Blending DSP and ML features into a low-power general-purpose processor – how far can we go?

Joseph Yiu, Distinguished Engineer, Arm

White Paper

With increasing signal processing requirements in various types of IoT and embedded systems, we have seen a number of new chips on the market that combine both a digital signal processor (DSP) and a general-purpose processor to address these increased processing demands. While these suit high-performance devices where silicon area and power are less of a concern, small embedded devices and could be difficult to program (e.g. need multiple toolchains due to the heterogeneous nature) and can have limitations.

To address this challenge, Arm has been working on technologies that boost the signal processing and machine learning capabilities for future embedded processors. In this paper, we will look at how the Arm Cortex-M55 processor with Helium technology compares to features found on traditional DSPs, and some of the fundamental differences between VLIW (Very Long Instruction Word) architecture and the Helium approach to the processor's pipeline design. We will also look into how the processing requirements affect the processor's level-one memory system design and system design considerations.

1. Background

While machine learning technologies are getting a lot of attention in the industry, these often come with increasing needs for traditional compute, in particular around signal processing areas, as these two types of computing tasks often go hand in hand. For example, in voice command control applications, a range of signal processing tasks like noise cancellation and beamforming are needed to provide high-quality data input for the machine learning processing, which is often carried out by neural networks.

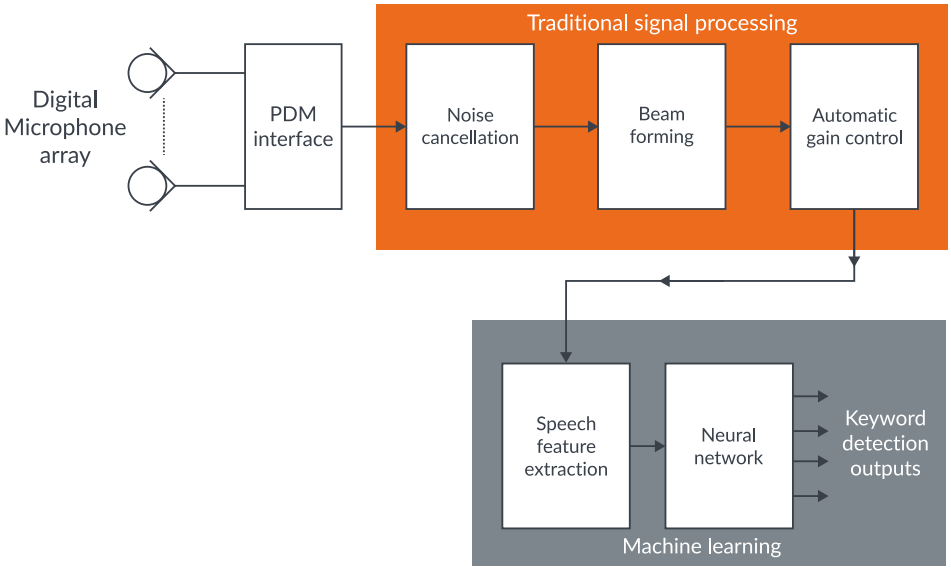


Fig. 1. Simple voice command control might involve a range of signal processing and ML operations

We have seen a number of projects using Arm Cortex-M4, Cortex-M7 and Cortex-M33 processors demonstrating keyword spotting and simple voice command control. As customer's expectation increases over time, there is a need to implement more sophisticated algorithms requiring higher processing capability, allowing more features to be added. In the context of keyword spotting, this improves the quality and number of keywords to be detected.

Of course, some system-on-chip (SoC) designers would just say "let's add a DSP into this chip" as a natural way to improve compute performance, especially with some of the modern DSPs that can deliver very high processing capabilities. However, doing so could potentially end up with a relatively large and power-hungry SoC design and break compatibility with existing software. Most of the IoT devices need a general-purpose processor to handle IoT software stacks, security and general control operations. Combining signal processing and ML capabilities by extending embedded general purpose architecture would bring many benefits, allowing lower cost and complexity while benefiting from the same programer's model.

2. What's New?

To address these signal processing and machine learning workloads needs for the next generation of deeply embedded systems, Arm has been busy working on a new technology called Arm Helium technology, a vector extension designed for low-power embedded systems such as an Arm Cortex-M processor. Helium was announced in 2019, and the first processor supporting Helium, a part of the Armv8.1-M architecture, is the new Cortex-M55 processor.

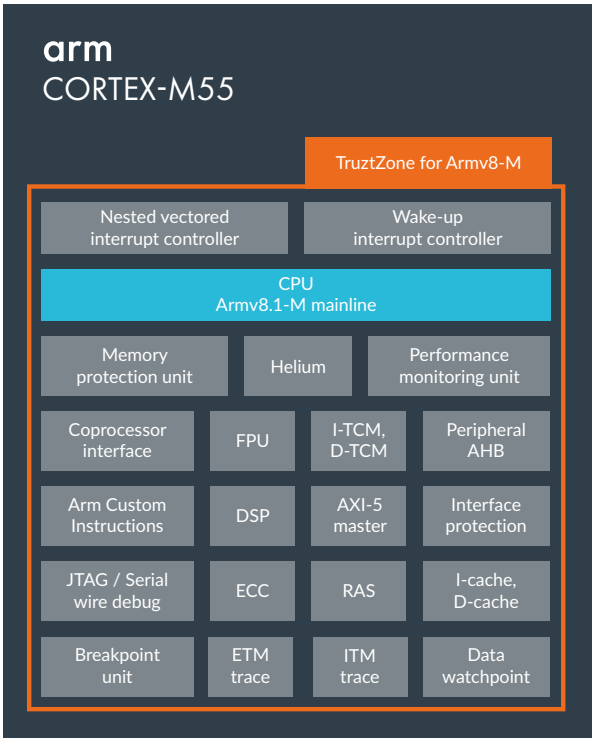


Fig. 2. Arm Cortex-M55 processor

The Cortex-M55 processor is the first Armv8.1-M processor. With Helium, typical signal processing performance can be up to 5x of the previous Cortex-M4, and neural network processing performance can be up to 15x of Cortex-M4 processor.

Please note that the new DSP extension, Helium is different from previous Armv7E-M DSP extensions in Cortex-M processors. Previously some of the Cortex-M processors like Cortex-M4, Cortex-M7, Cortex-M33 and Cortex-M35P processors already have signal processing capability supporting in-register SIMD extension using general-purpose registers. This allows the 32-bit internal datapath to be used for two 16-bit or four 8-bit data processing. While this DSP extension enables basic signal processing in small Cortex-M systems, this is not enough for the increasing workload and new processing requirements in machine learning applications. In Helium, the vector size is 128-bit and there is support for more data types and the instruction set support in Helium is much richer.

