arm

Learn the architecture - Migrate Neon to SVE

Version 1.0

Non-Confidential

Issue 03
102131_0100_03_en

Learn the architecture - Migrate Neon to SVE

Release information

Document history

Issue	Date	Confidentiality	Change
0100-03	20 April 2022	Non-Confidential	Format changes

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.

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

This guide summarizes the important differences between coding for the Scalable Vector Extension (SVE) and coding for Neon. For users who have already ported their applications to Armv8-A Neon hardware, the guide also highlights the key differences to consider when porting an application to SVE.

Arm Neon technology is the Advanced Single Instruction Multiple Data (SIMD) feature for the Armv8-A architecture profile. Neon is a feature of the Instruction Set Architecture (ISA), providing instructions that can perform mathematical operations in parallel on multiple data streams.

SVE is the next-generation SIMD extension of the Armv8-A instruction set. It is not an extension of Neon, but is a new set of vector instructions that were developed to target HPC workloads. In short, SVE enables vectorization of loops which would be impossible, or not beneficial, to vectorize with Neon. Importantly, and unlike other SIMD architectures, SVE can be Vector Length Agnostic (VLA). VLA means that the size of the vector registers is not fixed. Instead, hardware implementors are free to choose the size that works best for the intended workloads.

At the end of this guide, you can check your knowledge. You will have learned the fundamental differences between SVE and Neon, including register types, predicating instructions, and Vector Length Agnostic programming.

The first part of this topic summarizes the important differences between developing code that uses Neon extensions and developing code that uses the SVE. The second and subsequent parts of this tutorial describe what to consider when preparing to migrate Neon code to SVE, and when migrating your Neon code, along with some examples.

2. Before you begin

References the resources to read before reading this tutorial.

If you are new to Arm® Neon® technology, read the Neon Programmer's Guide for Armv8-A for a general introduction to the subject.

If you are new to the Scalable Vector Extension (SVE), read our Introduction to SVE tutorial. This tutorial provides background information about SVE.

3. Part One - Neon and SVE fundamentals

Arm Neon technology is the Advanced SIMD (Single Instruction Multiple Data) feature for the Arm®v8-A architecture profile. Neon® is a feature of the Instruction Set Architecture (ISA), providing instructions that can perform mathematical operations in parallel on multiple data streams.

SVE is the next-generation SIMD extension of the Armv8-A instruction set. It is not an extension of Neon, but is a new set of vector instructions developed to target High Performance Computing (HPC) workloads. In short, SVE enables vectorization of loops which would be impossible, or not beneficial, to vectorize with Neon. Importantly, and unlike other SIMD architectures, SVE code can be Vector Length Agnostic (VLA): SVE does does not fix the size of the vector registers, instead SVE allows hardware implementors to choose the vector size that is best for their intended workloads.

Instruction sets

AArch64 is the name that is used to refer to the 64-bit execution state of the Armv8 architecture. In AArch64, the processor executes the A64 Instruction Set, which contains Neon instructions (also referred to as Advanced SIMD instructions). The SVE extension is introduced in version Armv8.2-A of the architecture, and adds a new subset of instructions to the existing Armv8-A A64 Instruction Set.

The following table compares the features that Neon and SVE instructions provide:

Extension	Key features
Neon	Provides instructions that can perform mathematical operations in parallel on multiple data streams. Support for double precision floating-point, enabling C code using double precision. For a full list of Neon instructions, see the Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile and for more information about the Neon instruction set, see the A64 Instruction set for Armv8-A.
SVE	SVE adds: * Support for variable-length vector and predicate registers (resulting in two main classes of instructions; predicated and unpredicated). * A set of instructions that operate on variable-length vectors. * Some minor additions to the configuration and identification registers.

For a full list of supported instruction categories, see the Arm Architecture Reference Manual Supplement, The Scalable Vector Extension.

Registers, vectors, lanes, and elements

Neon instructions operate on a separate register file of 128-bit registers and are fully integrated into Armv8-A processors. Neon uses a simple programming model.

The Neon register file is a collection of scalar registers. Scalar registers can be considered as vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit values called elements. The vectors are also divided into lanes, where each lane contains element data values.

All elements in a vector have the same data type.

The number of lanes in a Neon vector depends on the size of the vector and the data elements in the vector. That is, a 128-bit Neon vector can contain the following element layouts:

- Sixteen 8-bit elements
- Eight 16-bit elements
- Four 32-bit elements
- Two 64-bit elements

However, Neon instructions always operate on 64-bit or 128-bit vectors.

In SVE, the instruction set operates on a new set of vector and predicate registers: 32 Z registers, 16 P registers, and one First Faulting Register (FFR):

- The Z registers are data registers. Z register bits are an **IMPLEMENTATION DEFINED** multiple of 128, up to an architectural maximum of up to 2048-bits. Data in these registers can be interpreted as 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit elements. The low 128 bits of each Z register overlap the corresponding Neon registers, and therefore also the scalar floating-point registers.
- The P registers hold one bit for each byte available in a Z register. In other words, a P register is always 1/8th the size of a Z register. Predicated instructions use a P register to determine which vector elements to process. Each individual bit in the P register specifies whether the corresponding byte in the Z register is active or inactive.
- The FFR register is a dedicated predicate register that that captures the cumulative fault status of a sequence of SVE vector load instructions. SVE provides a first-fault option for some SVE vector load instructions. This option suppresses memory access faults if they do not occur as a result of the first active element of the vector. Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded.

Both P registers and the FFR register are unique to SVE.

VLA programming

SVE introduces the concept of VLA programming.

Unlike traditional SIMD architectures, which define a fixed size for their vector registers, SVE supports a variable size. This freedom of choice enables different Arm architectural licensees to develop their own implementation, targeting specific workloads and technologies, and trading-off between performance and cost. In short, hardware implementors can choose the vector size for their hardware.

A goal of SVE is to allow the same application image to be run on any SVE-enabled implementation of the architecture. To allow this, SVE includes instructions which permit vector code to adapt automatically to the implemented vector length at runtime.

