

Show syntax errors but do not perform any compilation.

Syntax

armclang -fsyntax-only
4.34  -ftrapping-math

Tell the compiler to assume (-ftrapping-math), or not to assume (-fno-trapping-math), that floating point operations can trap.

Possible traps include:

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

Syntax

armclang -ftrapping-math, -fno-trapping-math
4.35 -funsafe-math-optimizations

Enable a subset of the optimizations allowed by -ffast-math.

Using --funsafe-math-optimizations is equivalent to specifying the following flags individually:
-fassociative-math, -freciprocal-math, -fno-signed-zeros, and -fno-trapping-math

Syntax

armclang -funsafe-math-optimizations, -fno-unsafe-math-optimizations
4.36 -fvectorize

Enable/disable loop vectorization (enabled by default).

Syntax

armclang -fvectorize, -fno-vectorize
4.37 -g

Generate source-level debug information.

**Syntax**

```
armclang -g
```
4.38 -g0

Disable generation of source-level debug information (default).

Syntax

armclang -g0
4.39  -gcc-toolchain=

Use the gcc toolchain at the given directory.

Syntax

armclang --gcc-toolchain=<arg>
4.40  \texttt{-gline-tables-only}

Emit debug line number tables only.

\textbf{Syntax}

\begin{verbatim}
armclang -gline-tables-only
\end{verbatim}
4.41 -help

Display available options.

Syntax

armclang -help, --help
4.42 -help-hidden

Display hidden options. Only use these options if advised to do so by your Arm representative.

Syntax

armclang --help-hidden
4.43 -I

Add directory to include 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).

Syntax

armclang -I<dir>
4.44 -include

Include file before parsing.

Syntax

armclang -include<file>, --include<file>
4.45 -iquote

Add directory to the include search path.

Directories specified with the -iquote option only apply to the quote form of the include directive, such as #include “file”.

Syntax

armclang -iquote<directory>
4.46 -isysroot

For header files, set the system root directory (usually /).

Syntax

armclang -isysroot<dir>
4.47 -isystem

Override the system include directory.

Directories specified with the -isystem option apply to both the quote form of the include directive, such as #include “file”, and the system header form, such as #include <file>.

Directories specified with this option will be searched after directories specified by the -iquote or -I options, but before the standard system include directory.

Syntax

armclang -isystem<directory>
4.48 -isystem-after

Add directory to end of the SYSTEM include search path.

Syntax

armclang -isystem-after<directory>
4.49 -l

Search for the library named <library> when linking.

Syntax

```
armclang -l<library>
```
4.50 -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.

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.

Note
When enabling either the sve2 or sve features, to link to the SVE-enabled version of Arm Performance Libraries, you must also include the -armpl=sve option. For more information about the supported options for -armpl, see the -armpl description.

Syntax
armclang -march=<arg>
4.51 -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.

- **a64fx**
  
  Optimize for Fujitsu A64FX based computers.

- **generic**
  
  Generate portable output suitable for any Armv8-A based computer.
4.52 -O

Specifies the level of optimization to use when compiling source files.

**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 armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards. -Ofast implies -ffast-math.
4.53 -o

Write output to <file>.

**Syntax**

```bash
armclang -o<file>
```
4.54 -print-search-dirs

Print the paths that are used for finding libraries and programs.

Syntax

```
armclang -print-search-dirs, --print-search-dirs
```
4.55 -Qunused-arguments

Do not emit a warning for unused driver arguments.

Syntax

armclang -Qunused-arguments
4.56 -S

Only run preprocess and compilation steps.

Syntax

armclang -S
4.57 -shared

Create a shared object that can be linked against.

Syntax

armclang -shared, --shared
4.58 -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
4.59 **-std=**

Language standard to compile for.

The list of valid standards depends on the input language, but adding `-std=` to a build line will generate an error message listing valid choices.

**Syntax**

```
armclang -std=<arg>, --std=<arg>
```
4.60 -U

Undefine macro <macro>.

Syntax

armclang -U<macro>
4.61 -v

Show commands to run and use verbose output.

Syntax

armclang -v
4.62 -version

Show the version number and some other basic information about the compiler.

**Syntax**

```
armclang --version, --vsn
```
4.63 -W

Enable the specified warning.
Similarly, warnings can be disabled with -Wno-<warning>.

Syntax

```
armclang -W<warning>
```
4.64 -Wall

Enable all warnings.

**Syntax**

```
armclang -Wall
```
4.65 -Warm-extensions

Enable warnings about the use of non-standard language features supported by armclang or armflang

Syntax

armclang -Warm-extensions
4.66 -Wdeprecated

Enable warnings for deprecated constructs and define __DEPRECATED.

Syntax

armclang -Wdeprecated
4.67 -Wl,

Pass the comma separated arguments in <arg> to the linker.

Syntax

armclang -Wl,<arg>,<arg2>...
4.68 -Wp,

Pass the comma separated arguments in <arg> to the preprocessor

Syntax

armclang -Wp,<arg>,<arg2>...
4.69  -w

Suppress all warnings.

