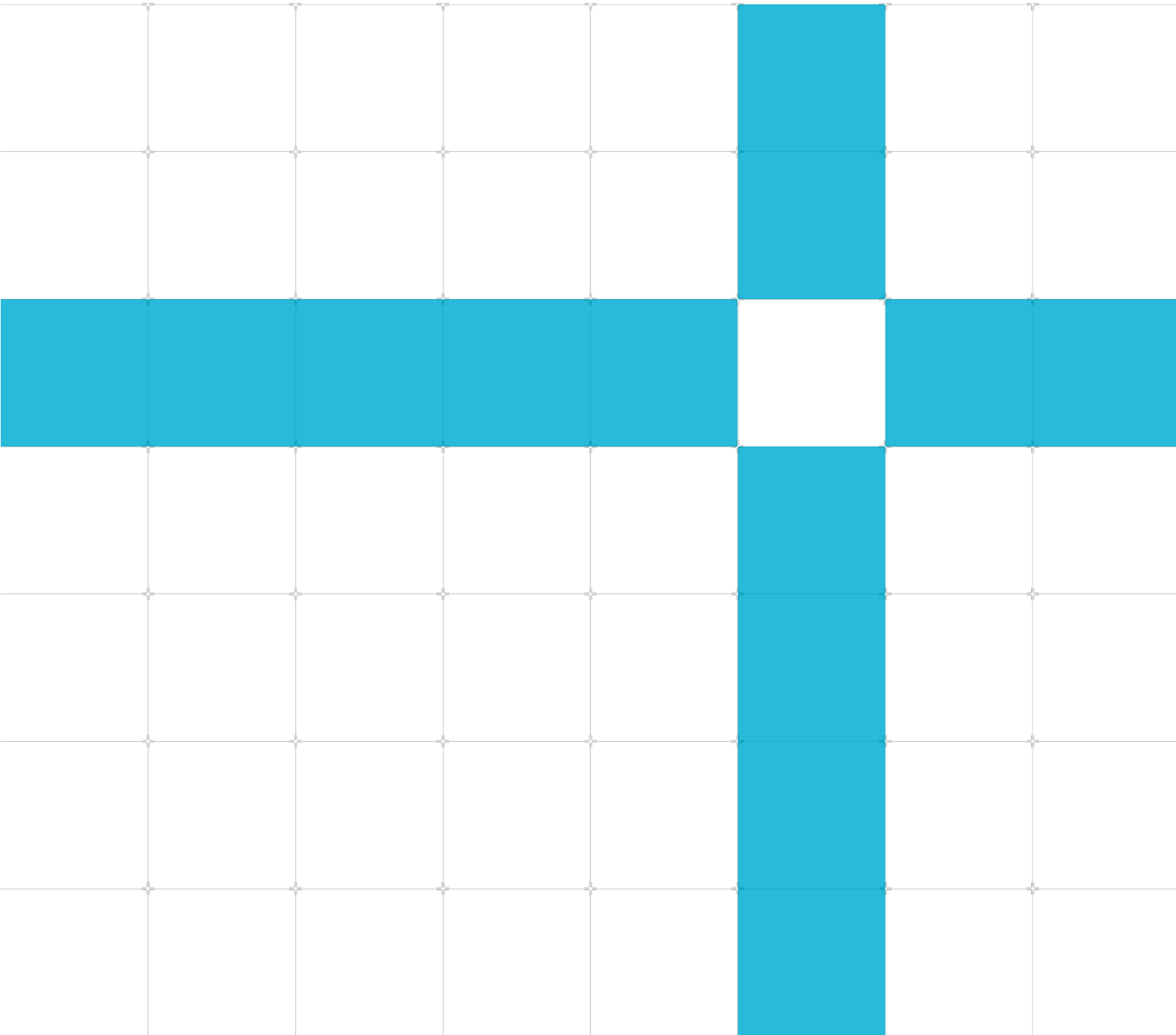
arm

Helium Programmer's Guide: Migrating to Helium from Neon

Non-Confidential

Issue 1.0
102107_0100_01



Helium Programmer's Guide: Migrating to Helium from Neon

Release information

Document history

Issue	Date	Confidentiality	Change
01	23rd March 2021	Non-confidential	First release.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

Progressive terminology commitment

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

We believe that this document contains no offensive terms. If you find offensive terms in this document, please email **terms@arm.com**.

Contents

1 Overview

This guide aims to help anyone migrating existing vector processing code that uses Neon intrinsics to Helium intrinsics. We will look at Neon code examples of varying complexity and investigate how to migrate this Neon code to Helium. By examining these examples, you will gain an understanding of some general migration principles that you can use to migrate your own Neon code to Helium.

Migration is necessary because although there are similarities between Neon and Helium, they are based on different architectures. As a result, software is not directly portable between the two.

The similarities between Neon and Helium include the following:

- Vector registers are 128-bit wide.
- Registers in the floating-point unit (FPU) are reused as vector registers.
- Some vector instructions are common between both Helium and Neon.

However, there are also differences between Neon and Helium, including the following:

- The number of vector registers. Helium supports eight vector registers, while Neon supports either 16 or 32 vector registers.
- Many Helium instructions use both vector and general-purpose registers. In Neon only a few instructions mix register usage in this way. However, some Neon instructions mix vector and scalar values either through immediate values or by using an individual vector element as a scalar.
- Helium supports newer data types like fp16 which the Neon extensions to older architectures do not support.
- Some Helium features like low-overhead branches and predication are specific to Helium.
- Neon provides a 64-bit half-vector size for narrowing and widening operations. Helium does not support this feature.
- Neon provides an interleaved three-way load and store. Helium does not support this feature.
- Neon and Helium provide different schemes for data widening and narrowing.

Because of these differences, there is no easy way to automatically translate Neon intrinsics code directly to Helium. However, this does not necessarily mean you must redesign the code from scratch. The following strategies can help migration:

- If the Neon code is already vectorized, the algorithmic structure in the software can be reused with adjustments for Helium.
- If the Neon code uses intrinsics, the compiler hides all register differences.
- If the Neon code uses intrinsics, some of the intrinsic functions are common between Neon and Helium.

This guide provides examples to illustrate the migration process, and each example includes the following:

- The original Neon code, together with a high-level explanation of what functions the Neon intrinsics perform.

- The direct Helium migration, designed to quickly get Neon code running on a Helium processor as an initial prototyping stage.

  These direct migration solutions do not usually provide an optimal solution. The resulting Helium code might miss important Helium optimization features and could perform poorly.
- The optimized Helium migration, designed to get the best performance out of Helium.

  These optimized migration solutions usually involve redesigning the original algorithm to take advantage of Helium-specific features like predication.

The examples in this guide cover a range of complexities. Simpler examples appear earlier in the guide, and complexity increases through subsequent sections.

The following are simple migration examples. In these simple examples the Neon code typically uses basic 128-bit operations. There is no complicated rearrangement of data between vector lanes. For simple cases, migration from Neon to Helium is generally possible with only small changes to the code:

- Single-precision vector logarithm
- Single-precision vector exponent
- Single-precision vector sine
- Vector minimum searching
- Floating-point vector complex dot product
- Single-precision 4x4 matrix multiplication

The following are intermediate migration examples. These examples cover some typical features that are found in Neon DSP code to highlight interesting Helium conversion challenges:

- Fixed-point 16-bit cross-correlation
- Floating-point 4x4 matrix transposition
- Integer 8-bit 4x4 matrix transposition
- RGB to grayscale conversion

Finally, the following advanced migration example looks at the Acoustic Echo Cancelation (AEC) implementation in Web Real-Time Communication (WebRTC):

- AEC in WebRTC

1.1 Before you begin

This guide assumes some familiarity with both Neon and Helium, and the intrinsics that they provide.

For background knowledge, you can read the Neon Programmer's Guide for Armv8-A and the Helium Programmer's Guide, especially the following sections:

- Neon Programmer's Guide:
  - Introducing Neon for Armv8-A
  - Optimizing C code with Neon intrinsics
  - Getting started with Neon Intrinsics on Android
  - Neon intrinsics Chromium case study
- Helium Programmer's Guide:
  - Introduction to Helium
  - Coding for Helium

The examples in this guide use both Neon and Helium intrinsics. You might find it useful to consult the following resources while reading this guide:

- Helium intrinsics reference
- Neon intrinsics reference
- Arm Helium Technology M-Profile Vector Extension (MVE) for Arm Cortex-M Processors Reference Book

2 Single-precision vector logarithm

The `vlogq_neon_f32` function implements single-precision floating-point vector logarithm calculations in both the Arm Compute Library and CMSIS DSP.

This implementation is based on Taylor series expansion.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/vmath.c

2.1 Neon implementation

The following code shows an implementation of a single-precision floating-point vector logarithm function using Neon intrinsics:

```
static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
{
  float32x4_t A   = vmlaq_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
  float32x4_t B   = vmlaq_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
  float32x4_t C   = vmlaq_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
  float32x4_t D   = vmlaq_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
  float32x4_t x2  = vmulq_f32(x, x);
  float32x4_t x4  = vmulq_f32(x2, x2);
  float32x4_t res = vmlaq_f32(vmlaq_f32(A, B, x2), vmlaq_f32(C, D, x2), x4);
  return res;
}

float32x4_t vlogq_neon_f32(float32x4_t x)
{
  // Extract exponent
  int32x4_t m = vsubq_s32(vreinterpretq_s32_u32(vshrq_n_u32(
     vreinterpretq_u32_f32(x), 23)), vdupq_n_s32(127));

  float32x4_t val = vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x),
     vshlq_n_s32(m, 23)));

  // Polynomial Approximation
  float32x4_t poly = vtaylor_polyq_f32(val, log_tab);

  // Reconstruct
  poly = vmlaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

  return poly;
}
```

2.2 Direct migration to Helium

The Neon implementation uses standard, common operations including the following:

- Contiguous vector load: `vld1q_f32`
- Vector multiplication: `vmulq_f32`
- Vector multiply-add: `vmlaq_f32`
- Unsigned integer to simple precision typecast: `vreinterpretq_s32_u32`
- Signed integer to single-precision floating-point conversion: `vcvtq_f32_s32`

All of these operations have direct equivalents in Helium.

One difference between Helium and Neon is that although Neon provides both fused and unfused multiply-add instructions, Helium only provides fused multiply-add instructions. The Neon implementation uses the unfused `vmlaq_f32` intrinsic. However, the Helium implementation must use the fused `vfmaq_f32` intrinsic.

The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
{
  float32x4_t A = vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 0]),
    vld1q_f32(&coeffs[4 * 4]), x);
  float32x4_t B = vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 2]),
    vld1q_f32(&coeffs[4 * 6]), x);
  float32x4_t C = vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 1]),
    vld1q_f32(&coeffs[4 * 5]), x);
  float32x4_t D = vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 3]),
    vld1q_f32(&coeffs[4 * 7]), x);
  float32x4_t x2 = vmulq_f32(x, x);
  float32x4_t x4 = vmulq_f32(x2, x2);
  float32x4_t res = vmlaq_emu_f32(vmlaq_emu_f32(A, B, x2), vmlaq_emu_f32(C, D, x2),
    x4);
  return res;
}

float32x4_t vlogq_helium_f32_direct(float32x4_t x)
{
  // Extract exponent
  int32x4_t m = vsubq_s32(vreinterpretq_s32_u32(vshrq_n_u32(
    vreinterpretq_u32_f32(x), 23)), vdupq_n_s32(127));

  float32x4_t val = vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x),
    vshlq_n_s32(m, 23)));

  // Polynomial Approximation
  float32x4_t  poly = vtaylor_polyq_f32(val, log_tab);

  // Reconstruct
  poly = vfmaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

  return poly;
}
```


Note

This implementation uses a macro, `vmlaq_emu_f32`, which maps directly to the Neon instruction `vfmaq_f32`. This macro is defined in the `helium_neon_helpers.h` header file.

The following image shows the differences between the Neon implementation and the direct Helium implementation:

Neon

```
static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
{
    float32x4_t     A =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
    float32x4_t     B =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
    float32x4_t     C =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
    float32x4_t     D =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
    float32x4_t     x2 = vmulq_f32(x, x);
    float32x4_t     x4 = vmulq_f32(x2, x2);
    float32x4_t     res = vmlaq_f32(vmlaq_f32(A, B, x2), vmlaq_f32(C, D, x2), x4);
    return res;
}


float32x4_t vlogq_neon_f32(float32x4_t x)
{
    // Extract exponent
    int32x4_t       m =
        vsubq_s32(vreinterpretq_s32_u32(vshrq_n_u32(vreinterpretq_u32_f32(x), 23)),
                  vdupq_n_s32(127));
    float32x4_t     val =
        vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x), vshlq_n_s32(m, 23)));

    // Polynomial Approximation
    float32x4_t     poly = vtaylor_polyq_f32(val, log_tab);

    // Reconstruct
    poly = vmlaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

    return poly;
}
```

Helium

```
static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
{
    float32x4_t     A =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
    float32x4_t     B =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
    float32x4_t     C =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
    float32x4_t     D =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
    float32x4_t     x2 = vmulq_f32(x, x);
    float32x4_t     x4 = vmulq_f32(x2, x2);
    float32x4_t     res = vmlaq_emu_f32(vmlaq_emu_f32(A, B, x2), vmlaq_emu_f32(C, D, x2), x4);
    return res;
}


float32x4_t vlogq_helium_f32_direct(float32x4_t x)
{
    // Extract exponent
    int32x4_t       m =
        vsubq_s32(vreinterpretq_s32_u32(vshrq_n_u32(vreinterpretq_u32_f32(x), 23)),
                  vdupq_n_s32(127));
    float32x4_t     val =
        vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x), vshlq_n_s32(m, 23)));

    // Polynomial Approximation
    float32x4_t     poly = vtaylor_polyq_f32(val, log_tab);

    // Reconstruct
    poly = vfmaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

    return poly;
}
```

The following GNU diff output shows the same differences in text form:

```
--- neon.c      Thu Oct 15 15:41:18 2020
+++ helium_direct.c     Thu Oct 15 15:57:36 2020
@@ -1,21 +1,21 @@
 static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
 {
     float32x4_t     A =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
     float32x4_t     B =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
     float32x4_t     C =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
     float32x4_t     D =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
     float32x4_t     x2 = vmulq_f32(x, x);
     float32x4_t     x4 = vmulq_f32(x2, x2);
-    float32x4_t     res = vmlaq_f32(vmlaq_f32(A, B, x2), vmlaq_f32(C, D, x2), x4);
+    float32x4_t     res = vmlaq_emu_f32(vmlaq_emu_f32(A, B, x2), vmlaq_emu_f32(C, D,
x2), x4);
     return res;
 }


-float32x4_t vlogq_neon_f32(float32x4_t x)
+float32x4_t vlogq_helium_f32_direct(float32x4_t x)
 {
     // Extract exponent
     int32x4_t       m =
@@ -28,7 +28,7 @@
     float32x4_t     poly = vtaylor_polyq_f32(val, log_tab);
```

```
     // Reconstruct
-    poly = vmlaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));
+    poly = vfmaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

     return poly;
 }
```

As you can see, the Helium implementation only differs from the Neon implementation by a few characters.

2.3 Optimized migration to Helium

Many Helium instructions allow flexible mixing of vector register and general-purpose register operands.

The optimized Helium implementation uses this flexible mixing of operands. The Neon implementation uses `vdupq_n_s32` to create a vector from a single scalar value in several places. Helium, on the other hand, can use the scalar values directly by using `_n_` variant intrinsics like `vsubq_n_s32` and `vfmaq_n_f32`.

The following code shows an optimized conversion of the Neon implementation to Helium:

```
float32x4_t vlogq_helium_f32(float32x4_t x)
{
  // Extract exponent
  int32x4_t m = vsubq_n_s32(vreinterpretq_s32_u32(vshrq_n_u32(
    vreinterpretq_u32_f32(x), 23)), 127);

  float32x4_t val = vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x),
    vshlq_n_s32(m, 23)));

  // Polynomial Approximation
  float32x4_t poly = vtaylor_polyq_f32(val, log_tab);

  // Reconstruct
  poly = vfmaq_n_f32(poly, vcvtq_f32_s32(m), LOG2);

  return poly;
}
```

The following image shows the differences between the Neon implementation and the optimized Helium implementation:

Neon

```
static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
{
    float32x4_t     A =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
    float32x4_t     B =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
    float32x4_t     C =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
    float32x4_t     D =
        vmlaq_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
    float32x4_t     x2 = vmulq_f32(x, x);
    float32x4_t     x4 = vmulq_f32(x2, x2);
    float32x4_t     res = vmlaq_f32(vmlaq_f32(A, B, x2), vmlaq_f32(C, D, x2), x4);
    return res;
}


float32x4_t vlogq_neon_f32(float32x4_t x)
{
    // Extract exponent
    int32x4_t       m =
        vsubq_s32(vreinterpretq_s32_u32(vshrq_n_u32(vreinterpretq_u32_f32(x), 23)),
                  vdupq_n_s32(127));
    float32x4_t     val =
        vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x), vshlq_n_s32(m, 23)));

    // Polynomial Approximation
    float32x4_t     poly = vtaylor_polyq_f32(val, log_tab);

    // Reconstruct
    poly = vmlaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

    return poly;
}
```

Helium

```
static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
{
    float32x4_t     A =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
    float32x4_t     B =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
    float32x4_t     C =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
    float32x4_t     D =
        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
    float32x4_t     x2 = vmulq_f32(x, x);
    float32x4_t     x4 = vmulq_f32(x2, x2);
    float32x4_t     res = vmlaq_emu_f32(vmlaq_emu_f32(A, B, x2), vmlaq_emu_f32(C, D, x2), x4);
    return res;
}


float32x4_t vlogq_helium_f32(float32x4_t x)
{
    // Extract exponent
    int32x4_t       m =
        vsubq_n_s32(vreinterpretq_s32_u32(vshrq_n_u32(vreinterpretq_u32_f32(x), 23)),
                    127);
    float32x4_t     val =
        vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x), vshlq_n_s32(m, 23)));

    // Polynomial Approximation
    float32x4_t     poly = vtaylor_polyq_f32(val, log_tab);

    // Reconstruct
    poly = vfmaq_n_f32(poly, vcvtq_f32_s32(m), LOG2);

    return poly;
}
```

The following GNU diff output shows the same differences in text form:

```
--- neon.c      Thu Oct 15 15:41:18 2020
+++ helium_optimized.c  Thu Oct 15 15:42:11 2020
@@ -1,26 +1,27 @@
 static inline float32x4_t vtaylor_polyq_f32(float32x4_t x, const float32_t * coeffs)
 {
     float32x4_t     A =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 0]), vld1q_f32(&coeffs[4 * 4]), x);
     float32x4_t     B =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 2]), vld1q_f32(&coeffs[4 * 6]), x);
     float32x4_t     C =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 1]), vld1q_f32(&coeffs[4 * 5]), x);
     float32x4_t     D =
-        vmlaq_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
+        vmlaq_emu_f32(vld1q_f32(&coeffs[4 * 3]), vld1q_f32(&coeffs[4 * 7]), x);
     float32x4_t     x2 = vmulq_f32(x, x);
     float32x4_t     x4 = vmulq_f32(x2, x2);
-    float32x4_t     res = vmlaq_f32(vmlaq_f32(A, B, x2), vmlaq_f32(C, D, x2), x4);
+    float32x4_t     res = vmlaq_emu_f32(vmlaq_emu_f32(A, B, x2), vmlaq_emu_f32(C, D,
x2), x4);
     return res;
 }


-float32x4_t vlogq_neon_f32(float32x4_t x)
+float32x4_t vlogq_helium_f32(float32x4_t x)
 {
     // Extract exponent
     int32x4_t       m =
-        vsubq_s32(vreinterpretq_s32_u32(vshrq_n_u32(vreinterpretq_u32_f32(x), 23)),
-                  vdupq_n_s32(127));
+        vsubq_n_s32(vreinterpretq_s32_u32(vshrq_n_u32(vreinterpretq_u32_f32(x), 23)),
+                    127);
     float32x4_t     val =
         vreinterpretq_f32_s32(vsubq_s32(vreinterpretq_s32_f32(x), vshlq_n_s32(m,
23)));

@@ -28,7 +29,7 @@
     float32x4_t     poly = vtaylor_polyq_f32(val, log_tab);

     // Reconstruct
-    poly = vmlaq_f32(poly, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));
+    poly = vfmaq_n_f32(poly, vcvtq_f32_s32(m), LOG2);
```

```
    return poly;
```

3 Single-precision vector exponent

The `vexpq_neon_f32` function implements single-precision floating-point vector exponent calculations in both the Arm Compute Library and CMSIS DSP.

This example is similar to the `vlogq_neon_f32` function that is described in Single-precision vector logarithm. The `vexpq_neon_f32` function uses the same Taylor series expansion algorithm as `vlogq_neon_f32`, but also uses conditional selection to round very small result values to zero.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/vmath.c

3.1 Neon implementation

The following code shows an implementation of a single-precision floating-point vector exponent function using Neon intrinsics:

```
float32x4_t vexpq_neon_f32(float32x4_t x)

{
  // Perform range reduction [-log(2),log(2)]
  int32x4_t       m = vcvtq_s32_f32(vmulq_f32(x, vdupq_n_f32(INVLOG2)));
  float32x4_t     val = vmlsq_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

  // Polynomial Approximation
  float32x4_t     poly = vtaylor_polyq_f32(val, exp_tab);