Compiling an application without knowing the vectorized length means:

- Vectors cannot be initialized from compile time constant in memory
- Predicates cannot be initialized
- Vector loop increment and trip counts are unknown
- Vector register spill and fill must adjust to vector length

At runtime, the SVE instructions allow the length to be allocated. Coding with this approach is called VLA programming.

VLA programming allows you to compile your code for the generic SVE architecture and run it on any SVE-enabled hardware.

To learn more about VLA programming, see SVE Vector Length Agnostic (VLA) programming examples.

4. Part Two - Preparing to migrate your optimized Neon code to SVE

As a programmer, there are various ways you can use Neon and SVE technology.

Programming in any high-level language is a tradeoff between the ease of writing code, and the amount of control that you have over the low-level instructions that the compiler outputs.

Until now, you might have optimized your code for Neon® using auto-vectorization, intrinsics, math libraries, or by using hand-written Neon assembly. Each of these approaches are also available to you when you are writing SVE code:

- Compiler auto-vectorization: Auto-vectorization features in your compiler allow the compiler to automatically optimize SVE code.
- Intrinsics: Intrinsics are function calls that the compiler replaces with appropriate instructions. SVE intrinsics are available and give you direct access to the exact instructions you want.
- Libraries: Math libraries are a set of optimized math routines for a particular architecture or target. Math libraries are available in SVE variants, containing optimized SVE routines, respectively. For example Arm Performance Libraries.
- Assembly: To fine tune your code and have the highest possible control over performance, experienced programmers can code in SVE assembly. If you are comfortable coding in assembly, you can use the SVE specification to find out what instructions are available for SVE code, and use them to write your application assembly.

Next, this tutorial discusses each of these programming approaches in more detail.

Compiler auto-vectorization

Arm® Compiler for Linux and GCC compilers can automatically generate code containing Armv8 Neon or SVE instructions. The options you pass in your compile command will determine whether Neon or SVE instructions are used in the compiled code.

One approach that the compiler can use is auto-vectorization. Auto-vectorization allows the compiler to automatically identify opportunities in your code to use Neon or SVE instructions.

In terms of specific compilation techniques, auto-vectorization includes:

- Loop vectorization: unrolling loops to reduce the number of iterations, while performing more operations in each iteration.
- Superword-Level Parallelism (SLP) vectorization: bundling scalar operations together to make use of full width Advanced SIMD instructions.

The benefits of relying on compiler auto-vectorization are:

- Programs implemented in high-level languages are portable, so long as there are no architecture-specific code elements such as inline assembly or intrinsics.
- Modern compilers are capable of performing advanced optimizations automatically.

- Targeting a given micro-architecture can be as easy as setting a single compiler option.

To enable the compiler to auto-vectorize your code:

- Enable auto-vectorization using the compiler options (`-O<level>`, `-fvectorize`, as appropriate for your compiler)
- Structure your code to provide hints to the compiler, including using pragmas.
- Include the relevant Neon or SVE header files, and tell the compiler which processor you will run the code on (`-mpcu=<target>` compiler option).

Auto-vectorization compiler options

The following table shows the supported optimization levels for `-O<level>` for both Neon and SVE code:

Option	Description	Auto-vectorization
`-O0`	Minimum optimization for the performance of the compiled binary. Turns off most optimizations. When debugging is enabled, this option generates code that directly corresponds to the source code. Therefore, this might result in a significantly larger image. This is the default optimization level.	Never
`-O1`	Restricted optimization. When debugging is enabled, this option gives the best debug view for the trade-off between image size, performance, and debug.	Disabled by default.
`-O2`	High optimization. When debugging is enabled, the debug view might be less satisfactory because the mapping of object code to source code is not always clear. The compiler might perform optimizations that cannot be described by debug information.	Enabled by default.
`-O3`	Very high optimization. When debugging is enabled, this option typically gives a poor debug view. Arm recommends debugging at lower optimization levels.	Enabled by default.
`-Ofast`	Enable all the optimizations from level 3, including those performed with the -ffp-mode=fast armclang option. This level also performs other aggressive optimizations that might violate strict compliance with language standards.	Enabled by default.

Auto-vectorization is enabled when you use the `-O2`, `-O3`, or `-Ofast` optimization levels. The `-fno-vectorize` option lets you disable auto-vectorization.

Note

- At optimization level `-O0`, auto-vectorization is always disabled. If you specify the `-fvectorize` option, the compiler ignores it.
- At optimization level `-O1`, auto-vectorization is disabled by default. The `-fvectorize` option lets you enable auto-vectorization.

In addition to the optimization level options, there are also a number of other math and routine-related optimization options available in your compiler. For example, in Arm Compiler for Linux, you can consider using `-fassociative-math`, `-fno-signed-zeroes`, `-fno-trapping-math`, and `-ffast-math`. For information about these options, see the Arm C/C++ Compiler options or Arm Fortran Compiler options documentation.

Use the restrict keyword

If appropriate, use the `restrict` keyword when writing C code. The C99 `restrict` keyword (or the non-standard C/C++ `__restrict__` keyword) tells the compiler that a specified pointer does not alias with any other pointers for the lifetime of that pointer. Therefore, `restrict` allows the compiler to vectorize loops more aggressively because the compiler knows loop iterations are independent and can be executed in parallel.

Provide hints to the compiler

As for Neon code, you can structure your code to provide hints to the compiler. Well-structured application code, that has hints, enables the compiler to detect code behaviors that it would otherwise not be able to detect. The more behaviors the compiler detects, the better vectorized your output code is.

As an algorithm becomes more complicated, the likelihood that the compiler can auto-vectorize the code decreases. For example, loops with the following characteristics are particularly difficult, or impossible, to vectorize:

- Loops with interdependencies between different loop iterations
- Loops with break clauses
- Loops with complex conditions

Neon and SVE have different conditions for auto-vectorization. For example, a necessary condition for auto-vectorizing Neon code is that the number of iterations in the loop size must be known at the start of the loop, at compile time. However, knowing the number of iterations in the loop is not required to auto-vectorize SVE code.

Note

Break conditions mean the number of loops iterations might not be knowable at the start of the loop, which prevents auto-vectorization for Neon code. If it is not possible to completely avoid a break condition, it might be worthwhile breaking up loops into multiple vectorizable and non-vectorizable parts.

