Arm® C/C++ Compiler

Version 20.1

Reference Guide

Arm® C/C++ Compiler

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

Non-Confidential Proprietary Notice

Contents

Arm® C/C++ Compiler Reference Guide

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

Preface

This preface introduces the *Arm® C/C++ Compiler Reference Guide*.

It contains the following:
- *About this book* on page 8.

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 describes how to use Arm C/C++ Compiler to compile C/C++ code for Arm-based and Arm SVE-based platforms, optimize your code, and generate an executable binary.

Chapter 2 Compiler options
This page lists the command-line options supported by `armclang|armclang++` in Arm C/C++ Compiler. You can also view the available options in the in-tool `man` pages. To view the `man` pages, use `man armflang`.

Chapter 3 Coding best practice
Discusses the best practices when writing C/C++ code for Arm C/C++ Compiler.

Chapter 4 Standards support
The support status of Arm C/C++ Compiler with the OpenMP standards.

Chapter 5 Arm Optimization Report
Arm Optimization Report builds on the llvm-opt-report tool available in open source LLVM. Arm Optimization Report shows you the optimization decisions that the compiler is making, in-line with your source code, enabling you to better understand the unrolling, vectorization, and interleaving behavior.

Chapter 6 Optimization remarks
Optimization remarks provide you with information about the choices that are made by the compiler. You can use them to see which code has been inlined or they can help you understand why a loop has not been vectorized.

Chapter 7 Vector routines support
Describes how to vectorize loops in C and C++ workloads that invoke the math routines from `libm`, how to interface user vector functions with serial code, and how to expose the vector variants that are available to the compiler with the attribute `acfl_simd_variant`.

Chapter 8 Troubleshoot
Describes how to diagnose problems when compiling applications using Arm Fortran Compiler.

Chapter 9 Further resources
Describes where to find more resources about Arm C/C++ Compiler (part of Arm Compiler for Linux).

Glossary

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

See the *Arm® Glossary* for more information.

Typographic conventions

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

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

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

`monospace`
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 Reference Guide*.
- The number 101458_2010_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® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Get started

This chapter describes how to use Arm C/C++ Compiler to compile C/C++ code for Arm-based and Arm SVE-based platforms, optimize your code, and generate an executable binary.

It contains the following sections:

1.1 Get started with Arm® C/C++ Compiler

Describes how to compile your C/C++ source code and generate an executable binary with Arm C/C++ Compiler (part of Arm Compiler for Linux).

Prerequisites

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

Procedure

1. Load the environment module for Arm Compiler for Linux:
   a. As part of the installation, your system administrator must make the Arm Compiler for Linux environment modules available. To see which environment modules are available, run:

      ```
      module avail
      ```

      —————— **Note** ——————

      Depending on the configuration of Environment Modules on your system, you might need to configure the `MODULEPATH` environment variable to include the installation directory:

      ```
      export MODULEPATH=$MODULEPATH:/opt/arm/modulefiles/
      ```

      If you chose to install Arm Compiler for Linux to a custom location, replace `/opt/arm/` with the path to your installation.

      ————————————————

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

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

      For example:

      ```
      module load Generic-AArch64/SUSE/12/suites/arm-linux-compiler/20.1
      ```

   c. Check your environment. Examine the `PATH` variable. `PATH` must contain the appropriate `bin` directory from `/opt/arm`, as installed in the previous section:

      ```
      echo $PATH
      /opt/arm/arm-linux-compiler-20.1_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. Create a “Hello World” program and save it in a file, for example: `hello.c`.

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

3. To generate an executable binary, compile your program with Arm C/C++ Compiler and specify (`-o`) the input file, `hello.c`, and the binary name, `hello`:

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

4. Run the generated binary `hello`:

   ```
   ./hello
   ```

Next Steps

For more information about compiling and linking as separate steps, and how optimization levels effect auto-vectorization, see *Using the compiler* on page 1-13.

Related references

Chapter 3 Coding best practice on page 3-39

Chapter 2 Compiler options on page 2-21

1.2 Using the compiler

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

Compile and link

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

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

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

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

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

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

Increase the optimization level

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

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

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

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

or when linking object files:

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

Compile and optimize using CPU auto-detection

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

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

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

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

Note

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

Common compiler options

See `man armclang`, `armclang --help`, or *Compiler options* on page 2-21, for more information about all the supported compiler options.

`-S`

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

`-c`

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

`-o <file>`

Specifies the name of the output file.

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

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

Note

If you know your target microarchitecture, Arm recommends using the `-mcpu` option instead of `-march`.

`-mcpu=native`

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

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

Note

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

`-Olevel`

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

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

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

`--help`

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

`--version`

Displays version information.

For a detailed description of all the supported compiler options, see *Compiler options* on page 2-21.

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

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

Related tasks

1.4 Compile C/C++ code for Arm SVE and SVE2 architectures on page 1-18

Related references

Chapter 2 Compiler options on page 2-21

1.5 Get help on page 1-20

1.3 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 architectures* on page 1-18.

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>/suites/arm-linux-compiler/
  <version>
  ```

Procedure

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

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

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

   The `-O<level>` option specifies the optimization level. The `-O0` option is the lowest optimization level, while `-O3` is the highest. Arm C/C++ Compiler only performs auto-vectorization at `-O2` and higher.
2. Inspect the `<assembly-filename>.s` file to see the annotated assembly code that was created.
3. Run the executable:

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

Example 1-1 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.

```
// example1.c
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
    for (int i = 0; i < ARRAYSIZE; i++)
    {
        a[i] = b[i] - c[i];
    }
}
int main()
{
    subtract_arrays(a, b, c);
}
```

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

   ```
   armclang -O1 -S -o example1.s example1.c
   ```

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

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

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

   ```
   subtract_arrays:                        // @subtract_arrays
   // 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.