  // Reconstruct
  poly =
      vreinterpretq_f32_s32(vqaddq_s32
                            (vreinterpretq_s32_f32(poly), vqshlq_n_s32(m, 23)));
  poly = vbslq_f32(vcltq_s32(m, vdupq_n_s32(-126)), vdupq_n_f32(0.0f), poly);
  return poly;
}
```

3.2 Direct migration to Helium

The `vexpq_neon_f32` function rounds very small result values to zero, using the following intrinsics:

- The `vcltq_s32` intrinsic performs the comparison and creates a bitmask identifying the small result values.
- The `vbslq_f32` intrinsic returns either the original result or zero as specified by the bitmap.

Helium uses predication to perform conditional operations. In this example, the following changes were made:

- The `vcmpltq_s32` intrinsic performs the comparison and sets the predication register P0 accordingly. This predication register is exposed at C level in a variable of type `mve_pred16_t`.

  The `vcmpltq_s32` intrinsic has the following prototype:

  ```
  mve_pred16_t [__arm_]vcmpltq[_s32] (int32x4_t a, int32x4_t b);
  ```

- The `vpselq_f32` intrinsic uses the contents of the predication register to perform a bytewise conditional select of 2 single-precision floating point vectors. The migrated Helium code uses this intrinsic to conditionally select either the original result or a vector of zeroes.

  The `vpselq_f32` intrinsic has the following prototype:

  ```
  float32x4_t [__arm_]vpselq[_f32] (float32x4_t a, float32x4_t b, mve_pred16_t p);
  ```

The original Neon code is as follows:

```
poly = vbslq_f32(vcltq_s32(m, vdupq_n_s32(-126)), vdupq_n_f32(0.0f), poly);
```

The corresponding migrated Helium code is:

```
poly = vpselq(vdupq_n_f32(0.0f), poly, vcmpltq_s32(m, vdupq_n_s32(-126)));
```

The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
float32x4_t vexpq_helium_f32_direct(float32x4_t x)
{
  // Perform range reduction [-log(2),log(2)]
  int32x4_t       m = vcvtq_s32_f32(vmulq_f32(x, vdupq_n_f32(INVLOG2)));
  float32x4_t     val = vmlsq_emu_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

  // Polynomial Approximation
  float32x4_t     poly = vtaylor_polyq_f32(val, exp_tab);

  // Reconstruct
  poly =
      vreinterpretq_f32_s32(vqaddq_s32
                            (vreinterpretq_s32_f32(poly), vqshlq_n_s32(m, 23)));
  poly = vpselq(vdupq_n_f32(0.0f), poly, vcmpltq_s32(m, vdupq_n_s32(-126)));
  return poly;
}
```

The following image shows the differences between the Neon implementation and the direct Helium implementation:

The following GNU diff output shows the same differences in text form:

```
--- neon.c      Fri Oct 16 12:02:19 2020
+++ helium_direct.c     Fri Oct 16 12:01:37 2020
@@ -1,8 +1,8 @@
-float32x4_t vexpq_neon_f32(float32x4_t x)
+float32x4_t vexpq_helium_f32_direct(float32x4_t x)
 {
     // Perform range reduction [-log(2),log(2)]
```

```
     int32x4_t       m = vcvtq_s32_f32(vmulq_f32(x, vdupq_n_f32(INVLOG2)));
-    float32x4_t     val = vmlsq_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));
+    float32x4_t     val = vmlsq_emu_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

     // Polynomial Approximation
     float32x4_t     poly = vtaylor_polyq_f32(val, exp_tab);
@@ -11,6 +11,7 @@
     poly =
         vreinterpretq_f32_s32(vqaddq_s32
                               (vreinterpretq_s32_f32(poly), vqshlq_n_s32(m, 23)));
-    poly = vbslq_f32(vcltq_s32(m, vdupq_n_s32(-126)), vdupq_n_f32(0.0f), poly);
+    poly = vpselq(vdupq_n_f32(0.0f), poly, vcmpltq_s32(m, vdupq_n_s32(-126)));
     return poly;
 }
```

3.3 Optimized migration to Helium

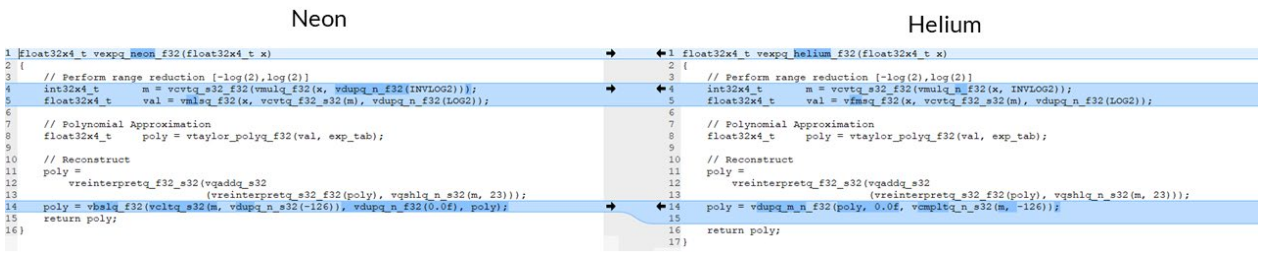
Like with the **Single-precision vector logarithm** example, we can use flexible mixing of vector register and general-purpose register operands to save several `VDUP` instructions.

Additionally, the `vdupq_m_n_f32` intrinsic lets us provide the predicate directly, which improves the readability of the code.

The following code shows an optimized conversion of the Neon implementation to Helium:

```
float32x4_t vexpq_helium_f32(float32x4_t x)
{
    // Perform range reduction [-log(2),log(2)]
    int32x4_t       m = vcvtq_s32_f32(vmulq_n_f32(x, INVLOG2));
    float32x4_t     val = vfmsq_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

    // Polynomial Approximation
    float32x4_t     poly = vtaylor_polyq_f32(val, exp_tab);

    // Reconstruct
    poly =
        vreinterpretq_f32_s32(vqaddq_s32
                              (vreinterpretq_s32_f32(poly), vqshlq_n_s32(m, 23)));
    poly = vdupq_m_n_f32(poly, 0.0f, vcmpltq_n_s32(m, -126));

    return poly;
}
```

The following image shows the differences between the Neon implementation and the optimized Helium implementation:



The following GNU diff output shows the same differences in text form:

```
--- neon.c      Fri Oct 16 12:02:19 2020
+++ helium_optimized.c  Fri Oct 16 12:02:07 2020
@@ -1,8 +1,8 @@
```

```
-float32x4_t vexpq_neon_f32(float32x4_t x)
+float32x4_t vexpq_helium_f32(float32x4_t x)
 {
     // Perform range reduction [-log(2),log(2)]
-    int32x4_t       m = vcvtq_s32_f32(vmulq_f32(x, vdupq_n_f32(INVLOG2)));
-    float32x4_t     val = vmlsq_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));
+    int32x4_t       m = vcvtq_s32_f32(vmulq_n_f32(x, INVLOG2));
+    float32x4_t     val = vfmsq_f32(x, vcvtq_f32_s32(m), vdupq_n_f32(LOG2));

     // Polynomial Approximation
     float32x4_t     poly = vtaylor_polyq_f32(val, exp_tab);
@@ -11,6 +11,7 @@
     poly =
         vreinterpretq_f32_s32(vqaddq_s32
                               (vreinterpretq_s32_f32(poly), vqshlq_n_s32(m, 23)));
-    poly = vbslq_f32(vcltq_s32(m, vdupq_n_s32(-126)), vdupq_n_f32(0.0f), poly);
+    poly = vdupq_m_n_f32(poly, 0.0f, vcmpltq_n_s32(m, -126));
     return poly;
 }
```

4 Single-precision vector sine

The `vsinq_neon_f32` function implements single-precision floating-point vector sine calculations in both the Arm Compute Library and CMSIS DSP.

This example involves several operations, including the following:

- Floating-point arithmetic
- Logical operations
- Comparison operations to manage changes of sign
- Rebasing operations

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/vmath.c

4.1 Neon implementation

The following code shows an implementation of a single-precision floating-point vector sine function using Neon intrinsics:

```
float32x4_t vsinq_neon_f32(float32x4_t val)
{
  const float32x4_t pi_v = vdupq_n_f32(M_PI);
  const float32x4_t pio2_v = vdupq_n_f32(M_PI / 2);
  const float32x4_t ipi_v = vdupq_n_f32(1 / M_PI);

  //Find positive or negative
  const int32x4_t c_v = vabsq_s32(vcvtq_s32_f32(vmulq_f32(val, ipi_v)));
  const uint32x4_t sign_v = vcleq_f32(val, vdupq_n_f32(0));
  const uint32x4_t odd_v = vandq_u32(vreinterpretq_u32_s32(c_v), vdupq_n_u32(1));

  uint32x4_t      neg_v = veorq_u32(odd_v, sign_v);

  //Modulus a - (n * int(a*(1/n)))
  float32x4_t     ma = vsubq_f32(vabsq_f32(val), vmulq_f32(pi_v,
    vcvtq_f32_s32(c_v)));

  const uint32x4_t reb_v = vcgeq_f32(ma, pio2_v);

  //Rebase a between 0 and pi/2
  ma = vbslq_f32(reb_v, vsubq_f32(pi_v, ma), ma);

  //Taylor series
  const float32x4_t ma2 = vmulq_f32(ma, ma);

  //2nd elem: x^3 / 3!
  float32x4_t     elem = vmulq_f32(vmulq_f32(ma, ma2), vdupq_n_f32(te_sin_coeff2));
  float32x4_t     res = vsubq_f32(ma, elem);

  //3rd elem: x^5 / 5!
```

```
  elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff3));
  res = vaddq_f32(res, elem);

  //4th elem: x^7 / 7!float32x2_t vsin_f32(float32x2_t val)
  elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff4));
  res = vsubq_f32(res, elem);

  //5th elem: x^9 / 9!
  elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff5));
  res = vaddq_f32(res, elem);

  //Change of sign
  neg_v = vshlq_n_u32(neg_v, 31);
  res = vreinterpretq_f32_u32(veorq_u32(vreinterpretq_u32_f32(res), neg_v));
  return res;
}
```

4.2 Direct migration to Helium

Like with the Single-precision vector exponent example, we must change the conditional selection operations to use predication.

In addition to straightforward conditional selects, the Neon code also uses a comparison vector bitmask to perform other logical operations. For example, the Neon implementation uses the `vcleq_f32` and `vcgeq_f32` intrinsics with vectors of duplicated values. These operations cannot be directly expressed using the Helium predication functionality. Instead, we must expand the predicate mask into a bitmask vector using the following operation:

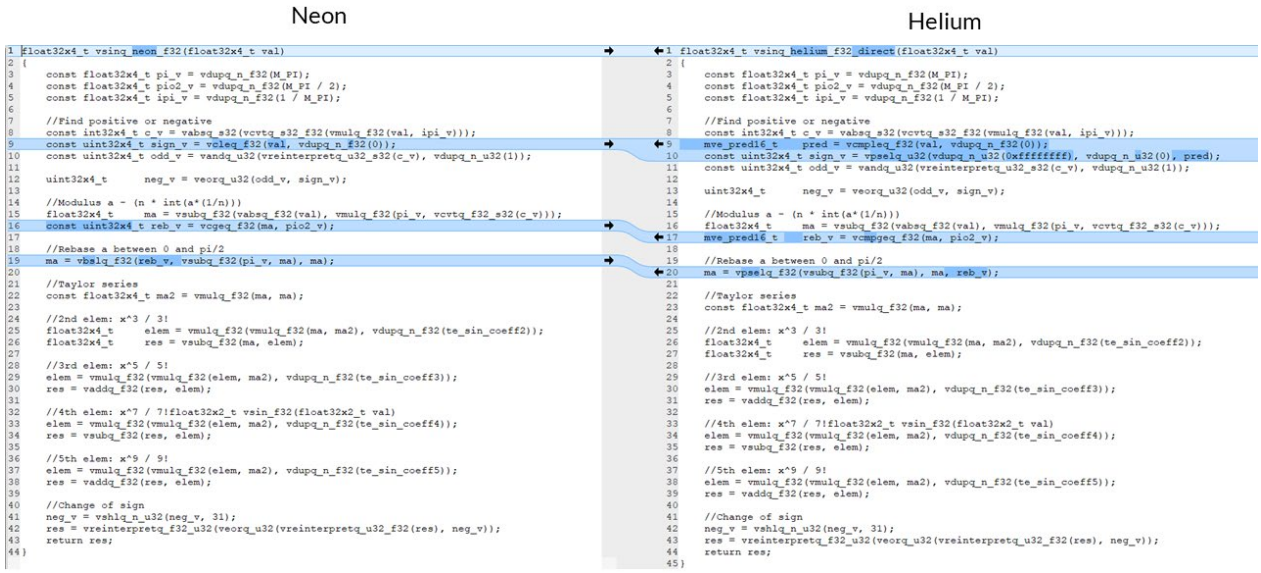
```
const uint32x4_t sign_v = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0), pred);
```

The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
float32x4_t vsinq_helium_f32_direct(float32x4_t val)
{
  const float32x4_t pi_v = vdupq_n_f32(M_PI);
  const float32x4_t pio2_v = vdupq_n_f32(M_PI / 2);
  const float32x4_t ipi_v = vdupq_n_f32(1 / M_PI);

  //Find positive or negative
  const int32x4_t c_v = vabsq_s32(vcvtq_s32_f32(vmulq_f32(val, ipi_v)));
  mve_pred16_t    pred = vcmpleq_f32(val, vdupq_n_f32(0));
  const uint32x4_t sign_v = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0), pred);
  const uint32x4_t odd_v = vandq_u32(vreinterpretq_u32_s32(c_v), vdupq_n_u32(1));

  uint32x4_t      neg_v = veorq_u32(odd_v, sign_v);

  //Modulus a - (n * int(a*(1/n)))
  float32x4_t     ma = vsubq_f32(vabsq_f32(val), vmulq_f32(pi_v, vcvtq_f32_s32(c_v)));
  mve_pred16_t    reb_v = vcmpgeq_f32(ma, pio2_v);

  //Rebase a between 0 and pi/2
  ma = vpselq_f32(vsubq_f32(pi_v, ma), ma, reb_v);

  //Taylor series
  const float32x4_t ma2 = vmulq_f32(ma, ma);

  //2nd elem: x^3 / 3!
  float32x4_t     elem = vmulq_f32(vmulq_f32(ma, ma2), vdupq_n_f32(te_sin_coeff2));
  float32x4_t     res = vsubq_f32(ma, elem);

  //3rd elem: x^5 / 5!
```

```
  elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff3));
  res = vaddq_f32(res, elem);

  //4th elem: x^7 / 7!float32x2_t vsin_f32(float32x2_t val)
  elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff4));
  res = vsubq_f32(res, elem);

  //5th elem: x^9 / 9!
  elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff5));
  res = vaddq_f32(res, elem);

  //Change of sign
  neg_v = vshlq_n_u32(neg_v, 31);
  res = vreinterpretq_f32_u32(veorq_u32(vreinterpretq_u32_f32(res), neg_v));
  return res;
}
```

The following image shows the differences between the Neon implementation and the direct Helium implementation:



The following GNU diff output shows the same differences in text form:

```
--- neon.c      Fri Oct 16 13:46:10 2020
+++ helium_direct.c     Fri Oct 16 13:50:02 2020
@@ -1,4 +1,4 @@
-float32x4_t vsinq_neon_f32(float32x4_t val)
+float32x4_t vsinq_helium_f32_direct(float32x4_t val)
 {
     const float32x4_t pi_v = vdupq_n_f32(M_PI);
     const float32x4_t pio2_v = vdupq_n_f32(M_PI / 2);
@@ -6,7 +6,8 @@

     //Find positive or negative
     const int32x4_t c_v = vabsq_s32(vcvtq_s32_f32(vmulq_f32(val, ipi_v)));
-    const uint32x4_t sign_v = vcleq_f32(val, vdupq_n_f32(0));
+    mve_pred16_t    pred = vcmpleq_f32(val, vdupq_n_f32(0));
+    const uint32x4_t sign_v = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0),
pred);
     const uint32x4_t odd_v = vandq_u32(vreinterpretq_u32_s32(c_v), vdupq_n_u32(1));

     uint32x4_t      neg_v = veorq_u32(odd_v, sign_v);
@@ -13,10 +14,10 @@
```

```
     //Modulus a - (n * int(a*(1/n)))
     float32x4_t     ma = vsubq_f32(vabsq_f32(val), vmulq_f32(pi_v,
vcvtq_f32_s32(c_v)));
-    const uint32x4_t reb_v = vcgeq_f32(ma, pio2_v);
+    mve_pred16_t    reb_v = vcmpgeq_f32(ma, pio2_v);

     //Rebase a between 0 and pi/2
-    ma = vbslq_f32(reb_v, vsubq_f32(pi_v, ma), ma);
+    ma = vpselq_f32(vsubq_f32(pi_v, ma), ma, reb_v);

     //Taylor series
     const float32x4_t ma2 = vmulq_f32(ma, ma);
```

4.3 Optimized migration to Helium

Like with the **Single-precision vector logarithm** example, we can use flexible mixing of vector register and general-purpose register operands. In this example, mixing scalar and vector operands saves a large number of VDUP instructions.

The following code shows an optimized conversion of the Neon implementation to Helium:

```
float32x4_t vsinq_helium_f32(float32x4_t val)
{
  const float32_t pi_v = M_PI;
  const float32_t pio2_v = M_PI / 2;
  const float32_t ipi_v = 1 / M_PI;

  //Find positive or negative
  const int32x4_t c_v = vabsq_s32(vcvtq_s32_f32(vmulq_n_f32(val, ipi_v)));
  mve_pred16_t    pred = vcmpleq_f32(val, vdupq_n_f32(0));
  const uint32x4_t sign_v = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0), pred);
  const uint32x4_t odd_v = vandq_u32(vreinterpretq_u32_s32(c_v), vdupq_n_u32(1));

  uint32x4_t      neg_v = veorq_u32(odd_v, sign_v);

  //Modulus a - (n * int(a*(1/n)))
  float32x4_t     ma = vsubq_f32(vabsq_f32(val), vmulq_n_f32(vcvtq_f32_s32(c_v),
                              pi_v));
  mve_pred16_t    reb_v = vcmpgeq_n_f32(ma, pio2_v);

  //Rebase a between 0 and pi/2
  ma = vpselq_f32(vsubq_f32(vdupq_n_f32(pi_v), ma), ma, reb_v);

  //Taylor series
  const float32x4_t ma2 = vmulq_f32(ma, ma);

  //2nd elem: x^3 / 3!
  float32x4_t     elem = vmulq_n_f32(vmulq_f32(ma, ma2), te_sin_coeff2);
  float32x4_t     res = vsubq_f32(ma, elem);

  //3rd elem: x^5 / 5!
  elem = vmulq_n_f32(vmulq_f32(elem, ma2), te_sin_coeff3);
  res = vaddq_f32(res, elem);

  //4th elem: x^7 / 7!float32x2_t vsin_f32(float32x2_t val)
  elem = vmulq_n_f32(vmulq_f32(elem, ma2), te_sin_coeff4);
  res = vsubq_f32(res, elem);

  //5th elem: x^9 / 9!
  elem = vmulq_n_f32(vmulq_f32(elem, ma2), te_sin_coeff5);
```

```
  res = vaddq_f32(res, elem);

  //Change of sign
  neg_v = vshlq_n_u32(neg_v, 31);
  res = vreinterpretq_f32_u32(veorq_u32(vreinterpretq_u32_f32(res), neg_v));
  return res;
}
```

The following image shows the differences between the Neon implementation and the optimized Helium implementation:

The following GNU diff output shows the same differences in text form:

```
--- neon.c      Fri Oct 16 13:46:10 2020
+++ helium_optimized.c  Fri Oct 16 13:46:34 2020
@@ -1,40 +1,41 @@
-float32x4_t vsinq_neon_f32(float32x4_t val)
+float32x4_t vsinq_helium_f32(float32x4_t val)
 {
-    const float32x4_t pi_v = vdupq_n_f32(M_PI);
-    const float32x4_t pio2_v = vdupq_n_f32(M_PI / 2);
-    const float32x4_t ipi_v = vdupq_n_f32(1 / M_PI);
+    const float32_t pi_v = M_PI;
+    const float32_t pio2_v = M_PI / 2;
+    const float32_t ipi_v = 1 / M_PI;

     //Find positive or negative
-    const int32x4_t c_v = vabsq_s32(vcvtq_s32_f32(vmulq_f32(val, ipi_v)));
-    const uint32x4_t sign_v = vcleq_f32(val, vdupq_n_f32(0));
+    const int32x4_t c_v = vabsq_s32(vcvtq_s32_f32(vmulq_n_f32(val, ipi_v)));
+    mve_pred16_t    pred = vcmpleq_f32(val, vdupq_n_f32(0));
+    const uint32x4_t sign_v = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0),
pred);
     const uint32x4_t odd_v = vandq_u32(vreinterpretq_u32_s32(c_v), vdupq_n_u32(1));

     uint32x4_t      neg_v = veorq_u32(odd_v, sign_v);

     //Modulus a - (n * int(a*(1/n)))
-    float32x4_t     ma = vsubq_f32(vabsq_f32(val), vmulq_f32(pi_v,
vcvtq_f32_s32(c_v)));
-    const uint32x4_t reb_v = vcgeq_f32(ma, pio2_v);
+    float32x4_t     ma = vsubq_f32(vabsq_f32(val), vmulq_n_f32(vcvtq_f32_s32(c_v),
```

```
pi_v));
+    mve_pred16_t    reb_v = vcmpgeq_n_f32(ma, pio2_v);

     //Rebase a between 0 and pi/2
-    ma = vbslq_f32(reb_v, vsubq_f32(pi_v, ma), ma);
+    ma = vpselq_f32(vsubq_f32(vdupq_n_f32(pi_v), ma), ma, reb_v);

     //Taylor series
     const float32x4_t ma2 = vmulq_f32(ma, ma);

