



# Helium Programmers Guide Coding for Helium

Version 1.1

## Non-Confidential

Copyright © 2020 Arm Limited (or its affiliates).  
All rights reserved.

## Issue 02

102095\_0101\_02\_en



# Helium Programmers Guide Coding for Helium

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

## Release information

### Document history

Issue	Date	Confidentiality	Change
0100-01	17 April 2020	Non-Confidential	Initial release
0101-02	1 June 2020	Non-Confidential	Updated images

## Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

## Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

## Product Status

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

## Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on <https://support.developer.arm.com>

To provide feedback on the document, fill the following survey: <https://developer.arm.com/documentation-feedback-survey>.

## Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email [terms@arm.com](mailto:terms@arm.com).

# Contents

1. Overview.....	6
2. Before you begin.....	7
3. Options for writing Helium-enabled code.....	8
4. Enabling Helium.....	11
5. Helium-enhanced libraries.....	13
6. Auto-vectorization and Helium.....	16
7. Helium intrinsics.....	21
8. Mixing C/C++ and Helium assembly code.....	26
9. Related information.....	33
10. Next steps.....	34

# 1. Overview

This guide provides information and examples for software programmers who want to use Arm Helium technology. We will discuss the benefits and drawbacks of the different approaches available, and examine real-world code examples to help you understand the key issues.

Arm Helium technology is the M-Profile Vector Extension (MVE) for the Arm Cortex-M processor series. Helium is an extension of the Armv8.1-M architecture and delivers a significant performance uplift for machine learning (ML) and digital signal processing (DSP) applications. For introductory information about Helium, please see the [Introduction to Helium](#) guide.

## 2. Before you begin

This guide forms part of the Helium Programmer's Guide. If you are not familiar with Helium, you should start by reading the [Introduction to Helium](#) guide.

The sections of this guide contain code examples. These code examples are available to [download as a ZIP file](#).

The examples in this guide use the [Arm Compiler 6 toolchain](#), designed for embedded application development running on bare-metal devices. If you do not already have access to Arm Compiler 6, it is included in the 30-day free trial of Arm Development Studio Gold Edition.

### 3. Options for writing Helium-enabled code

Programming in any high-level language is a tradeoff between the ease of writing code, and the amount of control that you have over the low-level instructions that are output by the compiler. This is true when targeting Helium-enabled hardware. The goal is to ensure that wherever your code contains vectorization opportunities where operations could be performed in parallel, Helium instructions are used.

At one end of the spectrum, you could write all your code in standard C/C++ and leave the implementation decisions to the compiler. If you are using an auto-vectorizing compiler, and your code is straightforward, this can produce excellent results. The compiler generates Helium instructions for all the vectorizable portions of your code.

The benefit of this approach is that it requires very little effort from the programmer, except for writing standard C/C++ code.

The drawback of this approach is that, if the compiler does not do what you want, for whatever reason, you might not have enough control to change that situation. For example, if your code is complex, the compiler might miss a vectorization opportunity and fail to use Helium. Modifying your code to follow best practices might be enough to help the compiler identify the vectorization opportunity, but you cannot be sure.

At the other end of the spectrum, you could write all your Helium code by hand in assembly. This gives you full control over the instructions used, but at the cost of vastly increased programmer effort.

The different options available for writing Helium-enabled code are:

- Helium-enabled libraries
- Auto-vectorization
- Helium intrinsics
- Assembly code

#### Helium-enabled libraries

Libraries that support Helium provide one of the easiest ways to take advantage of Helium.

Libraries provide a suite of functions that you can use in your own code. When you compile for a Helium-enabled target, a library variant using Helium instructions is selected. When you compile for a target that does not support Helium, a library variant using standard Arm instructions is selected. This means that the same source code can easily be compiled for both Helium-enabled targets and non-Helium-enabled targets.

Examples of Helium-enabled libraries include:

- [CMSIS-DSP](#) - A suite of common signal processing functions for use on Cortex-M processor-based devices.



- **CMSIS-NN** - A collection of efficient neural network kernels that are developed to maximize the performance, and minimize the memory footprint, of neural networks on Cortex-M processor cores.

Libraries are easy to incorporate into your code, and the implementations of the functions have already been optimized. For example, CMSIS-DSP has been designed to provide many of the functions that you would need to write signal-processing code like audio filters or Fast Fourier Transform (FFT).

The disadvantage of libraries is that you only have access to the functions that the library designer has provided.

## Auto-vectorization

Auto-vectorization features in your compiler can automatically optimize your code to take advantage of Helium.

Auto-vectorization means allowing the compiler to automatically identify the areas of your code that would benefit from Single Instruction Multiple Data (SIMD) optimizations.

The benefit of using auto-vectorization is that the programmer leaves everything to the compiler.

The disadvantage of auto-vectorization is that, if the compiler does not do what you want, you might not have enough control to change that situation. For example, the compiler might fail to identify that a particular part of your code is vectorizable. You can use coding best practices to help the compiler identify that code is vectorizable, but they might not be enough to guide the compiler in the right direction. In these situations, you might need to use other options, for example intrinsics or inline assembly, to ensure that Helium instructions are used.

## Helium intrinsics

Helium intrinsics are function calls that the compiler replaces with appropriate Helium instructions. Using Helium intrinsics gives you direct, low-level access to the exact Helium instructions that you want, all from C/C++ code.

The benefit of using intrinsics is that they provide almost as much control as writing assembly language, but leave details like register allocation to the compiler, so that developers can focus on the algorithms.

The disadvantage of using Helium intrinsics is that programming with intrinsics can be more complex than writing standard C/C++ code, and requires the programmer to learn about the available Helium intrinsics.

## Assembly code

For very high performance, hand-coded Helium assembly code is an alternative approach for experienced programmers.

You can use pure assembly code modules (.s files) in your code, or you can use inline assembly code to embed assembler instructions in your C and C++ code.

The benefit of using assembly code is that it provides absolute control over the Helium instructions that are used.

The disadvantage of using assembly code is that writing assembly code can be a very complex process that most people would rather not have to do. Optimizing hand-written assembly code often requires detailed knowledge of the target hardware pipeline, especially for in-order Cortex-M processors. You might need to write and maintain different code variants for different targets to achieve optimal performance.

