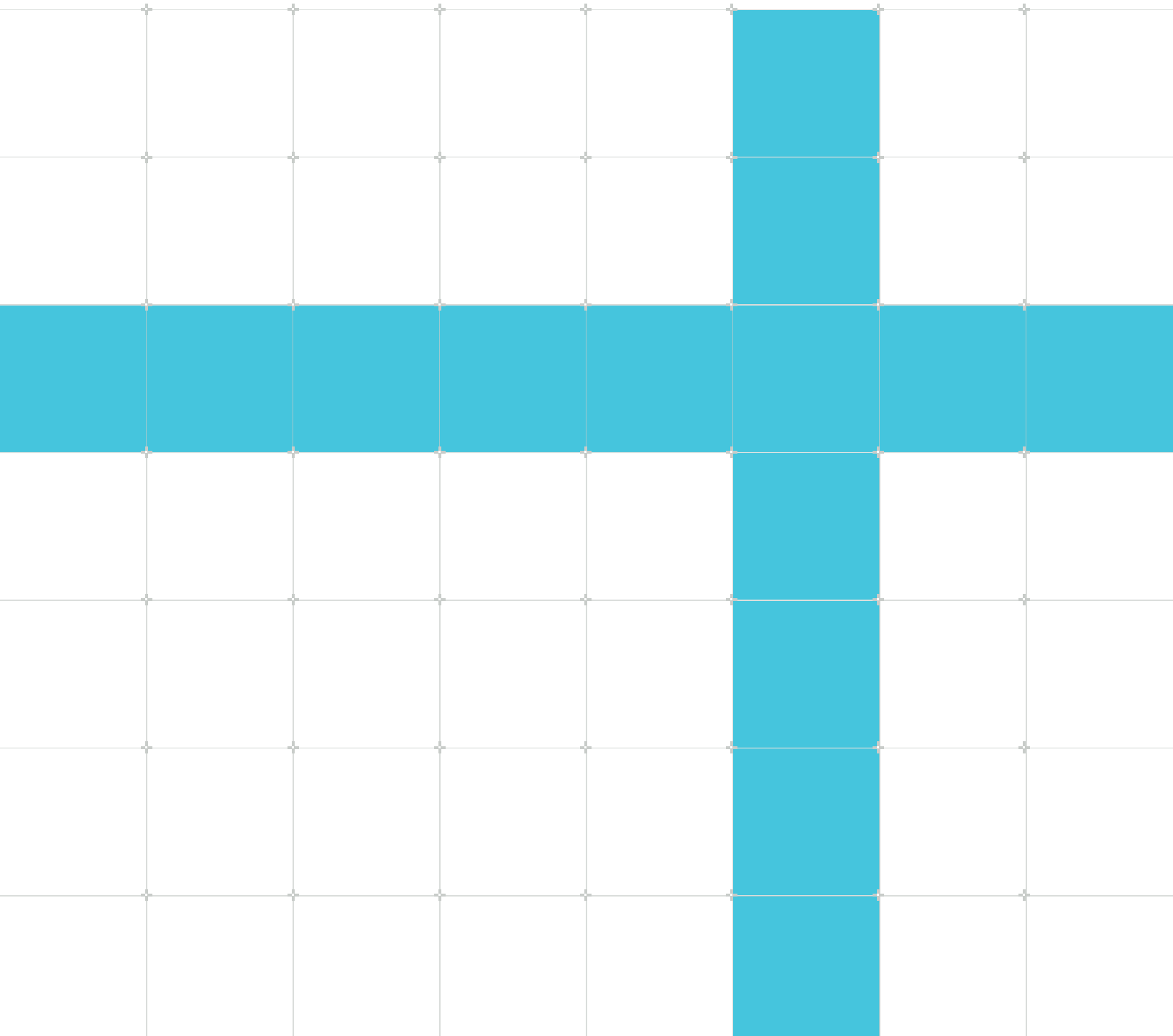
Lighter Yet Better - Powering Edge Through Armv8.1-M Features

Version 1.0

White Paper

Issue 01
107564_0100_01_en



Lighter Yet Better - Powering Edge Through Armv8.1-M Features

White Paper

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	28 September 2021	Non-Confidential	First release

Proprietary Notice

and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

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

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.

Feedback

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

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

Inclusive language commitment

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

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1 Abstract

With potentially trillions of connected devices by 2035, the requirements on edge compute devices are increasing all the time. To reduce the amount of data transmitted to the cloud, endpoint devices need to become smarter, using the latest Artificial Intelligence (AI), Machine Learning (ML), and Digital Signal Processing (DSP) techniques to process large amounts of data locally.

To address this need, Arm® introduced the Armv8.1-M architecture, including the M-profile Vector Extension (MVE), also known as Helium technology in Arm Cortex®-M processors.

Helium™ technology brings vector processing capabilities to Cortex-M based devices. Helium enables these devices to handle DSP and ML tasks much more quickly, while still allowing these devices to be small, energy-efficient, and retain real-time functionality. In addition, Armv8.1-M also introduced the Low Overhead Branch (LOB) extension, which reduces branch overheads in loop operations.

The MVE and LOB extensions allow C/C++ compilers to deliver code with better performance in general-purpose compute applications. This whitepaper shows how the features of the Armv8.1-M architecture benefit a range of workloads including AI, DSP, ML, and commonly used EEMBC® benchmarks. Analyzing how these workloads benefit provides insights into how you can optimize your own applications.

This whitepaper focuses on the Cortex-M55 processor, which is the first Cortex-M processor to implement the Armv8.1-M architecture and Helium technology.

1.1 Authors

Shivani Jayaprakash [Performance Engineer]

Mal Reddy [Staff Performance Engineer]

Joseph Yiu [Distinguished Engineer]

2 Introduction

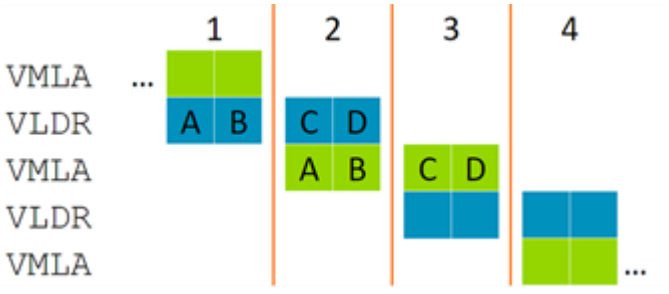
This section introduces the Helium and Low Overhead Branch (LOB) extensions, and describes C/C ++ compiler support for auto-vectorization.

2.1 M-Profile Vector Extension (Helium)

Vector processing support in the Arm architecture is not a new development. For many years, Arm Cortex-A processors have supported a vector extension called Neon™ technology (Reference 1), which is now also available in the latest Cortex-R processors including Cortex-R52 and Cortex-R82. Some people might think that developing a vector extension for Cortex-M processors simply means reusing the existing Neon instructions. However, Neon instructions are not optimized for small, embedded processors like the Cortex-M series where the focus is on finding the right balance between performance, energy, area, and cost constraints. This led to the introduction of Helium technology (Reference 2), a brand-new Single Instruction, Multiple Data (SIMD) instruction set extension designed for the Cortex-M processor family. In the Armv8-M architecture document, this instruction set extension is referred to as the M-Profile Vector Extension (MVE). Helium technology is the name of the feature on Arm Cortex-M processors designed to support MVE. The name Helium is a play on words – it is a noble gas like neon but has a lower atomic mass.

