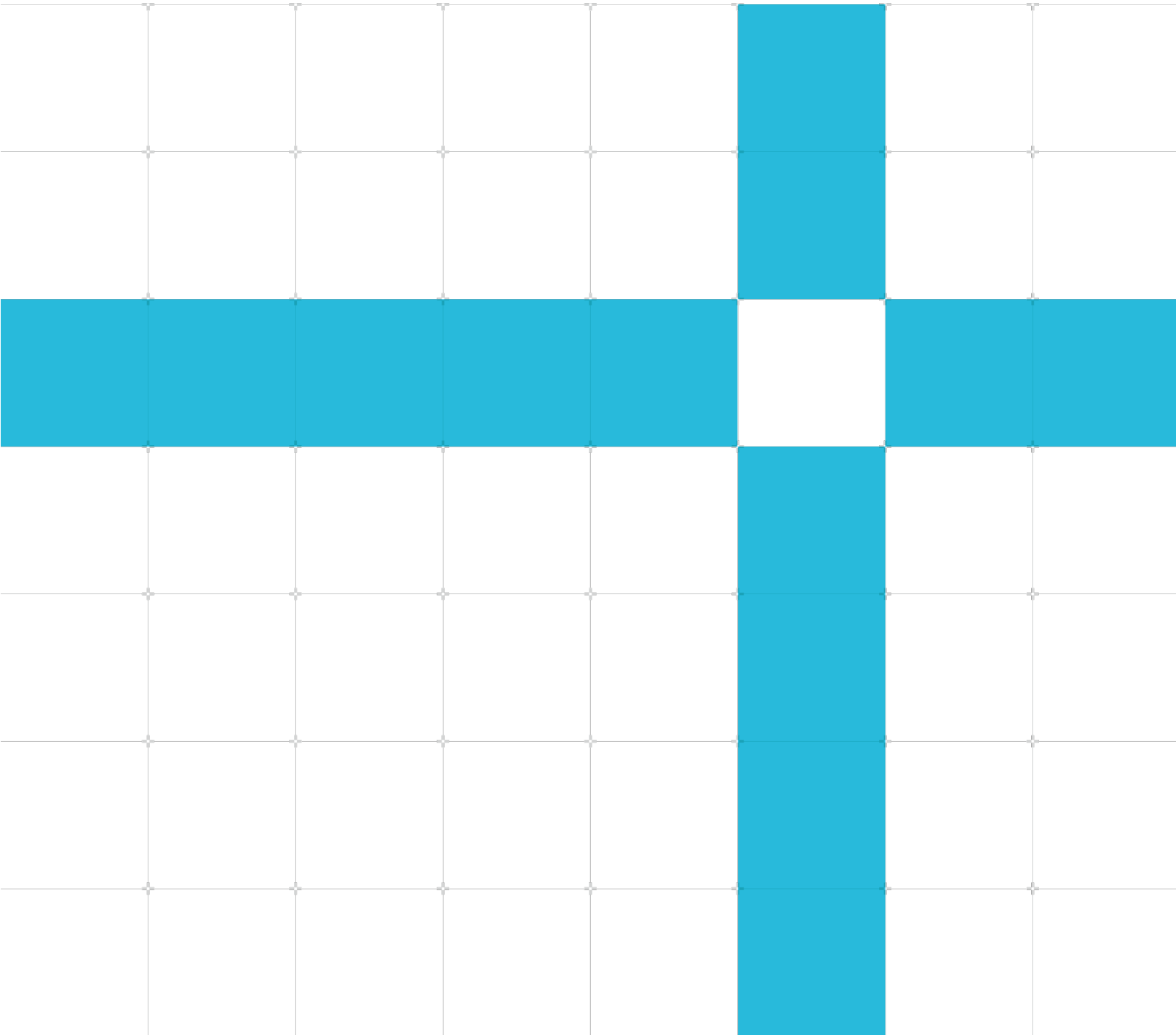
arm

SVE Optimization Guide

Non-Confidential

Issue 01
102699



SVE Optimization Guide

Release information

Document history

Issue	Date	Confidentiality	Change
01	14-Oct-2021	Non-Confidential	First issue

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

developer.arm.com

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

The Scalable Vector Extension (SVE) is an extension of the Armv8-A Architecture, available from Armv8.2-A.

SVE is designed to improve integer and floating-point performance of Arm processors through enhanced vectorization compared to NEON, Arm's existing Advanced SIMD instruction set.

Applications of SVE include machine learning (ML), high performance computing (HPC), data analysis, and potentially any compute bound software. Please see the Introduction to SVE guide for a more complete introduction.

SVE2 is an extension of the architecture which is the next phase of the technology after SVE. The main new features of SVE2 are as follows:

- An extended instruction set designed to replicate the full functionality of NEON
- Extended instructions to cover wider application domains

The examples in this guide apply to both SVE and SVE2. This guide does not make a distinction between SVE and SVE2, because the SVE Instruction Set Architecture (ISA) is a subset of the SVE2 ISA. Code written for SVE runs on SVE2 machines with no modification.

This guide shows you how to use SVE in your C and C++ code, and how to perform some basic optimizations. We compare an implementation of a numerical routine in pure C, ACLE SVE intrinsics, and SVE assembly. Comparing the performance of these different ways of using SVE will demonstrate some techniques for optimizing codes with SVE.

Any C or C++ programmer should be able to use this guide, but some experience using intrinsics or writing Arm assembly code may help. This guide is intended for:

- Beginner SVE users
- Library writers who want to use SVE
- Anyone else testing and experimenting with SVE

At the end of this guide, Check Your Knowledge tests if you understand the following concepts:

- The basics of using SVE intrinsics in the C language
- Where to find the SVE intrinsics reference and the SVE instruction set
- How to combine C and assembly code

2 Advanced SVE features for optimization

The defining feature of SVE is a vector register file which has a variable vector width. SVE is designed so that the size of the vector register does not need to be known at compile time. A single binary image can run on any microarchitecture implementing SVE, regardless of the vector width used by that implementation.

