Arm® C/C++ Compiler

Version 20.3

Developer and Reference Guide

arm

101458_2030_01_en

Arm® C/C++ Compiler

Developer and Reference Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
1900-00	02 November 2018	Non-Confidential	Document release for Arm C/C++ Compiler version 19.0
1910-00	08 March 2019	Non-Confidential	Update for Arm C/C++ Compiler version 19.1
1920-00	07 June 2019	Non-Confidential	Update for Arm C/C++ Compiler version 19.2
1930-00	30 August 2019	Non-Confidential	Update for Arm C/C++ Compiler version 19.3
2000-00	29 November 2019	Non-Confidential	Update for Arm C/C++ Compiler version 20.0
2010-00	23 April 2020	Non-Confidential	Update for Arm C/C++ Compiler version 20.1
2010-01	23 April 2020	Non-Confidential	Documentation update 1 for Arm C/C++ Compiler version 20.1
2020-00	25 June 2020	Non-Confidential	Update for Arm C/C++ Compiler version 20.2
2030-00	04 September 2020	Non-Confidential	Update for Arm C/C++ Compiler version 20.3
2030-01	16 October 2020	Non-Confidential	Documentation update 1 for Arm C/C++ Compiler version 20.3

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

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

developer.arm.com

Contents
Arm® C/C++ Compiler Developer and Reference Guide

List of Tables
Arm® C/C++ Compiler Developer and Reference Guide

Preface

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

Provides information to help you use the Arm® C/C++ Compiler component of Arm Compiler for Linux. Arm C/C++ Compiler is an auto-vectorizing, Linux-space C and C++ compiler, tailored for Server and High Performance Computing (HPC) workloads. Arm C/C++ Compiler supports Standard C and C++ source code and is tuned for Armv8-A based processors.

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

Chapter 4 Compiler options

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

Chapter 5 Standards support

This chapter describes the support status of Arm C/C++ Compiler with the C/C++ language and OpenMP standards.

Chapter 6 Troubleshoot

This chapter describes how to diagnose problems when compiling applications using Arm C/C++ 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.

`mono`space

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`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_2030_01_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® Documentation*.
- *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

Arm C/C++ Compiler is a Linux user space C/C++ compiler for server and High Performance Computing (HPC) Arm-based platforms. Arm C/C++ Compiler is built on the open-source Clang front-end and the LLVM 9-based optimization and code generation back-end.

Arm C/C++ Compiler supports modern C/C++ (see *Supported C/C++ standards in Arm® C/C++ Compiler* on page 5-139), *OpenMP 4.0* on page 5-140, and *OpenMP 4.5* on page 5-141 standards, has a built-in autovectorizer, and is tuned for the 64-bit Armv8-A architecture. Arm C/C++ Compiler also supports compiling for Scalable Vector Extension- (SVE-) and SVE2-enabled target platforms.

- A Linux user space C/C++ compiler for server and High-Performance Computing (HPC) Arm-based platforms.
- Built on the open-source Clang front-end and the LLVM-based optimization and code generation back-end.
- Tuned for the 64-bit Armv8-A architecture, and includes a built-in autovectorizer.
- Packaged with Arm Fortran Compiler and Arm Performance Libraries in a single package called Arm Compiler for Linux.

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

To use Arm Compiler for Linux, you must have a valid license for Arm Allinea Studio. Arm Allinea Studio is an end-to-end commercial suite of tools for developing Linux applications to run on Armv8-A-based targets. For more information about Arm Allinea Studio and how to license the tools, see the *Arm Allinea Studio web page*.

Resources

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

- **Arm 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 *Learn about the Scalable Vector Extension (SVE)*.
- For a list of SVE and SVE2 instructions, see the *Arm A64 Instruction Set Architecture*.
- *White Paper: A sneak peek into SVE and VLA programming*. An overview of SVE with information on the new registers, the new instructions, and the Vector Length Agnostic (VLA) programming technique, with some examples.
- *White Paper: Arm Scalable Vector Extension and application to Machine Learning*. In this white paper, code examples are presented that show how to vectorize some of the core computational kernels that are part of 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 6-142
  - *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, but you know the installation location, use `module use` to update your `MODULEPATH` environment variable to include that location:

      ```
      module use <path/to/installation>/modulefiles/
      ```

      replacing `<path/to/installation>` with the path to your installation of Arm Compiler for Linux. The default installation location is `/opt/arm/`.

      `module use` sets your `MODULEPATH` environment variable to include the installation directory:

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

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

      For example:

      ```
      module load Generic-AArch64/SUSE/12/arm-linux-compiler/20.3
      ```

   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-20.3_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 the input source filename, `<source>.c|cpp`, and use `-o` to specify the output binary file, `<binary>`:

   ```
   {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`:

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

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

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

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

4. Run the generated binary `hello`:

   ```
   ./hello
   ```

Next Steps

For more information about compiling and linking as separate steps, and how optimization levels effect auto-vectorization, see *Compile and Link* on page 2-17.

1.3 Get support

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

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

or

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

A description of each supported command-line option is available in *Compiler options* on page 4-63.

If you encounter a problem when developing your application and compiling with the Arm Compiler for Linux, see the *Troubleshoot* on page 6-142 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

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

Compile and link

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

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

A binary with the filename `source` is output.

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

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

You can specify multiple source files on a single line. Each source file is compiled individually and then linked into a single executable binary. For example, to compile the source files `source1.c` and `source2.c`, use:

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

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

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

Increase the optimization level

To control the optimization level, specify the `-O<level>` option on your compile line, and replace `<level>` with one of `0`, `1`, `2`, `3`, or `fast`. `-O0` option is the lowest, and the default, optimization level. `-Ofast` is the highest optimization level. Arm C/C++ Compiler performs auto-vectorization at levels `-O2`, `03`, and `-Ofast`.

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

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

Compile and optimize using CPU auto-detection

If you tell Arm C/C++ Compiler what target CPU your application will run on, the compiler can make target-specific optimization decisions. Target-specific optimization decisions help ensure your application runs as efficiently as possible. To tell the compiler to make target-specific compilation decisions, use the `-mcpu=<target>` option and replace `<target>` with your target processor (from a supported list of targets). To see what processors are supported by the `-mcpu` option, see *4.49 -mcpu= on page 4-118*.

In addition, the `-mcpu` option also supports a `native` argument. `-mcpu=native` enables Arm C/C++ Compiler to auto-detect the architecture and processor type of the CPU that you are running the compiler on.

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

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

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

Note

- The optimizations that are performed from setting the `-mcpu` option (also known as hardware, or CPU, tuning) are independent of the optimizations that are performed from setting the `-O<level>` option.
- If you run the compiler on one target, but will run the application you are compiling on a different target, do not use `-mcpu=native`. Instead, use `-mcpu=<target>` where `<target>` is the target processor that you will run the application on.

Common compiler options

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

Note

For more information about all the supported compiler options, run `man armclang`, `armclang --help`, or see *Compiler options* on page 4-63.

`-S`

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

`-c`

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

`-o <file>`

Specifies the name of the output file.

`-march=name[+[no]feature]`

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

Note

If you know what your target CPU is, Arm recommends using the `-mcpu` option instead of `-march`. For a complete list of supported targets, see *4.48 -march=* on page 4-117.

`-mcpu=native`

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

`-mcpu` supports the following Armv8-A-based Systems-on-Chip (SoCs): 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 target.

For more information, see *4.49 -mcpu=* on page 4-118.

`-O<level>`

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

——— **Warning** ———

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

———————————

For more information, see *4.50 -O* on page 4-119.

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

Passes the location of a configuration file to the compile command. Use a configuration file to specify a set of compile options to be run at compile time. The configuration file can be passed at compile time, or an environment variable can be set for it to be used for every invocation of the compiler. For more information about creating and using a configuration file, see *Configure Arm Compiler for Linux*.

`--help`

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

`--version`

Displays version information.

For a detailed description of all the supported compiler options, see *Compiler options* on page 4-63.

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

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

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

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

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 4-63
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

- Your target must be SVE- or SVE2-enabled hardware, or you must download, install, and load the correct environment module for Arm Instruction Emulator. For more information about installing and setting up your environment for Arm Instruction Emulator, see *Install Arm Instruction Emulator*.
- 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:

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

     For SVE2:

     ```
     armclang -O<level> -march=armv8-a+sve2 -o <binary> <source>.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:

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

     For SVE2:

     ```
     armclang -O<level> -march=armv8-a+sve2 -armpl=sve -o <binary> <source>.c
     ```

     For more information about the supported options for `-armpl`, see the `-armpl` description in *Arm C/C++ Compiler Options by Function* on page 4-66.

Note

For all of the preceding command lines, to get the best vectorizations for SVE, set `-O<level>` to be `-O2` or higher.

There are several SVE2 Cryptographic Extensions available: `sve2-aes`, `sve2-bitperm`, `sve2-sha3`, and `sve2-sm4`. Each extension is enabled using the `march` compiler option. For a full list of supported `-march` options, see *Arm C/C++ Compiler Options by Function* on page 4-66.

Note

`sve2` also enables `sve`.

2. Run the executable:
   - On SVE(2)-enabled hardware:

     ```
     ./<binary>
     ```
   - Using Arm Instruction Emulator to emulate the SVE(2) instructions:

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

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

     Note

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

Example 2-1 Example

This example compiles some application source, `source`, into assembly with auto-vectorization enabled, and runs it on a SVE-enabled hardware.

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

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

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

int main()
{
    subtract_arrays(a, b, c);
}
```