3. Signal Processing without VLIW

To enable higher processing capability in the Cortex-M55 processor, various innovative approaches have been developed and utilized. Many traditional techniques used by DSP for enhancing processing cannot be used on Cortex-M processors today. For example, many DSP architectures enable high performance by utilizing VLIW (Very Long Instruction Word) pipeline designs with two to five parallel execution slots, with different ranges of computational hardware for each slot.

Due to the nature of VLIW approach, software must be recompiled when moving between DSPs of different performance points. Whereas in Cortex-M, the software architecture generally follows upward compatibility to allow compiled program binaries to be reused on different processors, even if it might not be fully optimized without recompilation.

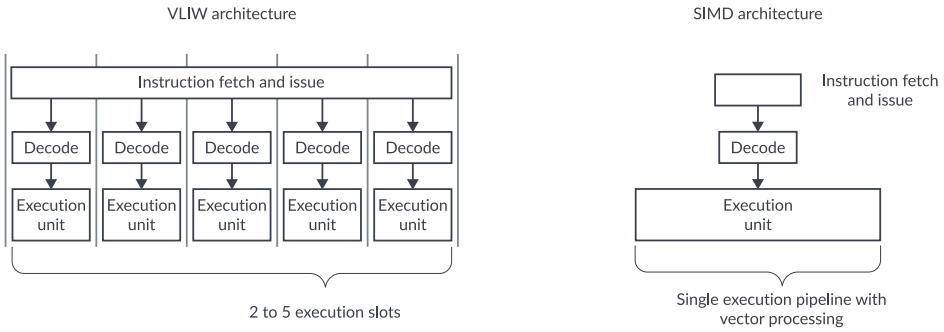


Fig. 3. VLIW (Very Long Instruction Word) in DSP vs SIMD (Single Instruction Multiple Data) in the Cortex-M55 processor

Helium architecture uses the Single Instruction Multiple Data (SIMD) technique, which is another way to increase processing performance. By using the SIMD approach, it is possible to design multiple processors with the same Helium architecture, maintaining full binary compatibility and to optimize the microarchitecture at different performance points, enabling ecosystem partners to make the most from their investments. SIMD approach is not only limited to Arm processors – some DSP also use the SIMD approach, and in some cases use a combination of SIMD and VLIW to maximize performance.

To enable better reusability, Helium and the Cortex-M55 processor are designed to be based on a conventional SIMD approach and utilize the vector extension and pipeline optimization techniques to enable higher processing performance.

It should be noted that Helium is not the only DSP extension in Arm. Arm Cortex-A processors have support for Neon (Advanced SIMD) for many years and SVE/SVE2 more recently. Instead of just porting Neon to Cortex-M, the Helium extension is a new design tailored for embedded architectures constraints. There are some similarities and differences when comparing Helium and Neon:

Similarities between Helium and Neon	Differences between Helium and Neon
✣ Registers in the floating-point unit are reused as vector registers ✣ Vector registers have size of 128-bit ✣ Some vector instructions are common	✣ Helium only has 8 vector registers ✣ Many Helium instructions use both vector and scalar registers In Neon only a few instructions use both ✣ Helium support new data types that Armv7-A/Armv8.0-A Neon does not support (e.g. fp16) ✣ Some features like low-overhead-branches, predication are not available in Neon ✣ Neon has half vector size (64-bit) for narrowing/widening, and interleaved-3 load/store. These are not supported on Helium ✣ Different widening/narrowing schemes

4. Key Design Goals

The key objective of the Cortex-M55 processor is not to get the highest signal processing and ML performance. Instead, efficiency is the key goal. At the same time, the Cortex-M55 design also needs to meet the requirements of a traditional Cortex-M processor including:

- Real-time/deterministic behaviour
- Security
- Ease-of-use and easy software migration
- Being cost-effective

To keep the Cortex-M55 processor energy-efficient and fit within the power budget for the majority of IoT endpoint systems, the Cortex-M55 processor internal datapath for its vector extension is 64-bit, which means it takes two clock cycles to operate on a 128-bit vector. However, the architecture behind Helium allows a processor's implementation to overlap execution cycles to enhance performance, providing that there are no hardware resource conflicts.

In a lot of signal processing functions (we refer them as DSP kernels), we often see code sequences having memory accesses instructions interleaved with data processing instructions. For example, in a simple FIR filter (Finite Impulse Response), we would have a sequence of interleaving vectored load and vectored multiply-accumulate.

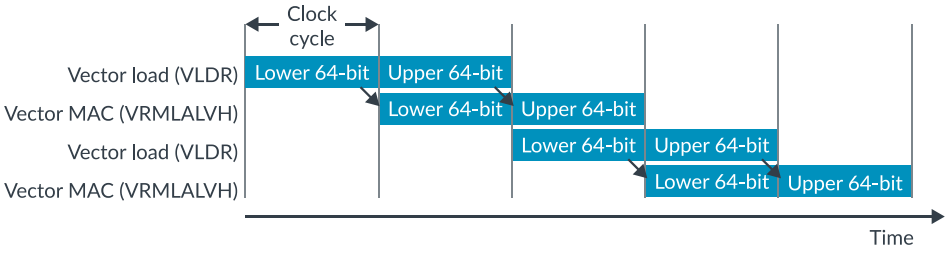


Fig. 4. Overlapping pipeline design in Cortex-M55 processor allows memory access and multiply-accumulate operations to be carried out at the same time

In the Cortex-M55 processor, such code sequence can take advantage of the pipeline design, which allows the execution stage of vector data to overlap – when loading the second half of the vector data from memory, the multiply-accumulate operation can be carried out on the first half of the data loaded from memory in the previous clock cycle. Such an arrangement enables the vector multiply-accumulate and vector load-store units to be occupied at every clock cycle, resulting in higher energy efficiency. In best instruction combinations, the Cortex-M55 processor can reach either.

- Two 32-bit load and MAC per cycle, or
- Four 16-bit load and MAC per cycle, or
- Eight 8-bit load and MAC per cycle

The instruction overlapping is not limited to the parallelism of memory accesses and vector processing. Cortex-M55 microarchitecture adopts a partition of the vector instruction set based on hardware resource requirements into separated instruction groups (e.g. a vector multiply and a vector shift are in different instruction groups). In simple terms, the pipeline partition allows overlapping of instructions belonging to vector load/store, vector integer, and vector floating-point categories.