SVE adds the following functionality to the Arm architecture:

- A configurable vector length, from 128 bits up to 2048 bits in multiples of 128 bits.
- Predication, a means of controlling which elements of a vector are active.
- A dedicated SIMD instruction set that supports predication and different vector lengths.
- Gather-load and scatter-store instructions.
- Support for software-managed speculative vectorization.

This functionality is defined in the Arm Architecture Reference Manual Supplement, The Scalable Vector Extension.

There are four main ways to write code that uses SVE:

- SVE-enabled libraries, such as the Arm Performance Libraries, provide one of the easiest ways to take advantage of SVE.
- Various compiler features, such as auto-vectorization, can use SVE.
- SVE intrinsics are function calls that the compiler replaces with appropriate SVE instructions. SVE intrinsics give you access to most of the SVE instruction set directly from C and C++ code.
- SVE assembly offers great performance in certain applications but is difficult to write and maintain.

Refer to the SVE programmers guide for a full introduction to SVE.

3 Optimizing with auto-vectorization

Arm Compiler for Linux is based on LLVM Clang which has two auto-vectorization features: loop vectorization, and Superword Level Parallelism (SLP) vectorization.

The loop vectorizer calculates the optimal amount of loop unrolling and vectorization to perform for a particular loop. Loop unrolling is beneficial because it increases the number of operations performed in each iteration of the loop. Loop unrolling also enables the use of SIMD instruction sets such as NEON or SVE.

SLP vectorization enables the compiler to combine multiple nearby independent operations into a single vector instruction.

Auto-vectorizations provides the following benefits:

- Ease of use. Auto-vectorization can be controlled using only compiler flags and `#pragma` directives.
- Portable. The same source code can be recompiled for different target CPUs easily using the `-mcpu` compiler option.

However, auto-vectorization has the following disadvantages:

- Lack of control. You rely on the compiler's code generation choices, which may not produce the instructions you want.
- Performance. If the auto-vectorizer fails to identify a particular optimization opportunity, some loops must be manually vectorized with intrinsics or assembly code.

3.1 Auto-vectorization hints and tips

Use the following compiler flags to control auto-vectorization for all loops in the source code:

Compiler option	Description	Notes
`-fvectorize`	Enables the loop vectorizer	default at `-O2` or higher
`-no-fvectorize`	Disables the loop vectorizer	default at `-O1` and `-O0`
`-mllvm -force-vector-width=<value>`	Set the SIMD vector width	The loop vectorizer will vectorize operations into SIMD operations of this width
`-mllvm -force-vector-interleave=<value>`	Set the loop unroll factor	The loop vectorizer will unroll loops by `<value>`
`-fno-slp-vectorize`	Disable the SLP vectorizer	SLP vectorization is enabled by default
`-Rpass=loop-vectorize`	Remarks on loops that were successfully vectorized	
`-Rpass-missed=loop-vectorize`	Remarks on loops were not vectorized	

Compiler option	Description	Notes
`-Rpass-analysis=loop-vectorize`	Remarks on code statements that caused vectorization to fail	

Table 1: Compiler options for auto-vectorization

To control vectorization of individual loops, use the following pragmas:

Pragma	Description
`#pragma clang loop vectorize(enable)`	Enable vectorization.
`#pragma clang loop vectorize(disable)`	Disable vectorization.
`#pragma clang loop interleave(enable)`	Enable loop unrolling.
`#pragma clang loop interleave(disable)`	Disable loop unrolling.
`#pragma clang loop vectorize_width(2)`	Enable vectorization and set the SIMD vector width.
`#pragma clang loop interleave_count(2)`	Enable loop unrolling and set the unroll count.
`#pragma clang loop vectorize(assume_safety)`	Instruct the compiler to assume that successive iterations of the loop are independent.

Table 2: Pragmas for loop vectorization

The pragmas must be placed before a loop and apply only to that loop. For each loop, LLVM uses a cost model to balance the expected performance gain from unrolling and vectorizing, with the increase in code size, loop tails, and extra instructions. The pragmas act as hints to ignore this cost model, but they are not guaranteed to be respected.

To increase the chance of a loop being vectorized, consider the following:

- Avoid dependencies between loop iterations
- Avoid switch statements
- Use the `-Rpass-analysis=loop-vectorize` to understand why code is not vectorizing
- Enable optimization by compiling at optimization level `-O2` or higher
- Use pragmas on specific loops to control vectorization

3.2 Auto-vectorization example: SAXPY

Single-precision scaled x plus y (SAXPY) is an L1 BLAS routine defining vector addition. SAXPY is commonly used in numerical software. Because the routine defines a basic mathematical operation, optimizing SAXPY could provide performance improvements for applications in domains such as signal processing, HPC, ML, or game engine design.

Consider the following operation: y[i] ← a*x[i] + y[i]. This operation takes an array of data x[i], multiplies it by the scalar a, then adds y[i], before assigning back to y. A simple C implementation would be as follows:

```
/*
```

```
Copyright (C) Arm Limited, 2021 All rights reserved.
The example code is provided to you as an aid to learning when working
with Arm-based technology, including but not limited to programming tutorials.
Arm hereby grants to you, subject to the terms and conditions of this Licence,
a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
to use and copy the Software solely for the purpose of demonstration and
evaluation.
You accept that the Software has not been tested by Arm therefore the Software
is provided "as is", without warranty of any kind, express or implied. In no
event shall the authors or copyright holders be liable for any claim, damages
or other liability, whether in action or contract, tort or otherwise, arising
from, out of or in connection with the Software or the use of Software.
*/

#include <arm_sve.h>

void saxpy_c(float32_t *x, float32_t *y, float32_t a, uint32_t n) {
        // scaled vector add: y = a*x + y
        int i;
        for (i=0; i<n; i++) {
                y[i] = a*x[i] + y[i];
        }
}
```

Since all the iterations of the loop are independent of each other, this example can be optimized by vectorizing the addition and assignment: y[i] = a*x[i] + y[i]. We can then compare the disassembly of the code with and without vectorization.