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

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

   The output assembly code is saved as `source.s`.
2. Inspect the output assembly code.

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

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

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

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

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

The output executable binary code is saved as `binary`.

4. Run the executable on SVE(2)-enabled hardware:

```
./binary
```

——————— Note ———————

Alternatively, to run on any Armv8-A-enabled hardware, with a vector length of 512 bits, use Arm Instruction Emulator:

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

———————————————

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:

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

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

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

Example 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.

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

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

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

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

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

This code shows that the compiler has not performed any vectorization, because we specified the `-O1` (low optimization) option. Array elements are iterated over one at a time. Each array element is a 32-bit or 4-byte integer, so the loop increments by 4 each time. The loop stops when it reaches the end of the array (1024 iterations * 4 bytes later).

2. Recompile the application with auto-vectorization enabled (`-O2`):

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

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

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

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

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

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

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

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

`r`

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

`w`

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

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

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

`%[name]`

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

`%w[name]`

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

`%s[name]`

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

`%d[name]`

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

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

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

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

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

Inputs

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

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

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

`r`

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

`w`

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

`[output-name]`

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

`i`

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

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

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

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

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

Side effects

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

`"memory"`

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

`"cc"`

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

`"xN"`

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

`"vN"`

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

"zN"

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

"pN"

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

Use of volatile

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

- The first sets the floating-point rounding mode.
- The second executes on the assumption that the rounding mode set by the first statement is in effect.
- The third statement restores the original floating-point rounding mode.

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

For example:

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

Note

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

Labels

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

```
1:
```

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

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

Example

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

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

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

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

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

Note

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

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

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

Chapter 3
Optimize

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

It contains the following sections:

3.1 Optimizing C/C++ code with Arm SIMD (Neon™)

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

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

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

For example, consider the following SIMD instruction:

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

The instruction specifies that an addition (`ADD`) operation is performed on two 64-bit data lanes (2D). `D` specifies the width of the data lane (doubleword, or 64 bits) and `2` specifies that two lanes are used (that is the full 128-bit register). Each lane in `V1` is added to the corresponding lane in `V2` and the result is stored in `V0`. Each lane is added separately. There are no carries between the lanes.

Coding with SIMD

To take advantage of SIMD instructions in your code:

- Let the compiler auto-vectorize your code.

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

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

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

  For more information about auto-vectorization best practice, see *Coding best practice for auto-vectorization* on page 3-34.
- Use intrinsics directly in your C code.

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

  Note

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

Related information

Overview of Neon technology
Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile
Coding for Neon
Arm Neon Programmer’s Guide
Arm C Language Extensions

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

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

To optimize your code using SVE, you can:

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

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

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

  At the lowest optimization level, `-O0`, auto-vectorization is never performed, even if you specify `-fvectorize`. See *Arm C/C++ Compiler Options by Function* on page 4-66 for more information on setting these options.

  For more information about auto-vectorization best practice, see *Coding best practice for auto-vectorization* on page 3-34.
- Use SVE intrinsics.

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

  The SVE instrinsics are defined in the *Arm C Language Extensions for SVE specification*.
- Write SVE assembly code.

  For more information, see *Writing inline SVE assembly* on page 2-26.

Related information

Porting and Optimizing HPC Applications for Arm SVE guide
Scalable Vector Extension (SVE, and SVE2) information
Learn about the Scalable Vector Extension (SVE)
Arm A64 Instruction Set Architecture
White Paper: A sneak peek into SVE and VLA programming
White Paper: Arm Scalable Vector Extension and application to Machine Learning
Arm C Language Extensions (ACLE) for SVE
DWARF for the ARM 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

3.3 Coding best practice for auto-vectorization

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

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

Use restrict

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

Note

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

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

Use pragmas

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

For more information, see *Control auto-vectorization with pragmas* on page 3-35.

Use < to construct loops

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

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

Use the -ffast-math option

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

Warning

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

For more information, see *4.11 -ffast-math* on page 4-80.

3.4 Control auto-vectorization with pragmas

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

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

Note

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

Enable auto-vectorization with pragmas

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

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

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

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

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

Note

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

Warning

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

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

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

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

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

```
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:

```
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:

```
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

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

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

Unrolling

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

Take the following example:

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

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

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

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

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

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

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

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

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

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

Interleaving

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

Note

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

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

- To interleave to the internal limit, insert the following pragma before your loop:

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

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

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

Note

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

3.5 Vector routines support

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

This section contains the following subsections:

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

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

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

Examples

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

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

C example with loop invoking `sin`:

```
/* 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 */
  }
}
```

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

3.5.2 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:

```
// declarations or definitions visible at compile time in myvecroutines.h
#include <arm_neon.h>

int32x2_t myneon_foo(float64x2_t);

#pragma omp declare variant(myneon_foo) \
        match(construct = {simd(simdlen(2), notinbranch)}, \
              device = {isa("simd")})
int foo(double);

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

To compile the code, invoke `armclang` with either the `-fopenmp` or the `-fopenmp-simd` options (automatic loop vectorization is used at the `-O2`, `O3`, and `-Ofast` optimization levels):

```
armclang -fopenmp -O2 -c source.c
```

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

The following example shows the basic functionality for SVE vectorization:

```
// declarations or definitions visible at compile time in myvecroutines.h
#include <arm_sve.h>

svint32_t mysve_foo(svfloat64_t, svbool_t);

#pragma omp declare variant(mysve_foo) \
        match(construct = {simd(notinbranch)}, \
              device = {isa("sve")}, \
              implementation = {extension("scalable")})
int foo(double);

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

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