1.4 Compile C/C++ code for Arm SVE and SVE2 architectures

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

SVE and SVE2 support enables you to:

- Assemble source code containing SVE and SVE2 instructions.
- Disassemble ELF object files containing SVE and SVE2 instructions.
- Compile C and C++ code for SVE and SVE2-enabled targets, with an advanced auto-vectorizer that is capable of taking advantage of the SVE and SVE2 features.

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

Prerequisites

- Install Arm Compiler for Linux. For information about installing Arm Compiler for Linux, see *Install Arm Compiler for Linux*.
- Load the module for Arm Compiler for Linux, run:

```
module load <architecture>/<linux_variant>/<linux_version>/suites/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-filename> <source-filename>.c
     ```

     For SVE2:

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

     For SVE:

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

     For SVE2:

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

     For more information about the supported options for `-armpl`, see the `-armpl` description in *Linker options* on page 2-35.

   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 `../compiler-options/optimization-options`.

   Note

   `sve2` also enables `sve`.

2. Run the executable:

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

Example 1-2 Example

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

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

```
// example1.c
#define ARRAYSIZE 1024
int a[ARRAYSIZE];
int b[ARRAYSIZE];
int c[ARRAYSIZE];
void subtract_arrays(int *restrict a, int *restrict b, int *restrict c)
{
    for (int i = 0; i < ARRAYSIZE; i++)
    {
        a[i] = b[i] - c[i];
    }
}
int main()
{
    subtract_arrays(a, b, c);
}
```

1. Compile `example1.c` and specify the output file to be assembly (`-S`):

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

   The output assembly code is saved as `example1.s`.
2. (Optional) Inspect the output assembly code.

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

   ```
   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, that is independent of the width of SVE vector registers.
3. Run the executable:

   ```
   ./example1
   ```

Related information
Porting and Optimizing HPC Applications for Arm SVE

1.5 Get help

Describes where to find help for Arm C/C++ Compiler.

In-tool

- The `--help` option:

```
armclang --help
```

- The `man` pages:

```
man armclang
```

- The offline HTML version of this, and more, documentation, in: <install-directory>/share.

On the Arm Developer website

See: *Further resources for Arm® C/C++ Compiler* on page 9-82

Arm Support team

Contact Arm Support

Chapter 2
Compiler options

This page lists the command-line options supported by `armclang|armclang++` in Arm C/C++ Compiler. You can also view the available options in the in-tool `man` pages. To view the `man` pages, use `man armflang`.

Note

For simplicity, we have only shown the command usage with `armclang`. The options can also be used with `armclang++`, unless otherwise stated.

It contains the following sections:

2.1 Actions

Options that control what action to perform on the input.

Table 2-1 Compiler actions

Option	Description
`-E`	Only run the preprocessor. **Usage** `armclang -E`
`-S`	Only run the preprocess and compile steps. The preprocess step is not run on files that do not need it. **Usage** `armclang -S`
`-c`	Only run the preprocess, compile, and assemble steps. The preprocess step is not run on files that do not need it. **Usage** `armclang -c`
`-fopenmp`	Enable OpenMP and link in the OpenMP library, libomp. **Usage** `armclang -fopenmp`
`-fsyntax-only`	Show syntax errors but do not perform any compilation. **Usage** `armclang -fsyntax-only`

2.2 File options

Options that specify input or output files.

Table 2-2 Compiler file options

Option	Description
`--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 you can set an environment variable 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*. **Usage** `armclang --config /path/to/this/<filename>.cfg`
`-I<dir>`	Add directory to include search path. **Usage** `armclang -I<dir>`

`-include <file>`	Include file before parsing.
	
	Usage
	
	`armclang -include <file>`
	
	Or
	
	`armclang --include <file>`

`-o <file>`	Write output to `<file>`. **Usage** `armclang -o <file>`

2.3 Basic driver options

Options that affect basic functionality of the `armclang` driver.

Table 2-3 Compiler basic driver options

Option	Description
`--gcc-toolchain=<arg>`	Use the gcc toolchain at the given directory. **Usage** `armclang --gcc-toolchain=<arg>`

`-help`
`--help`
	Display available options.
	Usage
	`armclang -help`
	`armclang --help`

`--help-hidden`	Display hidden options. Only use these options if advised to do so by your Arm representative. **Usage** `armclang --help-hidden`

`-v`	Show the commands to run and use verbose output.
	Usage
	`armclang -v`
	`--version`

`--vsn`	Show the version number and some other basic information about the compiler.
	Usage
	`armclang --version`
	`armclang --vsn`

2.4 Optimization options

Options that control optimization behavior and performance.

Table 2-4 Compiler optimization options

Option	Description
`-O0`	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 larger image. This is the default optimization level. **Usage** `armclang -O0`
`-O1`	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug. **Usage** `armclang -O1`
`-O2`	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. **Usage** `armclang -O2`
`-O3`	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels. **Usage** `armclang -O3`
`-Ofast`	Enable 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. **Usage** `armclang -Ofast`

Table 2-4 Compiler optimization options (continued)

Option	Description

`-fassociative-math`
 	
`-fno-associative-math`
	Allows (`-fassociative-math`) or prevents (`-fno-associative-math`) the re-association of operands in a series of floating-point operations.
	
	For example, (a * b) + (a * c) => a * (b + c).
	
	The default is `-fno-associative-math.`
	
	Note
	This violates the ISO C and C++ language standard because it changes the program order of operations.
	