     //2nd elem: x^3 / 3!
-    float32x4_t     elem = vmulq_f32(vmulq_f32(ma, ma2), vdupq_n_f32(te_sin_coeff2));
+    float32x4_t     elem = vmulq_n_f32(vmulq_f32(ma, ma2), te_sin_coeff2);
     float32x4_t     res = vsubq_f32(ma, elem);

     //3rd elem: x^5 / 5!
-    elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff3));
+    elem = vmulq_n_f32(vmulq_f32(elem, ma2), te_sin_coeff3);
     res = vaddq_f32(res, elem);

     //4th elem: x^7 / 7!float32x2_t vsin_f32(float32x2_t val)
-    elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff4));
+    elem = vmulq_n_f32(vmulq_f32(elem, ma2), te_sin_coeff4);
     res = vsubq_f32(res, elem);

     //5th elem: x^9 / 9!
-    elem = vmulq_f32(vmulq_f32(elem, ma2), vdupq_n_f32(te_sin_coeff5));
+    elem = vmulq_n_f32(vmulq_f32(elem, ma2), te_sin_coeff5);
     res = vaddq_f32(res, elem);
```

5 Vector minimum searching

The `arm_min_f32` function computes the minimum value of an array of floating-point values. The function returns both the minimum value and its position within the array.

The function is implemented in the CMSIS DSP library.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/vmin.c

5.1 Neon implementation

The following code shows an implementation of a single-precision floating-point vector sine function using Neon intrinsics:

```
void arm_min_neon_f32(const float32_t * pSrc,
                      uint32_t blockSize, float32_t * pResult, uint32_t * pIndex)
{
  float32_t   maxVal1, out;       /* Temporary variables to store the output value. */
  uint32_t    blkCnt, outIndex;   /* loop counter */
  float32x4_t outV, srcV;
  float32x2_t outV2;
  uint32x4_t  idxV;
  static const uint32_t indexInit[4] = { 4, 5, 6, 7 };
  static const uint32_t countVInit[4] = { 0, 1, 2, 3 };
  uint32x4_t  maxIdx;
  uint32x4_t  index;
  uint32x4_t  delta;
  uint32x4_t  countV;
  uint32x2_t  countV2;

  maxIdx = vdupq_n_u32(ULONG_MAX);
  delta = vdupq_n_u32(4);
  index = vld1q_u32(indexInit);
  countV = vld1q_u32(countVInit);

  /* Initialize the index value to zero. */
  outIndex = 0U;

  /* Load first input value that act as reference value for comparison */
  if (blockSize <= 3) {
    out = *pSrc++;
    blkCnt = blockSize - 1;
    while (blkCnt > 0U) {
      /* Initialize maxVal to the next consecutive values one by one */
      maxVal1 = *pSrc++;

      /* compare for the maximum value */
      if (out > maxVal1) {
        /* Update the maximum value and its index */
        out = maxVal1;
        outIndex = blockSize - blkCnt;
      }
```

```
      /* Decrement the loop counter */
      blkCnt--;
    }
  } else {
    outV = vld1q_f32(pSrc);
    pSrc += 4;

    /* Compute 4 outputs at a time */
    blkCnt = (blockSize - 4) >> 2U;

    while (blkCnt > 0U) {
      srcV = vld1q_f32(pSrc)
      pSrc += 4;

      idxV = vcltq_f32(srcV, outV);
      outV = vbslq_f32(idxV, srcV, outV);
      countV = vbslq_u32(idxV, index, countV);

      index = vaddq_u32(index, delta);

      /* Decrement the loop counter */
      blkCnt--;
    }

    outV2 = vpmin_f32(vget_low_f32(outV), vget_high_f32(outV));
    outV2 = vpmin_f32(outV2, outV2);
    out = vget_lane_f32(outV2, 0);

    idxV = vceqq_f32(outV, vdupq_n_f32(out));
    countV = vbslq_u32(idxV, countV, maxIdx);

    countV2 = vpmin_u32(vget_low_u32(countV), vget_high_u32(countV));
    countV2 = vpmin_u32(countV2, countV2);
    outIndex = vget_lane_u32(countV2, 0);

    /* if (blockSize - 1U) is not multiple of 4 */
    blkCnt = (blockSize - 4) % 4U;

    while (blkCnt > 0U) {
      /* Initialize maxVal to the next consecutive values one by one */
      maxVal1 = *pSrc++;

      /* compare for the maximum value */
      if (out > maxVal1) {
        /* Update the maximum value and its index */
        out = maxVal1;
        outIndex = blockSize - blkCnt;
      }

      /* Decrement the loop counter */
      blkCnt--;
    }
  }

  /* Store the maximum value and its index into destination pointers */
  *pResult = out;
  *pIndex = outIndex;
}
```

The core loop of the Neon code is as follows:

```
    /* Compute 4 outputs at a time */
    blkCnt = (blockSize - 4) >> 2U;
```

```
while (blkCnt > 0U) {
  srcV = vld1q_f32(pSrc)
  pSrc += 4;

  idxV = vcltq_f32(srcV, outV);
  outV = vbslq_f32(idxV, srcV, outV);
  countV = vbslq_u32(idxV, index, countV);

  index = vaddq_u32(index, delta);

  /* Decrement the loop counter */
  blkCnt--;
}
```

This loop iterates over each element of the data array, four elements at a time. The main loop compares the value in each lane of the input vector with the value in the corresponding lane of the vector `outV`. If a lower value is found, that lane of `outV` is updated with the new lowest value. After all iterations have finished, `outV` contains four values with each value being the minimum seen in each lane.

The global minimum for the whole array is then calculated by selecting the lowest of these four values using the pairwise minimum operation:

```
outV2 = vpmin_f32(vget_low_f32(outV), vget_high_f32(outV));
outV2 = vpmin_f32(outV2, outV2);
out = vget_lane_f32(outV2, 0);

idxV = vceqq_f32(outV, vdupq_n_f32(out));
countV = vbslq_u32(idxV, countV, maxIdx);

countV2 = vpmin_u32(vget_low_u32(countV), vget_high_u32(countV));
countV2 = vpmin_u32(countV2, countV2);
outIndex = vget_lane_u32(countV2, 0);
```

5.2 Direct migration to Helium

Like with the Single-precision vector exponent example, we must change the conditional selection operation in the main loop to use predication. The following code shows the Helium main loop:

```
/* Compute 4 outputs at a time */
blkCnt = (blockSize - 4) >> 2U;

while (blkCnt > 0U) {
  srcV = vld1q_f32(pSrc);
  pSrc += 4;

  pred = vcmpleq_f32(srcV, outV);
  outV = vpselq_f32(srcV, outV, pred);
  countV = vpselq_f32(index, countV, pred);

  index = vaddq_u32(index, delta);

  /* Decrement the loop counter */
  blkCnt--;
}
```

To calculate the global minimum, Helium does not provide a pairwise minimum operation. However, Helium instead provides across-vector operators including vector minimum for both floating point, `vminnmvq_f32`, and integers, `vminvq_u32`.

This simplifies the global minimum calculation as follows:

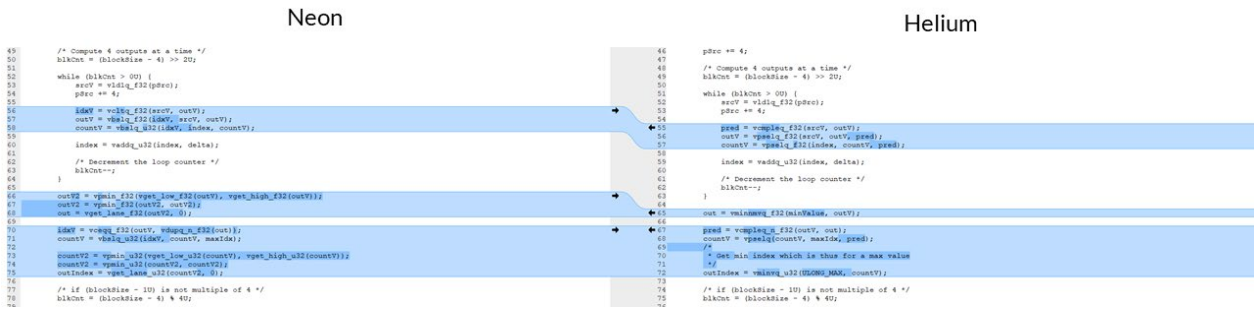
```
  out = vminnmvq_f32(minValue, outV);
  pred = vcmpleq_n_f32(outV, out);
  countV = vpselq(countV, maxIdx, pred);
  outIndex = vminvq_u32(ULONG_MAX, countV);
```

The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
void arm_min_helium_f32_direct(const float32_t * pSrc,
                   uint32_t blockSize, float32_t * pResult, uint32_t * pIndex)
{
  float32_t       maxVal1, out;       /* Temporary variables to store the output value.
*/
  uint32_t        blkCnt, outIndex;   /* loop counter */
  float32x4_t     outV, srcV;
  static const uint32_t indexInit[4] = { 4, 5, 6, 7 };
  static const uint32_t countVInit[4] = { 0, 1, 2, 3 };
  uint32x4_t      maxIdx;
  uint32x4_t      index;
  uint32x4_t      delta;
  uint32x4_t      countV;
  mve_pred16_t    pred;
  float32_t       minValue = F32_MAX;

  maxIdx = vdupq_n_u32(ULONG_MAX);
  delta = vdupq_n_u32(4);
  index = vld1q_u32(indexInit);
  countV = vld1q_u32(countVInit);

  /* Initialize the index value to zero. */
  outIndex = 0U;

  /* Load first input value that act as reference value for comparison */
  if (blockSize <= 3) {
    out = *pSrc++;

    blkCnt = blockSize - 1;

    while (blkCnt > 0U) {
      /* Initialize maxVal to the next consecutive values one by one */
      maxVal1 = *pSrc++;

      /* compare for the maximum value */
      if (out > maxVal1) {
        /* Update the maximum value and its index */
        out = maxVal1;
        outIndex = blockSize - blkCnt;
      }

      /* Decrement the loop counter */
      blkCnt--;
    }
  } else {
    outV = vld1q_f32(pSrc);
    pSrc += 4;

    /* Compute 4 outputs at a time */
    blkCnt = (blockSize - 4) >> 2U;

    while (blkCnt > 0U) {
      srcV = vld1q_f32(pSrc);
      pSrc += 4;

      pred = vcmpleq_f32(srcV, outV);
```

```
      outV = vpselq_f32(srcV, outV, pred);
      countV = vpselq_f32(index, countV, pred);

      index = vaddq_u32(index, delta);

      /* Decrement the loop counter */
      blkCnt--;
    }

    out = vminnmvq_f32(minValue, outV);

    pred = vcmpleq_n_f32(outV, out);
    countV = vpselq(countV, maxIdx, pred);
    /*
     * Get min index which is thus for a max value
     */
    outIndex = vminvq_u32(ULONG_MAX, countV);

    /* if (blockSize - 1U) is not multiple of 4 */
    blkCnt = (blockSize - 4) % 4U;

    while (blkCnt > 0U) {
      /* Initialize maxVal to the next consecutive values one by one */
      maxVal1 = *pSrc++;

      /* compare for the maximum value */
      if (out > maxVal1) {
        /* Update the maximum value and its index */
        out = maxVal1;
        outIndex = blockSize - blkCnt;
      }

      /* Decrement the loop counter */
      blkCnt--;
    }
  }

  /* Store the maximum value and its index into destination pointers */
  *pResult = out;
  *pIndex = outIndex;
}
```

The following image shows the major differences between the Neon implementation and the direct Helium implementation:



The following GNU diff output shows the same differences in text form:

```
--- neon.c      Mon Oct 19 16:26:12 2020
+++ helium_direct.c     Mon Oct 19 16:26:37 2020
@@ -1,11 +1,9 @@
-void arm_min_neon_f32(const float32_t * pSrc,
```

```
-                      uint32_t blockSize, float32_t * pResult, uint32_t * pIndex)
+void arm_min_helium_f32_direct(const float32_t * pSrc,
+                               uint32_t blockSize, float32_t * pResult, uint32_t *
pIndex)
 {
     float32_t       maxVal1, out;       /* Temporary variables to store the output
value. */
     uint32_t        blkCnt, outIndex;   /* loop counter */
     float32x4_t     outV, srcV;
-    float32x2_t     outV2;
-    uint32x4_t      idxV;
     static const uint32_t indexInit[4] = { 4, 5, 6, 7 };
     static const uint32_t countVInit[4] = { 0, 1, 2, 3 };
     uint32x4_t      maxIdx;
@@ -12,7 +10,8 @@
     uint32x4_t      index;
     uint32x4_t      delta;
     uint32x4_t      countV;
-    uint32x2_t      countV2;
+    mve_pred16_t    pred;
+    float32_t       minValue = F32_MAX;

     maxIdx = vdupq_n_u32(ULONG_MAX);
     delta = vdupq_n_u32(4);
@@ -53,9 +52,9 @@
             srcV = vld1q_f32(pSrc);
             pSrc += 4;

-            idxV = vcltq_f32(srcV, outV);
-            outV = vbslq_f32(idxV, srcV, outV);
-            countV = vbslq_u32(idxV, index, countV);
+            pred = vcmpleq_f32(srcV, outV);
+            outV = vpselq_f32(srcV, outV, pred);
+            countV = vpselq_f32(index, countV, pred);

             index = vaddq_u32(index, delta);

@@ -63,17 +62,15 @@
             blkCnt--;
         }

-        outV2 = vpmin_f32(vget_low_f32(outV), vget_high_f32(outV));
-        outV2 = vpmin_f32(outV2, outV2);
-        out = vget_lane_f32(outV2, 0);
+        out = vminnmvq_f32(minValue, outV);

-        idxV = vceqq_f32(outV, vdupq_n_f32(out));
-        countV = vbslq_u32(idxV, countV, maxIdx);
+        pred = vcmpleq_n_f32(outV, out);
+        countV = vpselq(countV, maxIdx, pred);
+        /*
+         * Get min index which is thus for a max value
+         */
+        outIndex = vminvq_u32(ULONG_MAX, countV);

-        countV2 = vpmin_u32(vget_low_u32(countV), vget_high_u32(countV));
-        countV2 = vpmin_u32(countV2, countV2);
-        outIndex = vget_lane_u32(countV2, 0);
-
         /* if (blockSize - 1U) is not multiple of 4 */
         blkCnt = (blockSize - 4) % 4U;
```

5.3 Optimized migration to Helium

Helium supports tail-predicated loops, which optimize processing when the amount of data to be processed is not an exact multiple of the vector length. For loops that do not require explicit unrolling, tail-predicted loops deal with the scalar residual parts.

Tail-predicated loops need to be expressed in a way that the compiler can recognize. The compiler can then issue loop instructions using the DLSTP, WLSTP, and LETP instructions. Arm Compiler 6 requires that intrinsics operators belonging to the loop are predicated using a `vctp` predicate.

The following code shows an optimized conversion of the Neon implementation to Helium:

```
void arm_min_helium_f32(const float32_t * pSrc,
                 uint32_t blockSize, float32_t * pResult, uint32_t * pIndex)
{
  int32_t          blkCnt = blockSize;
  float32x4_t      vecSrc;
  float32x4_t      curExtremValVec = vdupq_n_f32(F32_MAX);
  float32_t        minValue = F32_MAX;
  uint32_t         idx = blockSize;
  uint32x4_t       indexVec;
  uint32x4_t       curExtremIdxVec;
  uint32_t         curIdx = 0;
  mve_pred16_t     pred;


  indexVec = vidupq_wb_u32(&curIdx, 1);
  curExtremIdxVec = vdupq_n_u32(0);

  do {
    mve_pred16_t    p = vctp32q(blkCnt);

    vecSrc = vld1q_z_f32(pSrc, p);
    /*
     * Get current min per lane and current index per lane
     * when a min is selected
     */
    pred = vcmpleq_m_f32(vecSrc, curExtremValVec, p);
    curExtremValVec = vorrq_m_f32(curExtremValVec, vecSrc, vecSrc, pred);
    curExtremIdxVec = vorrq_m_f32(curExtremIdxVec, indexVec, indexVec, pred);

    indexVec = vaddq_n_u32(indexVec, 4);

    pSrc += 4;
    blkCnt -= 4;
  }
  while (blkCnt > 0);

  /*
   * Get min value across the vector
   */
  minValue = vminnmvq(minValue, curExtremValVec);
  /*
   * set index for lower values to min possible index
   */
  pred = vcmpleq(curExtremValVec, minValue);
  indexVec = vpselq(curExtremIdxVec, vdupq_n_u32(blockSize), pred);
  /*
   * Get min index which is thus for a min value
   */
  idx = vminvq(idx, indexVec);
  /*
```

```
   * Save result
   */
  *pIndex = idx;
  *pResult = minValue;
}
```

6 Floating-point vector complex dot product

The `arm_cmplx_dot_prod_neon_f32` function computes the dot product of two single-precision floating-point complex vectors.

The vectors are multiplied element-by-element and then summed, using the following underlying algorithm:

```
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];
}
```

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/cmplx_dot.c

6.1 Neon implementation

The following code shows an implementation of a single-precision floating-point vector exponent function using Neon intrinsics:

```
void arm_cmplx_dot_prod_neon_f32(const float32_t * pSrcA,
                                 const float32_t * pSrcB,
                                 uint32_t numSamples,
                                 float32_t * realResult, float32_t * imagResult)
{
  uint32_t        blkCnt;     /* Loop counter */
  float32_t       real_sum = 0.0f, imag_sum = 0.0f;   /* Temporary result variables */
  float32_t       a0, b0, c0, d0;

  float32x4x2_t   vec1, vec2, vec3, vec4;
  float32x4_t     accR, accI;
  float32x2_t     accum = vdup_n_f32(0);

  accR = vdupq_n_f32(0.0f);
  accI = vdupq_n_f32(0.0f);

  /* Loop unrolling: Compute 8 outputs at a time */
  blkCnt = numSamples >> 3U;

  while (blkCnt > 0U) {
    /* C = (A[0]+jA[1])*(B[0]+jB[1]) + ...  */
    /* Calculate dot product and then store the result in a temporary buffer. */

    vec1 = vld2q_f32(pSrcA);
    vec2 = vld2q_f32(pSrcB);

    /* Increment pointers */
```

```
    pSrcA += 8;
    pSrcB += 8;

    /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
    accR = vmlaq_f32(accR, vec1.val[0], vec2.val[0]);
    accR = vmlsq_f32(accR, vec1.val[1], vec2.val[1]);

    /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
    accI = vmlaq_f32(accI, vec1.val[1], vec2.val[0]);
    accI = vmlaq_f32(accI, vec1.val[0], vec2.val[1]);

    vec3 = vld2q_f32(pSrcA);
    vec4 = vld2q_f32(pSrcB);

    /* Increment pointers */
    pSrcA += 8;
    pSrcB += 8;

    /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
    accR = vmlaq_f32(accR, vec3.val[0], vec4.val[0]);
    accR = vmlsq_f32(accR, vec3.val[1], vec4.val[1]);

    /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
    accI = vmlaq_f32(accI, vec3.val[1], vec4.val[0]);
    accI = vmlaq_f32(accI, vec3.val[0], vec4.val[1]);

    /* Decrement the loop counter */
    blkCnt--;
  }

  accum = vpadd_f32(vget_low_f32(accR), vget_high_f32(accR));
  real_sum += vget_lane_f32(accum, 0) + vget_lane_f32(accum, 1);

  accum = vpadd_f32(vget_low_f32(accI), vget_high_f32(accI));
  imag_sum += vget_lane_f32(accum, 0) + vget_lane_f32(accum, 1);

  /* Tail */
  blkCnt = numSamples & 0x7;


  while (blkCnt > 0U) {
    a0 = *pSrcA++;
    b0 = *pSrcA++;
    c0 = *pSrcB++;
    d0 = *pSrcB++;

    real_sum += a0 * c0;
    imag_sum += a0 * d0;
    real_sum -= b0 * d0;
    imag_sum += b0 * c0;

    /* Decrement loop counter */
    blkCnt--;
  }

  /* Store real and imaginary result in destination buffer. */
  *realResult = real_sum;
  *imagResult = imag_sum;
}
```

The Neon implementation of floating-point complex dot product uses the following vector operations:

- De-interleaved load `vld2q_f32` to separate the real and imaginary parts

- A series of `vmlaq_f32` and `vmlsq_f32` operations to perform the computation of the per-lane accumulated complex multiplications
- Pairwise addition `vpadd_f32` to perform final summation outside the loop

6.2 Direct migration to Helium

Most of the Neon intrinsics used in this example have Helium counterparts. The exceptions are vector pairwise addition, unfused multiply-accumulate, and unfused multiply-subtract.

The Neon implementation uses the `vld2q_f32` intrinsic to perform an interleaved two-element load for the alternating real and imaginary components of the complex numbers. Neon and Helium both support interleaved two- and four-element loads and stores. Neon and Helium provide the same `vld2q_f32`, `vld4q_f32`, `vst2q_f32`, and `vst4q_f32` intrinsics, but the resulting instructions are different. Helium requires two or four calls with different pattern IDs to fill a pair or a quadruplet of 128-bit vectors.

For example, Helium and Neon both provide the following intrinsic:

```
float32x4x2_t vld2q_f32 (float32_t const * ptr)
```

Compiling for Neon results in the following single A64 instruction:

```
LD2 {Vt.4S - Vt2.4S},[Xn]
```

Compiling for Helium results in the following pair of instructions:

```
VLD20.32 {Qt, Qt+1}, [Rn]
VLD21.32 {Qt, Qt+1}, [Rn]
```


Note

Helium does not provide interleaved three-element load and store intrinsics, but the RGB to grayscale conversion example shows how to mimic this behavior using scatter-gather operations.