**Syntax**

```
armclang -w
```
4.70 -working-directory

Resolve file paths relative to the specified directory.

**Syntax**

```
armclang -working-directory<arg>
```
4.71 -Xassembler

Pass `<arg>` to the assembler.

**Syntax**

```
armclang -Xassembler <arg>
```
4.72 -Xlinker

Pass `<arg>` to the linker.

**Syntax**

```
armclang -Xlinker <arg>
```
4.73 -Xpreprocessor

Pass <arg> to the preprocessor.

Syntax

armclang -Xpreprocessor <arg>
Chapter 5
Standards support

The support status of Arm C/C++ Compiler with the C/C++ language and OpenMP standards.
It contains the following sections:
• 5.1 Supported C/C++ standards in Arm® C/C++ Compiler on page 5-132.
• 5.2 OpenMP 4.0 on page 5-133.
• 5.3 OpenMP 4.5 on page 5-134.
5.1 Supported C/C++ standards in Arm® C/C++ Compiler

This topic describes the supported C/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 \texttt{4.59-std=} on page 4-116 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 \texttt{-std=gnu11}.

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).

\begin{itemize}
  \item Exported templates, as included in the C++98 standard, were removed in the C++11 standard, and are not supported.
\end{itemize}

To select which language standard Arm C/C++ Compiler should use, use the \texttt{4.59-std=} on page 4-116 compiler option with the argument:

• c++98 or c++03 for the ‘ISO C++ 1998 with amendments’ standard
• gnu++98 or gnu++03 for the ‘ISO C++ 1998 with amendments and GNU extensions’ standard
• c++11 for the ‘ISO C++ 2011 with amendments’ standard
• gnu++11 for 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 \texttt{-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 20.2 is based on Clang 9.

Related references

5.2 OpenMP 4.0 on page 5-133
5.3 OpenMP 4.5 on page 5-134
5.2 OpenMP 4.0

Describes which OpenMP 4.0 features are supported by Arm C/C++ Compiler.

<table>
<thead>
<tr>
<th>OpenMP 4.0 Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Array Sections</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread affinity policies</td>
<td>Yes</td>
</tr>
<tr>
<td>“simd” construct</td>
<td>Yes</td>
</tr>
<tr>
<td>“declare simd” construct</td>
<td>No</td>
</tr>
<tr>
<td>Device constructs</td>
<td>No</td>
</tr>
<tr>
<td>Task dependencies</td>
<td>Yes</td>
</tr>
<tr>
<td>“taskgroup” construct</td>
<td>Yes</td>
</tr>
<tr>
<td>User defined reductions</td>
<td>Yes</td>
</tr>
<tr>
<td>Atomic capture swap</td>
<td>Yes</td>
</tr>
<tr>
<td>Atomic seq_cst</td>
<td>Yes</td>
</tr>
<tr>
<td>Cancellation</td>
<td>Yes</td>
</tr>
<tr>
<td>OMP_DISPLAY_ENV</td>
<td>Yes</td>
</tr>
</tbody>
</table>
5.3 OpenMP 4.5

Describes which OpenMP 4.5 features are supported by Arm C/C++ Compiler.

<table>
<thead>
<tr>
<th>Open MP 4.5 Feature</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>doacross loop nests with ordered</td>
<td>Yes</td>
</tr>
<tr>
<td>“linear” clause on loop construct</td>
<td>Yes</td>
</tr>
<tr>
<td>“simdlen” clause on simd construct</td>
<td>Yes</td>
</tr>
<tr>
<td>Task priorities</td>
<td>Yes</td>
</tr>
<tr>
<td>“taskloop” construct</td>
<td>Yes</td>
</tr>
<tr>
<td>Extensions to device support</td>
<td>No</td>
</tr>
<tr>
<td>“if” clause for combined constructs</td>
<td>Yes</td>
</tr>
<tr>
<td>“hint” clause for critical construct</td>
<td>Yes</td>
</tr>
<tr>
<td>“source” and “sink” dependence types</td>
<td>Yes</td>
</tr>
<tr>
<td>C++ Reference types in data sharing attribute clauses</td>
<td>Yes</td>
</tr>
<tr>
<td>Reductions on C/C++ array sections</td>
<td>Yes</td>
</tr>
<tr>
<td>“ref”, “val”, “uval” modifiers for linear clause.</td>
<td>Yes</td>
</tr>
<tr>
<td>Thread affinity query functions</td>
<td>Yes</td>
</tr>
<tr>
<td>Hints for lock API</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Chapter 6
Troubleshoot

Describes how to diagnose problems when compiling applications using Arm Fortran Compiler.

It contains the following sections:

• 6.1 Application segfaults at -Ofast optimization level on page 6-136.
• 6.2 Compiling with the -fpic option fails when using GCC compilers on page 6-137.
• 6.3 Error messages when installing Arm® Compiler for Linux on page 6-138.
6.1 Application segfaults at -Ofast optimization level

A Fortran program runs correctly when the binary is built with armflang at -O3 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 at the compilation with armflang.

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. This 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.

If you continue to experience problems, Contact Arm Support.
6.2 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.
6.3 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.