```
armclang -march=armv8-a+sve -fopenmp -O2 -c source.c
```

You must link the output object file against an object file or library that provides the symbol `mysve_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(f) x N` is either 8 or 16.

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

  ———— **Note** ————

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

  ————————————

- To allow scalable vectorization when targeting SVE, you must omit the `simdlen` clause, and you must specify the implementation trait extension `extension("scalable")`.
- The supported scalar function signature in C and C++ are in the forms:

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

  where `Ty#n` are:

  1. Any of the integral type values of size 1, 2, 4, or 8 (in bytes), signed and unsigned.
  2. Floating-point type values of half, single or double-precision.
  3. Pointers to any of the previous types.

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

Mapping rules

Common mapping rules

1. Each parameter and the return value of the scalar function, maps to a correspondent parameter and return value in the vector signature, in the same order.
2. A parameter that is marked with `linear` is left unchanged in the vector signature.
3. The `void` return type is left unchanged in the vector signature.

Mapping rules for Advanced SIMD

1. Each parameter type `Ty#n` maps to the correspondent Neon ACLE type `<Ty#n>x<N>_t`, where `N` is the value that is specified in the `simdlen(N)` clause. Values of `N` that do not correspond to NEON ACLE types are unsupported.
2. If you specify `inbranch`, an extra `mask` parameter is added as the last parameter of the vector signature. The type of the parameter is the NEON ACLE type `uint<BITS>x<N>_t`, where:
   a. `N` is the value that is specified in the `simdlen(N)` clause.
   b. `BITS` is the size (in bits) of the *Narrowest Data Size (NDS)* associated to the scalar function, as defined in the VFABI.
   c. To select active or inactive lanes, set all bits to 1 (active) or 0 (inactive) in the corresponding `uint<BITS>_t` integer in the mask vector.

Mapping rules for SVE

1. Each parameter type `Ty#n` is mapped to the correspondent SVE ACLE type `sv<Ty#n>_t`.
2. An extra *mask* parameter of type `svbool_t` is always added to the signature of the vector function, whether `inbranch` or `notinbranch` is used. Active and inactive lanes of the mask are set as described in the section **SVE Masking** of the VFABI:

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

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

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 required for the loop auto-vectorization process.
- `-fopenmp` to enable the parsing of the OpenMP directives.

Note

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

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

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

```
#include <arm_sve.h>

void CustomSinCos(svfloat64_t, double *, double *);

#pragma omp declare variant(CustomSinCos) \
        match(construct = {simd(notinbranch, linear(sinp), linear(cosp))}, \
              device = {isa("sve")}, \
              implementation = {extension("scalable")})
double sincos(double in, double *sinp, double *cosp);

void f(double *in, double *sin, double *cos, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    sincos(in[i], &sin[i], &cos[i]);
}
```

Examples: Advanced SIMD

Simple:

```
// filename: source1.c
#include <arm_neon.h>

__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo(float64x2_t a);

#pragma omp declare variant(user_vector_foo) \
        match(construct = {simd(simdlen(2), notinbranch)}, \
              device = {isa("simd")})
double foo(double);

void do_something(double * restrict a, double * b, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    a[i] = foo(b[i]);
}
```

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

```
armclang -fopenmp -O2 -c -S -o source1.c -mllvm -force-vector-interleave=1
```

```
//...
.LBB0_4:                                // =>This Inner Loop Header: Depth=1
    ldr     q0, [x25], #16
    bl      user_vector_foo
    subs    x23, x23, #2            // =2
    str     q0, [x24], #16
    b.ne    .LBB0_4
```

With `linear`:

```
// filename: source2.c
#include <arm_neon.h>

__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo_linear(float64x2_t, float *);

 #pragma omp declare variant(user_vector_foo_linear) \
        match(construct = {simd(simdlen(2), notinbranch, linear(b))}, \
              device = {isa("simd")})
double foo_linear(double a, float* b);

void do_something_linear(double * restrict a, double * b, float * x, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    a[i] = foo_linear(b[i], &x[i]);
}
```

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

```
armclang -fopenmp -O2 -c -S -o source2.c -mllvm -force-vector-interleave=1
```

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

Examples: SVE

Simple:

```
// 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 the code with:

```
armclang example03.c -march=armv8-a+sve -O2 -o -S -fopenmp
```

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

```
    whilelo p4.d, x21, x22
    b.mi    .LBB0_2
```

With `linear`:

```
// filename: example04.c
#include <arm_sve.h>

svfloat64_t user_vector_foo_linear_sve(svfloat64_t, float *, svbool_t);

#pragma omp declare variant(user_vector_foo_linear_sve) \
        match(construct = {simd(notinbranch, linear(b))}, \
              device = {isa("sve")}, \
              implementation = {extension("scalable")})
double foo_linear(double a, float* b);

void do_something_linear(double * restrict a, double * b, float * x, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    a[i] = foo_linear(b[i], &x[i]);
}
```

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

```
armclang 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, x22, x23
    b.mi    .LBB0_2
```

Related concepts

3.5.1 How to vectorize math routines in Arm® C/C++ Compiler on page 3-38

3.5.2 How to declare custom vector routines in Arm® C/C++ Compiler on page 3-39

3.6 Link Time Optimization (LTO)

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

This section contains the following subsections:

3.6.1 What is Link Time Optimization (LTO)

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

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

- Without LTO:
  1. Source files are translated into separate ELF object files (`.o`) and passed to the linker.
  2. The linker processes the separate ELF object files, together with library code, to create the ELF executable.
- With LTO:
  1. Source files are translated into a bitcode object files (`.o`), and passed to the linker. LLVM Bitcode is an intermediate form of code that is understood by the optimizer.
  2. To extract the module dependency information, the linker processes the bitcode and object files together and passes them to the LLVM optimizer utility, libLTO.
  3. The LLVM optimizer utility, libLTO, uses the module dependency information to filter out unused modules, and create a single highly optimized ELF object file. Additional optimizations are possible by knowing the module dependency information. The new ELF object file is returned to the linker.
  4. The linker links the new ELF object file with the remaining ELF object files and library code, to generate an ELF executable.

Limitations

LTO in Arm Compiler for Linux has some limitations:

- To compile static libraries, you must create a library archive file that libLTO can use at link time. `armllvm-ar`, as well as some open-source utility tools can create this archive file. For more information about `armllvm-ar`, see *armllvm-ar and reference* on page 3-47.
- Partial linking is not supported with LTO because partial linking only works with ELF objects, rather than bitcode files.
- If your library code calls a function that was defined in the source code, but is removed by libLTO, you might get linking errors.
- Bitcode objects are not guaranteed to be compatible across Arm Compiler for Linux versions. When linking with LTO, ensure that all your bitcode files are built using the same version of the compiler.
- You can not analyze LTO-optimized code using Arm Optimization Reports. Arm Optimization Reports analyzes object files that are generated by Arm Compiler for Linux before they are passed to the linker. Therefore, you can not use Arm Optimization Reports to investigate the vectorization decisions that LTO enables the linker to make.

3.6.2 Compile with Link Time Optimization (LTO)

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

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

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

  Note

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

  For more information about `armllvm-ar`, see *armllvm-ar and reference* on page 3-47. For more information about `armllvm-ranlib`, see *armllvm-ranlib reference* on page 3-48.

