



Learn the architecture - Introducing SVE2

Version 1.0

Non-Confidential

Copyright © 2020–2021, 2023 Arm Limited (or its affiliates).
All rights reserved.

Issue 04

102340_0100_04_en



Learn the architecture - Introducing SVE2

Copyright © 2020–2021, 2023 Arm Limited (or its affiliates). All rights reserved.

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	11 December 2020	Non-Confidential	First release
0100-02	22 January 2021	Non-Confidential	Documentation update 1 for Introduction to SVE2
0100-03	17 May 2021	Non-Confidential	Documentation update 2 for Introduction to SVE2
0100-04	3 May 2023	Non-Confidential	Documentation update 3 for Introduction to SVE2

Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND

REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

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

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

Copyright © 2020–2021, 2023 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

Contents

- 1. Overview..... 6
- 2. Introducing SVE2..... 7
- 3. SVE2 architecture fundamentals.....9
- 4. New features in SVE2..... 13
- 5. Program with SVE2.....15
- 6. Check your knowledge..... 18
- 7. Related information..... 19

1. Overview

This guide is a short introduction to version two of the Scalable Vector Extension (SVE2) for the Armv9-A architecture. In this guide, you can learn about the concept and main features of SVE2, the application domains of SVE2, and how SVE2 compares to SVE and to Neon. We also describe how to develop a program for an SVE2-enabled target.

Before you begin

This article assumes you are already familiar with the following concepts:

- Single Instruction Multi Data (SIMD)
- Neon
- Scalable Vector Extension (SVE)

If you are not familiar with these concepts, read:

- [Introducing Neon for Armv8-A](#)
- [What is the Scalable Vector Extension \(SVE\)](#)
- [SVE and Neon coding compared](#)

2. Introducing SVE2

This section introduces the Scalable Vector Extension version two (SVE2) of the Arm AArch64 architecture.

Following the development of the Neon architecture extension, which has a fixed 128-bit vector length for the instruction set, Arm designed the Scalable Vector Extension (SVE). SVE is a new Single Instruction Multiple Data (SIMD) instruction set that is used as an extension to AArch64, to allow for flexible vector length implementations. SVE improves the suitability of the architecture for High Performance Computing (HPC) applications, which require very large quantities of data processing.

SVE2 is a superset of SVE and Neon. SVE2 allows for more function domains in data-level parallelism. SVE2 inherits the concept, vector registers, and operation principles of SVE. SVE and SVE2 define 32 scalable vector registers. Silicon partners can choose a suitable vector length design implementation for hardware that varies between 128 bits and 2048 bits, at 128-bit increments. The advantage of SVE and SVE2 is that only one vector instruction set uses the scalable variables.

The SVE design concept enables developers to write and build software once, then run the same binaries on different AArch64 hardware with various SVE vector length implementations. The portability of the binaries means that developers do not have to know the vector length implementation for their system. Removing the requirement to rebuild binaries allows software to be ported more easily. In addition to the scalable vectors, SVE and SVE2 include:

- Per-lane predication
- Gather-load and scatter-store
- Speculative vectorization

These features help vectorize and optimize loops when you process large datasets.

The main difference between SVE2 and SVE is the functional coverage of the instruction set. SVE was designed for HPC and ML applications. SVE2 extends the SVE instruction set to enable data-processing domains beyond HPC and ML. The SVE2 instruction set can also accelerate the common algorithms that are used in the following applications:

- Computer vision
- Multimedia
- Long-Term Evolution (LTE) baseband processing
- Genomics
- In-memory database
- Web serving
- General-purpose software

To help compilers vectorize more effectively for these domains, SVE2 adds a vector-width-agnostic version of the Neon instructions in most of the integer Digital Signal Processing (DSP) and media processing functionality.

SVE and SVE2 both enable the collection and processing of a large amount of data.

SVE and SVE2 are not an extension of the Neon instruction set. Instead, SVE and SVE2 are redesigned for better data parallelism than Neon provides. However, the hardware logic of SVE and SVE2 overlays the Neon hardware implementation. When a microarchitecture supports SVE or SVE2, it also supports Neon. To use SVE and SVE2, software that runs on that microarchitecture must first use Neon.

3. SVE2 architecture fundamentals

This section introduces the basic architecture features that SVE and SVE2 share.