In some cases, instruction overlapping cannot take place due to two adjacent instructions using the same hardware resource. This is called structural hazard. To help performance, additional unrolling techniques can be applied to optimize overlapping. As an example, here is a single precision floating complex dot product. The naive version would be written as following:

```
    vmov.i32    q7, #0x0     @ accumulator zeroing
    wlstp.32    lr, r2, 1f   @ low overhead while loop start tail predication
2b:
    vldrw.u32   q0, [r0], #16    @ 1st complex vector load
    vldrw.u32   q1, [r1], #16    @ 2nd complex vector load, no overlap
    vcmla.f32   q7, q0, q1, #0   @ 1st part of the complex multiplication,
                                 @ accumulation, can be overlapped with vldrw
    vcmla.f32   q7, q0, q1, #90  @ 2nd part of the complex multiplication,
                                 @ accumulation, no overlap
    letp        lr, 2b           @ low overhead while loop end
1f:
                                 @ partial parts summation
    vadd.f32    s0, s28, s30     @ real part
    vadd.f32    s1, s29, s31     @ imaginary part
```

Fig. 5. Non-optimized code due to back-to-back instructions that use same hardware resources

The use of back-to-back vector load (vldrw.u32) and back-to-back complex vector MAC (vcmla.f32) instruction does not allow this complex dot-product code sequence to reach the best of the Cortex-M55 performance because instructions belonging to the same group cannot be overlapped.

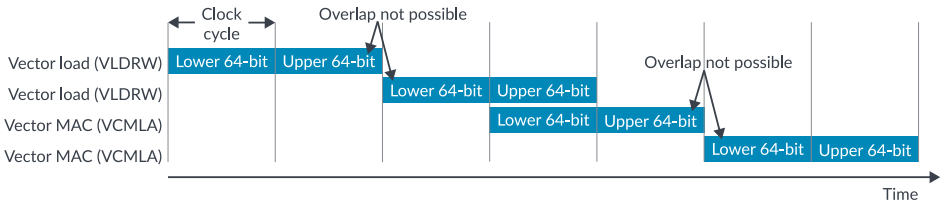


Fig. 6. Back to back vector operations of the same group cannot overlap

However, by unrolling the loop and using load-scheduling techniques to avoid back-to-back VLRDW and VCMLA, the same loop can run with all instructions being overlapped.

```
    lsrs        lr, r2, #2       @ 4 complex pair per loop handling
    vmov.i32    q7, #0x0         @ accumulator zeroing
                 @ load scheduling
    vldrw.u32   q0, [r0], #16    @ 1st complex vector load
    vldrw.u32   q0, [r1], #16    @ 2nd complex vector load

    wls         lr, lr, 1f       @ low overhead while loop start
2:
    vcmla.f32   q7, q0, q1, #0   @ 1st part of the complex multiplication,
                                 @ accumulation
    vldrw.u32   q2, [r0], #16    @ 1st complex vector load,
                                 @ can be overlapped with vcmla
    vcmla.f32   q7, q0, q1, #0   @ 2nd part of the complex multiplication,
                                 @ accumulation, can be overlapped with vldrw
    vldrw.u32   q1, [r1], #16    @ 2nd complex vector load,
                                 @ can be overlapped with vcmla
    vcmla.f32   q7, q2, q1, #0   @ 1st part of the complex multiplication,
                                 @ accumulation, can be overlapped with vldrw
    vldrw.u32   q0, [r0], #16    @ 1st complex vector load,
                                 @ can be overlapped with vcmla
    vcmla.f32   q7, q2, q1, #90  @ 2nd part of the complex multiplication,
                                 @ accumulation, can be overlapped with vldrw
    vldrw.u32   q1, [r1], #16    @ 2nd complex vector load,
                                 @ can be overlapped with vcmla
    le          lr, 2b           @ low overhead while loop end
1:

    @ tail handling eluded

    @ partial parts summation
    vadd.f32    s0, s28, s30     @ real part
    vadd.f32    s1, s29, s31     @ imaginary part
```

Fig. 7. Codes optimized to interleave instructions of different group, allowing overlapping

5. Looping Optimizations

To get the most out of the data processing hardware resources, we also need to enable efficiency in loop operations as signal processing often operates on an array of data. Traditionally, many DSPs have Zero-Overhead-Loop supported by adding dedicated loop counter registers and related pipeline enhancements. While the loop operations can be extremely efficient in these DSPs, the presence of additional loop counter registers adds complexity to software in the cases where the Zero-Overhead-Loop feature can be utilized by multiple software contexts – e.g. between interrupt handlers and interrupted codes, and between multiple applications when using this feature in a system with an RTOS.

In the Armv8.1-M architecture, the low-overhead-branch (LOB) extension is added to the instruction set. This allows branch penalties to be minimized in loop operations, without the hardware cost of a branch predictor. Some forms of these instructions are available even without Helium, so applications running on Cortex-M55 devices without Helium implemented can still take advantage of some of the new capabilities in the LOB extension.

The most basic form of a low-overhead loop in Armv8.1-M contains a WLS (While-Loop-Start) and a LE (Loop-End) instruction:

```
    MOVS   R0, #10 ; loop 10 times
    WLS    LR, R0, loopend_1 ; while loop start
loopstart_1
    LDR    R1, [R2], #4 ; Memory copy
    STR    R1, [R3], #4 ; Memory copy
    LE     LR, loopstart_1     ; loop end
loopend_1
    B      label_x ; branch to somewhere else
```

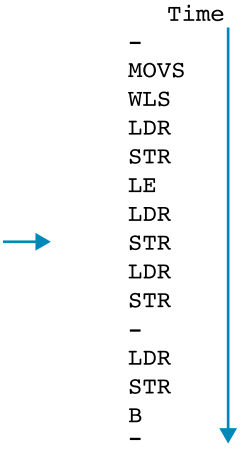


Fig. 8. Simple example of LOB instruction usage – memory copy

Before initial loop execution, the WLS instruction checks the loop counter in R0 and branch to the end of the loop if 0. If not, it starts the loop body and executes to LE marking the end of the loop. The LE instruction then caches the loop information and executes the loop body again. In the second round of iteration, since the loop information is cached internally to the processor, the WLS and LE instructions will not be executed until the loop operation is completed.