You can find a full discussion of the compiler directives used to control vectorization of loops in the LLVM-Clang documentation. The two most important directives are:

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

These pragmas are instructions to the compiler to control loop vectorization, and are also discussed in the Arm Compiler for Linux documentation:

- Arm C/C++ Compiler
- Arm Fortran Compiler

Header files and processor optimization

The following quick reference table details the different header files that should be used to compile Neon and SVE code, in addition to an example compile command line.

<table>
<tr><th>Extension</th><th>Header file form</th><th>Recommended compilation options</th><th>Notes</th></tr>
<tr><td>Neon</td><td><code>#include <arm_neon.h></code></td><td><code>armclang -O<2|3|fast> -mcpu={native|<target>} -o <binary_name> <filename>.c</code></td><td><code>-mcpu</code> enables the compiler to use micro-architectural optimizations, and can be set to a specific target (<code><target></code>), or you can allow the compiler to determine what processor it is running on (<code>native</code>).</td></tr>
<tr><td>SVE</td><td>#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */</td><td>To run on SVE-enabled hardware:
<code>armclang -O<2|3|fast> -mcpu={native|<target>} -o <binary_name> <filename>.c</code>

To produce SVE code for emulation on hardware that is not SVE-enabled: <code>armclang -O<2|3|fast> -march=armv8-a+sve -o <binary_name> <filename>.c</code></td><td>Like for Neon, <code>-mcpu</code> enables the compiler to use micro-architectural optimizations.
If the target is SVE-enabled, the compiler will produce optimized SVE code, rather than optimized Neon code.

As a less-optimal alternative to <code>-mcpu</code>, <code>-march=armv8-a+sve</code> tells the compiler to optimize for SVE-enabled Armv8-A hardware, but without the microarchitectural optimizations.

SVE code is produced and you can use Arm Instruction Emulator (ArmIE) to emulate the SVE instructions on any Armv8-A hardware.</td></tr>
</table>

Note

When SVE-enabled hardware is available and you are compiling on that target SVE hardware, Arm recommends using `-mcpu=native` so that micro-architectural optimizations can be taken advantage of.

To read more about the `-march` and `-mcpu` options, as well as `-mtune`, see the Compiler flags across architectures: -march, -mtune, and -mcpu blog.

Intrinsics

Intrinsics are functions whose precise implementation is known to a compiler. Intrinsics let you use Neon or SVE without having to write assembly code because the functions themselves contain short assembly kernels, which are inlined into the calling code. Also, register allocation and pipeline optimization are handled by the compiler. This avoids many of the difficulties often seen when developing assembly code.

Using intrinsics has several benefits:

- Powerful: Intrinsics give you direct access to the Neon and SVE instruction sets during development. You do not need to hand-write assembly code.
- Portable: You might need to rewrite hand-written Neon or SVE assembly instructions for different target processors. You can compile C and C++ code containing Neon intrinsics for a new AArch64 target, or a new Execution state, with minimal or no code changes. However, C and C++ code containing SVE intrinsics only runs on SVE-enabled hardware, unless under emulation on another Armv8-A target.
- Flexible: You can exploit Neon when needed, or use C/C++ when it is not. You do not need an in-depth knowledge of writing assembly.

However, intrinsics might not be the right choice in all situations:

- You need more learning to use intrinsics than you need to import a library, or to rely on a compiler.
- Hand-optimized assembly code might offer the greatest scope for performance improvement, even if it is more difficult to write.

Program conventions: macros, types, and functions

The Arm C Language Extensions (ACLE) enable C/C++ programmers to exploit the Arm architecture with minimal restrictions on source code portability. The ACLE includes a set of macros, types, and functions to make features of the Arm architecture directly available in C and C ++ programs. The key to applying SVE intrinsics is reading the SVE ACLE Specification.

This section of the guide provides an overview of these features.

For more detailed information, the Neon macros, types, and functions are described in the Arm C Language Extensions (ACLE). The SVE macros, types, and functions are described in the Arm C Language Extensions for SVE specification.

Macros

The feature test macros allow programmers to determine the availability of target architectural features. For example, to use the Neon or SVE intrinsics, the target platform must support the Advanced SIMD or SVE architecture. When a macro is defined and equal to 1, the corresponding feature is available.

Note

The lists in this section are not exhaustive. Other macros are described on the Arm C Language Extensions web page.

The following table lists some of the macros supported in Neon.

Macro	Description
`__aarch64__`	Selection of architecture-dependent source at compile time.
`__ARM_ACLE`	Defined as an integer value. Expands to the value that represents the ACLE version implementation.
`_ARM_NEON`	Defined as 1 if Advanced SIMD is supported by the compiler.
`_ARM_NEON_FP`	Defined as 1 if Neon floating-point operations are supported.
`_ARM_FEATURE_CRYPTO`	Defined as 1 if the Armv8-A Crypto instructions are supported and intrinsics targeting them are available.
`_ARM_FEATURE_FMA`	Defined as 1 if the hardware floating-point architecture supports fused floating-point multiply-accumulate.
`__ARM_FEATURE_COMPLEX`	Defined as an integer value. Expands to 1 if the system supports complex addition and complex multiply-accumulate vector instructions.
`__ARM_FEATURE_DOTPROD`	Defined as an integer value. Expands to 1 if the system: • Supports dot product data manipulation instructions • Has vector intrinsics available
`__FP_FAST_FMA`	Defined as an integer value. Expands to 1 if the supported `fma()` function evaluates faster than executing the expression `(x * y) + z`.

The following table lists some of the macros supported in SVE.