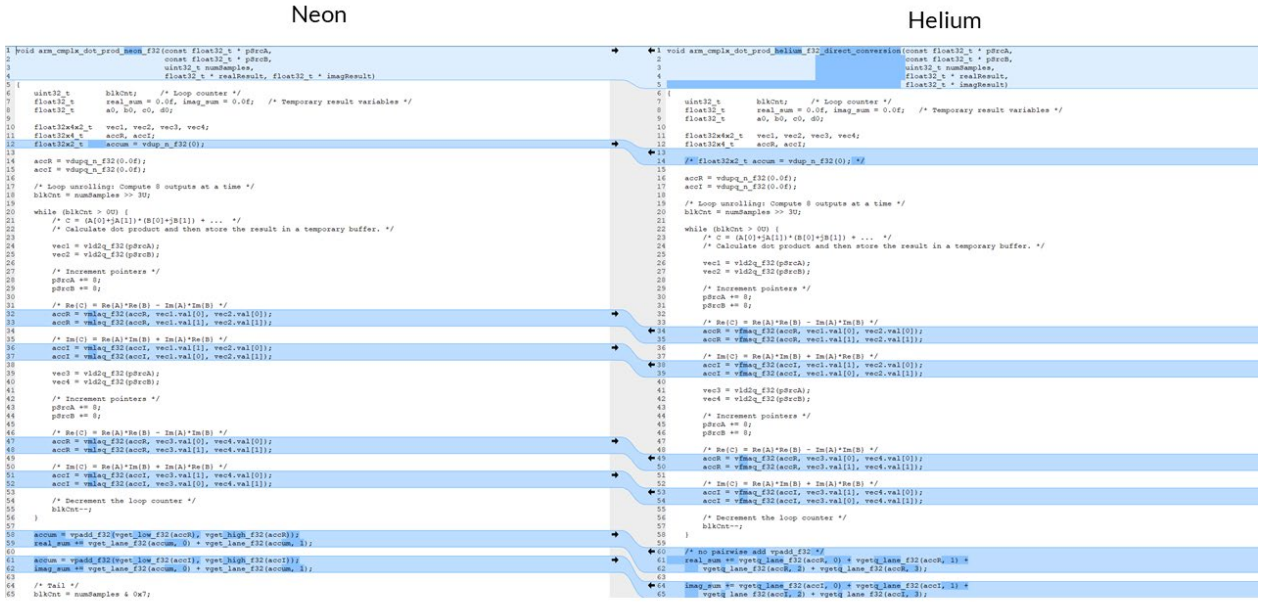
The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
void arm_cmplx_dot_prod_helium_f32_direct_conversion(const float32_t * pSrcA,
                                                     const float32_t * pSrcB,
                                                     uint32_t numSamples,
                                                     float32_t * realResult,
                                                     float32_t * imagResult)
{
  uint32_t        blkCnt;     /* Loop counter */
  float32_t       real_sum = 0.0f, imag_sum = 0.0f;   /* Temporary result variables */
  float32_t       a0, b0, c0, d0;

  float32x4x2_t   vec1, vec2, vec3, vec4;
  float32x4_t     accR, accI;

  /* float32x2_t accum = vdup_n_f32(0); */

  accR = vdupq_n_f32(0.0f);
  accI = vdupq_n_f32(0.0f);

  /* Loop unrolling: Compute 8 outputs at a time */
  blkCnt = numSamples >> 3U;
```

```
while (blkCnt > 0U) {
  /* C = (A[0]+jA[1])*(B[0]+jB[1]) + ...  */
  /* Calculate dot product and then store the result in a temporary buffer. */
  vec1 = vld2q_f32(pSrcA);
  vec2 = vld2q_f32(pSrcB);

  /* Increment pointers */
  pSrcA += 8;
  pSrcB += 8;

  /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
  accR = vfmaq_f32(accR, vec1.val[0], vec2.val[0]);
  accR = vfmsq_f32(accR, vec1.val[1], vec2.val[1]);

  /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
  accI = vfmaq_f32(accI, vec1.val[1], vec2.val[0]);
  accI = vfmaq_f32(accI, vec1.val[0], vec2.val[1]);

  vec3 = vld2q_f32(pSrcA);
  vec4 = vld2q_f32(pSrcB);

  /* Increment pointers */
  pSrcA += 8;
  pSrcB += 8;

  /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
  accR = vfmaq_f32(accR, vec3.val[0], vec4.val[0]);
  accR = vfmsq_f32(accR, vec3.val[1], vec4.val[1]);

  /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
  accI = vfmaq_f32(accI, vec3.val[1], vec4.val[0]);
  accI = vfmaq_f32(accI, vec3.val[0], vec4.val[1]);

  /* Decrement the loop counter */
  blkCnt--;
}

/* no pairwise add vpadd_f32 */
real_sum += vgetq_lane_f32(accR, 0) + vgetq_lane_f32(accR, 1) +
    vgetq_lane_f32(accR, 2) + vgetq_lane_f32(accR, 3);

imag_sum += vgetq_lane_f32(accI, 0) + vgetq_lane_f32(accI, 1) +
    vgetq_lane_f32(accI, 2) + vgetq_lane_f32(accI, 3);

/* Tail */
blkCnt = numSamples & 0x7;
while (blkCnt > 0U) {
  a0 = *pSrcA++;
  b0 = *pSrcA++;
  c0 = *pSrcB++;
  d0 = *pSrcB++;

  real_sum += a0 * c0;
  imag_sum += a0 * d0;
  real_sum -= b0 * d0;
  imag_sum += b0 * c0;

  /* Decrement loop counter */
  blkCnt--;
}

/* Store real and imaginary result in destination buffer. */
*realResult = real_sum;
```

```
  *imagResult = imag_sum;
}
```

The following image shows the major differences between the Neon implementation and the direct Helium implementation:



The following GNU diff output shows the same differences in text form:

```
--- neon.c      Tue Oct 20 11:55:50 2020
+++ helium_direct.c     Tue Oct 20 11:54:50 2020
@@ -1,7 +1,8 @@
-void arm_cmplx_dot_prod_neon_f32(const float32_t * pSrcA,
-                                 const float32_t * pSrcB,
-                                 uint32_t numSamples,
-                                 float32_t * realResult, float32_t * imagResult)
+void arm_cmplx_dot_prod_helium_f32_direct_conversion(const float32_t * pSrcA,
+                                                     const float32_t * pSrcB,
+                                                     uint32_t numSamples,
+                                                     float32_t * realResult,
+                                                     float32_t * imagResult)
 {
     uint32_t        blkCnt;     /* Loop counter */
     float32_t       real_sum = 0.0f, imag_sum = 0.0f;   /* Temporary result variables
*/
@@ -9,8 +10,9 @@

     float32x4x2_t   vec1, vec2, vec3, vec4;
     float32x4_t     accR, accI;
-    float32x2_t     accum = vdup_n_f32(0);

+    /* float32x2_t accum = vdup_n_f32(0); */
+
     accR = vdupq_n_f32(0.0f);
     accI = vdupq_n_f32(0.0f);

@@ -29,12 +31,12 @@
         pSrcB += 8;

         /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
-        accR = vmlaq_f32(accR, vec1.val[0], vec2.val[0]);
```

```
-        accR = vmlsq_f32(accR, vec1.val[1], vec2.val[1]);
+        accR = vfmaq_f32(accR, vec1.val[0], vec2.val[0]);
+        accR = vfmsq_f32(accR, vec1.val[1], vec2.val[1]);

         /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
-        accI = vmlaq_f32(accI, vec1.val[1], vec2.val[0]);
-        accI = vmlaq_f32(accI, vec1.val[0], vec2.val[1]);
+        accI = vfmaq_f32(accI, vec1.val[1], vec2.val[0]);
+        accI = vfmaq_f32(accI, vec1.val[0], vec2.val[1]);

         vec3 = vld2q_f32(pSrcA);
         vec4 = vld2q_f32(pSrcB);
@@ -44,27 +46,26 @@
         pSrcB += 8;

         /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
-        accR = vmlaq_f32(accR, vec3.val[0], vec4.val[0]);
-        accR = vmlsq_f32(accR, vec3.val[1], vec4.val[1]);
+        accR = vfmaq_f32(accR, vec3.val[0], vec4.val[0]);
+        accR = vfmsq_f32(accR, vec3.val[1], vec4.val[1]);

         /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
-        accI = vmlaq_f32(accI, vec3.val[1], vec4.val[0]);
-        accI = vmlaq_f32(accI, vec3.val[0], vec4.val[1]);
+        accI = vfmaq_f32(accI, vec3.val[1], vec4.val[0]);
+        accI = vfmaq_f32(accI, vec3.val[0], vec4.val[1]);

         /* Decrement the loop counter */
         blkCnt--;
     }

-    accum = vpadd_f32(vget_low_f32(accR), vget_high_f32(accR));
-    real_sum += vget_lane_f32(accum, 0) + vget_lane_f32(accum, 1);
+    /* no pairwise add vpadd_f32 */
+    real_sum += vgetq_lane_f32(accR, 0) + vgetq_lane_f32(accR, 1) +
+        vgetq_lane_f32(accR, 2) + vgetq_lane_f32(accR, 3);

-    accum = vpadd_f32(vget_low_f32(accI), vget_high_f32(accI));
-    imag_sum += vget_lane_f32(accum, 0) + vget_lane_f32(accum, 1);
+    imag_sum += vgetq_lane_f32(accI, 0) + vgetq_lane_f32(accI, 1) +
+        vgetq_lane_f32(accI, 2) + vgetq_lane_f32(accI, 3);

     /* Tail */
     blkCnt = numSamples & 0x7;
     while (blkCnt > 0U) {
         a0 = *pSrcA++;
         b0 = *pSrcA++;
```

6.3 Optimized migration to Helium

Helium supports new complex operations which allow a more natural implementation and do not require a de-interleaving stage. For example, Helium provides the Vector Complex Multiply Accumulate VCMLA instruction, which operates on complex numbers that are represented in registers as pairs of elements.

The following code shows an optimized core loop that uses these complex instructions:

```
while (blkCnt > 0U) {
  vecSrcA = vld1q((const float32_t *) pSrcA);
  vecSrcB = vld1q((const float32_t *) pSrcB);
  /* Re{C} = Re{A}*Re{B} - Im{A}*Im{B} */
```

```
  /* Im{C} = Re{A}*Im{B} + Im{A}*Re{B} */
  vec_acc = vcmlaq(vec_acc, vecSrcA, vecSrcB);
  vec_acc = vcmlaq_rot90(vec_acc, vecSrcA, vecSrcB);
  /*
   * Decrement the blkCnt loop counter
   * Advance vector source and destination pointers
   */
  pSrcA += 4;
  pSrcB += 4;
  blkCnt--;
}

/* lane summation */
real_sum = vgetq_lane(vec_acc, 0) + vgetq_lane(vec_acc, 2);
imag_sum = vgetq_lane(vec_acc, 1) + vgetq_lane(vec_acc, 3);
```

Further performance improvements are possible on dual beat cores like Cortex-M55. By unrolling loops, the compiler can perfectly interleave vector loads and vector complex multiplications. The following code shows a fully optimized Cortex-M55 version of the function:

```
void arm_cmplx_dot_prod_helium_f32(const float32_t * pSrcA,
                                   const float32_t * pSrcB,
                                   uint32_t numSamples,
                                   float32_t * realResult, float32_t * imagResult)
{
  int32_t         blkCnt;
  float32_t       real_sum, imag_sum;
  float32x4_t     vecSrcA, vecSrcB;
  float32x4_t     vec_acc = vdupq_n_f32(0.0f);
  float32x4_t     vecSrcC, vecSrcD;

  blkCnt = numSamples / 4;
  blkCnt -= 1;

  if (blkCnt > 0) {
    /* will give more freedom to the compiler to generate stall-free code */
    vecSrcA = vld1q(pSrcA);
    vecSrcB = vld1q(pSrcB);
    pSrcA += 4;
    pSrcB += 4;

    while (blkCnt > 0U) {
      vec_acc = vcmlaq(vec_acc, vecSrcA, vecSrcB);
      vecSrcC = vld1q(pSrcA);
      pSrcA += 4;

      vec_acc = vcmlaq_rot90(vec_acc, vecSrcA, vecSrcB);
      vecSrcD = vld1q(pSrcB);
      pSrcB += 4;

      vec_acc = vcmlaq(vec_acc, vecSrcC, vecSrcD);
      vecSrcA = vld1q(pSrcA);
      pSrcA += 4;

      vec_acc = vcmlaq_rot90(vec_acc, vecSrcC, vecSrcD);
      vecSrcB = vld1q(pSrcB);
      pSrcB += 4;
      /*
       * Decrement the blockSize loop counter
       */
      blkCnt--;
    }
```

```
    /* process last elements out of loop to keep the SW pipeline */
    vec_acc = vcmlaq(vec_acc, vecSrcA, vecSrcB);
    vecSrcC = vld1q(pSrcA);

    /* Tail handling code elided... */
  }
}
```


Note

Armv8.3A provides the FCMLA Floating-point Complex Multiply Accumulate instruction, which provides similar functionality to the Helium instructions in this code implementation.

7 Fixed-point vector complex dot product

The `arm_cmplx_dot_prod_neon_q15` function computes the dot product of two Q15 fixed-point complex vectors.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/cmplx_dot.c

7.1 Neon implementation

The Neon implementation has a similar construction to the Floating-point vector complex dot product. The main difference is the complex accumulation handling which uses 2 levels of widening, as follows:

- Long multiplication of 16-bit values using `vmull_s16` gives 32-bit results.
- Long pairwise addition of 32-bit values using `vpadalq_s32` gives 64-bit results.

The following code shows an implementation of a fixed-point vector exponent function using Neon intrinsics:

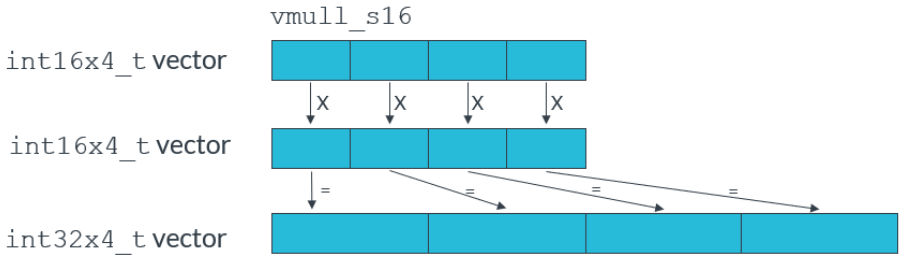
```
void arm_cmplx_dot_prod_neon_q15(const q15_t * pSrcA,
                                 const q15_t * pSrcB,
                                 uint32_t numSamples, q31_t * realResult,
                                 q31_t * imagResult)
{
  uint32_t        blkCnt;     /* Loop counter */
  q63_t           real_sum = 0, imag_sum = 0; /* Temporary result variables */
  q15_t           a0, b0, c0, d0;

  int16x8x2_t     vec1, vec2;
  int32x4_t       tempL, tempH;
  int64x2_t       resr1, resr2, resi;

  resr1 = vdupq_n_s64(0);
  resr2 = vdupq_n_s64(0);
  resi = vdupq_n_s64(0);

  /* loop unrolling */
  blkCnt = numSamples >> 3U;

  while (blkCnt > 0U) {
    vec1 = vld2q_s16(pSrcA);
    vec2 = vld2q_s16(pSrcB);
    pSrcA += 16;
    pSrcB += 16;

    tempL = vmull_s16(vget_low_s16(vec1.val[0]), vget_low_s16(vec2.val[0]));
    tempH = vmull_s16(vget_high_s16(vec1.val[0]), vget_high_s16(vec2.val[0]));
```

```
    resr1 = vpadalq_s32(resr1, tempL);
    resr1 = vpadalq_s32(resr1, tempH);

    tempL = vmull_s16(vget_low_s16(vec1.val[1]), vget_low_s16(vec2.val[1]));
    tempH = vmull_s16(vget_high_s16(vec1.val[1]), vget_high_s16(vec2.val[1]));

    resr2 = vpadalq_s32(resr2, tempL);
    resr2 = vpadalq_s32(resr2, tempH);

    tempL = vmull_s16(vget_low_s16(vec1.val[0]), vget_low_s16(vec2.val[1]));
    tempH = vmull_s16(vget_high_s16(vec1.val[0]), vget_high_s16(vec2.val[1]));

    resi = vpadalq_s32(resi, tempL);
    resi = vpadalq_s32(resi, tempH);

    tempL = vmull_s16(vget_low_s16(vec1.val[1]), vget_low_s16(vec2.val[0]));
    tempH = vmull_s16(vget_high_s16(vec1.val[1]), vget_high_s16(vec2.val[0]));

    resi = vpadalq_s32(resi, tempL);
    resi = vpadalq_s32(resi, tempH);

    /* Decrement the blockSize loop counter */
    blkCnt--;
  }
  real_sum += resr1[0] + resr1[1];
  real_sum -= (resr2[0] + resr2[1]);
  imag_sum += resi[0] + resi[1];

  /* tail */
  blkCnt = numSamples & 0x7;

  while (blkCnt > 0U) {
    a0 = *pSrcA++;
    b0 = *pSrcA++;
    c0 = *pSrcB++;
    d0 = *pSrcB++;

    real_sum += (q31_t) a0 *c0;
    imag_sum += (q31_t) a0 *d0;
    real_sum -= (q31_t) b0 *d0;
    imag_sum += (q31_t) b0 *c0;

    /* Decrement loop counter */
    blkCnt--;
  }

  /* Store real and imaginary result in 8.24 format  */
  /* Convert real data in 34.30 to 8.24 by 6 right shifts */
  *realResult = (q31_t) (real_sum >> 6);
  /* Convert imaginary data in 34.30 to 8.24 by 6 right shifts */
  *imagResult = (q31_t) (imag_sum >> 6);
}
```

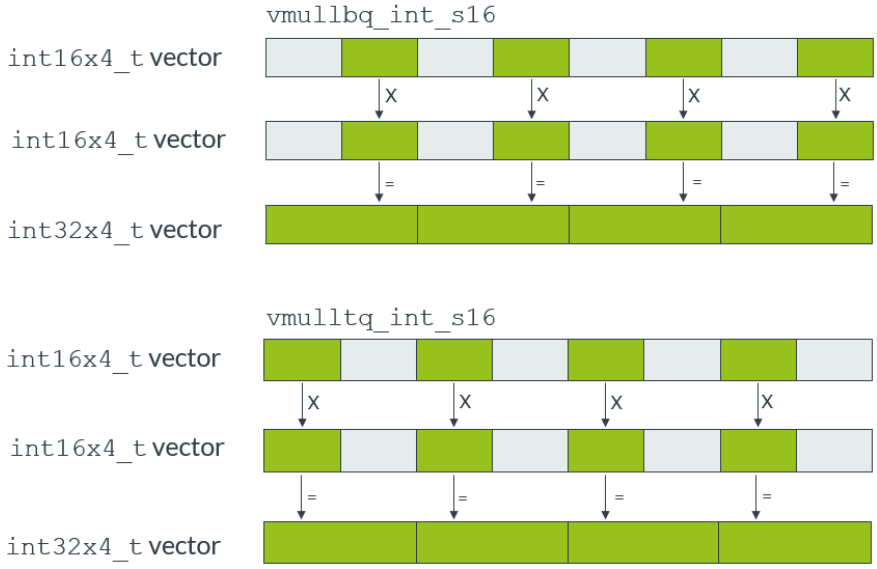
7.2 Optimized migration to Helium

Long multiplication is available on Helium, but operates differently from Neon.

Neon long multiplication operates on half-vectors (`int16x4_t`), as shown in the following diagram:



Helium long multiplication, however, operates on a full 128-bit vector selecting either the top parts (odd-indexed elements) or bottom parts (even-index elements). This process is shown in the following diagram:



Helium does not provide a direct equivalent for the `vpadalq_s32` Neon long pairwise addition intrinsic, but integer vectors have support for across operations like sum. Either a 32-bit or 64-bit general-purpose register can be used for accumulation. To align with the Neon implementation, we use `vaddlvaq_s32` to operate on an int32x4_t vector and sum in a 64-bit accumulator.

Implementing these substitutions allows an almost one-to-one Neon to Helium conversion. The following code shows this simple, direct conversion of the Neon implementation to Helium:

```
void arm_cmplx_dot_prod_helium_q15_direct_conversion(const q15_t * pSrcA,
                                                     const q15_t * pSrcB,
                                                     uint32_t numSamples,
                                                     q31_t * realResult,
                                                     q31_t * imagResult)
{
  uint32_t        blkCnt;     /* Loop counter */
  q63_t           real_sum = 0, imag_sum = 0; /* Temporary result variables */
  q15_t           a0, b0, c0, d0;

  int16x8x2_t     vec1, vec2;
  int32x4_t       tempL, tempH;
  int64_t         resr1, resr2, resi;
```

```
  resr1 = 0LL;
  resr2 = 0LL;
  resi = 0LL;

  /* loop Unrolling */

  blkCnt = numSamples >> 3U;

  while (blkCnt > 0U) {
    vec1 = vld2q_s16(pSrcA);
    vec2 = vld2q_s16(pSrcB);
    pSrcA += 16;
    pSrcB += 16;

    tempL = vmullbq_int_s16(vec1.val[0], vec2.val[0]);
    tempH = vmulltq_int_s16(vec1.val[0], vec2.val[0]);

    resr1 = vaddlvaq_s32(resr1, tempL);
    resr1 = vaddlvaq_s32(resr1, tempH);

    tempL = vmullbq_int_s16(vec1.val[1], vec2.val[1]);
    tempH = vmulltq_int_s16(vec1.val[1], vec2.val[0]);

    resr2 = vaddlvaq_s32(resr2, tempL);
    resr2 = vaddlvaq_s32(resr2, tempH);

    tempL = vmullbq_int_s16(vec1.val[0], vec2.val[1]);
    tempH = vmulltq_int_s16(vec1.val[0], vec2.val[1]);

    resi = vaddlvaq_s32(resi, tempL);
    resi = vaddlvaq_s32(resi, tempH);

    tempL = vmullbq_int_s16(vec1.val[1], vec2.val[0]);
    tempH = vmulltq_int_s16(vec1.val[1], vec2.val[0]);

    resi = vaddlvaq_s32(resi, tempL);
    resi = vaddlvaq_s32(resi, tempH);