Here is the code produced by compiling with `armclang -march=armv8.2-a+sve -O1`, which performs no vectorization:

```
0000000000000000 <saxpy_c>:
       0: 22 01 00 34   cbz     w2, 0x24 <saxpy_c+0x24>
       4: e8 03 02 2a   mov     w8, w2
       8: 01 44 40 bc   ldr     s1, [x0], #4
       c: 22 00 40 bd   ldr     s2, [x1]
      10: 08 05 00 f1   subs    x8, x8, #1
      14: 21 08 20 1e   fmul    s1, s1, s0
      18: 21 28 22 1e   fadd    s1, s1, s2
      1c: 21 44 00 bc   str     s1, [x1], #4
      20: 41 ff ff 54   b.ne    0x8 <saxpy_c+0x8>
      24: c0 03 5f d6   ret
```

The `FMUL` and `FADD` instructions perform the operation a*x[i] + y[i] and operate on the scalar floating-point registers `s1` and `s0`, rather than vector registers. Although the Arm v8.2-A architecture contains NEON `FMUL` and `FADD` operations, at optimization level `-O1`, no vector operations are used.

For comparison, the following code is generated when compiling with optimization level `-O2`, which enables auto-vectorization:

```
0000000000000000 <saxpy_c>:
       0: 82 03 00 34   cbz     w2, 0x70 <saxpy_c+0x70>
       4: e8 03 02 2a   mov     w8, w2
       8: 09 f5 7e d3   lsl     x9, x8, #2
       c: 0a 00 09 8b   add     x10, x0, x9
      10: 5f 01 01 eb   cmp     x10, x1
      14: 89 01 00 54   b.ls    0x44 <saxpy_c+0x44>
      18: 29 00 09 8b   add     x9, x1, x9
      1c: 3f 01 00 eb   cmp     x9, x0
      20: 29 01 00 54   b.ls    0x44 <saxpy_c+0x44>
```

```
      24: 01 44 40 bc   ldr     s1, [x0], #4
      28: 22 00 40 bd   ldr     s2, [x1]
      2c: 08 05 00 f1   subs    x8, x8, #1
      30: 21 08 20 1e   fmul    s1, s1, s0
      34: 21 28 22 1e   fadd    s1, s1, s2
      38: 21 44 00 bc   str     s1, [x1], #4
      3c: 41 ff ff 54   b.ne    0x24 <saxpy_c+0x24>
      40: 0c 00 00 14   b       0x70 <saxpy_c+0x70>
      44: e9 03 1f aa   mov     x9, xzr
      48: e0 1f a8 25   whilelo p0.s, xzr, x8
      4c: 00 20 24 05   mov     z0.s, s0
      50: 01 40 49 a5   ld1w    { z1.s }, p0/z, [x0, x9, lsl #2]
      54: 22 40 49 a5   ld1w    { z2.s }, p0/z, [x1, x9, lsl #2]
      58: 21 08 80 65   fmul    z1.s, z1.s, z0.s
      5c: 21 00 82 65   fadd    z1.s, z1.s, z2.s
      60: 21 40 49 e5   st1w    { z1.s }, p0, [x1, x9, lsl #2]
      64: e9 e3 b0 04   incw    x9
      68: 20 1d a8 25   whilelo p0.s, x9, x8
      6c: 24 ff ff 54   b.mi    0x50 <saxpy_c+0x50>
      70: c0 03 5f d6   ret
```

Now let us compare with fast optimization `-Ofast`:

```
0000000000000000 <saxpy_c>:
       0: 62 03 00 34   cbz     w2, 0x6c <saxpy_c+0x6c>
       4: e8 03 02 2a   mov     w8, w2
       8: 09 f5 7e d3   lsl     x9, x8, #2
       c: 0a 00 09 8b   add     x10, x0, x9
      10: 5f 01 01 eb   cmp     x10, x1
      14: 69 01 00 54   b.ls    0x40 <saxpy_c+0x40>
      18: 29 00 09 8b   add     x9, x1, x9
      1c: 3f 01 00 eb   cmp     x9, x0
      20: 09 01 00 54   b.ls    0x40 <saxpy_c+0x40>
      24: 01 44 40 bc   ldr     s1, [x0], #4
      28: 22 00 40 bd   ldr     s2, [x1]
      2c: 08 05 00 f1   subs    x8, x8, #1
      30: 21 08 00 1f   fmadd   s1, s1, s0, s2
      34: 21 44 00 bc   str     s1, [x1], #4
      38: 61 ff ff 54   b.ne    0x24 <saxpy_c+0x24>
      3c: 0c 00 00 14   b       0x6c <saxpy_c+0x6c>
      40: e9 03 1f aa   mov     x9, xzr
      44: e1 1f a8 25   whilelo p1.s, xzr, x8
      48: 00 20 24 05   mov     z0.s, s0
      4c: e0 e3 98 25   ptrue   p0.s
      50: 01 44 49 a5   ld1w    { z1.s }, p1/z, [x0, x9, lsl #2]
      54: 22 44 49 a5   ld1w    { z2.s }, p1/z, [x1, x9, lsl #2]
      58: 01 80 a2 65   fmad    z1.s, p0/m, z0.s, z2.s
      5c: 21 44 49 e5   st1w    { z1.s }, p1, [x1, x9, lsl #2]
      60: e9 e3 b0 04   incw    x9
      64: 21 1d a8 25   whilelo p1.s, x9, x8
      68: 44 ff ff 54   b.mi    0x50 <saxpy_c+0x50>
      6c: c0 03 5f d6   ret
```

Compare lines 58 and 5C when compiling at `-O2`, with line offset 58 at `-Ofast`. We can see that the main difference between `-Ofast` and `-O2` is the presence of `FMAD`, a single fused multiply-add, instead of `FMUL` and `FADD`. Since this computation now takes only one instruction there may be a small performance gain in the loop.

