Arm C/C++ Compiler Reference Guide

Version 19.1.0

Document Number 101458_1910_00

CONTENTS:

Release information

Table 1: Document history

Issue	Date	Confidentiality	Change
1900_00	2 November 2018	Non-Confidential	19.0.0
1910_00	8 March 2019	Non-Confidential	19.1.0

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

Product Status

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

Web Address

http://www.arm.com

CHAPTER

ONE

GETTING STARTED

Arm C/C++ Compiler is an auto-vectorizing compiler for the 64-bit Arm®v8-A architecture. This getting started tutorial shows you how to install, compile C/C++ code, use different optimization levels, and generate an executable.

The Arm C/C++ Compiler tool chain for the 64-bit Arm®v8-A architectureenables you to compile C/C++ code for Arm®v8-A compatible platforms, with an advanced auto-vectorizer capable of taking advantage of SIMD features.

1.1 Getting started with Arm C/C++ Compiler

Arm C/C++ Compiler is an auto-vectorizing compiler for the 64-bit Arm®v8-A architecture, with optional support for the Scalable Vector Extension (SVE). This tutorial shows how to compile and generate executables that will run on any 64-bit Arm®v8-A architecture.

1.1.1 Installing Arm C/C++ Compiler

Refer to Help and tutorials for details on how to perform the installation on Linux.

1.1.2 Environment Configuration

Note: Full instructions on configuring your environment for Arm C/C++ Compiler are included in the installation guide.

Your administrator should have already installed Arm C/C++ Compiler and made the environment module available.

To see which environment modules are available:

```
module avail
```

Note: You may need to configure the MODULEPATH environment variable to include the installation directory:

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

To configure your Linux environment to make Arm C/C++ Compiler available:

```
module load <architecture>/<linux_variant>/<linux_version>/suites/arm-compiler-for-
↪hpc/<version>
```

For example:

```
module load Generic-AArch64/SUSE/12/suites/arm-compiler-for-hpc/19.1
```

You can check your environment by examining the `PATH` variable. It should contain the appropriate `bin` directory from `/opt/arm`, as installed in the previous section:

```
echo $PATH /opt/arm/arm-compiler-for-hpc-19.1_Generic-AArch64_SUSE-12_aarch64-linux/
↪bin:...
```

You can also use the `which` command to check that the Arm C/C++ Compiler `armclang` command is available:

```
which armclang /opt/arm/arm-compiler-for-hpc-19.1_Generic-AArch64_SUSE-12_aarch64-
↪linux/bin/armclang
```

Note: You might want to consider adding the `module load` command to your `.profile` to run it automatically every time you log in.

1.1.3 Compiling and running a simple 'Hello World' program

This simple example illustrates how to compile and run a simple Hello World program.

1. Create a simple "Hello World" program and save it in a file. In our case, we have saved it in a file named hello.c.

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

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

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

3. Now you can run the generated binary `hello` as shown below:

```
./hello
```

In the following sections we discuss the available compiler options in more detail and, towards the end of this tutorial, illustrate using them with a more advanced example.

1.1.4 Generating executable binaries from C and C++ code

To generate an executable binary, compile a program 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:

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

1.1.5 Compiling and linking object files as separate steps

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
```

1.1.6 Increasing 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 flag can be specified when generating a binary, such as:

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

The optimization flag 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
```

1.1.7 Compiling and optimizing 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 it is being run on, and optimize accordingly.

This option supports a range of Arm®v8-A based SoCs, including ThunderX2.

Note: The optimization performed according to the auto-detected architecture and processor is independent of the optimization level denoted by the `-Olevel` option.

1.1.8 Advanced example: Generating Arm assembly code from C and C++ code

Arm C/C++ Compiler can produce annotated assembly, and this is a good first step to see how the compiler vectorizes loops.

Note: Different compiler options are required to make use of SVE functionality. If you are using SVE, please refer to Compiling C/C++ code for Arm SVE architectures.

1.1.9 Example

The following C program 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);
}
```

Compile the program as follows:

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

The flag `-S` is used to output assembly code.The output assembly code is saved as `example1.s`. The section of the generated assembly language file containing the compiled `subtract_arrays` function appears 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).

1.1.10 Enable auto-vectorization

To enable auto-vectorization, increase the optimization level using 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:

```
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 containing the compiled `subtract_arrays` function appears 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
```

(continues on next page)

(continued from previous page)

```
        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. Consequently each loop iteration moves through the array by 32 bytes (2 sets * 4 elements * 4 bytes) at a time.

1.1.11 Common compiler options

See `armclang --help`, *Compiler options*, and the LLVM documentation for more information about all supported 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+sve`.

`-mcpu=native`

Enables the compiler to automatically detect the CPU it is being run on and optimize accordingly. This supports a range of Arm®v8-A based SoCs, including ThunderX2.

`-Olevel`

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

`--help`

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

`--version`

Displays version information.

1.1.12 Get help

For a list of all the supported options, use:

```
armclang --help
```

To see detailed descriptions of all supported options, use:

```
man armclang
```

For a list of command-line options, see *Compiler options*.

