

SVE Programming Examples



Release information

Date	Version	Changes
2024/May/31	REL-01.2	<ul style="list-style-type: none">Fixed error in section A6.1 code example.
2021/Feb/01	REL-01.1	<ul style="list-style-type: none">Removed referencies to fixed-point arithmetic and data.
2020/Jul/24	REL-01	<ul style="list-style-type: none">First release.

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

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

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

Copyright © 2019-2024 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Contents

SVE Programming Examples

Release information	ii
Non-Confidential Proprietary Notice	iii

Preface

About this book	ix
Using this book	x
Conventions	xi
Typographical conventions	xi
Numbers	xi
Pseudocode descriptions	xi
Assembler syntax descriptions	xi
Rules-based writing	xii
Identifiers	xii
Examples	xii
Additional reading	xiii
Feedback	xiv
Feedback on this book	xiv

Part A Arm Scalable Vector Extension (SVE) overview

Chapter A1

Introduction

A1.1 Overview	17
A1.2 Register file	18
A1.2.1 Vector register file	18
A1.2.2 Predicate register file	18
A1.2.3 First Fault Register	19
A1.3 Predicate Condition Flags	20
A1.3.1 Overview	20
A1.3.2 AArch64 Condition Codes and Flags	20
A1.3.3 SVE Condition Codes and Flags	20

Chapter A2

Compiler support

A2.1 Autovectorization	23
A2.1.1 Compiler options and pragmas	23
A2.1.2 Generating the Autovectorization diagnostics	23
A2.2 Calling conventions	24

Chapter A3

SVE Instruction Set

A3.1 Overview	26
A3.1.1 Constructive and destructive instructions	26
A3.1.2 Predication of constructive and destructive instructions	26
A3.1.3 Move prefix	27
A3.2 Element size and type conversion instructions	28
A3.2.1 Element size	28
A3.2.2 Element size conversions	28
A3.2.3 Type conversions	28
A3.3 Loads and Stores	29
A3.3.1 Contiguous loads and stores	29

A3.3.2	Loads with broadcast	29
A3.3.3	Gather loads and Scatter stores	29
A3.3.4	Unpredicated load and store	29
A3.3.5	Non-temporal continuous loads and stores	30
A3.3.6	First faulting loads	30
A3.3.7	Non-faulting loads	30
A3.3.8	Prefetching	30
A3.3.9	Memory ordering	30
A3.4	Control flow	31
A3.4.1	Predicate initialization	31
A3.4.2	Index initialization	31
A3.4.3	Predicate count	31
A3.4.4	Conditional loop break	31
A3.5	Dot product	32
A3.6	Complex floating-point operations	33
A3.6.1	Complex-valued addition	33
A3.6.2	Complex-valued multiply-accumulate	33
A3.7	Vector by indexed element instructions	34
A3.8	Other instructions	35
A3.8.1	Reduction instructions	35
A3.8.2	Permutation instructions	35
A3.8.3	Instructions with elements crossing the 128-bits vector granule	35
A3.8.4	Instructions moving data between general-purpose register file and scalable vector register file	35
Chapter A4	SVE Vector Length Agnostic programming approach	
A4.1	For/While loop vectorization	37
A4.1.1	For/While loop Advanced SIMD vectorization	37
A4.1.2	For/While loop SVE vectorization	38
A4.1.3	For/While loop SVE vectorization with prefetching	39
A4.2	Do-while loop SVE vectorization	40
A4.3	Effective vector length bandwidth utilization tips	42
Chapter A5	SVE2 Instruction set innovations	
A5.1	Widening and narrowing instructions	45
A5.1.1	Top-Bottom processing approach	45
A5.1.2	Top-Bottom widening instructions	45
A5.1.3	Top-Bottom narrowing instruction	45
A5.2	Complex-valued integer operations	46
A5.2.1	Complex-valued addition	46
A5.2.2	Complex-valued multiply-accumulate	46
A5.2.3	Complex-valued integer dot product	46
A5.3	Other instructions	48
A5.3.1	Element size conversions	48
A5.3.2	Element pairwise instructions	48
A5.3.3	Bitwise permute instructions	48
A5.3.4	Histogram acceleration instructions	48
A5.3.5	String processing acceleration instructions	48
A5.3.6	Multiprecision arithmetic support instructions	48
A5.3.7	Cryptography support instructions	49
Chapter A6	SVE2 vectorization examples	
A6.1	Effective vector length bandwidth utilization tips	51
A6.2	Vectorization with termination by the reduction	52
A6.3	Loop unrolling example	53

A6.4	Two-level nested loop and loop interchange vectorization	55
A6.5	Memory ordering example	58

Part B Generic vector and matrix operations examples

Chapter B1	Vector maximum element	
B1.1	Code example: vector maximum with real 16-bit integer elements	61
Chapter B2	Vector multiply	
B2.1	Code example: vector multiply with complex 16-bit integer elements	64
Chapter B3	Vector dot-product	
B3.1	Vector dot-product calculation	68
B3.2	Code example: vectors dot-product with complex SP floating-point elements and result	69
B3.3	Code example: vectors dot-product with real HP floating-point input elements and real SP floating-point result	71
B3.4	Code example: vectors dot-product with complex 16-bit integer elements and result	73
Chapter B4	FIR filter	
B4.1	FIR filtering theory and C implementation	77
B4.2	Code example: FIR filtering with real SP floating-point elements	78
B4.3	Code example: FIR filtering with real-valued 16-bit integer elements	79
Chapter B5	Matrix multiply	
B5.1	Matrix multiplication with real DP floating-point elements	82
B5.1.1	Code example	82
B5.2	Matrix multiplication with real HP floating-point elements	85
B5.2.1	Code example	86
B5.3	Matrix multiplication with real 8-bit integer input elements and real 32-bit integer output elements	90
B5.3.1	Code example	91

Part C Computer vision operations examples

Chapter C1	Sobel filter	
C1.1	Sobel 3x3 2-dimensional filtering C implementation	99
C1.2	Code example: Sobel 3x3 2-dimensional filtering with real SP floating-point elements	100
Chapter C2	Integral image	
C2.1	Integral image C implementation	103
C2.2	Code example: Integral image with real SP floating-point elements	104
Chapter C3	Histogram	
C3.1	Histogram C implementation	109
C3.2	Code example: 8-bit pixels Image Histogram computation	110

Part D String processing examples

Chapter D1	Skip white space	
D1.1	Code example: skip over white spaces	116

Chapter D2	Skip word	
	D2.1 Code example: skip over a word	118
Chapter D3	String compare	
	D3.1 Code example: string compare	120
Glossary		

Preface

About this book

This document illustrates Arm Scalable Vector Extension and provides programming guidance and the selection of code examples.

This document has the following parts:

Part A

Introduction to the SVE architecture [1] (the Scalable Vector Extension for Armv8-A [2]), the SVE2 architecture (the Scalable Vector Extension v2), and effective programming advices.

Part B

Provides code examples for the selection of generic vector and matrix operations.

Part C

Provides code examples for the selection of computer vision operations.

Part D

Provides code examples for the selection of string processing operations.

Using this book

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example <http://developer.arm.com>

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by 0x. In both cases, the prefix and the associated value are written in a monospace font, for example 0xFFFF0000. To improve readability, long numbers can be written with an underscore separator between every four characters, for example 0xFFFF_0000_0000_0000. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font.

Rules-based writing

This specification consists of a set of individual rules. Each rule is clearly identified by the letter R.

Rules must not be read in isolation, and where more than one rule relating to a particular feature exists, individual rules are grouped into sections and subsections to provide the proper context. Where appropriate, these sections contain a short introduction to aid the reader. An implementation which is compliant with the architecture must conform to all of the rules in this specification.

Some architecture rules are accompanied by rationale statements which explain why the architecture was specified as it was. Rationale statements are identified by the letter X.

Some sections contain additional information and guidance that do not constitute rules. This information and guidance is provided purely as an aid to understanding the architecture. Information statements are clearly identified by the letter I.

Implementation notes are identified by the letter U.

Software usage descriptions are identified by the letter S.

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

Rules, rationale statements, information statements, implementation notes and software usage statements are collectively referred to as *content items*.

Identifiers

Each content item may have an associated identifier which is unique within the context of this specification.

When the document is prior to beta status:

- Content items are assigned numerical identifiers, in ascending order through the document (*0001, 0002, ...*).
- Identifiers are volatile: the identifier for a given content item may change between versions of the document.

After the document reaches beta status:

- Content items are assigned random alphabetical identifiers (*HJQS, PZWL, ...*).
- Identifiers are preserved: a given content item has the same identifier across versions of the document.

Examples

Below are examples showing the appearance of each type of content item.

R	This is a rule statement.
R _{x001}	This is a rule statement.
I	This is an information statement.
X	This is a rationale statement.
U	This is an implementation note.
S	This is a software usage description.

Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer (<http://developer.arm.com>) for access to Arm documentation.

- [1] *The Scalable Vector Extension (SVE), Arm Architecture Reference Manual Supplement for Armv8-A*. Arm.
- [2] *Arm Architecture Reference Manual Armv8-A edition*. Arm.
- [3] *Arm HPC tools and libraries* (<https://developer.arm.com/products/software-development-tools/hpc>). Arm.
- [4] *Procedure Call Standard for the Arm 64-bit Architecture (AArch64) with SVE support*. Arm.
- [5] *Arm C Language Extensions for SVE*. Arm.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title (SVE Programming Examples).
- The number (ARM-DAI-0548 REL-01.2).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Part A

Arm Scalable Vector Extension (SVE) overview

Chapter A1

Introduction

This chapter introduces the Scalable Vector Extension (SVE) to the Armv8-A architecture.

A1.1 Overview

The Armv8-A Advanced SIMD instruction set operates on fixed length vectors of 64-bit or 128-bit. For industry applications where large amounts of data are processed, for example in computer vision and machine learning, servers, data analytics, it's been recognized that processors operating on longer vector lengths might be a better fit. However, there is no specific vector length that is well suited to all applications.

Due to the relatively high encoding cost of specifying a new instruction set each time a need for a new vector length arises, Arm adopted a novel approach and created a unique instruction set that scales on different vector lengths. This next generation Arm SIMD instruction set is called Scalable Vector Extension (SVE). It permits vector length agnostic coding style where the code does not need to be re-written or re-compiled since it dynamically adapts to the implemented vector length. The SVE architecture allows implementations with a vector length up to 2048-bits, where vector length must be a multiple of 128-bits. SVE also supports code written for a fixed vector length.

The vectorization approach facilitated by the Armv8-A Advanced SIMD instruction set is sensitive to situations where the loop count is not a multiple of the vector length. To accommodate this, an Armv8-A Advanced SIMD vectorized loop is followed by a scalar loop that processes the remaining data. The vector partitioning mechanism supported by SVE allows control over which elements of a vector are operated on. This enables easier handling of loop termination conditions. The suite of carefully selected instructions converts conditional statements or control flow into predicated vector operations enabling seamless loop control. Consequently, most complex nested loops become vectorizable.

SVE supports reduction instructions for a number of arithmetic operations. These instructions enable such vectorizations where partial results are calculated in distinct vector lanes. The final result is obtained from these partial results with the help of a reduction instruction after the completion of the vectorized loop.

The SVE instruction set provides extensive load/store instruction support, predicate and loop control support, and logical and bitwise instructions support. The SVE instruction set also provides thorough floating-point arithmetic support, and basic integer arithmetic support.

SVE2, the Scalable Vector Extension v2, is a superset of the Armv8-A SVE with expanded functionality. The SVE2 instruction set adds thorough integer arithmetic support.

A1.2 Register file

The SVE register file consists of vector registers and predicate registers.

A1.2.1 Vector register file

There are 32 vector registers, Z0-Z31. The size of a vector register, VL, is an IMPLEMENTATION DEFINED multiple of 128 bits, i.e. $n \times 128$ -bits. The lower 128 bits of the SVE vector registers overlap with the SIMD and floating point registers, as shown in Figure A1.1. This makes it possible to interchangeably use SVE instructions with the Advanced SIMD instructions, and a particular register can be written to by the SVE instruction and then read by the Advanced SIMD instruction, and vice versa. When an Advanced SIMD or a floating-point instruction writes to the SIMD and floating point registers, bits[VL-1 : 128] of the corresponding SVE vector register are zeroed.

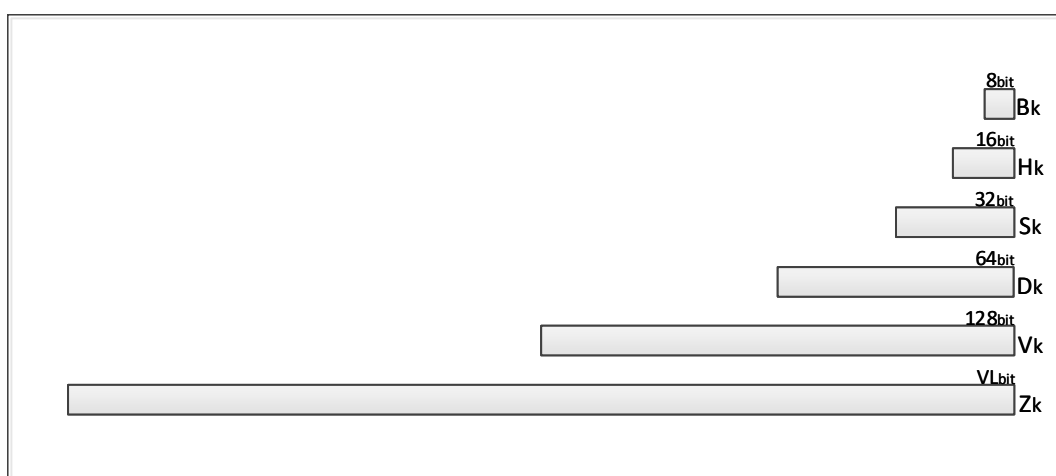


Figure A1.1: SIMD/Floating-point registers and SVE registers

A1.2.1.1 Vector register elements

The vector register contains data considered as elements, and a vector element can be 8-, 16-, 32- or 64-bits long.

A1.2.2 Predicate register file

The SVE introduces predicates that determine whether the corresponding vector element (SIMD) lane of the vector register it is used with is active or not.

There are 16 predicate registers: P0-P15. The size of a predicate register is $\frac{1}{8}$ of the vector register size, i.e. $n \times 16$ -bits. Each bit of the predicate register corresponds to one byte of the vector register.

A1.2.2.1 Predicate register elements

A predicate element can be 1-, 2-, 4- or 8-bits long.

Even though the predicate element can be up to 8-bits long, only the least significant bit of a predicate element determines whether the corresponding element is active (bit[0]=1) or not active (bit[0]=0). The higher bits of a predicate element are ignored.

A carefully initialised predicate can be shared between instructions that operate on different element sizes. For example, if a predicate register is initialised such that all bits of its active elements are set, and all bits of its inactive

elements are cleared, then the same predicate can be used for instructions acting on the individual bytes in the active elements, and instructions acting directly on the active elements. This is illustrated in [Figure A1.2](#).

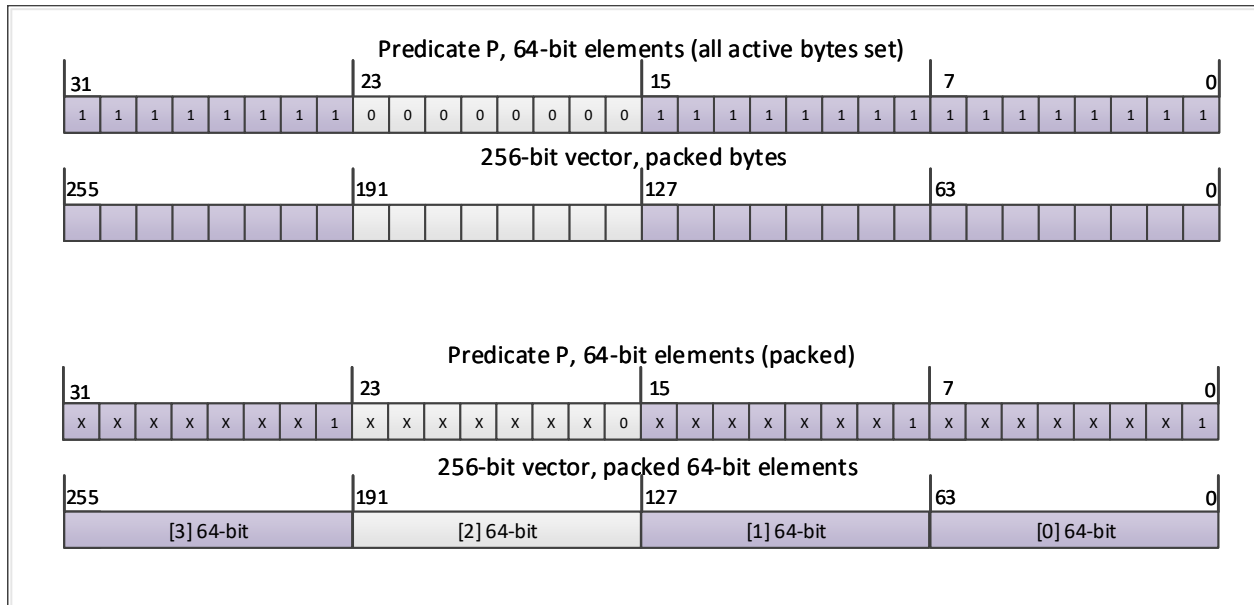


Figure A1.2: Predicate reuse example for packed elements

Alternatively, a predicate initialised for elements packed in a vector register is applicable to instructions operating on a smaller size elements that are organized unpacked in a vector register, as shown in [Figure A1.3](#).

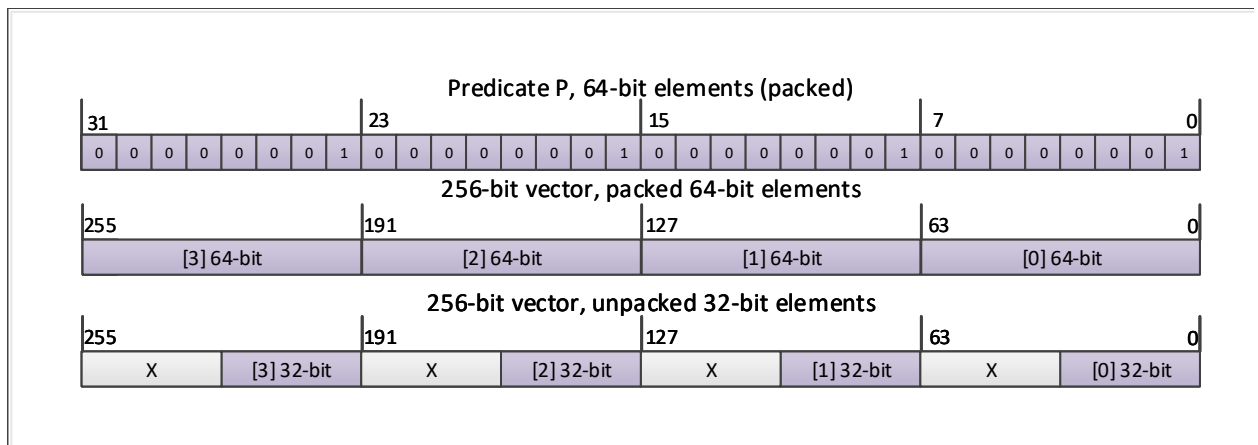


Figure A1.3: Predicate reuse example for unpacked elements

A1.2.3 First Fault Register

The first fault register (FFR) is a dedicated predicate-like register. It captures the cumulative fault status of a sequence of first-fault vector load instructions described in [A3.3.6 First faulting loads](#) and non-fault vector load instructions described in [A3.3.7 Non-faulting loads](#). Upon the completion of these load instructions, the active elements in FFR indicate a set of consecutive vector elements that contain valid memory data.

A1.3 Predicate Condition Flags

A1.3.1 Overview

The state of predicates is used to manage the control flow of a vectorized loop.

The SVE architecture does not specify new conditional branch instructions. Instead, the existing AArch64 condition flags and conditional branch instructions are overloaded with functionality capturing the state of predicates.

A1.3.2 AArch64 Condition Codes and Flags

The behavior of conditional branches is determined by the conditional code and the state of AArch64 flags. The four AArch64 flags N, Z, C, V, are set by compare instructions or by the arithmetic instructions setting flags.

The following table describes the relationship between AArch64 condition codes and flags.

Table A1.1: AArch64 condition codes and flags

AArch64 Condition Code	Description	Flags tested
EQ	Equal	Z == 1
NE	Not Equal	Z == 0
HS / CS	Unsigned Higher or Same	C == 1
LO / CC	Unsigned Lower	C == 0
MI	Minus	N == 1
PL	Positive or Zero	N == 0
VS	Overflow	V == 1
VC	No Overflow	V == 0
HI	Unsigned Higher	C == 1 & Z == 0
LS	Unsigned Lower or Same	C == 0 Z == 1
GE	Greater Than or Equal	N == V
LT	Less Than	N != V

Compare instructions are frequently used to set AArch64 flags. Note that `CMP Xn, #imm` is an alias of `SUBS XZR, Xn, #imm` and the flags are set according to the behavior of `SUBS` instruction.

A1.3.3 SVE Condition Codes and Flags

Based on the state of predicates, SVE architecture overloads the existing AArch64 flags N, Z, C, V adding a new meaning to both the flags and condition codes, as described in the following two tables.

Table A1.2: SVE interpretation of AArch64 condition flags

AArch64 Flag	SVE interpretation	Condition
N	First	Has a value of 1 if the first active element was true, otherwise has a value of 0.
Z	None	Has a value of 1 if no active element was true, otherwise has a value of 0.
C	!Last	Has a value of 0 if the last active element was true, otherwise has a value of 1.

AArch64 Flag	SVE interpretation	Condition
V	-	Used only in serialized vector loops for termination condition detection. Cleared to 0 by the SVE flag-setting instructions, except CTERMEQ and CTERMNE.

Table A1.3: SVE interpretation of AArch64 condition codes

AArch64 Condition Code	SVE name	Flags tested	SVE interpretation
EQ	NONE	Z == 1	No active elements were true.
NE	ANY	Z == 0	An active element was true.
HS / CS	NLAST	C == 1	The last active element was not true.
LO / CC	LAST	C == 0	The last active element was true.
MI	FIRST	N == 1	The first active element was true.
PL	NFRST	N == 0	The first active element was not true.
HI	PMORE	C == 1 & Z == 0	An active element was true but not the last active element.
LS	PLAST	C == 0 Z == 1	The last active element was true or none were true.
VS		V == 1	CTERM* comparison failed, but end of partition reached.
VC		V == 0	CTERM* comparison succeeded or end of partition not reached.
GE	TCONT	N == V	CTERM* termination condition not detected.
LT	TSTOP	N != V	CTERM* termination condition detected.

The condition flags are set either by an explicit test on a predicate (PTEST), or by some instructions that generate predicate result (CMP*, WHILE*, MOVs, NOTs, ..).

Chapter A2

Compiler support

The SVE support has been added to LLVM and GCC compilers. This section is applicable to the SVE LLVM compiler.

A2.1 Autovectorization

A2.1.1 Compiler options and pragmas

In the LLVM compiler, if the optimization level enabled is at least O2, the loop vectorizer is enabled by default. The loop vectorizer can be disabled through clang using the command line flag: `-fno-vectorize`. This is useful for measuring the scalar code performance.