	Usage
	
	`armclang -fassociative-math`
	
	`armclang -fno-associative-math`

`-ffast-math`	Allow aggressive, lossy floating-point optimizations. **Usage** `armclang -ffast-math`
`-ffinite-math-only`	Enable optimizations that ignore the possibility of NaN and +/-Inf. **Usage** `armclang -ffinite-math-only`
`-ffp-contract={fast|on|off}`	Controls when the compiler is permitted to form fused floating-point operations (for example, FMAs). These instructions typically operate to a higher degree of accuracy than individual multiply and add instructions: • fast: Always (default for Fortran workloads). Note: They are not strictly allowed according to the C/C++ standard because they can lead to deviates from expected results. • on: Only in the presence of the FP_CONTRACT pragma (default for C/C++ workloads). • off: Never. **Usage** `armclang -ffp-contract={fast|on|off}`

`-finline`
 	
`-fno-inline`
	Enable or disable inlining (enabled by default).
	
	Usage
	
	`armclang -finline`
	
	(enable)
	
	`armclang -fno-inline`
	
	(disable)

Table 2-4 Compiler optimization options (continued)

Option	Description

`-flto`
`-fno-lto`
	Enable (`-flto`) or disable (`-fno-lto`) link time optimization. Disabled by default.
	You must pass the option to both the link and compile commands.
	Usage
	`armclang -flto`
	`armclang -fno-lto`

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

`-fsigned-zeros`
`-fno-signed-zeros`
	Allow (`-fsigned-zeros`) or prevent (`-fno-signed-zeros`) optimizations that ignore the sign of floating point zeros. Default is `-fsigned-zeros`.
	Usage
	`armclang -fsigned-zeros`
	`armclang -fno-signed-zeros`

`-fsimdmath`
`-fno-simdmath`
	Enables (`fsimdmath`) or disables (`fno-simdmath`) the use of vectorized libm libraries, to support the vectorization of loops containing calls to basic library functions, such as those declared in math.h and string.h.
	For more information, see *https://developer.arm.com/docs/101458/latest*.
	Default is `-fno-simdmath`.
	Usage
	`armclang -fsimdmath`
	`armclang -fno-simdmath`

`-fstrict-aliasing`	Tells the compiler to adhere to the aliasing rules defined in the source language. In some circumstances, this flag allows the compiler to assume that pointers to different types do not alias. Enabled by default when using `-Ofast`. **Usage** `armclang -fstrict-aliasing`

Table 2-4 Compiler optimization options (continued)

Option	Description

`-ftrapping-math`
 	
`-fno-trapping-math`
	`-ftrapping-math` tells the compiler to assume that floating point operations will cause a trap.
	
	`-fno-trapping-math` tells the compiler to assume that none of the floating point operations will cause a trap, for example, divide by zero.
	
	Possible traps include:
	• Division by zero
	• Underflow
	• Overflow
	• Inexact result
	• Invalid operation.
	
	Usage
	
	`armclang -ftrapping-math`
	
	`armclang -fno-trapping-math`

`-funsafe-math-optimizations`
 	
`-fno-unsafe-math-optimizations`
	This option enables reassociation and reciprocal math optimizations, and does not honor trapping nor signed zero.
	
	Usage
	
	`armclang -funsafe-math-optimizations`
	
	(enable)
	
	`armclang-fno-unsafe-math-optimizations`
	
	(disable)

`-fvectorize`
 	
`-fno-vectorize`
	Enable loop vectorization (default).
	
	Disable loop vectorization.
	
	Usage
	
	`armclang -fvectorize`
	
	(enable)
	
	`armclang -fno-vectorize`
	
	(disable)

Table 2-4 Compiler optimization options (continued)

Option	Description
`-mcpu=<arg>`	Select which CPU architecture to optimize for. Choose from: • `a64fx`: Optimize for A64FX-based computers. • `generic` (Default): Generates portable output suitable for any Armv8-A-based computer. To enable portable code, this is the default option when `-mcpu` is not specified. • `native`: Auto-detect the CPU architecture from the build computer. • `neoverse-n1`: Optimize for Neoverse N1-based computers. • `thunderx2t99`: Optimize for Cavium ThunderX2-based computers. **Usage** `armclang -mcpu=<arg>`

Table 2-4 Compiler optimization options (continued)

Option	Description

`-march=<arg>`	Specifies the base architecture and extensions available on the target.
	
	`-march=<arg>` where `<arg>` is constructed as *name[+[no]feature+...]*:
	
	name
	
	`armv8-a`: Armv8-A 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.
	
	Usage
	
	`armclang -march=<arg>`
	
	Examples
	
	`armclang -march=armv8-a`
	
	`armclang -march=armv8-a+sve`
	
	`armclang -march=armv8-a+sve2`

2.5 Workload compilation options

Options that affect the way C language workloads compile.

Table 2-5 Workload compilation options

Option	Description

`-std=<arg>`
`--std=<arg>`
	Language standard to compile for. The list of valid standards depends on the input language, but adding `-std=<arg>` to a build line will generate an error message listing valid choices.
	Usage
	`armclang -std=<arg>`
	`armclang --std=<arg>`

2.6 Development options

Options that support code development.

Table 2-6 Compiler development options

Option	Description

`-fcolor-diagnostics`
`-fno-color-diagnostics`
	Use colors in diagnostics.
	Usage
	`armclang -fcolor-diagnostics`
	Or
	`armclang -fno-color-diagnostics`

`-g`
`-g0` (default)
`-gline-tables-only`
	`-g`, `-g0`, and `-gline-tables-only` control the generation of source-level debug information:
	• `-g` enables debug generation.
	• `-g0` disables generation of debug and is the default setting.
	• `-gline-tables-only` enables DWARF line information for location tracking only (not for variable tracking).
	Note
	If more than one of these options are specified on the command line, the option specified last overrides any before it.
	Usage
	`armclang -g`
	Or
	`armclang -g0`
	Or
	`armclang -gline-tables-only`

2.7 Warning options

Options that control the behavior of warnings.

Table 2-7 Compiler warning options

Option	Description
`fno-math-errno`	Require math functions to indicate errors. Use this flag if your source code never uses errno to check the status of math function calls. This will 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.