If an interrupt occurred during the low-overhead loop execution, the loop information cache would be cleared and when the loop is resumed after the interrupt service, the LE instruction needs to be executed again and restore the loop information cache again.

To avoid the implication on context handling and to make the Cortex-M55 processor easy to use, the LOB extension reuses R14 (Link Register) as loop counter, which means in the events that an interrupt or OS context switch occurred in the middle of a low-overhead loop, the loop counter would be saved automatically and will be restored automatically when the loop is resumed. This also means existing C/C++ codes don't need to be changed to take advantage of the LOB extension, enabling better software portability.

There are other variants of low-overhead loops. For example, the DLS (Do-Loop-Start) instruction is similar to WLS, but without the checking of loop-counter in the first loop iteration (i.e. the first iteration always executes).

Another variant of low-overhead loop is tail predication: This includes WLSTP and DLSTP, which need to be used with LETP – Loop-end with tail predication. In some application codes, the number of elements to be processed is not necessary multiple of the number of elements in a vector. To help improve the handling of this data, the tail predication uses the number of elements to be processed as for loop control, and the loop counter LR decrement by the number of elements in a vector instead. At the last iteration of tail predication loops, the processing of elements inside the loop is conditionally executed.

To enable tail predication, a few extra registers are introduced:

- Vector Predication Status and Control Register (VPR) – contains the predication mask, which is updated at the end of each iteration in a low-overhead loop with tail predication
- The Floating-Point Status and Control Register (FPSCR) introduced a new bit field LTPSIZE to indicate the size of vector element. This is setup when executing WLSTP or DLSTP

At the last iteration, the state of VPR contains the required lane mask (generated during decrement of LR, and dependent on FPSCR.LTPSIZE) that allows the vector operations to be conditionally carried out.

For example, the following code sequence is a vector scaling routing, with

- r4 – the number of elements to process
- r0 – address of input vector
- r3 – address of output vector

```
    @ Q.15 vector scaling routine

    vdup.16      q1, r1       @ fractional part duplication
    wlstp.16     lr, r4, lf   @ low overhead while loop start with tail predication
2b:
    vldrh.u16    q0, [r0], #16    @ input vector load
    vmulh.s16    q0, q0, q1       @ vector multiplication, returns high part
    vqshl.s16    q0, r2           @ vector scale
    vstrh.16     q0, [r3], #16    @ vector store
    letp         lr, 2b           @ low overhead while loop end
1f:
```

Fig. 9. Simple code for vector scaling

Now let's assume we are processing 15 elements (r4 = 15). In the first iteration, all elements in the vector are processed:

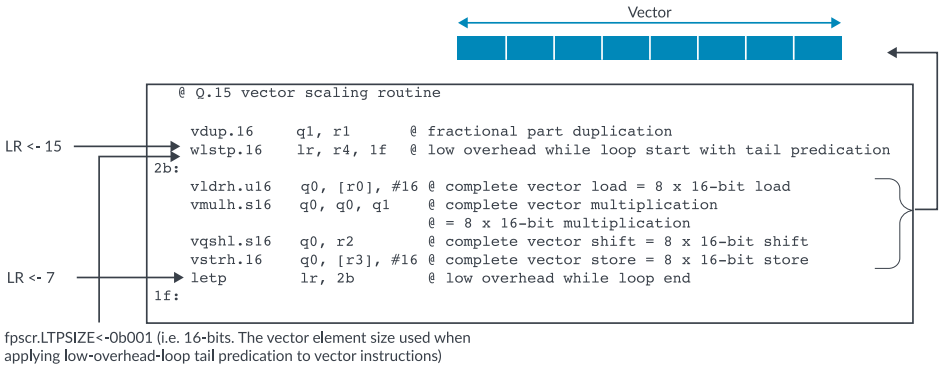


Fig. 10. First iteration of simple vector scaling with tail predication. All eight elements are processed

In the second iteration, the last element in the vector is not processed:

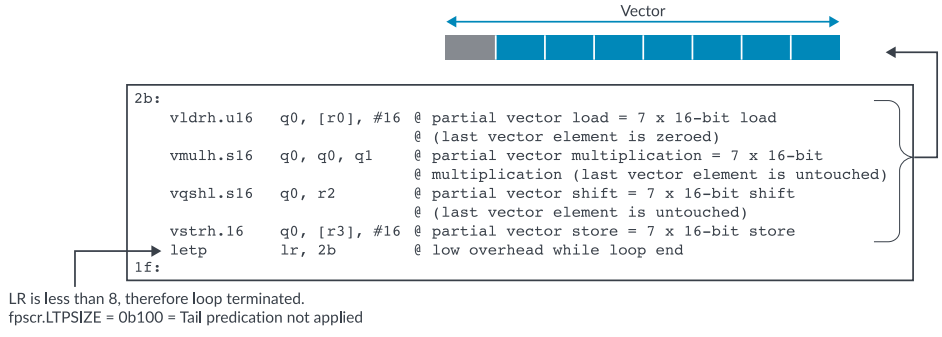


Fig. 11. Second iteration of simple vector scaling with tail predication, only seven elements are processed

To allow tail predication to be used by multiple software context:

- Exception stack frame is updated so that VPR is part of the extended exception stack frame (this uses a reserved 32-bit space in the Armv8.0-M stack frame so that there is no impact to existing software when moving to Armv8.1-M)
- LTPSIZE is part of FPSCR, which is also part of the extended exception stack frame

As such, interrupt handlers and multiple application threads in an OS environment can all benefit from the tail predication, without the need for additional software support overhead.

Using low-overhead-branch extension, an Armv8.1-M processor can get the same looping performance as in a DSP with Zero-overhead-loop support.

6. Data Type Supports

To enable the Cortex-M55 processor to be used in a wide range of signal processing applications, Armv8.1-M introduces support for a wide range of vector data types:

- Vector 8-bit integer/fixed-point
- Vector 16-bit integer/fixed-point
- Vector 32-bit integer/fixed-point
- Vector 16-bit half-precision floating-point
- Vector 32-bit single-precision floating-point

Additional instructions have been added to deal with complex data (with real and imaginary parts), which is common in signal processing. Complex vectors data are organized with interleaved real and imaginary parts. Helium provides native complex support for integer and floating-point date enables efficient complex data processing.

The support of half-precision floating-point is new in Cortex-M processors. This can be useful in a range of sensing applications where the input data precision constraints can be relaxed, but can have a high dynamic range. One such usage is for voice and sound sensing – in many cases the audio frontend components do not need full single-precision floating-point support and can be implemented in half-precision without compromising algorithm performances. Half-precision floating-point format support might also help reduce memory size and execution cycle count required in some applications.