Macro	Description
`__ARM_FEATURE_SVE`	Defined as an integer value. Expands to 1 if SVE is supported and all the base SVE functions are available.
`__ARM_FEATURE_SVE_BF16`	Defined as an integer value. Expands to 1 if all the BFloat 16 extension function are available.
`__ARM_FEATURE_SVE_BITS`	Defined as an integer value. Expands to a non-zero value, `N`, if: • The implementation generates code for an SVE target • The `arm_sve_vector_bits(N)` attribute is available
`__ARM_FEATURE_SVE_MATMUL_FP32`	Defined as an integer value. Expands to 1 if all the FP32 matrix multiply extension functions are available.
`__ARM_FEATURE_SVE_BF16`	Defined as an integer value. Expands to 1 if all the BFloat 16 extension function are available.
`__ARM_FEATURE_SVE_MATMUL_FP64`	Defined as an integer value. Expands to 1 if all the FP64 matrix multiply extension functions are available.
`__ARM_FEATURE_SVE_MATMUL_INT8`	Defined as an integer value. Expands to 1 if all theINT8 matrix multiple extension functions are available.
`__ARM_FEATURE_SVE_NONMEMBER_OPERATORS`	Defined as an integer value. Expands to 1 if C++ code can define non-member operator functions for SVE types.
`__ARM_FEATURE_SVE_PREDICATE_OPERATORS`	Defined as an integer value. Expands to 1 if, when you apply the `arm_sve_vector_bits` attribute to `svbool_t`, the attribute creates a type that supports basic built-in vector operations.
`__ARM_FEATURE_SVE_VECTOR_OPERATORS`	Defined as an integer value. Expands to 1 if, when you apply the `arm_sve_vector_bits` attribute to an SVE vector type, the attribute creates a type that supports the GNU vector extensions.

A full list of the supported ACLE predefined macros, that includes more detailed descriptions of the macros and their dependencies, is available in:

- For Neon: Arm C Language Extensions (ACLE) specification.
- For SVE: ACLE for SVE specification.

Data types

The ACLE defines several data types that support SIMD processing. These data types are different for Neon and for SVE.

Data types - Neon

For Neon, there are three main categories of data type available in `arm_neon.h`. These data types are named according to the following patterns:

Data type	Description
`baseW_t`	Scalar data types. For example, `int64_t`.
`baseWxL_t`	Vector data types. For example, `int32x2_t`.
`baseWxLxN_t`	Vector array data types. For example, `int16x4x2_t`.

Where:

- `base` refers to the fundamental data type.
- `W` is the width of the fundamental type.
- `L` is the number of scalar data type instances in a vector data type, for example an array of scalars.
- `N` is the number of vector data type instances in a vector array type, for example a struct of arrays of scalars.

Generally, W and L are values where the vector data types are 64 bits or 128 bits long, and so fit completely into a Neon register. N corresponds with those instructions which operate on multiple registers at once.

Data types - SVE

For SVE, there is no existing mechanism that maps directly to the concept of an SVE vector or predicate. The ACLE classifies SVE vectors and predicates as belonging to a new category of type called sizeless data types. Sizeless data types are composed of vector types and predicate types, and have the prefix `sv`, for example `svint64_t`.

The following table shows the different data types that the ACLE defines:

Data type	Description
`svbaseW_t`	Sizeless vector data types for single vectors. For example, `svint64_t`.
`svbaseWxN_t`	Sizeless vector data types for two, three, and four vectors. For example, `svint64x2_t`.
`svbool_t`	Sizeless single predicate data type which has enough bits to control an operation on a vector of bytes.

Where:

- `base` refers to the fundamental data type.
- `bool` refers to the bool type from `stdbool.h`.
- `W` is the width of the fundamental type.
- `N` is the number of vector data type instances in a vector array type, for example a tuple of vector types.

Functions

Neon and SVE intrinsics are provided as function prototypes in the header files `arm_neon.h` and `arm_sve.h` respectively. These functions follow common naming patterns.

Functions - Neon

For Neon, the function prototypes from `arm_neon.h` follow a common pattern. This is similar to the naming pattern of the ACLE.

At the most general level, this is:

```
ret_type v[p][q][r]name[u][n][q][x][_high][_lane | laneq][_n][_result]_type(args)
```

For example:

```
int8x16_t vmulq_s8 (int8x16_t a, int8x16_t b)
```

The `mul` in the function name is a hint that this intrinsic uses the `MUL` instruction. The types of the arguments and the return value (sixteen bytes of signed integers) map to the following instruction:

```
MUL Vd.16B, Vn.16B, Vm.16B
```

This function multiplies corresponding elements of a and b and returns the result.

Some of the letters and names are overloaded, but the meaning of the elements in the order they appear in the naming pattern is as follows:

Pattern element	Description
`ret_type`	The return type of the function.
`v`	Short for `vector` and is present on all the intrinsics.
`p`	Indicates a pairwise operation. (`[value]` means `value` might be present).
`q`	Indicates a saturating operation (except for `vqtb[l][x]` in AArch64 operations, where the `q` indicates 128-bit index and result operands).
`r`	Indicates a rounding operation.
`name`	The descriptive name of the basic operation. Often, this is an Advanced SIMD instruction, but it does not have to be.
`u`	Indicates signed-to-unsigned saturation.
`n`	Indicates a narrowing operation.
`q`	Postfixing the name indicates an operation on 128-bit vectors.
`x`	Indicates an Advanced SIMD scalar operation in AArch64. It can be one of `b`, `h`, `s`, or `d` (that is, 8, 16, 32, or 64 bits).
`_high`	In AArch64, used for widening and narrowing operations involving 128-bit operands. For widening 128-bit operands, `high` refers to the top 64-bits of the source operand (or operands). For narrowing, it refers to the top 64-bits of the destination operand.
`_n`	Indicates a scalar operand that is supplied as an argument.
`_lane`	Indicates a scalar operand taken from the lane of a vector. `_laneq` indicates a scalar operand taken from the lane of an input vector of 128-bit width. (`left | right` means only `left` or `right` would appear).
`_result`	The result type, in short form.
`type`	The primary operand type in short form.
`args`	The arguments of the function.

For more information, see the ARM C Language Extensions Architecture specification.

Functions - SVE

For SVE, the function prototypes from `arm_sve.h` also follow a common pattern. At the most general level, this is:

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

For example, `svclz[_u16]_m` tells you that the full name is `svclz_u16_m` and that its overloaded alias is `svclz_m`.

The following table describes the different pattern elements:

Pattern element	Description
`base`	The lowercase name of an SVE instruction, with some adjustments.
`_disambiguator`	Distinguishes between different forms of a function.
`_type0|_type1|...`	List the types of vectors and predicates, starting with the return type and continuing with the argument types. In many cases, these are optional.
`_predication`	This suffix describes the inactive elements in the result of a predicated operation. It can be one of `z` (zero predication), `m` (merge predication), or `x` ('Do not care' predication).