`-W<warning>`
`-Wno-<warning>`
	Enable or disable the specified warning.
	Usage
	`armclang -W<warning>`

`-Wall`	Enable all warnings. **Usage** `armclang -Wall`
`-Warm-extensions`	Enable warnings about the use of non-standard language features supported by Arm Compiler for Linux. **Usage** `armclang -Warm-extensions`
`-Warm-warnings`	Enable warnings about deprecated features which will not be supported in newer versions of Arm Compiler for Linux. **Usage** `armclang -Warm-warnings`
`-w`	Suppress all warnings. **Usage** `armclang -w`

2.8 Pre-processor options

Options that control pre-processor behavior.

Table 2-8 Compiler pre-processing options

Option	Description
`-D <macro>=<value>`	Define `<macro>` to `<value>` (or 1 if `<value>` is omitted). **Usage** `armclang -D<macro>=<value>`
`-U`	Undefine macro `<macro>`. **Usage** `armclang -U<macro>`

2.9 Linker options

Options that control linking behavior and performance.

Table 2-9 Compiler linker options

Option	Description
`-Wl,<arg>`	Pass the comma-separated arguments in `<arg>` to the linker. **Usage** `armclang -Wl,<arg>, <arg2>...`
`-Xlinker <arg>`	Pass `<arg>` to the linker. **Usage** `armclang -Xlinker <arg>`

Table 2-9 Compiler linker options (continued)

Option	Description
`-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, and auto-vectorization of mathematical functions (disable this using `-fno-simdmath`). Supported arguments are: • `sve`: Use the SVE library from <pl>. ——— **Note** ——— To target SVE-enabled architectures and use the SVE library of Arm Performance Libraries library, use `-armpl=sve,<arg2>,<arg3>` with `-march=armv8-a+sve`. ——————— • `lp64`: Use 32-bit integers. (default) • `ilp64`: Use 64-bit integers. Inverse of `lp64`. • `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`) Separate multiple arguments using a comma, for example: `-armpl=<arg1>,<arg2>`. **Default option behavior** 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`. If the `-fopenmp`*Actions* on page 2-22 option is also specified, the default behavior of `armpl` becomes `-armpl=lp64,parallel`. For more information on using `-armpl`, see the *Library selection* web page. **Usage** `armclang code_with_math_routines.c -armpl{=<arg1>,<arg2>}` **Examples** To specify a 64-bit integer, OpenMP multi-threaded implementation for an A64FX-based computer: `armclang code_with_math_routines.c -armpl=lp64,parallel -mcpu=a64fx` ——— **Note** ——— Specifying the A64FX target enables the compiler to use SVE instructions and to link in the SVE-enabled A64FX library (without the requirement to specify `sve` as one of the arguments passed to `-armpl`). ——————— To specify a 32-bit integer single-threaded implementation for a Neoverse N1-based computer: `armclang code_with_math_routines.c -armpl=lp64,sequential -mcpu=neoverse-n1` To use the serial, ilp64 ArmPL libraries that are optimized for the CPU architecture of the build computer: `armclang code_with_math_routines.c -armpl=ilp64 -mcpu=native` To use the parallel, lp64 ArmPL libraries, with portable output suitable for any Armv8-A-based computer: `armclang code_with_math_routines.c -armpl -fopenmp -mcpu=generic` To use the parallel, lp64 ArmPL SVE libraries, with output suitable for any SVE-enabled Armv8-A-based computer: `armclang code_with_math_routines.c -armpl=sve -fopenmp -march=armv8-a+sve` To use the parallel, ilp64 ArmPL libraries, optimized for a Neoverse N1-based computer: `armclang code_with_math_routines.c -armpl=parallel,ilp64 -mcpu=neoverse-n1`

Table 2-9 Compiler linker options (continued)

Option	Description
`-l<library>`	Search for the library named `<library>` when linking.
`-l<library>`	Search for the library named `<library>` when linking. **Usage** `armclang -l<library>`
`-larmflang`	At link-time, include this option to use the default Fortran libarmflang runtime library for both serial and parallel (OpenMP) Fortran workloads. **Note** • This option is set by default when linking using `armflang`. • You need to explicitly include this option if you are linking with `armclang` instead of `armflang` at link-time. • This option only applies to link-time operations. **Usage** `armclang -larmflang` See notes in description.
`-larmflang-nomp`	At link-time, use this option to avoid linking against the OpenMP Fortran runtime library. **Note** • Enabled by default when compiling and linking using `armflang` with the `-fno-openmp` option. • You need to explicitly include this option if you are linking with `armclang` instead of `armflang` at link-time. • Do not use `-larmflang-nomp` if your code has been compiled with the `-lomp` or `-fopenmp` options. • Use this option with care. When using this option, do not link to any OpenMP-utilizing Fortran runtime libraries in your code. • This option only applies to link-time operations. **Usage** `armclang -larmflang-nomp` See notes in description.

Table 2-9 Compiler linker options (continued)

Option	Description

`-shared`
`--shared`
	Causes library dependencies to be resolved at runtime by the loader.
	This is the inverse of -static. If both options are given, all but the last option will be ignored.
	Usage
	`armclang -shared`
	Or
	`armclang --shared`

`-static`
`--static`
	Causes library dependencies to be resolved at link-time.
	This is the inverse of `-shared`. If both options are given, all but the last option is ignored.
	Usage
	`armclang -static`
	Or
	`armclang --static`

To link serial or parallel Fortran workloads using `armclang` instead of `armflang`, include the `-larmflang` option to link with the default Fortran runtime library for serial and parallel Fortran workloads. You also need to pass any options that are required to link using the required mathematical routines for your code.

To statically link, in addition to passing `-larmflang` and the mathematical routine options, you also need to pass:

- `-static`
- `-lomp`
- `-lrt`

To link serial or parallel Fortran workloads using `armclang` instead of `armflang`, without linking against the OpenMP runtime libraries, instead pass `-armflang-nomp`, at link-time. For example, pass:

- `-larmflang-nomp`
- Any mathematical routine options, for example: `-lm` or `-lamath`.