The vectorization approach can be tuned with the following command line flag and pragmas. The specific loop unroll factor is set with the command line flag: `-force-vector-unroll=<desiredFactor>`. The following pragmas affect only the loop statements immediately following it:

- `#pragma clang loop vectorize(assume_safety)` indicates to the compiler that the loop contains no data dependencies between loop iterations that would prevent vectorization.
- `#pragma clang loop vectorize(disable)` indicates to the compiler to suppress auto-vectorization of the loop.
- `#pragma clang loop vectorize(enable)vectorize_style(fixed_width)` signals to the compiler that the Advanced SIMD instructions are preferred for vectorization.
- `#pragma clang loop vectorize(enable)vectorize_style(scaled_width)` signals to the compiler that SVE instructions are preferred for vectorization.

Refer to [3] for more information on using pragmas to control auto-vectorization.

A2.1.2 Generating the Autovectorization diagnostics

The compiler loop vectorizer can generate optimization diagnostics which can be queried using command line options to identify and analyze the loops that are skipped by the vectorizer.

The optimization remarks can be enabled by the following:

- `-Rpass=loop-vectorize`, to identify the loops that were successfully vectorized.
- `-Rpass-missed=loop-vectorize`, to identify the loops that failed vectorization.
- `-Rpass-analysis=loop-vectorize`, to identify statements that caused vectorization to fail.

Refer to [3] for more information on using optimization remarks with the compiler.

A2.2 Calling conventions

The AArch64 procedure call standard [4] specifies that:

- The general-purpose registers, X0-X7, are used to pass argument values into a subroutine and to return result values from a function.
- If a floating-point or vector type is passed to or returned from a subroutine, SIMD and floating-point registers V0-V7 are used to pass argument values into a subroutine and to return result values from a function.
- The general-purpose registers X0-X7 and X9-X15 are corruptible by the subroutine (i.e. caller saved), while X19-X28 must be preserved by a callee if modified.
- The registers V0-V7 and V16-V31 are corruptible by the subroutine (i.e. caller saved), while the low 64 bits of V8-V15 must be preserved by a callee if modified.

In the SVE procedure call standard [4] it is specified that:

- If a scalable vector or predicate type is passed to or returned from a subroutine, vector registers Z0-Z7 are used to pass scalable vector arguments to a subroutine and to return scalable vector results from a function, and predicate registers P0-P3 are used to pass scalable predicate arguments to a subroutine and to return scalable predicate results from a function.
- The SVE vector registers Z0-Z7 are corruptible by the subroutine (i.e. caller saved). If a subroutine takes arguments or returns results in SVE or predicate registers, the entire content of SVE vector registers Z8-Z31 must be preserved by a callee if modified. In other cases, a callee needs to preserve only the low 64 bits of Z8-Z15 across the call.
- The predicate registers P0-P3 are corruptible by the subroutine (i.e. caller saved). If a subroutine takes arguments or returns results in SVE or predicate registers, predicate registers P4-P15 must be preserved by a callee if modified. In other cases, a callee does not need to preserve any predicate registers across the call.

Chapter A3

SVE Instruction Set

This chapter provides an overview of the SVE Instruction Set.

A3.1 Overview

The SVE instruction set consists of:

- Load and store instructions.
- Floating-point arithmetic instructions.
- Integer arithmetic instructions.
- Reduction instructions.
- Type conversion and element size conversion instructions.
- Bitwise and logical operations instructions.
- Permutation instructions.
- Instructions operating on predicates and/or producing predicates.
- Loop control instructions.

A3.1.1 Constructive and destructive instructions

The SVE instructions are either constructive or destructive. In a constructive instruction the destination register is different than any of the source registers. Hence, the content of all source registers is preserved by a constructive instruction. In a destructive instruction one of the source registers is also the destination register. Hence, the content of this source register is destroyed by a destructive instruction. This is illustrated in [Figure A3.1](#).

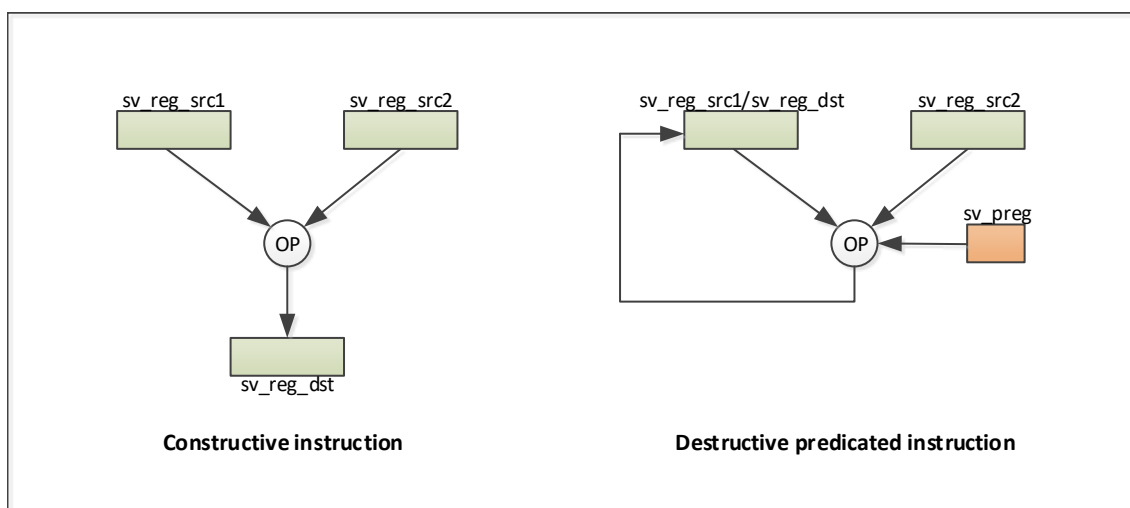


Figure A3.1: Constructive and Destructive instructions

In order to improve the programming flexibility with destructive instructions, certain non-commutative operation destructive instructions exist in two forms: one that destroys the content of the first source register, and the other ‘reversed’ instruction that destroys the content of the second source register. The ‘reversed’ instruction mnemonic name ends with the letter ‘R’.

A3.1.2 Predication of constructive and destructive instructions

The SVE instruction can be predicated or unpredicated, and most of the operations are represented in either predicated or unpredicated form. Although, a small subset of operations is represented in SVE instruction set in both predicated and unpredicated form, where predicated instruction is destructive and unpredicated instruction is constructive.

A3.1.3 Move prefix

The `MOVPRFX` instruction is a special instruction that provides a hint to the hardware that it can be combined with the destructive instruction that follows it in program order, to create a single constructive operation. Moreover, with the preceding `MOVPRFX` instruction the destructive instruction with merging predication can be converted to use zeroing predication.

The `MOVPRFX` instruction and prefixed destructive instruction can be combined if:

- They use the same destination register,
- If `MOVPRFX` is predicated, it must be used with predicated prefixed instruction controlled by the same predicate register. Additionally, `MOVPRFX` element size must be the same as the largest element size of the prefixed instruction.
- If `MOVPRFX` is unpredicated, then the prefixed instruction may use any predicate register and any element size, or it may be unpredicated.

A3.2 Element size and type conversion instructions

A3.2.1 Element size

The instructions support four different element sizes: 8-, 16-, 32- and 64-bits in a vector register.

A3.2.2 Element size conversions

The element size conversion instructions perform a variety of packing and unpacking operations on both vector elements and predicate elements:

- Pack with truncate exists for both vector elements and predicate elements: `TRN1` and `TRN2`, `UZP1` and `UZP2`, `ZIP1` and `ZIP2`.
- Signed and Unsigned Pack without saturation for vector elements: `SUNPKHI`, `SUNPKLO`, `UUNPKHI`, `UUNPKLO`.
- Signed and Unsigned Unpack for vector elements: `UXTB`, `UXTH`, `UXTW`, `SXTB`, `SXTH`, `SXTW`.
- Unpack for predicate elements: `PUNPKHI`, `PUNPKLO`.

A3.2.3 Type conversions

The following type conversion instructions are available:

- Converting one floating-point type to another: `FCVT`.
- Converting floating-point types to integer types: `FCVTZS`, `FCVTZU`.
- Converting integer types to floating-point types: `SCVTF`, `UCVTF`.

A3.3 Loads and Stores

The SVE instruction set provides a suite of predicated load and store instructions and unpredicated load and store instructions.

In the predicated load and store instructions the memory data size may be the same (packed data access) or smaller (unpacked data access) than the vector elements size. With the unpacked data access, the memory content is loaded to vector register(s) with zero or sign extension, and only truncated lower bits of a vector register elements are stored to the memory.

A3.3.1 Contiguous loads and stores

The predicated load instructions `LD1B`, `LD1H`, `LD1W` and `LD1D` read consecutive memory locations to a single vector register elements. The predicated structure load instructions `LD[N]B`, `LD[N]H`, `LD[N]W` and `LD[N]D` read `N` consecutive memory locations to the same-numbered element in each of the `N` vector registers, where `N = 2, 3` or `4`.

The predicated store instructions `ST1B`, `ST1H`, `ST1W` and `ST1D` write from a single vector register elements to consecutive memory locations. The predicated structure store instructions `ST[N]B`, `ST[N]H`, `ST[N]W` and `ST[N]D` write from the same-numbered element in each of the `N` consecutive vector registers to `N` consecutive memory locations, where `N = 2, 3` or `4`.

A3.3.2 Loads with broadcast

Also supported are predicated load instructions with the broadcast of one element: `LD1RB`, `LD1RH`, `LD1RW` and `LD1RD`, and with the broadcast of 128-bits long group of elements: `LD1RQB`, `LD1RQH`, `LD1RQW` and `LD1RQD`.

A3.3.3 Gather loads and Scatter stores

The SVE architecture supports complex memory data access patterns with predicated gather-load and scatter-store instructions.

The supported vector register element sizes are only 32- and 64-bits. Memory data size can be any of 8-, 16-, 32- or 64-bits. This means that any non-contiguous memory element access of less than a 32-bits is unpacked.

A complex gather-load and scatter-store element accesses address non-contiguous memory locations specified by:

- a scalar base register plus a vector of offsets. The offsets in a vector register can either be 32-bits or 64-bits long. With the 32-bits vector register offset, data vector register element can either be 32-bits or 64-bits long. With the 64-bits vector register offset, data vector register element can only be 64-bits long.
- a vector of base addresses plus an immediate offset. The vector register base can either be 32-bits or 64-bits long. With the 32-bits vector register base, data vector register element can only be 32-bits long. With the 64-bits vector register base, data vector register element can only be 64-bits long.

Note that depending on the implementation, gather-loads and scatter-stores may be expensive in terms of the execution latency.

A3.3.4 Unpredicated load and store

The SVE instruction set includes unpredicated vector load and store instructions, `LDR` and `STR`. The unpredicated load instruction `LDR` fully updates a vector register with the consecutive bytes from data memory. The unpredicated store instruction `STR` writes an entire vector register content to a contiguous data memory locations. The unpredicated load and store do not take endianness into account.

A3.3.5 Non-temporal continuous loads and stores

Non-temporal contiguous loads `LDNT1B`, `LDNT1H`, `LDNT1W` and `LDNT1D`, and non-temporal contiguous stores `STNT1B`, `STNT1H`, `STNT1W` and `STNT1D`, are normal contiguous loads and stores that include a hint to the memory system that the accessed memory locations are not expected to be accessed again soon and do not need to be kept in local caches.

A3.3.6 First faulting loads

First-faulting loads acknowledge memory faults and raise an exception only for faults occurring due to the memory access performed for the first active element, while other active elements memory faults are ignored. The ignored faults are recorded in the FFR register. First-faulting loads `LDF1B`, `LDF1H`, `LDF1W`, `LDF1D`, `LDF1SB`, `LDF1SH` and `LDF1SW`, can be contiguous or gather loads.

First-faulting loads acknowledgment is enabled by initializing the FFR register with instruction `SETFFR`. After one or more first-faulting loads, the state of FFR register may be read by the `RDFFR` instruction.

A3.3.7 Non-faulting loads

Non-faulting loads ignore memory faults of all active elements. The ignored faults are recorded in the FFR register.

A3.3.8 Prefetching

The instruction set enables a programmer to send a hint to the memory system that memory accesses from a specified address are likely to occur in the near future. Then, the memory system can respond by preloading the specified address into one or more caches, therefore speeding up the upcoming memory access execution. The prefetch instructions are only hints. Hence, they have a minimal impact on actual execution but can improve performance significantly if issued in a timely manner.

The instructions `PRFB`, `PRFH`, `PRFW` and `PRFD`, allow contiguous or gather prefetch of bytes, half-words, words or double-words. Prefetch can be for load or for store, and the target is one of L1, L2, L3 caches.

The coding example with prefetch instructions can be found in [A4.1.3 For/While loop SVE vectorization with prefetching](#).

A3.3.9 Memory ordering

Memory reads occur out of order:

- Two reads of the same location by different load instructions occur in any order.
- Multiple element reads of the same vector load instruction occur in any order.

Memory writes mostly occur out of order:

- Multiple element writes by the same vector store instruction occur in any order, except when writes are to the same location and then they occur in element indices order.

A3.4 Control flow

A3.4.1 Predicate initialization

A predicate register can be initialized to a fixed pattern with instructions `PTRUE` and `PFALSE`, or can be set as a result of a vector compare instruction (`CMPEQ`, `CMPGT`, ..) or a loop partition instruction (`WHILELT`, `WHILELS`, ..). Additionally, a predicate register can be a result of a logical operation on predicate registers, or can be loaded from memory.

A3.4.2 Index initialization

With the `INDEX` instruction, vector register lanes can be initialized in vector length agnostic manner to uniformly descending or ascending indexes.

A3.4.3 Predicate count

There exist instructions useful for loop counter updates.

Instructions `CNTB`, `CNTH`, `CNTW` and `CNTD` produce the number of active elements, at the specified element size, within a predicate register. Similarly, instructions `INCB`, `INCH`, `INCW` and `INCD` increment and instructions `DECB`, `DECH`, `DECW` and `DECD` decrement scalar register or vector register elements by the number of active predicate elements in a predicate register.

A3.4.4 Conditional loop break

There exist instructions enabling exit or break out of a vectorized loop due to a data-dependent condition: `BRKA`, `BRKB`, `BRKN`, `BRKPA` and `BRKPB`, and its flag setting alternatives: `BRKAS`, `BRKBS`, `BRKNS`, `BRKPAS` and `BRKPBS`. With these instructions, the result of a vector comparison is converted into a loop partition predicate which truncates the current vector iteration at the point where the condition first succeeds.

A3.5 Dot product

There exist two dot-product instructions in SVE, `SDOT` and `UDOT`. The `SDOT` instruction operates on signed integer elements, and produces widened signed integer partial dot product results. The `UDOT` instruction operates on unsigned integer elements, and produces widened unsigned integer partial dot product results. The elements of two source vector registers are divided in groups of 4 consecutive elements. Within each group, the elements in the first source vector register are multiplied by the corresponding elements in the second source vector register. The resulting widened products are summed and added to the destination vector register element that aligns with the group of four elements in the first source vector register, as:

$$c_i += a_{4i} * b_{4i} + a_{4i+1} * b_{4i+1} + a_{4i+2} * b_{4i+2} + a_{4i+3} * b_{4i+3}$$

The result element size can be either a 32-bits or 64-bits integer, while the input element size is $1/4$ of the result element size to minimize the risk of overflow. In the case of a 32-bits result element, four 8-bits input elements are taken from the corresponding 32-bits lane of each input operand. In the case of a 64-bits result element, four 16-bits input elements are taken from the corresponding 64-bits lane of each input operand.

In [Figure A3.2](#), dot product instruction with 64-bits result element is illustrated for 256-bits vector length.

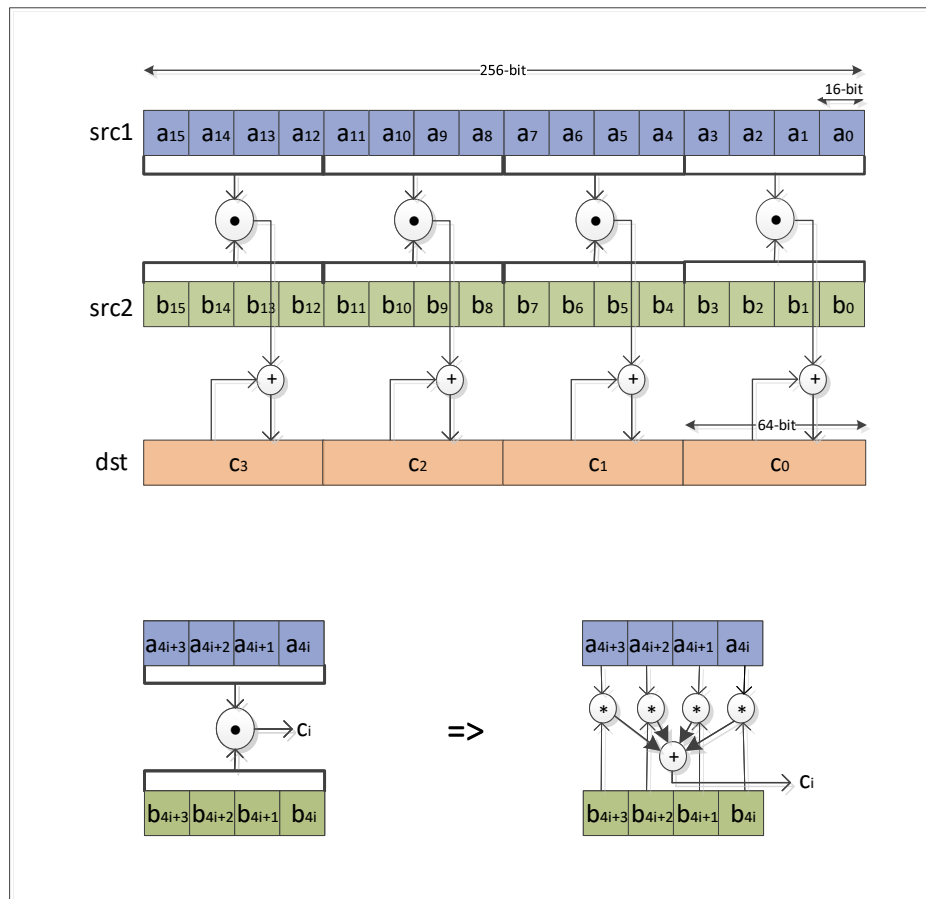


Figure A3.2: Real-valued dot product instruction

A3.6 Complex floating-point operations

The SVE instructions performing complex-valued addition and multiply-accumulate operations assume that the real and the imaginary part of a complex data are organized in the interleaved order in a vector register. Hence, one complex data is held in a vector register with its real part in an even element, and its imaginary part in a subsequent odd element.

A3.6.1 Complex-valued addition

The SVE instruction set implements complex-valued addition of floating-point data with the instruction `FCADD`. This instruction takes an additional argument, the rotation parameter that can have either `#90` or `#270` values:

src1	src2	result	FCADD
$a + i * b$	$x + i * y$	$src1 + i * src2 = (a - y) + i * (b + x)$	<code>#90</code>
$a + i * b$	$x + i * y$	$src1 - i * src2 = (a + y) + i * (b - x)$	<code>#270</code>

A3.6.2 Complex-valued multiply-accumulate

The SVE instruction set does not implement a full complex-valued multiply-accumulate in a single instruction, but the instruction provided produces partial products and accumulations which can be used to construct a full complex-valued multiply-accumulate. Therefore, two complex multiply-accumulate instructions `FCMLA` are necessary to calculate the entire complex-valued multiply-accumulate. This instruction takes an additional argument, the rotation parameter, with values `#0`, `#90`, `#180` or `#270`.

Here are some complex multiplication calculation examples with the accumulation part omitted:

src1	src2	result	FCMLA
$a + i * b$	$x + i * y$	$src1 * src2 = (a * x - b * y) + i * (a * y + b * x)$	<code>#0, #90</code>
$a + i * b$	$x + i * y$	$src1 * * src2 = (a * x + b * y) + i * (a * y - b * x)$	<code>#0, #270</code>
$a + i * b$	$x + i * y$	$-src1 * src2 = (-a * x + b * y) + i * (-a * y - b * x)$	<code>#270, #180</code>

A3.7 Vector by indexed element instructions

Some of the SVE instructions exist in vector by indexed element form. An operation is performed between each element of a vector register and the one particular (indexed) element of another vector register. In the Advanced SIMD vector by indexed element instructions the operation is performed between all elements of one operand and the one indexed element of the other operand. However, in SVE vector by indexed element instructions, the operation is performed per each 128-bits granule, between all elements of the first operands' granule and one indexed element of the second operands' granule.

Floating-point real-valued and complex-valued multiplication instructions `FCMLA`, `FMLA`, `FMLS` and `FMUL` support this feature, as well as integer dot-product instructions `SDOT` and `UDOT`.

In [Figure A3.3](#) the example of instruction `FMUL z3.h, z1.h, z2.h[2]` is illustrated for 256-bits vector length:

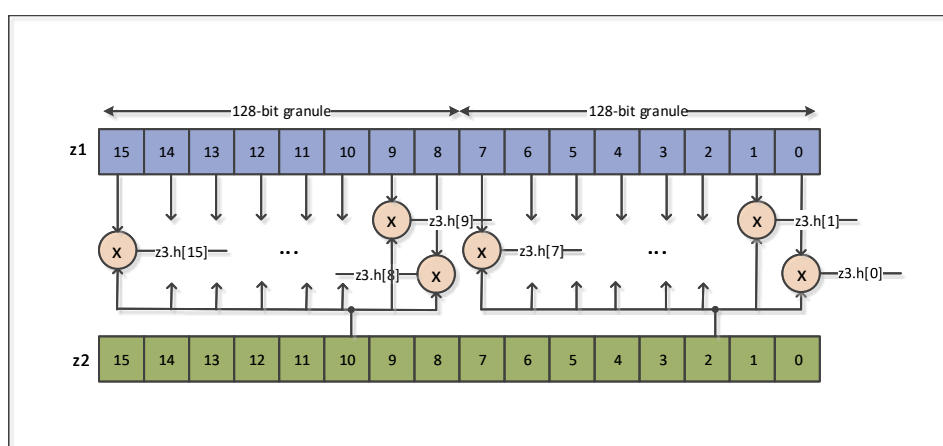


Figure A3.3: Vector by indexed element instruction

A3.8 Other instructions

A3.8.1 Reduction instructions

Reduction instructions perform arithmetic across active elements of a single input vector register, and produce a scalar result. These instructions make possible vectorization approaches where partial results are calculated per-lane, and the final scalar result is computed by a reduction instruction.

There exist reduction instructions performing basic integer and floating-point arithmetic and logical operations on a vector register elements: `ANDV`, `ORV`, `EORV`, `FADDV`, `SADDV`, `UADDV`, `FMAXV`, `SMAXV`, `UMAXV`, `FMINV`, `SMINV` and `UMINV`.

Note that the reduction instructions are expensive in terms of the execution latency especially for high vector lengths.

A3.8.2 Permutation instructions

Predicated instructions `COMPACT` and `SPLICE`, and unpredicated instructions `EXT`, `TRN1`, `TRN2`, `ZIP1`, `ZIP2`, `UZP1` and `UZP2` combine elements of two source vector registers into the resulting vector register.

Predicated instructions `REVB`, `REVB` and `REVV`, and unpredicated instructions `TBL` and `REV` permute elements of one source vector register into the resulting vector register.

A3.8.3 Instructions with elements crossing the 128-bits vector granule

In most of the SVE instructions the elements of source vectors contribute to destination vector resulting elements located within the same 128-bits vector granule, allowing for a minimum instruction execution time. Although, there exist a small number of the SVE instructions that do not satisfy this non-crossing 128-bits granule boundary property. In these SVE instructions, some or all source vector elements contribute to destination vector resulting elements from a different 128-bits vector granule, hence crossing the 128-bits granule boundary. The instructions with elements crossing the 128-bits granule boundary might be more expensive in terms of the execution time than instructions with elements non-crossing this boundary.