## 4. Enabling Helium

Helium is an optional extension to the Armv8.1-M architecture. This means that Helium may or may not be present on your target.

Because of this, you should check whether Helium is available on your target before running Helium code. This applies whether you are using a software model or a hardware target.

### Enabling Helium on the ARM\_AEMv8M Fixed Virtual Platform

FVP models provide a number of different parameters to configure the optional features of the model.

See the [Fast Models Reference Manual](#) for a complete list of all ARM\_AEMv8M parameters.

To enable Helium on the ARM\_AEMv8M FVP, set the following parameters:

- `cpu0.enable_helium_extension=1`
- `cpu0.vfp-present=1`
- `cpu0.vfp-enable_at_reset=1`

### Checking if Helium is present for hardware targets

The Media and VFP Feature Register 1 (`MVFR1`), describes the features that are provided by the floating-point extension. In particular, the `MVFR1.MVE` bitfields, bits [11:8], indicate support for the M-profile vector extension.

The possible values of this field are:

- `0b0000` indicates that Helium instructions are not available.
- `0b0001` indicates that Helium integer instructions are available, but Helium floating-point instructions are not available.
- `0b0010` indicates that Helium integer and floating-point instructions are available.

For more information, see the [Armv8-M Architecture Reference Manual](#).

The `__ARM_FEATURE_MVE` macro provides another mechanism for checking whether Helium is present.

### Enabling Helium for hardware targets

For hardware targets, access control registers specify whether Helium instructions are available from privileged and unprivileged code.

- The Non-secure Access Control Register (`NSACR`) specifies the Non-secure access permissions for Helium in bitfields `CP10` and `CP11`. The possible values of these fields are:
  - `0` - Non-secure accesses to the Floating-point Extension or MVE, unless otherwise specified, generate a NOCP UsageFault.
  - `1` - Non-secure access to the Floating-point Extension or MVE is permitted.

- The Coprocessor Access Control Register (CPACR) specifies the access privileges for Helium in bit field CP10. The possible values of this field are:
  - 0b00 - All accesses to the FP Extension and MVE result in NOCP UsageFault.
  - 0b01 - Unprivileged accesses to the FP Extension and MVE result in NOCP UsageFault.
  - 0b11 - Full access to the FP Extension and MVE. For more information, see the [Armv8-M Architecture Reference Manual](#).

## 5. Helium-enhanced libraries

Libraries provide a suite of functions you can use in your own code. Helium-enhanced libraries provide implementations of those functions that use Helium instructions.

When you compile for a Helium-enabled target, a library variant using Helium instructions is selected. When you compile for a target that does not support Helium, a library variant using standard Arm instructions is selected. This means that you can easily compile the same source code for both Helium and non-Helium enabled targets.

Examples of Helium-enabled libraries include:

- [CMSIS-DSP](#) - A suite of common signal processing functions for use on Cortex-M processor-based devices
- [CMSIS-NN](#) - A collection of efficient neural network kernels that are developed to maximize the performance, and minimize the memory footprint, of neural networks on Cortex-M processor cores

In this section of the guide, we examine the CMSIS-DSP library to see how to use libraries to write Helium-enabled code.

### Getting the CMSIS-DSP library

The [CMSIS-DSP](#) library provides functions that are specifically designed for signal processing. The library provides over sixty common signal processing and mathematical functions for various data types.

CMSIS is the Arm Cortex Microcontroller Software Interface Standard. CMSIS provides a vendor-independent hardware abstraction layer for microcontrollers that are based on Arm Cortex processors. CMSIS defines generic tool interfaces and enables consistent device support. Its software interfaces simplify software reuse, reduce the learning curve for microcontroller developers, and improve time to market for new devices.

CMSIS is integrated into IDEs like Keil MDK and Arm Development Studio, but can also be used with a standalone compiler.

To use CMSIS:

- For Keil MDK, use the [MDK Pack Installer](#) to install the ARM::CMSIS pack.
- For Arm Development Studio, use the [Pack Manager](#) to install the Generic > ARM.CMSIS pack.
- For a standalone compiler, [download CMSIS from git hub](#) and follow the [instructions for using CMSIS with Arm Compiler 6](#).

### Writing code using the CMSIS-DSP library

The CMSIS-DSP pack includes [various examples](#). For example, the [variance example](#) demonstrates the use of basic math functions to calculate the variance of an input sequence.

Let's examine the source code of this example to look at some key features:

```
#include "arm_math.h"
```

The preceding code shows the CMSIS-DSP header file that declares the basic math functions used by the example. This code must be included to use the CMSIS-DSP functions.

The following code shows how the variance example uses CMSIS-DSP library functions to implement an algorithm that calculates the statistical variance of an input stream. The CMSIS-DSP functions are `arm_fill_f32()`, `arm_dot_prod_f32()`, and `arm_mult_f32()`:

```
.
.
.
/* Calculation of mean value of input */
/* x' = 1/blockSize * (x(0)* 1 + x(1) * 1 + ... + x(n-1) * 1) */
/* Fill wire1 buffer with 1.0 value */
arm_fill_f32(1.0, wire1, blockSize);

/* Calculate the dot product of wire1 and wire2 */
/* (x(0)* 1 + x(1) * 1 + ...+ x(n-1) * 1) */
arm_dot_prod_f32(testInput_f32, wire1, blockSize, &mean);

/* Calculation of 1/blockSize */
oneByBlockSize = 1.0 / (blockSize);

/* 1/blockSize * (x(0)* 1 + x(1) * 1 + ... + x(n-1) * 1) */
arm_mult_f32(&mean, &oneByBlockSize, &mean, 1);

.
.
.
```

The CMSIS-DSP functions used in this example are:

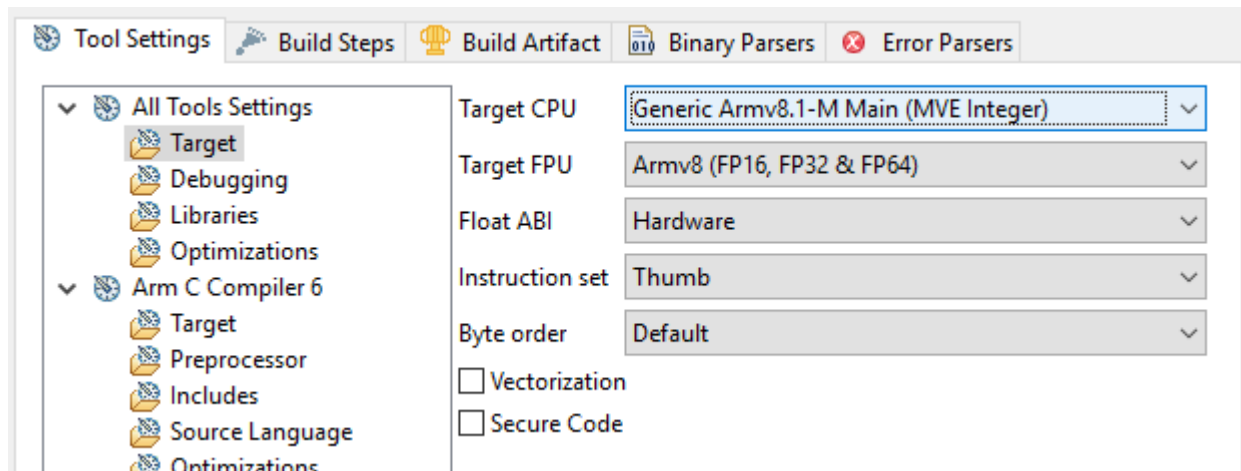
- `arm_fill_f32()` fills a constant value into a floating-point vector.
- `arm_dot_prod_f32()` calculates the dot product of floating-point vectors.
- `arm_mult_f32()` performs floating-point vector multiplication.

All available CMSIS-DSP functions are described in the [CMSIS DSP Software Library Reference](#).

## Compiling CMSIS-DSP code for Helium

When compiling for a Helium-enabled target, the compiler will automatically select the CMSIS-DSP variant that uses Helium instructions.

For example, in Arm Development Studio select Generic Armv8.1-M Main (MVE Integer) to target any Helium-enabled Armv8-M platform, as shown in the following screenshot:

**Figure 5-1: ds select helium target**

When compiling with a standalone compiler, you must ensure that the CMSIS header files are on the include path, and the CMSIS libraries are on the library path.

For example, to target the architecture, use this command:

```
armclang -target arm-arm-none-eabi -march=armv8.1-m.main+mve.fp+fp.dp
-I <cmsis_dir>/DSP/Include/ -L <cmsis_dir>/DSP/Lib/ ...
```

To target the Cortex-M55, use this command:

```
armclang -target arm-arm-none-eabi -mcpu=cortex-m55 -I <cmsis_dir>/DSP/Include/
-L <cmsis_dir>/DSP/Lib/ ...
```

## 6. Auto-vectorization and Helium

There are many different ways to write code that takes advantage of Helium technology. Writing hand-optimized assembly kernels, or C code containing Helium intrinsics, provides a high level of control over the Helium code in your software. However, these methods can result in significant lack of portability and engineering complexity costs.

Often a high-quality compiler can generate code which is just as good, but requires significantly less design time. Auto-vectorization is the process of allowing the compiler to automatically identify opportunities in your code to use Helium instructions.

Auto-vectorization includes the following compilation techniques:

- Loop vectorization – Unrolling loops to reduce the number of iterations, while performing more operations in each iteration.
- Superword-Level Parallelism (SLP) vectorization – Bundling scalar operations together to use full width Helium instructions.

Auto-vectorizing compilers for Cortex-M processors include Arm Compiler 6 and LLVM-clang.

The benefits of relying on compiler auto-vectorization include:

- Programs implemented in high-level languages are portable, if there are no architecture-specific code elements like inline assembly or intrinsics.
- Modern compilers can perform advanced optimizations automatically.
- Targeting a given micro-architecture can be as easy as setting a single compiler option. Optimizing an assembly program requires deep knowledge of the target hardware.

However, auto-vectorization might not be the right choice in all situations:

- While source code can be architecture agnostic, it may have to be compiler specific to get the best code generation.
- Small changes in a high-level language or the compiler options can result in significant and unpredictable changes in generated code.

Using the compiler to generate Helium instructions is appropriate for most projects. Other methods for exploiting Helium are necessary only when the generated code does not deliver the necessary performance, or when particular hardware features are not supported by high-level languages.

### Compiling for Helium with Arm Compiler 6

To enable automatic vectorization, you must specify appropriate compiler options.

These compiler options must do the following:

- Target a processor that has Helium capabilities
- Specify an optimization level that includes auto-vectorization



## Specifying a Helium-capable target

If you want to run code on one processor, you can target that specific processor with the `-mcpu` option. Performance is optimized for the micro-architectural specifics of that processor. However, code is only guaranteed to run on that processor.

Alternatively, if you want your code to run on a range of processors, you can target an architecture with the `-march` option. Generated code runs on any processor implementation of that target architecture, but performance might be impacted.

In both cases, you can use one of the following feature modifiers to enable Helium:

- `+mve` enables MVE instructions for integer operations.
- `+mve.fp` enables MVE instructions for integer and single-precision floating-point operations.
- `+mve.fp+fp.dp` enables MVE instructions for integer, single-precision, and double-precision floating-point operations.

The Helium extension is always enabled on the Cortex-M55, so there is no need to use a feature modifier. Targeting the processor is sufficient to generate Helium code, as in the following command:

```
armclang --target arm-arm-none-eabi -mcpu=cortex-m55 ...
```

To target Helium for any Helium-enabled Armv8-M platform, you must specify a feature modifier, as in the following command:

```
armclang -target arm-arm-none-eabi -march=armv8.1-m.main+mve.fp+fp.dp ...
```

## Specifying an auto-vectorizing optimization level

Arm Compiler 6 provides a wide range of optimization levels, selected with the `-O` option.