Again, to statically link, in addition to `-larmflang-nomp` and the mathematical routine options, you also need to pass:

- `-static`
- `-lrt`

——— **warn** ———

- Do not link against any OpenMP-utlizing Fortran runtime libraries when using this option.
- All lockings and thread local storage will be disabled.
- Arm does not recommend using the `-larmflang-nomp` option for typical workloads. Use this option with caution..

——— **Note** ———

The `-lompstub` option (for linking against libompstub) might still be needed if you have imported `omp_lib` in your Fortran code but not compiled with `-fopenmp`.

Chapter 3
Coding best practice

Discusses the best practices when writing C/C++ code for Arm C/C++ Compiler.

It contains the following sections:

3.1 Coding best practice for auto-vectorization

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

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

Use restrict

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

Note

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

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

Use pragmas

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

For more information, see *Control auto-vectorization with pragmas* on page 3-41.

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.

warn

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

3.2 Control auto-vectorization with pragmas

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

For more information about the `pragma clang loop` directives, see .

Note

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

Enable auto-vectorization with pragmas

Auto-vectorization is enabled at the optimization level `-O2` or higher. When enabled, auto-vectorization examines all loops.

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

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

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

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

Note

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

warn

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

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

Unrolling

Unrolling a scalar loop, for example:

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

by a factor of two, gives:

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

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

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

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

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

Interleaving

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

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

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

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

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

——— **Note** ———

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

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

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

The Arm SIMD (or Advanced SIMD) architecture, its associated implementations, and supporting software, are commonly referred to as Neon technology. There are SIMD instruction sets for both AArch32 (equivalent to the Armv7 instructions) and for AArch64. Both can be used to accelerate repetitive operations on the large data sets commonly encountered with High Performance Computing (HPC) applications.

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

For example, consider the following SIMD instruction:

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

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

Coding with SIMD

To take advantage of SIMD instructions in your code:

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

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

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

  At the lowest optimization level `-O0` auto-vectorization is never performed, even if you specify `-fvectorize`.
- Use intrinsics directly in your C code.

  Intrinsics are C or C++ pseudo-function calls that the compiler replaces with the appropriate SIMD instructions. Intrinsics let you use the data types and operations available in the SIMD implementation, while allowing the compiler to handle instruction scheduling and register allocation. The available intrinsics are defined in the *language extensions document*.
- Write SIMD assembly code.

  Although it is technically possible to optimize SIMD assembly by hand, it can be difficult because the pipeline and memory access timings have complex inter-dependencies. Instead of hand-writing assembly, Arm recommends the use of intrinsics.

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

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

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

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

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

Optimize your code for SVE

To optimize your code using SVE, you can either:

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

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

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

  At the lowest optimization level, `-O0`, auto-vectorization is never performed, even if you specify `-fvectorize`. See *Optimization options* on page 2-25 for more information on setting these options.
- Write SVE assembly code.

  For more information, see *Writing inline SVE assembly* on page 3-48.

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

Related information

Scalable Vector Extension (SVE, and SVE2) information
Explore the Scalable Vector Extension (SVE)
Arm A64 Instruction Set Architecture
White Paper: A sneak peek into SVE and VLA programming
White Paper: Arm Scalable Vector Extension and application to Machine Learning
Arm C Language Extensions (ACLE) for SVE
DWARF for the ARM 64-bit Architecture (AArch64) with SVE support
Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support
Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A

3.5 Prefetching with __builtin_prefetch

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

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

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

__builtin_prefetch syntax

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

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

where:

`addr` (required)

Represents the address of the memory.

`rw` (optional)

A compile-time constant which can take the values:

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

`locality` (optional)

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

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

Note

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

Note

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

Example

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

```
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
```

Related information

Explore the Scalable Vector Extension (SVE)

SVE Vector Length Agnostic programming

3.6 Writing inline SVE assembly

Inline assembly (or inline asm) provides a mechanism for inserting hand-written assembly instructions into C and C++ code. This lets you vectorize parts of a function by hand 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-hand 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 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 4
Standards support

The support status of Arm C/C++ Compiler with the OpenMP standards.

It contains the following sections:

4.1 OpenMP 4.0

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

Table 4-1 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

4.2 OpenMP 4.5

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

Table 4-2 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 5
Arm Optimization Report

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

Unrolling

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

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

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

Vectorization

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

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

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

———— **Note** ————

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

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

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

```
SVE scaling factor = <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 5-60.

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>
```

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**.

It contains the following sections:
- *5.1 How to use Arm Optimization Report* on page 5-58.
- *5.2 arm-opt-report reference* on page 5-60.

5.1 How to use Arm Optimization Report

This topic describes how to use Arm Optimization Report.

Prerequisites

Download and install Arm Compiler for Linux version 20.0+. For more information, see *Download Arm Compiler for Linux* and *Installation*.

Procedure

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

   An `<filename>.opt.yaml` report is generated by Arm Compiler, where `<filename>` is the name of the binary.
2. To inspect the `<filename>.opt.yaml` report, as augmented source code, use `arm-opt-report`:

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

   Annotated source code appears in the terminal.