Procedure

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

     ```
     {armclang|armclang++} -O<level> -flto -o <binary> <sources>
     ```
   - For static library compilation:
     1. Compile, but do not link, your code with LTO:

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

        The result is one or more `.o` files, one per source file that was passed to `armclang|armclang++`.
     2. Create the archive file for your static library object files:

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

        For example:

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

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

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

        For a more detailed description of `armllvm-ar`, see *armllvm-ar and reference* on page 3-47. For a more detailed description of `armllvm-ar`, see *armllvm-ranlib reference* on page 3-48.
     3. Link your remaining object files together with your archive file:

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

Note

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

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

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

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

To run your binary, use:

```
./<binary>
```

Example 3-1 Example: Compare code compiled with and without LTO

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

1. Create the example source code files:
   a. Write and save the following code as a `main.c` source file:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

extern double foo(double);

int main(int argc, char *argv[]) {
   // Expected command line:
   if (argc != 3) {
      fprintf(stderr, "Incorrect arguments.");
      fprintf(stderr, " Usage: %s <filename> <size>", argv[0]);
      exit(1);
   }

   char *filename = argv[1];
   int numelts = atoi(argv[2]);
   FILE *file = fopen(filename, "rw");

   // Read in some binary data
   double *data = (double*)malloc(numelts * sizeof(double));
   fread(data, sizeof(double), numelts, file);

   // Do 'something' to the data
   for (int i = 0; i < numelts; i++)
      data[i] = foo(data[i]);

   // Overwrite the file.
   rewind(file);
   fwrite(data, sizeof(double), numelts, file);
   fclose(file);
   free(data);
```

```
   return EXIT_SUCCESS;
}
```

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

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

2. Use `armclang` to compile the code both without and with LTO enabled:
   a. To compile without LTO, into a binary called `binary-no-lto`, use:

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

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

```
armclang -O3 -flto -o binary-lto main.c foo.c
```

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

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

In the following pseudo code:

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

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

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

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

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

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

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

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

Related references

3.6.3 armllvm-ar and reference on page 3-47
3.6.4 armllvm-ranlib reference on page 3-48

3.6.3 armllvm-ar and reference

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

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

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

Note

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

Syntax

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

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

Note

`armllvm-ar` inherits the same syntax as `llvm-ar`.

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

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

Options, Operations, and Modifiers

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

Outputs

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

Related references

3.6.4 armllvm-ranlib reference on page 3-48

3.6.4 armllvm-ranlib reference

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

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

Note

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

Related concepts

3.6.1 What is Link Time Optimization (LTO) on page 3-44

Related tasks

3.6.2 Compile with Link Time Optimization (LTO) on page 3-45

Related references

3.6.3 armllvm-ar and reference on page 3-47

3.6.4 armllvm-ranlib reference on page 3-48

3.7 Profile Guided Optimization (PGO)

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

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

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

The profile information can be generated from either:

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

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

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

This section contains the following subsections:

- *3.7.1 How to compile with Profile Guided Optimization (PGO)* on page 3-50.
- *3.7.2 llvm-profdata reference* on page 3-52.

3.7.1 How to compile with Profile Guided Optimization (PGO)

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

Note

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

Prerequisites

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

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

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

Note

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

Procedure

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

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

Note

- For good optimization, use `-O2` optimization level or higher.
- To ensure that the instrumented executable represents the real executable, compile your application code with the same compiler options.
- By default, if you do not specify a `<profdata_file>.profraw`, when you compile the application with `-fprofile-instr-generate`, the profile data is written to `default.profraw`.

  To change this behavior, either specify `<profdata_file>.profraw` on the compile line, or set the `LLVM_PROFILE_FILE="<profdata_file>.profraw"` when you run your application (see next step). Both `-fprofile-instr-generate` and `LLVM_PROFILE_FILE` can use the following modifiers to uniquely set profile data filename:
  — `%p` to state the process ID
  — `%h` to state the hostname
  — `%m` to state the unique profile name.

  For example, `LLVM_PROFILE_FILE="example-{%p|%h|%m}.profraw"`.

  If both `-fprofile-instr-generate` and `LLVM_PROFILE_FILE` are set, `LLVM_PROFILE_FILE` takes priority.

2. Run your application code with a typical workload. Either:
   - Run it with default behavior:

```
./<binary>
```

     The profile data is written to the profile data file specified in the previous step, or if no file was specified, to `default.profraw`.
   - Run the application and specify a new filename for the `.profraw` file using the `LLVM_PROFILE_FILE` environment variable:

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

     The profile data is written to `<profdata_file>.profraw`.
3. Combine and convert your `.profraw` files into a single processed `.profdata` file using the `llvm-profdata` tool `merge` command:
   - If you have a single `.profraw` file, use:

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

     Note

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

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

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

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

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

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

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

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

Example 3-2 Example: Compiling code with PGO

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

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

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

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

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

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

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

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

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

Related concepts

3.7.2 llvm-profdata reference on page 3-52

Related information

LLVM's documentation

LLVM Command Guide

3.7.2 llvm-profdata reference

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

Note

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

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

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

Table 3-1 Describes the commands for llvm-profdata

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

Global options that all of the commands accept include:

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

Related information

LLVM Command Guide

Related concepts

3.7.2 llvm-profdata reference on page 3-52

Related tasks

3.7.1 How to compile with Profile Guided Optimization (PGO) on page 3-50

3.8 Arm Optimization Report

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

Unrolling

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

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

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

Vectorization

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

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

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

Note

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

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

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

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

Loops vectorized using scalable vectors are annotated with `VS<F,I>`. For more information, see *arm-opt-report reference* on page 3-56.

Interleaving

Example question: What was the interleave count?

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

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

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

Note

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

Reference

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

This section contains the following subsections:

3.8.1 How to use Arm Optimization Report

This topic describes how to use Arm Optimization Report.

Prerequisites

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

Procedure

1. To generate a machine-readable `.opt.yaml` report, at compile time add `-fsave-optimization-record` to your command line.

   A `<filename>.opt.yaml` report is generated by Arm 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 3-3 Example

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

   ```
   void bar();
   void foo() { bar(); }

   void Test(int *res, int *c, int *d, int *p, int n) {
   int i;

   #pragma clang loop vectorize(assume_safety)
   for (i = 0; i < 1600; i++) {
       res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
   }

   for (i = 0; i < 16; i++) {
       res[i] = (p[i] == 0) ? res[i] : res[i] + d[i];
   }

   foo();