Like SVE, SVE2 is based on the scalable vectors. In addition to the existing register banks that Neon provides, SVE and SVE2 adds the following registers:

- 32 scalable vector registers, `z0–z31`
- 16 scalable predicate registers, `p0–p15`
 - One First Fault predicate Register (FFR)
- Scalable vector system control registers `zcr_elx`

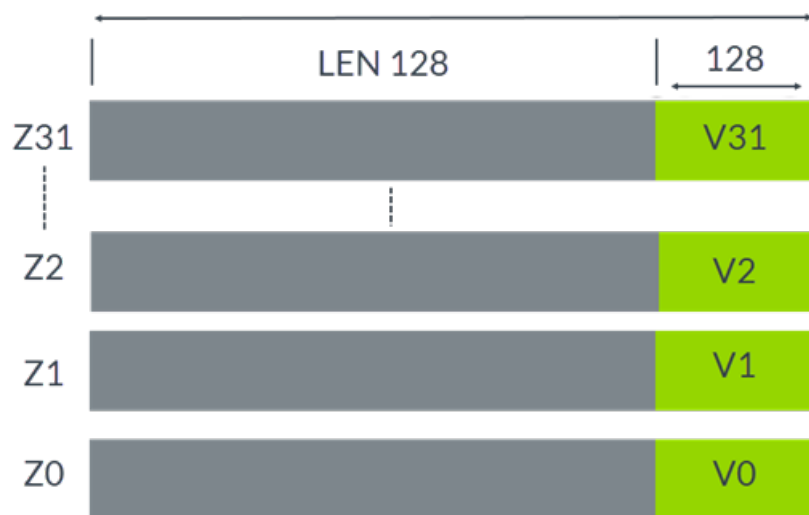
Let's look at each of these in turn.

Scalable vector registers `z0–z31`

Ears of Neon.

The figure below shows the scalable vector registers `z0–z31`:

Figure 3-1: Scalable vector registers `z0–z31`

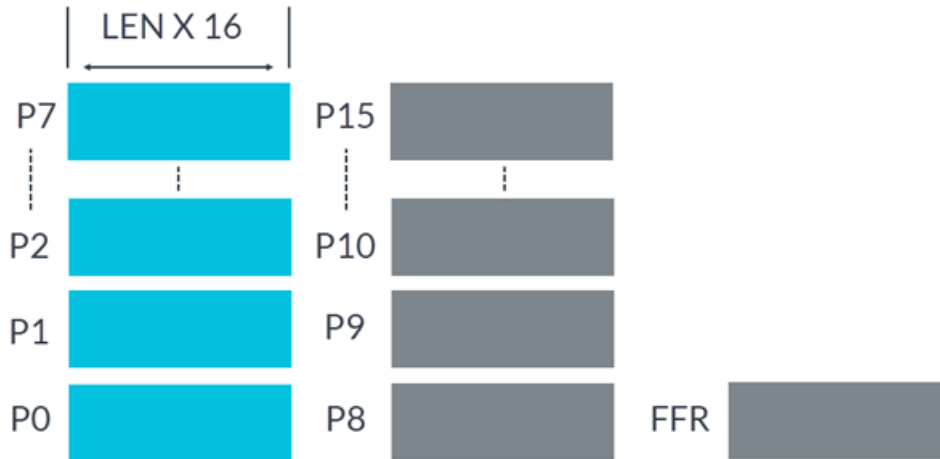


The scalable vectors can:

- Hold 64, 32, 16, and 8-bit elements
- Support integer, double-precision, single-precision, and half-precision floating-point elements
- Be configured with the vector length in each Exception level (EL)

Scalable predicate registers `p0–p15`

The figure below shows the scalable predicate registers `p0–p15`:

Figure 3-2: Scalable predicate registers P0–P15

The predicate registers are usually used as bit masks for data operations, where:

- Each predicate register is 1/8 of the z_x length.
- P0–P7 are governing predicates for load, store, and arithmetic.
- P8–P15 are extra predicates for loop management.
- First Fault Register (FFR) is for Speculative memory accesses.

If the predicate registers are not used as bit masks, they are used as operands.