To optimize the use of expensive hardware resources, the MVE extension reuses the registers in the floating-point unit (FPU) as vector registers. Each vector is 128 bits and the architecture splits the 128-bit wide vector data into four equally sized 32-bit wide blocks known as beats. The Cortex-M55 processor (Reference 3) is a dual-beat implementation of Helium technology which means that the Cortex-M55 processes two beats of data for every clock cycle. When comparing the Cortex-M55 processor to previous Cortex-M4 and Cortex-M33 processors, the Helium implementation in Cortex-M55 is targeted to achieve a greater than 4x performance improvement with a 2x increase in data path width. This is achieved by overlapping execution cycles of MVE instructions for the different beats in a vector. The beat-wise execution of 128-bit vector processing in a 64-bit data width implemented in Cortex-M55 is shown in the following diagram:

Figure 2-1: Beat-wise execution in MVE



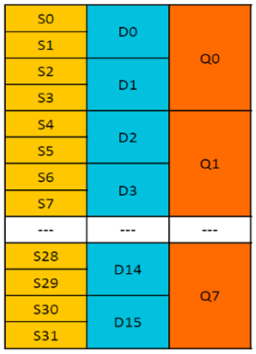
By overlapping the execution cycle of vector load and vector data processing instructions, the pipeline diagram above demonstrates that with Helium technology, Cortex-M55 based

products can achieve a significant increase in performance and energy efficiency (Reference 4). By using optimized instruction scheduling, we can see these benefits on signal processing and ML applications compared to previous Cortex-M based products.

The key characteristics of the vector processing hardware include:

- 8 vector registers, each of which is 128-bit wide
- FPU registers reused, with D0:D15 mapped to Q0:Q7 as shown in the figure below
- Over 150 new scalar and vector instructions
- Maximized use of both vector engine hardware and general-purpose registers simultaneously
- A range of new features enhancing the MVE capability, including complex data processing and interleaved memory accesses

Figure 2-2: Register implementation



Helium technology is designed to provide versatile processing capabilities for multiple vector data types including:

- Vectored integer and fixed-point (MVE-I): 32-bit, 16-bit, 8-bit
- Vectored floating-point (MVE-F): single-precision and half-precision arithmetic

In addition, the Cortex-M55 processor also supports the following scalar floating-point data types:

- Scalar floating-point: double, single, and half-precision arithmetic

Like other Cortex-M processors, the Cortex-M55 processor is highly configurable. Both Helium and floating-point support are optional with five supported combinations, as follows:

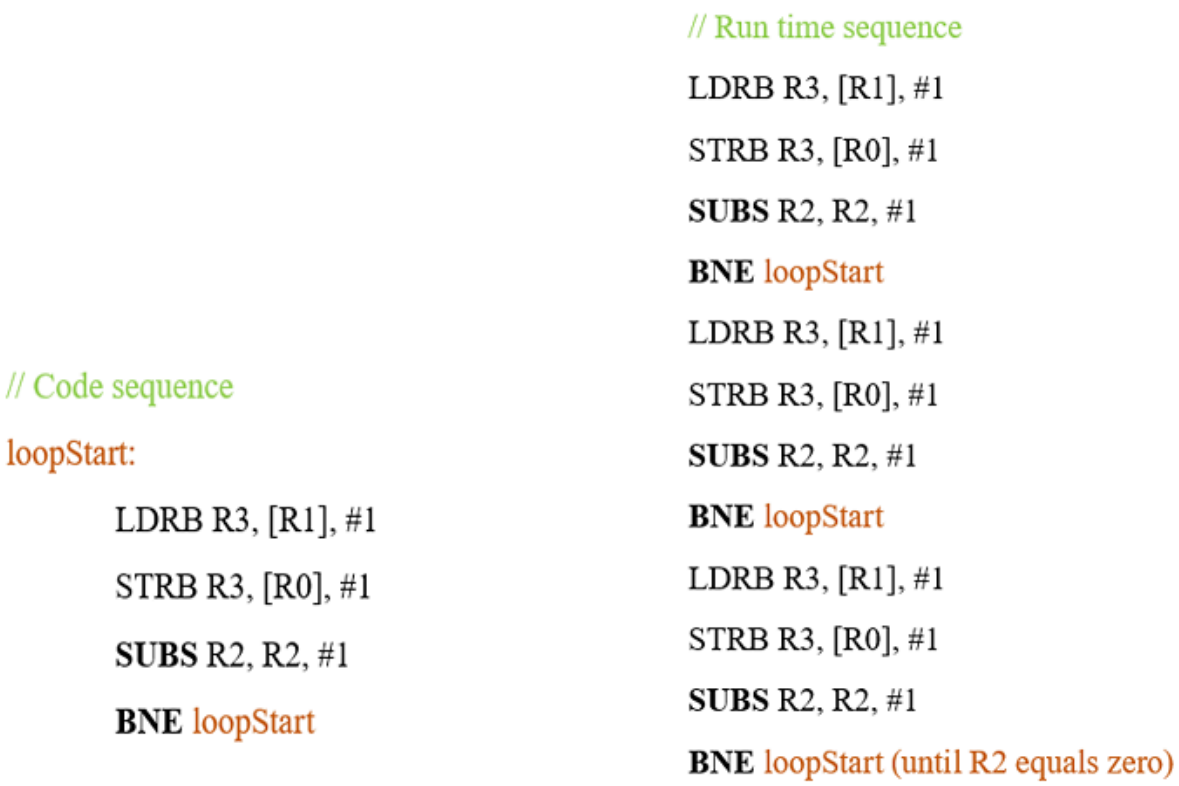
Table 2-1: Configurations supported in Cortex®-M55

Configuration	Base	FPU	MVE-I	MVE-F
1	Yes			
2	Yes	Yes		
3	Yes		Yes	
4	Yes	Yes	Yes	
5	Yes	Yes	Yes	Yes

2.2 Low Overhead Branch (LOB) extension

To minimize area and energy consumption, most of the previous Cortex-M processors were not built with branch predictors. As a result, the runtime overhead of program loops due to loop counter update and branch penalty is often unavoidable in those processors. This is demonstrated in the following diagram, where a simple byte copying loop which contains just 4 instructions, has a runtime loop overhead of 50% in the instruction count.

Figure 2-3: Loop execution without LOB cache



Because signal processing and machine learning applications often contain many loop operations, there is a strong demand for a solution to optimize loop execution and reduce the loop overhead. As a result, the Armv8.1-M architecture introduced several new loop support instructions, known as the Low Overhead Branch (LOB) extension (Reference 5). When using LOB, the loop structure starts with either a `WLS` or `DLS` instruction:

- `WLS` (while-loop start) instruction, which specifies the loop count, and branches to the end of loop if the loop counter is already zero.
- `DLS` (Do-loop start), which specifies the loop count but does not check the loop counter before entering the loop.

The loop structure ends with an LE (loop-end) instruction which decrements the loop counter (R14) and branches to the start of loop if the loop counter is not zero. The key element that makes this implementation efficient is that the processor can cache the loop address information (the starting and ending positions of the loop) provided by the LE instruction to a cache called LO_BRANCH_INFO during the first iteration. From the next iteration, the branching to the start of the loop happens without executing the LE instruction, drastically reducing the loop overhead.