For more information, see the ACLE for SVE specification.

Intrinsics resources

To learn more about optimizing your code for SVE with intrinsics, see the SVE(2) Programmers Guide.

The following intrinsics resources are also available:

- Searchable Neon intrinsics index
- Arm C Language Extensions (ACLE) engineering specification
- Arm C Language Extensions (ACLE) for SVE engineering specification

Libraries

Arm Performance Libraries provide optimized math libraries for both Neon and SVE-enabled processors.

When linking against Arm Performance Libraries, to use the SVE variant instead of the Neon variant, use the `-armpl` and `-mcpu=<target>` Arm Compiler for Linux options, (or `-larmpl[_lp64|_ilp64]{_mp}]` for GCC compilers).

Note

Arm Performance Libraries v21.0.0+ uses information passed by the compiler to determine whether the Neon or SVE variant of the libraries should be used. Therefore, you must also include `-mcpu=<target>` during your linking step.

For more information about Arm Performance Libraries library selection, see the library selection and the accessing the library topics.

Assembly

If you are familiar with writing Neon assembly, you can use the Arm C Language Extensions (ACLE) for SVE specification to find out about the instructions SVE provides, and the Procedure Call Standard (PCS) with SVE support to learn about SVE register assignment. Once you understand these documents, you can analyze your Neon assembly and update it to use SVE instructions, where possible.

Some useful tips to improve performance in SVE assembly:

- To avoid stalls in the pipelines, load registers as far in advance as possible.
- Retain values in a register as long as possible.
- Avoid using destination registers as source registers in the next instructions
- Avoid spilling registers to the stack.
- Ensure you are writing VLA-compatible assembly, for example, by avoiding hard-coded loop increments.
- Where a target has multiple SVE pipelines, unroll your loops to improve pipeline efficiency.
- Use prefetching instructions to preload data into caches, however be aware of the cache size and cache eviction.

An alternative to just writing assembly is to use inline assembly (or inline `asm`). Inline assembly is where assembly instructions are written into high-level C/C++ code to manually vectorize parts of a function, without having to write the entire function in assembly code.

Note

Writing inline assembly assumes that you are familiar with details of the SVE architecture, including vector-length agnostic registers, predication, and `WHILE` operations.

Using inline assembly instead of writing a separate `.s` file has the following advantages:

- Inline assembly code shifts the burden of handling the procedure call standard (PCS) from the programmer to the compiler. This includes allocating the stack frame and preserving all necessary callee-saved registers.
- Inline assembly code can give the compiler more information about what the assembly code does.
- The compiler can inline the function that contains the assembly code into that functions caller. callers.
- Inline assembly code can take immediate operands that depend on C-level constructs, such as the size of a structure or the byte offset of a particular structure field.

While coding in assembly allows you the greatest control over the tuning of your performance. Arm recommends that only experienced assembly programmers optimize their code using this approach because tuning a sequence of instructions to a particular pipeline of a processor is non-trivial, and is a task that a compiler will often complete more efficiently.

To learn more about optimizing your code for SVE with assembly, see the SVE(2) Programmers Guide.

5. Part Three - When it is sometimes useful to keep optimized Neon code

The tutorial so far has focused on the numerous benefits that migrating your code to SVE can bring. For completeness, this section discusses a couple of corner cases where it can remain an advantage to keep some Neon®-optimized code, instead of rewriting it in VLA for SVE: 1) sparse predication overhead, and 2) general VLA overhead.

1. Sparse predication overhead: Sometimes, the cost of using complete scalar Neon register can be lower than computing a partial vector using a predicated full-length SVE register. For example, where `inner` is <= 0.5*VL in:

   ```
   void foo (float * __restrict__ x, float* __restrict__ y, int outer, int inner)
   {
   for (int j = 0; j < outer; j++)
     for (int i = 0 ; i < inner; i++)
       x[i + (j * inner)] = y[i + (j*inner)] * y[i + (j * inner)];
   }
   ```

   there is a small overhead to using predication in SVE, compared to scalar or Neon instructions. If the predicate is mostly false, then the amount of work being done per vector operation is relatively small and unlikely to improve performance.

2. General VLA overhead: Sometimes, the overhead of VLA code might not give an advantage over highly optimized Neon code. For example, loop trip counts can be explicitly calculated with Neon, but they can not when they are written in VLA. Therefore, a compiler might fully unroll a Neon loop and produce code that is more register-efficient than had it been written as an SVE loop. For example, in:

   ```
   void foo (float * __restrict__ x, float* __restrict__ y)
   {
     for (unsigned i = 0; i < 8; i++)
         x[i] = y[i] * y[i];
   }
   ```

   the predication required needs more work in SVE, compared to in Neon alone.

6. Part Four - Migrate your Neon code to SVE

The steps to take to migrate your code are slightly different, and depend on whether your code is in a high-level language, like C/C++ or Fortran, or if your code is in assembly:

- To migrate to SVE VLA code in a high-level language like C/C++ or Fortran, you should:
    1. Update your compiler options (including auto-vectorization options).
    2. Where possible, and by referring to the SVE specification, replace or rewrite your code to use SVE intrinsics instead of Neon® intrinsics.
    3. Link your code with the SVE variants of the math libraries.
- If you are writing VLA assembly, you should:
    1. Update your compiler options (including auto-vectorization options).
    2. Where possible, and by referring to the SVE specification, rewrite your code to use SVE instructions instead of Neon instructions.
    3. Link your code with the SVE variants of the math libraries.

The following examples demonstrate how optimized Neon code (in C) can be rewritten using the ACLE SVE intrinsics to become optimized SVE code.

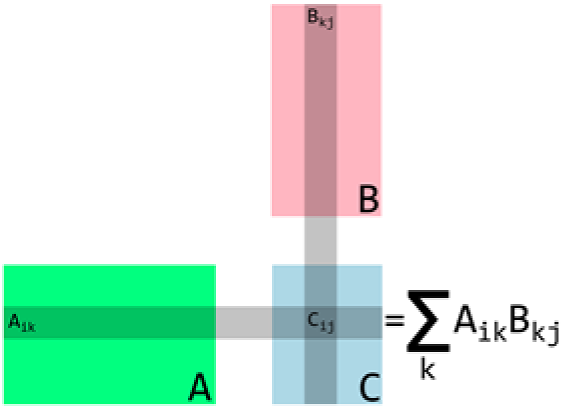
The examples do not cover re-writing Neon optimized assembly.