   foo(); bar(); foo();
   }
   ```
2. Compile the file, for example to a shared object `example.o`:

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

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

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

   Note

   This example compiles to a shared object, however, you could also compile to a static object or to a binary.
3. View the `example.opt.yaml` file using `arm-opt-report`:

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

Annotated source code is displayed in the terminal:

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

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

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

Related references

3.8.2 arm-opt-report reference on page 3-56

Related information

Arm Compiler for Linux and Arm Allinea Studio

Take a trial

Help and tutorials

3.8.2 arm-opt-report reference

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

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

Note

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

Possible annotations are:

Annotation	Description
`I`	A function was inlined.
`U<N>`	A loop was unrolled `<N>` times.

(continued)

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

Syntax

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

Options

Generic Options:

`--help`

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

`--help-list`

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

`--version`

Displays the version of this program.

llvm-opt-report options:

`--hide-detrimental-vectorization-info`

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

`--hide-inline-hints`

Hides suggestions to inline function calls which are preventing vectorization.

`--hide-lib-call-remark`

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

`--hide-vectorization-cost-info`

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

`--no-demangle`

Does not demangle function names.

`-o=<string>`

Specifies an output file to write the report to.

`-r=<string>`

Specifies the root for relative input paths.

`-s`

Omits vectorization factors and associated information.

`--strip-comments`

Removes comments for brevity

`--strip-comments=<arg>`

Removes comments for brevity. Arguments are:

- `none`: Do not strip comments.
- `c`: Strip C-style comments.
- `c++`: Strip C++-style comments.
- `fortran`: Strip Fortran-style comments.

Outputs

Annotated source code.

Related tasks

3.8.1 How to use Arm Optimization Report on page 3-55

Related tasks

3.8.1 How to use Arm Optimization Report on page 3-55

Related references

3.8.2 arm-opt-report reference on page 3-56

3.9 Optimization remarks

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

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

To enable optimization remarks, pass one or more of the following `Rpass` options to `armclang|armclang++` at compile time:

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

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

- `loop-vectorize`: Remarks about vectorized loops.
- `inline`: Remarks about inlining.
- `loop-unroll`: Remarks about unrolled loops.

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

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

Note

Use `.*` with caution; depending on the size of code, and the level of optimization, the compiler can print a lot of information.

The general syntax to compile with optimization remarks enabled (`-Rpass[-<option>]`) and redirect the information to an output file (`<remarks-file.txt>`), is:

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

Note

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

This section contains the following subsection:

- *3.9.1 Enable Optimization remarks* on page 3-59.

3.9.1 Enable Optimization remarks

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

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. Compile your code with optimization remarks. To enable optimization remarks, pass one or more of `-Rpass=<regex>`, `-Rpass-missed=<regex>`, or `Rpass-analysis=<regex>` on your compile line.

For example, to report all the information about what the compiler has optimized (`-Rpass`), and what the compiler has analyzed (`-Rpass-analysis`) when compiling an input file called `source.c`, use:

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

Result:

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

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

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

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

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

Related information

Arm C/C++ Compiler

Related tasks

3.9.1 Enable Optimization remarks on page 3-59

3.10 Prefetching with __builtin_prefetch

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

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

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

__builtin_prefetch syntax

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

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

where:

`addr` (required)

Represents the address of the memory.

`rw` (optional)

A compile-time constant which can take the values:

- `0` (default): prepare the prefetch for a read
- `1` : prepare the prefetch for a write to the memory

`locality` (optional)

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

- `0`: None, the data can be removed from the cache after the access.
- `1`: Low, L3 cache, leave the data in the L3 cache level after the access.
- `2`: Moderate, L2 cache, leave the data in L2 and L3 cache levels after the access.
- `3` (default): High, L1 cache, leave the data in the L1, L2, and L3 cache levels after the access.

Note

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

Note

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

Example

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

```
void streaming_load(void *foo) {         // Streaming load
   __builtin_prefetch(foo + 1024,        // Address can be offset
                      0,                 // Read
                      0                  // No locality - streaming access
                      );
}

void l3_load(void *foo) {
   __builtin_prefetch(foo, 0, 1);        // L3 load prefetch (locality)
}
```

```
void l2_load(void *foo) {
   __builtin_prefetch(foo, 0, 2);        // L2 load prefetch (locality)
}
void l1_load(void *foo) {
   __builtin_prefetch(foo, 0, 3);        // L1 load prefetch (locality)
}

void streaming_store(void *foo) {
   __builtin_prefetch(foo + 1024, 1, 0); // Streaming store
}

void l3_store(void *foo) {
   __builtin_prefetch(foo, 1, 1);        // L3 store prefetch (locality)
}

void l2_store(void *foo) {
   __builtin_prefetch(foo, 1, 2);        // L2 store prefetch (locality)
}

void l1_store(void *foo) {
   __builtin_prefetch(foo, 1, 3);        // L1 store prefetch (locality)
}
```

Which, when compiled using the `-c -march=armv8-a -O3` compiler options, generates the following assembly:

```
streaming_load:
      prfm    PLDL1STRM, [x0, 1024]      ; Streaming load
      ret
l3_load:
      prfm    PLDL3KEEP, [x0]            ; L3 load prefetch (locality)
      ret
l2_load:
      prfm    PLDL2KEEP, [x0]            ; L2 load prefetch (locality)
      ret
l1_load:
      prfm    PLDL1KEEP, [x0]            ; L1 load prefetch (locality)
      ret
streaming_store:
      prfm    PSTL1STRM, [x0, 1024]      ; Streaming store
      ret
l3_store:
      prfm    PSTL3KEEP, [x0]            ; L3 store prefetch (locality)
      ret
l2_store:
      prfm    PSTL2KEEP, [x0]            ; L2 store prefetch (locality)
      ret
l1_store:
      prfm    PSTL1KEEP, [x0]            ; L1 store prefetch (locality)
      ret
```

Chapter 4
Compiler options

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

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

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

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

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

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