4 Optimizing with intrinsics

The SVE intrinsics are a set of functions and macros defined in the `arm_sve.h` header file. The definitions of these functions are known to the compiler, and there is a close correspondence between the Instruction Set Architecture (ISA) and the intrinsics. For example, the `svld1` intrinsics correspond to various forms of the `LD1` instruction. Using intrinsics gives you control over SVE code generation, without the need for assembly programming.

See the ARM C Language Extensions for SVE for the specification of the intrinsics.

Intrinsics provide the following benefits:

- Powerful. Intrinsics allow the programmer to inline assembly code without having to explicitly program in assembly.
- Portable. Code containing intrinsics can be compiled for different SVE enabled platforms by setting the `-mcpu` option. However, this may not yield the same performance.
- Flexible. The programmer can use intrinsics or C/C++ code in the same program.

However, intrinsics have the following disadvantages:

- Ease-of-Use. The learning curve for intrinsics may make them unsuitable for some projects. Some knowledge of the target hardware is needed.
- Performance. User-optimized assembly may still offer the greatest scope for performance gains.
- Compiler auto-vectorization may be sufficient for many projects.
- Performance portability. Code with intrinsics written for one platform may not perform well on another platform.

Arm recommends using intrinsics over assembly because code with intrinsics is easier to port and maintain.

The key to applying SVE intrinsics is reading the ARM C Language Extensions for SVE specification. The following sections in this guide outline some of the important parts of this specification.

4.1 Macros

The following macros indicate which features are enabled by the compiler.

Macro	Description
`__ARM_FEATURE_SVE==1`	The compiler is SVE enabled and the intrinsics are available
`__ARM_FEATURE_SVE_BF16==1`	The BFloat16 extensions are enabled
`__ARM_FEATURE_SVE_MATMUL_INT8==1`	The INT8 matrix multiply extensions are enabled
`__ARM_FEATURE_SVE_MATMUL_FP32==1`	The FP32 matrix multiply extensions are enabled
`__ARM_FEATURE_SVE_MATMUL_FP64==1`	The FP64 matrix multiply extensions are enabled
`__ARM_FEATURE_SVE2==1`	The compiler is SVE2 enabled and the SVE2 intrinsics are enabled

Macro	Description
`__ARM_FEATURE_SVE2_BITPERM==1`	The SVE2 bit permutation instructions are available
`__ARM_FEATURE_SVE2_AES==1`	The SVE2 AES-128 functions are enabled
`__ARM_FEATURE_SVE2_SHA3==1`	The SVE2 SHA-3 functions are enabled
`__ARM_FEATURE_SVE2_SM4==1`	The SM4 functions are available
`__ARM_FEATURE_SVE_BITS==N`	The compiler is generating code for a target of known SVE vector length
`__ARM_FEATURE_SVE_VECTOR_OPERATORS==1`	If __ARM_FEATURE_SVE_BITS is nonzero, SVE vector types support the GNU vector extensions
`__ARM_FEATURE_SVE_PREDICATE_OPERATORS==1`	If __ARM_FEATURE_SVE_BITS is nonzero, svbool_t supports vector operations
`__ARM_FEATURE_SVE_NONMEMBER_OPERATORS==1`	C++ code can define non-member operator functions for SVE types

Table 3: Macros

4.2 Types

Several sizeless types are defined by the ACLE specification. Sizeless types are necessary because the length of the SVE vector registers are not known to the compiler. The arithmetic types have the following pattern:

```
sv<type>_t
```

For example, a scalable vector type of 16-bit signed integers is: `svint16_t`.

There is also the type `svbool_t` which is used to represent predicates.

4.3 Functions

The SVE intrinsics use the following general naming convention:

```
svbase[_disambiguator][_type0][_type1]…[_predication]
```

Where:

- `base` is the name of an instruction. For example, `svmla_n_f32_m()` corresponds to the `MLA` instruction.
- `_disambiguator` indicates any special behavior of the function. For example, `svld1_gather_s32_offset_u32()` indicates a gather load `LD1`.
- `_type0`, `_type1` and so on indicate the types of the vectors being operated on. For example, `svld1_f32()` loads 32-bit floats.
- `_predication` indicates a zeroing, `z`, or merging, `m`, operation. Predicates control which lanes in a vector are active, and which are inactive. Zeroing operations set inactive elements to zero in an operation. Merging operations leave the existing inactive elements unchanged.

4.4 Intrinsics hints and tips

Writing code with intrinsics is very similar to writing standard C code. The compiler automates many optimizations, including the following:

- Optimized allocation of variables to registers.
- Application of the procedure call standard rules.
- Auto-vectorization of any remaining parts of the loop without intrinsics
- Instruction selection - where there is no intrinsic, the compiler will choose optimal instructions.

However, the programmer is responsible for the following:

- Intrinsics selection. The programmer must choose the correct intrinsics for their program. The programmer must be aware of the instruction timings to manually optimize the program.
- Loop unrolling. The programmer must unroll loops that contain intrinsics. Loop tails must be written manually.
- Vectorization. The correct vector load, store, and arithmetic operations must be selected.