Example One - Rewriting a simple matrix multiplication code using intrinsics

This example implements some C functions using Neon intrinsics. The example chosen does not demonstrate the full complexity of the application, but illustrates the use of intrinsics, and is a starting point for more complex code. In this example, we rewrite the code to use SVE intrinsics.

Matrix multiplication is an operation performed in many data intensive applications and consists of groups of arithmetic operations which are repeated in a simple way:

Figure 6-1: Matrix multiplication diagram



The matrix multiplication process is as follows:

1. Take a row in the first matrix - 'A'
2. Perform a dot product of this row with a column from the second matrix - 'B'
3. Store the result in the corresponding row and column of a new matrix - 'C'

For matrices of 32-bit floats, the multiplication could be written as:

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

Assume a column-major layout of the matrices in memory. That is, an `n` x `m` matrix `M`, is represented as an array `M_array`, where `Mij` = `M_array[n*j + i]`.

This code is suboptimal, because it does not make full use of Neon. Intrinsics can be used to improve it.

The following code uses Neon intrinsics to multiply two 4x4 matrices. The loops can be completely unrolled because there is a small, fixed number of values to process, all of which can fit into the Neon registers of the processor at the same time.

```
void matrix_multiply_4x4_neon(const float32_t *A, const float32_t *B, float32_t *C)
 {
    // these are the columns A
    float32x4_t A0;
    float32x4_t A1;
    float32x4_t A2;
    float32x4_t A3;
```

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

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

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

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

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

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

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

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

Fixed-size 4x4 matrices are chosen because:

- Some applications need 4x4 matrices specifically, for example: graphics or relativistic physics.
- The Neon vector registers hold four 32-bit values. Matching the application to the architecture makes it easier to optimize.
- This 4x4 kernel can be used in a more general kernel. | The Neon intrinsics that are used in this example are:

Code element	What is it?	Why are they used?
`float32x4_t`	An array of four 32-bit floats.	One `uint32x4_t` fits into a 128-bit register and ensures that there are no wasted register bits, even in C code.
`vld1q_f32(…)`	A function which loads four 32-bit floats into `float32x4_t`.	To get the matrix values needed from A and B.
`vfmaq_lane_f32(…)`	A function which uses the fused multiply accumulate instruction. Multiplies a `float32x4_t` value by a single element of another `float32x4_t` then adds the result to a third `float32x4_t` before returning the result.	Since the matrix row-on-column dot products are a set of multiplications and additions, this operation fits naturally.
`vst1q_f32(…)`	A function which stores `float32x4_t` at a given address.	To store the results after they are calculated.

Rewriting the code to use SVE intrinsics instead of Neon intrinsics, could give you:

```
void matrix_multiply_nx4_sve(const float32_t *A, const float32_t *B, float32_t *C,
 uint32_t n) {
    // these are the columns A
    svfloat32_t A0;
    svfloat32_t A1;
    svfloat32_t A2;
    svfloat32_t A3;

    // these are the columns B
    svfloat32_t B0;
    svfloat32_t B1;
    svfloat32_t B2;
    svfloat32_t B3;

    // these are the columns C
    svfloat32_t C0;
    svfloat32_t C1;
    svfloat32_t C2;
    svfloat32_t C3;

    svbool_t pred = svwhilelt_b32_u32(0, n);
    A0 = svld1_f32(pred, A);
    A1 = svld1_f32(pred, A+n);
    A2 = svld1_f32(pred, A+2*n);
    A3 = svld1_f32(pred, A+3*n);

    // Zero accumulators for C values
    C0 = svdup_n_f32(0);
    C1 = svdup_n_f32(0);
    C2 = svdup_n_f32(0);
    C3 = svdup_n_f32(0);

    // Multiply accumulate in 4x1 blocks, that is each column in C
    B0 = svld1rq_f32(svptrue_b32(), B);
    C0 = svmla_lane_f32(C0, A0, B0, 0);
    C0 = svmla_lane_f32(C0, A1, B0, 1);
    C0 = svmla_lane_f32(C0, A2, B0, 2);
    C0 = svmla_lane_f32(C0, A3, B0, 3);
    svst1_f32(pred, C, C0);

    B1 = svld1rq_f32(svptrue_b32(), B+4);
    C1 = svmla_lane_f32(C1, A0, B1, 0);
    C1 = svmla_lane_f32(C1, A1, B1, 1);
    C1 = svmla_lane_f32(C1, A2, B1, 2);
    C1 = svmla_lane_f32(C1, A3, B1, 3);
    svst1_f32(pred, C+4, C1);

    B2 = svld1rq_f32(svptrue_b32(), B+8);
    C2 = svmla_lane_f32(C2, A0, B2, 0);
```

```
    C2 = svmla_lane_f32(C2, A1, B2, 1);
    C2 = svmla_lane_f32(C2, A2, B2, 2);
    C2 = svmla_lane_f32(C2, A3, B2, 3);
    svst1_f32(pred, C+8, C2);

    B3 = svld1rq_f32(svptrue_b32(), B+12);
    C3 = svmla_lane_f32(C3, A0, B3, 0);
    C3 = svmla_lane_f32(C3, A1, B3, 1);
    C3 = svmla_lane_f32(C3, A2, B3, 2);
    C3 = svmla_lane_f32(C3, A3, B3, 3);
    svst1_f32(pred, C+12, C3);
}
```

The SVE intrinsics that are used in this example are:

Code element	What is it?	Why are they used?
`svfloat32_t`	An array of 32-bit floats, where the exact number is defined at runtime based on the SVE vector length.	`svfloat32_t` enables you to use SVE vectors and predicates directly, without relying on the compiler for auto-vectorization.
`svwhilelt_b32_u32(…)`	A function which computes a predicate from two `uint32_t` integers.	When loading from A and storing to C, `svwhilelt_b32_u32(...)` ensures you do not read or write past the end of each column.
`svld1_f32(…)`	A function which loads 32-bit `svfloat32_t` floats into an SVE vector.	To get the matrix values needed from A. This also takes a predicate to make sure we do not load off the end of the matrix (unpredicated elements are set to zero).
`svptrue_b32(…)`	A function which sets a predicate for 32-bit values to all-true.	When loading from B, `svptrue_b32(...)` ensures the vector fills completely because the precondition of calling this function is that the matrix has a dimension which is a multiple of four.
`svld1rq_f32(…)`	A function which loads an SVE vector with copies of the same 128-bits (four 32-bit values).	To get the matrix values needed from B. Only loads four replicated values because the `svmla_lane_f32` instruction only indexes in 128-bit segments.
`svmla_lane_f32(…)`	A function which uses the fused multiply accumulate instruction. The function multiplies each 128-bit segment of an `svfloat32_t` value by the corresponding single element of each 128-bit segment of another `svfloat32_t`. The `svmla_lane_f32(…)` function then adds the result to a third `svfloat32_t` before returning the result.	single element of each 128-bit segment of another svfloat32_t. The svmla_lane_f32(…) function then adds the result to a third svfloat32_t before returning the result. This operation naturally fits the row-on-column dot products because they are a set of multiplications and additions.
`svst1_f32(…)`	A function which stores `svfloat32_t` at a given address.	To store the results after they are calculated. The predicate ensures we do not store results past the end of each column.

The important difference is the ability to ignore one of the dimensions of the matrix because of the variable-length vectors that are available in SVE. Instead, you can explicitly pass the length of the n dimension, and use predication to ensure it is not exceeded.

Example Two - Rewriting a larger matrix multiplication code with intrinsics

To multiply larger matrices, treat them as blocks of 4x4 matrices. However, this approach only works with matrix sizes which are a multiple of four in both dimensions. To use this method without changing it, pad the matrix with zeroes.

The Neon code for a more general matrix multiplication is listed below. The structure of the kernel has changed with the addition of loops and address calculations being the major changes. Like in the 4x4 kernel, unique variable names are used for the B columns. The alternative would be to use one variable and re-load it. This acts as a hint to the compiler to assign different registers to these variables. Assigning different registers enables the processor to complete the arithmetic instructions for one column, while waiting on the loads for another.

```
void matrix_multiply_neon(const float32_t *A, const float32_t *B, float32_t *C,
 uint32_t n, uint32_t m, uint32_t k) {
    /*
     * Multiply matrices A and B, store the result in C.
     * It is the users responsibility to make sure the matrices are compatible.
     */

    int a_idx;
    int b_idx;
    int c_idx;

    // these are the columns of a 4x4 sub matrix of A
    float32x4_t A0;
    float32x4_t A1;
    float32x4_t A2;
    float32x4_t A3;

    // these are the columns of a 4x4 sub matrix of B
    float32x4_t B0;
    float32x4_t B1;
    float32x4_t B2;
    float32x4_t B3;

    // these are the columns of a 4x4 sub matrix of C
    float32x4_t C0;
    float32x4_t C1;
    float32x4_t C2;
    float32x4_t C3;

    for (int i_idx=0; i_idx<n; i_idx+=4) {
        for (int j_idx=0; j_idx<m; j_idx+=4) {
            // zero accumulators before matrix op
            C0 = vmovq_n_f32(0);
            C1 = vmovq_n_f32(0);
            C2 = vmovq_n_f32(0);
            C3 = vmovq_n_f32(0);
            for (int k_idx=0; k_idx<k; k_idx+=4){
                // compute base index to 4x4 block
                a_idx = i_idx + n*k_idx;
                b_idx = k*j_idx + k_idx;

                // load most current a values in row
                A0 = vld1q_f32(A+a_idx);
                A1 = vld1q_f32(A+a_idx+n);
                A2 = vld1q_f32(A+a_idx+2*n);
                A3 = vld1q_f32(A+a_idx+3*n);

                // multiply accumulate 4x1 blocks, that is each column C
                B0 = vld1q_f32(B+b_idx);
                C0 = vfmaq_laneq_f32(C0,A0,B0,0);
                C0 = vfmaq_laneq_f32(C0,A1,B0,1);
                C0 = vfmaq_laneq_f32(C0,A2,B0,2);
                C0 = vfmaq_laneq_f32(C0,A3,B0,3);

                B1 = vld1q_f32(B+b_idx+k);
                C1 = vfmaq_laneq_f32(C1,A0,B1,0);
                C1 = vfmaq_laneq_f32(C1,A1,B1,1);
                C1 = vfmaq_laneq_f32(C1,A2,B1,2);
                C1 = vfmaq_laneq_f32(C1,A3,B1,3);
```

```
                B2 = vld1q_f32(B+b_idx+2*k);
                C2 = vfmaq_laneq_f32(C2,A0,B2,0);
                C2 = vfmaq_laneq_f32(C2,A1,B2,1);
                C2 = vfmaq_laneq_f32(C2,A2,B2,2);
                C2 = vfmaq_laneq_f32(C2,A3,B2,3);

                B3 = vld1q_f32(B+b_idx+3*k);
                C3 = vfmaq_laneq_f32(C3,A0,B3,0);
                C3 = vfmaq_laneq_f32(C3,A1,B3,1);
                C3 = vfmaq_laneq_f32(C3,A2,B3,2);
                C3 = vfmaq_laneq_f32(C3,A3,B3,3);
            }
            // compute base index for stores
            c_idx = n*j_idx + i_idx;
            vst1q_f32(C+c_idx, C0);
            vst1q_f32(C+c_idx+n,C1);
            vst1q_f32(C+c_idx+2*n,C2);
            vst1q_f32(C+c_idx+3*n,C3);
        }
    }
}
```

Compiling and disassembling this function, and comparing it with the C function shows:

- Fewer arithmetic instructions for a given matrix multiplication, because it utilizes the Advanced SIMD technology with full register packing. Typical C code, generally, does not.
- `FMLA` instead of `FMUL` instructions. As specified by the intrinsics.
- Fewer loop iterations. When used properly intrinsics allow loops to be unrolled easily.