If you have problems and would like to contact our support team, get in touch:

Contact Arm Support

1.1.13 Related information

- Coding best practice for auto-vectorization.
- Optimizing C/C++ code with Arm SIMD.
- Using pragmas to control auto-vectorization.
- *Compiler options*.

CHAPTER

TWO

COMPILER OPTIONS

Command-line options supported by `armclang` and `armclangc++` within Arm C/C++ Compiler.

The supported options are also available in the `man` pages in the tool. To view them, use:

```
man armclang
```

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

2.1 Actions

Options that control what action to perform on the input.

Table 1: Compiler actions

Option	Description
`-E`	Only run the preprocessor. **Usage** `armclang -E`
`-S`	Only run preprocess and compilation steps. **Usage** `armclang -S`
`-c`	Only run preprocess, compile, and assemble steps. **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: Compiler file options

Option	Description
`-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 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 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 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 significantly 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`
`-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 (such as FMAs). fast: Always (default). on: Only in the presence of the FP_CONTRACT pragma. 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)

Continued on next page

Table 4 – continued from previous page

Option	Description
-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
-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/disable loop vectorization (enabled by default). **Usage** armclang -fvectorize (enable) armclang -fno-vectorize (disable)
-mcpu=<arg>	Select which CPU architecture to optimize for -mcpu=native causes the compiler to auto-detect the CPU architecture from the build computer. **Usage** armclang -mcpu=<arg>

2.5 Workload compilation options

Options that affect the way C language workloads compile.

Table 5: Compiler linker options

Option	Description
-fsimdmath -fno-simdmath	Enable use of vectorized libm library (libsimdmath) to aid vectorization of loops containing calls to libm. **Usage** armclang -fsimdmath Or armclang -fno-simdmath
-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 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	Generate source-level debug information. **Usage** armclang -g

2.7 Warning options

Options that control the behavior of warnings.

Table 7: Compiler warning options

Option	Description
-W<warning> -Wno-<warning>	Enable or disable the specified warning. **Usage** armclang -W<warning>
-Wall	Enable all warnings. **Usage** armclang -Wall
-w	Suppress all warnings. **Usage** armclang -w

2.8 Pre-processor options

Options that control pre-processor behavior.

Table 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 9: Compiler linker options

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

Continued on next page

Table 9 – continued from previous page

Option	Description
`-Xlinker <arg>`	Pass `<arg>` to the linker. **Usage** `armclang -Xlinker <arg>`
`-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. • Should not be used when 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.

`-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 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`

Warning:

- 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

THREE

CODING BEST PRACTICE

Topics about best practice with writing C/C++ code for Arm C/C++ Compiler.

3.1 Coding best practice for auto-vectorization

To encourage the Arm C/C++ Compiler to produce optimal auto-vectorized output, code can be structured and hints can be provided to inform the compiler of program features that it would otherwise not be able to determine. This allows the compiler to produce optimal auto-vectorized output.

3.1.1 Use the restrict keyword if appropriate 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. This guidance allows the compiler to vectorize loops more aggressively, since it becomes possible to prove that loop iterations are independent and can be executed in parallel.

Note: C code may use either the `restrict` or `__restrict__` keywords. C++ code must use only the `__restrict__` keyword.

If these keywords are used erroneously (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.

3.1.2 Use pragmas

The compiler supports pragmas that you can use to explicitly indicate that loop iterations are completely independent from each other.

3.1.3 Use < to construct loops

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

The compiler might also be able to perform more loop optimizations if signed integers are used, because the C standard allows for undefined behavior in the case of signed integer overflow. This is not the case for unsigned integers.

3.1.4 Use the -ffast-math option if it is safe to do so

This can significantly improve the performance of generated code, but it does so at the expense of strict compliance with IEEE and ISO standards for mathematical operations. Ensure that your algorithms are tolerant of potential inaccuracies that could be introduced by the use of this option.

3.2 Using pragmas to control auto-vectorization

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

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

Note: In all the following cases, 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 in order to affect all the loops in the nest.

3.2.1 Encouraging auto-vectorization with pragmas

If SVE auto-vectorization is enabled with `-O2` or above, then by default it 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, you can use the `vectorize` directive to indicate this to the compiler by placing the following line immediately before the loop:

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

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

Note: Use of this pragma does not guarantee auto-vectorization. There might be other reasons why auto-vectorization is not possible or worthwhile for a particular loop.

Ensure that you only use this pragma when it is safe to do so. Using this pragma when there are data dependencies between loop iterations may 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 the programmer knows 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 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] ]++;
  }
}
```

3.2.2 Suppressing auto-vectorization with pragmas

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

You can suppress auto-vectorization on a specific loop by adding `#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 may still be used with a fixed-width predicate (such as gather loads or scatter stores).

`vectorize_style(scaled_width)` 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. This is the default.

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;
  }
}
```

3.2.3 Unrolling and interleaving with pragmas

To enable better use of processor resources, loops can be duplicated to reduce the loop iteration count and increase the instruction-level parallelism (ILP). For scalar loops, the method is called unrolling. For vectorizable loops, interleaving is performed.