Scalable vector system control registers ZCR_Elx

The figure below shows the scalable vector system control registers zcr_elx :

Figure 3-3: Scalable vector system control registers ZCR_E1x

The scalable vector system control registers indicate the SVE implementation features:

- The $zcr_elx.LEN$ field is for the vector length of the current and lower exception levels
- Most bits are currently reserved for future use.

SVE2 assembly syntax

SVE2 follows the same assembly syntax format that SVE follows. The following instruction examples show this format.

Example 1:

```
LDFF1D {<Zt>.D}, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]
```

Where:

- z_{t7} are the vectors, z_0 – z_{31}
- D , vector and predicate registers have known element type but unknown element numbers
- P_g are the predicates, P_0 – P_{15}
- z is the zeroing predication
- z_m is gather-scatter or vector addressing

Example 2:

```
ADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>
```

Where:

- M is the merging predication

Example 3

```
ORRS <Pd>.B, <Pg>.Z, <Pn>.B, <Pm>.B
```

Where:

- s is a new interpretation of predicate condition flags $NZCV$
- P_g , a predicate, is a “bit mask”.

Key SVE architecture features that SVE2 inherits:

SVE2 architecture features

SVE2 inherits the following important SVE architecture features:

- Gather-load and scatter-store

The flexible address mode in SVE allows vector base address or vector offset, which enables loading to a single vector register from non-contiguous memory locations. For example:

```
LD1SB Z0.S, P0/Z, [Z1.S, #4] // Gather load of signed bytes to active 32-bit
                             elements of Z0 from memory addresses generated by 32-bit vector base Z1 plus
                             immediate index #4.
LD1SB Z0.D, P0/Z, [X0, Z1.D] // Gather load of signed bytes to active elements of
                             Z0 from memory addresses generated by a 64-bit scalar base X0 plus vector index in
                             Z1.D.
```

- Per-lane predication

To allow flexible operations on selected elements, SVE and SVE2 introduce 16 governing predicate registers, `P0-P15`, to indicate the valid operation on active lanes of the vectors. For example:

```
ADD Z0.D, P0/M, Z1.D, Z2.D // Add the active elements Z1 and Z2 and put the
                           result in Z0. P0 indicates which elements of the
                           operands are active and inactive. 'M' after P0
                           indicates that the inactive element will be merged,
                           meaning Z0 inactive element will remain its original
                           value before the ADD operation. If it was 'Z' after
                           P0, then it would mean that inactive element will
                           be zeroed in the destination vector register.
```

- Predicate-driven loop control and management

Predicate-driven loop control and management is an efficient loop control feature. This feature allows loop heads and tails overhead, caused by the processing of partial vectors, to be removed by registering the active and inactive elements index in the predicate registers. This means that, in the next loop, only the active elements do the expected options. For example:

```
WHILELO P0.S, x8, x9 // Generate a predicate in P0 that starting from the lowest
                     numbered element is true while the incrementing value of the
                     first, unsigned scalar X8 operand is lower than the second
                     scalar operand X9 and false thereafter, up to the highest
                     numbered element.
```

- Vector partitioning for software-managed speculation

SVE improved the Neon vectorization restrictions on Speculative load. SVE introduces the first-fault vector load instructions, for example `LDRFF`, and the First-Fault predicate Registers (FFRs) to allow vector accesses to cross into invalid pages. For example:

```
LDRFF1D Z0.D, P0/Z, [Z1.D, #0] // Gather load with first-faulting behaviour of
                                doublewords to active elements of Z0 from memory
                                addresses generated by the vector base Z1 plus 0.
                                Inactive elements will not read Device memory or
                                signal faults and are set to zero in the destination
                                vector. Successful load to the valid memory will
                                set true to the first-fault register (FFR), and the
                                first-faulting load will set false to the
                                according element and the rest elements in FFR.

RDRFFR P0.B // Read the first-fault register (FFR) and place in the destination
             predicate without predication.
```

- Extended floating-point and bitwise horizontal reductions

SVE enhances floating-point and bitwise horizontal reduction operations. Examples of these operations include in-order or tree-based floating-point sum. These operations trade off repeatability and performance. Here is some example code:

```
FADDP Z0.S, P0/M, Z1.S, Z2.S // Add pairs of adjacent floating-point elements
                              within each source vector Z1 and Z2, and interleave
                              the results from corresponding lanes. The interleaved
                              result values are destructively placed in the first
                              source vector Z0.
```

4. New features in SVE2

This section introduces the new features that SVE2 adds to the Arm AArch64 architecture.