    /* Decrement the blockSize loop counter */
    blkCnt--;
  }
  real_sum = resr1;
  real_sum -= resr2;
  imag_sum = resi;

  /* tail */
  blkCnt = numSamples & 0x7;

  while (blkCnt > 0U) {
    a0 = *pSrcA++;
    b0 = *pSrcA++;
    c0 = *pSrcB++;
    d0 = *pSrcB++;

    real_sum += (q31_t) a0 *c0;
    imag_sum += (q31_t) a0 *d0;
    real_sum -= (q31_t) b0 *d0;
    imag_sum += (q31_t) b0 *c0;

    /* Decrement loop counter */
    blkCnt--;
  }

  /* Store real and imaginary result in 8.24 format  */
```

```
  /* Convert real data in 34.30 to 8.24 by 6 right shifts */
  *realResult = (q31_t) (real_sum >> 6);
  /* Convert imaginary data in 34.30 to 8.24 by 6 right shifts */
  *imagResult = (q31_t) (imag_sum >> 6);
}
```

7.3 Optimized migration to Helium

Like Floating-point vector complex dot product, complex operations can be used to implement a more efficient Helium implementation.

Helium fixed-point complex multiplication is implemented using the `vmlsldav{a}q_s16` and `vmlaldav{a}xq_s16` intrinsics.

This optimized implementation produces a very compact implementation of the routine, as shown by the following code:

```
while (blkCnt > 0) {
  vec1 = vld1q(pSrcA);
  vec2 = vld1q(pSrcB);

  real_sum = vmlsldavaq_s16(real_sum, vec1, vec2);
  imag_sum = vmlaldavaxq_s16(imag_sum, vec1, vec2);

  pSrcA += 8;        pSrcB += 8;
  blkCnt--;
}
```

Further performance improvements are possible on dual beat cores such as Cortex-M55. By unrolling loops, the compiler can perfectly interleave vector loads and vector complex multiplications.

Note that Helium also provides 64-bit shift operators such as `asrl` which can be used to scale the final real and imaginary parts of the complex result to the final format. It is expected that future compilers will be able to use these instructions without requiring explicit intrinsics.

The following code shows a fully optimized Cortex-M55 version of the function:

```
void arm_cmplx_dot_prod_helium_q15(const q15_t * pSrcA,
                                   const q15_t * pSrcB,
                                   uint32_t numSamples,
                                   q31_t * realResult, q31_t * imagResult)
{
    int32_t         blkCnt;
    q63_t           accReal = 0LL;
    q63_t           accImag = 0LL;
    int16x8_t       vecSrcA, vecSrcB;
    int16x8_t       vecSrcC, vecSrcD;

    blkCnt = numSamples >> 3;
    blkCnt -= 1;
    if (blkCnt > 0) {
        /* should give more freedom to generate stall free code */
        vecSrcA = vld1q(pSrcA);
        vecSrcB = vld1q(pSrcB);
        pSrcA += 8;
        pSrcB += 8;

        while (blkCnt > 0U) {

            accReal = vmlsldavaq(accReal, vecSrcA, vecSrcB);
            vecSrcC = vld1q(pSrcA);
```

```
            pSrcA += 8;

            accImag = vmlaldavaxq(accImag, vecSrcA, vecSrcB);
            vecSrcD = vld1q(pSrcB);
            pSrcB += 8;

            accReal = vmlsldavaq(accReal, vecSrcC, vecSrcD);
            vecSrcA = vld1q(pSrcA);
            pSrcA += 8;

            accImag = vmlaldavaxq(accImag, vecSrcC, vecSrcD);
            vecSrcB = vld1q(pSrcB);
            pSrcB += 8;

            blkCnt--;
        }

        accReal = vmlsldavaq(accReal, vecSrcA, vecSrcB);
        vecSrcC = vld1q(pSrcA);

        accImag = vmlaldavaxq(accImag, vecSrcA, vecSrcB);
        vecSrcD = vld1q(pSrcB);

        accReal = vmlsldavaq(accReal, vecSrcC, vecSrcD);
        vecSrcA = vld1q(pSrcA);

        accImag = vmlaldavaxq(accImag, vecSrcC, vecSrcD);
        vecSrcB = vld1q(pSrcB);

        /*tail elided */
    }
    *realResult = asrl(accReal, (14 - 8));
    *imagResult = asrl(accImag, (14 - 8));
}
```

8 Single-precision 4x4 matrix multiplication

The `mat_multiply_4x4_neon` function uses intrinsics to multiply two single-precision floating point 4x4 matrices. The Neon implementation of this example function is described in the Neon Programmer's Guide for Armv8-A: Optimizing C Code with Neon Intrinsics

The matrices are multiplied together using the following underlying algorithm:

```
void matrix_multiply_c(float32_t *A, float32_t *B, float32_t *C, uint32_t n,
       uint32_t m, uint32_t k) {
  for (int i_idx=0; i_idx < n; i_idx++) {
    for (int j_idx=0; j_idx < m; j_idx++) {
      C[n*j_idx + i_idx] = 0;
      for (int k_idx=0; k_idx < k; k_idx++) {
        C[n*j_idx + i_idx] += A[n*k_idx + i_idx]*B[k*j_idx + k_idx];
      }
    }
  }
}
```

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/matrix.c

8.1 Neon implementation

The following code shows an implementation of a 4x4 single-precision floating-point vector multiply function using Neon intrinsics:

```
void mat_multiply_4x4_neon(float32_t * A, float32_t * B, float32_t * C)
{
  // these are the columns A
  float32x4_t     A0;
  float32x4_t     A1;
  float32x4_t     A2;
  float32x4_t     A3;

  // these are the columns B
  float32x4_t     B0;
  float32x4_t     B1;
  float32x4_t     B2;
  float32x4_t     B3;

  // these are the columns C
  float32x4_t     C0;
  float32x4_t     C1;
  float32x4_t     C2;
  float32x4_t     C3;

  A0 = vld1q_f32(A);
  A1 = vld1q_f32(A + 4);
  A2 = vld1q_f32(A + 8);
```

```
  A3 = vld1q_f32(A + 12);

  // Zero accumulators for C values
  C0 = vmovq_n_f32(0);
  C1 = vmovq_n_f32(0);
  C2 = vmovq_n_f32(0);
  C3 = vmovq_n_f32(0);

  // Multiply accumulate in 4x1 blocks, i.e. each column in C
  B0 = vld1q_f32(B);
  C0 = vfmaq_laneq_f32(C0, A0, B0, 0);
  C0 = vfmaq_laneq_f32(C0, A1, B0, 1);
  C0 = vfmaq_laneq_f32(C0, A2, B0, 2);
  C0 = vfmaq_laneq_f32(C0, A3, B0, 3);
  vst1q_f32(C, C0);

  B1 = vld1q_f32(B + 4);
  C1 = vfmaq_laneq_f32(C1, A0, B1, 0);
  C1 = vfmaq_laneq_f32(C1, A1, B1, 1);
  C1 = vfmaq_laneq_f32(C1, A2, B1, 2);
  C1 = vfmaq_laneq_f32(C1, A3, B1, 3);
  vst1q_f32(C + 4, C1);

  B2 = vld1q_f32(B + 8);
  C2 = vfmaq_laneq_f32(C2, A0, B2, 0);
  C2 = vfmaq_laneq_f32(C2, A1, B2, 1);
  C2 = vfmaq_laneq_f32(C2, A2, B2, 2);
  C2 = vfmaq_laneq_f32(C2, A3, B2, 3);
  vst1q_f32(C + 8, C2);

  B3 = vld1q_f32(B + 12);
  C3 = vfmaq_laneq_f32(C3, A0, B3, 0);
  C3 = vfmaq_laneq_f32(C3, A1, B3, 1);
  C3 = vfmaq_laneq_f32(C3, A2, B3, 2);
  C3 = vfmaq_laneq_f32(C3, A3, B3, 3);
  vst1q_f32(C + 12, C3);
}
```

The Neon code uses the `vld1q_f32` intrinsic to load contiguous data elements into vectors. These vectors represent columns in the source matrices. The code then uses `vfmaq_laneq_f32` intrinsics to multiply each column in turn by a single element from the other matrix. This operation is repeated until all columns have been multiplied and summed.

8.2 Direct migration to Helium

The `vfmaq_laneq_f32` intrinsic used by the Neon code multiplies a `float32x4_t` vector by a single element of another `float32x4_t` vector, and then accumulates the result in a third `float32x4_t`.

Helium does not provide instructions that mix vector registers and single lane value operands in this way. However, Helium does provide instructions that let you mix vector and general-purpose registers.

The direct conversion from Helium to Neon is therefore to mimic the Neon `vfmaq_laneq_f32` intrinsic using the Helium `vfmaq_n_f32` and `vgetq_lane_f32` intrinsics, for example:

```
#define vfmaq_laneq_emu_f32(acc, A, B, idx)
          vfmaq_n_f32(acc, A, vgetq_lane_f32(B, idx))
```

The Neon `vmovq_n_f32` intrinsic can be directly substituted with the Helium `vdupq_n_f32` intrinsic, for example:

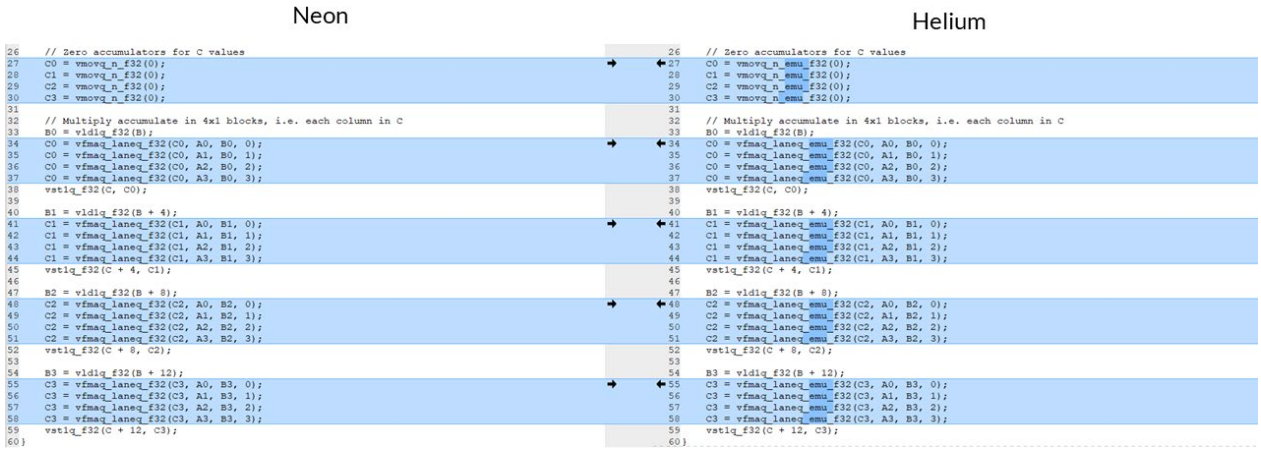
```
#define vmovq_n_emu_f32         vdupq_n_f32
```

The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
void mat_multiply_4x4_helium_direct(float32_t * A, float32_t * B, float32_t * C)
{
    // these are the columns A
    float32x4_t     A0;
    float32x4_t     A1;
    float32x4_t     A2;
    float32x4_t     A3;

    // these are the columns B
    float32x4_t     B0;
    float32x4_t     B1;
    float32x4_t     B2;
    float32x4_t     B3;

    // these are the columns C
    float32x4_t     C0;
    float32x4_t     C1;
    float32x4_t     C2;
    float32x4_t     C3;

    A0 = vld1q_f32(A);
    A1 = vld1q_f32(A + 4);
    A2 = vld1q_f32(A + 8);
    A3 = vld1q_f32(A + 12);

    // Zero accumulators for C values
    C0 = vmovq_n_emu_f32(0);
    C1 = vmovq_n_emu_f32(0);
    C2 = vmovq_n_emu_f32(0);
    C3 = vmovq_n_emu_f32(0);

    // Multiply accumulate in 4x1 blocks, i.e. each column in C
    B0 = vld1q_f32(B);
    C0 = vfmaq_laneq_emu_f32(C0, A0, B0, 0);
    C0 = vfmaq_laneq_emu_f32(C0, A1, B0, 1);
    C0 = vfmaq_laneq_emu_f32(C0, A2, B0, 2);
    C0 = vfmaq_laneq_emu_f32(C0, A3, B0, 3);
    vst1q_f32(C, C0);

    B1 = vld1q_f32(B + 4);
    C1 = vfmaq_laneq_emu_f32(C1, A0, B1, 0);
    C1 = vfmaq_laneq_emu_f32(C1, A1, B1, 1);
    C1 = vfmaq_laneq_emu_f32(C1, A2, B1, 2);
    C1 = vfmaq_laneq_emu_f32(C1, A3, B1, 3);
    vst1q_f32(C + 4, C1);

    B2 = vld1q_f32(B + 8);
    C2 = vfmaq_laneq_emu_f32(C2, A0, B2, 0);
    C2 = vfmaq_laneq_emu_f32(C2, A1, B2, 1);
    C2 = vfmaq_laneq_emu_f32(C2, A2, B2, 2);
    C2 = vfmaq_laneq_emu_f32(C2, A3, B2, 3);
    vst1q_f32(C + 8, C2);

    B3 = vld1q_f32(B + 12);
    C3 = vfmaq_laneq_emu_f32(C3, A0, B3, 0);
    C3 = vfmaq_laneq_emu_f32(C3, A1, B3, 1);
    C3 = vfmaq_laneq_emu_f32(C3, A2, B3, 2);
```

```
    C3 = vfmaq_laneq_emu_f32(C3, A3, B3, 3);
    vst1q_f32(C + 12, C3);
}
```

The following image shows the major differences between the Neon implementation and the direct Helium implementation:



8.3 Optimized migration to Helium

The direct conversion of `vfmaq_laneq_f32` to `vfmaq_n_f32` and `vgetq_lane_f32` intrinsics requires additional moves from vector registers to scalar registers.

An optimized implementation can use scalar loads to directly read scalar elements using scalar loads.

The following code shows an optimized core loop that uses these scalar loads:

```
void mat_multiply_4x4_helium(float32_t * A, float32_t * B, float32_t * C)
{
    float32_t const *pSrBVec;
    float32_t *pInB = B;
    float32_t *pInA = A;
    float32_t *pOut = C;
    float32_t *pInA0, *pInA1, *pInA2, *pInA3;
    float32x4_t vecMac0, vecMac1, vecMac2, vecMac3;
    float32x4_t vecInB;
    const uint32_t MATRIX_DIM = 4;

    pSrBVec = (float32_t const *) pInB;

    pInA0 = pInA;
    pInA1 = pInA0 + MATRIX_DIM;
    pInA2 = pInA1 + MATRIX_DIM;
    pInA3 = pInA2 + MATRIX_DIM;
    /*
     * load {b0,0, b0,1, b0,2, b0,3}
     */
    vecInB = vld1q(pSrBVec);  pSrBVec += MATRIX_DIM;

    vecMac0 = vmulq(vecInB, *pInA0++);
    vecMac1 = vmulq(vecInB, *pInA1++);
    vecMac2 = vmulq(vecInB, *pInA2++);
    vecMac3 = vmulq(vecInB, *pInA3++);
    /*
```

```
     * load {b1,0, b1,1, b1,2, b1,3}
     */
    vecInB = vld1q(pSrBVec);  pSrBVec += MATRIX_DIM;

    vecMac0 = vfmaq(vecMac0, vecInB, *pInA0++);
    vecMac1 = vfmaq(vecMac1, vecInB, *pInA1++);
    vecMac2 = vfmaq(vecMac2, vecInB, *pInA2++);
    vecMac3 = vfmaq(vecMac3, vecInB, *pInA3++);
    /*
     * load {b2,0, b2,1, b2,2, b2,3}
     */
    vecInB = vld1q(pSrBVec);  pSrBVec += MATRIX_DIM;

    vecMac0 = vfmaq(vecMac0, vecInB, *pInA0++);
    vecMac1 = vfmaq(vecMac1, vecInB, *pInA1++);
    vecMac2 = vfmaq(vecMac2, vecInB, *pInA2++);
    vecMac3 = vfmaq(vecMac3, vecInB, *pInA3++);
    /*
     * load {b3,0, b3,1, b3,2, b3,3}
     */
    vecInB = vld1q(pSrBVec);  pSrBVec += MATRIX_DIM;

    vecMac0 = vfmaq(vecMac0, vecInB, *pInA0++);
    vecMac1 = vfmaq(vecMac1, vecInB, *pInA1++);
    vecMac2 = vfmaq(vecMac2, vecInB, *pInA2++);
    vecMac3 = vfmaq(vecMac3, vecInB, *pInA3++);

    vst1q(pOut, vecMac0);  pOut += MATRIX_DIM;
    vst1q(pOut, vecMac1);  pOut += MATRIX_DIM;
    vst1q(pOut, vecMac2);  pOut += MATRIX_DIM;
    vst1q(pOut, vecMac3);
}
```

This example uses `vfmaq`, the polymorphic implementation of the `vfmaq_n_f32` intrinsic. Helium provides polymorphic implementations of most intrinsics. The polymorphic name of an intrinsic is indicated by leaving out the type suffix, leading to a more concise syntax.

For example:

```
vfmaq_n_f32(vecMac0, vecInB, *pInA0++)
```

is equivalent to:

```
vfmaq(vecMac0, vecInB, *pInA0++)
```

9 Fixed-point 16-bit cross-correlation

This section of the guide examines the implementation of cross-correlation in the Opus CELT algorithm.

Cross-correlation is a mathematical measure of the similarity between two vectors.

Opus is a codec for interactive speech and audio transmission over the Internet developed by the Xiph.Org Foundation. Opus uses the Constrained Energy Lapped Transform (CELT) algorithm to compress audio.

The `xcorr_kernel_neon_fixed` function uses Neon intrinsics to perform cross-correlation on two vectors.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/opus_xcorr.c

9.1 Neon implementation

The following code shows an implementation of a 16-bit fixed-point cross-correlation function using Neon intrinsics:

```
void xcorr_kernel_neon_fixed(const opus_val16 * x, const opus_val16 * y,
                             opus_val32 sum[4], int len)
{
  int              j;
  int32x4_t        a = vld1q_s32(sum);
  /* Load y[0...3] */
  /* This requires len>0 to always be valid (which we assert in the C code). */
  int16x4_t        y0 = vld1_s16(y);
  y += 4;

  for (j = 0; j + 8 <= len; j += 8) {
    /* Load x[0...7] */
    int16x8_t        xx = vld1q_s16(x);
    int16x4_t        x0 = vget_low_s16(xx);
    int16x4_t        x4 = vget_high_s16(xx);
    /* Load y[4...11] */
    int16x8_t        yy = vld1q_s16(y);
    int16x4_t        y4 = vget_low_s16(yy);
    int16x4_t        y8 = vget_high_s16(yy);
    int32x4_t        a0 = vmlal_lane_s16(a, y0, x0, 0);
    int32x4_t        a1 = vmlal_lane_s16(a0, y4, x4, 0);

    int16x4_t        y1 = vext_s16(y0, y4, 1);
    int16x4_t        y5 = vext_s16(y4, y8, 1);
    int32x4_t        a2 = vmlal_lane_s16(a1, y1, x0, 1);
    int32x4_t        a3 = vmlal_lane_s16(a2, y5, x4, 1);

    int16x4_t        y2 = vext_s16(y0, y4, 2);
```

```
    int16x4_t       y6 = vext_s16(y4, y8, 2);
    int32x4_t       a4 = vmlal_lane_s16(a3, y2, x0, 2);
    int32x4_t       a5 = vmlal_lane_s16(a4, y6, x4, 2);

    int16x4_t       y3 = vext_s16(y0, y4, 3);
    int16x4_t       y7 = vext_s16(y4, y8, 3);
    int32x4_t       a6 = vmlal_lane_s16(a5, y3, x0, 3);
    int32x4_t       a7 = vmlal_lane_s16(a6, y7, x4, 3);

    y0 = y8;
    a = a7;
    x += 8;
    y += 8;
  }

  for (; j < len; j++) {
    int16x4_t       x0 = vld1_dup_s16(x);   /* load next x */
    int32x4_t       a0 = vmlal_s16(a, y0, x0);

    int16x4_t       y4 = vld1_dup_s16(y);   /* load next y */
    y0 = vext_s16(y0, y4, 1);
    a = a0;
    x++;
    y++;
  }

  vst1q_s32(sum, a);
}
```

9.2 Direct migration to Helium

As we see in this section of the guide, direct migration by simply replacing Neon intrinsics with Helium counterparts is not the recommended approach. The resulting Helium code would miss important optimization features and perform poorly. However, direct migration allows us to quickly get Neon code running on a Helium processor as an initial prototyping stage.