Example 5-1 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

5.2 arm-opt-report reference on page 5-60

Related information

Arm Compiler for Linux and Arm Allinea Studio

Take a trial

Help and tutorials

5.2 arm-opt-report reference

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

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

Note

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

Possible annotations are:

Annotation	Description
`I`	A function was inlined.
`U<N>`	A loop was unrolled `<N>` times.
`V<F, I>`	A loop has been vectorized. Each vector iteration that is performed has the equivalent of `F*I` scalar iterations. Vectorization Factor, `F`, is the number of scalar elements that are processed in parallel. Interleave count, `I`, is the number of times the vector loop was unrolled.
`VS<F,I>`	A loop has been vectorized using scalable vectors. Each vector iteration performed has the equivalent of `N*F*I` scalar iterations, where `N` is the number of vector granules, which can vary according to the machine the program is run on. For example, 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

5.1 How to use Arm Optimization Report on page 5-58

Chapter 6
Optimization remarks

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

By default, Arm C/C++ Compiler prints compilation information to `stderr`. Optimization remarks prints this optimization information to the terminal, or you can choose to pipe them to an output file.

To enable optimization remarks, choose from following `Rpass` options:

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

For each option, replace `<regex>` with an expression for the type of remarks you wish to view.

Recommended `<regexp>` queries are:

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

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

Note

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

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

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

It contains the following section:

- *6.1 Enable Optimization remarks* on page 6-64.

6.1 Enable Optimization remarks

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

Procedure

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

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

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

   Result:

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

2. Pipe the loop vectorization optimization remarks to a file. For example, to pipe to a file called `vecreport.txt`, use:

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

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

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

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

Related information
Arm C/C++ Compiler

Chapter 7
Vector routines support

Describes how to vectorize loops in C and C++ workloads that invoke the math routines from `libm`, how to interface user vector functions with serial code, and how to expose the vector variants that are available to the compiler with the attribute `acfl_simd_variant`.

It contains the following sections:

7.1 Vector math routines in Arm® C/C++ Compiler

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

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

Examples

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

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

C example with loop invoking `sin`:

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

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

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

How it works

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

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

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

- *Arm Optimized Routines*
- *SLEEF*
- *PGMath*

Limitations

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

Contact Arm Support

Related information
SLEEF
Arm Optimized Routines

PGMath
Vector function ABI specification for AArch64

7.2 Support level for declare simd

`declare simd` cannot be used to auto-vectorize scalar function declarations using Arm Compiler for Linux.

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 neon_foo(float64x2_t);
#pragma omp declare variant(neon_foo) \
        match(construct = {simd(simdlen(2), notinbranch)}, \
              device = {isa("simd")})
int foo(double);
// loop in the user code, in user_code.c
#include "path/to/myvecroutines.h"
void do_something(int * a, double * b, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    a[i] = foo(b[i]);
}
```

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

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

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

The following example shows the basic functionality for SVE vectorization:

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

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

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

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

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

declare variant support

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

The current level of support covers the following features:

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

  ———— **Note** ————

  There is no support for the following clauses in the `simd` trait of the `construct` set:

  — `uniform`

  — `aligned`

  The `linear` clause in the `simd` trait is only supported for pointers with a linear step of 1. There is no support for linear modifiers.

  ————————————

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

- `simdlen(N)` is supported when targeting Advanced SIMD vectorization. Its value must be a power of 2 so that the `WDS(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 to allow 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: example01.c
#include <arm_neon.h>
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo(float64x2_t a);
#pragma omp declare variant(user_vector_foo) \
        match(construct = {simd(simdlen(2), notinbranch)}, \
              device = {isa("simd")})
double foo(double);
void do_something(double * restrict a, double * b, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    a[i] = foo(b[i]);
}
```

To produce a vector loop that invokes `user_vector_foo`, compile the example code with `armclang -fopenmp -O2 -c -S -o - example01.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: example02.c
#include <arm_neon.h>
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo_linear(float64x2_t, float *);
 #pragma omp declare variant(user_vector_foo_linear) \
        match(construct = {simd(simdlen(2), notinbranch, linear(b))}, \
              device = {isa("simd")})
double foo_linear(double a, float* b);
void do_something_linear(double * restrict a, double * b, float * x, unsigned N) {
  for (unsigned i = 0; i < N; ++i)
    a[i] = foo_linear(b[i], &x[i]);
}
```

To produce a vector loop that invokes `user_vector_foo_linear`, compile this code with `armclang -fopenmp -O2 -c -S -o - example02.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 this 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
```

7.3 Attribute acfl_simd_variant

`armclang` can interface with user-provided vector functions to vectorize loops that invoke serial functions. In the following test we refer to such vector functions as *vector variants*.

To expose the vector variants that are available to the compiler, use the attribute `acfl_simd_variant` on the declarations of the scalar functions.

```
#include <arm_sve.h>
// Declaration of the vector function.
svint32_t sve_foo(svfloat64_t, svbool_t);
// Declaration of the scalar function.
int foo(double) __attribute__((acfl_simd_variant(sve_foo, 0, "mask", "sve")));
// Loop invoking scalar `foo`.
void do_something(int * a, double * b, unsigned N) {
  for (unsigned i = 0; i < N; ++i) {
    a[i] = foo(b[i]);
  }
}
```

The compiler vectorizes the loop in the example when targeting SVE with at least `-O2` optimization level, invoking `sve_foo` in the body of the vector loop:

```
$> armclang -march=armv8a+sve -O2 -c user_code.c -o objfile.o
```

The attribute can target the following cases:

1. Advanced SIMD (Neon) vector variants.
2. Vector Length Agnostic (VLA) SVE vector variants.

The compiler checks that the signature of the vector variant conforms to the Vector Function ABI specification for AArch64 (VFABI), available at *https://developer.arm.com/architectures/system-architectures/software-standards/abi*.