The following table defines the available optimization levels:

Option	Description	Auto-vectorization
<code>-O0</code>	Minimum optimization	Never
<code>-O1</code>	Restricted optimization	Disabled by default.
<code>-O2</code>	High optimization	Enabled by default.
<code>-O3</code>	Very high optimization	Enabled by default.
<code>-Os</code>	Reduce code size, balancing code size against code speed	Enabled by default.
<code>-Oz</code>	Smallest possible code size	Enabled by default.
<code>-Ofast</code>	Optimize for high performance beyond <code>-O3</code>	Enabled by default.
<code>-Omax</code>	Optimize for high performance beyond <code>-Ofast</code>	Enabled by default.

See [Selecting optimization options, in the Arm Compiler User Guide](#) and `-O`, in the [Arm Compiler armclang Reference Guide](#) for more details about these options.

Auto-vectorization is enabled by default at optimization level `-O2` and higher. The `-fno-vectorize` option lets you disable auto-vectorization.

At optimization level `-O1`, auto-vectorization is disabled by default. The `-fvectorize` option lets you enable auto-vectorization.

At optimization level `-O0`, auto-vectorization is always disabled. If you specify the `-fvectorize` option, the compiler ignores it.

To enable auto-vectorization, do one of the following:

- Select an optimization level of `-O2` or higher.
- Select an optimization level of `-O1` and specify `-fvectorize`.

### Helium auto-vectorization example

The following Helium auto-vectorization example shows how an auto-vectorizing compiler identifies optimization opportunities in source code, and uses Helium instructions to maximize performance.

The example function clips floating point values if they fall outside of a specified range. The function takes the following parameters:

- `*pSrc`, a pointer to an array input data
- `*pDst`, a pointer to an array where output data will be stored
- `low`, the lower bound of the clipping range. Input data values lower than `low` are replaced with `low`.
- `high`, the upper bound of the clipping range. Input data values higher than `high` are replaced with `high`.
- `numSamples`, the number of data values in the input array (and therefore also the output array once the function has finished).

The example function is implemented as follows:

```
#include "arm_math.h"
void arm_clip_f32(float32_t * pSrc, float32_t * pDst, float32_t low, float32_t high,
uint32_t N) {
    for (uint32_t i = 0; i < N; i++) {
        float32_t x = pSrc[i];
        if (x > high)
            x = high;
        if (x < low)
            x = low;
        pDst[i] = x;
    }
}
```

```
}
```

Compile this code with Arm Compiler 6 as follows:

```
armclang -target arm-arm-none-eabi -march=armv8.1-m.main+mve.fp+fp.dp -Ofast
        -S arm_clip_f32.c
```

In this example, using the `-s` option means that the compiler outputs the disassembly of the compiled code to the file `arm_clip_f32.s`.

Examining the `arm_clip_f32.s` file shows the instructions the compiler has generated:

```
...      dls          lr, lr
.LBB0_13:                                @ =>This Inner Loop Header: Depth=1
        vldrw.u32     q3, [r4], #16
        vminnm.f32   q3, q3, q1
        vmaxnm.f32   q3, q3, q2
        vstrb.8      q3, [r3], #16
        le           lr, .LBB0_13 ...
```

The Helium instruction `VLDRW.U32` loads our data into vector lanes. In this example the data values are 32-bit, so each vector is loaded with four data values at a time.

The `VCMPE.F32` Helium instructions then compare those vector lanes concurrently against the upper and lower clipping values.

Helium predication instructions such as `VBST` selectively perform the clipping operation only on data where the comparison reveals that clipping is needed.

## Coding best practice for auto-vectorization

As an implementation becomes more complicated, the likelihood that the compiler can auto-vectorize the code decreases.

For example, loops with the following characteristics are particularly difficult, or impossible, to vectorize:

- Loops with interdependencies between different loop iterations
- Loops with break clauses
- Loops with complex conditions

Arm recommends modifying your source code implementation to eliminate these situations where possible.

For example, a necessary condition for auto-vectorization is that the number of iterations in the loop size must be known at the start of the loop. Break conditions mean that the loop size may not be knowable at the start of the loop, which will prevent auto-vectorization. If it is not possible to completely avoid a break condition, it may be worthwhile breaking up the loops into multiple vectorizable and non-vectorizable parts.

A full discussion of the compiler directives that are used to control vectorization of loops can be found in the LLVM-Clang documentation, but the two most important are:

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

These pragmas are hints to the compiler to perform Superword Level Parallelism (SLP) and loop vectorization respectively. They are [\[COMMUNITY\] features of Arm Compiler](#).

More detailed guides covering auto-vectorization are available for the Arm C/C++ Compiler Linux user-space compiler, although many of the points apply across LLVM-Clang variants:

- [Arm C/C++ Compiler: Coding best practice for auto-vectorization](#)
- [Arm C/C++ Compiler: Using pragmas to control auto-vectorization](#)

## 7. Helium intrinsics

Intrinsics are functions whose precise implementation is known to a compiler. The Helium intrinsics are a set of C and C++ functions that are defined in the header file `arm_mve.h`. The Arm compilers and GCC support these intrinsics.

Helium intrinsics provide direct access to Helium instructions from C and C++ code without having to write assembly code by hand. The intrinsics map to short assembly kernels which are inlined into the calling code. Also, the compiler handles register allocation and pipeline optimization. This means that many difficulties that are faced by the assembly programmer are avoided.

See the [Arm MVE Intrinsics Reference Architecture specification](#) (also available as [interactive HTML](#)) for a list of all the Helium intrinsics. This specification forms part of the [Arm C Language Extensions \(ACLE\)](#).

Using the Helium intrinsics has several benefits:

- **Powerful:** Intrinsics give the programmer direct access to the Helium instruction set without the need for hand-written assembly code.
- **Portable:** Hand-written Helium assembly instructions might need to be rewritten for different target processors. C and C++ code containing Helium intrinsics can be compiled for a new target with minimal or no code changes.
- **Flexible:** The programmer can exploit Helium when needed, or use C/C++ otherwise, while avoiding many low-level engineering concerns.