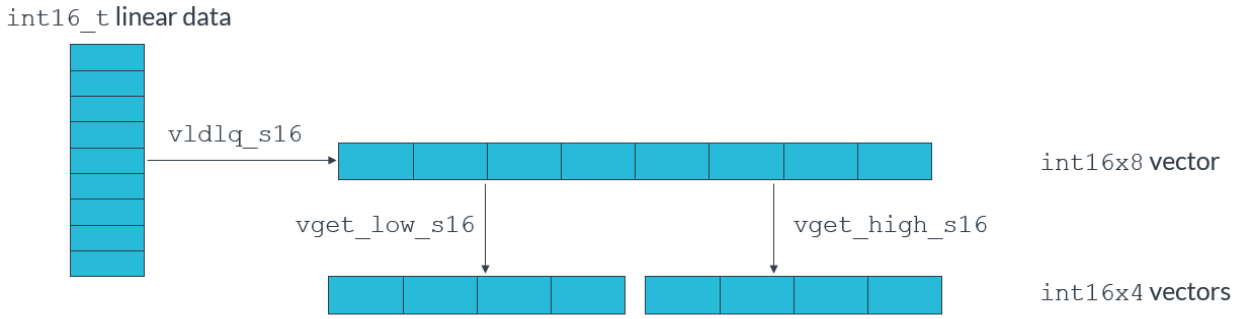
Examining the Neon implementation of the cross-correlation function reveals several obstacles to Helium migration. The Neon code uses 64-bit `int16x4_t` vectors and associated instructions including `vget_low_s16`, `vget_high_s16`, `vext_s16`, and `vmlal_lane_s16` which do not exist in Helium.

Because Helium does not provide 64-bit half vectors, the migrated code uses full 128-bit vectors but only considers the lower half of each element. These emulated half vectors are filled using vector load with widening. The widened size is double the size of the loaded vector element. For example, an `int8x8_t` vector can be loaded with a 16-bit widened byte load. An `int16x4_t` vector can be loaded with a 32-bit widened short load.

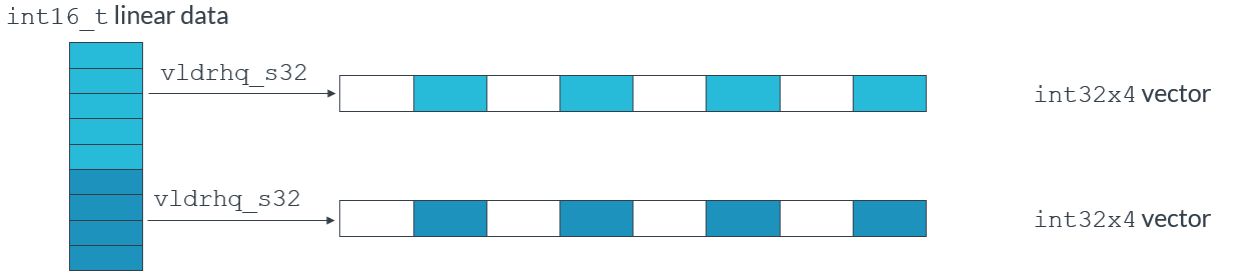

Note

Helium does not support 64-bit widening.

The Neon code uses `vld1q_s16` with `vget_low_s16` and `vget_high_s16` to load linear 16-bit data into `int16x4_t` vectors, as show in the following diagram:



With Helium, we can use the `vldrhq_s32` widening load intrinsic to load four 16-bit elements into the lower halves of a 4x32 128-bit vector, as show in the following diagram. The upper halves contain the sign bits.



The Neon code uses the `vmlal_lane_s16` intrinsic to perform a long multiply-accumulate with the scalar multiplier residing in a single lane.

In the migrated Helium code, the vectors have already been widened from `int16_t` elements into an `int32x4_t` vector, so no further widening is needed. The multiply-accumulate operation is emulated with `vmlaq_n_s32` and `vgetq_lane` intrinsics as follows:

```
#define vmlal_lane_emu_s16(a, b, c, idx) vmlaq_n_s32(a, b, vgetq_lane(c, idx))
```

The Neon code uses `vext_s16` to merge two vectors after applying an immediate offset. This is a common operation in Neon code. For more background information, see Coding for Neon - Part 5: Rearranging Vectors.

Helium does not provide a direct equivalent for the Neon `vext_s16` intrinsic. The easiest way to support an equivalent operation without refactoring the entire algorithm, is to directly permute the lanes using VMOV operations.

If your target supports floating-point, the compiler issues S or D register moves. However, if floating-point is not supported then lane moves use the general-purpose registers.

Both Arm Compiler and GCC support the `__builtin_shufflevector` function which provides a useful way to shuffle a vector pair and create a destination vector with immediate indexes.

The `__builtin_shufflevector` function has the following syntax:

```
__builtin_shufflevector(vec1, vec2, index1, index2, ...)
```

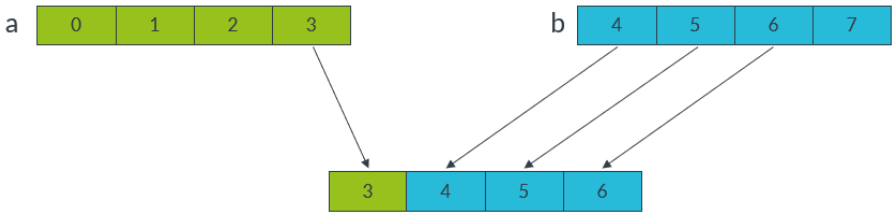

Note

For more information about the `__builtin_shufflevector` function, see Clang Language Extensions.

We can use the `__builtin_shufflevector` function to emulate the `vext_s16` intrinsic as follows:

```
vext_emu_s16 (a, b, 1) __builtin_shufflevector(a, b, 1, 2, 3, 4);
vext_emu_s16 (a, b, 2) __builtin_shufflevector(a, b, 2, 3, 4, 5);
vext_emu_s16 (a, b, 3) __builtin_shufflevector(a, b, 3, 4, 5, 6);
```

```
__builtin_shufflevector(a, b, 3, 4, 5, 6)
```



The following code shows a simple, direct conversion of the Neon implementation to Helium:

```
void xcorr_kernel_mve_fixed_direct(const opus_val16 * x, const opus_val16 * y,
     opus_val32 sum[4], int len)
{
  int              j;
  int32x4_t        a = vld1q_s32(sum);
  /* Load y[0...3] */
  /* This requires len>0 to always be valid (which we assert in the C code). */
  int32x4_t        y0 = vldrhq_s32(y);
  y += 4;

  for (j = 0; j + 8 <= len; j += 8) {
    /* Load x[0...7] */
    int32x4_t        x0 = vldrhq_s32(x);
    int32x4_t        x4 = vldrhq_s32(x + 4);
    /* Load y[4...11] */
    int32x4_t        y4 = vldrhq_s32(y);
    int32x4_t        y8 = vldrhq_s32(y + 4);
    int32x4_t        a0 = vmlaq_n_s32(a, y0, vgetq_lane(x0, 0));
    int32x4_t        a1 = vmlaq_n_s32(a0, y4, vgetq_lane(x4, 0));

    int32x4_t        y1 = vext_emu_s16(y0, y4, 1);
    int32x4_t        y5 = vext_emu_s16(y4, y8, 1);
    int32x4_t        a2 = vmlaq_n_s32(a1, y1, vgetq_lane(x0, 1));
    int32x4_t        a3 = vmlaq_n_s32(a2, y5, vgetq_lane(x4, 1));

    int32x4_t        y2 = vext_emu_s16(y0, y4, 2);
    int32x4_t        y6 = vext_emu_s16(y4, y8, 2);
    int32x4_t        a4 = vmlaq_n_s32(a3, y2, vgetq_lane(x0, 2));
    int32x4_t        a5 = vmlaq_n_s32(a4, y6, vgetq_lane(x4, 2));

    int32x4_t        y3 = vext_emu_s16(y0, y4, 3);
    int32x4_t        y7 = vext_emu_s16(y4, y8, 3);
    int32x4_t        a6 = vmlaq_n_s32(a5, y3, vgetq_lane(x0, 3));
    int32x4_t        a7 = vmlaq_n_s32(a6, y7, vgetq_lane(x4, 3));
```

```
    y0 = y8;
    a = a7;
    x += 8;
    y += 8;
  }

  for (; j < len; j++) {
    int16_t          x0 = *x;         /* load next x */
    int32x4_t        a0 = vmlaq_n_s32(a, y0, (int32_t) x0);

    y0 = vldrhq_s32(y - 3); /* load next y */
    a = a0;
    x++;
    y++;
  }

  vst1q_s32(sum, a);
}
```

The following image shows the major differences between the Neon implementation and the direct Helium implementation:

The following GNU diff output shows the same differences in text form:

```
--- neon.c      Thu Nov 12 10:38:06 2020
+++ helium_direct.c     Thu Nov 12 12:01:31 2020
@@ -1,39 +1,37 @@
-void xcorr_kernel_neon_fixed(const opus_val16 * x, const opus_val16 * y,
-                             opus_val32 sum[4], int len)
+void xcorr_kernel_mve_fixed_direct(const opus_val16 * x, const opus_val16 * y,
+                            opus_val32 sum[4], int len)
 {
     int              j;
     int32x4_t        a = vld1q_s32(sum);
     /* Load y[0...3] */
     /* This requires len>0 to always be valid (which we assert in the C code). */
-    int16x4_t        y0 = vld1_s16(y);
+    int32x4_t        y0 = vldrhq_s32(y);
     y += 4;

     for (j = 0; j + 8 <= len; j += 8) {
         /* Load x[0...7] */
-        int16x8_t        xx = vld1q_s16(x);
-        int16x4_t        x0 = vget_low_s16(xx);
-        int16x4_t        x4 = vget_high_s16(xx);
+        int32x4_t        x0 = vldrhq_s32(x);
+        int32x4_t        x4 = vldrhq_s32(x + 4);
         /* Load y[4...11] */
-        int16x8_t        yy = vld1q_s16(y);
-        int16x4_t        y4 = vget_low_s16(yy);
-        int16x4_t        y8 = vget_high_s16(yy);
-        int32x4_t        a0 = vmlal_lane_s16(a, y0, x0, 0);
-        int32x4_t        a1 = vmlal_lane_s16(a0, y4, x4, 0);
+        int32x4_t        y4 = vldrhq_s32(y);
+        int32x4_t        y8 = vldrhq_s32(y + 4);
+        int32x4_t        a0 = vmlaq_n_s32(a, y0, vgetq_lane(x0, 0));
+        int32x4_t        a1 = vmlaq_n_s32(a0, y4, vgetq_lane(x4, 0));

-        int16x4_t        y1 = vext_s16(y0, y4, 1);
-        int16x4_t        y5 = vext_s16(y4, y8, 1);
-        int32x4_t        a2 = vmlal_lane_s16(a1, y1, x0, 1);
-        int32x4_t        a3 = vmlal_lane_s16(a2, y5, x4, 1);
+        int32x4_t        y1 = vext_emu_s16(y0, y4, 1);
+        int32x4_t        y5 = vext_emu_s16(y4, y8, 1);
+        int32x4_t        a2 = vmlaq_n_s32(a1, y1, vgetq_lane(x0, 1));
+        int32x4_t        a3 = vmlaq_n_s32(a2, y5, vgetq_lane(x4, 1));

-        int16x4_t        y2 = vext_s16(y0, y4, 2);
-        int16x4_t        y6 = vext_s16(y4, y8, 2);
-        int32x4_t        a4 = vmlal_lane_s16(a3, y2, x0, 2);
-        int32x4_t        a5 = vmlal_lane_s16(a4, y6, x4, 2);
+        int32x4_t        y2 = vext_emu_s16(y0, y4, 2);
+        int32x4_t        y6 = vext_emu_s16(y4, y8, 2);
+        int32x4_t        a4 = vmlaq_n_s32(a3, y2, vgetq_lane(x0, 2));
+        int32x4_t        a5 = vmlaq_n_s32(a4, y6, vgetq_lane(x4, 2));

-        int16x4_t        y3 = vext_s16(y0, y4, 3);
-        int16x4_t        y7 = vext_s16(y4, y8, 3);
-        int32x4_t        a6 = vmlal_lane_s16(a5, y3, x0, 3);
-        int32x4_t        a7 = vmlal_lane_s16(a6, y7, x4, 3);
+        int32x4_t        y3 = vext_emu_s16(y0, y4, 3);
+        int32x4_t        y7 = vext_emu_s16(y4, y8, 3);
+        int32x4_t        a6 = vmlaq_n_s32(a5, y3, vgetq_lane(x0, 3));
+        int32x4_t        a7 = vmlaq_n_s32(a6, y7, vgetq_lane(x4, 3));

         y0 = y8;
         a = a7;
         x += 8;
         y += 8;
     }

     for (; j < len; j++) {
-        int16x4_t        x0 = vld1_dup_s16(x);   /* load next x */
-        int32x4_t        a0 = vmlal_s16(a, y0, x0);
+        int16_t          x0 = *x;        /* load next x */
+        int32x4_t        a0 = vmlaq_n_s32(a, y0, (int32_t) x0);

-        int16x4_t        y4 = vld1_dup_s16(y);   /* load next y */
-        y0 = vext_s16(y0, y4, 1);
+        y0 = vldrhq_s32(y - 3); /* load next y */
+        a = a0;
+        x++;
+        y++;
     }

     vst1q_s32(sum, a);
 }
```

```
     for (j = 0; j + 8 <= len; j += 8) {
         /* Load x[0...7] */
-        int16x8_t       xx = vld1q_s16(x);
-        int16x4_t       x0 = vget_low_s16(xx);
-        int16x4_t       x4 = vget_high_s16(xx);
+        int32x4_t       x0 = vldrhq_s32(x);
+        int32x4_t       x4 = vldrhq_s32(x + 4);
         /* Load y[4...11] */
-        int16x8_t       yy = vld1q_s16(y);
-        int16x4_t       y4 = vget_low_s16(yy);
-        int16x4_t       y8 = vget_high_s16(yy);
-        int32x4_t       a0 = vmlal_lane_s16(a, y0, x0, 0);
-        int32x4_t       a1 = vmlal_lane_s16(a0, y4, x4, 0);
+        int32x4_t       y4 = vldrhq_s32(y);
+        int32x4_t       y8 = vldrhq_s32(y + 4);
+        int32x4_t       a0 = vmlaq_n_s32(a, y0, vgetq_lane(x0, 0));
+        int32x4_t       a1 = vmlaq_n_s32(a0, y4, vgetq_lane(x4, 0));

-        int16x4_t       y1 = vext_s16(y0, y4, 1);
-        int16x4_t       y5 = vext_s16(y4, y8, 1);
-        int32x4_t       a2 = vmlal_lane_s16(a1, y1, x0, 1);
-        int32x4_t       a3 = vmlal_lane_s16(a2, y5, x4, 1);
+        int32x4_t       y1 = vext_emu_s16(y0, y4, 1);
+        int32x4_t       y5 = vext_emu_s16(y4, y8, 1);
+        int32x4_t       a2 = vmlaq_n_s32(a1, y1, vgetq_lane(x0, 1));
+        int32x4_t       a3 = vmlaq_n_s32(a2, y5, vgetq_lane(x4, 1));

-        int16x4_t       y2 = vext_s16(y0, y4, 2);
-        int16x4_t       y6 = vext_s16(y4, y8, 2);
-        int32x4_t       a4 = vmlal_lane_s16(a3, y2, x0, 2);
-        int32x4_t       a5 = vmlal_lane_s16(a4, y6, x4, 2);
+        int32x4_t       y2 = vext_emu_s16(y0, y4, 2);
+        int32x4_t       y6 = vext_emu_s16(y4, y8, 2);
+        int32x4_t       a4 = vmlaq_n_s32(a3, y2, vgetq_lane(x0, 2));
+        int32x4_t       a5 = vmlaq_n_s32(a4, y6, vgetq_lane(x4, 2));

-        int16x4_t       y3 = vext_s16(y0, y4, 3);
-        int16x4_t       y7 = vext_s16(y4, y8, 3);
-        int32x4_t       a6 = vmlal_lane_s16(a5, y3, x0, 3);
-        int32x4_t       a7 = vmlal_lane_s16(a6, y7, x4, 3);
+        int32x4_t       y3 = vext_emu_s16(y0, y4, 3);
+        int32x4_t       y7 = vext_emu_s16(y4, y8, 3);
+        int32x4_t       a6 = vmlaq_n_s32(a5, y3, vgetq_lane(x0, 3));
+        int32x4_t       a7 = vmlaq_n_s32(a6, y7, vgetq_lane(x4, 3));

         y0 = y8;
         a = a7;
@@ -42,11 +40,10 @@
     }

     for (; j < len; j++) {
-        int16x4_t       x0 = vld1_dup_s16(x);   /* load next x */
-        int32x4_t       a0 = vmlal_s16(a, y0, x0);
+        int16_t         x0 = *x;        /* load next x */
+        int32x4_t       a0 = vmlaq_n_s32(a, y0, (int32_t) x0);

-        int16x4_t       y4 = vld1_dup_s16(y);   /* load next y */
-        y0 = vext_s16(y0, y4, 1);
+        y0 = vldrhq_s32(y - 3); /* load next y */
         a = a0;
         x++;
         y++;
```

9.3 Optimized migration to Helium

If we disassemble and examine the assembly code that the compiler produces, we see that the `vext` emulation used in the direct migration is costly in terms of VMOV operations. Arm Compiler generates the following code with the `-mcpu=cortex-m55` option:

```
      dls             lr, lr
1:
      vldrh.s32       q4, [r0], #16
      vmov            r12, s16
      vmla.u32        q0, q1, r12
      vldrh.s32       q2, [r0, #-8]
      vmov.f32        s20, s5
      vmov            r12, s8
      vldrh.s32       q3, [r1], #16
      vmla.u32        q0, q3, r12
      vmov.f32        s21, s6
      vmov.f32        s22, s7
      vmov            r12, s17
      vmov.f32        s23, s12
      vmla.u32        q0, q5, r12
      vmov.f64        d10, d3
      vmov.f32        s21, s7
      vmov.f32        s22, s12
      vmov.f32        s4, s7
      vmov            r12, s18
      vmov.f32        s23, s13
      vmov.f32        s5, s12
      vmla.u32        q0, q5, r12
      vmov.f32        s6, s13
      vmov            r12, s19
      vmov.f32        s7, s14
      // etc...
      lr, #1b
```

If we examine how `vext` is used in the code, we see a successive series of `vext` instructions. This series of `vext` instructions extract 64-bit sub-vectors from 128-bit vectors with incremental offsets of 1, 2 and 3. This implementation minimizes the number of vector load operations, as shown in the following Neon code fragment:

```
    int16x4_t       y1 = vext_s16(y0, y4, 1);
    int16x4_t       y5 = vext_s16(y4, y8, 1);
    int32x4_t       a2 = vmlal_lane_s16(a1, y1, x0, 1);
    int32x4_t       a3 = vmlal_lane_s16(a2, y5, x4, 1);

    int16x4_t       y2 = vext_s16(y0, y4, 2);
    int16x4_t       y6 = vext_s16(y4, y8, 2);
    int32x4_t       a4 = vmlal_lane_s16(a3, y2, x0, 2);
    int32x4_t       a5 = vmlal_lane_s16(a4, y6, x4, 2);

    int16x4_t       y3 = vext_s16(y0, y4, 3);
    int16x4_t       y7 = vext_s16(y4, y8, 3);
    int32x4_t       a6 = vmlal_lane_s16(a5, y3, x0, 3);
    int32x4_t       a7 = vmlal_lane_s16(a6, y7, x4, 3);
```

We can more efficiently code this using a successive vector load with an incrementing base address. This code is more efficient because Helium processors can overlap vector load and vector data operations. The following Helium code shows the optimized implementation:

```
    int32x4_t       y1 = vldrhq_s32(y - 3);
    int32x4_t       y5 = vldrhq_s32(y + 1);
    int32x4_t       a2 = vmlaq_n_s32(a1, y1, vgetq_lane(x0, 1));
```

```
    int32x4_t       a3 = vmlaq_n_s32(a2, y5, vgetq_lane(x4, 1));

    int32x4_t       y2 = vldrhq_s32(y - 2);
    int32x4_t       y6 = vldrhq_s32(y + 2);
    int32x4_t       a4 = vmlaq_n_s32(a3, y2, vgetq_lane(x0, 2));
    int32x4_t       a5 = vmlaq_n_s32(a4, y6, vgetq_lane(x4, 2));

    int32x4_t       y3 = vldrhq_s32(y - 1);
    int32x4_t       y7 = vldrhq_s32(y + 3);
    int32x4_t       a6 = vmlaq_n_s32(a5, y3, vgetq_lane(x0, 3));
    int32x4_t       a7 = vmlaq_n_s32(a6, y7, vgetq_lane(x4, 3));
```

This optimized code gives significantly better performance than the version that uses the emulated `vext` operation.

Thinking again about what the code is doing, we notice that the algorithm multiply-accumulates the X vector with four different shifted versions of the Y vector. This is called the cross-correlation sliding dot-product. We can use the Helium integer dot-product intrinsics to further optimize this portion of the algorithm.

Several variants of the Helium integer dot-product intrinsics are available. In this situation, `vmladavaq_s16`, would be a good fit. Because `vmladavaq_s16` takes 16-bit signed integer input vectors and accumulating in a 32-bit scalar register, it aligns with the original Neon implementation.


Note

Helium also provides 64-bit accumulator variants of the integer dot-product intrinsics.