Here are some of the 128-bits granule boundary crossing instructions: `EXT`, all reduction instructions, `TBL`, `SUNPKHI`, `SUNPKLO`, `ZIP1`, `ZIP2`, `UZP1`, `UZP2`, ..

A3.8.4 Instructions moving data between general-purpose register file and scalable vector register file

The instructions `LASTA` and `LASTB`, and its conditional alternatives `CLASTA` and `CLASTB`, allow the extraction of a particular element of a vector register to a general-purpose register based on the predicate state.

The SVE instructions that move data from general-purpose register file to scalable vector register file and vice versa are more expensive in terms of the execution time than instructions operating only on scalable vector registers.

Here are other instructions moving data between general-purpose register file and scalable vector register file: `INSR`, `INDEX`, `MOV` and `DUP`.

Chapter A4

SVE Vector Length Agnostic programming approach

This chapter provides SVE programming tips.

A4.1 For/While loop vectorization

The following is an example of a simple vectorization of *For* and *While* loops, showing the vectorization approach difference between the Advanced SIMD instruction set with fixed vector length and the vector length agnostic SVE instruction set.

A simple *For* loop example that computes the addition of two integer array elements:

```
void example_for( int *out, int *a, int *b, int N) {  
    for (int i=0; i<N; i++) {  
        out[i] = a[i] + b[i];  
    }  
}
```

A simple *While* loop example that computes the addition of two integer array elements:

```
void example_while( int *out, int *a, int *b, int N) {  
    while (N > 0) {  
        *out = *a + *b;  
        a++;  
        b++;  
        out++;  
        N = N - 1;  
    }  
}
```

A4.1.1 For/While loop Advanced SIMD vectorization

```
1      AND      w6, w3, 0xffffffffc  
2      CBZ      w6, .L_tail  
3  
4      .L_vector_loop:  
5          LD1    {v0.4s}, [x1], #16  
6          LD1    {v1.4s}, [x2], #16  
7          ADD    v5.4s, v0.4s, v1.4s  
8          ST1    {v5.4s}, [x0], #16  
9          SUB    w6, w6, #4  
10         CBNZ   w6, .L_vector_loop  
11  
12         .L_tail:  
13             AND    w3, w3, #3  
14             CBZ    w3, .L_end  
15  
16         .L_scalar_loop:  
17             LDRSW  w9, [x1], #4  
18             LDRSW  w10, [x2], #4  
19             ADD    w11, w9, w10  
20             STR    w11, [x0], #4  
21             SUB    w3, w3, #1  
22             CBNZ   w3, .L_scalar_loop  
23  
24         .L_end:  
25             RET
```

In this example, the working element size is 32-bits, hence 4 output results are computed in vector lanes of the Advanced SIMD vectorized loop iteration (the loop at lines 4-10) filling up 128-bit vector length. To cope with

arrays of lengths that are not multiple of 4, it is necessary to implement the scalar loop to calculate the remaining results if any (the loop at lines 16-22).

A4.1.2 For/While loop SVE vectorization

The SVE vector length agnostic vectorization approach consists of carefully setting the predicates to manage register partitioning, predicate handling, loop counter and pointer offset updates over loop iterations with the help of specific loop control instructions. The SVE vectorized code is shorter and more compact than the Advanced SIMD vectorized code. The vectorized loop is seamlessly terminated, and there's no need for the scalar loop.

```

1      MOV      w3, w3
2      MOV      x9, #0
3
4      WHILELT   p1.s, x9, x3
5      B.NONE    .L_return
6
7      .L_loopStart:
8      LD1W      z1.s, p1/Z, [x1, x9, LSL #2]
9      LD1W      z2.s, p1/Z, [x2, x9, LSL #2]
10     ADD       z1.s, p1/M, z1.s, z2.s
11     ST1W      z1.s, p1, [x0, x9, LSL #2]
12     INCW      x9
13     WHILELT   p1.s, x9, x3
14     B.FIRST   .L_loopStart
15
16     .L_return:
17     RET

```

The processing lane width is 32-bits as with the Advanced SIMD vectorization. At line 4, the governing predicate `p1` is initialized for 32-bit elements with the instruction `WHILELT` that sets its active lanes based on the comparison of the loop counters' current state (register `x9`) and the array length (register `x3`).

In the vectorized loop at lines 7-14, the governing predicate controls the active lanes of the load, addition and store instructions. Only valid data are accessed from the memory and loaded to the vector registers (lines 8-9), and consequently processed (line 10) and stored to the memory (line 11). The inactive lanes of vector registers are zeroed by load instructions (lines 8-9).

The value in register `x9` is used both as the loop counter and the load and store pointer offset, and each vectorized loop iteration it is incremented in the vector length agnostic manner. With the instruction `INCW`, this value is incremented by the number of 32-bit elements in the implemented vector length (line 12). Note that the number of output results computed in one vectorized loop iteration depends on the implemented vector length.

Based on the updated loop counter, the new vector partitioning is performed with `WHILELT` instruction (line 13). The next loop iteration occurs if there is at least one more input array element to process. Otherwise the loop exits. This is managed by the conditional branch `B.FIRST` (line 14).

The above SVE vector length agnostic vectorization code can be re-written in C with the Arm C language extension (ACLE) for SVE [5] as:

```

1      void example_for( int *out, int *a, int *b, int N) {
2          uint64_t    i = 0;
3          uint64_t    vl = svcntw();
4          svbool_t     pred;
5          svint32_t    sva, svb, svres;
6
7          pred = svwhilelt_b32( i, (uint64_t)N);
8
9          while(svptest_first(svptrue_b32(), pred)) {
10             sva = svld1( pred, &a[i]);

```

```

11         svb  = svld1( pred, &b[i]);
12         svres = svadd_m( pred, sva, svb);
13         svst1( pred, &out[i], svres);
14         i    += vl;
15         pred  = svwhilelt_b32( i, (uint64_t)N);
16     }
17 }

```

A4.1.3 For/While loop SVE vectorization with prefetching

The execution speed of the optimized loop from the previous paragraph can be improved by using the prefetch instructions. Prefetch instruction signals to the memory system to preload beforehand memory addresses that will be used in the code that follows.

```

1      PTRUE      p0.s
2      PRFW      PLDL1STRM, p0, [x1]
3      PRFW      PLDL1STRM, p0, [x1, #1, MUL VL]
4      PRFW      PLDL1STRM, p0, [x1, #2, MUL VL]
5      PRFW      PLDL1STRM, p0, [x2]
6      PRFW      PLDL1STRM, p0, [x2, #1, MUL VL]
7      PRFW      PLDL1STRM, p0, [x2, #2, MUL VL]
8
9      MOV        w3, w3
10     MOV        x9, #0
11     ADDVL      x10, x1, #3
12     ADDVL      x12, x2, #3
13
14     WHILELT    p1.s, x9, x3
15     B.NONE     .L_return
16
17     .L_loopStart:
18     LD1W       z1.s, p1/Z, [x1, x9, LSL #2]
19     LD1W       z2.s, p1/Z, [x2, x9, LSL #2]
20     PRFW      PLDL1STRM, p0, [x10, x9, LSL #2]
21     PRFW      PLDL1STRM, p0, [x12, x9, LSL #2]
22     ADD        z1.s, p1/M, z1.s, z2.s
23     ST1W      z1.s, p1, [x0, x9, LSL #2]
24     INCW       x9
25     WHILELT    p1.s, x9, x3
26     B.FIRST   .L_loopStart
27
28     .L_return:
29     RET

```

In the above example, prefetch instructions with **PLDL1STRM** specifier (lines 2-7, 20-21) give a hint to the memory system that specified memory addresses will be accessed by the load instructions (**PLDL1STRM**) at lines 18-19. The memory system would take actions to preload the specified memory addresses to L1 cache (**PLDL1STRM**). With prefetch instruction it is also signaled to the memory system that these memory addresses will be used only once and that there's no need to keep them in the local cache (**PLDL1STRM**).

In this example, all true predicate **p0** is used with prefetch instructions. Hence, an additional 3 vector lengths of data just after both of the input arrays will be prefetched, but not loaded. In certain applications where these redundant prefetches could cause a problem, it is preferred to set predicate **p0** independently for each vector length prefetch with the appropriate **WHILELT** instruction within the inner loop.

A4.2 Do-while loop SVE vectorization

The following is an example of a simple vectorization of *Do-while* loop with the SVE instruction set in vector length agnostic manner.

Here is the *Do-while* loop that computes sum of the first N squares:

```
void example_sum_squares( int N, int * sum) {
    int res = 0;
    if (N > 0) {
        do {
            res += N*N;
            N--;
        } while (N > 0);
    }
    *sum = res;
}
```

The vectorization approach consists in computing one partial sum in each 32-bit vector lane, and then outside of the loop calculating the final sum by the reduction addition of partial sums.

1	PTRUE	p1.s
2	MOV	z5.s, #0
3	MOVI	d6, #0
4		
5	INDEX	z0.s, x0, #-1
6	CMPGT	p0.s, p1/Z, z0.s, #0
7	B.NONE	.L_result
8		
9	.L_loopStart:	
10	MLA	z5.s, p0/M, z0.s, z0.s
11	DECW	z0.s
12	CMPGT	p0.s, p1/Z, z0.s, #0
13	B.FIRST	.L_loopStart
14		
15	UADDV	d6, p1, z5.s
16		
17	.L_result:	
18	STR	s6, [x1]
19	RET	

The number of partial sums computed in a vectorized loop is equal to the number of 32-bit lanes in the implemented vector length. The partial sums are held in vector register z5.

Firstly, for each vector lane the starting input element is set in vector length agnostic manner with instruction INDEX (line 5).

The vector partitioning and the setting of loop processing governing predicate p0 is performed with instruction CMPGT (line 6). The lanes holding positive elements are set to active, while the rest of lanes are deactivated.

In the vectorized loop (lines 9-13), the governing predicate controls the active lanes that contribute to the partial sums computed with instruction MLA (line 10).

The input elements in register z0 are also acting as the loop counter (in vector form). The next group of input elements is computed in vector length agnostic manner with instruction DECW (line 11). The current input element in each vector lane is decremented by the number of 32-bit elements in the implemented vector length, to obtain the input element of that lane for the next loop iteration. Then, the new vector partitioning is performed based on the newly calculated input elements with instruction CMPGT (line 12). If there exists at least one positive input element to process the next loop iteration occurs, otherwise the loop exits. This is checked with the conditional branch B.FIRST (line 13).

Lastly, the final sum is calculated outside of the loop by the reduction instruction `UADDV` (line 15). The partial sums from all lanes of vector register `z5` are summed, and this is controlled by the predicate `p1` that has got all elements set to active.

The above SVE vector length agnostic vectorization code can be re-written in C with the Arm C language extension (ACLE) for SVE [5] as:

```
1  void example_sum_squares( int N, int * sum) {
2      svbool_t    pred_N;
3      svint32_t   svN_tmp;
4      svbool_t    p_all = svptrue_b32();
5      svint32_t   acc = svdup_s32(0);
6
7      if (N > 0) {
8          svN_tmp = svindex_s32( N, -1);
9          pred_N = svcmpgt( p_all, svN_tmp, 0);
10
11         do {
12             acc      = svmla_m( pred_N, acc, svN_tmp, svN_tmp);
13             svN_tmp = svsub_x( p_all, svN_tmp, svcntw());
14             pred_N = svcmpgt( p_all, svN_tmp, 0);
15         } while ( svptest_first( p_all, pred_N));
16     }
17
18     *sum = (int) svaddv( p_all, acc);
19 }
```

A4.3 Effective vector length bandwidth utilization tips

Effective vectorization involves taking maximum advantage of the available load/store and processing bandwidth. The goal is to load and process enough elements to maximally fill up the available vector length in a vector length agnostic manner. In order to achieve this, it is often necessary to arrange data in a specific way within a vector register. Sometimes, it is even required to unroll the loop to accomplish this. In later sections, more complex data rearrangements and loop unrolling examples and tips will be illustrated.

In the following FIR filtering example, a multiply-add operation on the 32-bit wide elements is performed in a loop. These data arrangement steps are applied preparing two vector operands for multiply-add operation:

- The first vector operand is filled with an array of 16-bit wide input data, sign extended to 32-bits,
- The second vector operand is populated with one 16-bit wide filter coefficient, sign extended to 32-bits, and broadcasted over the vector length.

C reference:

```
void fir( int N, int T, short * in, short * coeff, short * out) {
    int i, j;
    int acc;

    for (i=0; i<N; i++) {
        acc = 0;
        for (j=0; j<T; j++) {
            acc += in[i+j] * coeff[j];
        }
        out[i] = (acc >> 16);
    }
}
```

Note: filter coefficients are stored in memory in the reverse order.

The SVE vectorized implementation:

```
1      MOV      x5, #0
2      SXTW     x0, w0
3      WHILELT  p4.s, x5, x0
4      B.NONE   .L_return
5
6      SXTW     x1, w1
7      ADD      x1, x3, x1, LSL #1
8      PTRUE    p5.s
9
10     .L_OuterLoop:
11         MOV      x6, #0
12         MOV      x7, x3
13         LD1SH    z10.s, p4/Z, [x2, x6, LSL #1]
14         LD1RSH   z1.s, p5/Z, [x7]
15         ADD      x6, x6, #1
16         ADD      x7, x7, #2
17         MUL      z10.s, p4/M, z10.s, z1.s
18         CMP      x7, x1
19         B.EQ     .L_InnerLoopEnd
20
21     .L_InnerLoop:
22         LD1SH    z2.s, p4/Z, [x2, x6, LSL #1]
23         LD1RSH   z1.s, p5/Z, [x7]
24         ADD      x6, x6, #1
25         ADD      x7, x7, #2
26         MLA      z10.s, p4/M, z2.s, z1.s
```

```

27         CMP        x7, x1
28         B.MI       .L_InnerLoop
29
30     .L_InnerLoopEnd:
31         ASR        z10.s, p4/M, z10.s, #16
32         ST1H       z10.s, p4, [x4]
33         INCH       x4
34         INCH       x2
35         INCW       x5
36         WHILELT    p4.s, x5, x0
37         B.FIRST    .L_OuterLoop
38
39     .L_return:
40         RET

```

Chapter A5

SVE2 Instruction set innovations

SVE2 architecture introduces:

- Integer arithmetic support including the signed/unsigned, saturated, real-valued and complex-valued arithmetic support. Many of these instructions are also available in the vector by indexed element form.
- Top-bottom processing approach where operand elements and result elements are of different size.
- Element pairwise processing instructions.
- Bitwise permute instructions.
- Histogram acceleration instructions.
- String processing acceleration instructions.
- Multiprecision arithmetic support instructions.
- Cryptography support instructions.

A5.1 Widening and narrowing instructions

A5.1.1 Top-Bottom processing approach

SVE2 architecture introduces a Top-Bottom processing approach for widening instructions and for some narrowing instructions. The instructions that employ Top-Bottom processing have mnemonic names that end either with the letter ‘B’ (Bottom instructions), or with the letter ‘T’ (Top instructions). There are few exceptions to the Top-Bottom naming convention. These are floating point convert precision instructions due to the alignment with legacy Advanced SIMD instruction mnemonics.

In the narrowing Top-Bottom SVE2 instructions, the element size of operands is larger than the element size of result.

- In the bottom narrowing instruction all input operands elements are processed, but only even elements of the result register are computed, while odd elements of the result register are zeroed.
- In the top narrowing instruction all input operands elements are processed, but only odd elements of the result register are produced, and merged with the existing even elements of the result register.

In the widening Top-Bottom SVE2 instructions, the element size of one or both operands is narrower than the element size of result.

- In the bottom widening instruction only the even elements of input operands with narrow elements are processed, while all result register elements are produced.
- In the top widening instruction only the odd elements of input operands with narrow elements are processed, while all result register elements are produced.
- For both bottom and top widening instruction, if there exist an input operand with elements of the same size as the elements of result, then all elements of that input operand are processed.

A5.1.2 Top-Bottom widening instructions

The SVE2 Instruction set provides widening instructions that operate on shorter elements of the input vector register, producing the resulting vector register with larger elements.

The one operand widening instructions produce result with elements larger than the operand elements. Here are these widening instructions: `SSHLLB` and `SSHLLT`, `USHLLB` and `USHLLT`, `FCVT` and `FCVTLT`.

The two operands integer arithmetic widening instructions whose operands contain elements of different size, produce the result with elements of the same size as the size of a wider operand elements. Here are these integer widening instructions: `SADDWB` and `SADDWT`, `UADDWB` and `UADDWT`, `SSUBWB` and `SSUBWT`, `USUBWB` and `USUBWT`.

The two operands integer arithmetic widening instructions whose operands contain elements of the same size, produce the result with larger elements than operands elements. Here are some of these integer widening instructions: `SADDLB` and `SADDLT`, `USUBLB` and `USUBLT`, `SABALB` and `SABALT`, `UABDLB` and `UABDLT`.

The three operands widening instructions have two operands with shorter elements, while the elements of the third operand and the result are larger. Here are some of these integer and floating-point multiply-accumulate and multiply-subtract widening instructions: `SMULLB` and `SMULLT`, `UMLSLB` and `UMLSLT`, `SQDMLALB` and `SQDMLALT`, `FMLALB` and `FMLALT`, `FMLSLB` and `FMLSLT`.

A5.1.3 Top-Bottom narrowing instruction

The SVE2 instruction set includes instructions that process operands with larger elements, and produce the result vector register with shorter elements.

Here are some of these narrowing instructions: `ADDHNB` and `ADDHNT`, `RSUBHNB` and `RSUBHNT`, `RSHRNB` and `RSHRNT`, `UQSHRNB` and `UQSHRNT`, `SQXTUNB` and `SQXTUNT`, `FCVT` and `FCVTNT`, `FCVTX` and `FCVTXNT`.

A5.2 Complex-valued integer operations

The SVE2 instructions performing complex-valued addition and multiply-accumulate operations assume that the real and the imaginary part of a complex data are organized in the interleaved order in a vector register. Hence, one complex data is held in a vector register with its real part in an even element, and its imaginary part in a subsequent odd element.

A5.2.1 Complex-valued addition

The SVE2 instruction set implements complex-valued addition of integer data with instructions `CADD` and `SQCADD`. These instructions take an additional argument, the rotation parameter that can have either `#90` or `#270` values:

src1	src2	result	CADD
$a + i * b$	$x + i * y$	$src1 + i * src2 = (a - y) + i * (b + x)$	<code>#90</code>
$a + i * b$	$x + i * y$	$src1 - i * src2 = (a + y) + i * (b - x)$	<code>#270</code>

A5.2.2 Complex-valued multiply-accumulate

The SVE2 instruction set does not implement a full complex-valued multiply-accumulate in a single instruction, but the instructions provided produce partial products and accumulations which can be used to construct a full complex-valued multiply-accumulate. Therefore, two complex multiply-accumulate instructions, `CMLA` or `SQRDCMLAH`, are necessary to calculate the entire complex-valued multiply-accumulate. These instructions take an additional argument, the rotation parameter, with values `#0`, `#90`, `#180` or `#270`.

Here are some complex multiplication calculation examples with the accumulation part omitted:

src1	src2	result	CMLA
$a + i * b$	$x + i * y$	$src1 * src2 = (a * x - b * y) + i * (a * y + b * x)$	<code>#0, #90</code>
$a + i * b$	$x + i * y$	$src1 * * src2 = (a * x + b * y) + i * (a * y - b * x)$	<code>#0, #270</code>
$a + i * b$	$x + i * y$	$-src1 * src2 = (-a * x + b * y) + i * (-a * y - b * x)$	<code>#270, #180</code>

A5.2.3 Complex-valued integer dot product

The SVE2 instruction set implements signed complex-valued integer dot product accumulation with instruction `CDOT`, as:

$$c_i += a_{2i} * b_{2i} + a_{2i+1} * b_{2i+1}$$

The result element size can be either complex-valued 32-bit or 64-bit, while the input element size is $1/4$ of the result element size to minimise the risk of overflow. In the case of a 32-bit result element, two signed complex-valued 8-bit input elements are taken from the corresponding 32-bit lane of each input operand. In the case of a 64-bit result element, two signed complex-valued 16-bit input elements are taken from the corresponding 64-bit lane of each input operand.

The complex-valued integer dot product instruction takes two source vectors with its elements real and imaginary parts organized in interleaved order. On the contrary, the result produced by these instructions is in de-interleaved format, and hence two `CDOT` instructions are needed: one for producing the real part of the result, and another one for producing the imaginary part of the result. These instructions take an additional argument, the rotation parameter, with values `#0`, `#90`, `#180` or `#270`. [Figure A5.1](#) illustrates the result computation depending on the rotation parameter value.

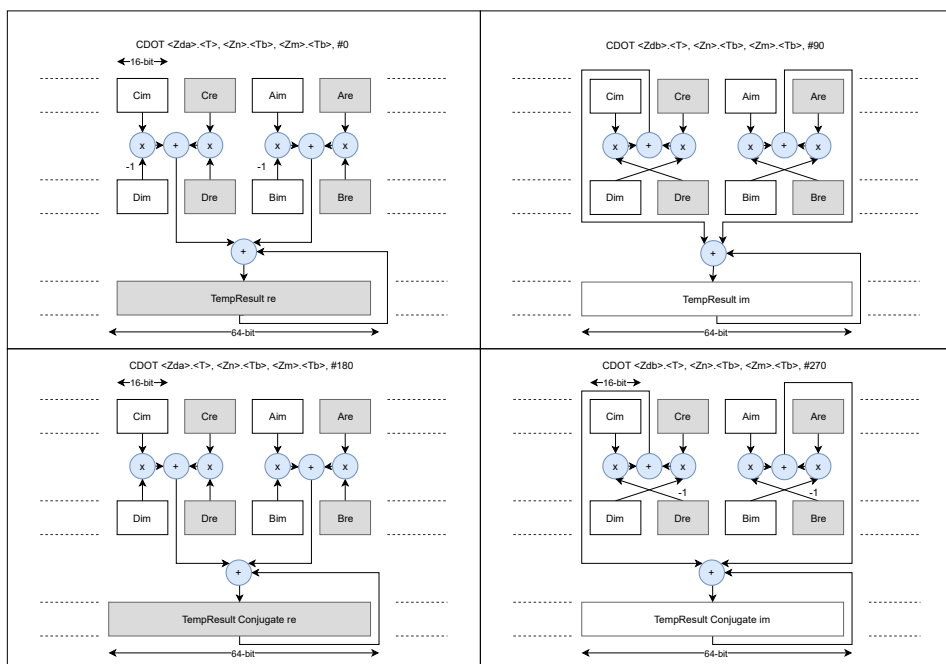


Figure A5.1: Complex-valued dot product and rotation parameter

Here are some complex-valued integer dot product calculation examples with the accumulation part omitted:

src1	src2	result	CDOT
$a0, a1$	$b0, b1$	$c0 = a0 * b0 + a1 * b1$	#0 $Re(c0)$, #90 $Im(c0)$
$a0, a1$	$b0, b1$	$c0 = a0^* * b0 + a1^* * b1$	#180 $Re(c0)$, #270 $Im(c0)$

A5.3 Other instructions

A5.3.1 Element size conversions

The SVE2 instruction set adds signed and unsigned packing instructions with saturation for vector register elements: `SQXTN`, `SQXTUN` and `UQXTN`.