```
fast  off  on
```

Note

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

It contains the following sections:

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.

Option	Description
4.7 -E on page 4-76	Stop after pre-processing. Output the pre-processed source.
4.54 -S on page 4-123	Stop after compiling the source and emit assembler files.
4.4 -c on page 4-73	Stop after compiling or assembling sources and do not link. This outputs object files.
4.22 -fopenmp on page 4-91	Enable (‘-fopenmp’) or disable (‘-fno-openmp’ [default]) OpenMP and link in the OpenMP library, libomp.
4.23 -fopenmp-simd on page 4-92	Enable processing of ‘simd’ and the ‘declare simd’ pragma, without enabling OpenMP or linking in the OpenMP library, libomp. Enabled by default.
4.30 -fsyntax-only on page 4-99	Show syntax errors but do not perform any compilation.

File options

Options that specify input or output files.

Option	Description
4.40 -I on page 4-109	Add directory to include search path and Fortran module search path.
4.5 -config on page 4-74	Passes the location of a configuration file to the compile command.
4.41 -idirafter on page 4-110	Add directory to include search path after system header file directories.
4.42 -include on page 4-111	Include file before parsing.
4.43 -iquote on page 4-112	Add directory to include search path. Directories specified with the ‘-iquote’ option apply only to the quote form of the include directive.
4.44 -isysroot on page 4-113	For header files, set the system root directory (usually /).
4.45 -isystem on page 4-114	Add a directory to the include search path, before system header file directories.
4.51 -o on page 4-120	Write output to <file>.
4.67 -working-directory on page 4-136	Resolve file paths relative to the specified directory.

Basic driver options

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

Option	Description
4.2 -### on page 4-70	Print (but do not run) the commands to run for this compilation.
4.36 -gcc-toolchain= on page 4-105	Use the gcc toolchain at the given directory.
4.38 -help on page 4-107	Display available options.
4.39 -help-hidden on page 4-108	Display hidden options. Only use these options if advised to do so by your Arm representative.
4.52 -print-search-dirs on page 4-121	Print the paths that are used for finding libraries and programs.

(continued)

Option	Description
4.59 -v on page 4-128	Show commands to run and use verbose output.
4.60 -version on page 4-129	Show the version number and some other basic information about the compiler.

Optimization options

Options that control what optimizations should be performed.

Option	Description
4.50 -O on page 4-119	Specifies the level of optimization to use when compiling source files.
4.3 -armpl= on page 4-71	Enable Arm Performance Libraries (ArmPL).
4.8 -fassociative-math on page 4-77	Allow (‘-fassociative-math’) or do not allow (‘-fno-associative-math’ [default]) the re-association of operands in a series of floating-point operations.
4.10 -fdenormal-fp-math= on page 4-79	Specify the denormal numbers the code is allowed to require.
4.11 -ffast-math on page 4-80	Enable (‘-ffast-math’) or disable (‘-fno-fast-math’ [default, except with ‘-Ofast’]) aggressive, lossy floating-point optimizations.
4.12 -ffinite-math-only on page 4-81	Enable (‘-ffinite-math-only’) or disable (‘-fno-finite-math-only’ [default, except with ‘-Ofast’]) optimizations that ignore the possibility of NaN and +/-Inf.
4.13 -ffp-contract= on page 4-82	Controls when the compiler is permitted to form fused floating-point operations (for example, Fused Multiply-Add (FMA) operations).
4.14 -fhonor-infinities on page 4-83	Allow (‘-fno-honor-infinites’) or do not allow (‘-fhonor-infinites’ [default, except with ‘-Ofast’]) optimizations that assume the arguments and results of floating point arithmetic are not +/-Inf.
4.15 -fhonor-nans on page 4-84	Allow (‘-fno-honor-nans’) or do not allow (‘-fhonor-nans’ [default, except with ‘-Ofast’]) optimizations that assume the arguments and results of floating point arithmetic are not NaN.
4.16 -finline-functions on page 4-85	Inline (‘-finline-functions’) or do not inline (‘-fno-inline-functions’) suitable functions.
4.17 -finline-hint-functions on page 4-86	Inline functions which are (explicitly or implicitly) marked inline.
4.18 -fiterative-reciprocal on page 4-87	Enable (‘-fiterative-reciprocal’) or disable (‘-fno-iterative-reciprocal’ [default, except with ‘-Ofast’]) optimizations that replace division by reciprocal estimation and refinement.
4.19 -flto on page 4-88	Enable (‘-flto’) or disable (‘-fno-lto’ [default]) Link Time Optimizations (LTO).
4.20 -fmath-errno on page 4-89	Require (‘-fmath-errno’) or do not require (‘-fno-math-errno’) math functions to indicate errors.
4.24 -freciprocal-math on page 4-93	Enable (‘-freciprocal-math’) or disable (‘-fno-reciprocal-math’ [default, except with ‘-Ofast’]) division operations to be reassociated
4.25 -fsave-optimization-record on page 4-94	Enable (‘-fsave-optimization-record’) or disable (‘-fno-save-optimization-record’ [default]) the generation of a YAML optimization record file.
4.27 -fsigned-zeros on page 4-96	Allow (‘-fno-signed-zeros’) or do not allow (‘-fsigned-zeros’ [default, except with ‘-Ofast’]) optimizations that ignore the sign of floating point zeros.
4.28 -fsimdmath on page 4-97	Enable (‘-fsimdmath’) or disable (‘-fno-simdmath’ [default]) the vectorized libm library to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h

(continued)

Option	Description
4.29 -fstrict-aliasing on page 4-98	Tells the compiler to adhere to the aliasing rules defined in the source language (enabled by default when using '-Ofast').
4.31 -ftrapping-math on page 4-100	Tell the compiler to assume ('-ftrapping-math'), or not to assume ('-fno-trapping-math'), that floating point operations can trap. For example, divide by zero.
4.32 -funsafe-math-optimizations on page 4-101	Enable ('-funsafe-math-optimizations') or disable ('-fno-unsafe-math-optimizations' [default, except with '-Ofast']) reassociation and reciprocal math optimizations.
4.33 -fvectorize on page 4-102	Enable/disable loop vectorization (enabled by default).
4.48 -march= on page 4-117	Specifies the base architecture and extensions available on the target.
4.49 -mcpu= on page 4-118	Select which CPU architecture to optimize for.

C/C++ Options

Options that affect the way C workloads are compiled.

Option	Description
4.26 -fsigned-char on page 4-95	Set the type of 'char' to be signed. Disabled by default.
4.57 -std= on page 4-126	Language standard to compile for.

Development options

Options that facilitate code development.

Option	Description
4.9 -fcolor-diagnostics on page 4-78	Enable ('-fcolor-diagnostics') or disable ('-fno-color-diagnostics' [default]) using colors in diagnostics.
4.34 -g on page 4-103	Generate source-level debug information.
4.35 -g0 on page 4-104	Disable generation of source-level debug information (default).
4.37 -gline-tables-only on page 4-106	Emit debug line number tables only.

Warning options

Options that control the behavior of warnings.

Option	Description
4.53 -Qunused-arguments on page 4-122	Do not emit a warning for unused driver arguments.
4.61 -W on page 4-130	Enable the specified warning.
4.62 -Wall on page 4-131	Enable all warnings.
4.63 -Warm-extensions on page 4-132	Enable warnings about the use of non-standard language features supported by armclang
4.64 -Wdeprecated on page 4-133	Enable warnings for deprecated constructs and define __DEPRECATED.
4.21 -fno-crash-diagnostics on page 4-90	Disable the auto-generation of preprocessed source files and a script for reproduction during a clang crash.
4.66 -w on page 4-135	Suppress all warnings.

Preprocessor options

Options controlling the behavior of the preprocessor.

Option	Description
4.6 -D on page 4-75	Define <macro> to <value> (or 1 if <value> omitted).
4.58 -U on page 4-127	Undefine macro <macro>.

Linker options

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

Option	Description
4.46 -L on page 4-115	Add a directory to the list of paths that the linker searches for user libraries.
4.65 -Wl, on page 4-134	Pass the comma separated arguments in <arg> to the linker.
4.68 -Xlinker on page 4-137	Pass <arg> to the linker.
4.47 -l on page 4-116	Search for the library named <library> when linking.
4.55 -shared on page 4-124	Create a shared object that can be linked against.
4.56 -static on page 4-125	Link against static libraries.

4.2 -###

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

Syntax

```
armclang -###
```

4.3 -armpl=

Enable Arm Performance Libraries (ArmPL).

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`)

sve

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

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

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

4.7 -E

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

Syntax

```
armclang -E
```

4.8 -fassociative-math

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

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

Default

Default is `-fno-associative-math`.

Syntax

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

4.9 -fcolor-diagnostics

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

Default

Default is `-fno-color-diagnostics`.

Syntax

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

4.10 -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.

4.11 -ffast-math

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

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

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

Default

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

Syntax

```
armclang -ffast-math
```

4.12 -ffinite-math-only

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

Default

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

Syntax

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

4.13 -ffp-contract=

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

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

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

- When `-ffp-contract` is set to `{off|on}`, `STDC FP_CONTRACT={OFF|ON}` is still honoured where it is specified, and can switch the behavior.
- When `-ffp-contract` is set to `fast`, the behavior is always set to `FAST` and the `STDC FP_CONTRACT` pragma is ignored.

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

Note

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

Default

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

Syntax

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

Arguments

`fast`

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

`on`

Use fused floating-point only when the language allows it. For example, for C/C++ code, floating-point contractions are allowed in a single C/C++ statement, however, for Fortran code, floating-point contractions are always enabled.

`off`

Do not use fused floating-point operations.

4.14 -fhonor-infinities

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

Default

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

Syntax

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

4.15 -fhonor-nans

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

Default

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

Syntax

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

4.16 -finline-functions

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

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

Default

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

Syntax

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

4.17 -finline-hint-functions

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

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

Default

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

Syntax

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

4.18 -fiterative-reciprocal

Enable (‘-fiterative-reciprocal’) or disable (‘-fno-iterative-reciprocal’ [default, except with ‘-Ofast’]) optimizations that replace division by reciprocal estimation and refinement.

Default

Default is `-fno-iterative-reciprocal`, except where `-Ofast` is used. Using `-Ofast` enables `-fiterative-reciprocal`.

Syntax