To achieve scalable performance, SVE2 builds on the foundations of SVE, allowing vector implementation up to 2048 bits.

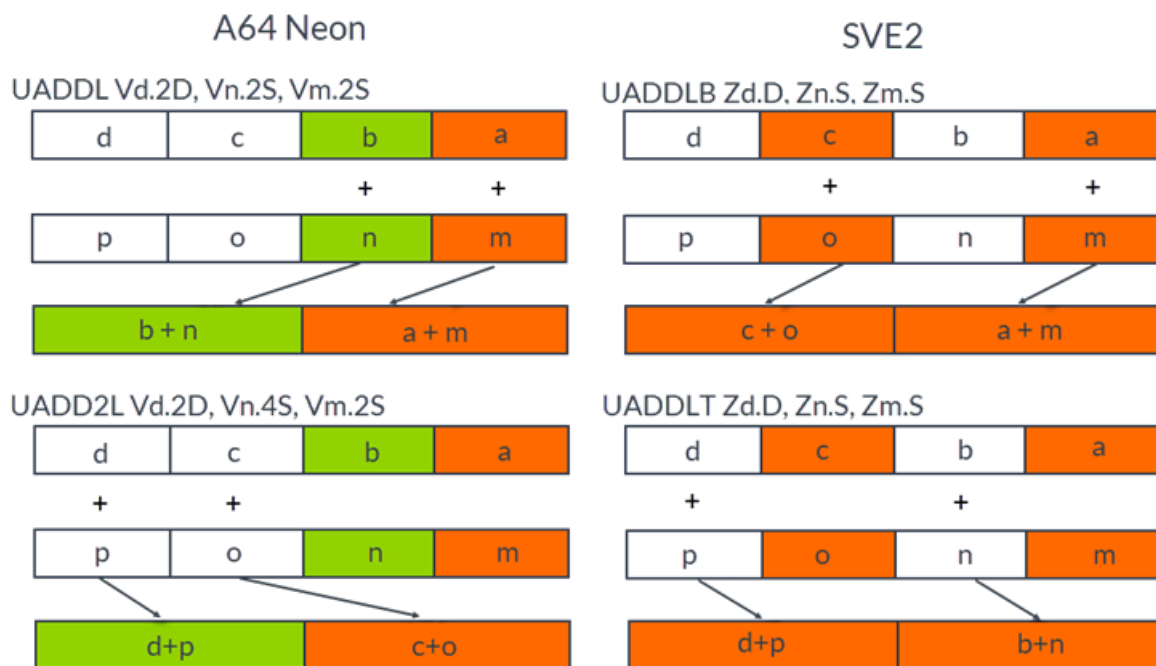
In SVE2, many instructions are added that replicate existing instructions in Neon, including:

- Transformed Neon integer operations, for example, Signed absolute difference and accumulate (`SAB`) and Signed halving addition (`SHADD`).
- Transformed Neon widen, narrow, and pairwise operations, for example, Unsigned add long – bottom (`UADDLb`) and Unsigned add long – top (`UADDLT`).

There are changes in the element processing orders. SVE2 processes on interleaving even and odd elements, and Neon processed on low and high half elements for narrow or wider operations.

The following diagram illustrates the difference between the Neon and SVE2 processes:

Figure 4-1: the difference between the Neon and SVE2 processes



- Complex arithmetic, for example complex integer multiply-add with rotate (`CMLA`).
- Multi-precision arithmetic for large integer arithmetic and cryptography, for example, Add with carry long – bottom (`ADC1b`), Add carry long – top (`ADC1t`), and SM4 encryption and decryption (`SM4E`).

For backwards compatibility, Neon and VFP are required in the latest architectures. Although SVE2 includes some of the functions of SVE and Neon, SVE2 does not exclude the Neon presence on the chip.

SVE2 enables optimizations for emerging applications beyond the HPC market, for example, in Machine Learning (ML) (`UDOT` instruction), Computer Vision (`TBL` and `TBX` instructions), baseband networking (`CADD` and `CMLA` instructions), genomics (`BDEP` and `BEXT` instructions), and server (`MATCH` and `NMATCH` instructions).