However, intrinsics might not be the right choice in all situations:

- It is more difficult to use Helium intrinsics than to import a library or rely on a compiler.
- Hand-optimized assembly code might offer the greatest scope for performance improvement even if it is more difficult to write.

### Helium header file

The header file `arm_mve.h` defines the Helium intrinsics. You must include this header file in every source file that uses Helium intrinsics.

You should test the `__ARM_FEATURE_MVE` macro before including the header. The `__ARM_FEATURE_MVE` macro is a 2-bit bitmap indicating M-profile Vector Extension (MVE) support:

- Bit 0 indicates whether Helium integer instructions are available.
- Bit 1 indicates whether Helium floating-point instructions are available.

The valid values of `__ARM_FEATURE_MVE` are therefore:

- 0 indicates that Helium is not available.
- 1 indicates that only the Helium integer intrinsics are available.
- 3 indicates that both the Helium integer and floating-point intrinsics are available.

The `__ARM_FEATURE_MVE` macro should be tested to check that Helium is enabled on the target platform before including the header:

```
#if ( __ARM_FEATURE_MVE & 3) == 3
#include <arm_mve.h>
    // MVE integer and floating point intrinsics are now available to use. //
#elif __ARM_FEATURE_MVE & 1
#include <arm_mve.h>
    // MVE integer intrinsics are now available to use. //
#endif
```

## Namespaces

By default, Helium intrinsics occupy both the user namespace and the `__arm_` namespace.

That is, both these lines of code are equivalent:

```
vecDst = vmulq_f32(vecA, vecB);
vecDst = __arm_vmulq_f32(vecA, vecB);
```

Defining the macro `__ARM_MVE_PRESERVE_USER_NAMESPACE` hides the definition of the user namespace variants:

```
#define __ARM_MVE_PRESERVE_USER_NAMESPACE
vecDst = vmulq_f32(vecA, vecB);           //Invalid. User namespace variants are
hidden.
vecDst = __arm_vmulq_f32(vecA, vecB);     // Valid.
```

## Compiling code containing Helium intrinsics with Arm Compiler 6

To compile code containing Helium intrinsics, you must do the following:

- Include the Helium intrinsics header file `arm_mve.h` in your code
- [Specify compiler options that identify a target with Helium capabilities.](#)

The preceding steps are the minimum that you must do to enable Helium intrinsics to be compiled into Helium instructions. However, you might also want to have the compiler perform auto-vectorization. This will allow you to identify further opportunities in your code to improve performance with Helium. In this case, [specify an appropriate optimization level to enable auto-vectorization.](#)

To target Helium for any Helium-enabled Armv8-M platform:

```
armclang -target arm-arm-none-eabi -march=armv8.1-m.main+mve.fp+fp.dp ...
```

## Helium intrinsics example

This example shows how you can use Helium intrinsics to perform vector multiplication.

Vector multiplication multiplies the value of the elements in the first source vector by the respective elements in the second source vector, then writes the result to a destination vector register. That is:

$$[A_i, A_j, B_k, \dots] \times [B_i, B_j, B_k, \dots] = [(A_i \times B_i), (A_j \times B_j), (A_k \times B_k), \dots]$$

The `main()` function does the following:

- Creates two source data arrays, each containing eight floating-point numbers
- Calls the `my_mult_f32_intr()` function, passing the following arguments:
  - Pointers to the two input arrays to use as the data source
  - The block size in memory of the input arrays
  - A pointer to the `A_src` array, to use as the result destination. The result will therefore overwrite the original data.

The `my_mult_f32_intr()` function does the following:

- Uses the block size to calculate how many vector loop iterations are required. Because we are dealing with 32-bit floating-point values, and the \*\* Helium registers are 128 bits wide, we can operate on four data values in each iteration.
- Loads data from the input arrays into the Helium vector registers, four values at a time
- Performs the vector multiplication on the input vectors
- Stores the result vector into the destination array
- Advances the array pointers by the size of four data elements
- Decrements the loop counter, and loops around until all loop iterations have finished

The `my_mult_f32_intr()` function is implemented as follows:

```
#include <stdio.h>
#include <arm_mve.h>

void my_mult_f32_intr(
    float32_t * pSrcA, float32_t * pSrcB,
    float32_t * pDst, uint32_t blockSize) {

    // Calculate memory block size for 4 x lanes of float32_t data
    const int blkSize_F32 = 4 * sizeof(float32_t);

    // Calculate how many loop iterations are required:
    //   size of array / size of 4 data items
    int blkCnt = blockSize / blkSize_F32;

    // Create source and destination vectors, configured for 4 lanes of float32_t data
    float32x4_t vecA, vecB, vecDst;

    // Main loop
    while (blkCnt > 0U) {
        // Load source vectors with data from the input arrays
        vecA = vldrwq_f32(pSrcA);
        vecB = vldrwq_f32(pSrcB);

        // Perform vector multiplication
```

```

    vecDst = vmulq_f32(vecA, vecB);

    // Store the result vector into the destination array
    vstrwq_f32(pDst, vecDst);

    // Decrement the loop count
    blkCnt--;

    // Advance source and destination pointer addresses by the size of 4 data
    elements
    pSrcA += blkSize_F32;
    pSrcB += blkSize_F32;
    pDst += blkSize_F32;
}

int main() {
    // Setup data in input arrays
    float32_t A_src[] = {1.1, 7.9, 8.2, 2.1, 5.3, 2.2, 3.1, 6.9};
    float32_t B_src[] = {7.2, 2.7, 9.9, 8.2, 1.3, 1.1, 6.9, 2.4};

    // Call the multiplication function
    my_mult_f32_intr(&A_src[0], &B_src[0], &A_src[0], sizeof(A_src));

    return 0;
}

```

The following table shows some additional information about the intrinsics that are used:

Intrinsic	Description
vldrwq_f32	Loads consecutive elements from memory into a destination vector register.
vmulq_f32	Multiplies the value of the elements in the first source vector register by the respective elements in the second source vector register. The result is then written to the destination vector register.
vstrwq_f32	Stores consecutive elements to memory from a vector register.