Combining Helium with the new Low Overhead Branch extension, the performance of signal processing with Armv8.1-M processors can be several times better than using a traditional SIMD approach as in the popular Cortex-M4 processor. A simplified code sequence that reads an array (VLDR) and uses vectored multiply-accumulate (VMLA) is shown in left-hand side of the code sample below. The right-hand side shows the expanded execution sequence where the WLS and LE instructions do not consume any execution cycles after the first iteration because all loop control information is cached.

Figure 2-4: Loop execution with LOB cache enabled

```
// Code sequence
    WLS
loopStart:
    VLDR
    VMLA
    LE
loopEnd:
```

```
// Runtime sequence
WLS
VLDR
VMLA
LE
VLDR
VMLA  // VMLA can overlap with
VLDR  // VLDR on next iteration
VMLA
VLDR
```

Note that the branch cache is disabled on reset. You must enable the branch cache to fully utilize the Low Overhead Branch extension. To enable the branch cache, set the LOB bit in the Configuration Control Register (CCR) and then issue an Instruction Synchronization Barrier (ISB) instruction. The processor must be in privileged mode to read from and write to the Configuration Control Register (CCR). If the TrustZone® Security Extension is implemented, the CCR.LOB bit is banked between Security states, so it must be enabled for each Security state that uses the LOB Extension. The following assembly code sequence demonstrates how to enable the branch cache for the current Security state when running in privileged mode (Reference 6).

```
CCR EQU 0xE000ED14
LDR R0, =CCR; Read CCR
LDR r1, [R0]; Set bits 19 to enable LOB
ORR R1, R1, # (0x8 << 16)
STR R1, [R0]; Write back the modified value to the CCR
DSB; Wait until the write is completed.
ISB; Reset pipeline now LOB is enabled.
```

The use of the DSB (Data Synchronization Barrier) and ISB instruction (Instruction Synchronization Barrier) is essential to force the change of LOB setting to take effect.

When using a C/C++ programming environment with CMSIS-CORE, the SystemInit() function which executes within the reset handler enables the LOB feature. The following portion of the example Cortex-M55 system initialization C code (Reference 7) enables the LOB feature:

```
/* Enable Loop and branch info cache */

SCB->CCR |= SCB_CCR_LOB_Msk;
__DSB();
__ISB();
```

The following simulation log excerpt shows that as soon as an `LE` or `LETP` instruction is executed, the `LO_BRANCH_INFO` cache is populated.

```
cpu0 IT (1869) 100028a8 f01fc00d T thread_s : LETP     LR, #-0xc
cpu0 R r14 000001f0
cpu0 R LO_BRANCH_INFO.VALID 00000001
cpu0 R LO_BRANCH_INFO.JUMP_ADDR 10002894
cpu0 R LO_BRANCH_INFO.END_ADDR 100028a8
cpu0 IT (1870) 10002894 ecbf0f04 T thread_s : VLDRH.S32 Q0, [R7], #0x8
```

There are additional branch hint instructions introduced in the Armv8.1-M architecture which may be able to take advantage of the hardware introduced by low overhead loop to enable better branching performance.

2.3 Auto-vectorization support in C compilers

To make use of the Helium and LOB features, software developers can use the optimized CMSIS-DSP and CMSIS-NN libraries to help speed up a range of specialized data processing operations. In addition, C compilers such as Arm Compiler 6 have been updated to support auto-vectorization using the Helium and LOB features. Auto-vectorization support is not new in Arm Compiler, as it has been available for Arm Neon technology for some time (Reference 8). Since the release of the Armv8.1-M architecture, Arm Compiler 6 has been updated to support auto-vectorization using Armv8.1-M features (Reference 9).

To enable or disable auto-vectorization in Arm Compiler 6, use the following command-line options:

- `-fvectorize` to enable auto-vectorization. This is turned on by default if using an optimization level option of `-O2` or above.
- `-fno-vectorize` to disable auto-vectorization.

The auto-vectorization feature is available only when MVE is available in the compilation target, for example `-mcpu=cortex-m55`.

3 Test environment details

To demonstrate the benefit of the Helium and LOB features to common applications, a range of benchmarks are tested on a Cortex-M55 based system.

The details of the test environment are as follows:

Compiler

Arm Compiler 6.16

Compiler auto-vectorization diagnostic options

```
-Rpass=loop-vectorize -Rpass-analysis=loop-vectorize -Rpass-missed=loop-vectorize
```

Frequency

32MHz

MVE Compiler flags

```
-mcpu=cortex-m55
```

No-MVE Compiler flags

```
-mcpu=cortex-m55+nomve
```

No-LOB cache

```
-mcpu=cortex-m55+nolob
```

Platform

FPGA/RTL emulation, Cortex-M55 r0

OS

BareMetal

4 Coding considerations

To get the best results from Helium technology, software developers might need to modify their source code to maximize the use of vectorization. Not all loops can be vectorized. Examples of code that cannot be vectorized, or is difficult to vectorize include the following:

- Loops with interdependencies between different loop iterations
- Loops with break clauses
- Loops with complex conditions
- Loops where the number of iterations is unknown at the start
- Loops with double-precision floating point processing, although low-overhead branching might still be used in these cases
- Loops that involve operations that cannot be vectorized, for example memory barriers

To help software developers optimize their code, Arm Compiler 6 and LLVM provide the following vectorization diagnostic options:

- `-Rpass=loop-vectorize`
- `-Rpass-analysis=loop-vectorize`
- `-Rpass-missed=loop-vectorize`

The following sections show loops in the CoreMark workload that cannot be vectorized because of the reasons listed previously.

`src/coremark/coremark_v1.0/core_matrix.c:220:2: remark: loop not vectorized: value that could not be identified as reduction is used outside the loop [-Rpass-analysis=loop-vectorize]`

```
ee_s16 matrix_sum(ee_u32 N, MATRES *C, MATDAT clipval) {
  MATRES tmp=(MATDAT)0,prev=0,cur=0;
  ee_s16 ret=0;
  ee_u32 i,j;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      cur=C[i*N+j];
      tmp+=cur;
      if (tmp>clipval) {
        ret+=10;
        tmp=0;
      } else {
        ret += (cur>prev) ? 1 : 0;
      }
      prev=cur;
    }
  }
  return ret;
}
```

src/coremark/coremark_v1.0/core_main.c:274:3: remark: loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-vectorize]

```
for (i=0 ; i<default_num_contexts; i++) {
  results[i].err=0;
  if ((results[i].execs & ID_LIST) && (results[i].crclist!
=list_known_crc[known_id])) {
    ee_printf("[%u]ERROR! list crc 0x%04x - should be    0x%04x\n",i,
 results[i].crclist,  list_known_crc[known_id]) ;
    results[i].err++;
  }
```

src/coremark/coremark_v1.0/core_state.c:127:4: remark: the cost-model indicates that vectorization is not beneficial [-Rpass-missed=loop-vectorize]

```
while ((total+next+1)<size) {
  if (next>0) {
    for(i=0;i<next;i++)
      *(p+total+i)=buf[i];
    *(p+total+i)=',';
    total+=next+1;
  }
```

In addition, the following compiler directives are also available to help guide compilers to vectorize specific parts of application code:

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

Additional information is available in the Helium Programmer's Guide (Reference 16), which is available on the Arm Developer website.

5 Helium (MVE) performance analysis

This section of the guide explains the impact of MVE across different real-world workload applications with examples of vectorized loops and the corresponding vector instructions that provide the performance uplift.

The instruction statistics sections only include the most frequently executed vector instructions for each benchmark.