A5.3.2 Element pairwise instructions

The SVE2 instruction set introduces integer and floating-point arithmetic instructions that are producing the resulting vector register elements by processing two neighboring source operand(s) elements.

Here are some of the pairwise instructions calculating addition, absolute difference and accumulate, finding maximum and minimum: `ADDP`, `FADDP`, `SADALP`, `UADALP`, `FMAXP`, `FMINNMP`, `SMAXP`, `UMINP`.

A5.3.3 Bitwise permute instructions

The SVE2 instruction set implements new bitwise permute instructions: `BDEP`, `BEXT`, `BGRP`. The bits of one source vector register are re-arranged by these instructions based on the bitmask from the other source vector register.

A5.3.4 Histogram acceleration instructions

The SVE2 instruction set introduces new instructions to speed up the histogram computation. The instruction `HISTCNT` delivers counts of the two vector registers particular elements match. The instruction `HISTSEG` delivers counts of matching elements in two vector registers 128-bit segments.

A5.3.5 String processing acceleration instructions

The SVE2 instruction set introduces new instructions to speed up the string processing. The instruction `MATCH` detects any matching elements, while the instruction `NMATCH` detects any no matching elements in two strings.

A5.3.6 Multiprecision arithmetic support instructions

The SVE2 instruction set introduces new instructions, `ADCLB` and `ADCLT`, to accelerate wide integer multiplication. These instructions can operate on either 32-bit or 64-bit elements. The instructions `ADCLB` and `ADCLT` add the first source vector element (even element with bottom or odd element with top instruction) and 1-bit carry from the second source vector odd element (bit-0) and accumulate into the destination vector even element. The results 1-bit carry is placed into the destination vector odd element (bit-0).

[Figure A5.2](#) illustrates the result computation within one element pair lane.

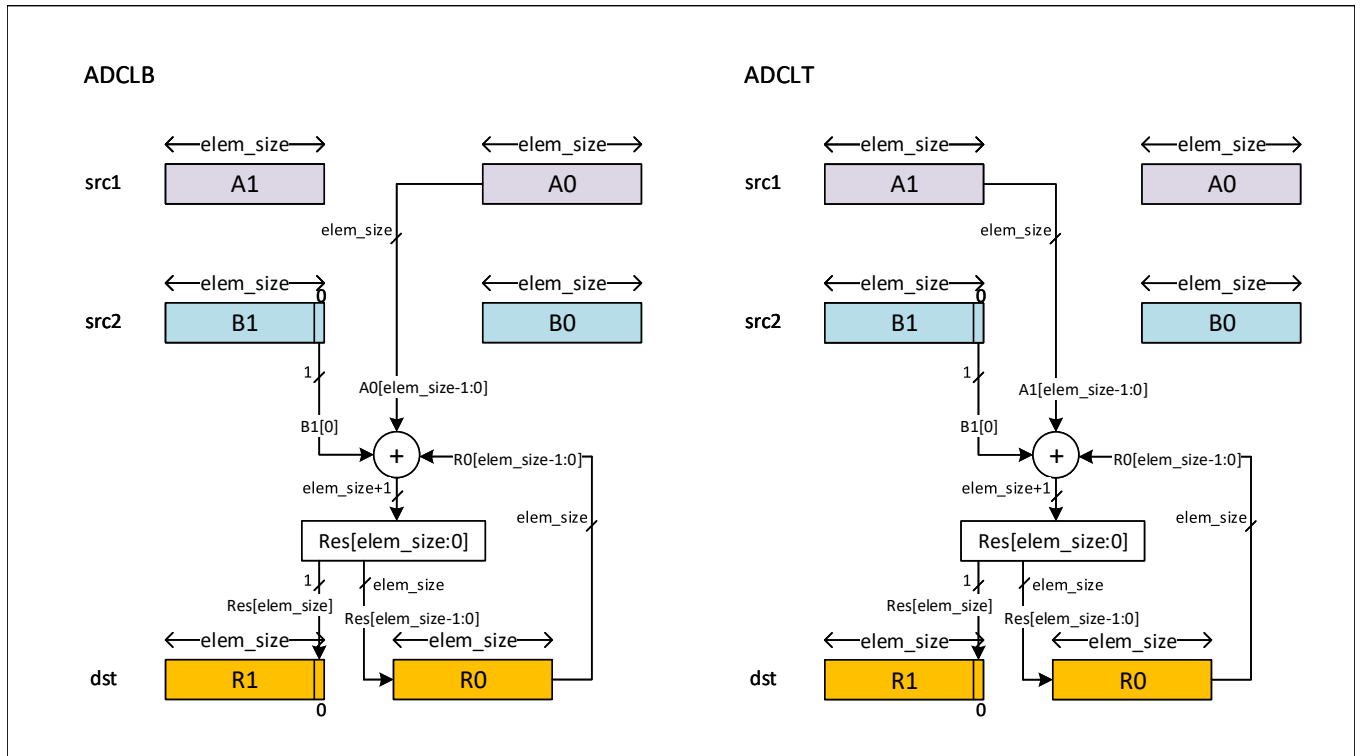


Figure A5.2: Multiprecision arithmetic instructions ADCLB and ADCLT

A5.3.7 Cryptography support instructions

The SVE2 instruction set implements new instructions as support to AES, SM4 and SHA3 standards:

- The AES standard support instructions AESE, AESD, AESIMC, AESMC.
- The SM4 standard support instructions SM4E, SM4EKEY.
- The SHA3 standard support instructions EOR3, RAX1, XAR, BCAX.

Chapter A6

SVE2 vectorization examples

This chapter provides SVE2 programming tips.

A6.1 Effective vector length bandwidth utilization tips

In Section [A4.3 Effective vector length bandwidth utilization tips](#) some tips for data rearrangement enabling effective load/store and processing bandwidth utilization were shown on the example of FIR filter. Here, FIR filtering vectorization is improved taking the advantage of new SVE2 Top-Bottom widening multiply and multiply-add instructions.

In the inner loop processing, the multiply-add `MLA` instruction is replaced by new SVE2 instruction set multiply-add widening Top-Bottom instructions `SMLALB` and `SMLALT` (lines 27-28). The usage of a widening multiply-add instruction seamlessly replaces the data expansion step performed by the load instructions in the SVE implementation. For this reason, in SVE2, twice as much data is loaded by one contiguous vector length load instruction. The loaded data is processed by two widening multiply-add instructions, one bottom and one top instruction, producing twice as much results while consuming the same load bandwidth as in the SVE implementation.

SVE2 vectorized implementation:

```

1      SXTW      x0, w0
2      SXTW      x1, w1
3      ADD       x5, x3, x1, LSL #1
4      ADD       x9, x4, x0, LSL #1
5
6      WHILELT   p4.b, x4, x9
7      B.NONE
8      PTRUE     p5.h
9
10     .L_OuterLoop:
11         MOV     x6, #0
12         MOV     x7, x3
13         LD1H    z2.h, p4/Z, [x2, x6, LSL #1]
14         LD1RH   z1.h, p5/Z, [x7]
15         ADD     x6, x6, #1
16         ADD     x7, x7, #2
17         SMULLB   z10.s, z2.h, z1.h
18         SMULLT   z11.s, z2.h, z1.h
19         CMP     x7, x5
20         B.EQ    .L_InnerLoopEnd
21
22     .L_InnerLoop:
23         LD1H    z2.h, p4/Z, [x2, x6, LSL #1]
24         LD1RH   z1.h, p5/Z, [x7]
25         ADD     x6, x6, #1
26         ADD     x7, x7, #2
27         SMLALB   z10.s, z2.h, z1.h
28         SMLALT   z11.s, z2.h, z1.h
29         CMP     x7, x5
30         B.MI    .L_InnerLoop
31
32     .L_InnerLoopEnd:
33         SHRNB   z10.h, z10.s, #16
34         SHRNT   z10.h, z11.s, #16
35         ST1H    z10.h, p4, [x4]
36         INCB    x4
37         INCB    x2
38         WHILELT   p4.b, x4, x9
39         B.FIRST  .L_OuterLoop
40
41     .L_return:
42         RET

```

A6.2 Vectorization with termination by the reduction

The usage of reduction instructions with loop vectorization is shown below in the example of the dot-product calculation:

```
void example_dotp( short *out, short *a, short *b, int N)
{
    int acc = 0;
    for (int i=0; i<N; i++) {
        acc += a[i] * b[i];
    }
    *out = acc >> 16;
}
```

In SVE2 vector length agnostic vectorization, the load bandwidth is maximized by loading 16-bit input data into 16-bit vector element lanes (lines 12-13). The partial sums are accumulated in 32-bit vector lanes using the new multiply-add widening instructions, the bottom instruction `SMLALB` (line 14) and the top instruction `SMLALT` (line 15). In this way, load bandwidth and multiply bandwidth are perfectly balanced.

1	SXTW	x7, w3
2	MOV	x9, #0
3	PTRUE	p0.s
4		
5	DUP	z3.s, #0
6	DUP	z4.s, #0
7		
8	WHILELT	p1.h, x9, x7
9	B.NONE	.L_return
10		
11	.L_loopStart:	
12	LD1H	z1.h, p1/Z, [x1, x9, LSL #2]
13	LD1H	z2.h, p1/Z, [x2, x9, LSL #2]
14	SMLALB	z3.s, z1.h, z2.h
15	SMLALT	z4.s, z1.h, z2.h
16		
17	INCH	x9
18	WHILELT	p1.h, x9, x7
19	B.FIRST	.loopStart
20		
21	ADD	z4.s, z3.s, z4.s
22	SADDV	d6, p0, z4.s
23	SQSHRNB	z6.h, z6.s, #16
24	STR	h6, [x0]
25		
26	.L_return:	
27	RET	

The even and odd partial dot-product accumulations are calculated respectively in vector lanes of registers `z3` and `z4`. Outside of the loop, at line 21, the even partial accumulations (register `z3`) are summed per lane with the odd partial accumulations (register `z4`). This is followed by the reduction addition instruction `SADDV` at line 22, such that the final dot product sum is computed in register `d6`. Finally, at line 23, there's a narrowing shift right instruction `SQSHRNB` extracting the high 16-bits of the 32-bit result to register `h6`.

A6.3 Loop unrolling example

The SVE2 vector length agnostic vectorization with loop unrolling is illustrated on the example of dot-product calculation:

```
void example_dotp( short *out, short *a, short *b, int N) {
    int acc = 0;
    for (int i=0; i<N; i++) {
        acc += a[i] * b[i];
    }
    *out = acc >> 16;
}
```

In the following SVE2 implementation the vectorized loop is unrolled once. Only the 2nd data processing part of the unrolled loop is truly predicated, while the predicate p2 used for the 1st data processing part is set such that its all elements are always active. The idea behind this approach is to perform processing within the loop while there's enough data to fully fill the vector lanes of the 1st data processing part, and to fill at least partially the vector lanes of the 2nd data processing part. If this condition is not satisfied the loop exits and the remaining data (if any) are processed outside of the loop starting at line 30.

```
1      MOV      z4.d, #0
2      MOV      z5.d, #0
3      MOV      z6.d, #0
4      MOV      z7.d, #0
5
6      PTRUE     p2.h
7      ADD      x3, x1, x3, LSL #1
8
9      INCB      x1, ALL, MUL #1
10     WHILELT   p4.b, x1, x3
11     B.NFRST   .L_loop_tail
12
13     .L_unrolled_loop:
14         LD1H    z0.h, p2/z, [x1, #-1, MUL VL]
15         LD1H    z2.h, p4/z, [x1, #0, MUL VL]
16         LD1H    z1.h, p2/z, [x2, #0, MUL VL]
17         LD1H    z3.h, p4/z, [x2, #1, MUL VL]
18
19         SMLALB   z4.s, z0.h, z1.h
20         SMLALT   z5.s, z0.h, z1.h
21         SMLALB   z6.s, z2.h, z3.h
22         SMLALT   z7.s, z2.h, z3.h
23
24         INCH     x1, ALL, MUL #4
25         INCH     x2, ALL, MUL #4
26
27         WHILELT   p4.b, x1, x3
28         B.FIRST   .L_unrolled_loop
29
30     .L_loop_tail:
31         DECH     x1, ALL, MUL #2
32         WHILELT   p4.b, x1, x3
33         B.NFRST   .L_return
34
35         LD1H    z0.h, p4/z, [x1, #0, mul vl]
36         LD1H    z1.h, p4/z, [x2, #0, mul vl]
37         SMLALB   z4.s, z0.h, z1.h
38         SMLALT   z5.s, z0.h, z1.h
39
```

```

40      .L_return:
41          ADD        z4.s, z4.s, z5.s
42          ADD        z6.s, z6.s, z7.s
43          ADD        z6.s, z6.s, z4.s
44          SADDV      d6, p2, z6.s
45          SQSHRNB    z6.h, z6.s, #16
46          STR        h6, [x0]
47          RET

```

All elements of the 1st data processing part predicate p2 are set active. The 2nd data processing part predicate p4 is carefully set by `WHILELT` instruction at line 27, comparing the current load pointer with the arrays' end memory byte address. Note that, even though the elements loaded from the memory are 16-bit, the predicate is enabled for 8-bit elements since the byte addresses are used for setting the predicate value in instruction `WHILELT`. The loop exit condition is checked by the conditional branch instruction at line 28.

One pointer per array is used for loading data. Observe the negative immediate offset used with the load of the 1st part of the first array (line 14). This is possible by the way of a specific addressing mode `[Xn, #imm, MUL VL]` available with the contiguous loads and stores. In this addressing mode the offset is an immediate constant multiplied by the vector's in-memory size. The immediate constant can be both positive and negative, in the range of -8 to 7. In the kernels requiring load with broadcast instruction with its `[Xn, #imm]` addressing mode, the immediate constant is byte offset and can hold only the positive values of a predefined range. Therefore, the pointers and offsets would have to be arranged differently than in this example.

With the instruction `INCH` at lines 24-25, the load pointers are updated in a vector length agnostic manner by the number of 16-bit elements in the vector length multiplied by 2 since two vector lengths of data are processed in the unrolled loop, and multiplied again by 2 since there are two bytes per 16-bit element. Notice that the array pointer `x1` is used both as the load pointer and as the loop counter.

This implementation requires specific loop termination step (starts at line 30) since in the 1st data processing part there may be remaining data to process.

Finally, at lines 41-44, the reductions are carried out to compute the final result from the partial sums.

A6.4 Two-level nested loop and loop interchange vectorization

Vector length agnostic vectorization of the two-level nested loops by the way of loop interchange is illustrated on the example of auto-correlation computation:

```

example_autocorr( short * in, short * out,
                  int N, int nb_delays, int scaling_factor) {
    int i, l;
    int product, acc;

    if (nb_delays>N) {
        nb_delays = N;
    }

    for (l=0; l<nb_delays; l++) {
        acc = 0;

        for (i=l; i<N; i++) {
            product = in[i] * in[i-l];
            acc += (product >> scaling_factor);
        }
        acc = (acc >> 16);
        out[j] = (short) acc;
    }
}

```

The natural approach to the vectorization of the above two-level nested loops is to vectorize the inner loop: computation of one output. Within the vectorized inner loop even and odd partial sums are calculated in each vector lane, and finally outside of the inner loop the output result is obtained by the reduction addition, and stored to the memory.

```

1      CMP      w2, #1      // N
2      B.MI     .L_return
3
4      CMP      w3, #1      // nb_delays
5      B.MI     .L_return
6
7      CMP      w3, w2
8      B.MI     .L_init
9      MOV      w3, w2
10
11     .L_init:
12     MOV      w2, w2
13     MOV      w3, w3
14     MOV      z4.s, w4     // scaling_factor
15
16     MOV      x9, #0
17     MOV      x10, x0
18     PTRUE    p0.s
19
20     .L_outer_loop_start:
21     MOV      x5, #0
22     MOV      z0.s, #0
23     MOV      z1.s, #0
24
25     WHILELT   p4.h, x5, x2
26
27     .L_inner_loop_start:
28     LD1H     z2.h, p4/Z, [x0, x5, LSL #1]

```

```

29      LD1H      z3.h, p4/Z, [x10, x5, LSL #1]
30      INCH      x5
31      SMULLB     z6.s,  z2.h,  z3.h
32      SMULLT     z7.s,  z2.h,  z3.h
33
34      ASR        z6.s, p0/M, z6.s, z4.s
35      ASR        z7.s, p0/M, z7.s, z4.s
36      ADD        z0.s, p0/M, z0.s, z6.s
37      ADD        z1.s, p0/M, z1.s, z7.s
38
39      WHILELT     p4.h, x5, x2
40      B.FIRST     .L_inner_loop_start
41
42      ADD        z0.s, p0/M, z0.s, z1.s
43      SADDV       d0, p0, z0.s
44      SSHR        d0, d0, #16
45      STR         h0, [x1, x9, LSL #1]
46
47      ADD        x9, x9, #1
48      SUB        x2, x2, #1
49      CMP        x9, x3
50      ADD        x10, x10, #2
51      B.MI        .L_outer_loop_start
52
53      .L_return:
54      RET

```

Alternatively, with the loop interchange approach, the outer loop is vectorized and multiple output results are calculated simultaneously in distinct vector lanes. This approach permits memory store of multiple results at once with one vector length contiguous store.

```

1      CMP        w3, #1      // nb_delays
2      B.MI        .L_return
3
4      MOV        w3, w3
5      MOV        z4.s, w4    // scaling_factor
6
7      CMP        w2, #1      // N
8      B.MI        .L_return
9
10     MOV        w2, w2
11     MOV        x9, #0
12     PTRUE       p0.s
13     WHILELT     p1.h, x9, x3
14
15     .L_outer_loop_start:
16     MOV        x12, x0
17     MOV        x14, x9
18     MOV        z0.s, #0
19     MOV        z1.s, #0
20     WHILELT     p2.h, x14, x2
21
22     .L_inner_loop_start:
23     LD1RH       z2.h, p2/Z, [x12]
24     LD1H        z3.h, p2/Z, [x0, x14, LSL #1]
25     ADD        x14, x14, #1
26     ADD        x12, x12, #2
27
28     SMULLB      z6.s, z2.h, z3.h
29     SMULLT      z7.s, z2.h, z3.h

```



```

30      ASR      z6.s, p0/M, z6.s, z4.s
31      ASR      z7.s, p0/M, z7.s, z4.s
32      ADD      z0.s, p0/M, z0.s, z6.s
33      ADD      z1.s, p0/M, z1.s, z7.s
34
35      WHILELT   p2.h, x14, x2
36      B.FIRST   .L_inner_loop_start
37
38      TRN2      z0.h, z0.h, z1.h
39      ST1H      z0.h, p1, [x1, x9, LSL #1]
40      INCH      x9
41      WHILELT   p1.h, x9, x3
42      B.FIRST   .L_outer_loop_start
43
44      .L_return:
45      RET

```

Within the inner loop, multiple output results are calculated at once, one per 32-bit vector lane (lines 22-36).

Two-level nested loops are optimized applying the two-dimensional loop control:

- The inner loop counter in `x14` varies with the value of the outer loop counter `x9` (line 17). The loop counters are re-used as load and store pointer offsets (lines 24, 39).
- The inner loop load predicate `p2` is determined based on the inner loop counter state (lines 20, 35). The outer loop store predicate `p1` is determined based on the outer loop counter state (line 41).

Outside of the inner loop, even and odd results are combined by the instruction `TRN2` at line 38, and at the same time high 16-bits are extracted from the 32-bit accumulation result. Finally, multiple output results are stored to the memory at once every outer loop iteration with one contiguous vector length store (line 39).

A6.5 Memory ordering example

The memory ordering is illustrated on the example of the following histogram function:

```
void example_histogram(uint32_t* histogram, uint32_t* records,
                      uint32_t num_records) {
    for (uint32_t i = 0; i < num_records; i++) {
        histogram[records[i]] += 1;
    }
}
```

In the vectorized implementation multiple `histogram[entry]` elements are loaded from the memory by one gather load instruction (line 8), then updated (line 10), and finally multiple updated `histogram[entry]` elements are written to the memory by one scatter store instruction (line 11).

1	MOV	w2, w2	// num_records
2	MOV	x4, #0	
3	WHILELT	p1.s, x4, x2	
4	B.NONE	.L_end	
5			
6	.Loop_start:		
7	LD1W	z1.s, p1/Z, [x1, x4, LSL #2]	
8	LD1W	z2.s, p1/Z, [x0, z1.s, UXTW #2]	
9	HISTCNT	z0.s, p1/Z, z1.s, z1.s	
10	ADD	z2.s, p1/M, z2.s, z0.s	
11	ST1W	z2.s, p1, [x0, z1.s, UXTW #2]	
12			
13	INCW	x4	
14	WHILELT	p1.s, x4, x2	
15	B.FIRST	.Loop_start	
16			
17	.L_end:		
18	RET		

Note that in this function the elements of `records` array are not necessarily distinct. Consequently, within one vectorized loop iteration and its scatter store, multiple updates of the same histogram entry can occur. The vectorization approach shown above exploits a memory ordering property of scattered vector stores where the element writes to the same location occur in element indices order.

Instruction `HISTCNT` at line 9 returns the number of matches up to each element. At line 10, the values of current histogram entries are incremented by the number of matches and written to the memory by a vector scatter store at line 11.

To illustrate the behavior when there are multiple updates of the same histogram entry within one vector length, let's say that by elements `k` and `k+step` the same histogram entry is updated. The instruction `HISTCNT` returns 1 for all active element locations, except for element `k+step` where it returns 2. The correct behavior is secured since within the vector scatter store first the element at position `k` is written to the memory, and then the element at position `k+step` is written.

Part B

Generic vector and matrix operations examples

Chapter B1

Vector maximum element

This chapter provides code examples of finding the maximum element of a vector.

B1.1 Code example: vector maximum with real 16-bit integer elements

The C code for finding the maximum element and its first location in a vector with real 16-bit integer elements, is shown below:

```
void vecmax_first( int16_t * src, uint16_t length, int16_t * ptrMaxElem,
                  uint16_t * ptrMaxIndex) {

    int16_t MaxVal = src[0];
    uint16_t MaxIndex = 0;

    for (uint16_t i=1; i<length; i++)
    {
        if (src[i] > MaxVal)
        {
            MaxVal = src[i];
            MaxIndex = i;
        }
    }

    *ptrMaxElem = MaxVal;
    *ptrMaxIndex = MaxIndex;
}
```

The following is the optimized SVE implementation:

```
1      UXTH      w1, w1
2      MOV      x8, #0
3      DUP      z14.h, #-1
4      WHILELT  p5.h, x8, x1
5      LD1H     z5.h, p5/Z, [x0]
6      INDEX    z10.h, #0, #1
7      INCH     x8
8      WHILELT  p4.h, x8, x1
9      B.NONE   .LoopEnd
10     MOV      z11.d, z10.d
11
12     .LoopStart:
13     LD1H     z6.h, p4/Z, [x0, x8, lsl #1]
14     INCH     z11.h
15     INCH     x8
16     CMPGT    p6.h, p4/Z, z6.h, z5.h
17     SMAX     z5.h, p4/M, z5.h, z6.h
18     SEL      z10.h, p6, z11.h, z10.h
19     WHILELT  p4.h, x8, x1
20     B.FIRST  .LoopStart
21
22     .LoopEnd:
23     SMAXV    h15, p5, z5.h
24     MOV      z6.h, h15
25     CMPEQ    p2.h, p5/Z, z5.h, z6.h
26     PTRUE    p0.h
27     SEL      z10.h, p2, z10.h, z14.h
28     UMINV    h12, p0, z10.h
29     STR      h15, [x2]
30     STR      h12, [x3]
31
32     RET
```

In the loop that starts at line 12, the comparisons are performed independently in each 16-bit vector register lane. When the loop exits, the identified per-lane maximum vector elements are available in 16-bit lanes of the vector register `z5`, while its indexes (the location within the input vector) are available in 16-bit lanes of the vector register `z10`.