However, there are unnecessary loads and stores because of memory allocation and initialization of data types (for example, `float32x4_t`) which are not used in the no-intrinsics C code.

Re-writing the code to use SVE intrinsics instead of Neon intrinsics, could give you:

```
void matrix_multiply_sve(const float32_t *A, const float32_t *B, float32_t *C,
 uint32_t n, uint32_t m, uint32_t k) {
    /*
     * Multiply matrices A and B, store the result in C.
     * It is the users responsibility to make sure the matrices are compatible.
     */

    int a_idx;
    int b_idx;
    int c_idx;

    // these are the columns of a nx4 sub matrix of A
    svfloat32_t A0;
    svfloat32_t A1;
    svfloat32_t A2;
    svfloat32_t A3;

    // these are the columns of a 4x4 sub matrix of B
    svfloat32_t B0;
    svfloat32_t B1;
    svfloat32_t B2;
    svfloat32_t B3;

    // these are the columns of a nx4 sub matrix of C
    svfloat32_t C0;
    svfloat32_t C1;
    svfloat32_t C2;
    svfloat32_t C3;

    for (int i_idx=0; i_idx<n; i_idx+=svcntw()) {
```

```
        // calculate predicate for this i_idx
        svbool_t pred = svwhilelt_b32_u32(i_idx, n);

        for (int j_idx=0; j_idx<m; j_idx+=4) {
            // zero accumulators before matrix op
            C0 = svdup_n_f32(0);
            C1 = svdup_n_f32(0);
            C2 = svdup_n_f32(0);
            C3 = svdup_n_f32(0);
            for (int k_idx=0; k_idx<k; k_idx+=4){
                // compute base index to 4x4 block
                a_idx = i_idx + n*k_idx;
                b_idx = k*j_idx + k_idx;

                // load most current a values in row
                A0 = svld1_f32(pred, A+a_idx);
                A1 = svld1_f32(pred, A+a_idx+n);
                A2 = svld1_f32(pred, A+a_idx+2*n);
                A3 = svld1_f32(pred, A+a_idx+3*n);

                // multiply accumulate 4x1 blocks, that is each column C
                B0 = svld1rq_f32(svptrue_b32(), B+b_idx);
                C0 = svmla_lane_f32(C0,A0,B0,0);
                C0 = svmla_lane_f32(C0,A1,B0,1);
                C0 = svmla_lane_f32(C0,A2,B0,2);
                C0 = svmla_lane_f32(C0,A3,B0,3);

                B1 = svld1rq_f32(svptrue_b32(), B+b_idx+k);
                C1 = svmla_lane_f32(C1,A0,B1,0);
                C1 = svmla_lane_f32(C1,A1,B1,1);
                C1 = svmla_lane_f32(C1,A2,B1,2);
                C1 = svmla_lane_f32(C1,A3,B1,3);

                B2 = svld1rq_f32(svptrue_b32(), B+b_idx+2*k);
                C2 = svmla_lane_f32(C2,A0,B2,0);
                C2 = svmla_lane_f32(C2,A1,B2,1);
                C2 = svmla_lane_f32(C2,A2,B2,2);
                C2 = svmla_lane_f32(C2,A3,B2,3);

                B3 = svld1rq_f32(svptrue_b32(), B+b_idx+3*k);
                C3 = svmla_lane_f32(C3,A0,B3,0);
                C3 = svmla_lane_f32(C3,A1,B3,1);
                C3 = svmla_lane_f32(C3,A2,B3,2);
                C3 = svmla_lane_f32(C3,A3,B3,3);
            }
            // compute base index for stores
            c_idx = n*j_idx + i_idx;
            svst1_f32(pred, C+c_idx, C0);
            svst1_f32(pred, C+c_idx+n,C1);
            svst1_f32(pred, C+c_idx+2*n,C2);
            svst1_f32(pred, C+c_idx+3*n,C3);
        }
    }
}
```

This code is almost identical to the earlier Neon code except for the differing intrinsics, and in addition, thanks to predication, there is no longer a constraint on the number of rows of A. However, you must ensure that the number of columns of A and C, and both dimensions of B, are multiples of four because the predication used above does not account for this. Adding such further predication is possible but would reduce the clarity of this example.

Comparing it with the C function and Neon functions, the SVE example:

- Uses `WHILELT` to determine the predicate for doing each iteration of the outer loop. This guarantees you have at least one element to do by the loop condition.

- Increments `i_idx` by `CNTW` (the number of 32-bit elements in a vector) to avoid hard-coding the number of elements calculated in an iteration of the outer loop.

7. Check your knowledge

Read the following questions to check your knowledge.

- Which header files define the Neon intrinsics and the SVE intrinsics?

`arm_neon.h` (Neon®) and `arm_sve.h` (SVE)

- What size are the Neon and SVE vector data registers?

Neon registers are 128-bit registers. SVE does not define a fixed size for its vector registers.

SVE vector registers are an **IMPLEMENTATION DEFINED** multiple of 128 bits, up to an architectural maximum of up to 2048 bits.

- What term describes allowing the compiler to automatically identify opportunities in your code to use Neon or SVE instructions?

Auto-vectorization

8. Related information

Lists useful reference information to use when migrating your Neon® code to SVE.

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

- Arm Neon technology
- Engineering specifications for the Neon intrinsics can be found in the Arm C Language Extensions (ACLE).
- Neon Programmer's Guide for Armv8-A
- The Architecture Exploration Tools let you investigate the Advanced SIMD instruction set.
- The Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile provides a complete specification of the Advanced SIMD instruction set.
- The Neon Intrinsics Reference provides a searchable reference of the functions specified by the ACLE.

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

- Arm Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for Armv8-A
- Arm Instruction Emulator
- Engineering specifications for the SVE intrinsics can be found in the Arm C Language Extensions for SVE specification.
- SVE Vector Length Agnostic programming
- The Arm HPC tools for SVE web page describes the tools to enable you to work with SVE code on AArch64.
- What is the Scalable Vector Extension?

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

- Arm C/C++ Compiler Reference
- Arm Fortran Compiler Reference
- Arm Performance Libraries
- Compiler flags across architectures: -march, -mtune, and -mcpu blog
- LLVM-Clang documentation