5.1 EEMBC-ConsumerBench MVE performance analysis

The first example is a code excerpt from the EEMBC® ConsumerBench™ benchmark, (Reference 10).

5.1.1 Code analysis of hpg

The following code shows a loop that is vectorized by a C compiler, highlighting the vectorized loop in the hpg benchmark source code that was obtained by passing the vectorization diagnostic options described previously.

```
src/eembc/consumer/filters/rgbhpg01/bmark_lite.c:236:4: remark: vectorized loop
(vectorization width: 8, interleaved count: 1) [-Rpass=loop-vectorize]
```

```
for (h = 1; h < (Height - 1); h++) {
  for (w = 1; w < (Width - 1); w++) {
    Center = (Width * h) + w;

    PelValue = (Short) (
    /* top row */
    (F11 * ImageInPtr[Center - Width - 1]) +
    (F21 * ImageInPtr[Center - Width]) +
    (F31 * ImageInPtr[Center - Width + 1]) +
    /* add middle row */
    (F12 * ImageInPtr[Center - 1]) +
    (F22 * ImageInPtr[Center]) +
    (F32 * ImageInPtr[Center + 1]) +
    /* add bottom row */
    (F13 * ImageInPtr[Center + Width - 1]) +
    (F23 * ImageInPtr[Center + Width]) +
    (F33 * ImageInPtr[Center + Width + 1])
    );
  }
```

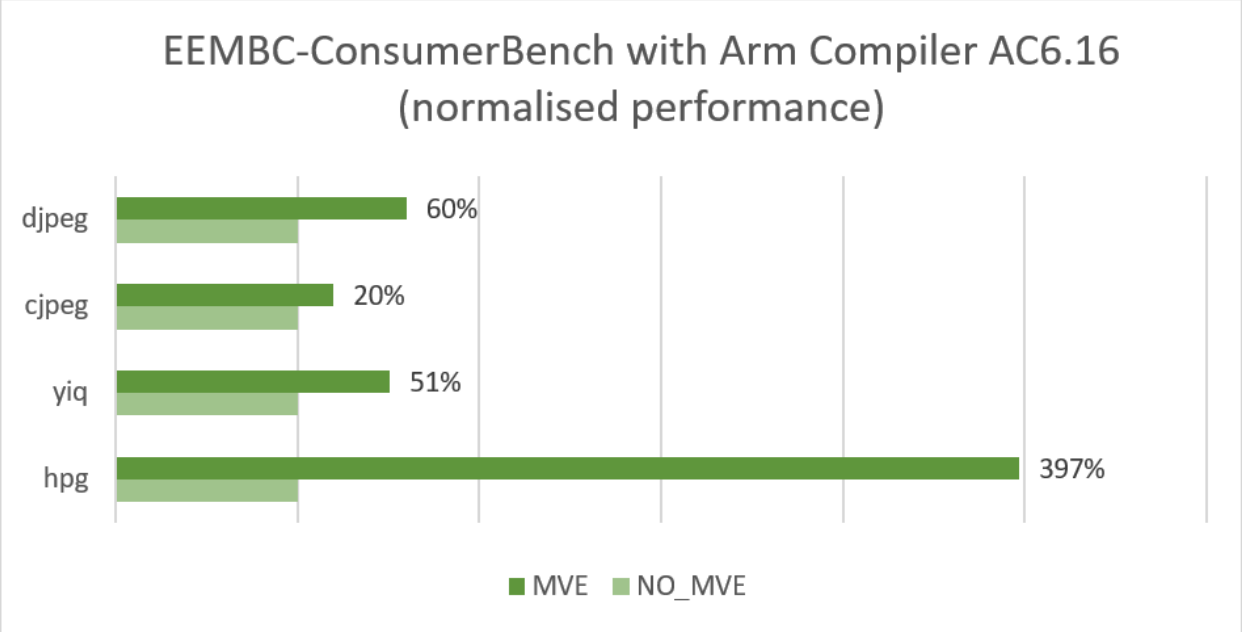
The following figure shows the code generation differences between MVE and non-MVE at assembly level for the hpg test:

Figure 5-1: Code generation differences between MVE and non-MVE for the hpg benchmark

MVE	NO_MVE
VADD.I16 q0,q0,q1	ADD r1,r1,r3
VLDRB.U16 q1,[r4,#-7]	LDRB r3,[r5,#0x295]
VLDRB.U16 q2,[r5],#8	ADD r1,r1,r2
VADD.I16 q0,q0,q2	LDRB r2,[r5,#0x296]
VMUL.I16 q1,q1,q4	ADD r1,r1,r3
VMUL.I16 q0,q0,q5	LDRB r3,[r5,#0x154]
VADD.I16 q0,q0,q1	ADD r1,r1,r2
VSHR.U16 q0,q0,#8	RSB r2,r3,r3,LSL #8
VSTRB.16 q0,[r7],#8	LDRB r3,[r5,#0x155]
LETP lr,#-0x5a	SUB r0,r0,r0,LSL #3
	RSB r3,r3,r3,LSL #8
	SUB r1,r1,r1,LSL #3
	ADD r0,r2,r0,LSL #2
	ADD r1,r3,r1,LSL #2

The EEMBC-Consumer performance results are as follows:

Figure 5-2: EEMBC-Consumer results



C-Flags: `--target=arm-arm-none-eabi -mcpu=Cortex-m55 -mfloat-abi=hard -mthumb -Omax`

5.1.2 Observations

The data in the previous figure shows that the EEMBC-ConsumerBench benchmarks have greatly benefitted from the MVE feature, with the hpg test showing an impressive 4x performance increase. The hpg test contains a high pass gray scale filter algorithm and uses MAC-based calculations to sharpen the image. The EEMBC-Consumer benchmark consists of frequently used algorithms in image processing and an increase in performance indicates that Cortex-M55 with MVE is well suited to DSP intense applications.

5.1.3 Instruction profiling for the MVE run

The following table shows instruction statistics for EEMBC-Consumer:

Table 5-1: Instruction statistics for EEMBC-Consumer

Instruction	Percentage
VADD.I16	34.99%
VLDRB.U16	39.37%
VMUL.I16	8.75%
VSHR.U16	4.37%
VSTRB.16	4.37%

The above analysis shows how MVE has impacted the code generation in the hpg benchmark and the increase in performance made possible by the Helium-based instructions.

5.2 EEMBC-TeleBench™ MVE performance analysis

Another benchmark that greatly benefits from the MVE feature is the EEMBC TeleBench telecommunications benchmark (Reference 11).

5.2.1 Code analysis of AutCorSpeech

The following code shows one of the vectorized loops in the AutCorSpeech workload that impacts the benchmark's performance.