Then, the maximum vector element is identified by performing signed maximum reduction across all 16-bit vector register lanes (instruction `SMAXV` at line 23). Another goal of this example is to determine the location of the maximum vector element. The maximum vector element can occur once, or more than once in the input vector. In this example, if the maximum element is present more than once, then the first occurrence of the maximum vector element is determined. With the `SEL` instruction at line 27, all indexes of the maximum vector element are kept in the vector register `z10`, while the indexes corresponding to other non-maximum vector elements are overwritten by the value `0xFFFF`. Finally, the first occurrence of the maximum vector element is obtained by performing the unsigned minimum reduction across indexes kept in 16-bit lanes of the vector register `z10` (instruction `UMINV` at line 28).

The equivalent SVE optimization steps are applicable to the Vector minimum element example by replacing adequately the instructions `CMPGT` (line 16), `SMAX` (line 17) and `SMAXV` (line 23) as necessary.

Chapter B2

Vector multiply

This chapter provides code example of element by element multiplication of two vectors.

B2.1 Code example: vector multiply with complex 16-bit integer elements

The C code for element-by-element multiplication of two vectors, with complex 16-bit input vector elements and complex 16-bit resulting vector elements, is shown below:

```

struct cplx_int16_t {
    int16_t re;
    int16_t im;
};

int16_t Sat(int32_t a) {
    int16_t b = (int16_t) a;
    if (a > MAX_INT16)    b = 0x7FFF;    // MAX_INT16 = 0x00007FFF
    if (a < MIN_INT16)    b = 0x8000;    // MIN_INT16 = 0xFFFF8000
    return b;
}

void vecmul(int64_t n, cplx_int16_t * a, cplx_int16_t * b, cplx_int16_t * c) {
    for (int64_t i=0; i<n; i++) {
        c[i].re = Sat(((int32_t)(a[i].re * b[i].re) +
                               (int32_t)0x4000)>>15) -
                    (((int32_t)(a[i].im * b[i].im) +
                               (int32_t)0x4000)>>15));
        c[i].im = Sat(((int32_t)(a[i].re * b[i].im) +
                               (int32_t)0x4000)>>15) +
                    (((int32_t)(a[i].im * b[i].re) +
                               (int32_t)0x4000)>>15));
    }
}

```

The optimized SVE2 implementation is taking advantage of the new powerful complex-valued integer arithmetic instruction `SQRDCMLAH`.

The vectorized loop is unrolled and two vector lengths of input elements are processed per loop iteration. For each input array, `a` and `b`, a separate pointer is defined for each of its two vectors processed within one loop iteration. All input and output pointers share the same offset `count`. For example, the pointer `aPtr_1st` with the offset `count` is used to load the first vector length elements group of input vector `a`, while the pointer `aPtr` with the offset `count` is used to load the second vector length elements group of input vector `a`. Similar applies to vector `b` loads and to the results store.

The `count` is re-used as a loop counter, and updated at line 43. In this way, the number of registers that need to be updated in each loop iteration is minimized. Only the second vector length elements load is truly predicated by `p4`. This works correctly since with the instruction `B.NFRST` at line 17 and the instruction `B.FIRST` at line 46, it is checked whether there are more than one vector length input elements remaining, and only in that case the vectorized loop starting at line 23 is entered. Otherwise, the termination code processing the remaining input elements of the first vector length group (lines 48-59) is executed.

1	size	.req	x0	//	int64_t n
2	aPtr	.req	x1	//	cplx_int16_t * a
3	bPtr	.req	x2	//	cplx_int16_t * b
4	outPtr	.req	x3	//	cplx_int16_t * c
5					
6	aPtr_1st	.req	x4		
7	bPtr_1st	.req	x5		
8	outPtr_1st	.req	x6		
9	count	.req	x7		
10					
11	PTRUE	p2.h			
12	LSL	size, size, #1			

Chapter B2. Vector multiply

B2.1. Code example: vector multiply with complex 16-bit integer elements

```

13      DUP          z31.s, #0
14
15      CNTH         count
16      WHILELT      p4.h, count, size
17      B.NFRST      .L_tail_vecmul
18
19      ADDVL         aPtr_1st, aPtr, #-1
20      ADDVL         bPtr_1st, bPtr, #-1
21      ADDVL         outPtr_1st, outPtr, #-1
22
23      .L_unrolled_loop_vecmul:
24      LD1H          z0.h, p2/z, [aPtr_1st, count, LSL #1]
25      LD1H          z2.h, p4/z, [aPtr, count, LSL #1]
26      LD1H          z1.h, p2/z, [bPtr_1st, count, LSL #1]
27      LD1H          z3.h, p4/z, [bPtr, count, LSL #1]
28
29      MOVPRFX       z4, z31
30      SQRDCMLAH     z4.h, z0.h, z1.h, #0
31      // c0.re += a0.re*b0.re | c0.im += a0.re*b0.im
32      SQRDCMLAH     z4.h, z0.h, z1.h, #90
33      // c0.re += -a0.im*b0.im | c0.im += a0.im*b0.re
34
35      MOVPRFX       z5, z31
36      SQRDCMLAH     z5.h, z2.h, z3.h, #0
37      // c1.re += a1.re*b1.re | c1.im += a1.re*b1.im
38      SQRDCMLAH     z5.h, z2.h, z3.h, #90
39      // c1.re += -a1.im*b1.im | c1.im += a1.im*b1.re
40
41      ST1H          z4.h, p2, [outPtr_1st, count, LSL #1]
42      ST1H          z5.h, p4, [outPtr, count, LSL #1]
43      INCH          count, ALL, MUL #2
44
45      WHILELT      p4.h, count, size
46      B.FIRST      .L_unrolled_loop_vecmul
47
48      .L_tail_vecmul:
49      DECH          count
50      WHILELT      p2.h, count, size
51      B.NFRST      .L_return_vecmul
52
53      LD1H          z0.h, p2/z, [aPtr, count, LSL #1]
54      LD1H          z1.h, p2/z, [bPtr, count, LSL #1]
55
56      MOVPRFX       z4, z31
57      SQRDCMLAH     z4.h, z0.h, z1.h, #0
58      SQRDCMLAH     z4.h, z0.h, z1.h, #90
59      ST1H          z4.h, p2, [outPtr, count, LSL #1]
60
61      .L_return_vecmul:
62      RET

```

The complex data is organized in a vector register with its real part in an even element, and its imaginary part in a subsequent odd element. The number of complex 16-bit elements in input vectors *a* and *b* is doubled at line 12 to correspond to the number of 16-bit real and imaginary sub-elements.

At line 29, the accumulating register is set to 0 using the instruction `MOVPRFX`. This instruction is a hint to the machine that it can be combined with the subsequent destructive instruction (line 30) to create a single constructive operation as described in [A3.1.3 Move prefix](#). One `SQRDCMLAH` instruction computes only a partial multiplication of two complex-valued integer data as described in [A5.2.2 Complex-valued multiply-accumulate](#), therefore two

SQRDCMLAH instructions are needed for the computation of a full complex multiplication (lines 30-33).

At line 30, there's first SQRDCMLAH instruction with rotation parameter #0. For two complex-valued inputs, $a = a_r + i * a_i$, $b = b_r + i * b_i$, the instruction SQRDCMLAH at line 30 produces the following result:

$$c_r+ = a_r * b_r, \quad c_i+ = a_r * b_i$$

At line 32, there's second SQRDCMLAH instruction with rotation parameter #90, that produces the accumulated result corresponding to the full complex-valued multiplication:

$$c_r+ = -a_i * b_i, \quad c_i+ = a_i * b_r$$

$$c+ = (a_r * b_r - a_i * b_i) + i * (a_r * b_i + a_i * b_r)$$

The same operations are performed for the second vector length group of elements (lines 35-39).

The two vector lengths results are stored at lines 41-42 similarly with only the second vector length results store truly predicated by p4.

Chapter B3

Vector dot-product

This chapter provides code examples of two vectors elements dot-product computation.

B3.1 Vector dot-product calculation

Vector dot-product is computed as:

$$c = \sum_{i=0}^{N-1} a[i] * b[i]$$

In the implementations that follow, it is assumed that the complex data is organized in a vector register with its real part in an even vector element, and its imaginary part in a subsequent odd vector element.

B3.2 Code example: vectors dot-product with complex SP floating-point elements and result

The C code for two vectors elements dot-product computation with complex SP floating-point input data and result, is shown below:

```

struct cplx_f32_t {
    float re;
    float im;
};

void vecdot(int64_t n, cplx_f32_t* a, cplx_f32_t* b, cplx_f32_t* c) {
    cplx_f32_t acc;
    acc.re = 0;
    acc.im = 0;

    for (int64_t i = 0; i < n; ++i) {
        acc.re += a[i].re * b[i].re - a[i].im * b[i].im;
        acc.im += a[i].re * b[i].im + a[i].im * b[i].re;
    }
    c->re = acc.re;
    c->im = acc.im;
}

```

The following is the optimized SVE implementation of two vectors elements dot-product computation with complex SP floating-point input data and result.

The vectorized loop is unrolled and two vector lengths of input elements are processed per loop iteration. Only the second vector length elements load is truly predicated (p4). This works correctly since with the instruction `B.NFRST` at line 13 and the instruction `B.FIRST` at line 29, it is checked whether there are more than one vector length input elements remaining, and only in that case the vectorized loop starting at line 15 is entered. Otherwise, the termination code processing the remaining input elements of the first vector length group is executed (lines 31-40).

1	size	.req	x0	//	N
2	aPtr	.req	x1	//	complex float32_t * a
3	bPtr	.req	x2	//	complex float32_t * b
4	outPtr	.req	x3	//	complex float32_t * c
5					
6	DUP		z8.d, #0		
7	DUP		z9.d, #0		
8	PTRUE		p2.s		
9					
10	ADD		size, aPtr, size, LSL #3		
11	INCB		aPtr		
12	WHILELT		p4.b, aPtr, size		
13	B.NFRST		.L_tail_vecdot		
14					
15	.L_unrolled_loop_vecdot:				
16	LD1W		z0.s, p2/z, [aPtr, #-1, MUL VL]		
17	LD1W		z1.s, p4/z, [aPtr]		
18	LD1W		z4.s, p2/z, [bPtr]		
19	LD1W		z5.s, p4/z, [bPtr, #1, MUL VL]		
20					
21	FCMLA		z8.s, p2/m, z0.s, z4.s, #0 // c0 += a0*b0		
22	FCMLA		z8.s, p2/m, z0.s, z4.s, #90		
23	FCMLA		z9.s, p2/m, z1.s, z5.s, #0 // c1 += a1*b1		
24	FCMLA		z9.s, p2/m, z1.s, z5.s, #90		

```

25
26         INCB          aPtr, ALL, MUL #2
27         INCB          bPtr, ALL, MUL #2
28         WHILELT       p4.b, aPtr, size
29         B.FIRST       .L_unrolled_loop_vecdot
30
31     .L_tail_vecdot:
32         DECB          aPtr
33         WHILELT       p4.b, aPtr, size
34         B.NFRST       .L_return_vecdot
35
36         LD1W          z3.s, p4/z, [aPtr]
37         LD1W          z7.s, p4/z, [bPtr]
38
39         FCMLA         z8.s, p4/m, z3.s, z7.s, #0
40         FCMLA         z8.s, p4/m, z3.s, z7.s, #90
41
42     .L_return_vecdot:
43         UZP1          z10.s, z8.s, z9.s
44         UZP2          z11.s, z8.s, z9.s
45         FADDV         s10, p2, z10.s
46         FADDV         s11, p2, z11.s
47         STP           s10, s11, [outPtr]
48
49         RET

```

As described in [A3.6.2 Complex-valued multiply-accumulate](#), one FCMLA instruction computes only a partial multiplication of two complex floating-point data, therefore two FCMLA instructions are needed for the computation of a full complex multiplication. At lines 6-7, the accumulating registers z8 and z9 are initialized to 0. At lines 21, 23, there's first FCMLA instruction with rotation parameter #0. For two complex-valued inputs, $a = a_r + i * a_i$, $b = b_r + i * b_i$, the first FCMLA instruction produces the following result:

$$c_r += a_r * b_r, \quad c_i += a_r * b_i$$

At lines 22, 24, there's second FCMLA instruction with rotation parameter #90, that produces the accumulated result corresponding to the full complex-valued multiplication:

$$c_r += -a_i * b_i, \quad c_i += a_i * b_r$$

Hence,

$$c += (a_r * b_r - a_i * b_i) + i * (a_r * b_i + a_i * b_r)$$

After completing the processing of the vectorized loop and the termination code, the partial vector dot-product complex SP floating-point results are available in each 64-bit lane of the vector registers z8 and z9, with the results real part and imaginary part interleaved. In order to compute the final two vectors elements dot-product result, it is necessary to perform the reduction addition. And more precisely, in order to compute the final result real part, it is necessary to perform the reduction addition of partial real part results only. And similarly, in order to compute the final result imaginary part, it is necessary to perform the reduction addition of partial imaginary part results only. Because of that, with the instructions UZP1 and UZP2 at lines 43-44, the real and imaginary parts of the partial sums are de-interleaved, with the partial real part sums moved to register z10, and the partial imaginary part sums moved to register z11. The reduction additions are computed with the instruction FADDV, for real part at line 45, and for imaginary part at line 46. A real-imaginary pair, i.e. one complex SP floating-point result is stored to the memory by instruction STP at line 47.

B3.3 Code example: vectors dot-product with real HP floating-point input elements and real SP floating-point result

The C code for two vectors elements dot-product computation with real HP floating-point input data and real SP floating-point result, is shown below:

```
void vecdot(int64_t n, float16_t* a, float16_t* b, float32_t* c) {
    float32_t acc = 0;

    for(int64_t i = 0; i < n; ++i) {
        acc += (float32_t)a[i] * (float32_t)b[i];
    }

    *c = acc;
}
```

The following is optimized SVE2 implementation of two vectors elements dot-product computation with real HP floating-point input data and real SP floating-point result.

The vectorized loop is unrolled and two vector lengths of input elements are processed per loop iteration. Only the second vector length elements load is truly predicated (p4). This works correctly since with the instruction `B.NFRST` at line 15 and the instruction `B.FIRST` at line 31, it is checked whether there are more than one vector length input elements remaining, and only in that case the vectorized loop starting at line 17 is entered. Otherwise, the termination code processing the remaining input elements of the first vector length group is executed (lines 33-42).

```
1      size      .req    x0      //    N
2      aPtr      .req    x1      //    float16_t * a
3      bPtr      .req    x2      //    float16_t * b
4      outPtr     .req    x3      //    float32_t * c
5
6      DUP       z4.d, #0
7      DUP       z5.d, #0
8      DUP       z6.d, #0
9      DUP       z7.d, #0
10
11     PTRUE      p2.h
12     ADD        size, aPtr, size, LSL #1
13     INCB       aPtr
14     WHILELT    p4.b, aPtr, size
15     B.NFRST    .L_tail_vecdot
16
17     .L_unrolled_loop_vecdot:
18     LD1H       z0.h, p2/z, [aPtr, #-1, MUL VL]
19     LD1H       z2.h, p4/z, [aPtr]
20     LD1H       z1.h, p2/z, [bPtr]
21     LD1H       z3.h, p4/z, [bPtr, #1, MUL VL]
22
23     FMLALB     z4.s, z0.h, z1.h
24     FMLALT     z5.s, z0.h, z1.h
25     FMLALB     z6.s, z2.h, z3.h
26     FMLALT     z7.s, z2.h, z3.h
27
28     INCB       aPtr, ALL, MUL #2
29     INCB       bPtr, ALL, MUL #2
30     WHILELT    p4.b, aPtr, size
31     B.FIRST    .L_unrolled_loop_vecdot
32
```

```

33      .L_tail_vecdot:
34          DECB          aPtr
35          WHILELT       p4.b, aPtr, size
36          B.NFRST       .L_return_vecdot
37
38          LD1H          z0.h, p4/z, [aPtr]
39          LD1H          z1.h, p4/z, [bPtr]
40
41          FMLALB        z4.s, z0.h, z1.h
42          FMLALT        z5.s, z0.h, z1.h
43
44      .L_return_vecdot:
45          FADD          z4.s, z4.s, z5.s
46          FADD          z6.s, z6.s, z7.s
47          FADD          z6.s, z6.s, z4.s
48          FADDV         s6, p2, z6.s
49          STR          s6, [outPtr]
50
51      RET

```

In order to minimize the precision loss when calculating dot-product of long vectors with HP floating-point elements, in this implementation the accumulation is performed into the SP floating-point accumulators. This is achieved with the new SVE2 instructions, bottom and top floating-point multiply-accumulate long instructions FMLALB and FMLALT (lines 23-26).

After the reductions (lines 45-47), the final dot-product result is obtained with the floating-point reduction addition instruction FADDV (line 48). The final dot-product SP floating-point result is stored to the memory at line 49.

B3.4 Code example: vectors dot-product with complex 16-bit integer elements and result

The C code for two vectors elements dot-product computation with complex 16-bit integer input data and result, is shown below:

```

struct cplx_int16_t {
    int16_t re;
    int16_t im;
};

struct cplx_int32_t {
    int32_t re;
    int32_t im;
};

void vecdot(int64_t n, cplx_int16_t* a, cplx_int16_t* b, cplx_int16_t* c) {
    cplx_int32_t acc;
    acc.re = 0;
    acc.im = 0;

    for (int64_t i = 0; i < n; ++i) {
        acc.re += (int32_t)((int64_t)a[i].re * (int64_t)b[i].re -
                           (int64_t)a[i].im * (int64_t)b[i].im);
        acc.im += (int32_t)((int64_t)a[i].re * (int64_t)b[i].im +
                           (int64_t)a[i].im * (int64_t)b[i].re);
    }

    c->re = (int16_t)(acc.re >> 15);
    c->im = (int16_t)(acc.im >> 15);
}

```

The following is optimized SVE2 implementation of two vectors elements dot-product computation with complex 16-bit integer input data and result.

The vectorized loop is unrolled and two vector lengths of input elements are processed per loop iteration. Only the second vector length elements load is truly predicated (p4). This works correctly since with the instruction `B.NFRST` at line 17 and the instruction `B.FIRST` at line 33, it is checked whether there are more than one vector length input elements remaining, and only in that case the vectorized loop starting at line 19 is entered. Otherwise, the termination code processing the remaining input elements of the first vector length group is executed (lines 35-44).

1	size	.req	x0	//	N
2	aPtr	.req	x1	//	complex int16_t * a
3	bPtr	.req	x2	//	complex int16_t * b
4	outPtr	.req	x3	//	complex int16_t * c
5					
6	DUP	z4.d, #0			
7	DUP	z5.d, #0			
8	DUP	z6.d, #0			
9	DUP	z7.d, #0			
10					
11	PTRUE	p2.h			
12	PTRUE	p1.h, VL1			
13	ADD	size, aPtr, size, LSL #2			
14					
15	INCB	aPtr			
16	WHILELT	p4.b, aPtr, size			

```

17      B.NFRST      .L_tail_vecdot
18
19      .L_unrolled_loop_vecdot:
20          LD1H      z0.h, p2/z, [aPtr, #-1, MUL VL]
21          LD1H      z2.h, p4/z, [aPtr]
22          LD1H      z1.h, p2/z, [bPtr]
23          LD1H      z3.h, p4/z, [bPtr, #1, MUL VL]
24
25          CDOT      z4.d, z0.h, z1.h, #0
26          CDOT      z5.d, z0.h, z1.h, #90
27          CDOT      z6.d, z2.h, z3.h, #0
28          CDOT      z7.d, z2.h, z3.h, #90
29
30          INCB      aPtr, ALL, MUL #2
31          INCB      bPtr, ALL, MUL #2
32          WHILELT   p4.b, aPtr, size
33          B.FIRST   .L_unrolled_loop_vecdot
34
35      .L_tail_vecdot:
36          DECB      aPtr
37          WHILELT   p4.b, aPtr, size
38          B.NFRST   .L_return_vecdot
39
40          LD1H      z0.h, p4/z, [aPtr]
41          LD1H      z1.h, p4/z, [bPtr]
42
43          CDOT      z4.d, z0.h, z1.h, #0
44          CDOT      z5.d, z0.h, z1.h, #90
45
46      .L_return_vecdot:
47          ADD       z4.d, z4.d, z6.d
48          ADD       z5.d, z5.d, z7.d
49          UADDV     d4, p2, z4.d
50          UADDV     d5, p2, z5.d
51          ASR       z4.d, z4.d, #15
52          ASR       z5.d, z5.d, #15
53          ST2H      {z4.h, z5.h}, p1, [outPtr]
54
55      RET

```

As described in [A5.2.3 Complex-valued integer dot product](#), two CDOT instructions are needed to compute the complex-valued dot-product with the result real and imaginary parts obtained in two different vector registers. At lines 6-9, the accumulating registers z4, z5, z6, z7 are initialized to 0. For complex-valued inputs a0 and a1 of the first source vector register, and b0 and b1 of the second source vector register, the CDOT instructions with the rotation parameter set to #0 (lines 25, 27) accumulate the real part of the result:

$$c_r + = a0_r * b0_r - a0_i * b0_i + a1_r * b1_r - a1_i * b1_i$$

The CDOT instructions with the rotation parameter set to #90 (lines 26, 28) accumulate the imaginary part of the result:

$$c_i + = a0_r * b0_i + a0_i * b0_r + a1_r * b1_i + a1_i * b1_r$$

In CDOT instructions (lines 25-28) input elements are complex 16-bit integer data, and the computed partial results are 64-bit integer real part or imaginary part of the result.

After completing the processing of the vectorized loop and the termination code, and after the reductions (lines 47-48), the 64-bit partial real-part results are located in the lanes of vector register z4, and the 64-bit partial imaginary-part results are located in the lanes of vector register z5. The final real-part result is obtained by the reduction addition instruction UADDV at line 49. The final imaginary-part result is obtained by the reduction

addition instruction `UADDV` at line 50.

Finally, after the adequate shift right at lines 51-52, the 16-bit real and imaginary part of the final result are interleaved before the store to the memory with instruction `ST2H` at line 53. Only one complex 16-bit integer result is stored to the memory as controlled by the predicate register `p1` which was set at line 12.

Chapter B4

FIR filter

This chapter provides code examples of FIR filtering computation.

B4.1 FIR filtering theory and C implementation

The implementable FIR filtering is presented by:

$$y[n] = \sum_{t=0}^{T-1} h[t] * x[n + T - 1 - t]$$

The following C code implements the FIR filtering as:

$$y[n] = \sum_{t'=0}^{T-1} h'[t'] * x[n + t']$$

under the assumption that the filter coefficients are organized in memory in the reversed order:

$$h'[0] = h[T - 1], \dots, h'[T - 1] = h[0].$$

```
#ifdef FLOAT
float32_t acc;
#else
int32_t acc;
#endif

for( int64_t i=0 ; i<n ; i++)
{
    acc = 0;

    for( int64_t j=0 ; j<t ; j++) {
        acc += x[i+j] * h[j];
    }
    #ifdef FLOAT
    y[i] = acc;
    #else
    y[i] = acc >> 16;
    #endif
}
```

B4.2 Code example: FIR filtering with real SP floating-point elements

The following is the optimized SVE implementation of FIR filtering with the real SP floating-point input data, result and FIR filter coefficients.