SVE2 enhances the overall performance of the large volume of data operations of a general-purpose processor, without requiring other off-chip accelerators.

5. Program with SVE2

This section describes the software tools and libraries that support SVE2 application development. This section also describes how to develop your application for an SVE2-enabled target, run it on SVE2-enabled hardware, and emulate your application on any Armv8-A hardware.

Software and libraries support

To build an SVE or SVE2 application, you must choose a compiler that supports SVE and SVE2 features. GNU tools versions 8.0+ support SVE. Arm Compiler for Linux versions 18.0+ support SVE, and versions 20.0+ support both SVE and SVE2. Both the GNU tools and Arm Compiler for Linux support optimizing C/C++/Fortran code. The LLVM (open-source Clang) version 5 and onwards includes support for SVE, and version 9 and onwards includes support for SVE2. To find out what SVE or SVE2 features each version of the LLVM tools support, see the [LLVM toolchain SVE support](#) page.

Arm Performance Libraries are highly optimized for math routines, and can be linked to your application. Arm Performance Libraries versions 19.3+ support math libraries for SVE.

Arm Compiler for Linux, which is part of Arm Allinea Studio, consists of the Arm C/C++ Compiler, Arm Fortran Compiler, and Arm Performance Libraries.

How to program for SVE2

There are a few ways to write or generate SVE and SVE2 code. In this section of the guide, we explore some of them.

To write or generate SVE and SVE2 code, you can write assembly with SVE and SVE2 instructions, or use intrinsics in C/C++/Fortran applications. You can let compilers auto-vectorize your code, and use the SVE-optimized libraries. Let's look at each option.

- Write assembly code: You can write assembly files using SVE instructions, or use inline assembly in GNU style. For example:

```
.globl subtract_arrays          // -- Begin function
    .p2align    2
    .type      subtract_arrays,@function
subtract_arrays:                // @subtract_arrays
    .cfi_startproc
// %bb.0:
    orr        w9, wzr, #0x400
    mov        x8, xzr
    whilelo    p0.s, xzr, x9
.LBB0_1:                        // =>This Inner Loop Header: Depth=1
    ld1w       { z0.s }, p0/z, [x1, x8, lsl #2]
    ld1w       { z1.s }, p0/z, [x2, x8, lsl #2]
    sub        z0.s, z0.s, z1.s
    st1w       { z0.s }, p0, [x0, x8, lsl #2]
    incw       x8
    whilelo    p0.s, x8, x9
    b.mi       .LBB0_1
// %bb.2:
    ret
.Lfunc_end0:
    .size      subtract_arrays, .Lfunc_end0-subtract_arrays
```

```
.cfi_endproc T
```

To program in assembly, you must know the Application Binary Interface (ABI) standard updates for SVE and SVE2. The Procedure Call Standard for Arm Architecture (AAPCS) specifies the data types and register allocations and is most relevant to programming in assembly. The AAPCS requires that:

- z0-z7, p0-p3 are used for parameter and results passing.
- z8-z15, p4-p15 are callee-saved registers.
- z16-z31 are the corruptible registers.
- Use instruction functions: You can call instruction functions directly in high-level languages like C, C++, or Fortran that match corresponding SVE instructions. These instruction functions, which are sometimes referred to as intrinsics, are detailed in the ACLE (Arm C Language Extension) for SVE. Intrinsics are functions that match to corresponding instructions, so that programmers can directly call them in high-level languages like C, C++, or Fortran. The instruction functions are inserted with specific instructions after compilation. The ACLE for SVE document also includes the full list of instruction functions for SVE2 that programmers can use.

For example, use the following code:

```
//intrinsic_example.c
#include <arm_sve.h>
svuint64_t uaddlb_array(svuint32_t Zs1, svuint32_t Zs2)
{
    // widening add of even elements
    svuint64_t result = svaddlb(Zs1, Zs2);
    return result;
}
```

Compile the code using Arm C/C++ Compiler, as you can see here:

```
armclang -O3 -S -march=armv8-a+sve2 -o intrinsic_example.s intrinsic_example.c
```

This generates the assembly code, as you can see here:

```
//intrinsic_example.s
uaddlb_array:                                // @uaddlb_array
    .cfi_startproc
    // %bb.0:
    uaddlb    z0.d, z0.s, z1.s
    ret
```

This example uses Arm Compiler for Linux 20.0.