4.5 Intrinsics example: SAXPY

This section explains how SAXPY can be broken down into parts and implemented using intrinsics.

There are two parts to SAXPY :

- The arithmetic operation y[i] ← a*x[i] + y[i]
- The loop that iterates over all the values.

With SVE intrinsics we must rethink the loop, and the operation, by considering what instructions are necessary.

We need the following:

- A measure of the vector length, used to increment the loop
- A loop to iterate over the vectors
- A governing predicate, to make sure there are no out of bounds memory accesses
- Load operations to get the values x[i] and y[i]
- Floating-point operations to perform the multiply (a*x[i]) and add (+y[i])
- A store operation for the result of the multiply and add.

Once we know what to look for and what types we are using, the SVE intrinsics specification can be used to identify the intrinsics we need.

The following code shows a C implementation of SAXPY using SVE intrinsics:

```
/*
Copyright (C) Arm Limited, 2021 All rights reserved.
The example code is provided to you as an aid to learning when working
with Arm-based technology, including but not limited to programming tutorials.
Arm hereby grants to you, subject to the terms and conditions of this Licence,
a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
to use and copy the Software solely for the purpose of demonstration and
evaluation.
You accept that the Software has not been tested by Arm therefore the Software
is provided "as is", without warranty of any kind, express or implied. In no
event shall the authors or copyright holders be liable for any claim, damages
or other liability, whether in action or contract, tort or otherwise, arising
from, out of or in connection with the Software or the use of Software.
*/

#include <arm_sve.h>

void saxpy_sve(float32_t *x, float32_t *y, float32_t a, uint32_t n) {
        uint32_t i;

        // define predicate and 'live' segments of x/y
        svbool_t predicate;
        svfloat32_t xseg;
        svfloat32_t yseg;

        // get the vector length being used, so we know how to increment the loop (1)
        uint64_t numVals = svlen_f32(xseg);

        for (i=0; i<n; i+=numVals) { // (2)
                // set predicate based off loop counter (3)
                predicate = svwhilelt_b32_s32(i, n);

                // load in a vectors worth of x and y values (4)
                xseg = svld1_f32(predicate, x+i); // ld1w for x
                yseg = svld1_f32(predicate, y+i); // ld1w for y

                // perform the a*x[i] + y[i] operation in one go with MLA (5)
                yseg = svmla_n_f32_m(predicate, yseg, xseg, a);  // y+a*x

                // store yvalues (6)
                svst1_f32(predicate, y+i, yseg); // st1w for y <-y+a*x
        }
}
```

This code uses the following intrinsics:

Code element	Description	Why are we using it?
`svbool_t`	SVE bool type declaration	We need a special type for the compiler to represent predicate registers
`svfloat32_t`	SVE float32_t type declaration	Since the compiler must generate vector length agnostic code, we need a special sizeless type to represent scalable vector registers.
`svlen_f32()`	Corresponds to the CNTB, CNTD, CNTH, CNTW instructions (CNTW in this case)	We need to know how long the SVE vector is to increment the loop. svlen_f32() lets us determine this at runtime.

Code element	Description	Why are we using it?
`svwhilelt_b32_s32()`	Corresponds to the WHILELO/WHILELE instructions	svwhilelt_b32_s32() is necessary to set the predicate based on the loop counter, and the loop length. When the counter gets close to the length, svwhilelt_b32_s32() writes zeroes into the predicate which will prevent out of bounds loads and stores from occurring.
`svld1_f32()`	Corresponds to the LD1 instruction	SVE intrinsics expose explicit load/store operations to the programmer. These are needed to retrieve data from memory which can then be used.
`svmla_n_f32_m()`	Corresponds to the MLA instruction	Since the MLA instruction performs a*b +c, it is ideal for the operation we are performing.
`svst1_f32()`	Corresponds to the ST1 instruction	SVE intrinsics expose explicit load/store operations to the programmer. These are needed to store data back to memory.

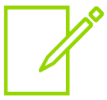
Table 4: Code intrinsics

We can examine the disassembly to see the generated instructions:

```
saxpy_sve(float*, float*, float, unsigned int):                       //
@saxpy_sve(float*, float*, float, unsigned int)
        cbz     w2, .LBB0_3
        rdvl    x9, #1
        lsr     x9, x9, #4
        mov     w8, wzr
        lsl     w9, w9, #2
        mov     z0.s, s0
.LBB0_2:                                // =>This Inner Loop Header: Depth=1
        whilelt p0.s, w8, w2
        mov     w10, w8
        ld1w    { z1.s }, p0/z, [x0, x10, lsl #2]
        ld1w    { z2.s }, p0/z, [x1, x10, lsl #2]
        add     w8, w8, w9
        cmp     w8, w2
        fmla    z2.s, p0/m, z1.s, z0.s
        st1w    { z2.s }, p0, [x1, x10, lsl #2]
        b.lo    .LBB0_2
.LBB0_3:
        ret
```

5 Optimizing with assembly

Instead of using a high-level language such as C/C++, we can use assembly code to write programs by explicitly specifying instructions. These instructions are then assembled by an assembler into machine code that can be executed by the processor.


Note

Compilers such as armclang have a built-in assembler.

Writing assembly code has the following advantages:

- Control. The program is exactly what the programmer specifies.
- Performance. It is possible to achieve great performance by coding directly in assembly, but the programmer must be better than the compiler. Modern compilers are hard to beat in many situations.