The outer loop starting at line 13 iterates over the input data array length. The inner loop starting at line 22 iterates over the filter taps producing vector length results (machine vector length/SP floating-point size). The FIR filtering computation is implemented with the SP floating-point multiply `FMUL` instruction (line 20) and the SP floating-point multiply-accumulate `FMLA` instruction (line 29).

```

1      size      .req    x0          //    int32_t n
2      taps      .req    x1          //    int32_t t
3      xPtr      .req    x2          //    float32_t * x
4      hPtr      .req    x3          //    float32_t * h
5      yPtr      .req    x4          //    float32_t * y
6
7      ADD        size, yPtr, size, LSL #2
8      WHILELT    p4.b, yPtr, size
9      ADD        taps, hPtr, taps, LSL #2
10     PTRUE      p5.s
11     B.NONE     .L_return
12
13     .L_FIR_outer_loop:
14         MOV      x6, #0
15         MOV      x8, hPtr
16         LD1W     z2.s, p4/z, [xPtr, x6, LSL #2]
17         LD1RW    z1.s, p5/z, [x8]
18         ADD      x6, x6, #1
19         ADD      x8, x8, #4
20         FMUL     z10.s, z2.s, z1.s
21
22     .L_FIR_inner_loop:
23         LD1W     z2.s, p4/z, [xPtr, x6, LSL #2]
24         // contiguous load of input data
25         LD1RW    z1.s, p5/z, [x8]
26         // load with broadcast of one FIR filter coefficient
27         ADD      x6, x6, #1
28         ADD      x8, x8, #4
29         FMLA     z10.s, p5/m, z2.s, z1.s
30         CMP      x8, taps
31         B.MI     .L_FIR_inner_loop
32
33         ST1W     z10.s, p4, [yPtr]
34         INCB     yPtr
35         ADDVL    xPtr, xPtr, #1
36         WHILELT    p4.b, yPtr, size
37         B.FIRST  .L_FIR_outer_loop
38
39     .L_return:
40         RET

```

B4.3 Code example: FIR filtering with real-valued 16-bit integer elements

The following is the optimized SVE2 implementation of FIR filtering with the real integer input data, result and FIR filter coefficients.

The outer loop starting at line 13 iterates over the input data array length. The inner loop starting at line 23 iterates over the filter taps producing vector length results (machine vector length/int16_t size). The FIR filtering computation is implemented with the top-bottom widening integer multiply `SMULLB` and `SMULLT` instructions (lines 20-21) and the top-bottom widening integer multiply-accumulate `SMLALB` and `SMLALT` instructions (lines 30-31) described in [A5.1.1 Top-Bottom processing approach](#) and [A5.1.2 Top-Bottom widening instructions](#). The instructions at lines 20-21 and 30-31, take as source elements 16-bit integer data and produce 32-bit integer results, hence maintaining the full precision. Since the result elements are twice as large as the source elements, to process all input data previously loaded, two instructions are needed: one bottom (lines 20, 30) and one top (lines 21, 31) instruction.

The final FIR filtering 16-bit integer results are obtained by extracting the high 16-bits out of the FIR filtering 32-bit integer results by bottom `SQSHRNB` (line 35) and top `SQSHRNT` (line 36) instructions. With the pair of instructions `SQSHRNB` and `SQSHRNT` (lines 35-36) also achieved is the interleaving of even and odd results, recovering the original input data sequence order before storing them to the memory at line 37.

```

1      size      .req    x0      //    int64_t n
2      taps      .req    x1      //    int64_t t
3      xPtr      .req    x2      //    int16_t * x
4      hPtr      .req    x3      //    int16_t * h
5      yPtr      .req    x4      //    int16_t * y
6
7      ADD        size, yPtr, size, LSL #1
8      WHILELT    p4.b, yPtr, size
9      ADD        taps, hPtr, taps, LSL #1
10     PTRUE      p5.h
11     B.NONE     .L_return
12
13     .L_FIR_outer_loop:
14         MOV      x6, #0
15         MOV      x8, hPtr
16         LD1H     z2.h, p4/z, [xPtr, x6, LSL #1]
17         LD1RH    z1.h, p5/z, [x8]
18         ADD      x6, x6, #1
19         ADD      x8, x8, #2
20         SMULLB   z10.s, z2.h, z1.h
21         SMULLT   z11.s, z2.h, z1.h
22
23     .L_FIR_inner_loop:
24         LD1H     z2.h, p4/z, [xPtr, x6, LSL #1]
25         // contiguous load of input data
26         LD1RH    z1.h, p5/z, [x8]
27         // load with broadcast of one FIR filter coefficient
28         ADD      x6, x6, #1
29         ADD      x8, x8, #2
30         SMLALB   z10.s, z2.h, z1.h
31         SMLALT   z11.s, z2.h, z1.h
32         CMP      x8, taps
33         B.MI     .L_FIR_inner_loop
34
35         SQSHRNB  z10.h, z10.s, #16
36         SQSHRNT  z10.h, z11.s, #16
37         ST1H     z10.h, p4, [yPtr]
38         INCB     yPtr
39         ADDVL    xPtr, xPtr, #1

```

Chapter B4. FIR filter

B4.3. Code example: FIR filtering with real-valued 16-bit integer elements

```
40          WHILELT      p4.b, yPtr, size
41          B.FIRST      .L_FIR_outer_loop
42
43      .L_return:
44          RET
```

Chapter B5

Matrix multiply

This chapter provides code examples of matrix multiplication.

B5.1 Matrix multiplication with real DP floating-point elements

The DP floating-point matrix multiplication $out[M, N] = inLeft[M, K] * inRight[K, N]$ is implemented with no previous rearrangement of input matrices, and it is illustrated by the following C code:

```
void matmul_f64_C( uint64_t M, uint64_t K, uint64_t N,
                  float64_t * inLeft, float64_t * inRight, float64_t * out) {
    uint64_t  x, y, z;
    float64_t acc;

    for (x=0; x<M; x++) {
        for (y=0; y<N; y++) {
            acc = 0.0;

            for (z=0; z<K; z++) {
                acc += inLeft[x*K + z] * inRight[z*N + y];
            }

            out[x*N + y] = acc;
        }
    }
}
```

Note that it's assumed that the input and output matrices are organized in memory in row-major order.

B5.1.1 Code example

The following SVE vectorization assumptions are taken:

- Minimum matrix dimension is 32.
- Matrix dimensions are multiple of 16.

The real DP floating-point matrix multiplication function is written in C with the Arm C language extensions (ACLE) for SVE [5]. The following is vector length agnostic implementation, for the vector lengths that are power of 2 and up to 1024-bits supported.

```
1  void matmul_f64( uint64_t M, uint64_t K, uint64_t N,
2                  float64_t * inLeft, float64_t * inRight, float64_t * out) {
3      uint64_t      x, y, z;
4      svbool_t      p64_all = svptrue_b64();
5      uint64_t      vl = svcntd();
6      uint64_t      offsetIN_1, offsetIN_2, offsetIN_3;
7      uint64_t      offsetOUT_1, offsetOUT_2, offsetOUT_3;
8
9      float64_t      *ptrIN_left;
10     float64_t      *ptrIN_right;
11     float64_t      *ptrOUT;
12
13     svfloat64_t      acc0, acc1, acc2, acc3;
14     svfloat64_t      inR_0, inR_1;
15     svfloat64_t      inL_0, inL_1, inL_2, inL_3;
16
17     offsetIN_1 = K;
18     offsetIN_2 = 2*K;
19     offsetIN_3 = 3*K;
20
21     offsetOUT_1 = N;
22     offsetOUT_2 = 2*N;
```

```

23         offsetOUT_3 = 3*N;
24
25     for (x=0; x<M; x+=4) {
26         ptrOUT = &out[x*N];
27
28         for (y=0; y<N; y+=vl) {
29             acc0 = svdup_f64(0.0);
30             acc1 = svdup_f64(0.0);
31             acc2 = svdup_f64(0.0);
32             acc3 = svdup_f64(0.0);
33
34             ptrIN_left = &inLeft[x*K];
35             ptrIN_right = &inRight[y];
36
37             for (z=0; z<K; z+=2) {
38                 inR_0 = svld1(p64_all, ptrIN_right);
39                 inR_1 = svld1(p64_all, &ptrIN_right[offsetOUT_1]);
40
41                 inL_0 = svld1rq(p64_all, ptrIN_left);
42                 inL_1 = svld1rq(p64_all, &ptrIN_left[offsetIN_1]);
43                 inL_2 = svld1rq(p64_all, &ptrIN_left[offsetIN_2]);
44                 inL_3 = svld1rq(p64_all, &ptrIN_left[offsetIN_3]);
45
46                 acc0 = svmla_lane(acc0, inR_0, inL_0, 0);
47                 acc0 = svmla_lane(acc0, inR_1, inL_0, 1);
48
49                 acc1 = svmla_lane(acc1, inR_0, inL_1, 0);
50                 acc1 = svmla_lane(acc1, inR_1, inL_1, 1);
51
52                 acc2 = svmla_lane(acc2, inR_0, inL_2, 0);
53                 acc2 = svmla_lane(acc2, inR_1, inL_2, 1);
54
55                 acc3 = svmla_lane(acc3, inR_0, inL_3, 0);
56                 acc3 = svmla_lane(acc3, inR_1, inL_3, 1);
57
58                 ptrIN_right += 2*N;
59                 ptrIN_left += 2;
60             }
61
62             svst1(p64_all, ptrOUT, acc0);
63             svst1(p64_all, &ptrOUT[offsetOUT_1], acc1);
64             svst1(p64_all, &ptrOUT[offsetOUT_2], acc2);
65             svst1(p64_all, &ptrOUT[offsetOUT_3], acc3);
66
67             ptrOUT += vl;
68         }
69     }
70 }

```

The SVE vectorization is implemented by unrolling:

- The outer loop by factor 4.
- The middle loop by factor vl (number of 64-bit elements in a machine vector length).
- The inner loop by factor 2.

The most inner loop at line 37 produces results of a subblock $out[4, vl] = inLeft[4, K] * inRight[K, vl]$. The computation is implemented using the floating-point multiply-add FMLA by indexed elements instruction (lines 46-56). The indexed elements are left input matrix elements which are effectively loaded (two elements per one vector load) and replicated by instruction LD1RQD (lines 41-44). Right input matrix elements are loaded by

contiguous vector length load `LD1D` (lines 38-39). At the most inner loop completion subblock `out[4, vl]` results are stored to the memory by contiguous vector length store `ST1D` (lines 62-65).

B5.2 Matrix multiplication with real HP floating-point elements

The HP floating-point matrix multiplication $out[M, N] = inLeft[M, K] * inRight[K, N]$ is implemented in two steps. In the first step the left matrix is rearranged in a specific manner tailored to the data processing of the second step. In the second step, dot-product calculations are performed to produce the final matrix multiplication results.

In the first step, the entire left matrix is rearranged such that each block of 8 full rows is transposed, as illustrated by the following C code:

```
void rearrangeLeft_fp16_C( uint64_t M, uint64_t K,
                          float16_t * inLeft, float16_t * inLeft_MOD) {
    uint64_t    x, y, z;

    float16_t   *ptr_in;
    float16_t   *ptr_out;

    for (x=0; x<M; x+=8) {
        ptr_in = &inLeft[x*K];
        ptr_out = &inLeft_MOD[x*K];

        for (y=0; y<K; y++) {
            for (z=0; z<8; z++) {
                *ptr_out = ptr_in[z*K+y];
                ptr_out++;
            }
        }
    }
}
```

In the second step, matrix multiplication results are obtained by performing dot-product calculations on:

- The elements of rearranged left input matrix.
- The elements of right input matrix.

This is illustrated by the following C code:

```
void matmul_dotp_fp16_C( uint64_t M, uint64_t K, uint64_t N,
                        float16_t * inLeft_MOD, float16_t * inRight, float16_t * out) {
    uint64_t    x, y, z;
    float16_t   acc;

    float16_t   *ptrIN_left;

    uint64_t    vl = svcnth();

    for (x=0; x<M; x++) {
        ptrIN_left = &inLeft_MOD[((x/8)*8)*K + (x%8)];

        for (y=0; y<N; y++) {
            acc = 0.0;

            for (z=0; z<K; z++) {
                acc += ptrIN_left[8*z] * inRight[z*N + y];
            }

            out[x*N + y] = acc;
        }
    }
}
```

It is assumed that all matrices are organized in memory in row-major order.

B5.2.1 Code example

The following SVE vectorization assumptions are taken:

- Minimum matrix dimension is 64.
- Matrix dimensions are multiple of 16.

The real HP floating-point matrix multiplication functions are written in C with the Arm C language extensions (ACLE) for SVE [5]. The implementations are vector length agnostic, for the vector lengths that are power of 2 and up to 1024-bits supported.

The following is the SVE implementation of left matrix rearrangement:

```

1      void rearrangeLeft_fp16( uint64_t M, uint64_t K,
2                               float16_t * inLeft, float16_t * inLeft_MOD) {
3          uint64_t      x, y, nb_st_elems, init_nb_elems;
4          svbool_t      p_ld;
5          svfloat16_t    r0, r1, r2, r3, r4, r5, r6, r7;
6          uint64_t      offsetIN_1, offsetIN_2, offsetIN_3, offsetIN_4;
7          uint64_t      offsetIN_5, offsetIN_6, offsetIN_7;
8
9          float16_t      *ptrIN;
10         float16_t      *ptrOUT;
11
12         uint64_t        vl = svcnth();
13         svbool_t        p16_all = svptrue_b16();
14
15         offsetIN_1 = K;
16         offsetIN_2 = 2*K;
17         offsetIN_3 = 3*K;
18         offsetIN_4 = 4*K;
19         offsetIN_5 = 5*K;
20         offsetIN_6 = 6*K;
21         offsetIN_7 = 7*K;
22
23         init_nb_elems = 8*K/vl;
24
25         for (x=0; x<M; x+=8) {
26             ptrIN = &inLeft[x*K];
27             ptrOUT = &inLeft_MOD[x*K];
28
29             nb_st_elems = init_nb_elems;
30
31             for (y=0; svptest_first( p16_all, p_ld = svwhilelt_b16(y, K)); y+=vl) {
32
33                 r0 = svld1(p_ld, ptrIN);
34                 r1 = svld1(p_ld, &ptrIN[offsetIN_1]);
35                 r2 = svld1(p_ld, &ptrIN[offsetIN_2]);
36                 r3 = svld1(p_ld, &ptrIN[offsetIN_3]);
37                 r4 = svld1(p_ld, &ptrIN[offsetIN_4]);
38                 r5 = svld1(p_ld, &ptrIN[offsetIN_5]);
39                 r6 = svld1(p_ld, &ptrIN[offsetIN_6]);
40                 r7 = svld1(p_ld, &ptrIN[offsetIN_7]);
41
42                 svfloat16_t t8 = svzip1(r0, r4);
43                 svfloat16_t t9 = svzip1(r2, r6);
44                 svfloat16_t t10 = svzip1(r1, r5);
45                 svfloat16_t t11 = svzip1(r3, r7);

```

```

46         svfloat16_t t12 = svzip2(r0, r4);
47         svfloat16_t t13 = svzip2(r2, r6);
48         svfloat16_t t14 = svzip2(r1, r5);
49         svfloat16_t t15 = svzip2(r3, r7);
50
51         svfloat16_t t16 = svzip1(t8, t9);
52         svfloat16_t t17 = svzip1(t10, t11);
53         svfloat16_t t18 = svzip2(t8, t9);
54         svfloat16_t t19 = svzip2(t10, t11);
55         r0 = svzip1(t16, t17);
56         r1 = svzip2(t16, t17);
57         r2 = svzip1(t18, t19);
58         r3 = svzip2(t18, t19);
59
60         t16 = svzip1(t12, t13);
61         t17 = svzip1(t14, t15);
62         t18 = svzip2(t12, t13);
63         t19 = svzip2(t14, t15);
64         r4 = svzip1(t16, t17);
65         r5 = svzip2(t16, t17);
66         r6 = svzip1(t18, t19);
67         r7 = svzip2(t18, t19);
68
69         switch(nb_st_elems) {
70             default :
71                 svst1_vnum(p16_all, ptrOUT, 7, r7);
72                 svst1_vnum(p16_all, ptrOUT, 6, r6);
73                 svst1_vnum(p16_all, ptrOUT, 5, r5);
74                 svst1_vnum(p16_all, ptrOUT, 4, r4);
75             case 4 :
76                 svst1_vnum(p16_all, ptrOUT, 3, r3);
77                 svst1_vnum(p16_all, ptrOUT, 2, r2);
78             case 2 :
79                 svst1_vnum(p16_all, ptrOUT, 1, r1);
80                 svst1(p16_all, ptrOUT, r0);
81         }
82
83         ptrIN += vl;
84         ptrOUT += 8*vl;
85         nb_st_elems -= 8;
86     }
87 }
88

```

The transpose of 8 matrix rows is implemented using the ZIP1 and ZIP2 instructions in 3 steps (lines 42-67).

Since matrix dimensions are multiples of 16 and element size is 16-bit, in the case of vector lengths higher than 256-bit the vector loads of the inner loop's last iteration may have inactive lanes. Therefore, the rearranged matrix elements stores to the memory are managed by the `switch` statement (lines 69-81).

The following is SVE implementation of matrix multiplication results computation:

```

1     void matmul_dotp_fp16( uint64_t M, uint64_t K, uint64_t N,
2                           float16_t * inLeft_MOD, float16_t * inRight, float16_t * out) {
3         uint64_t      x, y, z;
4         svfloat16_t inR, inL;
5         svfloat16_t acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7;
6         svbool_t      p_ld_st;
7         uint64_t      offsetOUT_1, offsetOUT_2, offsetOUT_3, offsetOUT_4;
8         uint64_t      offsetOUT_5, offsetOUT_6, offsetOUT_7;

```

```

9
10     svbool_t    p16_all = svptrue_b16();
11     uint64_t    vl = svcnth();
12
13     float16_t    *ptrIN_left;
14     float16_t    *ptrIN_right;
15     float16_t    *ptrOUT;
16
17     offsetOUT_1 = N;
18     offsetOUT_2 = 2*N;
19     offsetOUT_3 = 3*N;
20     offsetOUT_4 = 4*N;
21     offsetOUT_5 = 5*N;
22     offsetOUT_6 = 6*N;
23     offsetOUT_7 = 7*N;
24
25     for (x=0; x<M; x+=8) {
26         ptrOUT = &out[x*N];
27
28         for(y=0; svptest_first( p16_all, p_ld_st = svwhilelt_b16(y, N)); y+=vl)
29         {
30             ptrIN_left = &inLeft_MOD[x*K];
31             ptrIN_right = &inRight[y];
32
33             acc0 = svdup_f16(0.0);
34             acc1 = svdup_f16(0.0);
35             acc2 = svdup_f16(0.0);
36             acc3 = svdup_f16(0.0);
37             acc4 = svdup_f16(0.0);
38             acc5 = svdup_f16(0.0);
39             acc6 = svdup_f16(0.0);
40             acc7 = svdup_f16(0.0);
41
42             for (z=0; z<K; z++) {
43                 inR = svld1(p_ld_st, &ptrIN_right[z*N]);
44                 inL = svldlrq(p16_all, &ptrIN_left[8*z]);
45
46                 acc0 = svmla_lane(acc0, inR, inL, 0);
47                 acc1 = svmla_lane(acc1, inR, inL, 1);
48                 acc2 = svmla_lane(acc2, inR, inL, 2);
49                 acc3 = svmla_lane(acc3, inR, inL, 3);
50                 acc4 = svmla_lane(acc4, inR, inL, 4);
51                 acc5 = svmla_lane(acc5, inR, inL, 5);
52                 acc6 = svmla_lane(acc6, inR, inL, 6);
53                 acc7 = svmla_lane(acc7, inR, inL, 7);
54             }
55
56             svst1(p_ld_st, ptrOUT, acc0);
57             svst1(p_ld_st, &ptrOUT[offsetOUT_1], acc1);
58             svst1(p_ld_st, &ptrOUT[offsetOUT_2], acc2);
59             svst1(p_ld_st, &ptrOUT[offsetOUT_3], acc3);
60             svst1(p_ld_st, &ptrOUT[offsetOUT_4], acc4);
61             svst1(p_ld_st, &ptrOUT[offsetOUT_5], acc5);
62             svst1(p_ld_st, &ptrOUT[offsetOUT_6], acc6);
63             svst1(p_ld_st, &ptrOUT[offsetOUT_7], acc7);
64
65             ptrOUT += vl;
66         }
67     }
68 }

```

The SVE vectorization is implemented by unrolling:

- The outer loop by factor 8.
- The middle loop by factor vl (number of 16-bit elements in a machine vector length).

The most inner loop at line 42 produces results of a subblock $out[8, vl] = inLeft[8, K] * inRight[K, vl]$. The computation is implemented using the floating-point multiply-add `FMLA` by indexed elements instruction (lines 46-53). The indexed elements are left input matrix elements which are effectively loaded (eight elements per one vector load) and replicated by instruction `LD1RQH` (line 44). Right input matrix elements are loaded by contiguous vector length load `LD1H` at line 43. At the most inner loop completion subblock $out[8, vl]$ results are stored to the memory by contiguous vector length store `ST1H` (lines 56-63).

Since matrix dimensions are multiple of 16 and element size is 16-bit, to accommodate vector lengths higher than 256-bit the contiguous vector loads of right matrix elements (line 43) and the contiguous vector stores of resulting matrix elements (lines 56-63) are carefully predicated by the predicate (`p_ld_st`) set at line 28.

B5.3 Matrix multiplication with real 8-bit integer input elements and real 32-bit integer output elements

The matrix multiplication with real 8-bit integer input elements and real 32-bit integer output elements $out[M, N] = inLeft[M, K] * inRight[K, N]$ is implemented in two steps. In the first step both left and right matrices are rearranged in a specific manner tailored to the data processing of the second step. In the second step, dot-product calculations are performed to produce the final matrix multiplication results.

In the first step, the entire left matrix is rearranged such that:

- All matrix elements are grouped into sets of 4 elements.
- Each block of 8 full rows of sets is transposed.

This is illustrated by the following C code:

```
void rearrangeLeft_fixp_C( uint64_t M, uint64_t K,
                          uint8_t * inLeft, uint8_t * inLeft_MOD) {
    uint64_t x, y, z, w;
    uint8_t  *ptr_in;
    uint8_t  *ptr_out;

    for (x=0; x<M; x+=8) {
        ptr_out = &inLeft_MOD[x*K];

        for (y=0; y<K; y+=4) {
            ptr_in = &inLeft[x*K+y];

            for (z=0; z<8; z++) {
                for (w=0; w<4; w++) {
                    *ptr_out = ptr_in[z*K+w];
                    ptr_out++;
                }
            }
        }
    }
}
```

In the first step, the entire right matrix is rearranged by dividing it into subblocks of size $[4, vl/4]$, where vl is the number of 8-bit elements in a machine vector length. Then each subblock is transposed, and stored as illustrated by the following C code:

```
void rearrangeRight_fixp_C( uint64_t K, uint64_t N,
                           uint8_t * inRight, uint8_t * inRight_MOD) {
    uint64_t x, y, z, w;

    uint8_t  *ptr_in;
    uint8_t  *ptr_out;

    uint64_t vl_4 = svcntw();    // svcntb()>>2

    for (y=0; y<N; y+=vl_4) {
        ptr_out = &inRight_MOD[y*K];

        for (x=0; x<K; x+=4) {
            ptr_in = &inRight[x*N+y];

            for (z=0; z<vl_4; z++) {
                for (w=0; w<4; w++) {
                    *ptr_out = ptr_in[w*N+z];
                }
            }
        }
    }
}
```

```

        ptr_out++;
    }
}
}
}
}

```

In the second step, matrix multiplication results are obtained by performing dot-product calculations on:

- The elements of rearranged left input matrix.
- The elements of rearranged right input matrix.