You can compile this code with Arm Compiler 6 as follows:

```

armclang -target arm-arm-none-eabi -march=armv8.1-m.main+mve.fp+fp.dp
         -Ofast -S my_mult_f32_intr.c

```

In this example, using the `-s` option means that the compiler outputs the disassembly of the compiled code to the file `my_mult_f32_intr.s`.

Examining the `my_mult_f32_intr.s` file shows the instructions that the compiler has generated:

```

...
.LBB0_1:                                @ =>This Inner Loop Header: Depth=1
    vldrw.u32    q0, [r1]
    vldrw.u32    q1, [r0]
    adds        r1, #64
    adds        r0, #64
    vmul.f32     q0, q1, q0
    vstrw.32     q0, [r2]
    adds        r2, #64
    le          lr, .LBB0_1
    ...

```

Here we can see that:



- The `vldr_wq_f32` intrinsics compile to `vldr_w.u32` instructions.
- The `vmul_q_f32` intrinsic compiles to a `vmul.f32` instruction.
- The `vstr_wq_f32` intrinsic compiles to a `vstr_w.32` instruction.

## 8. Mixing C/C++ and Helium assembly code

Arm Compiler 6 provides an inline assembler that enables you to incorporate assembly code directly in your C or C++ source code. The benefit of using inline assembly rather than writing pure assembly code is that you can read and write C variables directly.

### Introduction to inline assembly

The `__asm` keyword provides the mechanism to incorporate inline GCC syntax assembly code into a function.

For example, here is a simple example that uses a single inline `ADD` assembly instruction:

```
#include <stdio.h>

int add(int i, int j)
{
    int res = 0;
    __asm (
        "ADD %[result], %[input_i], %[input_j]"
        : [result] "=r" (res)
        : [input_i] "r" (i), [input_j] "r" (j)
    );
    return res;
}

int main(void)
{
    int a = 1;
    int b = 2;
    int c = 0;

    c = add(a,b);

    printf("Result of %d + %d = %d\n", a, b, c);
}
```

The general form of the `__asm` inline assembly statement is:

```
__asm(
    code
    [: output_operand_list
    [: input_operand_list
    [: clobbered_register_list]]
);
```

Where:

- `code` is the assembly code.

In our example, this is:

```
"ADD %[result], %[input_i], %[input_j]"
```

This single instruction adds two inputs, represented by the symbolic names `input_i` and `input_j`, and assigns the sum to the symbolic name `result`.

- `output_operand_list` maps output symbolic names to C variable names.

In our example, there is just one output:

```
: [result] "=r" (res)
```

This indicates that the symbolic name `result` maps to the C variable `res`. The `"=r"` constraint indicates that value resides in a register, and that any previous value in that register is overwritten.

- `input_operand_list` maps input symbolic names to C variable names.

In our example, there are two inputs:

```
: [input_i] "r" (i), [input_j] "r" (j)
```

This indicates that the symbolic name `input_i` maps to the C variable `i`, and `input_j` maps to `j`.

- `clobbered_register_list` is a comma-separated list of register names. These are registers that the assembly code potentially modifies, but for which the final value is not important. Including a register in the clobber list prevents the compiler from using that register for other purposes in the code.

In our example, there are no clobbered registers, because only the previously declared input and result registers are used.

Constraints and modifiers let you tell the compiler how values will be used in the assembly code. In our example, we used the `"=r"` constraint to identify that the output register is overridden. There are many other constraints that let you identify more complex requirements. For example, you can use the `%Q` and `%R` constraint modifiers to access the lower and higher halves of a 64-bit register pair.

The [Arm Compiler Reference Guide: armclang Inline Assembler](#) provides more information about inline assembly, constraints, and modifiers.

For more information about GCC and the format used by the `__asm` keyword, see the [Gnu documentation](#).

## Complex vector dot product example

Helium is ideally suited for the types of operations that are commonly performed by DSP and ML applications.

This example shows how you can use inline assembly and Helium instructions to calculate the vector dot product of two arrays of complex numbers.

[Complex numbers](#) have two parts: real and imaginary, expressed as:

```
a + bi
```

Each complex number is therefore represented as a pair of numbers, *a* and *b*.

The example uses the Q31 fixed-point numbers format to represent the data. In the Q31 format, 1 bit is used to represent the sign (0 for positive, 1 for negative) and the remaining 31 bits represent the fractional data. The Q31 format can therefore express numbers in the range -1 to almost 1.

For example:

Q31 value (decimal)	Q31 value (hex)	Floating-point value
0	0x0000 0000	0
1	0x0000 0001	0.0000000004656613
984267707	0x3AAA BBBB	0.4583353674970567
2147483647	0x7FFF FFFF	0.999999995343387
-2,147,483,648	0x8000 0000	-1
-1,288,490,189	0xB333 3333	-0.6000000000931323
-1	0xFFFF FFFF	-0.0000000004656613

For two complex numbers:

```
a + bi
c + di
```

The vector dot product is calculated as:

```
((a x c) - (b x d)) + ((a x d) + (b x c))i
~~~~~                ~~~~~
      ^                ^
      |                |
Real part          Imaginary part
```

To calculate the dot product for vectors of complex numbers, the individual dot products for each input pair are summed.

That is, the underlying algorithm is:

```
realResult = 0;
imagResult = 0;
for (n = 0; n < numSamples; n++) {
    realResult += pSrcA[(2*n)+0] * pSrcB[(2*n)+0] - pSrcA[(2*n)+1] * pSrcB[(2*n)+1];
    imagResult += pSrcA[(2*n)+0] * pSrcB[(2*n)+1] + pSrcA[(2*n)+1] * pSrcB[(2*n)+0];
}
```

Where:

- `realResult` is the real component of the summed dot products.
- `imagResult` is the imaginary component of the summed dot products.
- `pSrcA` and `pSrcB` are pointers to the two input arrays. Each input array contains complex numbers with the real and imaginary parts interleaved. \* That is, the array {1, 2, 3, 4, 5, 6} represents the three complex numbers 1 + 2i, 3 + 4i, 5 + 6i.

- `numSamples` is the number of complex number pairs in each input array.

We can implement this algorithm using inline assembly code using Helium instructions as follows:

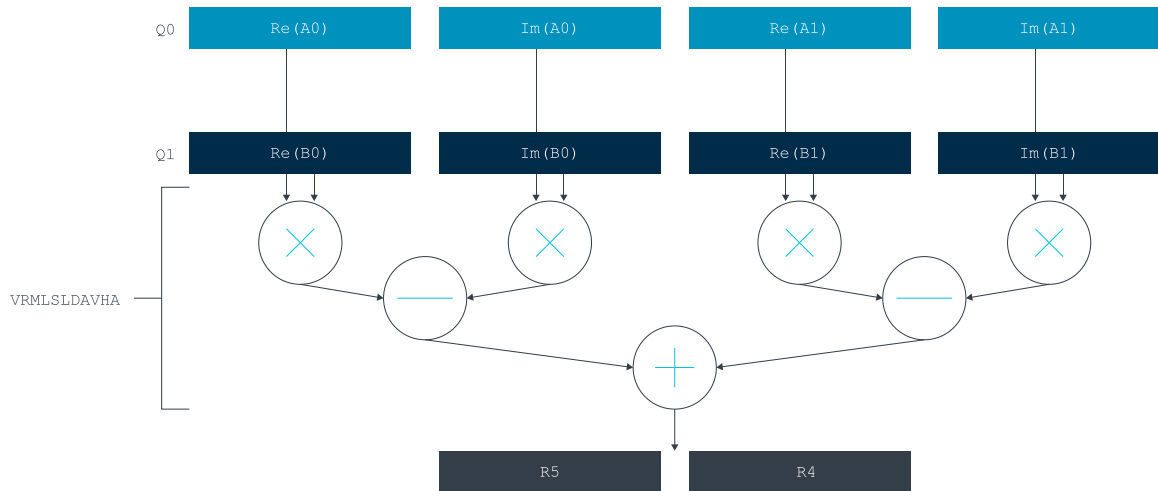
```
void my_cmplx_dot_prod_q31(
    q31_t * pSrcA,
    q31_t * pSrcB,
    uint32_t numSamples,
    q63_t * realResult,
    q63_t * imagResult)
{
    __asm volatile (
        "      clrm          {r4-r7}          \n"
        "      wlstp.32      lr, %[cnt], 1f    \n"
        "2:                  \n"
        "      vldrw.32      q0, [%pA], 16      \n"
        "      vldrw.32      q1, [%pB], 16      \n"
        "      vrmlsldavha.s32 r4, r5, q0, q1    \n"
        "      vrmlaldavhax.s32 r6, r7, q0, q1   \n"
        "      letp          lr, 2b             \n"
        "1:                  \n"
        "      asrl          r4, r5, #6         \n"
        "      asrl          r6, r7, #6         \n"
        "      strd          r4, r5, [%realResult] \n"
        "      strd          r6, r7, [%imagResult] \n"
        : [pA] "+r"(pSrcA), [pB] "+r"(pSrcB)
        : [cnt] "r"(numSamples * 2), [realResult] "r"(realResult),
          [imagResult] "r"(imagResult)
        : "r4", "r5", "r6", "r7", "lr", "memory");
    }
```

Key features of this code include:

- The `WLSTP` (While Loop Start with Tail Predication) and `LETP` (Loop End) instructions form the main loop. This loop iterates over all elements of the input arrays, decrementing `numSamples` by one on each loop and continuing until it reaches zero.
- The `VLDWR` (Vector Load Register) instruction loads the next two complex numbers from each array into Helium registers Q0 and Q1.
- The real component of the dot product is calculated by the `VRMLSLDAVHA` (Vector Rounding Multiply Subtract Long Dual Accumulate Across Vector Returning High 64 bits) instruction. This instruction multiplies corresponding elements from the vectors in the registers Q0 and Q1. The results of the pairs of multiply instructions are subtracted from each other. Finally, the scalar result is then added to the running total that is held in the two registers R5 (high 32 bits) and R4 (low 32 bits). This implements the following part of the algorithm:

```
realResult += pSrcA[(2*n)+0] * pSrcB[(2*n)+0] - pSrcA[(2*n)+1] * pSrcB[(2*n)+1];
```

The following diagram illustrates this calculation:

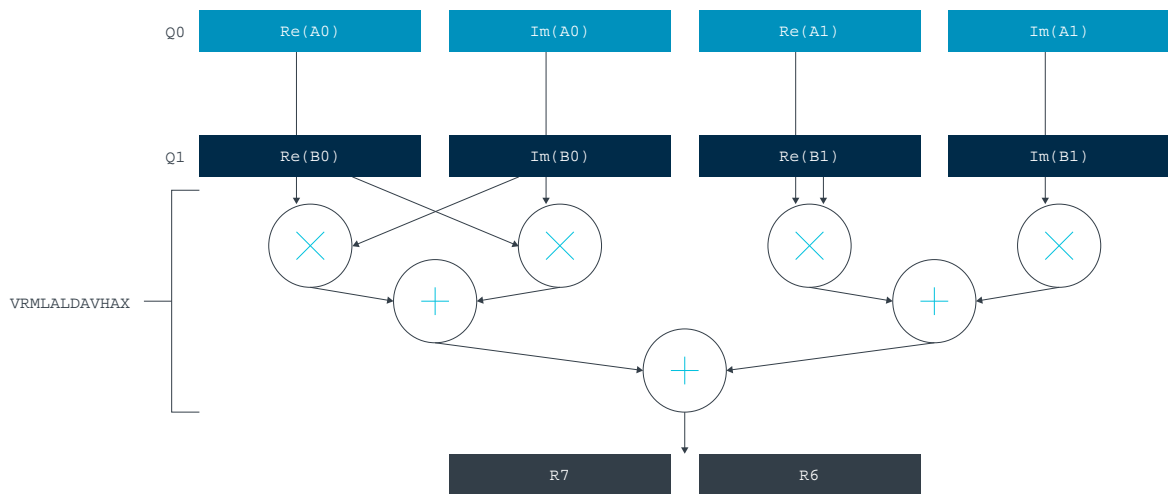
**Figure 8-1: ARM1564 Update to Images ST2 V8**

- The imaginary component of the dot product is calculated by the `VRMLALDAVHA` (Vector Rounding Multiply Subtract Long Dual Accumulate Across Vector Returning High 64 bits With Exchange) instruction. This instruction first swaps the values in each pair of values read from the first source register `Q1`, before multiplying them with the values from the second source register `Q0`. The results of the pairs of multiply operations are combined by adding them together. Finally, the scalar result is then added to the running total held in the two registers `R7` (high 32 bits) and `R6` (low 32 bits).

This implements the following part of the algorithm:

```
imagResult += pSrcA[(2*n)+0] * pSrcB[(2*n)+1] + pSrcA[(2*n)+1] * pSrcB[(2*n)+0];
```

The following diagram illustrates this calculation:

**Figure 8-2: ARM1564 Update to Images ST2 V7**

- The `ASRL` (Arithmetic Shift Right Long) instruction performs an arithmetic shift right by 6 bits, to convert the result into a Q16.48 number.
- The `STRD` (Store Register Dual) instruction returns the real and imaginary components of the result by writing them to the result registers.

The following complete code example shows how to interface between the C and assembly code:

```
#include <arm_mve.h>
#include <arm_math.h>
#include <stdio.h>

void my_cmplx_dot_prod_q31(
    q31_t * pSrcA,
    q31_t * pSrcB,
    uint32_t numSamples,
    q63_t * realResult,
    q63_t * imagResult)
{
    __asm volatile (
        "    clrm                    {r4-r7}                \n"
        "    wlstp.32                lr, %[cnt], 1f         \n"
        "2:                                \n"
        "    vldrw.32                 q0, [%[pA]], 16        \n"
        "    vldrw.32                 q1, [%[pB]], 16        \n"
        "    vrmlsldavha.s32           r4, r5, q0, q1         \n"
        "    vrmlaldavhax.s32          r6, r7, q0, q1         \n"
        "    letp                      lr, 2b                \n"
        "1:                                \n"
        "    asrl                      r4, r5, #6            \n"
        "    asrl                      r6, r7, #6            \n"
        "    strd                       r4, r5, [%[realResult]] \n"
        "    strd                       r6, r7, [%[imagResult]] \n"
        " : [pA] "+r"(pSrcA), [pB] "+r"(pSrcB)
        " : [cnt] "r"(numSamples * 2), [realResult] "r"(realResult),
        "   [imagResult] "r"(imagResult)
        " : "r4", "r5", "r6", "r7", "lr", "memory");
    }
}
```

```

int main() {
    // Setup data in input arrays
    q31_t A_src[] = {947483647, 834662098, 111222333, 555666777, 101202303,
                    555000222, 432654876, 999888777};
    q31_t B_src[] = {147483647, 623333999, 623957233, 876543098, 337744884,
                    112233445, 909808707, 543098765};

    // Create pointers to the two input arrays
    q31_t *pA_src = A_src;
    q31_t *pB_src = B_src;

    // Setup result variables
    q63_t res_real;
    q63_t res_imag;

    // Create pointers to the two result variables
    q63_t *pres_real = &res_real;
    q63_t *pres_imag = &res_imag;

    // Get the number of elements in the array
    int num_array_elements = sizeof(A_src) / sizeof(q31_t);

    // Divide by 2 to get the number of vector elements
    // (each vector element is a pair: a real and a complex component)
    int num_vector_elements = num_array_elements / 2;

    // Call the dot product function, reusing one of the input arrays as the result
    // array
    my_cmplx_dot_prod_q31(pA_src, pB_src, num_vector_elements, pres_real, pres_imag);

    // Print the result
    printf("\n\nreal=%lld ; complex=%lld\n", (long long)res_real, (long
    long)res_imag);

    return 0;
}

```

You can compile this code with Arm Compiler 6 as follows:

```
armclang -target arm-arm-none-eabi -march=armv8.1-m.main+mve.fp+fp.dp test.c
```

## Further reading

- [Q fixed-point number format \(Wikipedia\)](#)
- [GCC Documentation: How to Use Inline Assembly Language in C Code](#)
- [Arm Compiler Reference Guide: armclang Inline Assembler](#)
- [Arm Compiler User Guide: Writing inline assembly code](#)



## 9. Related information

Here are some resources related to material in this guide:

- [Armv8-M Architecture Reference Manual](#)
- [Procedure Call Standard for the Arm Architecture \(AAPCS\)](#)
- [Arm C Language Extensions \(ACLE\).](#)
- [Arm MVE Intrinsics Reference Architecture specification](#) (also available as [interactive HTML](#)).
- [Arm Compiler 6 toolchain](#)
- [Fast Models Reference Manual](#)
- [CMSIS DSP Software Library Reference](#)
- [Arm Compiler Reference Guide: -O](#)
- [Arm Compiler Reference Guide: armclang Inline Assembler](#)
- [Arm Compiler Reference Guide: Coding best practice for auto-vectorization](#)
- [Arm Compiler Reference Guide: Using pragmas to control auto-vectorization](#)
- [Arm Compiler User Guide: Selecting optimization options](#)
- [Arm Compiler User Guide: Writing inline assembly code](#)
- [Arm Compiler User Guide, Calling assembly functions from C and C++](#)
- [GCC Documentation: How to Use Inline Assembly Language in C Code](#)
- [Complex numbers \(Wikipedia\)](#)
- [Q fixed-point number format \(Wikipedia\)](#)
- Making Helium" blog post series:
  - [Making Helium: Why not just add Neon? \(1/4\)](#)
  - [Making Helium: Sudoku, registers and rabbits \(2/4\)](#)
  - [Making Helium: Going around in circles \(3/4\)](#)
  - [Making Helium: Bringing Amdahl's law to heel \(4/4\)](#)

## 10. Next steps

This guide has introduced a number of different techniques for writing Helium-enabled code.

After reading this guide, you will be ready to start writing your own code. The examples supplied with this guide are a good place to start learning. Further examples can be found by examining the source code for the [various examples](#) in the CMSIS-DSP pack.