In addition to data types supported in common DSPs, Armv8.1-M also supports 8-bit data type which is widely used in ML applications. Armv8.1-M supports 8-bit vector dot product, which is the central piece for neural network computation. As a result, we see a significant performance uplift in ML performance compared to the previous generation of Cortex-M processors. This feature is not available in most DSPs.

7. Memory Access Instructions

To help with signal processing, Armv8.1-M also supports various types of interleaving memory accesses. This requirement is common for dealing with audio data (e.g. left and right stereo channels) and image data (e.g. RGBA, CMYK). Helium in Armv8.1-M supports instructions for dealing with data arrangements of stride-2 and stride-4, which enables higher performance in those use cases.

In many DSPs, dedicated hardware support is available for circular addressing and bit reverse address. In some cases:

- special hardware registers are introduced in these DSPs for controlling such address generation
- special instructions are introduced to be used in conjunction with these special addressing modes

Such hardware requirements in DSP can lead to the same software challenges as the hardware registers for zero-overhead-loops, where the OS must include additional context saving/restore to enable multiple contexts of software to use these features.

To avoid these drawbacks, Helium supports scatter-gather memory access instructions and a range of supporting instructions to handle those operations. In short, scatter-store and gather-load instructions allow the software to transfer multiple data between vector registers and memory using a vector of addresses or a base address with a vector address offset.

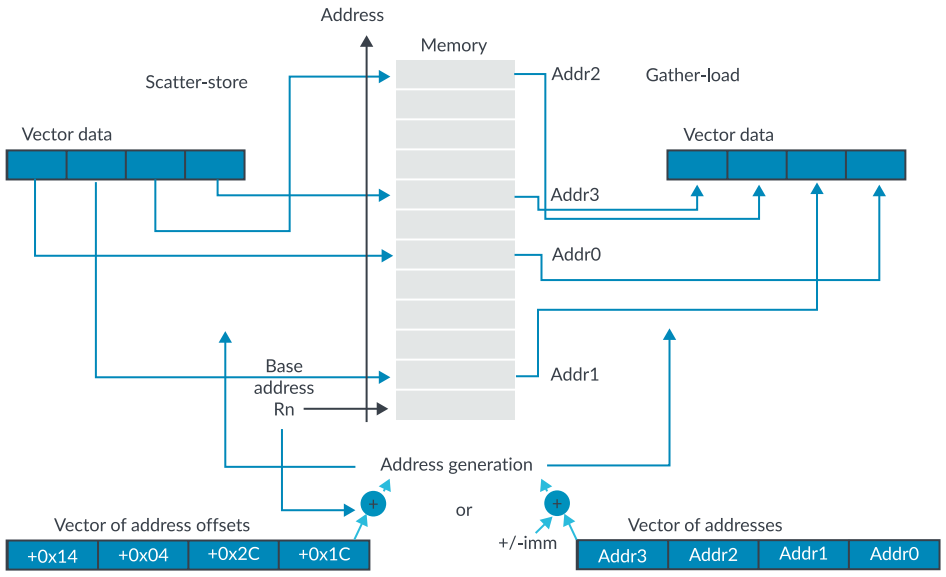


Fig. 12. Concept of scatter-store and gather-load instructions

Gather-load and scatter-store are available in two different forms:

- Vector of offsets: VLDR {B, H, W, D}.<dt> Qd, [Rn, Qm]
- Vector of addresses: VLDR {B, H, W, D}.<dt> Qd, [Qm. {+/-<imm>}]

Scatter-gather instructions can be used in many different ways. Helium included several instructions to help to generate a vector of address offsets:

Instruction	Usage
VBRSR	For bit-reversed addressing. This is useful for FFT (Fast Fourier Transform)
VIWDUP	For circular address offset generation (increment)
VDWDUP	For circular address offset generation (decrement)
VIDUP	Like VIWDUP but without wrapping (increment)
VDDUP	Like VDWDUP but without wrapping (decrement)

For example, the VIWDUP instruction is designed to generate vector of address offsets for circular buffer:

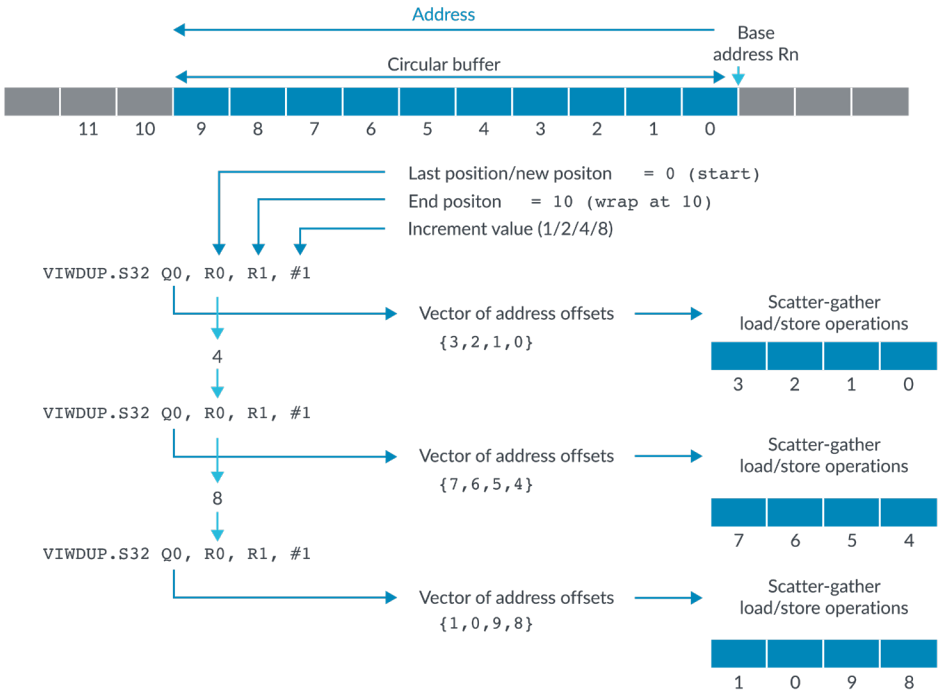


Fig. 13. Implementing a circular buffer in Helium

Different increment values can be used to provide support of various data type sizes.