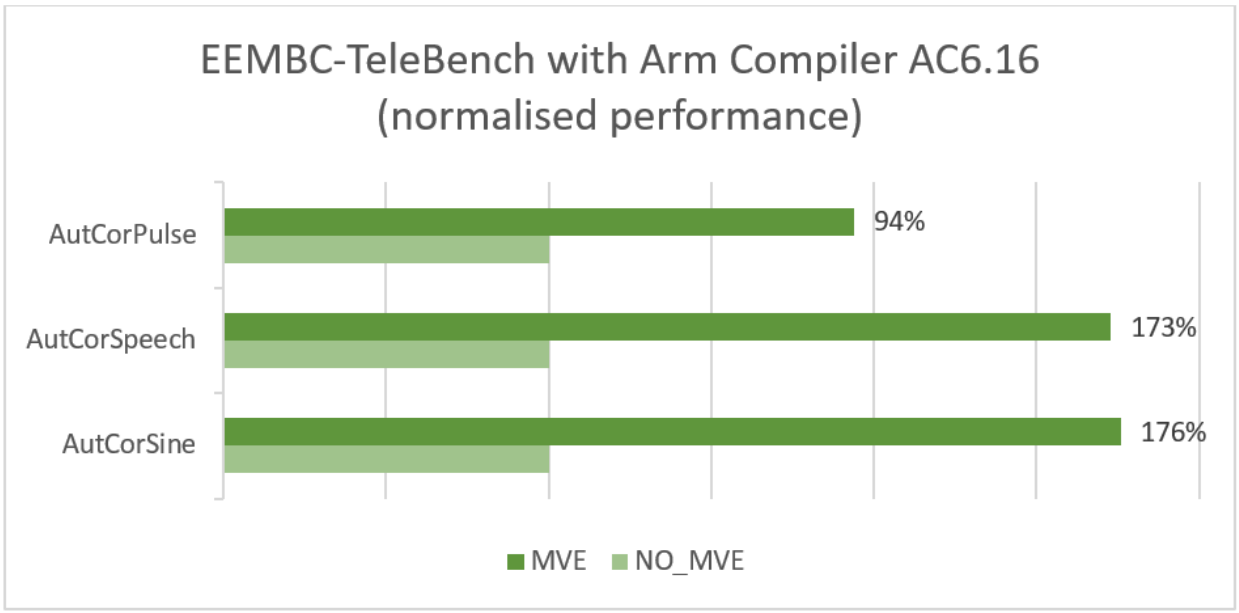
```
src/eembc/telecom/autcor00/autcor00.c:111:9: remark: vectorized loop (vectorization
width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

```
/* Compute AutoCorrelation */
for (lag = 0; lag < NumberOfLags; lag++) {
  Accumulator = 0;
  LastIndex = DataSize - lag;
  for (i = 0; i < LastIndex; i++) {
      Accumulator += ((e_s32) InputData[i] * (e_s32) InputData[i+lag]) >> Scale;
  }
```

The inner `for` loop is where the AutoCorrelation algorithm is used. From the source code, the inner loop uses multiply, accumulate, and add arithmetic instructions. With Helium enabled, vectorization benefits the execution of inner loop and provides a significant performance improvement.

The EEMBC-Telecom performance results are as follows:

Figure 5-3: EEMBC-Telecom results



C-Flags: `--target=arm-arm-none-eabi -mcpu=Cortex-m55 -mfloat-abi=hard -mthumb -Omax`

5.2.2 Observations

The impact of MVE in the Telecom benchmark is clearly seen in AutoCorrelation-based tests with AutCorSpeech and AutCorSine tests showing a performance increase of 176% and 173% respectively. This indicates that the MVE functionality helps boost the signal processing capabilities of the processor.

5.2.3 Instruction profiling for the MVE run

The following table shows instruction statistics for EEMBC Telecom:

Table 5-2: Instruction statistics for EEMBC Telecom

Instruction	Percentage
`VADDVA.U32`	19.30%
`VLDRH.S32`	38.61%
`VMUL.I32`	19.30%
`VSHR.S32`	19.30%

From the instruction profiling table, vector instructions contribute 97% of the total instructions run.

5.3 EEMBC-FPMark™ MVE performance analysis

General mathematic-intensive workloads can also benefit from MVE. In this example, we see a range of performance gains with the single-precision version of EEMBC-FPMark (Reference 12). Only the single-precision version of the FPMark and small data set are used in the following study.

5.3.1 Code analysis of inner_product_sml_1k_sp

The EEMBC-FPMark benchmark contains more than 20 kernels. Some examples of the vectorized loops are shown below.

```
src/eembc_fpmark-v1.5/benchmarks/fp/loops/loops.c:687:9: remark: vectorized loop
(vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

Kernel 3: Inner product

```
for ( l=1 ; l<=loop ; l++ ) {
  for ( k=0 ; k<n ; k++ ) {
    q += z[k]*x[k];
  }
#if (BMDEBUG && DEBUG_ACCURATE_BITS)
  th_printf("\niprod %d:",debug_counter++);
  th_print_fp(q);
#endif
}
```

```
src/eembc_fpmark-v1.5/benchmarks/fp/loops/loops.c:728:5: remark: vectorized loop
(vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

Kernel4: Banded linear equations

```
m = ( 1001-7 )/2;
  for ( l=1 ; l<=loop ; l++ ) {
    for ( k=6 ; k<1001 ; k=k+m ) {
      lw = k - 6;
      temp = x[k-1];
```

```
src/eembc_fpmark-v1.5/benchmarks/fp/loops/loops.c:732:13: remark: vectorized loop
(vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

Kernel 4: Banded linear equations (inner loop)

```
m = ( 1001-7 )/2;
  for ( l=1 ; l<=loop ; l++ ) {
    for ( k=6 ; k<1001 ; k=k+m ) {
      lw = k - 6;
      temp = x[k-1];
      for ( j=4 ; j<n ; j=j+5 ) {
```

```
        temp -= x[lw]*y[j];
        lw++;
      }
      x[k-1] = y[4]*temp;
    }
```

src/eembc_fpmark-v1.5/benchmarks/fp/loops/loops.c:950:3: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]

Kernel 9: Integrate predictors

```
for ( i=0 ; i<n ; i++ )
  ret+=px[i*13];
ret/=(e_fp)n; /* feedback to make sure all loops get executed */
```

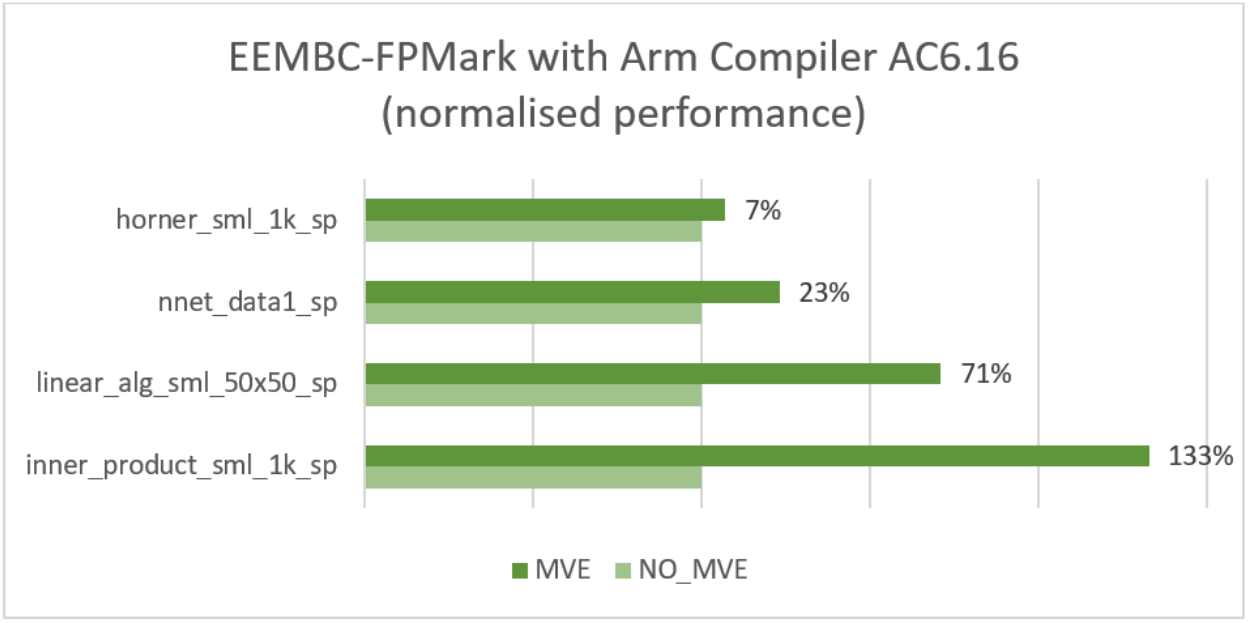
src/eembc_fpmark-v1.5/benchmarks/fp/loops/loops.c:1078:9: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]