The following code shows how the Helium main loop can be rewritten using the `vmladavaq_s16` intrinsic:

```
  for (j = 0; j + 8 <= len; j += 8) {
    /* Load x[0...7] */
    int16x8_t       x0 = vld1q(x);
    /* Load y[0...7] */
    int16x8_t       y0 = vld1q(y);
    int16x8_t       y1 = vld1q(y + 1);
    int16x8_t       y2 = vld1q(y + 2);
    int16x8_t       y3 = vld1q(y + 3);

    sum[0] = vmladavaq_s16(sum[0], x0, y0);
    sum[1] = vmladavaq_s16(sum[1], x0, y1);
    sum[2] = vmladavaq_s16(sum[2], x0, y2);
    sum[3] = vmladavaq_s16(sum[3], x0, y3);

    x += 8;
    y += 8;
  }
```

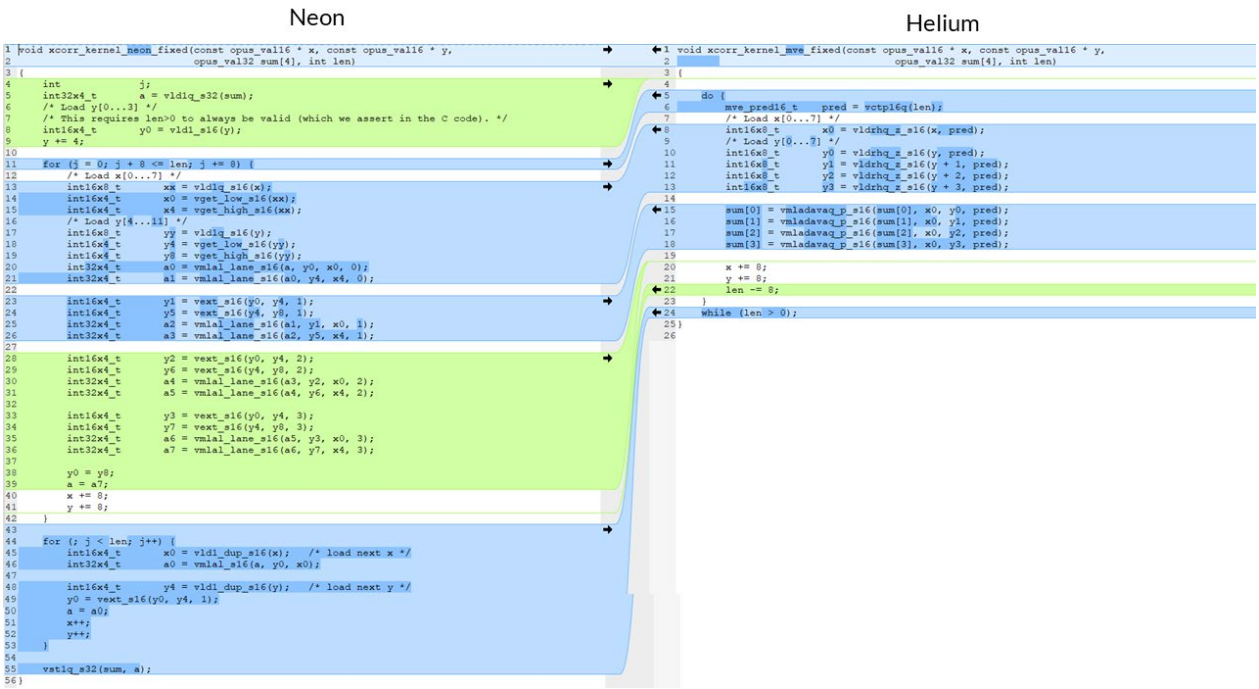
This optimized code does not need to perform any widening, and the loop can directly operate on full `int16x8_t` vectors. These optimizations give another significant performance boost compared to the direct migration.

The final version of our optimized Helium code uses all the optimizations described in this section. In addition, we can use predicated intrinsics to avoid performing tail handling. The final result is a compact implementation which is still fully aligned with the original Neon code behavior.

The following code shows the final version of the optimized Helium code:

```
void xcorr_kernel_mve_fixed(const opus_val16 * x, const opus_val16 * y,
                                    opus_val32 sum[4], int len)
{
  do {
    mve_pred16_t    pred = vctp16q(len);
    /* Load x[0...7] */
    int16x8_t       x0 = vldrhq_z_s16(x, pred);
    /* Load y[0...7] */
    int16x8_t       y0 = vldrhq_z_s16(y, pred);
    int16x8_t       y1 = vldrhq_z_s16(y + 1, pred);
    int16x8_t       y2 = vldrhq_z_s16(y + 2, pred);
    int16x8_t       y3 = vldrhq_z_s16(y + 3, pred);

    sum[0] = vmladavaq_p_s16(sum[0], x0, y0, pred);
    sum[1] = vmladavaq_p_s16(sum[1], x0, y1, pred);
    sum[2] = vmladavaq_p_s16(sum[2], x0, y2, pred);
    sum[3] = vmladavaq_p_s16(sum[3], x0, y3, pred);

    x += 8;
    y += 8;
    len -= 8;
  }
  while (len > 0);
}
```

The following image shows the differences between the Neon implementation and the optimized Helium implementation:



The following GNU diff output shows the same differences in text form:

```
--- neon.c      Thu Nov 12 10:38:06 2020
+++ helium_optimized.c  Thu Nov 12 16:14:07 2020
@@ -1,57 +1,26 @@
-void xcorr_kernel_neon_fixed(const opus_val16 * x, const opus_val16 * y,
-                             opus_val32 sum[4], int len)
+void xcorr_kernel_mve_fixed(const opus_val16 * x, const opus_val16 * y,
```

```
+                                    opus_val32 sum[4], int len)
 {
-    int             j;
-    int32x4_t       a = vld1q_s32(sum);
-    /* Load y[0...3] */
-    /* This requires len>0 to always be valid (which we assert in the C code). */
-    int16x4_t       y0 = vld1_s16(y);
-    y += 4;

-    for (j = 0; j + 8 <= len; j += 8) {
+    do {
+        mve_pred16_t    pred = vctp16q(len);
         /* Load x[0...7] */
-        int16x8_t       xx = vld1q_s16(x);
-        int16x4_t       x0 = vget_low_s16(xx);
-        int16x4_t       x4 = vget_high_s16(xx);
-        /* Load y[4...11] */
-        int16x8_t       yy = vld1q_s16(y);
-        int16x4_t       y4 = vget_low_s16(yy);
-        int16x4_t       y8 = vget_high_s16(yy);
-        int32x4_t       a0 = vmlal_lane_s16(a, y0, x0, 0);
-        int32x4_t       a1 = vmlal_lane_s16(a0, y4, x4, 0);
+        int16x8_t       x0 = vldrhq_z_s16(x, pred);
+        /* Load y[0...7] */
+        int16x8_t       y0 = vldrhq_z_s16(y, pred);
+        int16x8_t       y1 = vldrhq_z_s16(y + 1, pred);
+        int16x8_t       y2 = vldrhq_z_s16(y + 2, pred);
+        int16x8_t       y3 = vldrhq_z_s16(y + 3, pred);

-        int16x4_t       y1 = vext_s16(y0, y4, 1);
-        int16x4_t       y5 = vext_s16(y4, y8, 1);
-        int32x4_t       a2 = vmlal_lane_s16(a1, y1, x0, 1);
-        int32x4_t       a3 = vmlal_lane_s16(a2, y5, x4, 1);
+        sum[0] = vmladavaq_p_s16(sum[0], x0, y0, pred);
+        sum[1] = vmladavaq_p_s16(sum[1], x0, y1, pred);
+        sum[2] = vmladavaq_p_s16(sum[2], x0, y2, pred);
+        sum[3] = vmladavaq_p_s16(sum[3], x0, y3, pred);

-        int16x4_t       y2 = vext_s16(y0, y4, 2);
-        int16x4_t       y6 = vext_s16(y4, y8, 2);
-        int32x4_t       a4 = vmlal_lane_s16(a3, y2, x0, 2);
-        int32x4_t       a5 = vmlal_lane_s16(a4, y6, x4, 2);
-
-        int16x4_t       y3 = vext_s16(y0, y4, 3);
-        int16x4_t       y7 = vext_s16(y4, y8, 3);
-        int32x4_t       a6 = vmlal_lane_s16(a5, y3, x0, 3);
-        int32x4_t       a7 = vmlal_lane_s16(a6, y7, x4, 3);

-
-        y0 = y8;
-        a = a7;
         x += 8;
         y += 8;
+        len -= 8;
     }
-
-    for (; j < len; j++) {
-        int16x4_t       x0 = vld1_dup_s16(x);   /* load next x */
-        int32x4_t       a0 = vmlal_s16(a, y0, x0);
-
-        int16x4_t       y4 = vld1_dup_s16(y);   /* load next y */
-        y0 = vext_s16(y0, y4, 1);
-        a = a0;
-        x++;
```

```
-         y++;
-     }
-
-     vst1q_s32(sum, a);
+     while (len > 0);
  }
```

10 Floating-point 4x4 matrix transposition

The `mat_transpose_inp_4x4_neon_f32` function uses intrinsics to transpose a 4x4 matrix that contains single-precision floating-point data.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/matrix.c

10.1 Neon implementation

The Neon implementation uses the `vtrnq_f32` intrinsic to perform matrix transposition. For more background information about the corresponding `VTRN` instruction, see Coding for Neon - Part 5: Rearranging Vectors.

The following code shows an implementation of a 4x4 floating-point matrix transposition function using Neon intrinsics:

```
void mat_transpose_inp_4x4_neon_f32(float32_t * matrix)
{
  float32x4_t     row0 = vld1q_f32(matrix);
  float32x4_t     row1 = vld1q_f32(matrix + 4);
  float32x4_t     row2 = vld1q_f32(matrix + 8);
  float32x4_t     row3 = vld1q_f32(matrix + 12);

  float32x4x2_t   row01 = vtrnq_f32(row0, row1);
  float32x4x2_t   row23 = vtrnq_f32(row2, row3);

  vst1q_f32(matrix,
            vcombine_f32(vget_low_f32(row01.val[0]), vget_low_f32(row23.val[0])));
  vst1q_f32(matrix + 4,
            vcombine_f32(vget_low_f32(row01.val[1]), vget_low_f32(row23.val[1])));
  vst1q_f32(matrix + 8,
            vcombine_f32(vget_high_f32(row01.val[0]), vget_high_f32(row23.val[0])));
  vst1q_f32(matrix + 12,
            vcombine_f32(vget_high_f32(row01.val[1]), vget_high_f32(row23.val[1])));
}
```

10.2 Direct migration to Helium

The Neon implementation uses the following intrinsics which do not have Helium equivalents:

- `vtrnq_f32`: vector transpose
- `vcombine_f32`: join two half vectors into a single full vector
- `vget_low_f32` and `vget_high_f32`: half-vector extraction

All these instructions can be emulated with the help of the `__builtin_shufflevector` function introduced in 16-bit fixed-point cross-correlation, as seen in the following code:

```
static inline float32x4x2_t vtrnq_emu_f32(float32x4_t a, float32x4_t b)
{
  float32x4x2_t   dst;

  dst.val[0] = __builtin_shufflevector(a, b, 0, 4, 2, 6);
  dst.val[1] = __builtin_shufflevector(a, b, 1, 5, 3, 7);
  return dst;
}
static inline float32x4_t vcombine_emu_f32(float32x4_t a, float32x4_t b)
{
  return __builtin_shufflevector(a, b, 0, 1, 4, 5);
}

static inline float32x4_t vget_low_emu_f32(float32x4_t a)
{
  return __builtin_shufflevector(a, a, 0, 1, DONT_CARE, DONT_CARE);
}

static inline float32x4_t vget_high_emu_f32(float32x4_t a)
{
  return __builtin_shufflevector(a, a, 2, 3, DONT_CARE, DONT_CARE);
}
```

However, emulating these instructions with `__builtin_shufflevector` comes with a cost to performance.

The same emulation strategy can also be used for other Neon data rearranging intrinsics. For example, the following code shows how you can emulate the `vzip` and `vuzp` intrinsics:

```
static inline float32x4x2_t vzipq_emu_f32(float32x4_t x, float32x4_t y)
{
  float32x4x2_t   dst;
  dst.val[0] = __builtin_shufflevector(x, y, 0, 4, 1, 5);
  dst.val[1] = __builtin_shufflevector(x, y, 2, 6, 3, 7);
  return dst;
}

static inline float32x4x2_t vuzpq_emu_f32(float32x4_t x, float32x4_t y)
{
  float32x4x2_t   dst;
  dst.val[0] = __builtin_shufflevector(x, y, 0, 2, 4, 6);
  dst.val[1] = __builtin_shufflevector(x, y, 1, 3, 5, 7);
  return dst;
}
```

10.3 Optimized migration to Helium

As we explained in 16-bit fixed-point cross-correlation, using emulation to simply replace Neon intrinsics is not the recommended approach. The resulting Helium code would miss important optimization features and perform poorly. However, direct migration allows us to quickly get Neon code running on a Helium processor as an initial prototyping stage.

When migrating Neon data rearranging intrinsics, `VREV` is the only intra-vector rearrangement instruction which is available on both Neon and Helium. All other operations requiring intricated vector element shuffling on Helium should be implemented using scatter-gather operations or interleaved-deinterleaved loads and stores to rearrange data from memory.

The following code shows the final version of the optimized Helium code:

```
void mat_transpose_inp_4x4_helium_f32(float32_t * matrix)
{
  float32x4x4_t   rows;

  rows.val[0] = vld1q_f32(matrix);
  rows.val[1] = vld1q_f32(matrix + 4);
  rows.val[2] = vld1q_f32(matrix + 8);
  rows.val[3] = vld1q_f32(matrix + 12);

  vst4q_f32(matrix, rows);
}
```


Note

This approach, which uses interleaved loads to rearrange the matrix data, could also have been used for the original Neon implementation.

11 Integer 8-bit 4x4 matrix transposition

The `mat_transpose_inp_4x4_neon_u8` function uses intrinsics to transpose a 4x4 matrix containing 8-bit integer data.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/matrix.c

11.1 Neon implementation

The Neon implementation uses the `vtbl2_u8` vector table lookup intrinsic to perform matrix transposition. For more background information about the corresponding `VTBL` instruction, see Coding for Neon - Part 5: Rearranging Vectors.

The following code shows an implementation of a 4x4 8-bit integer matrix transposition function using Neon intrinsics:

```
static const uint8_t mat_4x4_transp_idx_u8[16] = {
  0, 4, 8, 12,
  1, 5, 9, 13,
  2, 6, 10, 14,
  3, 7, 11, 15,
};

void mat_transpose_inp_4x4_neon_u8(uint8_t * matrix)
{
  uint8x16_t      mat = vld1q_u8(matrix);
  uint8x16_t      offset = vld1q_u8(mat_4x4_transp_idx_u8);
  uint8x8_t       bot, top;
  uint8x8x2_t     tmp;

  tmp.val[0] = vget_low_u8(mat);
  tmp.val[1] = vget_high_u8(mat);

  bot = vtbl2_u8(tmp, vget_low_u8(offset));
  top = vtbl2_u8(tmp, vget_high_u8(offset));

  vst1q_u8(matrix, vcombine_u8(bot, top));
}
```

11.2 Optimized migration to Helium

The Neon implementation uses the following intrinsics which do not have Helium equivalents:

- `vtbl2_u8`: vector table lookup
- `vget_low_f32` and `vget_high_f32`: half-vector extraction

Unlike the Floating-point 4x4 matrix transposition example, we cannot use the `__builtin_shufflevector` function to emulate `vtbl2_u8`. This is because the table index is dynamic. Therefore, a direct migration to Helium is not possible, and we will move straight to an optimized solution.


Note

In this specific transposition example, the permutation indexes are fixed. This means that `__builtin_shufflevector` could be used. However, we will show a solution that can be used for the general case.

Helium provides scatter-gather support which we can use to rearrange the vector during the load and store process. In the Neon implementation, data is loaded linearly from memory then rearranged within the vector itself.

The `vldrbq_gather_offset_u8` intrinsic lets us specify a base address and vector index as input. The intrinsic then gathers elements using the address calculation `base + index[]`.

The following code shows the final version of the optimized Helium code:

```
void mat_transpose_inp_4x4_helium_u8(uint8_t * matrix)
{
  uint8x16_t      mat;
  uint8x16_t      offset = vld1q_u8(mat_4x4_transp_idx_u8);

  mat = vldrbq_gather_offset_u8(matrix, offset);
  vst1q_u8(matrix, mat);
}
```

The Helium scatter-gather intrinsics support all data types, and data narrowing and widening. This range of abilities make these intrinsics powerful for manipulating a wide variety of data.

The vector index range is constrained to the vector element range. That is, for an 8-bit gather load, the index must also be 8-bit with a value between 0 and 255. If you need to transpose larger matrices, the gather-load-with-widening intrinsics let you increase the index range. For example, the `vldrbq_gather_offset_u16` intrinsic uses a vector of 16-bit index. This approach is needed to transpose a matrix with a size greater than 256 elements.

12 RGB to grayscale conversion

The `rgb_to_gray_neon` function uses intrinsics to convert RGB color pixel data to grayscale values. RGB pixel data is transferred from memory using a de-interleaving load, then the individual red, green, and blue components are combined using different weights to obtain a grayscale value.

The source code for this example is available at the following location:

https://github.com/ARM-software/EndpointAI/blob/master/Kernels/Migrating_to_Helium_from_Neon_Companion_SW/rgb.c

12.1 Neon implementation

The Neon implementation uses the `vld3_u8` 3-element de-interleaving load to retrieve the red, green, and blue component data from memory. For more background information about using Neon intrinsics to load RGB data, see Optimizing C Code with Neon Intrinsics: RGB deinterleaving.

The following code shows an implementation of an RGB to grayscale conversion function using Neon intrinsics:

```
void rgb_to_gray_neon(const uint8_t * rgb, uint8_t * gray, int num_pixels)
{
  uint8x8_t       w_r = vdup_n_u8(77);
  uint8x8_t       w_g = vdup_n_u8(150);
  uint8x8_t       w_b = vdup_n_u8(29);
  uint16x8_t      temp;
  uint8x8_t       result;

  num_pixels /= 8;

  for (int i = 0; i < num_pixels; ++i, rgb += 8 * 3, gray += 8) {
    uint8x8x3_t     src = vld3_u8(rgb);

    temp = vmull_u8(src.val[0], w_r);

    temp = vmlal_u8(temp, src.val[1], w_g);
    temp = vmlal_u8(temp, src.val[2], w_b);

    result = vshrn_n_u16(temp, 8);

    vst1_u8(gray, result);
  }
}
```

The function uses the following Neon intrinsics:

- `vld3_u8`: three-element de-interleaving load to extract individual red, green, and blue components from pixel data
- `vmull_u8` and `vmlal_u8`: multiply each component by a different weight to obtain a 16-bit grayscale value
- `vshrn_n_u16`: shift the most significant bits of 16-bit grayscale value to get an 8-bit value
- `vst1_u8`: contiguous store of the 8-bit grayscale pixel values

12.2 Optimized migration to Helium

Helium does not support the `vld3_u8` interleaved three-element load intrinsic. However, Helium provides scatter-gather support which we can use to emulate the three-element load operation.

The following macros emulate the Neon `vld3` and `vld3q` intrinsics by executing three successive gather loads. Each of the gather loads use incrementing base addresses together with multiple-of-3 indexes. The code for the macros is as follows:

```
static inline int16x8x3_t vld3_emu_u8(const uint8_t * base)
{
  int16x8x3_t     dst;
  int16x8_t       index;

  index = vidupq_n_u16(0, 1);
  index = vmulq_n_u16(index, 3);

  dst.val[0] = vldrbq_gather_offset_u16(base, index);
  dst.val[1] = vldrbq_gather_offset_u16(base + 1, index);
  dst.val[2] = vldrbq_gather_offset_u16(base + 2, index);
  return dst;
}

static inline int8x16x3_t vld3q_emu_u8(const uint8_t * base)
{
  int8x16x3_t     dst;
  int16x8_t       index;

  index = vidupq_n_u16(0, 1);
  index = vmulq_n_u16(index, 3);

  dst.val[0] = vldrbq_gather_offset_u8(base, index);
  dst.val[1] = vldrbq_gather_offset_u8(base + 1, index);
  dst.val[2] = vldrbq_gather_offset_u8(base + 2, index);
  return dst;
}

typedef struct {
  int16x8_t       val[3];
} int16x8x3_t;

typedef struct {
  int8x16_t       val[3];
} int8x16x3_t;
```

The following code shows the final version of the optimized Helium code, which uses the macros described in the preceding code:

```
void rgb_to_gray_helium(const uint8_t * rgb, uint8_t * gray, int num_pixels)
{
    uint16_t        w_r = (77);
    uint16_t        w_g = (150);
    uint16_t        w_b = (29);
    uint16x8_t      temp;
    uint16x8_t      result;

    num_pixels /= 8;
    for (int i = 0; i < num_pixels; ++i, rgb += 8 * 3, gray += 8) {
        int16x8x3_t     src = vld3_emu_u8(rgb); /* widened */

        temp = vmulq_n_u16(src.val[0], w_r);
        temp = vmlaq_n_u16(temp, src.val[1], w_g);
        temp = vmlaq_n_u16(temp, src.val[2], w_b);
```

```
        result = vshrq_n_u16(temp, 8);

        vstrbq_u16(gray, result); /* narrowed store */
    }
}
```

The `vidupq_n_u16(0, 1)` and `vmulq_n_u16` operations generate a sequence of index values pointing to multiples of three. First, `vidupq_n_u16(0, 1)` generates a vector containing an incrementing sequence from 0 to 7. This sequence is then multiplied by 3 to generate an index sequence containing: {0, 3, 6, 9, 12, 15, 18, 21}

Subsequent gather loads use three different bases addresses and the same multiple-of-three index sequence we just generated to perform the de-interleaved load.

Using the emulated `vld3_emu_u8` function widens the vector elements so that no further long multiplication is needed. The multiplication operation can therefore use the Helium `vmulq_n_u16` and `vmlaq_n_u16` intrinsics.

Neon uses the `vshrn_n_u16` intrinsic to shift and narrow the final result from `uint16x8_t` to `uint8x8_t`. Helium also supports this shift and narrowing operation with the `vshrq_n_u16` intrinsic.