```
armclang -fiterative-reciprocal, -fno-iterative-reciprocal
```

4.19 -flto

Enable (‘-flto’) or disable (‘-fno-lto’ [default]) Link Time Optimizations (LTO).

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

Default

Default is `-fno-lto`.

Syntax

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

4.20 -fmath-errno

Require (‘-fmath-errno’) or do not require (‘-fno-math-errno’) math functions to indicate errors.

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

1. In C/C++ it allows sin() and cos() calls that take the same input to be combined into a more efficient sincos() call.
2. In C/C++ it allows certain pow(x, y) function calls to be eliminated completely when y is a small integral value.

Default

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

Syntax

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

4.21 -fno-crash-diagnostics

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

Default

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

Syntax

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

4.22 -fopenmp

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

Default

Default is `-fno-openmp`.

Syntax

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

4.23 -fopenmp-simd

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

Syntax

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

4.24 -freciprocal-math

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

Default

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

Syntax

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

4.25 -fsave-optimization-record

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

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

Default

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

Syntax

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

4.26 -fsigned-char

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

Syntax

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

4.27 -fsigned-zeros

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

Default

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

Syntax

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

4.28 -fsimdmath

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

Default

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

Syntax

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

4.29 -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
```

4.30 -fsyntax-only

Show syntax errors but do not perform any compilation.

Syntax

```
armclang -fsyntax-only
```

4.31 -ftrapping-math

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

Possible traps include:

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

Default

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

Syntax

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

4.32 -funsafe-math-optimizations

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

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

- `-fassociative-math`
- `-freciprocal-math`
- `-fno-signed-zeros`
- `-fno-trapping-math`

Default

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

Syntax

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

4.33 -fvectorize

Enable/disable loop vectorization (enabled by default).

Syntax

```
armclang -fvectorize, -fno-vectorize
```

4.34 -g

Generate source-level debug information.

Syntax

```
armclang -g
```

4.35 -g0

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

Syntax

```
armclang -g0
```

4.36 -gcc-toolchain=

Use the gcc toolchain at the given directory.

Syntax

```
armclang --gcc-toolchain=<arg>
```

4.37 -gline-tables-only

Emit debug line number tables only.

Syntax

```
armclang -gline-tables-only
```

4.38 -help

Display available options.

Syntax

```
armclang -help, --help
```

4.39 -help-hidden

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

Syntax

```
armclang --help-hidden
```

4.40 -I

Add directory to include search path and Fortran module search path.

Directories specified with the `-I` option apply to both the quote form of the include directive and the system header form. For example, #include “file” (quote form), and #include <file> (system header form). Directories specified with `-I` are searched before system include directories and, in `armclang|armclang++` only, after directories specified with `-iquote` (for the quoted form). If any directory is specified with both `-I` and `-isystem` then the directory is searched for as if it were only specified with `-isystem`.

For `armflang`, search for module-files in the directories that are specified with the `-I` option. Directories that are specified with `-I` are searched after the current working directory and before standard system module locations.

Syntax

```
armclang -I<dir>
```

4.41 -idirafter

Add directory to include search path after system header file directories.

Directories specified with the `-idirafter` option apply to both the quote form of the include directive and the system header form. For example, #include “file” (quote form), and #include <file> (system header form). Directories specified with the `-idirafter` option are searched after system header file directories. Directories specified with `-idirafter` are treated as system directories.

Syntax

```
armclang -idirafter<arg>
```

4.42 -include

Include file before parsing.

Syntax

```
armclang -include<file>, --include<file>
```

4.43 -iquote

Add directory to include search path. Directories specified with the '-iquote' option apply only to the quote form of the include directive.

Directories specified with the `-iquote` option only apply to the quote form of the include directive, such as #include "file". For such directives, directories specified with `-iquote` are searched first, before directories specified by `-I`.

Syntax

```
armclang -iquote<directory>
```

4.44 -isysroot

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

Syntax

```
armclang -isysroot<dir>
```

4.45 -isystem

Add a directory to the include search path, before system header file directories.

Directories specified with the `-isystem` option apply to both the quote form of the include directive and the system header form. For example, #include “file” (quote form), and #include <file> (system header form). Directories specified with the `-isystem` option are searched after directories specified with `-I` and before system header file directories. Directories specified with `-isystem` are treated as system directories. If any directory is specified with both `-I` and `-isystem` then the directory is searched for as if it were only specified with `-isystem`.

Syntax

```
armclang -isystem<directory>
```

4.46 -L

Add a directory to the list of paths that the linker searches for user libraries.

Syntax

```
armclang -L<dir>
```

4.47 -l

Search for the library named <library> when linking.

Syntax

```
armclang -l<library>
```

4.48 -march=

Specifies the base architecture and extensions available on the target.

Usage: `-march=<arg>` where `<arg>` is constructed as *name[+[no]feature+...]*:

name

`armv8-a` : Armv8 application architecture profile.

`armv8.1-a` : Armv8.1 application architecture profile.

`armv8.2-a` : Armv8.2 application architecture profile.

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>
```

4.49 -mcpu=

Select which CPU architecture to optimize for.

Syntax

```
armclang -mcpu=<arg>
```

Arguments

`native`

Auto-detect the CPU architecture from the build computer.

`thunderx2t99`

Optimize for Marvell ThunderX2 based computers.

`neoverse-n1`

Optimize for Neoverse N1 based computers.

`a64fx`

Optimize for Fujitsu A64FX based computers.

`generic`

Generate portable output suitable for any Armv8-A based computer.

4.50 -O

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

Default

The default is `-O0`. However, for the best balance between ease of debugging, code size, and performance, it is important to choose an optimization level that is appropriate for your goals.

Syntax

```
armclang -O<level>
```

Arguments

`0`

Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.

`1`

Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.

`2`

High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.

`3`

Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.

`fast`

Enables all the optimizations from level 3 including those performed with the -ffp-mode=fast armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards. -Ofast implies -ffast-math.

4.51 -o

Write output to <file>.

Syntax

```
armclang -o<file>
```

4.52 -print-search-dirs

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

Syntax

```
armclang -print-search-dirs, --print-search-dirs
```

4.53 -Qunused-arguments

Do not emit a warning for unused driver arguments.

Syntax

```
armclang -Qunused-arguments
```

4.54 -S

Stop after compiling the source and emit assembler files.

Syntax

```
armclang -S
```

4.55 -shared

Create a shared object that can be linked against.

Syntax

```
armclang -shared, --shared
```

4.56 -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
```

4.57 -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>
```

4.58 -U

Undefine macro <macro>.

Syntax

```
armclang -U<macro>
```

4.59 -v

Show commands to run and use verbose output.

Syntax

```
armclang -v
```

4.60 -version

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

Syntax

```
armclang --version, --vsn
```

4.61 -W

Enable the specified warning.

Similarly, warnings can be disabled with `-Wno-<warning>`.

Syntax

```
armclang -W<warning>
```

4.62 -Wall

Enable all warnings.

Syntax

```
armclang -Wall
```

4.63 -Warm-extensions

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

Syntax