Kernel 12: First difference

```
reinit_vec(p,x,n);
for ( l=0 ; l<=loop ; l++ ) {
  reinit_vec(p,y,n+1);
  for ( k=0 ; k<n ; k++ ) {
    x[k]+= y[k+1] - y[k];
```

The EEMBC-FPMark-SP performance results are as follows:

Figure 5-4: EEMBC-FPMark-SP results



C-Flags: `--target=arm-arm-none-eabi -mcpu=Cortex-m55 -mfloat-abi=hard -mthumb -Omax`

5.3.2 Observations

From the data, MVE has increased the performance of the FPMark-SP benchmarks with inner_product and linear algorithm tests showing a 2.3x and 1.7x improvement respectively. As these FPMark scores indicate the computational capabilities of the processor, an increase in performance numbers shows that Cortex-M55 with MVE enabled has higher computational power.

5.3.3 Instruction profiling for the MVE run

The following table shows instruction statistics for EEMBC FPMark-SP:

Table 5-3: Instruction statistics for EEMBC FPMark-SP

Instruction	Percentage
VLDRW.U32	9.38%
VFMA.F32	4.55%
VSTRW.32	0.27%
VADD.F32	0.24%

5.4 EEMBC-AutoBench™ MVE performance analysis

We have also investigated the results of running EEMBC AutoBench (Automotive benchmark) (Reference 13) and see performance gain in a range of workloads.

5.4.1 Code analysis of aifftr01

The vectorized loops in the respective source code files for the aifftr01 workload are shown in the following code.

```
src/eembc/automotive/aifftr01/bmark_lite.c:402:5: remark: vectorized loop
(vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

```
/* Compute the bit reversal indices  -- used by all the instances */
  for( i_1 = 0 ; i_1 < NUM_POINTS ; i_1++ )
  {
    index = i_1 ;
    brIndex = 0 ;
    for( j_1 = 0 ; j_1 < FFT_LENGTH ; j_1++ )
    {
      brIndex <<= 1 ;
      if( 0x01 &index )
      {
        brIndex |= 0x01 ;
      }
      index >>= 1 ;
    }
    bitRevInd[i_1] = brIndex ;
```

```
  }
```

src/eembc/automotive/aifftr01/bmark_lite.c:830:9: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]

```
/* Compute power spectrum */
  for( i_3 = 0 ; i_3 < NUM_POINTS ; i_3++ )
  {
    /* This can all easily overflow! */
    realData_3[i_3] *= realData_3[i_3] ;
    realData_3[i_3] += imagData_3[i_3] * imagData_3[i_3] ;
  }
```

src/eembc/automotive/aifftr01/bmark_lite.c:779:17: remark: vectorized loop (vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]