Attribute syntax

The attribute operates with the syntax:

```
 acfl_simd_variant(<variant-func-id>, <simdlen>, <mask>, <isa>{, <linears>})
 <variant-func-id>:= The name of a function variant that is a
                     base language identifier.
<simdlen> := <non negative number> ( 0 is for "scalable")
 <mask>    := "mask" | "nomask"
 <isa>     := "simd" | "sve"
<linears> :=  <parameter_position>{,<parameter_position>, ...}
<parameter_position> := Position of the linear parameter (starts with 1).
```

Level of support

The supported scalar function signature in C and C++ (template functions excluded) are in the forms:

- `void (Ty1, Ty2,..., TyN)`
- `Ty1 (Ty2, Ty3,..., TyN)`

where `Ty#n` are:

- Any of the integral type values of size 1, 2, 4, or 8 (in bytes), signed and unsigned.
- Floating-point type values of half, single, or double precision.
- Pointers to any of the previous types, which must be listed in the `<linears>` section of the attribute. Note that this feature is limited to only enable vectorization of functions whose pointer parameters are operating on contiguous memory that is traversed during the loop execution. In particular, vectorization of calls that operates on loop-invariant pointers is disabled.

Common mapping rules

1. Each parameter and the return value of the scalar function, maps to a correspondent parameter and return value in the signature of the vector variant, in the same order.
2. A parameter that is listed in the `<linears>` of the attribute 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 specified in the `<simdlen>` parameter of the attribute. Values of `N` that do not correspond to Neon ACLE types are unsupported.
2. If you specify `<mask>="mask"` an additional 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:
   - `N` is the value specified in the `<simdlen>` field of the attribute.
   - `BITS` is the size (in bits) of the *Narrowest Data Size (NDS)* associated to the scalar function, as defined in the VFABI.
   - 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.

For example, consider the vector variant `float64x2_t F(float32x2_t vx, uint43x2_t mask)`, associated to the scalar function `double f(float f)` with `<simdlen>=2`, `<mask>="mask"`, and `<isa>="simd"`. The NDS of `f` is 4, therefore the lane subdivision of the mask parameter of the vector variant is 32-bit. Active lanes are set by the byte sequence `0xffffffff`, inactive lanes are set with `0x00000000`. Conversely, consider the vector variant `int16x4_t G(float32x4_t vx, uint16x4_t)`, associated to the scalar function `int16_t g(float)`. The NDS of `g` is 2, therefore the lane subdivision of the mask is 16-bit. Active lanes are set by the byte sequence `0xffff`, inactive lanes are set with `0x0000`.

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 as the last parameter in the signature of the vector variant, whether `<mask>` is set to `"mask"` or `"nomask"`. 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, consider the vector variant `svfloat64_t f_vector(svfloat32_t vx, svbool_t)`, associated to the scalar function `double f_scalar(float f)` with `<simdlen>=0`, `<mask>="mask"`, and `<isa>="sve"`. The WDS of `f_scalar` is 8, therefore the lane subdivision of the mask parameter of the vector variant is 8-bit. Active lanes are set by the bit sequence `00000001`, inactive lanes are set with `00000000`. Conversely, consider the vector variant `svfloat16_t g_vector(svfloat32_t vx, svbool_t)`, associated to the scalar function `float16_t g_scalar(float)`, with `<simdlen>=0`, `<mask>="mask"`, and `<isa>="sve"`. The WDS of `g_scalar` is 4, therefore the lane subdivision of the mask is 2-bit. Active lanes are set by the bit sequence `01`, inactive lanes are set with `00`.

Examples

The following examples show you how to vectorize with the custom user vector function. The examples use `-O2` to enable the minimal level of optimizations to allow the loop auto-vectorization process.

Note that the use of `-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 20.1.

For both Advanced SIMD and SVE, the `<linears>` lits of parameters of the attribute can improve the vectorization of functions accessing memory through contiguous pointers (check **Level of support** for a list of limitation of this feature). 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, the position of the pointers in the scalar definition (in positions `2` and `3` in the signature) must be passed to the attribute as follows:

```
#include <arm_sve.h>
void CustomSinCos(svfloat64_t, double *, double *, svbool_t);
void sincos(double in, double *, double *) \
  __attribute__((acfl_simd_variant(CustomSinCos, 0, "mask", "sve", 2, 3)));
```

```
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

The following two examples demonstrate using the `acfl_simd_variant` attribute, without and with passing pointer parameters in the `<linears>` list of the attribute, in Advanced SIMD code.

Note that the attribute `aarch64_vector_pcs` (see VFABI) needs to be manually specified to the definition of the Neon vector variants to enable better calling conventions for vector functions.

Simple

```
// filename: example01.c
#include <arm_neon.h>
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo(float64x2_t a);
double foo(double) __attribute__((acfl_simd_variant(user_vector_foo, 2, "nomask", "simd")));
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 -O2 -c -S example01.c -o - -mllvm -force-vector-interleave=1`:

```
.LBB0_4:                                // %vector.body
        ldr     q0, [x25], #16
        bl      user_vector_foo
        subs    x23, x23, #2            // =2
        str     q0, [x24], #16
        b.ne    .LBB0_4
```

With linear parameters

Refer to **Level of support** for the limitation of this feature.

```
// filename: example02.c
#include <arm_neon.h>
__attribute__((aarch64_vector_pcs)) float64x2_t user_vector_foo_linear(float64x2_t, float *);
double foo_linear(double a, float* b) \
  __attribute__((acfl_simd_variant(user_vector_foo_linear, 2, "nomask", "simd", 2)));
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`, compile the example code with `armclang -O2 -c -S example02.c -o - -mllvm -force-vector-interleave=1`:

```
.LBB0_4:                                // %vector.body
        ldr     q0, [x26], #16
        shl     v1.2d, v16.2d, #2
        add     v1.2d, v17.2d, v1.2d
        fmov    x0, d1
        bl      user_vector_foo_linear
        str     q0, [x25], #16
        subs    x24, x24, #2            // =2
        add     v16.2d, v16.2d, v18.2d
        b.ne    .LBB0_4
```

SVE examples

The following two examples demonstrate using the `acfl_simd_variant` attribute, without and with passing pointer paramters in the `<linears>` list of the attribute, in SVE code.

Simple