This is illustrated by the following C code:

```

void matmul_dotp_fixp_C( uint64_t M, uint64_t K, uint64_t N,
                        uint8_t * inLeft_MOD, uint8_t * inRight_MOD, uint32_t * out) {
    uint64_t  x, y, z;
    uint32_t  acc;

    uint8_t  *ptrIN_left;
    uint8_t  *ptrIN_right;

    uint64_t vl_4 = svcntw(); // svcntb()>>2

    for (x=0; x<M; x++) {
        ptrIN_left = &inLeft_MOD[(x/8)*8*K];

        for (y=0; y<N; y++) {
            ptrIN_right = &inRight_MOD[(y/vl_4)*vl_4*K];
            acc = 0;

            for (z=0; z<K; z++) {
                acc += ptrIN_left[(z/4)*4*8 + (x%8)*4 + (z%4)] *
                    ptrIN_right[(z/4)*4*vl_4 + (y%vl_4)*4 + (z%4)];
            }

            out[x*N + y] = acc;
        }
    }
}

```

It is assumed that all matrices are organized in memory in row-major order.

B5.3.1 Code example

The following SVE vectorization assumptions are taken:

- Minimum matrix dimension is 128.
- Matrix dimensions are multiple of 32.

The matrix multiplication with real 8-bit integer input elements and real 32-bit output elements functions are written in C with the Arm C language extensions (ACLE) for SVE [5]. The implementations are vector length agnostic, for the vector lengths that are power of 2 and up to 1024-bits supported.

The following is SVE implementation of left matrix rearrangement:

```

1 void rearrangeLeft_fixp( uint64_t M, uint64_t K,
2                        uint8_t * inLeft, uint8_t * inLeft_MOD) {
3     uint64_t  x, y, nb_st_elems, init_nb_elems;
4     svbool_t  p_ld;

```

```

5      uint64_t      offsetIN_1, offsetIN_2, offsetIN_3, offsetIN_4;
6      uint64_t      offsetIN_5, offsetIN_6, offsetIN_7;
7
8      svuint8_t      r0, r1, r2, r3, r4, r5, r6, r7;
9      svuint32_t      r00, r11, r22, r33, r44, r55, r66, r77;
10     svuint32_t      t8, t9, t10, t11, t12, t13, t14, t15, t16, t17, t18, t19;
11
12     uint8_t         *ptrIN;
13     uint8_t         *ptrOUT;
14
15     uint64_t         vl = svcntb();
16     svbool_t         p8_all = svptrue_b8();
17
18     offsetIN_1 = K;
19     offsetIN_2 = 2*K;
20     offsetIN_3 = 3*K;
21     offsetIN_4 = 4*K;
22     offsetIN_5 = 5*K;
23     offsetIN_6 = 6*K;
24     offsetIN_7 = 7*K;
25
26     init_nb_elems = 8*K/vl;
27
28     for (x=0; x<M; x+=8) {
29         ptrIN = &inLeft[x*K];
30         ptrOUT = &inLeft_MOD[x*K];
31
32         nb_st_elems = init_nb_elems;
33
34         for (y=0; svptest_first( p8_all, p_ld = svwhilelt_b8(y, K)); y+=vl) {
35             r0 = svld1(p_ld, ptrIN);
36             r1 = svld1(p_ld, &ptrIN[offsetIN_1]);
37             r2 = svld1(p_ld, &ptrIN[offsetIN_2]);
38             r3 = svld1(p_ld, &ptrIN[offsetIN_3]);
39             r4 = svld1(p_ld, &ptrIN[offsetIN_4]);
40             r5 = svld1(p_ld, &ptrIN[offsetIN_5]);
41             r6 = svld1(p_ld, &ptrIN[offsetIN_6]);
42             r7 = svld1(p_ld, &ptrIN[offsetIN_7]);
43
44             t8 = svzip1(svreinterpret_u32(r0), svreinterpret_u32(r4));
45             t9 = svzip1(svreinterpret_u32(r2), svreinterpret_u32(r6));
46             t10 = svzip1(svreinterpret_u32(r1), svreinterpret_u32(r5));
47             t11 = svzip1(svreinterpret_u32(r3), svreinterpret_u32(r7));
48             t12 = svzip2(svreinterpret_u32(r0), svreinterpret_u32(r4));
49             t13 = svzip2(svreinterpret_u32(r2), svreinterpret_u32(r6));
50             t14 = svzip2(svreinterpret_u32(r1), svreinterpret_u32(r5));
51             t15 = svzip2(svreinterpret_u32(r3), svreinterpret_u32(r7));
52
53             t16 = svzip1(t8, t9);
54             t17 = svzip1(t10, t11);
55             t18 = svzip2(t8, t9);
56             t19 = svzip2(t10, t11);
57             r00 = svzip1(t16, t17);
58             r11 = svzip2(t16, t17);
59             r22 = svzip1(t18, t19);
60             r33 = svzip2(t18, t19);
61
62             t16 = svzip1(t12, t13);
63             t17 = svzip1(t14, t15);
64             t18 = svzip2(t12, t13);

```

```

65         t19 = svzip2(t14, t15);
66         r44 = svzip1(t16, t17);
67         r55 = svzip2(t16, t17);
68         r66 = svzip1(t18, t19);
69         r77 = svzip2(t18, t19);
70
71         switch(nb_st_elems) {
72             default :
73                 svstl_vnum(p8_all, ptrOUT, 7, svreinterpret_u8(r77));
74                 svstl_vnum(p8_all, ptrOUT, 6, svreinterpret_u8(r66));
75                 svstl_vnum(p8_all, ptrOUT, 5, svreinterpret_u8(r55));
76                 svstl_vnum(p8_all, ptrOUT, 4, svreinterpret_u8(r44));
77             case 4 :
78                 svstl_vnum(p8_all, ptrOUT, 3, svreinterpret_u8(r33));
79                 svstl_vnum(p8_all, ptrOUT, 2, svreinterpret_u8(r22));
80             case 2 :
81                 svstl_vnum(p8_all, ptrOUT, 1, svreinterpret_u8(r11));
82                 svstl(p8_all, ptrOUT, svreinterpret_u8(r00));
83         }
84
85         ptrIN      += vl;
86         ptrOUT     += 8*vl;
87         nb_st_elems -= 8;
88     }
89 }
90 }

```

The rearrangement of left matrix consists in grouping matrix elements into sets of four consecutive elements and transposing multiple 8-rows blocks of sets. This is implemented with ZIP1 and ZIP2 instructions in 3 steps (lines 44-69).

Since matrix dimensions are multiple of 32 and element size is 8-bit, in the case of vector lengths higher than 256-bit the vector loads of the inner loop's last iteration may have inactive lanes. Therefore, the rearranged matrix elements stores to the memory are managed by the `switch` statement (lines 71-83).

The following is SVE implementation of right matrix rearrangement:

```

1  void rearrangeRight_fixp( uint64_t K, uint64_t N,
2                          uint8_t * inRight, uint8_t * inRight_MOD) {
3      uint64_t      x, y, nb_st_elems;
4      svbool_t      p_ld;
5      svuint8_t     r0, r1, r2, r3;
6      uint64_t      offsetIN_1, offsetIN_2, offsetIN_3;
7      uint64_t      offsetOUT_1, offsetOUT_2, offsetOUT_3;
8
9      uint8_t        *ptrIN;
10     uint8_t         *ptrOUT;
11
12     svbool_t        p8_all = svptrue_b8();
13     uint64_t         vl = svcntb();
14     uint64_t         vl_4 = (vl >> 2);
15
16     offsetIN_1      = N;
17     offsetIN_2      = 2*N;
18     offsetIN_3      = 3*N;
19
20     offsetOUT_1     = K*vl_4;
21     offsetOUT_2     = K*vl_4*2;
22     offsetOUT_3     = K*vl_4*3;
23

```

```

24         nb_st_elems = 4*N/vl;
25
26         for (y=0; svptest_first( p8_all, p_ld = svwhilelt_b8(y, N)); y+=vl) {
27             ptrOUT = &inRight_MOD[y*K];
28
29             for (x=0; x<K; x+=4) {
30                 ptrIN = &inRight[x*N+y];
31
32                 r0 = svld1(p_ld, ptrIN);
33                 r1 = svld1(p_ld, &ptrIN[offsetIN_1]);
34                 r2 = svld1(p_ld, &ptrIN[offsetIN_2]);
35                 r3 = svld1(p_ld, &ptrIN[offsetIN_3]);
36
37                 svuint8_t t4 = svzip1(r0, r2);
38                 svuint8_t t5 = svzip1(r1, r3);
39                 svuint8_t t6 = svzip2(r0, r2);
40                 svuint8_t t7 = svzip2(r1, r3);
41
42                 r0 = svzip1(t4, t5);
43                 r1 = svzip2(t4, t5);
44                 r2 = svzip1(t6, t7);
45                 r3 = svzip2(t6, t7);
46
47                 switch(nb_st_elems) {
48                     default :
49                         svst1(p8_all, &ptrOUT[offsetOUT_3], r3);
50                         svst1(p8_all, &ptrOUT[offsetOUT_2], r2);
51                     case 2 :
52                         svst1(p8_all, &ptrOUT[offsetOUT_1], r1);
53                     case 1 :
54                         svst1(p8_all, ptrOUT, r0);
55                 }
56
57                 ptrOUT += vl;
58             }
59
60             nb_st_elems -= 4;
61         }
62     }

```

The rearrangement of right matrix consists in transposing of multiple $[4, vl/4]$ size subblocks (vl is the number of 8-bit elements in a machine vector length). This is implemented with ZIP1 and ZIP2 instructions in 2 steps (lines 37-45).

Since matrix dimensions are multiple of 32 and element size is 8-bit, in the case of vector lengths higher than 256-bit the vector loads of the inner loop's last iteration may have inactive lanes. Therefore, the rearranged matrix elements stores to the memory are managed by the `switch` statement (lines 47-55).

The following is SVE implementation of matrix multiply results calculation:

```

1         void matmul_dotp_fixp( uint64_t M, uint64_t K, uint64_t N,
2                               uint8_t * inLeft_MOD, uint8_t * inRight_MOD, uint32_t * out) {
3             uint64_t      x, y, z;
4             svuint32_t  acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7;
5             uint64_t      offsetOUT_1, offsetOUT_2, offsetOUT_3, offsetOUT_4;
6             uint64_t      offsetOUT_5, offsetOUT_6, offsetOUT_7;
7
8             uint8_t      *ptrIN_left;
9             uint8_t      *ptrIN_right;
10            uint32_t      *ptrOUT;

```

```

11
12     svbool_t    p8_all = svptrue_b8();
13     uint64_t    vl = svcntb();
14     uint64_t    vl_4 = (vl >> 2);
15
16     offsetOUT_1 = N;
17     offsetOUT_2 = 2*N;
18     offsetOUT_3 = 3*N;
19     offsetOUT_4 = 4*N;
20     offsetOUT_5 = 5*N;
21     offsetOUT_6 = 6*N;
22     offsetOUT_7 = 7*N;
23
24     for (x=0; x<M; x+=8) {
25         ptrOUT      = &out[x*N];
26         ptrIN_right = &inRight_MOD[0];
27
28         for (y=0; y<N; y+=vl_4) {
29             ptrIN_left = &inLeft_MOD[x*K];
30
31             acc0 = svdup_u32(0);
32             acc1 = svdup_u32(0);
33             acc2 = svdup_u32(0);
34             acc3 = svdup_u32(0);
35             acc4 = svdup_u32(0);
36             acc5 = svdup_u32(0);
37             acc6 = svdup_u32(0);
38             acc7 = svdup_u32(0);
39
40             for (z=0; z<K; z+=4) {
41                 svuint8_t a0123 = svldlrq(p8_all, ptrIN_left);
42                 svuint8_t a4567 = svldlrq(p8_all, &ptrIN_left[16]);
43                 svuint8_t b_vec = svldl(p8_all, ptrIN_right);
44
45                 acc0 = svdot_lane(acc0, b_vec, a0123, 0);
46                 acc1 = svdot_lane(acc1, b_vec, a0123, 1);
47                 acc2 = svdot_lane(acc2, b_vec, a0123, 2);
48                 acc3 = svdot_lane(acc3, b_vec, a0123, 3);
49                 acc4 = svdot_lane(acc4, b_vec, a4567, 0);
50                 acc5 = svdot_lane(acc5, b_vec, a4567, 1);
51                 acc6 = svdot_lane(acc6, b_vec, a4567, 2);
52                 acc7 = svdot_lane(acc7, b_vec, a4567, 3);
53
54                 ptrIN_left += 32;
55                 ptrIN_right += vl;
56             }
57
58             svst1(p8_all, ptrOUT, acc0);
59             svst1(p8_all, &ptrOUT[offsetOUT_1], acc1);
60             svst1(p8_all, &ptrOUT[offsetOUT_2], acc2);
61             svst1(p8_all, &ptrOUT[offsetOUT_3], acc3);
62             svst1(p8_all, &ptrOUT[offsetOUT_4], acc4);
63             svst1(p8_all, &ptrOUT[offsetOUT_5], acc5);
64             svst1(p8_all, &ptrOUT[offsetOUT_6], acc6);
65             svst1(p8_all, &ptrOUT[offsetOUT_7], acc7);
66
67             ptrOUT += vl_4;
68         }
69     }
70 }

```

The SVE vectorization is implemented by unrolling:

- The outer loop by factor 8.
- The middle loop by factor $vl/4$ (1/4 of the number of 8-bit elements in a machine vector length).
- The inner loop by factor 4.

The most inner loop at line 40 produces results of a subblock $out[8, vl/4] = inLeft[8, K] * inRight[K, vl/4]$. The computation is implemented using the unsigned integer dot-product by indexed elements `UDOT` instruction at lines 45-52 and described in [A3.5 Dot product](#). The indexed elements are left input matrix elements that are effectively loaded and replicated by instruction `LD1RQB` (16 elements per one vector load) at lines 41-42. Right input matrix elements are loaded by contiguous vector length load `LD1B` at line 43. At completion of the most inner loop a subblock $out[8, vl/4]$ results are stored to the memory by contiguous vector length store `ST1W` (lines 58-65).

Due to the constraint that matrix dimensions are multiple of 32, in the vector length agnostic (up to 1024-bit) implementation of this function all lanes of vector registers are active.

Part C

Computer vision operations examples

Chapter C1

Sobel filter

This chapter provides code examples of Sobel filtering computation.

C1.1 Sobel 3x3 2-dimensional filtering C implementation

Sobel 3x3 2D filtering is implemented in two steps: with 1D horizontal (row) 3-taps filtering followed by 1D vertical (column) 3-taps filtering.

The following C code implements Sobel 3x3 horizontal filtering step with real SP floating-point elements:

```
void sobel_hor_c( float * input, float * out_hor,
                 const float * kx, int64_t height, int64_t width) {
    for (int64_t j=0; j<height; j++) {
        for (int64_t i=0; i<width-2; i++) {
            float res = 0.0;
            for (int64_t k=0; k<3; k++) {
                res += kx[k]*input[j*width + i + k];
            }
            out_hor[j*(width-2) + i] = res;
        }
    }
}
```

The following C code implements Sobel 3x3 vertical filtering step with real SP floating-point elements:

```
void sobel_vert_c( float * out_hor, float * output,
                  const float * ky, int64_t height, int64_t width) {
    for (int64_t j=0; j<height-2; j++) {
        for (int64_t i=0; i<width-2; i++) {
            float res = 0.0;
            for (int64_t k=0; k<3; k++) {
                res += ky[k]*out_hor[(j + k)*(width-2) + i];
            }
            output[j*(width-2) + i] = res;
        }
    }
}
```

C1.2 Code example: Sobel 3x3 2-dimensional filtering with real SP floating-point elements

The following is optimized SVE implementation of Sobel 3x3 horizontal filtering step with real SP floating-point elements.

```

1      void sobel_hor_sve( float * input,    // pointer to image[height, width]
2                          float * out_hor, // pointer to horizontal filtering output
3                          const float * kx, // Sobel horizontal filter coefficients
4                          int64_t height,  // image height
5                          int64_t width) { // image width
6          int64_t j, i;
7          svbool_t p_row;
8          svfloat32_t in1, in2, in3, res;
9          uint64_t vl = svcntw();
10
11      for (j=0; j<height; j++) {
12          float32_t * in_ptr = &input[j*width];
13          float32_t * out_ptr = &out_hor[j*(width-2)];
14
15          for (i=0; i<width-2; i+=vl) {
16              p_row = svwhilelt_b32(i, width-2);
17              in1 = svld1(p_row, &in_ptr[i]);
18              in2 = svld1(p_row, &in_ptr[i + 1]);
19              in3 = svld1(p_row, &in_ptr[i + 2]);
20
21              res = svmul_z(p_row, in1, kx[0]);
22              res = svmla_m(p_row, res, in2, kx[1]);
23              res = svmla_m(p_row, res, in3, kx[2]);
24
25              svst1(p_row, &out_ptr[i], res);
26          }
27      }
28  }
```

The following is optimized SVE implementation of Sobel 3x3 vertical filtering step with real SP floating-point elements.

```

1      void sobel_vert_sve( float * out_hor, // pointer to horizontal filtering output
2                          float * output,  // pointer to Sobel 2D filtering output
3                          const float * ky, // Sobel vertical filter coefficients
4                          int64_t height,  // image height
5                          int64_t width) { // image width
6          int64_t j, i;
7          svbool_t p_row;
8          svfloat32_t in1, in2, in3, res;
9          uint64_t vl = svcntw();
10
11      for (j=0; j<height-2; j++) {
12          float32_t * in_ptr = &out_hor[j*(width-2)];
13          float32_t * out_ptr = &output[j*(width-2)];
14
15          for (i=0; i<width-2; i+=vl) {
16              p_row = svwhilelt_b32(i, width-2);
17              in1 = svld1(p_row, &in_ptr[i]);
18              in2 = svld1(p_row, &in_ptr[i + width-2]);
19              in3 = svld1(p_row, &in_ptr[i + 2*(width-2)]);
20
21              res = svmul_z(p_row, in1, ky[0]);
22              res = svmla_m(p_row, res, in2, ky[1]);
23              res = svmla_m(p_row, res, in3, ky[2]);
24
25              svst1(p_row, &out_ptr[i], res);
26          }
27      }
28  }
```

```
21             res = svmul_z(p_row, in1, ky[0]);
22             res = svmla_m(p_row, res, in2, ky[1]);
23             res = svmla_m(p_row, res, in3, ky[2]);
24
25             svstl(p_row, &out_ptr[i], res);
26         }
27     }
28 }
```

At lines 22-24 of both the horizontal and vertical filtering functions, there are vector by scalar `FMUL` and `FMLA` instructions. In these instructions each active SP element of the first multiplicand (vector) is multiplied by the second (scalar) SP multiplicand.

Chapter C2

Integral image

This chapter provides code examples of Integral image computation.

C2.1 Integral image C implementation

The Integral image algorithm computes the sum of pixel values in an image.

For a $(height-1) \times (width-1)$ size image with real SP floating-point pixels from the *input* buffer, the following C code produces the integral image sums in the *output* buffer, at the indexes $[i, j]$ where $i = 1, \dots, height-1$; $j = 1, \dots, width-1$. A $height \times width$ size *output* buffer is initialized to 0 in advance.

```
void integral_image_c( float input[height-1][width-1],
                      float output[height][width],
                      int64_t height, int64_t width) {
    float v;
    for (int64_t x=1; x<height; x++) {
        for (int64_t y=1; y<width; y++) {
            v = input[x-1][y-1];
            v += output[x][y-1];
            v += output[x-1][y];
            v -= output[x-1][y-1];
            output[x][y] = v;
        }
    }
}
```

C2.2 Code example: Integral image with real SP floating-point elements

The following is vector length agnostic SVE implementation of Integral image with real SP floating-point pixels computation. The image is taken from a $(height-1) \times width$ size *input* buffer. The integral image sums are computed into the *output* buffer at the indexes $[i, j]$ where $i = 1, \dots, height-1$; $j = 0, \dots, width-1$. A $height \times width$ size *output* buffer is initialized to 0 in advance.

```

1      void integral_image_vla( float input[height-1][width],
2                             // pointer to image buffer[height-1, width]
3                             float output[height][width],
4                             // pointer to integral image buffer[height, width]
5                             int64_t height, // image height
6                             int64_t width) { // image width
7          svbool_t    p_gov, p_gov_update, p_single, p_x_0;
8          int64_t      x, y;
9          float32_t * in_row;
10         float32_t * out_row;
11         svfloat32_t acc, carry_sum, acc_row_up, vec_zero;
12
13         vec_zero      = svdup_f32(0.0);
14         p_x_0         = svpfalse();
15         uint64_t vl = svcntw();
16
17         for (x=1; x<height; x++) {
18             carry_sum = vec_zero;
19             out_row    = &output[x][0];
20             in_row     = &input[x-1][0];
21
22             for (y=0; y<width; y+=vl) {
23                 p_gov      = svwhilelt_b32(y, width);
24                 p_gov_update = svmov_z(p_x_0, p_gov);
25                 p_single    = svpfalse();
26                 acc         = svld1(p_gov, &in_row[y]);
27                 p_single    = svpnxt_b32(p_gov, p_single);
28
29                 do {
30                     acc      = svadd_m(p_single, acc, carry_sum);
31                     carry_sum = svclastb(p_single, vec_zero, acc);
32                     p_single = svpnxt_b32(p_gov, p_single);
33                 } while (svptest_any(p_gov, p_single));
34
35                 acc_row_up = svld1(p_gov_update, &output[x-1][y]);
36                 acc        = svadd_m(p_gov_update, acc, acc_row_up);
37
38                 svst1(p_gov, &out_row[y], acc);
39             }
40             p_x_0 = svptrue_b32();
41         }
42     }

```

Although this implementation is vector length agnostic, due to the scalar loop at line 28, the vector length bandwidth will not bring the performance gain. The `do` loop at line 28 implements partial horizontal sum computation for one output at a time. This is managed by the instruction `PNEXT` at line 31 that determines the next active 32-bit element lane in `p_gov` and sets the corresponding predicate element to active in the output predicate `p_single`. The next pixel to process is therefore extracted with the instruction `CLASTB` at line 30 based on the state of `p_single` predicate.