```
/* Process butterflies with the same twiddle factors */
  for( i_3 = j_3 ; i_3 < NUM_POINTS ; i_3 += n1_3, passCount_3++ )
  {
    l_3 = i_3 + n2_3 ;
    realLow_3 = &realData_3[l_3] ;
    imagLow_3 = &imagData_3[l_3] ;
    realHi_3 = &realData_3[i_3] ;
    imagHi_3 = &imagData_3[i_3] ;
```

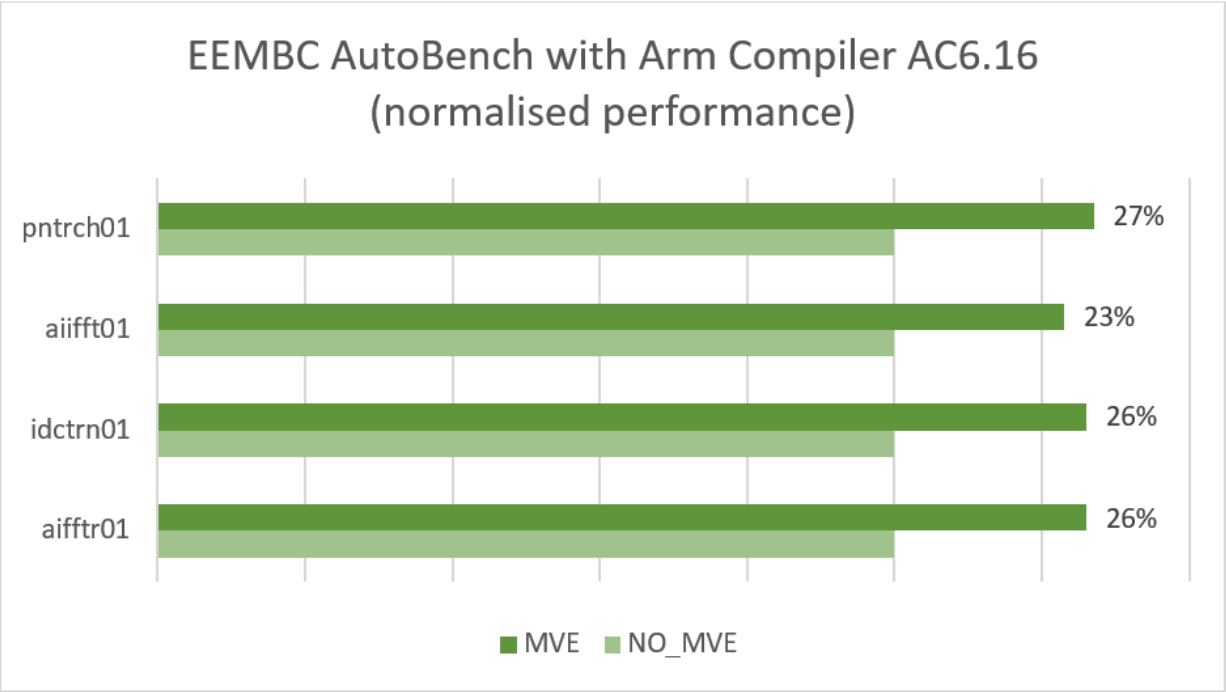
This benchmark implements the Fast-Fourier Transformation algorithm, which includes the following steps:

1. Input values are fetched from arrays.
2. Time domain values are converted to the equivalent frequency domain values.
3. The power spectrum is calculated.

As can be seen from these steps, this benchmark is computationally intensive and would benefit from MVE.

The EEMBC-Automotive performance results are as follows:

Figure 5-5: EEMBC-Automotive results



C-Flags: `--target=arm-arm-none-eabi -mcpu=Cortex-m55 -mfloat-abi=hard -mthumb -Omax`

5.4.2 Observations

This suite consists of benchmarks used in automotive, industrial, and general-purpose areas. From the data, there is an average of 25% improvement in the performance of Cortex-M55 with MVE enabled across these benchmarks, making it a suitable choice for a combination of generic, computation-based, and signal processing applications.

5.4.3 Instruction profiling for the MVE run

The following table shows instruction statistics for EEMBC AutoBench:

Table 5-4: Instruction statistics for EEMBC-Consumer

Instruction	Percentage
`VMLADAVA.U32`	3.41%
`VLDRW.U32`	2.21%
`VADD.I32`	0.48%
`VSTRW.32`	0.43%
`VORR`	0.36%

5.5 CMSIS-CFFT/FIR MVE performance analysis

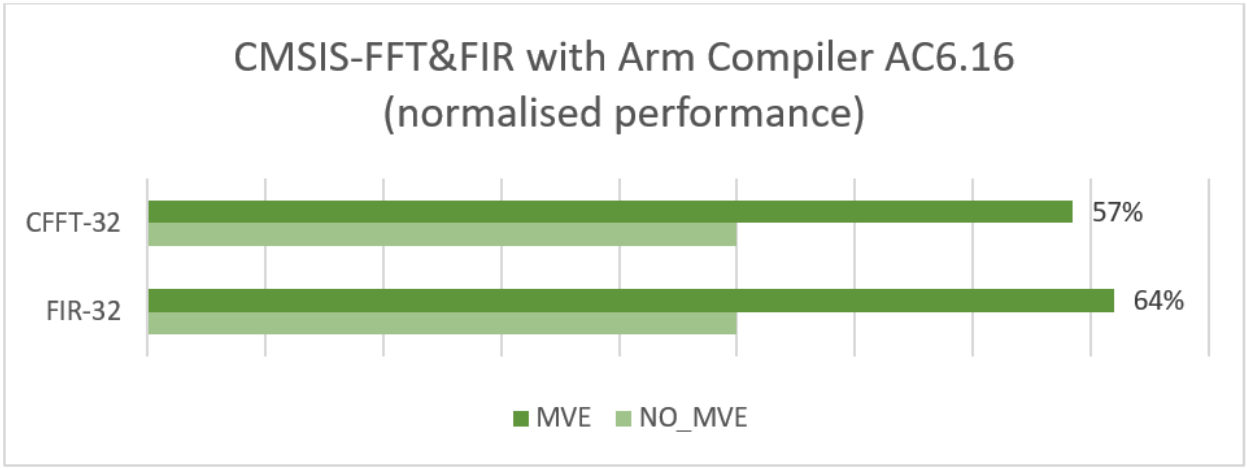
One of the common workloads in DSP applications consists of transformation functions such as Fast Fourier Transform (FFT).

The Fast Fourier Transform is an efficient algorithm for computing the Discrete Fourier Transform (DFT). The FFT can be orders of magnitude faster than the DFT, especially for long lengths. The algorithms described in this section operate on complex data.

The FIR filter algorithm is based on a sequence of multiply-accumulate (MAC) operations. This set of functions implements Finite Impulse Response (FIR) filters for Q7, Q15, Q31, and floating-point data types. The functions operate on blocks of input and output data and each call to the function processes block size samples through the filter.

To demonstrate how DSP operations benefit from MVE, we used a C version of FFT in the CMSIS-DSP library (Reference 14) and compiled the code with and without the MVE option. In practice, if software developers use the hand-optimized FFT code in the CMSIS-DSP library, the performance gain would be even better.

Figure 5-6: CMSIS-DSP FFT and FIR results



C-Flags: `--target=arm-arm-none-eabi -mcpu=Cortex-m55 -mfloat-abi=hard -mthumb -Omax`

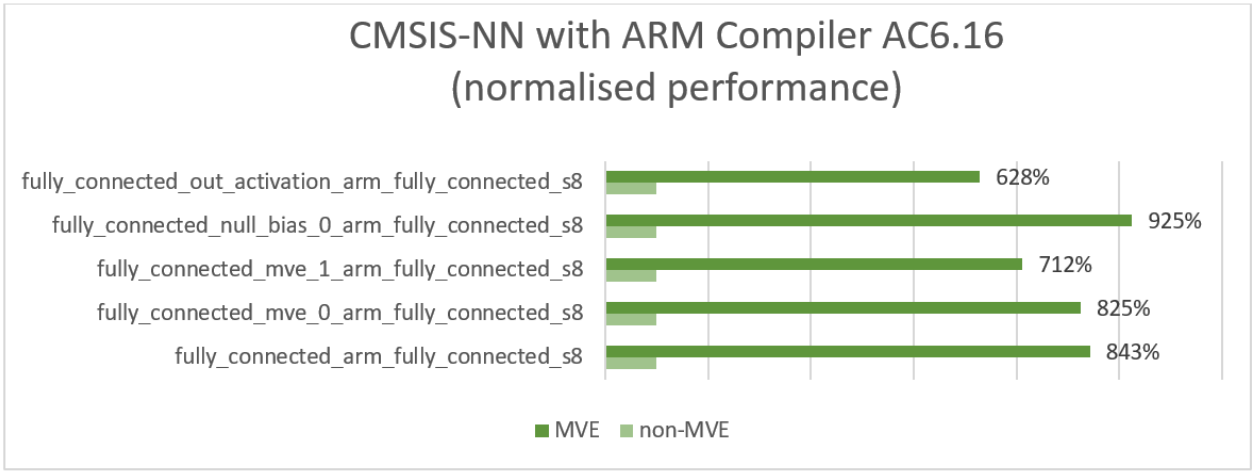
5.5.1 Observations

The data shows that MVE improves performance significantly for signal processing algorithms such as FIR and FFT with improvements of 64% and 57% respectively over non-MVE.

5.6 CMSIS-NN

Similarly, a benchmark was run using the CMSIS-NN code with and without MVE support. The result of processing a fully connected layer is shown below. A fully connected layer is basically a matrix-vector multiplication with bias. The matrix is the weights, and the input and output vectors are the activation values.

Figure 5-7: CMSIS-NN fully connected network results



C-Flags: `--target=arm-arm-none-eabi -mcpu=Cortex-m55 -mfloat-abi=hard -mthumb -Ofast`

5.6.1 Observations

Data shows that efficient neural network kernels can leverage the advantages of MVE to increase performance more than 7x and reduce the memory footprint.

6 Low Overhead Branch (LOB) performance analysis

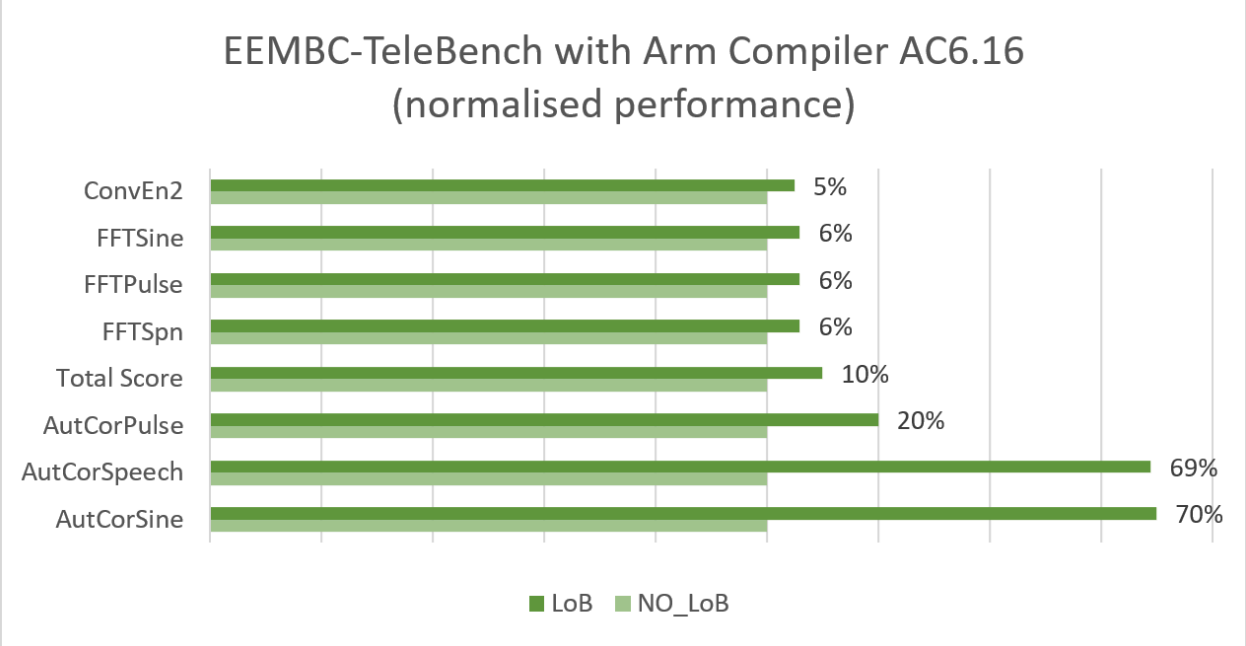