3.2.4 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 this example, two is the unrolling factor (UF). To unroll to the internal limit, the following 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_)
```

3.2.5 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 following 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 will not unroll the loop correctly.

3.3 Optimizing C/C++ code with Arm SIMD (NEON)

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 Arm®v7 instructions) and for AArch64. Both can be used to significantly accelerate repetitive operations on the large data sets commonly encountered with High Performance Computing applications.

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

For example, consider the following SIMD instruction:

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

This 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 stored in `V0`. Each lane is added separately. There are no carries between the lanes.

3.3.1 Coding with SIMD

There are a number of different methods you can use 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. This lets you use the data types and operations available in the SIMD implementation, while allowing the compiler to handle instruction scheduling and register allocation. These intrinsics are defined in the language extensions document.

- Write SIMD assembly code.

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

3.3.2 Related information

Further information about NEON is available as follows:

- The Arm NEON Programmer's Guide provides a guide for programmers to effectively use SIMD technology.
- The Arm Developer website provides an overview of NEON technology.
- The Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile provides information about the SIMD instructions.
- The Arm C Language Extensions document provides information about the available SIMD intrinsics.

3.4 Note about building Position Independent Code (PIC) on AArch64

3.4.1 Issue

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

Note:

- This issue does not occur when using the `-fpic` flag with Arm C/C++ Compiler (`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.

3.4.2 Cause

Using the `-fpic` compiler flag 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 C/C++ Compiler on AArch64, or building PIC on x86_64, `-fpic` does not set a limit for the GOT, and this issue does not occur.

3.4.3 Solution

Consider using the `-fPIC` compiler flag 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.

CHAPTER

FOUR

STANDARDS SUPPORT

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

4.1 OpenMP 4.0

The table describes which OpenMP 4.0 features are supported by Arm C/C++ Compiler.

OpenMP 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

The table describes which OpenMP 4.5 features are supported by Arm C/C++ Compiler.

OpenMP 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

FIVE

OPTIMIZATION REMARKS

This short tutorial describes how to enable and use optimization remarks with Arm C/C++ Compiler.

5.1 Using Optimization remarks

This short tutorial describes how to enable and use optimization remarks with Arm C/C++ Compiler.

5.1.1 Overview

Optimization remarks provide you with information about the choices made by the compiler. They can be used to see which code has been inlined or to understand why a loop has not been vectorized.

By default, Arm C/C++ Compiler prints compilation information to stderr. Using optimization remarks, optimization information is printed to the terminal or can be piped to an output file.

5.1.2 Enabling optimization remarks

To enable optimization remarks, pass the following `-Rpass` options to `armclang`:

Flag	Description
`-Rpass=<regexp>`	Request information about what Arm C/C++ Compiler optimized
`-Rpass-analysis=<regexp>`	Request information about what Arm C/C++ Compiler optimized
`-Rpass-missed=<regexp>`	Request information about what Arm C/C++ Compiler optimized

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

Recommended `<regexp>` queries are:

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

where `loop-vectorize` will filter remarks regarding vectorized loops, and `inline` for remarks regarding inlining.

To search for all remarks, use the expression `.*`. However, use this expression with care because a lot of information may be printed depending on the size of your code and the level of optimization performed.

5.1.3 C/C++ example using armclang

To pass the `-Rpass` and `-Rpass-analysis` flags to `armclang`, use:

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

which can give the following example output in the terminal:

```
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) {
       ^
```

5.1.4 Piping optimization remarks to a file

To pipe loop vectorization optimization remarks 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
```

5.1.5 Related information

- More about Arm C/C++ Compiler.

CHAPTER

SIX

VECTOR MATH ROUTINES

Describes how to use the `libsimdmath` library which contains the SIMD implementation of the routines provided by `libm`.

6.1 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>` in the case of C++) can be vectorized by invoking the compiler with the option `-fsimdmath`, together with the usual options that are needed to activate the auto-vectorizer (optimization level `-O2` and above).

6.1.1 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
  }
}
```

6.1.2 How it works

Arm C/C++ Compiler contains `libsimdmath`, a library with SIMD implementations of the routines provided by `libm`, along with a `math.h` file that declares the availability of these SIMD functions to the compiler, using the OpenMP `#pragma omp declare simd` directive.

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 `libsimdmath` function that takes a vector of double precision arguments, and returns a result vector of doubles.

The `libsimdmath` library is built using code based on SLEEF, an open source math library available from the SLEEF website.

A future release of Arm C/C++ Compiler will describe a workflow to allow users to declare and link against their own vectorized routines, allowing them to be used in auto-vectorized code.

6.1.3 Limitations

This is an experimental feature which can lead to performance degradations in some cases. We encourage 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

6.1.4 Related information

- Get the SLEEF library from the SLEEF website.
- Vector function ABI specification for AArch64.
- More about Arm C/C++ Compiler.
- Help and tutorials.