Writing assembly code also has the following disadvantages, however:

- Lack of portability. Code written for one processor may not run on another if the ISAs are different. For example, SVE assembly code will run on the Cortex-X2, but not the Cortex-A72. The programmer would have to rewrite the code.
- Lack of performance portability. Programs must be tuned for the specific processor you want to run the code on. Running the code on a different processor is likely to have a performance penalty.
- Difficulty. There is a steep learning curve for assembly programming.
- Hard to maintain. It can be difficult to understand existing assembly programs, which makes them hard to maintain in long-term projects.
- Hardware is not abstract, the programmer must be aware of the implementation details such as caches, pipelines, and the memory system, to write optimal programs.

5.1 Assembly hints and tips

Programming in assembly is difficult because the program must be written by hand. Everything from program correctness to optimization is the responsibility of the programmer.

5.1.1 Instruction selection

The following hints and tips relate to instruction selection:

- Instructions can have different execution times on different microarchitecture implementations. The programmer must choose which instructions to use to manually optimize the program.
- Be aware of the trade-offs of performance and convenience for some instructions, for example gather load and scatter store.

- Some logical operations, for example `CSEL` could perform better than branch instructions in very unpredictable branches. However normal predictable branches perform better than instructions like `CSEL`.
- Instructions such as `TBL` can gain performance improvement if you can fit the lookup table in one vector.

5.1.2 Register usage

The following hints and tips relate to register usage:

- Always comply with the Procedure Call Standard.
- Load registers as far in advance as possible before the value is used to avoid stalls in the pipelines.
- Moves, loads, and stores have a cost. Retain individual values in registers for as long as possible.
- Where possible, use destination registers immediately as source registers in the next instruction. This minimizes register usage.
- Backup registers by spilling registers to the stack to ensure values are not lost or corrupted.

5.1.3 Loops

Loop unrolling can improve performance by increasing pipeline efficiency. Consider the following pipeline diagram for the Cortex-X2:

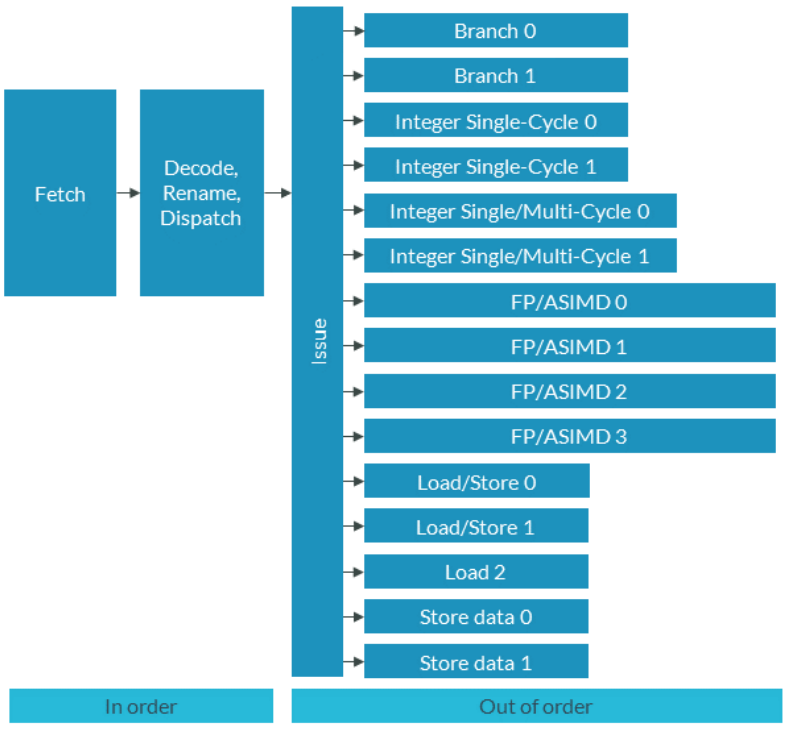


Figure 1 – Cortex-X2 pipeline diagram

If each loop iteration is independent of the previous and next, the extra arithmetic instructions from the loop unrolling can be issued to different pipelines. Since we have four integer and four floating-point pipelines, we could unroll the loop at least four times to make full use of them. The Cortex-X2 Software Optimization Guide contains specific information about the Cortex-X2 that is useful to the programmer, including the following:

- Instruction timings
- Pipeline diagram
- Information about the memory system and cache utilization

5.1.4 Cache

The following hints and tips relate to caching:

- The ordering of load and store instructions affect data being loaded into and evicted from the cache.
- Preloading data into caches in advance might help to improve performance. However, misusing these instructions could lead to a loss of useful data from the cache which will reduce performance.

5.2 Assembly example: SAXPY

Let us see how the assembly kernel is built up for SAXPY. First, we must define the interface between our C code and the assembly kernel. In the C code we add the following declaration:

```
extern void saxpy_asm(float32_t *x, float32_t *y, float32_t a, uint32_t n);
```

This creates the function symbol and the signature defines the input arguments. The ordering of the input arguments is important because it determines which registers those values will be in. The full set of rules is defined in the Procedure Call Standard and we apply them in the following table:

Input argument	Corresponding register	Notes
`float32_t *x`	x0	64-bit pointers in AArch64
`float32_t *y`	x1	64-bit pointers in AArch64
`float32_t a`	s0	Bottom 32 bits of v0
`uint32_t n`	w2	Bottom 32 bits of x2