8. Memory System Design

Designing the instructions for optimal memory accesses for signal processing and ML is only one part of the story. In order to support the higher data processing capability, the processor's memory interfaces need to be designed to cope with high throughput, and potentially larger memory sizes are needed for some of the signal processing and ML applications. At the same time, many Cortex-M based systems require real-time response capability. To address this requirement, the memory system support on the Cortex-M55 processor is designed as two halves:

- A closely coupled memory system with I-TCM (Tightly Coupled Memory), D-TCM and an AHB peripheral bus – supporting real-time response at high performance
- A 64-bit AXI interface with optional I-cache and D-cache – supporting memories with higher latency and supporting multiple outstanding transfers to maximize memory bandwidth

A block diagram of the memory system is shown in Figure 14.

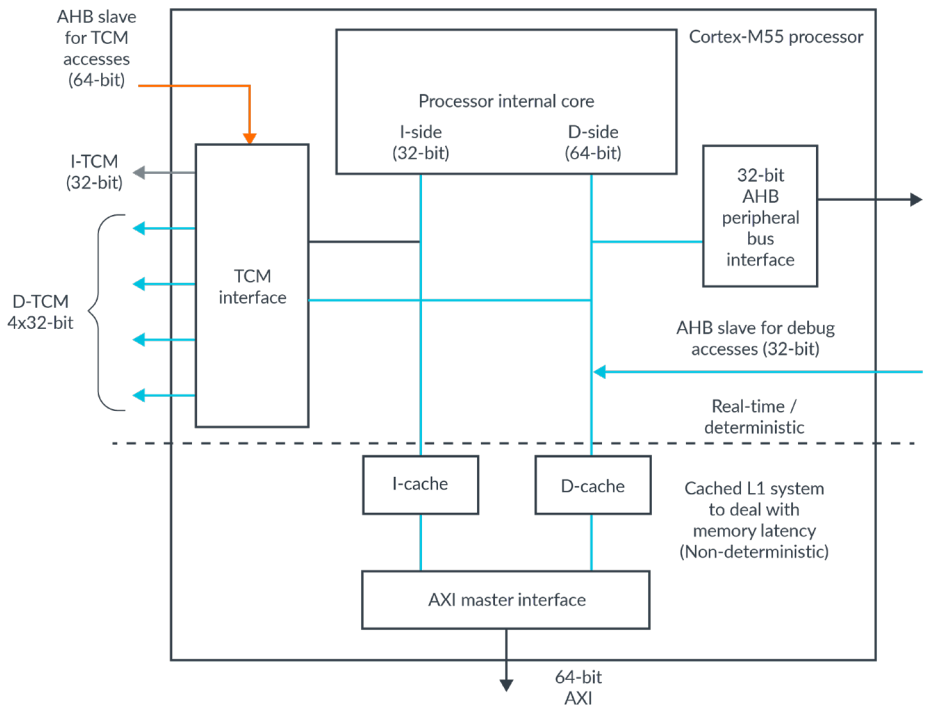


Fig. 14. Cortex-M55 memory system design

With this arrangement, the Cortex-M55 processor can serve interrupt requests at a very low latency, given that the vector table, program code and data memory spaces needed by the interrupt service routines (ISR) are placed in the TCMs. The peripheral AHB allows

legacy AHB/APB peripherals to be connected to the processor and be accessed without latency impact from the AXI interconnect at the system level.

While the Cortex-M55 processor is not super scalar, with gather-load and scatter-store, it is possible for the processor to access two independent data addresses at the same time. As a result, the D-TCM interface is designed as separated banks so that two data accesses targeting different D-TCM banks can be carried out at the same time. It can also support 64-bit accesses, and can also enable faster stacking and unstacking operations in interrupt handling.

Meanwhile, memory blocks with higher access latency can be connected via the 64-bit AMBA AXI master interface. With the optional I-cache and D-cache, most of the accesses to slow memories, such as off-chip DDR and embedded flash memories, can be handled by the caches, and the slow memories are accessed only when there is a cache miss. In this way, the performance of the system is less likely to be penalized by the use of slower memories.

In order to fully utilize the processing capability, we don't want the processor to spend time transferring data between TCMs and the main memory system. As a result, a 64-bit AHB slave port is added to allow other bus masters, such as a DMA controller, to access to the TCMs. In this way, while the processor is processing a block of data, the processing result for the previous block can be read out from D-TCM and the next block of input data can be written into another location in D-TCM at the same time.

Since the processor can also access up to 64-bits of data at the same time, the D-TCM is designed to have four 32-bit D-TCM interfaces, which are separated by bit 2 and bit 3 of the data address. In this way, the total D-TCM bandwidth is increased to 128-bit per cycle to allow good performance for both the processor and software execution as well as DMA transfers. If both the processor and DMA controller access the same memory bank on the D-TCM interface, the processor has higher priority, and it is likely that in the next cycle the processor will move on to access data in another D-TCM bank, so the DMA access can be carried out with very little delay.

In some of the DSPs, we see dedicated memory interface ports for DSP data operations. When combining VLIW with such memory features, it is possible to access these memories in parallel with data processing instructions to get a higher performance. In the development of the Helium architecture, such a feature is not considered because it:

- Requires special memory access instructions – which means that the data cannot be directly accessed using standard C/C++ pointers, making software less portable. While special C intrinsic can be introduced, the software is not portable
- Might result in additional overhead in order to transfer data between such memory and the main memory

- Does not work with the current TrustZone architecture. TrustZone for Armv8-M implements a range of architectural features (e.g. SAU, IDAU and TT instructions) related to the 4GB address range to handle TrustZone security management. However, if a separated address space is added, additional security features are needed to support security inside those extra memories and would increase system complexity, power and cost

While the Cortex-M55 processor does not have dedicated memory interface ports for DSP data, with the Cortex-M55 pipeline design and the D-TCM arrangement, we can already handle two data accesses and data processing at the same time. The D-TCM is also a part of the standard system address space, so all data within TCMs can be accessed using standard C/C++ codes, making it easier to use. In order to provide TrustZone security support for the TCMs, TrustZone access filters are included on the TCM interfaces to manage accesses generated by the processor and the DMA controller. As the TCMs are part of the standard address space, standard TrustZone security handling (e.g. security attribution checking) can work for TCMs in exactly the same ways as the main memories connected via AXI.

9. Performance of the Cortex-M55 Processor

With Helium technology, the innovative pipeline design and the memory system features, the performance of Cortex-M55 processor in signal processing and machine learning applications is significantly better than the previous generations of Cortex-M processors.