Because the Cortex-M55 can be configured without Helium, we also carried out some analysis of performance gains using just the LOB feature, which is available even when Helium is omitted from the implementation. The impact of the LOB cache feature can be predominantly seen in the EEMBC-Telecom and EEMBC-FPMark benchmark results.

6.1 EEMBC-TeleBench LOB performance analysis

The first analysis of LOB benefit is based on EEMBC TeleBench (telecommunications benchmark) (Reference 11).

The performance gain is shown in the following chart:

Figure 6-1: EEMBC-Telecom results



6.1.1 Observations

Most of the EEMBC-Telecom benchmarks benefit when the LOB cache is enabled. These benchmarks are heavily based on loops and when the low overhead loop is introduced, this enhances performance by optimizing the loop execution and reducing the branch overhead. Similar performance gain can also be achieved using branch predication features in high-end processors.

However, branch predication hardware typically has a larger area and power impact to a processor design.

6.1.2 Instruction analysis for AutCorSpeech

The following table shows instruction analysis for AutCorSpeech:

Table 6-1: Execution overhead for the loop is reduced when the LOB cache is enabled

LE (loop end) with LOB cache enabled	LE (loop end) without LOB cache enabled
0.04%	5.12%

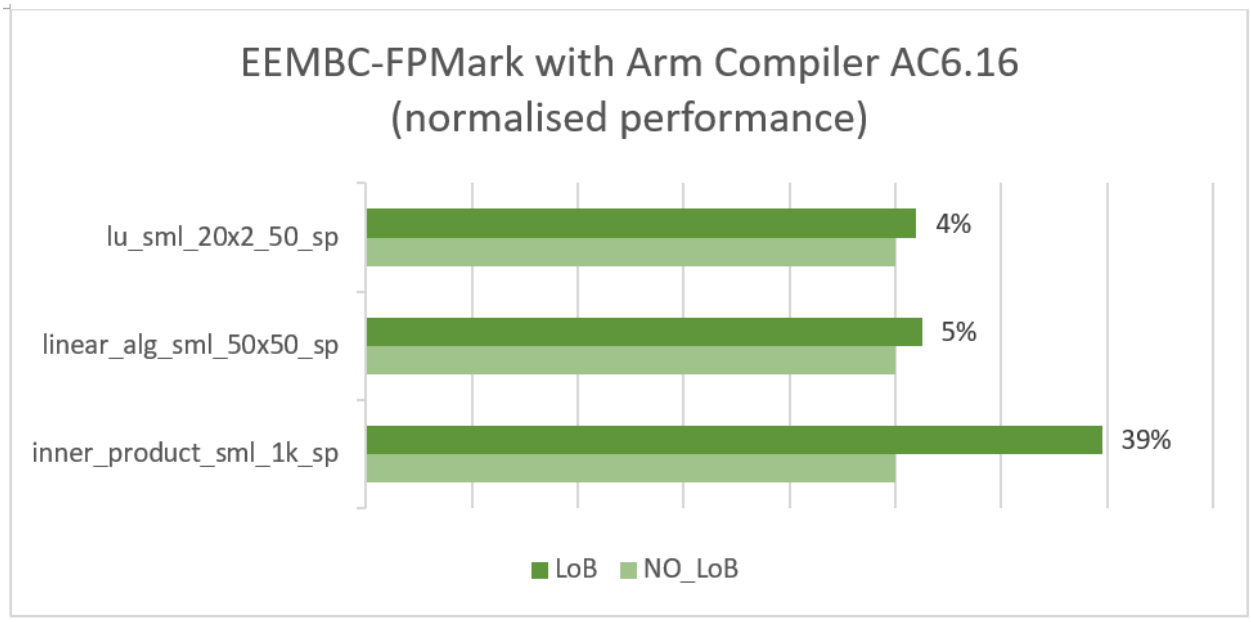
The above table illustrates that when the Branch_info_cache is enabled, the `LE` instruction is executed only once for every loop as the branch information is being cached after the first iteration, drastically reducing the loop overhead.

6.2 EEMBC-FPMark LOB performance analysis

The second analysis of LOB benefit is based on the EEMBC FPMark benchmark (Reference 12).

The performance gain is shown in the following chart:

Figure 6-2: EEMBC-FPMark-SP results



6.2.1 Observations

The `inner_product_sml_1k_sp` benchmark performance has significantly increased with the low overhead loop feature, showing a 39% performance improvement compared to not using the LOB feature.

7 Conclusions

In summary, after seeing significant performance improvements with MVE and LOB extensions we can conclude that software developers can benefit from Helium and LOB features with standard C and C++ code. The benefit is more significant in applications that contain data processing operations which can be vectorized by C compilers, as well as loop-intensive applications. Even when a Cortex-M55 processor is configured without Helium support, we can still see the benefit of the LOB extension, which is a part of the Armv8.1-M architecture. Of course, the performance and efficiency benefits can be even higher when using the CMSIS-DSP and CMSIS-NN libraries, which are already optimized for the Cortex-M55 processor.

Combining Armv8.1-M features with an energy-efficient microarchitecture design, the Cortex-M55 processor is Arm's most energy-efficient Cortex-M processor for applications that require signal processing and machine learning operations. Further, with auto-vectorization support in C compilers, some of the general C code can also be vectorized. As a result, some of the general application code can also take advantage of the Helium technology to achieve better performance and energy efficiency.

8 References

Here are the references cited in this guide:

1. Arm Neon Technology
2. Arm Helium Technology
3. Arm Cortex-M Processors
4. Details of overlapping pipeline in Helium
5. Details of the Low-overhead branch extension
6. Details related to enabling Low-Overhead Branch cache
7. Example Cortex-M55 system initialization code
8. Auto-vectorization support for Arm Neon Technology
9. Arm Compiler 6.16 command line options for enabling and disabling auto-vectorization
10. EEMBC ConsumerBench™
11. EEMBC Telecom
12. EEMBC FPMark
13. EEMBC Automotive
14. CMSIS-DSP library
15. CMSIS-NN library
16. Helium Programmer's Guide