If high-performance is of greater interest than a vector length agnostic implementation, one can choose to implement

the Integral image algorithm in a vector length specific manner. The following is one implementation of that kind, where the Integral image algorithm is implemented in 2 steps. In the first step, only the vertical sums are calculated, and the following is its vector length agnostic implementation:

```

1      void integral_image_vert_vla( float input[height][width], // image
2                                  int64_t height,             // image height
3                                  int64_t width) {             // image width
4          int64_t      x, y;
5          svbool_t      p_ld_st;
6          svfloat32_t acc, acc_up;
7
8          uint64_t      vl = svcntw();
9
10         for (x=1; x<height; x++) {
11             float32_t * row      = &input[x][0];
12             float32_t * row_up   = &input[x-1][0];
13
14             for (y=0; y<width; y+=vl) {
15                 p_ld_st = svwhilelt_b32(y, width);
16                 acc      = svld1(p_ld_st, &row[y]);
17                 acc_up   = svld1(p_ld_st, &row_up[y]);
18                 acc      = svadd_m(p_ld_st, acc, acc_up);
19                 svst1(p_ld_st, &row[y], acc);
20             }
21         }
22     }

```

In the second step, the final integral image sums are computed by calculating the horizontal sums of the vertical sums computed in the first step. The following is its 128-bit vector length specific implementation with the assumption that the image height is multiple of 2.

```

1      void integral_image_hor_vls128( float input[height][width], // image
2                                     float output[height][width], // integral image
3                                     int64_t height, // image height (multiple of 2)
4                                     int64_t width) { // image width
5          uint64_t      x, y;
6          uint64_t      vl = svcntw();
7          svint32_t      vec_offsets;
8          svfloat32_t in0, in1, in2, in3;
9          svfloat32_t tmp4, tmp5, tmp6, tmp7;
10         svfloat32_t acc_sums;
11         svbool_t      p_ld_st, p_ld_st_34, p_gather;
12
13         svbool_t      p_true  = svptrtrue_b32();
14         svbool_t      p_row34 = p_true;
15         svbool_t      p_vert  = p_true;
16
17         int32_t      value_offset = 16;
18         int32_t      ptr_offset = (int32_t)(width << 2);
19
20         svint32_t vec_offsets_init = svmul_z(p_true, svindex_s32(0,1), ptr_offset);
21         vec_offsets_init = svsub_x(p_true, vec_offsets_init, 4);
22
23         for (x=0; x<height; x+=4) {
24             if (height<(x+3)) {
25                 p_row34 = svpfalse();
26                 p_vert  = svptrtrue_pat_b32(SV_VL2);
27             }
28
29             float32_t * ptrOut_0 = &output[x][0];

```

```

30         float32_t * ptrOut_1 = &output[x+1][0];
31         float32_t * ptrOut_2 = &output[x+2][0];
32         float32_t * ptrOut_3 = &output[x+3][0];
33
34         vec_offsets = vec_offsets_init;
35
36         for (y=0; y<width; y+=vl) {
37             p_ld_st      = svwhilelt_b32(y, width);
38             p_ld_st_34    = svmov_z(p_row34, p_ld_st);
39
40             in0          = svld1(p_ld_st,      &input[x][y]);
41             in1          = svld1(p_ld_st,      &input[x+1][y]);
42             in2          = svld1(p_ld_st_34,    &input[x+2][y]);
43             in3          = svld1(p_ld_st_34,    &input[x+3][y]);
44
45             tmp4         = svtrn1(in0, in1);
46             tmp5         = svtrn2(in0, in1);
47             tmp6         = svtrn1(in2, in3);
48             tmp7         = svtrn2(in2, in3);
49
50             in0          = svreinterpret_f32(svtrn1(svreinterpret_f64(tmp4),
51                                                     svreinterpret_f64(tmp6)));
52             in1          = svreinterpret_f32(svtrn1(svreinterpret_f64(tmp5),
53                                                     svreinterpret_f64(tmp7)));
54             in2          = svreinterpret_f32(svtrn2(svreinterpret_f64(tmp4),
55                                                     svreinterpret_f64(tmp6)));
56             in3          = svreinterpret_f32(svtrn2(svreinterpret_f64(tmp5),
57                                                     svreinterpret_f64(tmp7)));
58
59             p_gather     = svwhilelt_b32((int64_t) 0, y);
60             p_gather     = svmov_z(p_vert, p_gather);
61             acc_sums     = svld1_gather_offset(p_gather, ptrOut_0, vec_offsets);
62             vec_offsets  = svadd_x(p_true, vec_offsets, value_offset);
63
64             in0          = svadd_x(p_true, in0, acc_sums);
65             in1          = svadd_x(p_true, in1, in0);
66             in2          = svadd_x(p_true, in2, in1);
67             in3          = svadd_x(p_true, in3, in2);
68
69             tmp4         = svtrn1(in0, in1);
70             tmp5         = svtrn2(in0, in1);
71             tmp6         = svtrn1(in2, in3);
72             tmp7         = svtrn2(in2, in3);
73
74             in0          = svreinterpret_f32(svtrn1(svreinterpret_f64(tmp4),
75                                                     svreinterpret_f64(tmp6)));
76             in1          = svreinterpret_f32(svtrn1(svreinterpret_f64(tmp5),
77                                                     svreinterpret_f64(tmp7)));
78             in2          = svreinterpret_f32(svtrn2(svreinterpret_f64(tmp4),
79                                                     svreinterpret_f64(tmp6)));
80             in3          = svreinterpret_f32(svtrn2(svreinterpret_f64(tmp5),
81                                                     svreinterpret_f64(tmp7)));
82
83             svst1(p_ld_st,      &ptrOut_0[y], in0);
84             svst1(p_ld_st,      &ptrOut_1[y], in1);
85             svst1(p_ld_st_34,    &ptrOut_2[y], in2);
86             svst1(p_ld_st_34,    &ptrOut_3[y], in3);
87         }
88     }
89 }

```

The horizontal sums are computed such that the block of $(4 \times vl)$ vertical sums is first transposed (lines 45-57), then the additions are performed vertically (lines 64-67) in each 32-bit processing lane, and then the processed block is transposed back (lines 69-81) before it is stored to the output buffer (lines 83-86).

The gather load instruction (line 61) loads previously computed integral sums of the $(y-1)th$ column (corresponding to the 4 rows being processed in the current loop iteration) to accumulate it with the vertical sums currently being processed.

Chapter C3

Histogram

This chapter provides code examples of Histogram computation.

C3.1 Histogram C implementation

The following is C code of the histogram computation for an image with `nb_records` 8-bit pixels in the `records` buffer:

```
void calc_histogram( unsigned int * histogram,
                    uint8_t * records,
                    unsigned int nb_records) {
    for (unsigned int i = 0; i < nb_records; i++) {
        histogram[records[i]] += 1;
    }
}
```

C3.2 Code example: 8-bit pixels Image Histogram computation

The following is vector length agnostic SVE2 implementation of Histogram computation for the image with 8-bit pixels:

```

1      MOV      w2, w2
2      MOV      x4, #0
3
4      WHILELO p1.s, x4, x2
5      B.NFRST .L_return
6
7      .L_loop:
8          LD1B    z1.s, p1/Z, [x1, x4]
9          LD1W    z2.s, p1/Z, [x0, z1.s, UXTW #2]
10         HISTCNT z0.s, p1/Z, z1.s, z1.s
11         ADD     z2.s, p1/M, z2.s, z0.s
12         ST1W    z2.s, p1, [x0, z1.s, UXTW #2]
13         INCW     x4
14         WHILELO p1.s, x4, x2
15         B.FIRST .L_loop
16
17     .L_return:
18         RET

```

The gather load instruction (line 9) fetches from the memory the values of histogram entries corresponding to the image pixels processed in the current loop iteration. At line 10 the instruction `HISTCNT` provides the repetition information of currently processed image pixels, and according to this the relevant histogram values are updated in the memory by the scatter store instruction (line 12).

The gather load (line 9) and scatter store (line 12) instructions are significantly more expensive in terms of the execution time than contiguous vector load and store instructions. Therefore, these instructions considerably slow down the loop (lines 7-15) execution.

If high-performance is of greater interest than a vector length agnostic implementation, one can choose to implement 8-bit pixels histogram computation in vector length specific manner, avoiding the inefficient gather load and scatter store instructions. The latter is one implementation of that kind, where temporary counters of all histogram entries are kept in registers. For 8-bit image pixels there are 256 histogram entries, and therefore 256 temporary counters to maintain. The temporary counters are monitored. When at least one of the counters reaches its limit, the histogram values are updated by the current temporary counters and dumped back to the memory. The temporary counters are then reset to zero. The process is repeated until the whole image is screened.

The following is 512-bit vector length specific SVE2 implementation with limitation that `nb_records` must be multiple of 64. If image size is not multiple of 64, the rest of pixels is processed in the C scalar loop.

```

1      MOV      w2, w2
2      PTRUE    p0.b
3      DUP      z29.b, #191                // histogram overflow guard: 255-4*16
4
5      CBZ      w2, .L_return
6
7      // Initialize histogram entries 0..255
8      INDEX    z10.b, #0, #1              // z10: 0..63
9      MOVPRFX  z11, z10
10     ADD      z11.b, z11.b, #64
11     MOVPRFX  z12, z10
12     ADD      z12.b, z12.b, #128
13     MOVPRFX  z13, z10
14     ADD      z13.b, z13.b, #192          // z13: 192..255
15

```

```

16      ADD      x6, x1, x2
17
18      .L_loop_outer:
19          MOV      z20.b, #0
20          MOV      z21.b, #0
21          MOV      z22.b, #0
22          MOV      z23.b, #0
23
24      .L_loop_inner:
25          LD1RQB   z0.b, p0/Z, [x1, #0]
26          LD1RQB   z1.b, p0/Z, [x1, #16]
27          LD1RQB   z2.b, p0/Z, [x1, #32]
28          LD1RQB   z3.b, p0/Z, [x1, #48]
29
30          HISTSEG   z14.b, z10.b, z0.b
31          HISTSEG   z15.b, z11.b, z0.b
32          HISTSEG   z16.b, z12.b, z0.b
33          HISTSEG   z17.b, z13.b, z0.b
34
35          HISTSEG   z18.b, z10.b, z1.b
36          HISTSEG   z19.b, z11.b, z1.b
37          HISTSEG   z4.b,  z12.b, z1.b
38          HISTSEG   z5.b,  z13.b, z1.b
39
40          ADD      x1, x1, #64
41          HISTSEG   z6.b,  z10.b, z2.b
42          HISTSEG   z7.b,  z11.b, z2.b
43          HISTSEG   z8.b,  z12.b, z2.b
44          HISTSEG   z9.b,  z13.b, z2.b
45
46          HISTSEG   z24.b, z10.b, z3.b
47          HISTSEG   z25.b, z11.b, z3.b
48          HISTSEG   z26.b, z12.b, z3.b
49          HISTSEG   z27.b, z13.b, z3.b
50
51          ADD      z20.b, z20.b, z14.b
52          ADD      z21.b, z21.b, z15.b
53          ADD      z22.b, z22.b, z16.b
54          ADD      z23.b, z23.b, z17.b
55
56          CMP      x1, x6
57          ADD      z20.b, z20.b, z18.b
58          ADD      z21.b, z21.b, z19.b
59          ADD      z22.b, z22.b, z4.b
60          ADD      z23.b, z23.b, z5.b
61
62          ADD      z20.b, z20.b, z6.b
63          ADD      z21.b, z21.b, z7.b
64          ADD      z22.b, z22.b, z8.b
65          ADD      z23.b, z23.b, z9.b
66
67          ADD      z20.b, z20.b, z24.b
68          ADD      z21.b, z21.b, z25.b
69          ADD      z22.b, z22.b, z26.b
70          ADD      z23.b, z23.b, z27.b
71
72          B.HS      .L_loop_inner_end
73
74          CMPHS     p1.b, p0/Z, z29.b, z20.b
75          CMPHS     p2.b, p0/Z, z29.b, z21.b

```

```

76      CMPHS    p3.b, p0/Z, z29.b, z22.b
77      CMPHS    p4.b, p0/Z, z29.b, z23.b
78      AND      p1.b, p1/Z, p2.b, p3.b      // p1 = p1 & p2 & p3
79      NANDS    p4.b, p0/Z, p1.b, p4.b      // p4 = ~(p0 & p1 & p4)
80      B.NONE    .L_loop_inner
81
82      .L_loop_inner_end:
83      LD4W      {z2.s, z3.s, z4.s, z5.s}, p0/Z, [x0, #0, MUL VL]
84      LD4W      {z6.s, z7.s, z8.s, z9.s}, p0/Z, [x0, #4, MUL VL]
85      UXTB      z14.h, p0/M, z20.h
86      LSR      z15.h, z20.h, #8
87      UXTB      z16.h, p0/M, z21.h
88      LSR      z17.h, z21.h, #8
89      UADDWB     z2.s, z2.s, z14.h
90      UADDWT     z4.s, z4.s, z14.h
91      UADDWB     z3.s, z3.s, z15.h
92      UADDWT     z5.s, z5.s, z15.h
93      UADDWB     z6.s, z6.s, z16.h
94      UADDWT     z8.s, z8.s, z16.h
95      UADDWB     z7.s, z7.s, z17.h
96      UADDWT     z9.s, z9.s, z17.h
97      ST4W      {z2.s, z3.s, z4.s, z5.s}, p0, [x0, #0, MUL VL]
98      LD4W      {z2.s, z3.s, z4.s, z5.s}, p0/Z, [x0, #8, MUL VL]
99      UXTB      z14.h, p0/M, z22.h
100     LSR      z15.h, z22.h, #8
101     UXTB      z16.h, p0/M, z23.h
102     LSR      z17.h, z23.h, #8
103     ST4W      {z6.s, z7.s, z8.s, z9.s}, p0, [x0, #4, MUL VL]
104     LD4W      {z6.s, z7.s, z8.s, z9.s}, p0/Z, [x0, #12, MUL VL]
105     UADDWB     z2.s, z2.s, z14.h
106     UADDWT     z4.s, z4.s, z14.h
107     UADDWB     z3.s, z3.s, z15.h
108     UADDWT     z5.s, z5.s, z15.h
109     UADDWB     z6.s, z6.s, z16.h
110     UADDWT     z8.s, z8.s, z16.h
111     UADDWB     z7.s, z7.s, z17.h
112     UADDWT     z9.s, z9.s, z17.h
113     ST4W      {z2.s, z3.s, z4.s, z5.s}, p0, [x0, #8, MUL VL]
114     ST4W      {z6.s, z7.s, z8.s, z9.s}, p0, [x0, #12, MUL VL]
115
116     CMP      x1, x6
117     B.LO      .L_loop_outer
118
119     .L_return:
120     RET

```

Each 512-bit vector register contains 64 histogram entries, hence four vector registers are initialized at lines 8-14 with 256 histogram entries. Similarly, four vector registers will contain 256 8-bit temporary counters, and these are initialized at lines 19-22.

In each iteration of the inner loop four groups of 16 image pixels, hence 64 pixels, are processed. Each group of 16 pixels is loaded from the memory and replicated over the vector length using the instruction `LD1RQB` (lines 25-28). Then, each group of 16 pixels is compared versus all 256 histogram entries by `HISTSEG` instructions. To produce the count of matches versus all 256 histogram entries four `HISTSEG` instructions are needed. For example, for the first group of 16 pixels these are `HISTSEG` instructions at lines 30-33. This is followed by the update of 256 temporary counters at lines 51-70. Finally, the current state of all 256 temporary counters is checked versus the threshold (lines 74-77), and if at least one of 256 counters reached the threshold the inner loop exits, otherwise the next 64 pixels are processed by the inner loop.

In the outer loop all 256 32-bit histogram values are loaded from the memory, updated by the 256 8-bit temporary counters, and stored back to the memory. The histogram values are loaded from the memory by four word structure contiguous load instruction `LD4W` (lines 83, 84, 98, 104). The 8-bit temporary counters are padded with zeros and expanded to 16-bit by instructions `UXTB` and `LSR` (lines 85-88, 99-102). The histogram values are updated by expanded temporary counters using the unsigned add widening instructions `UADDWB` and `UADDWT` (lines 89-96, 105-112) described in [A5.1.2 Top-Bottom widening instructions](#). Finally, the updated histogram values are stored back to memory by four structure contiguous store instruction `ST4W` (lines 97, 103, 113, 114).

Part D

String processing examples

Chapter D1

Skip white space

This chapter provides code example of string processing that would skip over white spaces.

D1.1 Code example: skip over white spaces

The C code for string processing function that would update the input string pointer such that it skips over the white spaces, is shown below:

```
const char* SkipWhiteSpace(const char* p, const char* end) {
    while (p != end && (*p == ' ' || *p == '\n' || *p == '\r' || *p == '\t')) {
        p++;
    }
    return p;
}
```

The optimized SVE2 implementation is taking advantage of the new string processing instruction *nmatch*.

```
1      MOV      w3, #0xD090000
2      ADD      w3, w3, #0xA20
3      MOV      z1.s, w3      // 0xD090A20, ascii for '\r\t\n '
4      WHILELT  p0.b, x0, x1
5
6      .L_start:
7          LD1B   z0.b, p0/z, [x0]
8          NMATCH p1.b, p0/z, z0.b, z1.b
9          B.ANY  .L_spacesEnd
10         INCB   x0
11         WHILELT p0.b, x0, x1
12         B.FIRST .L_start
13
14         MOV    x0, x1
15         B      .L_return
16
17         .L_spacesEnd:
18             BRKB   p2.b, p0/z, p1.b
19             INCP   x0, p2.b
20
21         .L_return:
22             MOV    x2, x0
23             RET
```

The goal is to recognize consecutive white space characters: single space (0x20), new line (0x0a), carriage return (0x0d) or horizontal tab (0x09), and advance the current string pointer such that these consecutive white spaces are skipped.

The group of white space characters is replicated into the vector register *z1* (line 1). With instruction *NMATCH* at line 6 the group of white space characters from vector register *z1* is compared with the input string loaded into vector register *z0*. Each vector register *z0* 8-bit element is compared with each 8-bit element of the corresponding *z1* vector register 128-bit granule. If there is a match, meaning the white space is found, the respective predicate bit of predicate register *p1* is set to false. Otherwise, when a non-white space character is detected, the respective predicate bit of predicate register *p1* is set to true.

The loop starting at line 4 can exit in two ways:

1. If the first non-white space character is found. This condition is checked by instruction *B.ANY* at line 7. If the first non-white space character is found, the string pointer is updated at lines 16-17. The instruction *BRKB* at line 16, sets to true predicate register *p2* bits that correspond to white space characters detected by the previous *NMATCH* instruction before the first non white space character occurrence. And with instruction *INCP* at line 17, the current string pointer is advanced for the number of these last white space characters found.
2. If the input string end is reached. This condition is checked by instruction *B.FIRST* at line 10. If the input string end is reached, at line 12 the string pointer is updated to the end of the string.

Chapter D2

Skip word

This chapter provides code example of string processing that would skip over a word.

D2.1 Code example: skip over a word

The C code for string processing function that would update the input string pointer such that it skips over a word, is shown below:

```
const char* SkipWord(const char* p, const char* end) {
    while (p != end && *p != ' ' && *p != '\n' && *p != '\r' && *p != '\t') {
        p++;
    }
    return p;
}
```

The optimized SVE2 implementation is taking advantage of the new string processing instruction *match*.

```
1      MOV      w3, #0xD090000
2      ADD      w3, w3, #0xA20
3      MOV      z1.s, w3      // 0xD090A20, ascii for '\r\t\n '
4      WHILELT  p0.b, x0, x1
5
6      .L_start:
7          LD1B   z0.b, p0/z, [x0]
8          MATCH  p1.b, p0/z, z0.b, z1.b
9          B.ANY  .L_wordEnd
10         INCB   x0
11         WHILELT p0.b, x0, x1
12         B.FIRST .L_start
13
14         MOV    x0, x1
15         B      .L_return
16
17         .L_wordEnd:
18             BRKB  p2.b, p0/z, p1.b
19             INCP  x0, p2.b
20
21         .L_return:
22             MOV   x2, x0
23             RET
```

The goal is to recognize a word, that is a group of consecutive non white space characters, and advance the current string pointer such that a word is skipped.

The group of white space characters: 0x20 for single space, 0x0a for new line, 0x0d for carriage return and 0x09 for horizontal tab is replicated into the vector register *z1* (line 1). With instruction *MATCH* at line 6 the group of white space characters from vector register *z1* is compared with the input string loaded into vector register *z0*. Each vector register *z0* 8-bit element is compared with each 8-bit element of the corresponding *z1* vector register 128-bit granule. If there is a match, meaning the white space is found, the respective predicate bit of predicate register *p1* is set to true. Otherwise, when a non white space character is detected, the respective predicate bit of predicate register *p1* is set to false.

The loop starting at line 4 can exit in two ways:

1. If the first white space character is found. This condition is checked by instruction *B.ANY* at line 7. If the first white space character is found, the string pointer is updated at lines 16-17. The instruction *BRKB* at line 16, sets to true predicate register *p2* bits that correspond to non white space characters detected by the previous *MATCH* instruction before the first white space character occurrence. And with instruction *INCP* at line 17, the current string pointer is advanced for the number of these last non white space characters found.
2. If the input string end is reached. This condition is checked by instruction *B.FIRST* at line 10. If the input string end is reached, at line 12 the string pointer is updated to the end of the string.

Chapter D3

String compare

This chapter provides code example of the two strings comparison.

D3.1 Code example: string compare

The C code for string compare function that returns the offset of the first different character between two strings is shown below:

```
int strcmp(const char* str1, const char* str2) {
    char c1, c2;

    do {
        c1 = *str1++;
        c2 = *str2++;
    } while (c1 != '\0' && c1 == c2)

    return c1 - c2;
}
```

The optimized SVE implementation is taking advantage of data load with first-faulting behavior instruction described in [A3.3.6 First faulting loads](#) and the instruction BRKB that enables a loop break to accomplish the vectorization of data-dependent do-while loop.

```
1      PTRUE    p5.b
2      SETFFR
3      MOV     x5, #0
4
5      .L_loop:
6          LDFF1B z0.b, p5/Z, [x0, x5]
7          LDFF1B z1.b, p5/Z, [x1, x5]
8          RDDFRS p7.b, p5/Z
9          B.NLAST .L_fault
10
11         INCB   x5
12         CMPEQ  p0.b, p5/Z, z0.b, #0
13         CMPNE  p1.b, p5/Z, z0.b, z1.b
14      .L_test:
15         ORRS   p4.b, p5/Z, p0.b, p1.b
16         B.NONE .L_loop
17
18      .L_return:
19         BRKB   p4.b, p5/Z, p4.b
20         LASTA  w0, p4, z0.b
21         LASTA  w1, p4, z1.b
22         SUB    w0, w0, w1
23         MOV    x2, x0
24         RET
25
26      .L_fault:
27         INCP   x5, p7.b
28         SETFFR
29         CMPEQ  p0.b, p7/Z, z0.b, #0
30         CMPNE  p1.b, p7/Z, z0.b, z1.b
31         B      .L_test
```

The SETFFR instruction (line 2) initializes the FFR register by setting its all lanes active. The vector registers z0 and z1 are populated by the two strings characters using the load with first-faulting behavior instructions LDFF1B at lines 6-7. Executing these load instructions also updates the FFR register. If the first active lane raised a fault, a data abort exception is taken. Otherwise, if one or more subsequent active lanes raised a fault, the faults are suppressed, and those lanes are marked as inactive in the FFR register. The state of the FFR register is read by

`RDFFRS` instruction at line 8 into the predicate register `p7`. The `p7` predicate register content is tested by `B.NLAST` instruction at line 9:

- If there were no suppressed faults, all lanes of the FFR register remained active, and the branch at line 9 is not taken. The required comparisons of the two strings vector length of data are performed at lines 12-13, and the loop break condition is checked by `B.NONE` instruction at line 16. When the loop brake is detected, the exact lane that caused the loop brake is determined by `BRKB` instruction at line 19. The characters corresponding to this lane are extracted by `LASTA` instructions (lines 20-21), and the difference of the extracted characters is returned (lines 22-23).
- If a suppressed fault occurred, all upper lanes of the FFR register starting from the faulting lane have been cleared, and hence the branch at line 9 is taken. The address offset register `x5` is advanced with instruction `INCP` (line 26) for the number of successfully loaded elements, hence the fault will be in the first active lane for the next iteration. The FFR register is re-initialized at line 27. The required comparisons of the two strings vector length of data are performed at lines 28-29, predicated by `p7`, meaning that active are only lanes up to the one that caused the suppressed fault. It is proceeded with the loop break condition testing (lines 14-16).

Glossary