- Auto-vectorization: C/C++/Fortran compilers, for example Arm Compiler for Linux and GNU compilers for Arm platforms, generate the SVE and SVE2 code from C/C++/Fortran loops. To generate SVE or SVE2 code, select the appropriate compiler options for the SVE or SVE2 features. For example, with `armclang`, one option that enables SVE2 optimizations is `-march=armv8-a+sve2`. Combine `-march=armv8-a+sve2` with `-armpl=sve` if you want to use the SVE version of the libraries.

- Use libraries that are optimized for SVE and SVE2: There are already highly optimized libraries with SVE available, for example Arm Performance Libraries and Arm Compute Libraries. Arm Performance Libraries contain the highly optimized implementations for BLAS, LAPACK, FFT, sparse linear algebra, and libamath optimized mathematical functions. You must install Arm Allinea Studio and include `armpl.h` in your code to be able to link any of the ArmPL functions. To build the application with ArmPL using Arm Compiler for Linux, you must specify `-armpl=<arg>` on the command line. If you use the GNU tools, you must include the ArmPL installation path on command line, and specify the GNU equivalent to the Arm Compiler for Linux `-armpl=<arg>` option.

How to run an SVE and SVE2 application: Hardware and model

If you do not have access to SVE hardware, you can use models and emulators to develop your code. There are a few models and emulators to choose from:

- QEMU: Cross and native models, supporting modeling on Arm AArch64 platforms with SVE
- Fast Models: Cross platform models, supporting modeling on Arm AArch64 platforms with SVE. Architecture Envelope Model (AEM) with SVE2 support is available for lead partners.
- Arm Instruction Emulator (ArmIE): Runs directly on Arm platforms. Supports SVE, and supports SVE2 from version 19.2+.

6. Check your knowledge

With the following questions, you can test your knowledge:

What does SVE2 inherit from SVE?

SVE2 inherits the fundamental architecture features of SVE, including the vectors, system control registers, and instructions.

What are the vectors for SVE2?

SVE2 uses the same SVE Z0-Z31 vectors, P0-P15 predicate registers, and FFR predicate register.

How many bits can SVE2 vectors have?

Z0-Z31 can be implemented from 128 bits up to 2048 bits (with 128 increment).

What features does SVE2 enable, in addition to those added by SVE?

While SVE was designed for HPC, SVE2 extends the SVE instruction set to address the requirements in application areas such as general-purpose software, web-serving server, computer vision, multi-media, LTE baseband process, genomics, and in-memory database.

What are the advantages of SVE2 compared to a traditional SIMD instruction set, for example Neon?

The advantages of SVE2, compared to Neon, include:

- SVE2 programs can be built once and run on HW with various vector lengths
- SVE2 has more vectorization flexibility
- SVE2 extends the instruction set enabling more application areas

7. Related information

Here are some resources related to material in this guide:

- [Arm architecture exploration tools](#)
- [Arm Architecture Reference Manual Supplement – The Scalable Vector Extension \(SVE\) for Armv8-A](#)
- [Arm Community](#) – Ask development questions, and find articles and blogs on specific topics from Arm experts.
- [Arm Instruction Emulator \(ArmIE\)](#)
- [Arm SVE intrinsics](#)
- [ACLE \(Arm C Language Extensions \(ACLE\) for SVE \(and SVE2\)](#)
- [Fast Models](#)
- [Introduction to Scalable Vector Extension \(SVE\)](#)
- [Neon](#)
- [QEMU](#)
- [Server and HPC software tooling documentation](#)
- [SVE and SVE2 instruction information: Arm A64 Instruction Set Architecture: Future Architecture Technologies in the A architecture profile](#)
- [SVE Supplement to Armv8-A ARM: ARM Architecture Reference Manual Supplement –The Scalable Vector Extension \(SVE\) for Armv8-A](#)
- [The Procedure Call Standard for Arm Architecture \(AAPCS\)](#)
- Learn more about porting your code to Arm, or Arm SVE-enabled, hardware in the HPC application porting guides:
 - [Porting and Optimizing HPC Applications for Arm SVE](#)
 - [Porting and Optimizing HPC Applications for Arm](#)