```
armclang -Warm-extensions
```

4.64 -Wdeprecated

Enable warnings for deprecated constructs and define __DEPRECATED.

Syntax

```
armclang -Wdeprecated
```

4.65 -Wl,

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

Syntax

```
armclang -Wl,<arg>,<arg2>...
```

4.66 -w

Suppress all warnings.

Syntax

```
armclang -w
```

4.67 -working-directory

Resolve file paths relative to the specified directory.

Syntax

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

4.68 -Xlinker

Pass <arg> to the linker.

Syntax

```
armclang -Xlinker <arg>
```

Chapter 5
Standards support

This chapter describes 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

This topic describes the support for the C and C++ language standards in Arm C/C++ Compiler.

C support

For C language compilation, Arm C/C++ Compiler fully supports the C17 standard (*ISO/IEC 9899:2018*), as well as the gnu17 extensions, and prior published standards (C89 and GNU89, GNUC99 and GNU99, and C11 and GNU11).

To select which language standard Arm C/C++ Compiler should use, use the *4.57 -std= on page 4-126* compiler option with the argument:

- `c89` or `c90` for the 'ISO C 1990' standard
- `gnu89` or `gnu90` for the 'ISO C 1990 with GNU extensions' standard
- `c99` for the 'ISO C 1999' standard
- `gnu99` for the 'ISO C 1999 with GNU extensions' standard
- `c11` for the 'ISO C 2011' standard
- `gnu11` for the 'ISO C 2011 with GNU extensions' standard
- `c17` or `c18`, or for the 'ISO C 2017' standard
- `gnu17` or `gnu18` for the 'ISO C 2017 with GNU extensions' standard

The default for C code compilation is `-std=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).

Note

Exported templates, as included in the C++98 standard, were removed in the C++11 standard, and are not supported.

To select which language standard Arm C/C++ Compiler should use, use the *4.57 -std= on page 4-126* compiler option with the argument:

- `c++98` or `c++03` for the the 'ISO C++ 1998 with amendments' standard
- `gnu++98` or `gnu++03` for the the 'ISO C++ 1998 with amendments and GNU extensions' standard
- `c++11` for the the 'ISO C++ 2011 with amendments' standard
- `gnu++11` for the the 'ISO C++ 2011 with amendments and GNU extensions' standard
- `c++14` for the 'ISO C++ 2014 with amendments' standard
- `gnu++14` for the 'ISO C++ 2014 with amendments and GNU extensions' standard
- `c++17` for the 'ISO C++ 2017 with amendments' standard
- `gnu++17` for the 'ISO C++ 2017 with amendments and GNU extensions' standard

The default for C++ code compilation is `-std=gnu++14`.

Specific features that are, and are not, supported in the C++ standards are detailed on the LLVM *C++ Support in Clang*. Arm C/C++ Compiler version 20.3 is based on Clang version 9.

Related references

5.2 OpenMP 4.0 on page 5-140

5.3 OpenMP 4.5 on page 5-141

5.2 OpenMP 4.0

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

Table 5-1 Supported OpenMP 4.0 features

Open MP 4.0 Feature	Support
C/C++ Array Sections	Yes
Thread affinity policies	Yes
“simd” construct	Yes
“declare simd” construct	No
Device constructs	No
Task dependencies	Yes
“taskgroup” construct	Yes
User defined reductions	Yes
Atomic capture swap	Yes
Atomic seq_cst	Yes
Cancellation	Yes
OMP_DISPLAY_ENV	Yes

5.3 OpenMP 4.5

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

Table 5-2 Supported OpenMP 4.5 features

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

Chapter 6
Troubleshoot

This chapter describes how to diagnose problems when compiling applications using Arm C/C++ Compiler.

It contains the following sections:

6.1 Application segfaults at -Ofast optimization level

A program runs correctly when the binary is built using the `-O3` optimization level, but encounters a runtime crash or segfault with `-Ofast` optimization level.

Condition

The runtime segfault only occurs when `-Ofast` is used to compile the code. The segfault disappears when you add the `-fno-stack-arrays` option to the compile line. .

The -fstack-arrays option is enabled by default at -Ofast

When the `-fstack-arrays` option is enabled, either on its own or enabled with `-Ofast` by default, the compiler allocates arrays for all sizes using the local stack for local and temporary arrays. This helps to improve performance, because it avoids slower heap operations with `malloc()` and `free()`. However, applications that use large arrays might reach the Linux stack-size limit at runtime and produce program segfaults. On typical Linux systems, a default stack-size limit is set, such as 8192 kilobytes. You can adjust this default stack-size limit to a suitable value.

Solution

Use `-Ofast -fno-stack-arrays` instead. The combination of `-Ofast -fno-stack-arrays` disables automatic arrays on the local stack, and keeps all other `-Ofast` optimizations. Alternatively, to set the stack so that it is larger than the default size, call `ulimit -s unlimited` before running the program.

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.
    ```

6.4 Error moving Arm® Compiler for Linux modulefiles

Describes a workaround to use if you move Arm Compiler for Linux environment modulefiles.

Condition

Affected: Arm Compiler for Linux 20.3 and prior releases.

Moving Arm® Compiler for Linux modulefiles causes them to stop working

Moving Arm Compiler for Linux modulefiles after they are installed causes Arm Compiler for Linux to stop working.

By default, Arm Compiler for Linux modulefiles are configured to find the Arm Compiler for Linux binaries at a location that is relative to the modulefiles. Moving or copying the modulefiles to a new location means that the installed binaries are no longer at the same relative location to the new modulefile location. When trying to locate binaries, the broken relative links between the new modulefile location and the location of the installed binaries causes the new modulefiles to fail.

Workaround

The workaround is to update the configuration of the modulefiles at the new location so that the new modulefiles point to the location of the installed Arm Compiler for Linux binaries.

1. Copy the `modulefiles` directory to a new location. For example:

   ```
   cp -r /opt/arm/modulefiles/ <path/to/new-location>/modulefiles
   ```

   Here, `<path/to/new-location>` represents the path to the new location that you have copied the `modulefiles` directory to.
2. For each modulefile in the new location, update the `package_prefix` line in that modulefile. The `package_prefix` line must point to the location of the original Arm Compiler for Linux installation so that the installed binaries can be correctly located.

   The path to the original Arm Compiler for Linux installation is represented by `<path/to/install-dir>`

   Change:

   ```
   set package_prefix     $root_prefix/<modulefile>
   ```

   to:

   ```
   set package_prefix     <path/to/install-dir>/<modulefile>
   ```

   Note

   `$root_prefix` is a variable that is defined and used in each Arm Compiler for Linux modulefile. `$root_prefix` is set to be the path to the `modulefiles` directory to which the modulefiles for Arm Compiler for Linux were originally installed.

   For example, for a default installation, `$root_prefix` is set to `/opt/arm/modulefiles`. If, at installation, you specified a custom install location, `$root_prefix` is set as the path to the `modulefiles` directory at the custom location your specified during installation.

   To change the `package_prefix` line in one step for all the modulefiles in a directory, you can use a tool like sed, for example:

   ```
   cd <path/to/new-location>
   find . -type f -print0 | xargs -0 sed -ri 's/set package_prefix          $root_prefix/set
   package_prefix          <path\/to\/install-dir>/g'
   ```

   replacing `<path/to/install-dir>` with the path to your Arm Compiler for Linux installation directory (`/opt/arm/modulefiles` for a default installation).

Related information

Arm Allinea Studio installation instructions