Table 5: Input arguments

Now we have defined the interface in C we are able to write the assembly kernel in a separate file, which we will compile and link to our C code. The assembly is as follows:

```
/*
Copyright (C) Arm Limited, 2021 All rights reserved.
The example code is provided to you as an aid to learning when working
with Arm-based technology, including but not limited to programming tutorials.
Arm hereby grants to you, subject to the terms and conditions of this Licence,
a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
to use and copy the Software solely for the purpose of demonstration and
```

```
evaluation.
You accept that the Software has not been tested by Arm therefore the Software
is provided "as is", without warranty of any kind, express or implied. In no
event shall the authors or copyright holders be liable for any claim, damages
or other liability, whether in action or contract, tort or otherwise, arising
from, out of or in connection with the Software or the use of Software.
*/

/* SAXPY, scaled X plus Y
* extern void saxpy_asm(float32_t *x, float32_t *y, float32_t a, uint32_t n)
* Y <- Y + a*X
*`
*/

# Input Argument Aliases
x_base_addr .req    x0
y_base_addr .req    x1
a   .req    s0
n   .req    x2

# Local Variable Aliases

p_op    .req    p0
i_idx   .req    x5

a_vals  .req    z0
x_vals  .req    z1
y_vals  .req    z2

#define RZERO(register) eor register, register, register

    .global saxpy_asm
    .type   saxpy_asm, %function
saxpy_asm:
    // save state, rules in the procedure call standard
    stp x29, x30, [sp, #-320]!
    mov x29, sp
    stp x19, x20, [sp, #224]
    stp x21, x22, [sp, #208]
    stp x23, x24, [sp, #192]
    stp x25, x26, [sp, #176]
    stp x27, x28, [sp, #160]
    stp d8, d9,   [sp, #80]
    stp d10, d11, [sp, #64]
    stp d12, d13, [sp, #48]
    stp d14, d15, [sp, #32]

    RZERO(i_idx)
    dup a_vals.s, a_vals.s[0]

.L_loop:
    // set predicate from our index and the total number of values
    whilelo p_op.s, i_idx, n

    // load x and y values
    ld1w x_vals.s, p_op/z, [x_base_addr, i_idx, lsl 2]
    ld1w y_vals.s, p_op/z, [y_base_addr, i_idx, lsl 2]

    // perform the y <- a*x + y operation
    fmla y_vals.s, p_op/m, a_vals.s, x_vals.s

    // store our new value for y over the old ones
    st1w y_vals.s, p_op, [y_base_addr, i_idx, lsl 2]
```

```
.L_cond:
    // increment the index by the number of 32 bit values in the Z registers
    incw i_idx
    b.first .L_loop

.L_saxpy_asm_end:
    // restore state
    ldp x19, x20, [sp, #224]
    ldp x21, x22, [sp, #208]
    ldp x23, x24, [sp, #192]
    ldp x25, x26, [sp, #176]
    ldp x27, x28, [sp, #160]
    ldp d8, d9,   [sp, #80]
    ldp d10, d11, [sp, #64]
    ldp d12, d13, [sp, #48]
    ldp d14, d15, [sp, #32]
    ldp x29, x30, [sp], #320
    ret
```

6 Complete code listing

This section of the guide contains the C code listing, including helper functions. This code is also available as an archive which you can download from the Arm Developer site.

Using Arm Compiler for Linux, we can compile the code as follows:

```
armclang -march=armv8.2a+sve -g -O3 -c saxpy_example.c -o saxpy_example.o
armclang -march=armv8.2a+sve -g -O3 -c saxpy_asm.S -o saxpy_asm.o
```

Then link the object files:

```
armclang -march=armv8.2a+sve -g -O3 saxpy_example.o saxpy_asm.o -o saxpy.out
```

These steps, along with the code, are included in a Makefile in the supplied archive.

If you have an Arm system without SVE, you can use the Arm Instruction Emulator to run SVE code as follows:

```
armie -msve-vector-bits=128 ./saxpy.out
```

6.1 saxpy_example.c

The code listing for the `saxpy_example.c` source file is as follows:

```
/*
Copyright (C) Arm Limited, 2021 All rights reserved.
The example code is provided to you as an aid to learning when working
with Arm-based technology, including but not limited to programming tutorials.
Arm hereby grants to you, subject to the terms and conditions of this Licence,
a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
to use and copy the Software solely for the purpose of demonstration and
evaluation.
You accept that the Software has not been tested by Arm therefore the Software
is provided "as is", without warranty of any kind, express or implied. In no
event shall the authors or copyright holders be liable for any claim, damages
or other liability, whether in action or contract, tort or otherwise, arising
from, out of or in connection with the Software or the use of Software.
*/

#include <arm_sve.h>
#include <stdio.h>

void saxpy_c(float32_t *x, float32_t *y, float32_t a, uint32_t n) {
        // scaled vector add: y = a*x + y
        int i;
        for (i=0; i<n; i++) {
                y[i] = a*x[i] + y[i];
        }
}


void saxpy_sve(float32_t *x, float32_t *y, float32_t a, uint32_t n) {
        uint32_t i;

        svbool_t predicate;
        svfloat32_t xseg;
        svfloat32_t yseg;
```

```
        uint64_t numVals = svlen_f32(xseg);

        for (i=0; i<n; i+=numVals) {
                predicate = svwhilelt_b32_s32(i, n);

                xseg = svld1_f32(predicate, x+i); // ld1w for x
                yseg = svld1_f32(predicate, y+i); // ld1w for y

                yseg = svmla_n_f32_m(predicate, yseg, xseg, a);  // y+a*x

                svst1_f32(predicate, y+i, yseg); // st1w for y <-y+a*x
        }
}

extern void saxpy_asm(float32_t *x, float32_t *y, float32_t a, uint32_t n);

void vecprint(float32_t *vec, uint32_t n) {
        int i;
        for (i=0; i<n; i++) {
                printf("[%f]", vec[i]);
        }
        printf("\n");
}

void vecset(float32_t *vec, float32_t val, uint32_t n) {
        int i;
        for (i=0; i<n; i++) {
                vec[i] = val;
        }
}


int main() {
        uint32_t N = 10;
        float32_t x[N];
        float32_t y[N];

        vecset(x, 1.5, N);
        vecset(y, 2.0, N);

        vecprint(x, N);
        vecprint(y, N);

        // choose implementation
        saxpy_c(x, y, 2.0, N);
        //saxpy_sve(x, y, 2.0, N);
        //saxpy_asm(x, y, 2.0, N);

        vecprint(y, N);
}
```