Note

Helium does not provide a half-vector type, so the intrinsics provide additional modifiers to specify top or bottom destination locations, for example `vshrnbq_n_s16` and `vshrntq_n_s16`.

Helium provides a narrowed store operation, so the simple shift is sufficient with no widening required. The `vstrbq_u16` intrinsic stores the lower parts of the 16-bit vector elements into a contiguous byte stream.

13 AEC in WebRTC

This section of the guide examines the implementation of the core AEC algorithm in WebRTC.

Web Real-Time Communication (WebRTC) is an open-source project that enables Real-Time Communications (RTC) capabilities in web browsers and mobile applications. WebRTC provides an extensive suite of API functions that developers can use to implement real-time multimedia applications like video chat in web browsers.

One of the features that WebRTC provides is Acoustic Echo Cancelation (AEC). Annoying echoes can occur when sounds that are played through speakers are captured by the microphone. AEC analyzes both input and output audio signals and removes these echoes in real-time.

Looking at the core AEC algorithm as a whole, the majority of the functions can be migrated from Neon to Helium using the techniques described in this guide. However, the `ScaleErrorSignalNEON_partial` function appears to be more difficult to migrate. This is because it uses both vector division and square root operations. Helium does not provide instructions for either of these operations.

The source code for this example is available at the following location:

https://chromium.googlesource.com/external/webrtc/+/ee0c100d5495cd8c440b767a7852532afbbcefb2/webrtc/modules/audio_processing/aec/aec_core_neon.c

13.1 Neon implementation

The following code shows the implementation of `ScaleErrorSignalNEON_partial` which forms part of the WebRTC core AEC algorithm:

```
void ScaleErrorSignalNEON_partial( float * aecxPow, float ef[2][PART_LEN1])
{
  const float mu = 1.0f;
  const float error_threshold = 1e-5f;
  const float32x4_t k1e_10f = vdupq_n_f32(1e-10f);
  const float32x4_t kMu = vmovq_n_f32(mu);
  const float32x4_t kThresh = vmovq_n_f32(error_threshold);
  int i;
  // vectorized code (four at once)
  for (i = 0; i + 3 < PART_LEN1; i += 4) {
    const float32x4_t xPow = vld1q_f32(&aecxPow[i]);
    const float32x4_t ef_re_base = vld1q_f32(&ef[0][i]);
    const float32x4_t ef_im_base = vld1q_f32(&ef[1][i]);
    const float32x4_t xPowPlus = vaddq_f32(xPow, k1e_10f);
    float32x4_t ef_re = vdivq_f32(ef_re_base, xPowPlus);
    float32x4_t ef_im = vdivq_f32(ef_im_base, xPowPlus);
    const float32x4_t ef_re2 = vmulq_f32(ef_re, ef_re);
    const float32x4_t ef_sum2 = vmlaq_f32(ef_re2, ef_im, ef_im);
    const float32x4_t absEf = vsqrtq_f32(ef_sum2);
    const uint32x4_t bigger = vcgtq_f32(absEf, kThresh);
    const float32x4_t absEfPlus = vaddq_f32(absEf, k1e_10f);
    const float32x4_t absEfInv = vdivq_f32(kThresh, absEfPlus);
    uint32x4_t ef_re_if = vreinterpretq_u32_f32(vmulq_f32(ef_re, absEfInv));
    uint32x4_t ef_im_if = vreinterpretq_u32_f32(vmulq_f32(ef_im, absEfInv));
    uint32x4_t ef_re_u32 = vandq_u32(vmvnq_u32(bigger),
                                     vreinterpretq_u32_f32(ef_re));
```

```
    uint32x4_t ef_im_u32 = vandq_u32(vmvnq_u32(bigger),
                                     vreinterpretq_u32_f32(ef_im));
    ef_re_if = vandq_u32(bigger, ef_re_if);
    ef_im_if = vandq_u32(bigger, ef_im_if);
    ef_re_u32 = vorrq_u32(ef_re_u32, ef_re_if);
    ef_im_u32 = vorrq_u32(ef_im_u32, ef_im_if);
    ef_re = vmulq_f32(vreinterpretq_f32_u32(ef_re_u32), kMu);
    ef_im = vmulq_f32(vreinterpretq_f32_u32(ef_im_u32), kMu);
    vst1q_f32(&ef[0][i], ef_re);
    vst1q_f32(&ef[1][i], ef_im);
  }
}
```

13.2 Direct migration to Helium

The Neon `ScaleErrorSignalNEON_partial` function uses the following intrinsics which do not have Helium equivalents:

- `vdivq_f32`: floating-point vector divide
- `vsqrtq_f32`: floating-point vector square root

A basic migration method might be to create an emulation function for each of these intrinsics which contains four separate calls to scalar division and square root instructions, one for each individual element.

13.3 Optimized migration to Helium

The basic migration method described in the preceding section does not take advantage of vectorization. A more efficient method is to implement vectorizing division and square root in Helium. This section of the guide looks at how to do this in detail.

13.3.1 Vector division

The Newton-Raphson method provides a high-speed method for performing division by computing the inverse of a floating-point number. We can extend this method to operate on vectors rather than individual values.

The `vinvq_helium_f32` function computes the reciprocal of elements of a single precision vector. This function uses three iterations but can be adjusted depending on the precision that you require. The initialization and step portions of the algorithm are wrapped in a macro to mimic the Neon intrinsics `vrecpeq_f32` and `vrecpsq_f32`. Additional code is included to handle division by zero.

The following code shows the implementation of `vinvq_helium_f32` together with the two macros `vrecpeq_f32` and `vrecpsq_f32` that mimic Neon reciprocal initial estimation and step:

```
float32x4_t vinvq_helium_f32(float32x4_t x)
{
  float32x4_t     recip;
  float32x4_t     ax = vabsq(x);

  recip = vrecpq_init_f32(ax);        /* vrecpeq_f32 equivalent */

  recip = vrecpq_step_f32(recip, ax); /* vrecpsq_f32 equivalent */
```

```
  recip = vrecpq_step_f32(recip, ax);
  recip = vrecpq_step_f32(recip, ax);

  /*
   * restore sign + handle division by 0
   */
  recip = vdupq_m(recip, INFINITY, vcmpeqq(x, 0.0f));
  recip = vnegq_x_f32(recip, vcmpltq(x, 0.0f));
  return recip;
}

static inline float32x4_t vrecpq_init_f32(float32x4_t ax)
{
  any32x4_t       xinv;
  int32x4_t       m;

  xinv.f = ax;
  m = vsubq(vdupq_n_s32(0x3F800000), vandq_s32(xinv.i, vdupq_n_s32(0x7F800000)));
  xinv.i = vaddq_s32(xinv.i, m);
  xinv.f = vfmsq_f32(vdupq_n_f32(1.41176471f), vdupq_n_f32(0.47058824f), xinv.f);
  xinv.i = vaddq_s32(xinv.i, m);
  return xinv.f;
}

static inline float32x4_t vrecpq_step_f32(float32x4_t x, float32x4_t ax)
{
  float32x4_t     b;
  float32x4_t     v2f = vdupq_n_f32(2.0f);

  b = vfmsq(v2f, x, ax);
  x = vmulq_f32(x, b);
  return x;
}
```

The final result of the division operation is then calculated by multiplying the reciprocal with the numerator, as follows:

```
static inline float32x4_t vdiv_helium_f32( float32x4_t num, float32x4_t den)
{
  return vmulq_f32(num, vinvq_helium_f32(den));
}
```

13.3.2 Vector square root

Like with vector division, we can use the Newton-Raphson method to compute the inverse square root. This method is described in detail in Lomont, Chris - Fast Inverse Square Root (2003).

The following code shows the implementation of `vinvsqrtq_helium_f32` together with the two macros that mimic the `vrsqrteq_f32` and `vrecpsq_f32` Neon floating-point reciprocal square root estimate and step intrinsics:

```
float32x4_t vinvsqrtq_helium_f32 (float32x4_t vecIn)
{
  float32x4_t     sqrt_reciprocal;
  float32x4_t     vecHalf;

  vecHalf = vmulq(vecIn, 0.5f);

  /* vrsqrteq_f32 equiv. */
  sqrt_reciprocal = vinvsqrtq_helium_init_f32(vecIn); /* vrsqrteq_f32 equiv. */

  /* vrsqrtsq_f32 equiv. */
```

```
  sqrt_reciprocal = vinvsqrtq_helium_step_f32(sqrt_reciprocal, vecHalf);
  sqrt_reciprocal = vinvsqrtq_helium_step_f32(sqrt_reciprocal, vecHalf);
  sqrt_reciprocal = vinvsqrtq_helium_step_f32(sqrt_reciprocal, vecHalf);

  /*
   * handle negative and 0 values
   */
  sqrt_reciprocal = vdupq_m(sqrt_reciprocal, NAN, vcmpltq(vecIn, 0.0f));
  sqrt_reciprocal = vdupq_m(sqrt_reciprocal, INFINITY, vcmpeqq(vecIn, 0.0f));
  return sqrt_reciprocal;
}


static inline float32x4_t vinvsqrtq_helium_step_f32(float32x4_t sqrt_reciprocal,
                                                    float32x4_t vecHalf)
{
  float32x4_t     vecOneHandHalf = vdupq_n_f32(1.5f);
  float32x4_t     tmp;
  /*
   * 1st iteration
   * (1.5f-xhalf*x*x)
   */
  tmp = vmulq(sqrt_reciprocal, sqrt_reciprocal);
  tmp = vfmsq_f32(vecOneHandHalf, tmp, vecHalf);
  /*
   * x = x*(1.5f-xhalf*x*x);
   */
  sqrt_reciprocal = vmulq(sqrt_reciprocal, tmp);
  return sqrt_reciprocal;
}

static inline float32x4_t vinvsqrtq_helium_init_f32(float32x4_t vecIn)
{
  int32x4_t       vecNewtonInit = vdupq_n_s32(INVSQRT_MAGIC_F32);
  float32x4_t     sqrt_reciprocal;
  int32x4_t       vecTmpInt;

  /*
   * cast input float vector to integer and right shift by 1
   */
  vecTmpInt = vshrq_n_s32((int32x4_t) vecIn, 1);
  /*
   * INVSQRT_MAGIC - ((vec_q32_t)vecIn >> 1)
   */
  sqrt_reciprocal = vreinterpretq_f32_s32(vsubq(vecNewtonInit, vecTmpInt));
  return sqrt_reciprocal;
}
```

The final square root result is calculated by multiplying the inverse square root of the input vector with itself. Any zero values are cleared, to avoid getting an infinite result from calculating the inverse square root of zero.

The following code shows the implementation of the vector square root function:

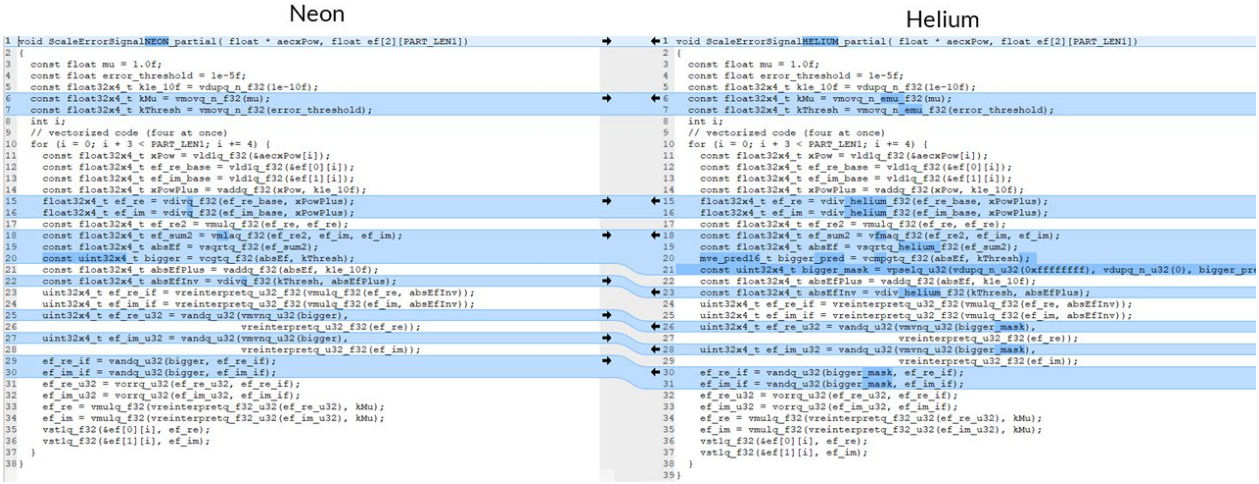
```
static inline float32x4_t vsqrtq_helium_f32( float32x4_t in)
{
  float32x4_t         dst;

  dst = vinvsqrtq_helium_f32_neon_like(in);
  dst = vdupq_m_n_f32(dst, 0.0f, vcmpeqq_n_f32(in, 0.0f));
  dst = vmulq_f32(dst, in);
  return dst;
}
```

13.3.3 Full code for optimized Helium migration

Using the vector division and square root functions described previously, the code for the optimized migration to Helium is as follows:

```
void ScaleErrorSignalHELIUM_partial( float * aecxPow, float ef[2][PART_LEN1])
{
  const float mu = 1.0f;
  const float error_threshold = 1e-5f;
  const float32x4_t k1e_10f = vdupq_n_f32(1e-10f);
  const float32x4_t kMu = vmovq_n_emu_f32(mu);
  const float32x4_t kThresh = vmovq_n_emu_f32(error_threshold);
  int i;
  // vectorized code (four at once)
  for (i = 0; i + 3 < PART_LEN1; i += 4) {
    const float32x4_t xPow = vld1q_f32(&aecxPow[i]);
    const float32x4_t ef_re_base = vld1q_f32(&ef[0][i]);
    const float32x4_t ef_im_base = vld1q_f32(&ef[1][i]);
    const float32x4_t xPowPlus = vaddq_f32(xPow, k1e_10f);
    float32x4_t ef_re = vdiv_helium_f32(ef_re_base, xPowPlus);
    float32x4_t ef_im = vdiv_helium_f32(ef_im_base, xPowPlus);
    const float32x4_t ef_re2 = vmulq_f32(ef_re, ef_re);
    const float32x4_t ef_sum2 = vfmaq_f32(ef_re2, ef_im, ef_im);
    const float32x4_t absEf = vsqrtq_helium_f32(ef_sum2);
    mve_pred16_t bigger_pred = vcmpgtq_f32(absEf, kThresh);
    const uint32x4_t bigger_mask = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0),
bigger_pred);
    const float32x4_t absEfPlus = vaddq_f32(absEf, k1e_10f);
    const float32x4_t absEfInv = vdiv_helium_f32(kThresh, absEfPlus);
    uint32x4_t ef_re_if = vreinterpretq_u32_f32(vmulq_f32(ef_re, absEfInv));
    uint32x4_t ef_im_if = vreinterpretq_u32_f32(vmulq_f32(ef_im, absEfInv));
    uint32x4_t ef_re_u32 = vandq_u32(vmvnq_u32(bigger_mask),
                                     vreinterpretq_u32_f32(ef_re));
    uint32x4_t ef_im_u32 = vandq_u32(vmvnq_u32(bigger_mask),
                                     vreinterpretq_u32_f32(ef_im));
    ef_re_if = vandq_u32(bigger_mask, ef_re_if);
    ef_im_if = vandq_u32(bigger_mask, ef_im_if);
    ef_re_u32 = vorrq_u32(ef_re_u32, ef_re_if);
    ef_im_u32 = vorrq_u32(ef_im_u32, ef_im_if);
    ef_re = vmulq_f32(vreinterpretq_f32_u32(ef_re_u32), kMu);
    ef_im = vmulq_f32(vreinterpretq_f32_u32(ef_im_u32), kMu);
    vst1q_f32(&ef[0][i], ef_re);
    vst1q_f32(&ef[1][i], ef_im);
  }
}
```

The following image shows the major differences between the Neon implementation and the optimized Helium implementation:



The following GNU diff output shows the same differences in text form:

```
--- neon.c      Thu Nov 19 15:25:27 2020
+++ helium_optimized.c  Thu Nov 19 15:25:06 2020
@@ -1,10 +1,10 @@
-void ScaleErrorSignalNEON_partial( float * aecxPow, float ef[2][PART_LEN1])
+void ScaleErrorSignalHELIUM_partial( float * aecxPow, float ef[2][PART_LEN1])
 {
   const float mu = 1.0f;
   const float error_threshold = 1e-5f;
   const float32x4_t k1e_10f = vdupq_n_f32(1e-10f);
-  const float32x4_t kMu = vmovq_n_f32(mu);
-  const float32x4_t kThresh = vmovq_n_f32(error_threshold);
+  const float32x4_t kMu = vmovq_n_emu_f32(mu);
+  const float32x4_t kThresh = vmovq_n_emu_f32(error_threshold);
   int i;
   // vectorized code (four at once)
   for (i = 0; i + 3 < PART_LEN1; i += 4) {
@@ -12,22 +12,23 @@
     const float32x4_t ef_re_base = vld1q_f32(&ef[0][i]);
     const float32x4_t ef_im_base = vld1q_f32(&ef[1][i]);
     const float32x4_t xPowPlus = vaddq_f32(xPow, k1e_10f);
-    float32x4_t ef_re = vdivq_f32(ef_re_base, xPowPlus);
-    float32x4_t ef_im = vdivq_f32(ef_im_base, xPowPlus);
+    float32x4_t ef_re = vdiv_helium_f32(ef_re_base, xPowPlus);
+    float32x4_t ef_im = vdiv_helium_f32(ef_im_base, xPowPlus);
     const float32x4_t ef_re2 = vmulq_f32(ef_re, ef_re);
-    const float32x4_t ef_sum2 = vmlaq_f32(ef_re2, ef_im, ef_im);
-    const float32x4_t absEf = vsqrtq_f32(ef_sum2);
-    const uint32x4_t bigger = vcgtq_f32(absEf, kThresh);
+    const float32x4_t ef_sum2 = vfmaq_f32(ef_re2, ef_im, ef_im);
+    const float32x4_t absEf = vsqrtq_helium_f32(ef_sum2);
+    mve_pred16_t bigger_pred = vcmpgtq_f32(absEf, kThresh);
+    const uint32x4_t bigger_mask = vpselq_u32(vdupq_n_u32(0xffffffff), vdupq_n_u32(0), bigger_pred);
     const float32x4_t absEfPlus = vaddq_f32(absEf, k1e_10f);
-    const float32x4_t absEfInv = vdivq_f32(kThresh, absEfPlus);
+    const float32x4_t absEfInv = vdiv_helium_f32(kThresh, absEfPlus);
     uint32x4_t ef_re_if = vreinterpretq_u32_f32(vmulq_f32(ef_re, absEfInv));
     uint32x4_t ef_im_if = vreinterpretq_u32_f32(vmulq_f32(ef_im, absEfInv));
-    uint32x4_t ef_re_u32 = vandq_u32(vmvnq_u32(bigger),
+    uint32x4_t ef_re_u32 = vandq_u32(vmvnq_u32(bigger_mask),
                                      vreinterpretq_u32_f32(ef_re));
-    uint32x4_t ef_im_u32 = vandq_u32(vmvnq_u32(bigger),
+    uint32x4_t ef_im_u32 = vandq_u32(vmvnq_u32(bigger_mask),
                                      vreinterpretq_u32_f32(ef_im));
-    ef_re_if = vandq_u32(bigger, ef_re_if);
-    ef_im_if = vandq_u32(bigger, ef_im_if);
+    ef_re_if = vandq_u32(bigger_mask, ef_re_if);
+    ef_im_if = vandq_u32(bigger_mask, ef_im_if);
     ef_re_u32 = vorrq_u32(ef_re_u32, ef_re_if);
     ef_im_u32 = vorrq_u32(ef_im_u32, ef_im_if);
     ef_re = vmulq_f32(vreinterpretq_f32_u32(ef_re_u32), kMu);
```

```
                                        vreinterpretq_u32_f32(ef_re));
-    uint32x4_t ef_im_u32 = vandq_u32(vmvnq_u32(bigger),
+    uint32x4_t ef_im_u32 = vandq_u32(vmvnq_u32(bigger_mask),
                                        vreinterpretq_u32_f32(ef_im));
-    ef_re_if = vandq_u32(bigger, ef_re_if);
-    ef_im_if = vandq_u32(bigger, ef_im_if);
+    ef_re_if = vandq_u32(bigger_mask, ef_re_if);
+    ef_im_if = vandq_u32(bigger_mask, ef_im_if);
     ef_re_u32 = vorrq_u32(ef_re_u32, ef_re_if);
     ef_im_u32 = vorrq_u32(ef_im_u32, ef_im_if);
     ef_re = vmulq_f32(vreinterpretq_f32_u32(ef_re_u32), kMu);
```

14 Related information

Here are some resources related to the material in this guide:

- Arm C Language Extensions Documentation: M-profile Vector Extension (MVE) intrinsics
- Arm Community provides a forum where you can ask questions, and find articles and blogs on specific topics from Arm experts.
- Clang Language Extensions.
- Git repository of companion code examples for this guide
- Helium intrinsics reference
- Helium Programmer's Guide
- Neon intrinsics reference
- Neon Programmer's Guide for Armv8-A
- Xiph.Org Foundation

15 Next steps

This guide has provided examples of how to migrate code from Neon to Helium using intrinsics.

The next step is for you to migrate your own Neon code to Helium.

Arm provides extensive reference material for both Neon and Helium intrinsics, which will help you during the migration process:

- Helium intrinsics reference
- Neon intrinsics reference

For further examples of how Neon intrinsics can be used in real-life scenarios, please see the Neon intrinsics Chromium case study.