For example, the performance of low-level DSP functions like FFT, filters (based on hand-optimized CMSIS-DSP), the performance of the Cortex-M55 processor is over 4 times better than Cortex-M4 on average.

Average performance per datatype for selected CMSIS-DSP kernels vs the Cortex-M4 processor

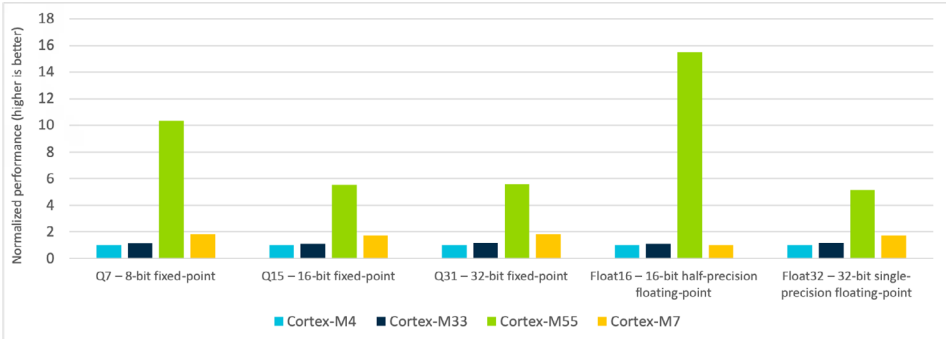


Fig. 15. Cortex-M55 low-level signal processing performance

We also see great improvements at higher-level audio applications. For example, Arm together with Dolby, have investigated the complexity of the Dolby Audio Processing (DAP) codec running on Cortex-M55 with Helium.

From the analysis results, we see that Cortex-M55 can provide over 60% reduction in execution time when compared to the Cortex-M4 processor.

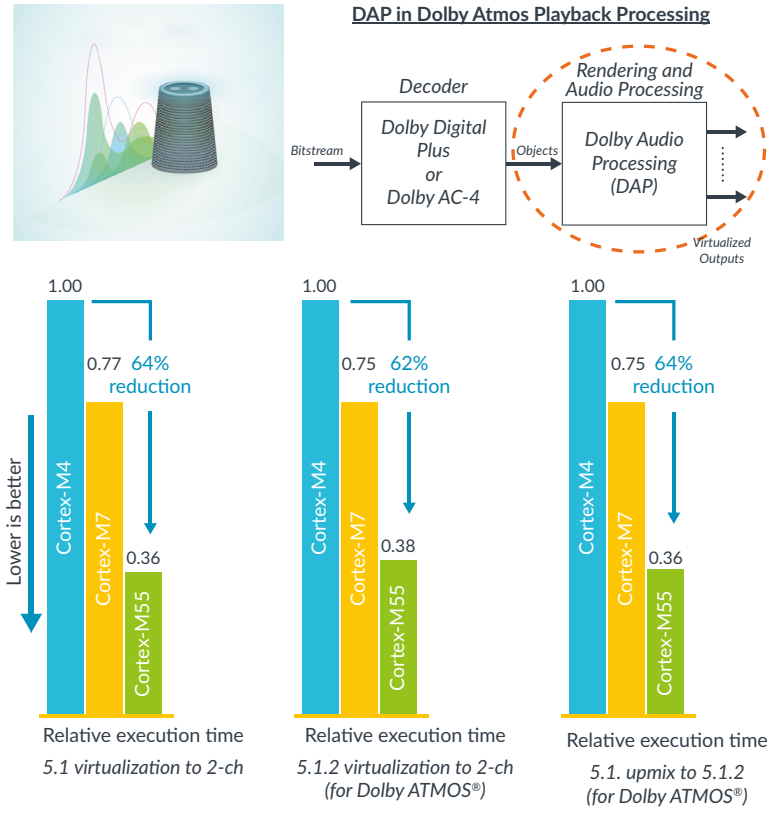


Fig. 16. Cortex-M55 performance benefits in Dolby Audio Processing (DAP)

Another area of interest is the ML processing performance. Several machine learning algorithms have been ported to Helium, including a keyword spotting library and a CiFAR10 image classification library. From this result, we see that the Cortex-M55 processor gives nearly ten times better performance than the Cortex-M4 processor in keyword spotting, and almost six times better than the Cortex-M4 processor in CiFAR10 image classification operations.

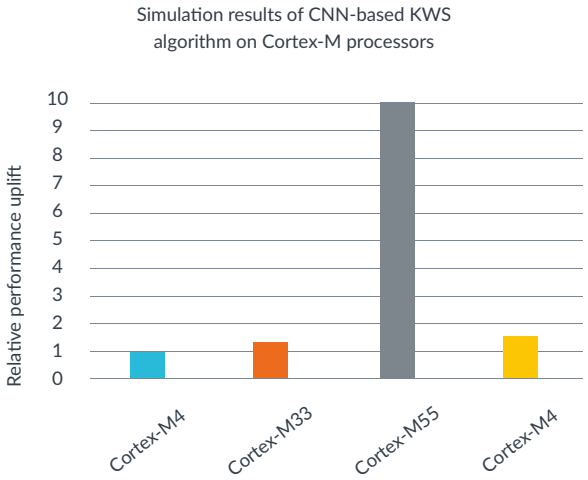
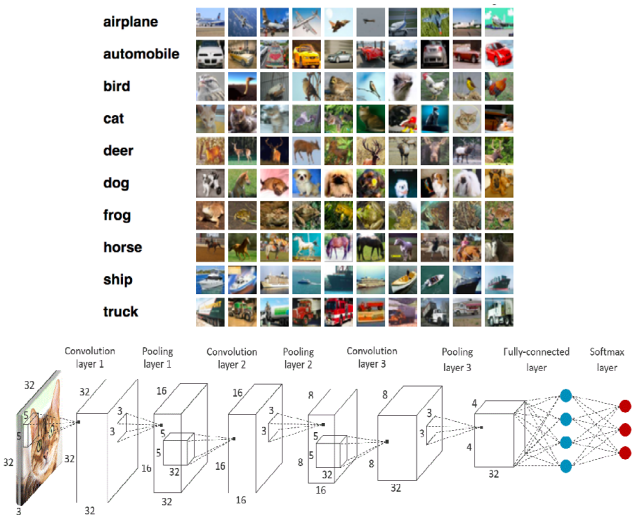
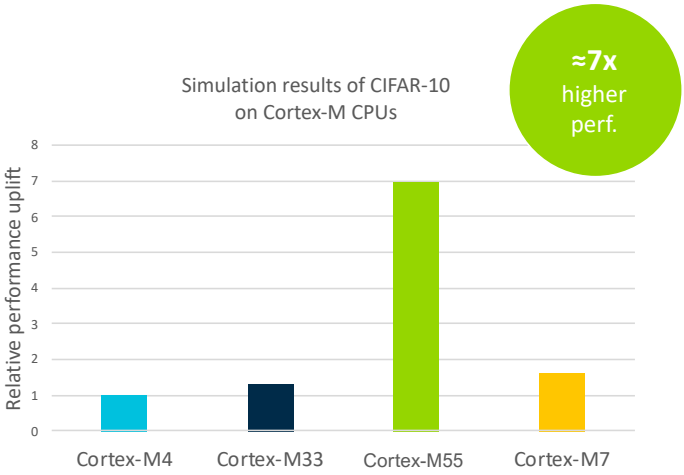