```
// filename: example03.c
#include <arm_sve.h>
svfloat64_t user_vector_foo_sve(svfloat64_t, svbool_t);
double foo(double) __attribute__((acfl_simd_variant(user_vector_foo_sve, 0, "nomask",
"sve")));
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 -march=armv8-a+sve -O2 -c -S example03.c -o - -mllvm -force-vector-interleave=1`:

```
.LBB0_2:                                // %vector.body
        ld1d    { z0.d }, p4/z, [x19, x21, lsl #3]
        mov     p0.b, p4.b
        bl      user_vector_foo_sve
        st1d    { z0.d }, p4, [x20, x21, lsl #3]
        incd    x21
        whilelo p4.d, x21, x22
        b.mi    .LBB0_2
```

With linear parameters

Refer to **Level of support** for the limitation of this feature.

```
// filename: example04.c
#include <arm_sve.h>
svfloat64_t user_vector_foo_linear_sve(svfloat64_t, float *, svbool_t);
double foo_linear(double a, float* b) \
  __attribute__((acfl_simd_variant(user_vector_foo_linear_sve, 0, "mask", "sve", 2)));
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 _sve`, compile the example code with `armclang -march=armv8-a+sve -O2 -c -S example04.c -o - -mllvm -force-vector-interleave=1`:

```
.LBB0_2:                                // %vector.body
        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
```

Chapter 8
Troubleshoot

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

It contains the following sections:

8.1 Application segfaults at -Ofast optimization level

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

Condition

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

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

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

Solution

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

If you continue to experience problems, *Contact Arm Support*.

8.2 Compiling with the -fpic option fails when using GCC compilers

Describes the difference between the `-fpic` and `-fPIC` options when compiling for Arm with GCC and Arm Compiler for Linux.

Condition

Failure can occur at the linking stage when building Position-Independent Code (PIC) on AArch64 using the lower-case `-fpic` compiler option with GCC compilers (gfortran, gcc, g++), in preference to using the upper-case `-fPIC` option.

Note

- This issue does not occur when using the `-fpic` option with Arm Compiler for Linux (`armflang`/`armclang`/`armclang++`), and it also does not occur on x86_64 because -fpic operates the same as `-fPIC`.
- PIC is code which is suitable for shared libraries.

Cause

Using the `-fpic` compiler option with GCC compilers on AArch64 causes the compiler to generate one less instruction per address computation in the code, and can provide code size and performance benefits. However, it also sets a limit of 32k for the Global Offset Table (GOT), and the build can fail at the executable linking stage because the GOT overflows.

Note

When building PIC with Arm Compiler for Linux on AArch64, or building PIC on x86_64, `-fpic` does not set a limit for the GOT, and this issue does not occur.

Solution

Consider using the `-fPIC` compiler option with GCC compilers on AArch64, because it ensures that the size of the GOT for a dynamically linked executable will be large enough to allow the entries to be resolved by the dynamic loader.

8.3 Error messages when installing Arm® Compiler for Linux

If you experience a problem when installing Arm Compiler for Linux, consider the following points.

- To perform a system-wide install, ensure that you have the correct permissions. If you do not have the correct permissions, the following errors are returned:
  - Systems using RPM Package Manager (RPM):

    ```
    error: can't create transaction lock on /var/lib/rpm/.rpm.lock (Permission denied)
    ```
  - Debian systems using dpkg:

    ```
    dpkg: error: requested operation requires superuser privilege
    ```
- If you install using the `--install-to <directory>` option, ensure that the system you are installing on has the required rpm or dpkg binaries installed. If it does not, the following errors are returned:
  - Systems using RPM Package Manager (RPM):

    ```
    Cannot find 'rpm' on your PATH. Unable to extract .rpm files.
    ```
  - Debian systems using dpkg:

    ```
    Cannot find 'dpkg' on your PATH. Unable to extract .deb files.
    ```

Chapter 9
Further resources

Describes where to find more resources about Arm C/C++ Compiler (part of Arm Compiler for Linux).

It contains the following section:

9.1 Further resources for Arm® C/C++ Compiler

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

Arm Allinea Studio:

- *Arm Allinea Studio*
- *Arm C/C++ Compiler web page*
- *Installation instructions*
- *Release history*
- *Supported platforms*

Porting guidance

- *Porting and tuning resources*
- *Arm GitLab Packages wiki*
- *Arm HPC Ecosystem*

SVE and SVE2 information

- *Scalable Vector Extension (SVE, and SVE2) information*
- For an overview of SVE and why it is useful for HPC, see *Explore the Scalable Vector Extension (SVE)*.
- For a list of SVE and SVE2 instructions, see the *Arm A64 Instruction Set Architecture*.
- *White Paper: A sneak peek into SVE and VLA programming*. An overview of SVE with information on the new registers, the new instructions, and the Vector Length Agnostic (VLA) programming technique, with some examples.
- *White Paper: Arm Scalable Vector Extension and application to Machine Learning*. In this white paper, code examples are presented that show how to vectorize some of the core computational kernels that are part of machine learning system. These examples are written with the Vector Length Agnostic (VLA) approach introduced by the Scalable Vector Extension (SVE).
- *Arm C Language Extensions (ACLE) for SVE*. The SVE ACLE defines a set of C and C++ types and accessors for SVE vectors and predicates.
- *DWARF for the ARM® 64-bit Architecture (AArch64) with SVE support*. This document describes the use of the DWARF debug table format in the Application Binary Interface (ABI) for the Arm 64-bit architecture.
- *Procedure Call Standard for the ARM 64-bit Architecture (AArch64) with SVE support*. This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the Arm 64-bit architecture.
- *Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*. This supplement describes the Scalable Vector Extension to the Armv8-A architecture profile.

Support and sales:

- If you encounter a problem when developing your application and compiling with the Arm C/C++ Compiler, see the *troubleshooting topics* on the Arm Developer website.
- *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.

———————————