6.2 saxpy_asm.S

The code listing for the `saxpy_asm.S` source file is as follows:

```
/*
Copyright (C) Arm Limited, 2021 All rights reserved.
The example code is provided to you as an aid to learning when working
with Arm-based technology, including but not limited to programming tutorials.
```

```
Arm hereby grants to you, subject to the terms and conditions of this Licence,
a non-exclusive, non-transferable, non-sub-licensable, free-of-charge licence,
to use and copy the Software solely for the purpose of demonstration and
evaluation.
You accept that the Software has not been tested by Arm therefore the Software
is provided "as is", without warranty of any kind, express or implied. In no
event shall the authors or copyright holders be liable for any claim, damages
or other liability, whether in action or contract, tort or otherwise, arising
from, out of or in connection with the Software or the use of Software.
*/

/* SAXPY, scaled X plus Y
* extern void saxpy_asm(float32_t *x, float32_t *y, float32_t a, uint32_t n)
* Y <- Y + a*X
*`
*/

# Input Argument Aliases
x_base_addr .req    x0
y_base_addr .req    x1
a   .req    s0
n   .req    x2

# Local Variable Aliases

p_op    .req    p0
i_idx   .req    x5

a_vals  .req    z0
x_vals  .req    z1
y_vals  .req    z2

#define RZERO(register) eor register, register, register

    .global saxpy_asm
    .type   saxpy_asm, %function
saxpy_asm:
    // save state, rules in the procedure call standard
    stp x29, x30, [sp, #-320]!
    mov x29, sp
    stp x19, x20, [sp, #224]
    stp x21, x22, [sp, #208]
    stp x23, x24, [sp, #192]
    stp x25, x26, [sp, #176]
    stp x27, x28, [sp, #160]
    stp d8, d9,   [sp, #80]
    stp d10, d11, [sp, #64]
    stp d12, d13, [sp, #48]
    stp d14, d15, [sp, #32]

    RZERO(i_idx)
    dup a_vals.s, a_vals.s[0]

.L_loop:
    // set predicate from our index and the total number of values
    whilelo p_op.s, i_idx, n

    // load x and y values
    ld1w x_vals.s, p_op/z, [x_base_addr, i_idx, lsl 2]
    ld1w y_vals.s, p_op/z, [y_base_addr, i_idx, lsl 2]

    // perform the y <- a*x + y operation
    fmla y_vals.s, p_op/m, a_vals.s, x_vals.s
```

```
    // store our new value for y over the old ones
    st1w y_vals.s, p_op, [y_base_addr, i_idx, lsl 2]

.L_cond:
    // increment the index by the number of 32 bit values in the Z registers
    incw i_idx
    b.first .L_loop

.L_saxpy_asm_end:
    // restore state
    ldp x19, x20, [sp, #224]
    ldp x21, x22, [sp, #208]
    ldp x23, x24, [sp, #192]
    ldp x25, x26, [sp, #176]
    ldp x27, x28, [sp, #160]
    ldp d8, d9,   [sp, #80]
    ldp d10, d11, [sp, #64]
    ldp d12, d13, [sp, #48]
    ldp d14, d15, [sp, #32]
    ldp x29, x30, [sp], #320
    ret
```

7 Conclusion

To test optimized performance, we ran the code on an SVE enabled machine and measured execution time using the Linux perf tool.

The results were as follows:

Method	Speed improvement	Notes
C, no vectorization	base	Compiled with -O1
C, with vectorization	27.5%	Compiled with -O3
SVE intrinsics	33.2%	
SVE assembly	34.7%	

Table 6: Results

Auto-vectorization offers a significant speedup, while being trivial to use. SVE intrinsics offer a noticeable speed improvement but are harder to use. Finally, SVE assembly offers great scope for performance but is much harder to write and maintain.

8 Check your knowledge

What are the four ways of using SVE instructions?

- SVE Assembly
- SVE Intrinsics
- SVE enabled compiler
- Library that uses SVE

What C header file must you include to use the intrinsics?

- arm_sve.h (#include <arm_sve.h>)

What does the svst1_f32(p, y, z) intrinsic function do?

From the name we can see that this intrinsic corresponds to the 32-bit form `ST1` instruction. This instruction will store the values in `z` at the address given by `y`, and the operation will be masked with the predicate `p`.

What is the difference between the float32_t and svfloat32_t types?

- float32_t is a fixed width 32 bit float.
- svfloat32_t is a sizeless type of 32 bit floats.
- Sizeless types are necessary for the compiler to represent the scalable vector registers.

What engineering resources can you use for SVE programming?

- The SVE ACLE Specification
- The Architecture Specification
- Arm Developer Guides
- The Procedure Call Standard

9 Related information

Here are some resources related to material in this guide:

- The Instruction Set Architecture Extension can be found in Arm Architecture Reference Manual Supplement, The Scalable Vector Extension
- Engineering specifications for the SVE intrinsics can be found in the Arm C Language Extensions for SVE.
- The Architecture Exploration Tools let you investigate the Arm instruction set.
- The Arm Architecture Reference Manual provides a complete specification of the Arm Instruction Set.
- Arm training:
  - Introduction to Armv8-A
  - Overview ISA
  - Arm's other architectures

10 Next steps

As a next step, continue learning about SVE and SVE2 programming with the SVE and SVE2 Programmer's Guide.

From the fundamentals to more advanced concepts, these guides introduce the SVE and SVE2 extensions to the Arm Armv8-A architecture.