Fig. 17. Cortex-M55 processor performance uplift in keyword spotting (KWS)

3 convolution layer, 3 pooling layer and 1 fully-connected layer



Efficient compute capabilities in next gen Cortex-M



Cortex-M55 performance results are based on RTL and C compiler in development. Subject to change.

Cortex-M4/M7/M33 using AC6.10

Fig. 18. Cortex-M55 processor performance uplift in image classification (CiFAR10)

10. Considerations When Developing Applications with the Cortex-M55 Processor

While the Cortex-M55 processor design enables significant performance uplift in signal processing and machine learning applications, not every application can gain the same level of performance boost. Since Helium technology is based on SIMD operations, it works very well when the data processing can be vectorized. However, there is a range of application codes that cannot be vectorized. The traditional VLIW approach, however, allows different operations to be scheduled at different execution slots. This potentially allows some very sequential code parts handling to be carried out quicker (e.g. variable length encoding/decoding in audio codecs). For Arm processors, it is also possible to achieve similar parallelism by introducing superscalar in the design. The Cortex-M55 processor, however, is not a superscalar processor, and therefore, does not have this capability. Nevertheless, with limited dual-issue capability in the Cortex-M55 processor and various new features in Armv8.1-M architecture (e.g. low-overhead loops, new conditional execution instructions, 64-bit shifts), scalar performance has been improved in various areas.

With VLIW architecture, it is possible to gain higher performance by increasing the width of the pipeline. For example, high-end DSPs have four or five parallel execution slots to enable high performance. This, however, also means a much larger silicon area, power and instruction memory bandwidth. For instance, some DSPs can consume up to 128-bit of instructions per cycle, while the Cortex-M55 processor can only execute 32-bit of instructions per cycle.

In order to get a higher efficiency, the datapath of the Cortex-M55 processor is designed to be 64-bit, even though the Helium vector is 128-bit. By interleaving different instruction groups, we can avoid hardware resource conflicts and achieve the same level of performance as with a 128-bit wide datapath. However, there are occurrences where we cannot avoid pipeline conflicts. For example, if two vector memory access instructions are next to each other, then the pipeline cannot allow overlapping of instruction execution and results in pipeline bubbles.

While the D-TCM allows the processor to access two sets of data at the same time, due to the nature of the memory bank partitioning, two data accesses to the same memory bank cannot be carried out at the same time. As a result, when defining data structures or creating algorithms that use interleaved data accesses, software developers need to be careful about the data memory layout and access sequences to avoid access conflicts.

The Cortex-M55 processor supports 8 vector registers. While this could appear as a limitation, our internal studies over a broad range of critical DSP and ML routines proved

that the amount of vector registers does not compromise targeted performance. For cases where more vector registers are needed, this limitation can be mitigated by judicious spilling where vector load/store can be overlapped with arithmetic operations with no or small penalty. Some of DSP architectures support 16 or even 32 vector registers, which makes instruction scheduling much easier, at a cost of larger silicon area and power. Since the Cortex-M55 processor is designed to target low-power embedded systems, this is an essential trade-off area.

To help software developers to utilize the capabilities of the Cortex-M55 processor, the CMSIS-DSP and CMSIS-NN libraries are being updated and optimized to support new features in Armv8.1-M architecture. With these libraries, existing applications utilizing CMSIS-DSP and CMSIS-NN can immediately take advantage of the Helium feature by switching over to the latest libraries. Additional optimization features are also being introduced by various C compilers. For example, auto-vectorization support for Helium is available in latest Arm Compiler and Arm toolchains.

11. Summary

In summary, we see that with Helium technology and the Cortex-M55 processor design, it is possible to create a highly capable signal processing and machine learning engine on top of a traditional general-purpose embedded processor architecture. Various features are introduced in Helium and the Cortex-M55 processor to match a range of traditional DSP features:

DSP features	Helium
Zero overhead loops	Low-overhead-branch extensions
Complex data processing	Complex data processing
Circular buffer	Scatter-gather memory access with instruction for bit-reverse address generation
Bit reverse addressing	Scatter-gather memory access with instruction for bit-reverse address generation
Dedicated DSP data memory interface	Multiple TCM interfaces to support vector memory accesses and pipeline optimization
Interleave data accesses	Interleave data acccesses

Table 1. Comparison of features on DSP to Helium and Cortex-M55 processor features

With all these features and innovative design techniques, the Cortex-M55 processor can match the performance of mid-range dual-MAC DSP (i.e. products that process ~64-bit of MAC per cycle) in a range of signal processing workloads. The Cortex-M55 processor is also designed for machine learning applications, whereas support for machine learning data types is rare in the mid-range and low-end DSP products. As a result, the Cortex-M55 processor can outperform a range of DSPs in neural network processing.

There are, however, a range of design considerations when creating a processor design based on Helium. In addition to the complexity of the overlapping pipeline, the memory system design also needs to be optimized to enable the signal processing and machine learning workloads. To make the most of Helium, software developers also need to be aware of some of the limitations of the architecture during software optimization.

Overall, Helium and the Cortex-M55 processor design are a good balance of performance and energy efficiency, and at the same time satisfies the requirements from embedded applications including real-time responsiveness, security and ease-of-use.

Click here for more information about the Cortex-M55 processor and to explore further resources.

Acknowledgement

I would like to express my gratitude to Dolby for the collaboration and sharing of various data that helped us which help us to investigate Cortex-M55 processor performance, Fabien Klein in Arm embedded software team for his help in preparing various data and the review, Arm research team for creating the Helium extension, and Cortex-M55 processor design team for make Helium technology possible.